NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

INÊS PINA FERNANDES

Bachelor in Science

AUTOMATED REFACTORING IN SOFTWARE AUTOMATION PLATFORMS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

AUTOMATED REFACTORING IN SOFTWARE AUTOMATION PLATFORMS

INÊS PINA FERNANDES

Bachelor in Science

Adviser: João Costa Seco
Associate Professor, NOVA University Lisbon

Co-adviser: Miguel Terra-Neves
PhD, Senior Research Scientist, Outsystems

Examination Committee:

Chair: João Baptista da Silva Araújo Júnior
Associate Professor, NOVA University Lisbon

Rapporteur: Jácome Miguel Costa da Cunha
Associate Professor, University of Porto

Adviser: João Costa Seco
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
November, 2021

Automated Refactoring in Software Automation Platforms

Acknowledgements

Throughout the writing of this dissertation, I have received a great deal of support and assistance. My sincerest thanks to my thesis advisor, Associate Professor João Costa Seco, for the valuable advice during this process.

I would also like to extend my deepest gratitude to my OutSystems advisor PhD Miguel Terra-Neves for his guidance throughout this thesis. In addition, I would like to thank NOVA Laboratory for Computer Science and Informatics (NOVA LINCS) and OutSystems for allowing me to work on this project.

I am thankful for my family and friends' patience, guidance, and support through my studies. Finally, a special thanks to my mother and my sister for their encouragement.

Abstract

Software Automation Platforms (SAPs) enable faster development and reduce the need to use code to construct applications. SAPs provide abstraction and automation, resulting in a low-entry barrier for users with less programming skills to become proficient developers. An unfortunate consequence of using SAPs is the production of code with a higher technical debt since such developers are less familiar with the software development best practices. Hence, SAPs should aim to produce a simpler software construction and evolution pipeline beyond providing a rapid software development environment.

One simple example of such high technical debt is the Unlimited Records anti-pattern, which occurs whenever queries are unbounded, i.e. the maximum number of records to be fetched is not explicitly limited. Limiting the number of records retrieved may, in many cases, improve the performance of applications by reducing screen-loading time, thus making applications faster and more responsive, which is a top priority for developers. A second example is the Duplicated Code anti-pattern that severely affects code readability and maintainability, and can even be the cause of bug propagation. To overcome this problem we will resort to automated refactoring as it accelerates the refactoring process and provides provably correct modifications.

This dissertation aims to study and develop a solution for automated refactorings in the context of OutSystems (an industry-leading SAP). This was carried out by implementing automated techniques for automatically refactoring a set of selected anti-patterns in OutSystems logic that are currently detected by the OutSystems technical debt monitoring tool.

Keywords: Automated Refactoring, Anti-Pattern, Duplicated Code, Technical Debt, OutSystems

Resumo

As Plataformas de Automação de *Software* (PAS) habilitam os seus utilizadores a desenvolver aplicações de forma mais rápida e reduzem a necessidade de escrever código. Estas fornecem abstração e automação, o que auxilia utilizadores com menos formação técnica a tornarem-se programadores proficientes. No entanto, a integração de programadores com menos formação técnica também contribui para a produção de código com alta dívida técnica, uma vez que os mesmos estão menos familiarizados com as melhores práticas de desenvolvimento de *software*. Desta forma, as PAS devem ter como objetivo a construção e evolução de *software* de forma simples para além de fornecer um ambiente de desenvolvimento de *software* rápido.

Um exemplo de alta dívida técnica é o anti-padrão *Unlimited Records*, que ocorre sempre que o número máximo de registos a ser retornado por uma consulta à base de dados não é explicitamente limitado. Limitar o número de registos devolvidos pode, em muitos casos, melhorar o desempenho das aplicações, reduzindo o tempo que demora a carregar o ecrã, tornando assim as aplicações mais rápidas e responsivas, sendo esta uma das principais prioridades dos programadores. Um segundo exemplo é o anti-padrão Código Duplicado que afeta gravemente a legibilidade e manutenção do código, e que pode causar a propagação de erros. Para superar este problema, recorreremos à reestruturação automatizada, pois acelera o processo de reestruturação através de modificações comprovadamente corretas.

O objetivo desta dissertação é estudar e desenvolver uma solução para reestruturação automatizada no contexto da OutSystems (uma PAS líder neste setor). Tal foi realizado através da implementação de técnicas automatizadas para reestruturar um conjunto de anti-padrões que são atualmente detetados pela ferramenta de monitorização de dívida técnica da OutSystems.

Palavras-chave: Reestruturação Automatizada, Anti-Padrão, Código Duplicado, Dívida Técnica, OutSystems

Contents

List of Figures

List of Tables

List of Listings

Acronyms

DBMS Database-Management System 20

ERD Entity Relationship Diagram 62, 65

IDE Integrated Development Environment 6, 7, 8, 34, 47, 51

MCS Maximum Common Sub-graph 44, 45

SAP Software Automation Platform v, 1, 2, 3, 4, 6, 16, 18, 19, 43, 51, 71, 83, 84, 99, 103, 104

1 Introduction

This dissertation starts by providing context and motivation for Automated Refactoring in Software Automation Platforms. Next, the main objectives of the work are identified along with the contributions to the state of the art. Finally, an overview of the document's global structure is presented.

1.1 Context

Information Technology teams currently need to deliver an increasing number of applications faster, while faced with pressures to respect budget costs and work with limited resources [31]. To address the scarcity of skillful developers, Software Automation Platforms (SAPs) have been pushed to the cutting edge. Low-Code is a software development approach used in some SAPs that enables the visual development of applications, allowing faster delivery and with minimal coding [32]. It provides abstraction and manual automation on every step of an application's lifecycle, resulting in a low-entry barrier that enables users with a less technical background to become proficient programmers.

As companies aim to expedite time-to-market and empower non-professional developers with the ability to become proficient programmers, a fast development process becomes the priority instead of high-quality code. With the emergence of SAPs, it has never been easier for one to become a productive software developer, even with very little coding knowledge. However, such users are more likely to write code with higher technical debt, since they are less familiar with the best practices of software development. Consequently, managing technical debt quickly becomes a top concern. Technical debt measures the amount of effort required to add new features to a system or to modify the current solution [33].

Refactoring is a foundational technique for improving software design and aiding in code evolution. The problem with finishing software design and architecture before coding is that it relies on having software requirements established in advance and this is generally not feasible [1]. As the software is modified and adapted to new requirements, the quality of the code decreases due to its increasing entropy. Consequently, the major

part of the total software development cost is devoted to software maintenance [2]. Refactoring ensures that it is possible to continue to add new features to the software easily, even when customers' needs change, by incrementally improving the internal software quality. In sum, refactoring improves the quality of code by enabling better internal design, readability, maintainability, and reducing bugs [1].

Anti-patterns are structures in code that suggest the possibility of refactoring and contribute to higher technical debt. These should be rigorously prevented, found, and fixed during the software lifecycle since they often lead to bugs, runtime errors, and software that is hard to maintain [3]. In this dissertation we target two anti-patterns, the first is the Unlimited Records anti-pattern that is predominant in OutSystems and that occurs whenever queries are unbounded, i.e. the maximum number of records to be fetched is not explicitly limited, resulting in the return of all records and contributing to higher technical debt. The process of fetching data from databases in OutSystems is simplified through Aggregates [4], visual elements defined to easily create and maintain queries. When defining an Aggregate, it is possible to limit the maximum number of records to be returned by the Aggregate. If there are limitations to the number of records that are fetched by a query, it is considered to be a good practice to define the maximum number of records to be returned by the query, to optimize the query execution time.

The second anti-pattern we address is the Duplicated Code anti-pattern that is commonplace in large software projects and can have a serious impact on their maintainability, readability and even contribute to bug propagation [5]. The more code there is, the harder it is to understand and modify correctly. Additionally, when adding new features, if there is duplication in the code, it is likely necessary to change the replicas by hand. This process makes software harder to evolve and maintain. By eliminating duplicated code, one can be confident that when changing a piece of code, it is only needed to apply the change once, and if this change fixed a bug, it is no longer present anywhere else. Thus, an important aspect of improving software design is to eliminate duplicated code [1].

1.2 Motivation

The first step before *manual-refactoring* is to ensure that there is a solid set of tests for that section of code, as tests are essential to avoid introducing bugs [1]. Writing a test suite consists of a lot of extra code and takes time, requiring a significant amount of labor. Although this test suite increases confidence, it is not enough to ensure correctness, as one cannot prove that a program has no bugs by testing. This level of confidence is also dependent on the quality of the test suite. Automated refactoring has come to solve this issue and accelerate the refactoring process, by providing provably correct modifications that, being done by machines, are not error-prone like humans.

SAPs must rely on the proficiency of less-skilled developers at fixing technical debt effectively. Furthermore, SAPs are auto-proclaimed automation tools that besides enabling

rapid software development should also contribute to simplifying software evolvement and improvement. To develop applications with quality that meet the expectations of software automation users, the resulting applications must not only be as efficient as possible but should also aim to be easy to maintain. The software automation development process is negatively affected by the increasing technical debt in SAPs. The current solution for solving the anti-patterns causing technical debt at OutSystems requires users to follow a set of guided instructions and instrumentalize the changes themselves. Considering the presence of less-skilled developers using SAPs, this solution carries inherent flaws. Challenges include, but are not limited to, the necessity of hiring skilled developers than can keep the levels of technical debt at a minimum or, in the case that this is not possible, having efficient ways of refactoring these applications to minimize the current technical debt.

The Architecture Dashboard is a static analysis tool for OutSystems code that identifies the anti-patterns that can lead to high technical debt problems. One of the anti-patterns returned by the tool is the Unlimited Records anti-pattern. These findings can easily escalate and become time-consuming to fix, as each application usually has several Aggregates (737,443 Aggregates in total across 45,405 application modules analyzed). Furthermore, there are scenarios where the limit was not defined because the query returns only one record by construction (e.g. get by primary key). These are still flagged by the Architecture Dashboard but could be easily fixed just by setting the value to one. Such scenarios should be solved automatically so that the developer can focus on the less trivial scenarios that require their attention. Therefore, developing techniques for automatically refactoring this anti-pattern is highly desired.

One important example of technical debt is the Duplicated Code Anti-pattern, identified in OutSystems as the most common anti-pattern, reaching as high as 39% in some code-bases [6]. The amount of duplication becomes especially concerning when we take into account that each application developed in OutSystems has several logic flows (724,820 flows and 5,382,011 nodes in total for the modules analyzed), highlighting the importance of reducing code duplication in OutSystems. Additionally, the type of clones detected can correspond to exact clones that are often due to "copy-paste" code which is a poor form of code reuse and regarded as a bad practice among developers.

Considering all the reasons mentioned above, it should be fairly evident that the automation of refactoring techniques in SAPs is a relevant topic to the industry nowadays. Therefore, the motivation behind this dissertation lies in the necessity of an automated approach that while improving software design using refactoring techniques, will also contribute to minimizing the current technical debt problem.

1.3 Objectives

The goal of this dissertation is to study and develop a solution for automated refactorings in the context of SAPs. This was achieved by exploring, designing, and implementing

automated techniques for automatically refactoring a set of selected high-impact anti-patterns that are currently detected by the OutSystems technical debt monitoring tool, Architecture Dashboard.

OutSystems is a state-of-the-art SAP that covers every stage of an application development process and in compliance with best practices of software development. The set of selected patterns includes the Duplicated Code anti-pattern, namely the refactoring of Type I duplicates. We aim that our solution contributes to lowering the high technical debt problem in SAPs, by correcting a set of anti-patterns whose presence increases technical debt. While the effectiveness of the developed techniques was evaluated in the context of the OutSystems ecosystem, their design allows them to be applied in other contexts.

1.4 Contribution

This project's initial contribution is an extended analysis of some of the anti-patterns detected by Architecture Dashboard, and possible automated solutions to solve them. Alongside that analysis, this dissertation provides an overview of the topic's Background and the current Related Work on automated refactoring techniques.

The main contribution of this work is the introduction and evaluation of a novel automated solution for refactoring two anti-patterns in OutSystems logic. The prototype developed in this dissertation aids in solving the current problem of increasing technical debt in SAPs. Being OutSystems a SAP, an automated solution for refactoring anti-patterns adds value to OutSystems' current capabilities and aids the less experienced users in developing projects with low technical debt. The techniques developed for our tool solve subcases of the selected anti-patterns from Architecture Dashboard in an automated way, with the intention that in the future OutSystems users no longer have to solve the technical debt problems by hand. The automated techniques proposed are expected to bring value in the automated-refactoring field, making the work contribution not limited to the OutSystems ecosystem.

The results accomplished during this dissertation motivated the writing of an article, which has been accepted for publication, for the Models and Evolution (ME) Workshop of the ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS), named "Automated Refactoring of Unbounded Queries in Software Automation Platforms" [7]. The publication was developed under the topic of transformation techniques for evolving models and targets the identification of subcases within the Unlimited Records Anti-pattern, including the automated refactoring of the detections. Furthermore, we intend to submit our latest efforts for the theme section on "Models and Evolution" for the Journal of Software and Systems Modeling (SoSyM).

1.5 Document structure

The remainder of this thesis is structured as follows:

- Chapter 2 - The OutSystems Platform provides an overview of the OutSystems' components and tools relevant for this work.
- Chapter 3 - Problem Statement defines and justifies the problem to be tackled by this work.
- Chapter 4 - Background presents the research performed regarding this dissertation's fundamental concepts, including the relational model, methods for estimating the cost of operations, refactoring definitions and methodologies, graph fundamentals, and an overview of Duplicated Code which this document relies on for full comprehension.
- Chapter 5 - Related Work presents the result of an extensive literature review of similar problems that are relevant in the context of this thesis, as they reveal important strategies to be used as a reference for future investigation.
- Chapter 6 - Technical Approach presents an overview of the solution implemented for the identified problem based on an analysis of the anti-patterns that contribute to technical debt and considering the background and related work.
- Chapter 7 - Unlimited Records Anti-Pattern outlines the approach to solve this anti-pattern concerning detection rules and algorithms, refactoring algorithms, and an extensive evaluation of the tool.
- Chapter 8 - Duplicated Code Anti-Pattern presents the approach to solve this anti-pattern concerning the detection of Type I Duplicates and its refactoring.
- Chapter 9 - Conclusion And Future Work covers the concluding reflections and future work.

2

The OutSystems Platform

This thesis is being developed in collaboration with NOVA Laboratory for Computer Science and Informatics (NOVA LINCS) and OutSystems, in particular, the Artificial Intelligence Team. In this chapter, we briefly describe the OutSystems platform's main components as well as the Architecture Dashboard tool for managing technical debt.

2.1 Overview

OutSystems is a pioneer Software Automation Platform (SAP) that empowers its users with the ability to develop enterprise-level Mobile and Web applications faster and easier than with traditional programming languages. OutSystems covers every stage of an application development process, providing a visual development environment and generating code that will be deployed to an enterprise-grade, full-stack system. OutSystems seeks to lower the skillset necessary for developing professional applications by requiring minimal coding and in compliance with best practices of software development.

2.2 Components and Tools

The OutSystems platform components cover all steps of an application lifecycle. To support the creation of applications and their integration, OutSystems offers Integrated Development Environments (IDEs), such as Service Studio and Integration Studio. On the other hand for administrating and managing applications, OutSystems offers **administration and operations tools** like Service Center and Lifetime. An overview of these components can be seen in Figure 2.1, below follows a brief description of each component.

- **Service Studio** is the OutSystems IDE for Web and Mobile Applications. It is a visual development tool for creating, changing, and deploying applications. In a single environment, it is possible to perform different actions, such as design the data model, implement business logic and processes and build User Interfaces.

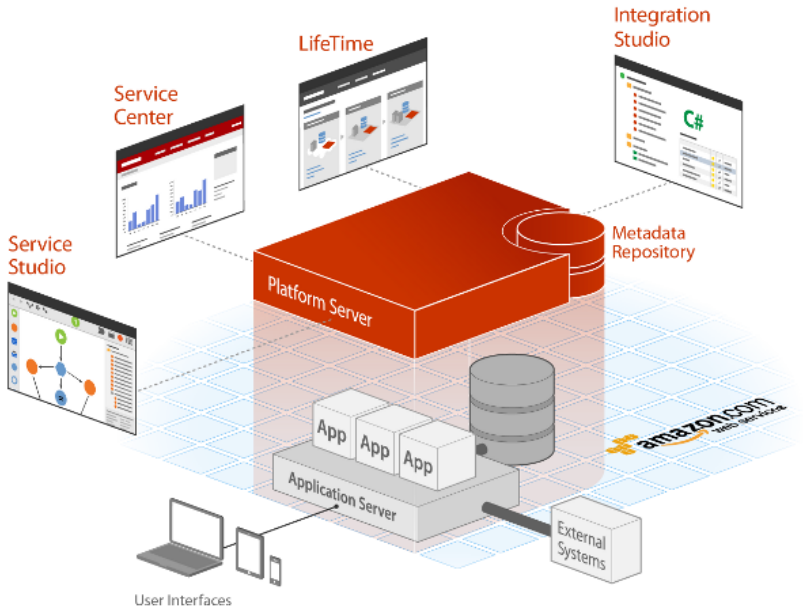


Figure 2.1: OutSystems Components in [34]

- **Integration Studio** is an IDE for implementing connectors to integrate OutSystems applications with other enterprise systems. This tool makes it possible to create extensions to the platform itself by taking external resources (e.g. C code and databases) and creating representations of them inside the OutSystems development environment. These can then be published to the server and used inside Service Studio as regular OutSystems resources.

- **Platform server** is the component where OutSystems applications run and is composed of a set of servers. It orchestrates all compilation, runtime, deployment, and management activities for all applications. A developer can connect to the platform server by pressing the 1-Click Publish button in Service Studio after creating an application. This button publishes the application to the platform server. The platform server then compiles and generates optimized code for the application and deploys it to a standard application server.

- **Service Center** is a platform server management and administration console. This tool makes it possible to see and configure the platform server from an administration and operational standpoint.

- **LifeTime** is a centralized console for managing the infrastructure, environments, applications, IT users, and security. LifeTime allows the users to manage the full application lifecycle across multiple environments.

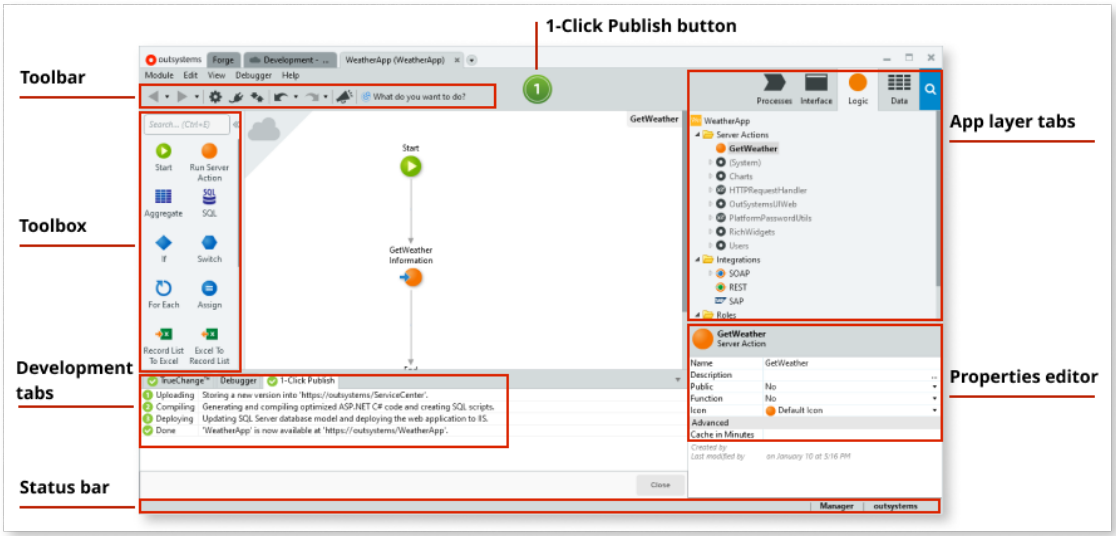


Figure 2.2: Service Studio Interface in [35]

2.3 Visual Language

The Service Studio IDE allows access to a variety of elements from business processes and timers, to user interface elements, business logic flows, and data-related elements. It is in Service Studio that the developer builds the user interface, the business logic, and creates the data model for an application.

Figure 2.2 illustrates the Service Studio interface. At the top right corner of Service Studio, there are four tabs, each tab corresponds to one of the following layers: Processes, Interface, Logic, and Data.

The first layer is the **Processes** layer, which includes business processes and human and automated tasks and their respective decisions and events. This layer has information on the logic and tasks that occur at the highest level. It is also possible to define scheduled actions, such as Timers.

The second layer is the **Interface** layer and focuses on the different components that will make up the user interface. In this layer groups of screens and blocks can be composed using widgets such as images, graphics, and icons.

In the third layer, there is the **Logic** of the application. The logic part of an application is implemented through Actions, such as Client Actions (that run on the client-side) and Server Actions (that run on the server). Roles can also be defined and assigned to users to secure and limit who has access to certain types of logic and components. It is also possible to define actions for data retrieval.

Finally, the fourth layer is the **Data** layer and is where one can define the data model of the application. Inside the data layer, we can model and define different Entities that hold data that can be stored in, and then retrieved from, a database.

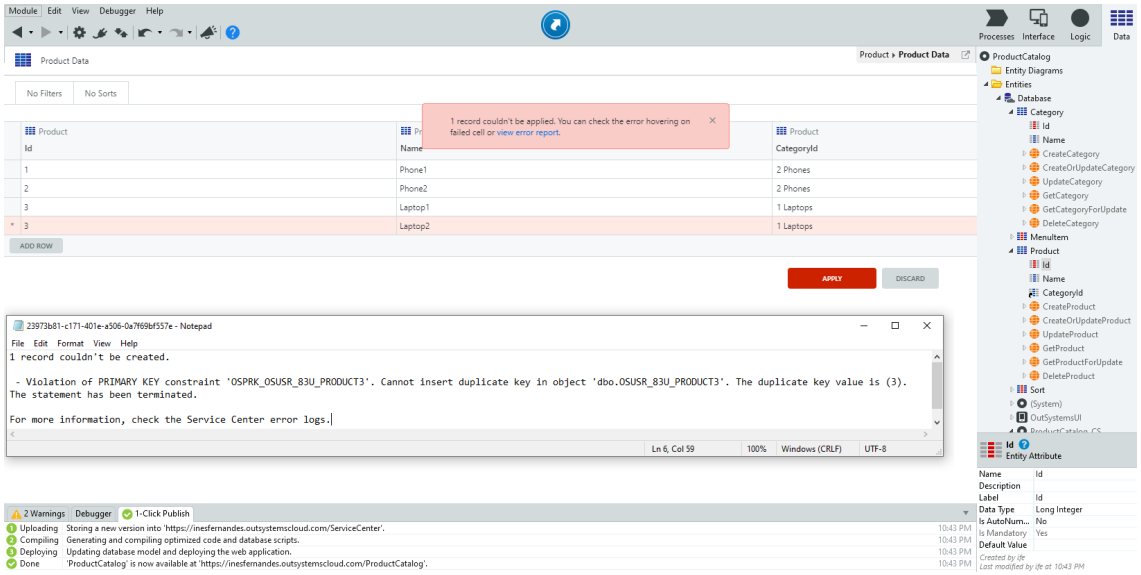


Figure 2.3: Primary Key Constraint Violation in Entity

2.4 Data Modeling

An **Entity** in OutSystems is an element used to persist information in the database and to implement the database model [36]. It is defined through attributes just like a database table. The **Entity Identifier** denominates the Entity's primary key. Once an Entity is created, an attribute named Id is automatically added as the Entity Identifier and by default, its value is a Long Integer and is automatically incremented. The Entity Identifier's attribute can be switched to a different attribute, its data type can also be changed as well as disabling its auto-increment behavior. If the Entity Identifier is automatically incremented, OutSystems signals that it is necessary to add one more attribute to the Entity. In OutSystems, it is not possible to define composite keys, as only one attribute can be the Entity Identifier.

When defining an Entity, it is possible to specify the relationships it has with other entities, through Reference Attributes that correspond to foreign keys [37]. OutSystems automatically creates the necessary database constraints for primary keys and foreign keys. When adding a new row to an Entity, if the uniqueness of the primary key is not respected, OutSystems signals this behavior as a primary key violation, and the row with the duplicated primary key can not be added to the Entity (as seen in Figure 2.3).

2.5 Querying

Most applications need to fetch data from a database, for this purpose OutSystems offers two different ways to fetch data. The first is the usual definition of SQL statements that retrieve information from an attributed database. The second is a visual user interface to perform queries which makes it possible for users with no background in SQL databases

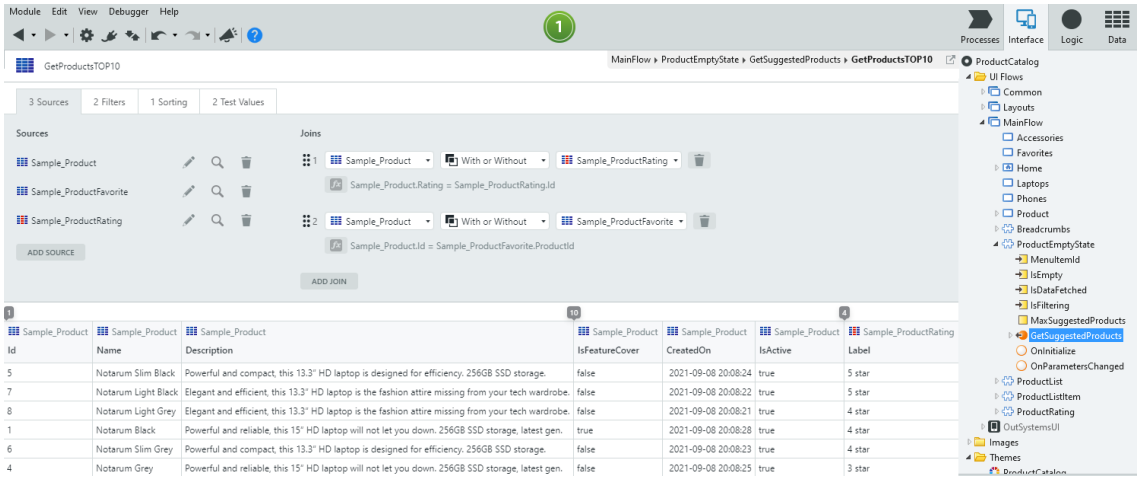


Figure 2.4: Preview of an Aggregate for getting the top 10 rated products

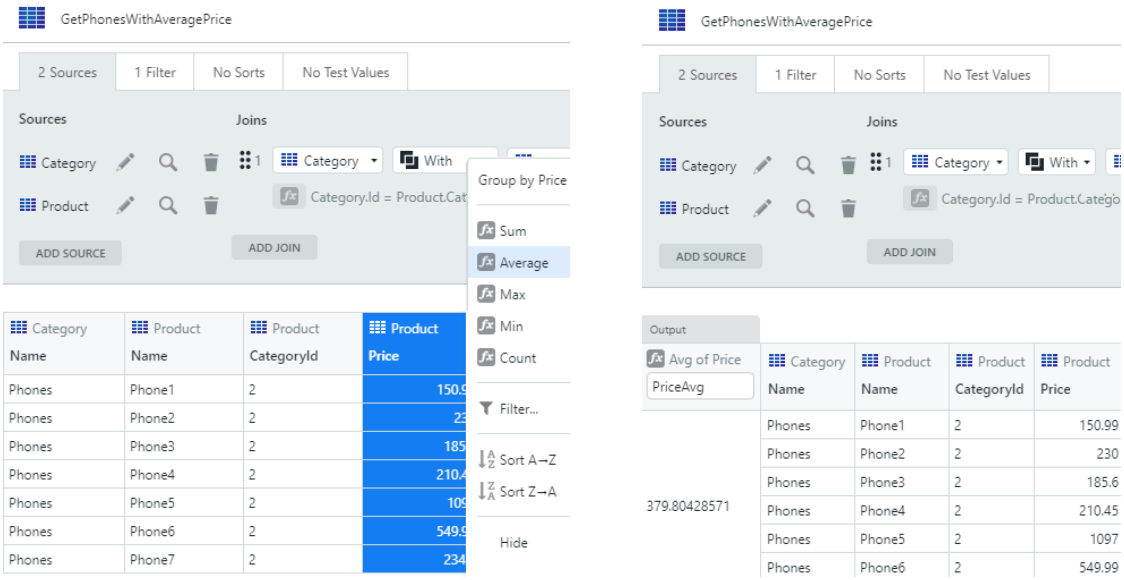
to implement database accesses in their applications.

Aggregates are visual elements of the OutSystems language that make it possible to query data in a relational database. Within an Aggregate, it is possible to define the source Entities, filter the data according to some criteria, sort the desired data and perform grouping/aggregation operations. Aggregates allow the developer to easily create and maintain queries, thus SQL knowledge is not required. The editor for Aggregates resembles a spreadsheet and can be used to preview the data being fetched (as seen in Figure 2.4). The Aggregate sources can be defined by simply dragging one or more previously defined Entities from the data tab into the Service Studio's flow editor. During the compilation step, **optimized SQL** is generated from the aggregate's definition. Regarding the Select part of the query, the OutSystems compiler automatically detects which attributes need to be fetched by performing code analysis.

Aggregates can be used to display a single **aggregated value**. Figure 2.5 shows the before and after of aggregating a column into a single value, resulting in an Aggregate (query) that returns the average price of phones in an application. The list of available aggregate functions depend on the column type. These functions are [38]:

- **Sum**: sums all the values in the column;
- **Average**: calculates the average of the values in the column;
- **Max**: finds the maximum value in the column;
- **Min**: finds the minimum value in the column;
- **Count**: counts how many rows there are in the column.

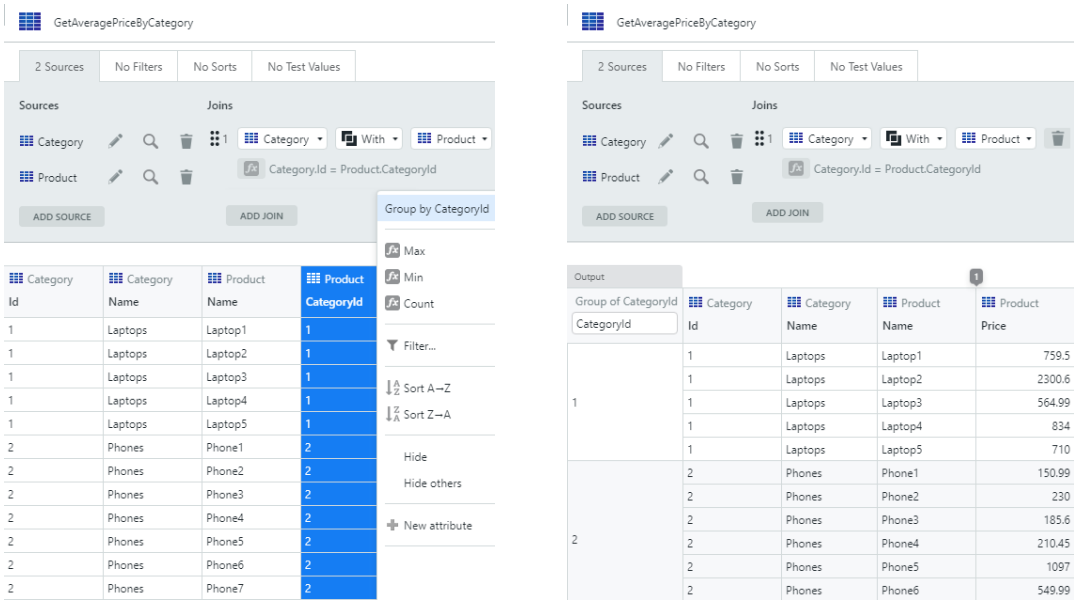
Aggregate functions are used to calculate values based on **groups of identical data** [39]. The **Group By** option is used to aggregate values into groups of rows that have the same



(a) Before Average aggregate function

(b) After Average aggregate function

Figure 2.5: Aggregate definition to get the average price of a phone



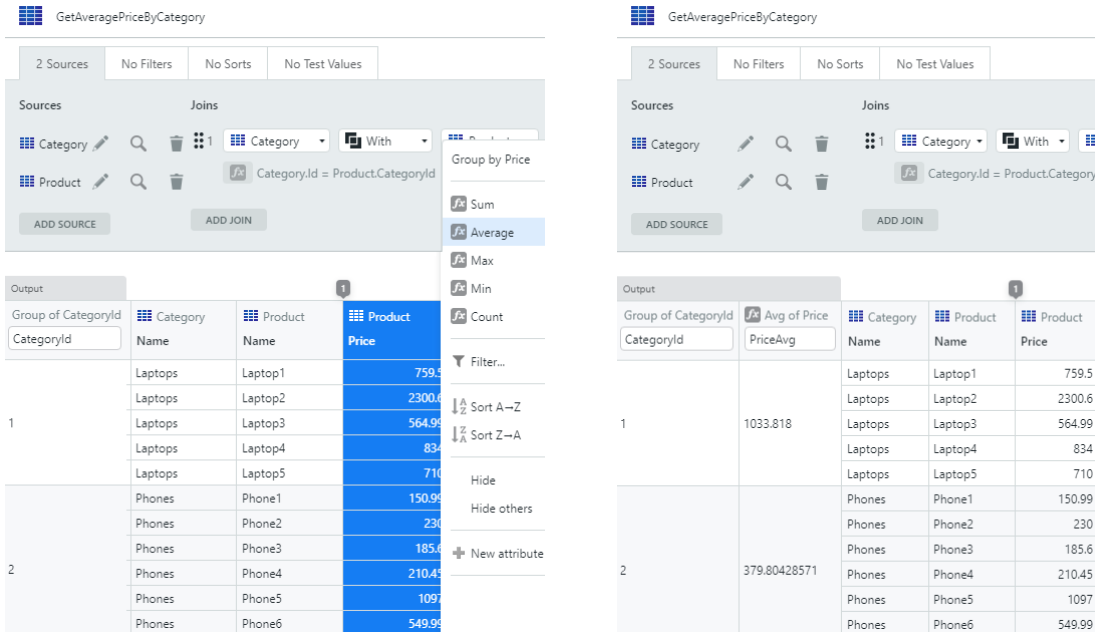
(a) Before Group By

(b) After Group By

Figure 2.6: Group products by id using an Aggregate

values, this can be done by selecting a column. After grouping or using aggregate functions on attributes, those attributes become the only output of the Aggregate. Figure 2.6 shows how to define an Aggregate that groups products by their category. It is then possible to calculate the average price of products per category as shown in Figure 2.7.

Calculated Attributes allow for records to be extended with additional information that can be computed from the data. This can be done by adding new attributes to



(a) Before Average aggregate function

(b) After Average aggregate function

Figure 2.7: Get the Average price of a product by category using an Aggregate

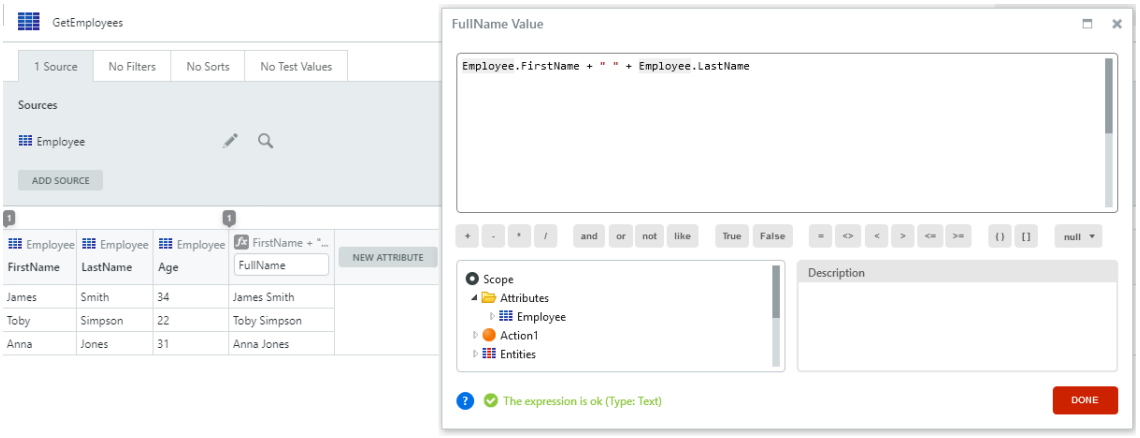


Figure 2.8: Aggregate for getting Employees and their Full Names

the records returned by the Aggregate based on the value of the other attributes [40]. Figure 2.8 shows the result of creating a **New Attribute** and defining an expression to concatenate the names of each employee (using the + operator).

2.6 Code Reuse and Refactor

Concerns over **maintainability** and **scalability** of programs rise as applications grow, this can be addressed by centralizing the logic in a reusable, modular way [41]. In OutSystems it is possible to define different types of modules in an environment and reference them as producer modules. The consumer modules use the elements of logic that the producer

modules expose.

Application modules built with OutSystems can share and reuse code that is made available for discovery and usage. Reusability is available at all application layers: UI, business logic, and database [42]. The reuse in each layer is described as follows:

- At the UI level developers can expose a variety of artifacts to be reused in any other OutSystems applications, such as **web blocks**, **themes** (that expose CSS), and **web screens**;
- Developers can expose any **block of business logic** to be reused in another business logic function, business process orchestration, web screen, mobile screen, or asynchronous job;
- **Roles** are frequently used in logic to control user interface elements (like screens) and business rules and can also be shared across multiple applications;
- **Database tables** (entities) designed and deployed using OutSystems when marked as public, can be reused by other applications.

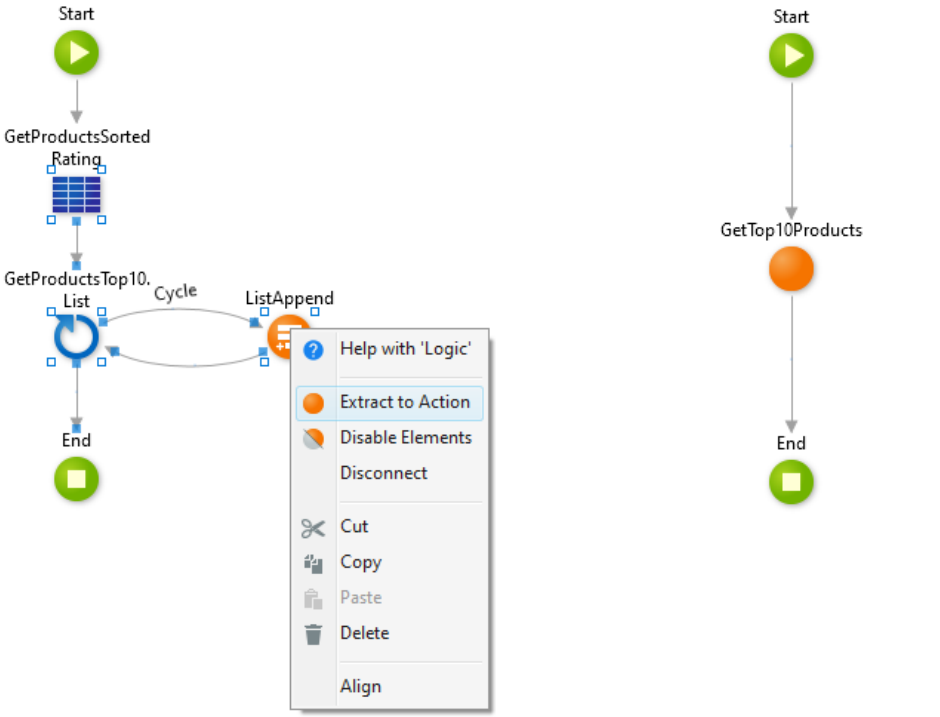
When implementing the logic of an application, it is possible to create actions in a module to invoke in other action flows through **Execute Action** nodes (for Client or Server Actions). This allows for the logic of an application to be centralized and makes the module easier to maintain, as a change in this logic will reverberate throughout the modules that reference it. If the logic is already implemented in a Client or Server action, it is possible to use the **Extract to Action** functionality to automatically create a new Action with that piece of logic. Figure 2.9 shows the before and after selecting the Extract to Action functionality in a logic flow of a Server Action.

More than just simplifying code reuse, OutSystems offers governance, i.e. the possibility to manage role capabilities, reports helping users understand the network of dependencies between modules/applications, and analysis of the impact of the changes made.

2.7 Architecture Dashboard

Technical debt is a measure of the cost of making changes to a piece of software. This can be caused by choosing an easy and limited solution, where any future changes to the software will become increasingly harder to perform. The time spent in carefully making these changes so that the system is not compromised may lead to a point where it might compensate to completely replace the solution with a new one [43] .

The **Architecture Dashboard** is the OutSystems technical debt monitoring tool. It is a static analysis tool for OutSystems code that identifies the anti-patterns that can lead to high technical debt problems. It allows Information Technology (IT) leaders to



(a) Before extract to action

(b) After extract to action

Figure 2.9: Extract logic to action in OutSystems

visualize complex cross-portfolio architectures and identify problems, while also helping developers follow best practices and avoid common pitfalls.

The goal of this tool is to effectively manage technical debt during every stage of development. This is accomplished by supplying a heat map visualization of mild-to-severe problem areas, IT leaders can quickly identify problem areas and prioritize accordingly. The Architecture Dashboard uses Artificial Intelligence (AI) to perform automated module classification and categorize the modules in an infrastructure. It is also possible to drill down into individual modules to view detailed reports on what best practices are being violated and their impact.

With Architecture Dashboard there is no longer the need to rewrite an application from scratch, as one can start managing the technical debt of OutSystems applications from the start [44].

2.7.1 Analysis

Architecture Dashboard performs two different types of analysis which are combined to provide context and greater relevance in findings: Code Analysis and Runtime Performance Analysis [45]. Additionally, the Architecture Dashboard uses AI for duplicated code detection.

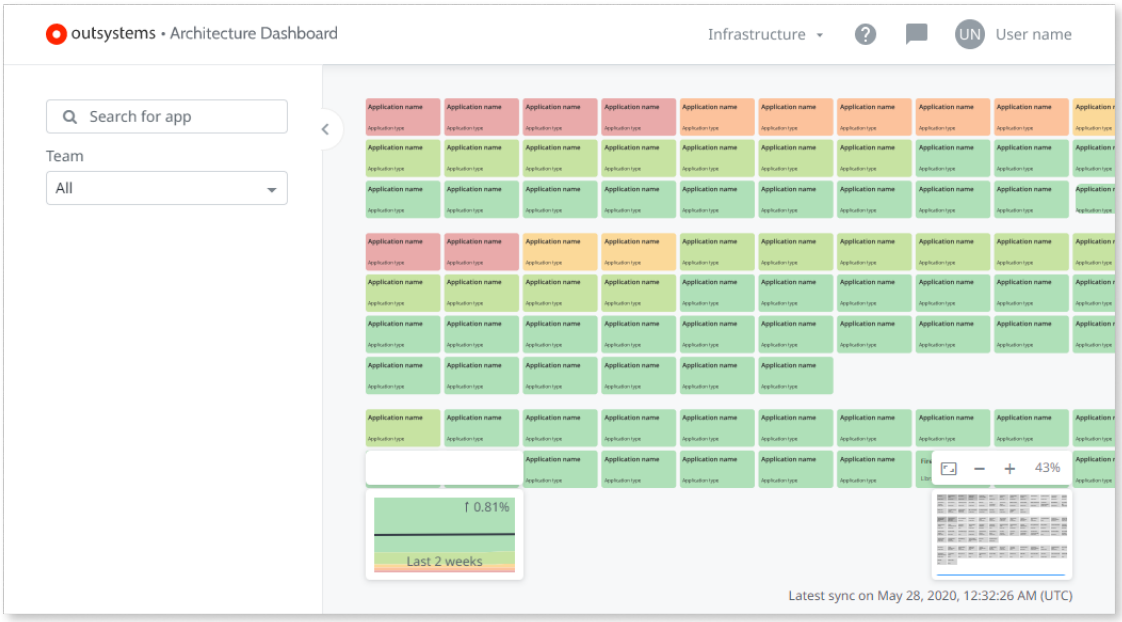


Figure 2.10: Architecture Dashboard Heat Map in [48]

In regards to **Code Analysis**, to analyze the produced low-code, Architecture Dashboard runs a set of predefined rules through probes connected with Modules, to uncover code patterns in the following categories: Performance, Architecture, Maintainability, and Security.

Regarding **Runtime Performance Analysis**, Architecture Dashboard uses the analytics data from LifeTime, which is crossed with the code analysis findings to display the most relevant and urgent targets for improvement, effectively helping address runtime performance issues.

2.7.2 Patterns

The patterns detected by Architecture Dashboard [46] are based on violations of the best practices from both the software development industry standards and development in OutSystems [47]. Therefore, each anti-pattern detected in an application adds technical debt [43].

Figure 2.10 shows an overview of all the applications in a user's infrastructure displayed by Architecture Dashboard. Each square is an application and the color of each square shows how high the technical debt is in that application. This measure of technical debt tells the user how difficult it is to change and maintain that application or module. By selecting an application, one opens a detailed report that can be used to can find and understand the causes of technical debt in that application.

3

Problem Statement

This chapter presents the methodology followed to identify and define the problem to be tackled in this thesis context.

3.1 Problem

SAPs enable the rapid development and delivery of applications and with minimal coding. Since OutSystems is an easy-to-use automation platform, it allows even the less skilled users to quickly build great and secure apps. Such users are more likely to not adhere to best practices of software development, producing applications with higher technical debt and that, consequently, are difficult to interpret, maintain, and/or have severe performance issues. The refactoring activity consists of applying changes that contribute to making software easier to understand and cheaper to modify. Duplicated code is commonplace in large software projects and is an anti-pattern that negatively influences technical debt, having a serious impact on maintainability.

Software systems are susceptible to be improved even in the presence of deficiencies in their internal quality that makes it harder to modify and extend them any further [33]. The increase in effort is a result of design and development decisions about the software that negatively affect its future [8].

Developers are often faced with issues such as changes in the requirements, interfaces, or functionality; time-to-market pressures that result in choosing the fastest and easiest approach and working with legacy systems of which they have insufficient knowledge [49]. Frequently, their response to these issues introduces technical debt.

As organizations strive to expedite time-to-market and empower non-professional developers to create business apps themselves, controlling technical debt naturally becomes a top concern. The Architecture Dashboard provides architects and OutSystems development teams with a high-level view of their modules' technical debt to identify problem areas and prioritize accordingly. It also points developers to occurrences of certain anti-patterns or lack of adherence to good software development practices, therefore avoiding code smells. With the Architecture Dashboard, technical debt can be effectively managed

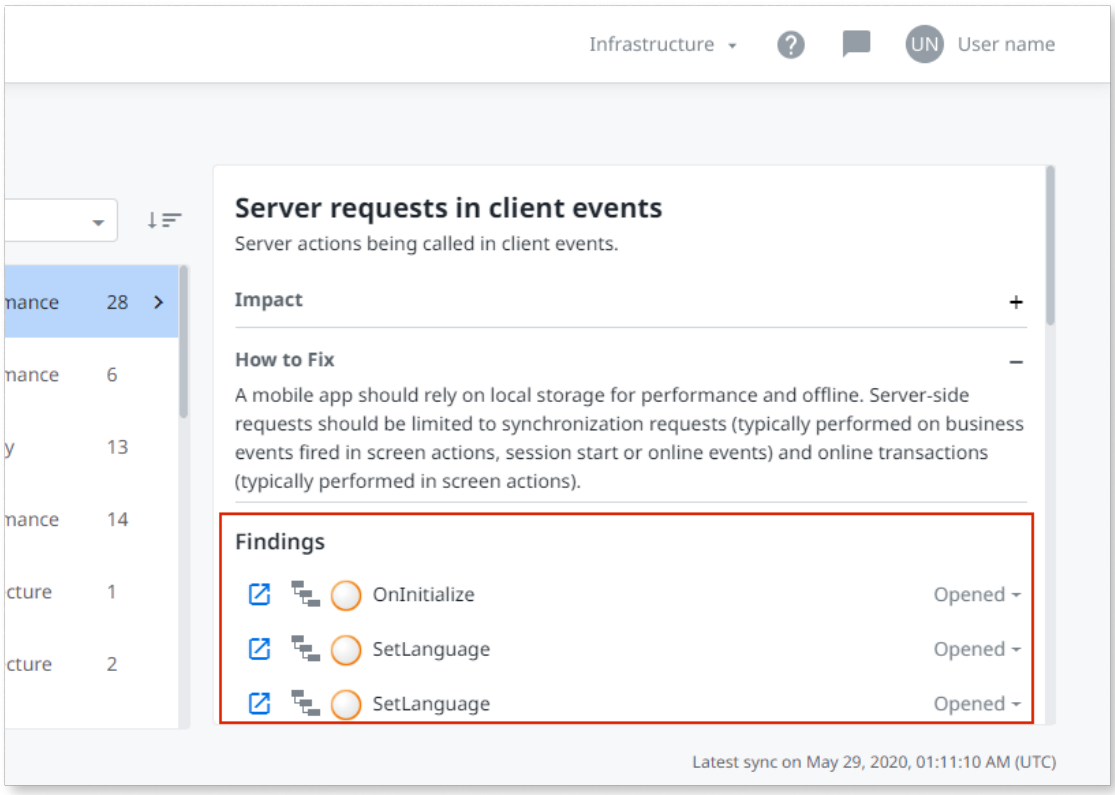


Figure 3.1: Architecture Dashboard Interface Details in [48]

at every stage of the development lifecycle so that when departmental applications evolve to become enterprise-wide solutions, nothing needs to be rewritten. Furthermore, developers can view detailed reports on what best practices are being violated, the impact of these violations, and how to fix them [45].

Figure 3.1 displays a snippet of the Architecture Dashboard Interface when analyzing the technical debt causes in an application. In this report, the user can visualize **The Impact**, which details why a code pattern creates technical debt, and the **How to fix** section, which explains how the user can fix that pattern. **Findings** shows all the occurrences of the code pattern, including the module and element in which it occurs. Nonetheless, the actual fixing of the causes of technical debt still needs to be performed by the developers.

The entire list of code analysis patterns detected by the Architecture Dashboard can be found in the OutSystems' documentation [46]. Currently, the list consists of 54 patterns affecting the following aspects of applications developed in OutSystems: **Architecture**, **Maintainability**, **Performance**, and **Security**. Figure 3.2 depicts the evolution of technical debt in OutSystems between January 2020 and January 2021. It is possible to deduce that, as the number of infrastructures increases, so does the amount of technical debt found. The tool automatically processes each registered infrastructure twice a day but also answers to manual synchronization requests. The small positive/negative variations

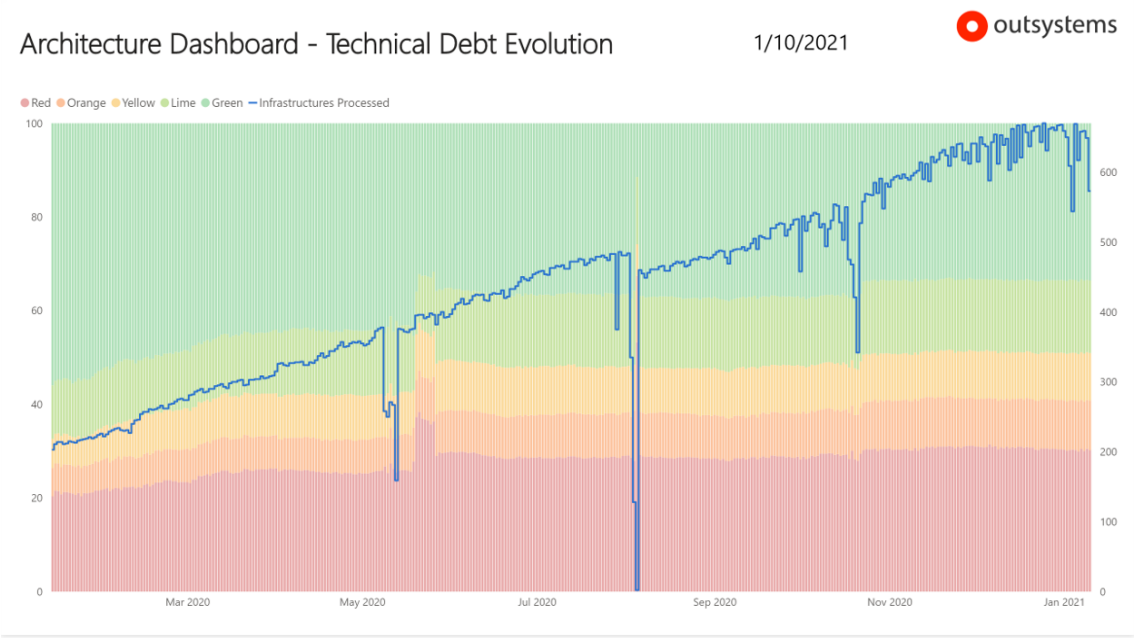


Figure 3.2: Architecture Dashboard - Technical Debt Evolution

between consecutive days correlate to these requests. The accentuated negative dips correspond to days where the tool has synchronization problems and is not able to process as many requests. As of 10/01/2021, 611 customers were using Architecture Dashboard, with a total of 6,726 registered users and 573 infrastructures. The previous statistics are the result of internal analysis.

3.2 Problem Statement

In most companies, different people need to work on the same project and might not be familiar with the code base. If the code base contains defective code, evolving it by adding new features may contribute to the propagation of bugs to the point that fixing a problem might trigger many others. Consequently, the effort it takes to correct software defects and maintain the existing code base might make such tasks unfeasible, and thus rewriting the code might become the developers' only reasonable option. The scale of the technical debt problem deepens in severity for SAPs since these platforms enable users with a less technical background to become proficient developers. As increasing technical debt is a reality of applications built with SAPs, and has been identified as also the case for the OutSystems platform, it appears as a relevant problem to be addressed by this thesis.

Regarding the OutSystems context, the Architecture Dashboard identifies with high precision where the technical debt is present in OutSystems applications, providing a link between the problem identified and the code from which it arose. The current solution relies on the competence of the developers in following a set of guided instructions to fix

the detected anti-patterns. As previously mentioned, these developers do not need to be the most experienced in software development, thus correcting the technical debt causes might prove to be an arduous task. The problem derives from the lack of instrumentation currently available to obtain a reliable and provably correct resolution. With this in mind, the need to introduce an automated resolution to solve the problem of high technical debt in SAPs was identified. The main research question that is being addressed in this work is:

> Can we lower the high technical debt problem in Software Automation Platforms with automated refactoring techniques?

The Architecture Dashboard tool successfully identifies the causes of high technical debt in OutSystems logic. Therefore, this work shall focus on the resolution of issues that cause high technical debt, including a severe problem that gravely influences code maintainability, that is the Duplicated Code anti-pattern. Nonetheless, the study of this problem domain as well as detection techniques are addressed in the following chapters: Background and Related Work. It would be unfeasible for this work to automatically solve all the causes of high technical debt in an application, because of their quantity and due to a lack of information required to fix some of these anti-patterns (e.g. knowledge of the application's high-level semantics for missing documentation). It was thus necessary to select a set of relevant high-impact anti-patterns to be tackled in this work. Furthermore, this observation will be extended in Technical Approach.

Decreasing the effort required from SAPs developers to resolve high technical debt problems is of utmost importance. This means that modifying OutSystems logic by hand, to solve a detected anti-pattern, should not be required from the developer, which is the current state of the art in OutSystems. The necessary changes may easily be discarded, forgotten, or badly executed. Such a scenario might endanger the overall sanity of the system, as it can lead to the introduction of new bugs, or even promote the increase of technical debt in some other area. This highlights the need for introducing an automated solution for resolving technical debt problems and is thus the focus of this thesis.

The goal of this work is to devise techniques for automated refactoring in SAPs, focusing on OutSystems as a case study.

4

Background

This chapter introduces the fundamental concepts necessary for a complete understanding of our work. To that end, we start by describing the fundamental concepts of Database Systems and query size estimation followed by an in-depth discussion on the subject of Refactoring. Lastly, we discuss the fundamentals of Graphs and present the state of the art on duplicated code detection including detection in visual programming languages.

4.1 Database Systems

A Database-Management System (DBMS) is a collection of interrelated data and a set of programs that allow users to access and modify these data [9]. DBMS are used to manage collections of data and contain information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information conveniently and efficiently.

To manage the data it is necessary to: define structures for the storage of information and to provide mechanisms for the manipulation of information. Additionally, the database system must ensure the safety of the information being stored, even in the event of system crashes or attempts at unauthorized access.

Underlying the structure of a database is the **data model** which is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. The data models can be classified into four different categories [9]:

- **Relational Model** - The relational data model is the most widely used and a vast majority of current database systems are based on the relational model. In the relational model, a collection of tables is used to represent both the data and the relationships between the data. Each table can have multiple columns, and each column has a unique name. Tables are also known as relations. The relational model is a record-based model, i.e. the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields or attributes. The columns of the table correspond to the attributes of the record type.

- **Entity-Relationship Model** - The entity-relationship data model (ER model) is widely used in database design. It uses a collection of basic objects, called entities that are distinguishable from other objects, and relationships among these objects.
- **Semi-structured Data Model** - The semi-structured data model allows for the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. JSON and Extensible Markup Language (XML) are widely used semi-structured data representations.
- **Object-Based Data Model** - The object-oriented data model extends the relational model with notions of encapsulation, methods, and object identity. The concept of objects is well integrated into relational databases thus, there are standards to store objects in relational tables and to store and execute procedures in the database system.

We will now focus on the relational model as it serves as the foundation for Entities in the OutSystems data model.

4.1.1 Relational Model

A relational database consists of a collection of tables and each table has a unique name. A row in a table represents a relationship between a set of values. A relational database takes its name because a table is a collection of such relationships having a correspondence between the concept of table and the mathematical concept of relation. In the relational model, the term **relation** is used to refer to a table, the term **tuple** is used to refer to a row and the term **attribute** refers to a column of a table. Since a relation is a set of tuples, the order in which tuples appear in a relation is irrelevant.

4.1.1.1 Database Schema

The **database schema** is the logical design of the database whereas the **database instance** is a snapshot of the data in the database at a given instant in time. The concept of **relation schema** corresponds to the programming language notion of **type definition** whilst the concept of relation corresponds to the notion of a **variable**. A relation schema consists of a list of attributes and their corresponding domains.

4.1.1.2 Keys

Two tuples in a relation cannot have the same value for all attributes. Tuples can then be distinguished because their attribute values uniquely identify them. The conception of keys poses as the means to identify these tuples, their different types are described as follows:

- A **superkey** is a set of one or more attributes that identify uniquely a tuple in the relation. Some attributes can be considered **extraneous** if they can be removed without changing the closure of the set of functional dependencies.

- A **candidate key** is a superkey for each no subset is also a superkey. There can be distinct sets of attributes that serve as a candidate key.

- The **primary key** is the candidate key that is chosen as the principal means to uniquely identify a record within a relation. The designation of a key represents a constraint in the real-world enterprise being modeled. Thus, primary keys are also referred to as **primary key constraints**.

- A **foreign key** constraint from attribute(s) a of relation R_1 to the primary key b of relation R_2 means that on any database instance, the value of a for each tuple in R_1 must also be the value of b for some tuple in R_2. The attribute set a is a foreign key from R_1, referencing R_2. In a foreign key constraint, the referenced attribute(s) must necessarily be the primary key of the referenced relation.

4.1.2 Relational Query Languages

A **query** is a computation upon relations that produces other relations whilst a **query system**, such as relational algebra, is a formal system for expressing queries. **Query languages** are programming languages used in database systems to formulate commands [10]. A query can also be defined as a "question about the data" [11].

Query languages can be categorized as [9]:

- **Imperative**, the user instructs the system to perform a sequence of operations on the database to compute the result.

- **Functional**, the computation is expressed as the evaluation of functions that operate on data in the database or on results of other functions.

- **Declarative**, the user describes the desired information without giving a specific sequence of steps or function calls for obtaining that information. It is the job of the database system to reason how to obtain the desired information.

The **relational algebra** is a functional query language and forms the theoretical basis of the SQL query language. The **SQL query language**, includes elements of the imperative, functional, and declarative definitions.

4.1.3 Relational Algebra

The **relational algebra** consists of a set of operations that take one or two relations as input and produce a new relation as their result. **Unary operations** operate on one

relation (select, project, rename, *etc*) whilst **binary operations** operate on two relations (union, Cartesian product, set difference, *etc*).

Since a relation is a set of tuples, relations **cannot contain duplicate tuples**. However, tables in database systems are permitted to contain duplicates unless a specific **constraint** prohibits it (e.g. primary key constraint).

Relational algebra supports only a small number of predefined functions, which define an algebra on relations. Next, we will describe the available operations in relational algebra [9] followed by a description of some of the extended operators of relational algebra (duplicate elimination, aggregation operators, grouping of tuples, extended projection, and sorting operator) [11].

4.1.3.1 The Select Operation

The select operation selects tuples that satisfy a given predicate. Selecting the tuples of relation R where attribute a of R is equal to some constant c is represented by:

$$\sigma_{a=c}(R)$$

The selection predicate may include comparisons between two attributes using the operators $=, \neq, <, \leq, >$, and $\geq$, as well as the combinations of several predicates using the connectives *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). Selecting the tuples of relation R where attribute a of R is equal to some constant c and attribute b of R is greater than a is represented by:

$$\sigma_{a=c \wedge b>d}(R)$$

4.1.3.2 The Project Operation

The project operation is a unary operation and results in displaying only the attributes specified in the argument. Duplicate rows are eliminated because a relation is a set. Projecting attribute b of relation R is represented by:

$$\Pi_b(R)$$

4.1.3.3 Composition of Relational Operations

The result of a relational operation is also a relation. Thus, **relational-algebra operations** can mainly be composed together into a **relational-algebra expression**. Selecting the tuples of relation R where attribute a of R is equal to some constant c and projecting only attribute b of R is represented by:

$$\Pi_b(\sigma_{a=c}(R))$$

4.1.3.4 The Cartesian-Product Operation

The Cartesian-product operation combines information from two relations and results in a tuple for each possible pair of tuples, one from the first relation and one from the

second. The Cartesian product of relations R_1 and R_2 is represented by:

$$R_1 \times R_2$$

4.1.3.5 The Join Operation

The join operation is the combination of a selection and a Cartesian product in one operation. Consider relations R_1 and R_2, and let θ be a predicate on attributes in the schema $R \cup S$. The join operation between R_1 and R_2 and is defined as follows:

$$R_1 \bowtie_\theta R_2 = \sigma_\theta(R_1 \times R_2)$$

4.1.3.6 The Natural Join Operation

The natural join operation uses an implicit predicate to replace the predicate θ in $\bowtie \theta$ that requires equality between the attributes that are common to both relations being joined. The natural join operation between $R_1(a, b)$ and $R_2(b, c)$, having b as the only common attribute between R_1 and R_2, is defined as follows:

$$R_1 \bowtie R_2 = R_1 \bowtie_{R_1.b=R_2.b} R_2$$

4.1.3.7 The Duplicate-Elimination Operation

The **duplicate-elimination operation** δ turns a bag into a set by eliminating all but one copy of each tuple. Thus, to return the set consisting of one copy of every tuple that appears one or more times in relation R we use the following:

$$\delta(R)$$

4.1.3.8 The Aggregation Operation

The **aggregation operation** takes a collection of values and returns a single value as a result. It allows a function to be computed over the set of values returned by a query by applying a function to the attributes (columns) of a relation. The standard operators of this type are:

- **AVG**: produces the average of a column with numerical values.
- **MIN**: produces the smallest value (when applied to a column with numerical values) or the first lexicographically value (when applied to character-string values).
- **MAX**: produces the largest value (when applied to a column with numerical values) or the last lexicographically value (when applied to character-string values).
- **SUM**: produces the sum of a column with numerical values.

- **COUNT**: produces the number of values in a column (these are not necessarily distinct). Equivalently, COUNT applied to any attribute of a relation produces the number of tuples of that relation, including duplicates.

It is also possible for these aggregations to be performed after splitting the set of values into groups (Group By operation). Having E as a relational-algebra expression, $F_1, F_2, ..., F_i$ as the set of aggregation operations to perform and $a_1, a_2, ..., a_i$ as attributes each having a collection of values to pass to the functions, the general formula for the aggregate operation in relational algebra if the set of attributes on which to group is empty, is defined as follows:

$$\mathcal{G}F_1(a_1), F_2(a_2), ..., F_i(a_i)(E)$$

4.1.3.9 The Grouping Operation

Grouping of tuples according to their value in one or more attributes has the effect of partitioning the tuples of a relation into "groups". Aggregation can then be applied to columns within each group. The **grouping operator** $\mathcal{G}$ (γ in some books) is an operator that combines the effect of grouping and aggregation.

Having E as a relational-algebra expression, $g_1, g_2, \ldots, g_k$ as attributes on which to group (can be empty), $F_1, F_2, ..., F_i$ as the set of aggregation operations to perform and $a_1, a_2, ..., a_i$ as attributes each having a collection of values to pass to the functions, the general formula for the group by operation in relational algebra is defined as follows:

$$g_1, g_2, \ldots, g_k \mathcal{G} F_1(a_1), F_2(a_2), ..., F_i(a_i)(E)$$

The grouping operation groups the result relation E into sets whose values are equal in the attributes $g_1, g_2, \ldots, g_k$ (if $k = 0$, E is the sole group). For each group, it returns a tuple with the values of $g_1, g_2, \ldots, g_k$ and the result of applying the functions $F_1, F_2, ..., F_i$ in the set of values of that group. The result of the aggregate operation will be a single tuple if the set of attributes to group is empty ($k = 0$) because one tuple is returned for each group and only one group was formed.

4.1.3.10 The Extended Projection Operation

In addition to the Π operator projecting some designated columns, the **extended projection** can perform computations involving the columns of its argument relation to produce new columns. Considering two numeric attributes a and b of relation R, in an extended projection we can produce a new column as the sum of the two elements as follows:

$$\Pi_{a+b}(R)$$

4.1.3.11 The Sorting Operation

The **sorting operator** τ turns a relation into a list of tuples, sorted according to one or more attributes. Getting the results of relation R sorted according to attributes $a_1, a_2, ..., a_n$ of R in the order they are indicated, is defined as follows:

$$\tau_{a_1,a_2,...,a_n}(R)$$

4.1.4 SQL

The SQL language can be used to define the structure of the data, modify data in the database, and specify security constraints. In this analysis, we will focus on the definition of SQL queries.

4.1.4.1 Query Structure

The basic structure of an SQL query consists of three clauses: SELECT, FROM, and WHERE. A query takes as its input the relations listed in the FROM clause, operates on them as specified in the WHERE and SELECT clauses, and then produces a relation as the result. The role of each clause is as follows:

- The **SELECT** clause is used to list the attributes desired in the result of a query.
- The **FROM** clause is a list of the relations to be accessed in the evaluation of the query.
- The **WHERE** clause is a predicate involving attributes of the relation in the FROM clause.

The typical SQL query has the following form:

```
SELECT $a_1, a_2,..., a_i$
FROM $R_1, R_2,..., R_n$
WHERE $P$;
```

Each a_i represents an attribute, each R a relation, and P is a predicate. If the where clause is omitted, the predicate is true. The FROM clause by itself defines a Cartesian product of the relations listed in the clause and the result relation has all attributes from all the relations in the FROM clause. When the same attribute name may appear in the relations of the FROM clause, the name of the relation from which the attribute originally came is prefixed before the attribute name (e.g. R.a). Some relevant basic operations supported in SQL include:

- **Attribute specification** - The asterisk symbol * can be used in the SELECT clause to denote the inclusion of all the attributes of a relation (e.g. if R is a relation, `SELECT R.* FROM R(a,b)`, selects all the attributes of relation R). A SELECT clause

of the form: `SELECT *`, indicates that all attributes of the result relation of the from clause are selected.

- **Ordering the display of tuples** - The **ORDER BY** clause causes the tuples in the result of a query to appear in sorted order.

4.1.4.2 Aggregation

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions: AVG, MIN, MAX, SUM, and COUNT (see Section 4.1.3.8). These operators are used by applying them to a "scalar-valued expression" in a SELECT clause with the exception of the expression COUNT(*), which counts all the tuples in the relation that is constructed from the FROM clause and WHERE clause of the query [11]. The input to "SUM" and "AVG" must be a collection of numbers, whilst the other operators can operate on collections of non-numeric data types (e.g. strings). Additionally, we have the option of eliminating duplicates from the column before applying the aggregation operator by using the keyword DISTINCT (e.g. an expression such as COUNT(DISTINCT a) counts the number of distinct values in column a). As an example, suppose we wanted to count the number of tuples in relation R, a query defined for this purpose would be:

```
SELECT COUNT(*)
FROM $R$;
```

4.1.4.3 Grouping

We can split the tuples of a relation in **groups**, according to the value of one or more other columns, when we do not want the result of the aggregate function to be computed over an entire column but rather, we want to aggregate only within each group. The attribute(s) given in the GROUP BY clause are used to form groups. Tuples with the same value on all attributes in the GROUP BY clause are placed in one group.

As an example, consider relation R and two attributes a (string) and b (integer), suppose we wanted to compute the average of b according to each a, a query defined for this purpose would be:

```
SELECT $a$, AVG($b$)
FROM $R$
GROUP BY $a$;
```

When an SQL query uses grouping, it is important to ensure that the only attributes that appear in the SELECT clause without being aggregated are those that are present in the GROUP BY clause, i.e. any attribute that is not present in the GROUP BY clause may appear in the SELECT clause only as an argument to an aggregate function. If we select a column, not in the GROUP BY clause, there is no way of choosing which column value

to output since we will have to display only one value for column, which is either being grouped or aggregated. Thus, such cases are disallowed by SQL [9].

4.1.5 Estimating the Cost of Operations

Garcia-Molina *et al.* proposed a methodology for estimating the cost of operations in relational databases (query plans) [11]. In the cost estimation methodology, the parsed query is transformed into an abstract, i.e. logical, query plan before being turned into a physical plan. Different physical plans can be derived from the abstract plan, these are then evaluated to estimate their cost. Cost-based enumeration consists of choosing the physical query plan with the least estimated cost.

In our work, we are concerned with estimating the number of tuples in the result relations after performing relational algebra operations, so we can define rules according to the size of query results. In this section, we will show estimated statistics on the results of various relational operations following the work by Garcia-Molina *et al.* [11] and by Silberschatz *et al.* [9].

To estimate costs of plans accurately, Garcia-Molina *et al.* used parameters that are computed from the data or "estimated by a process of statistics gathering". We use the same notation as in the book for the parameters that define the number of tuples in a relation:

- $\mathrm{T}(R)$ is the number of tuples of relation R.
- $\mathrm{V}(R, a)$ is the value count for attribute a of relation R, that is, the number of distinct values relation R has for attribute a [11]. This value is the same as the size of $\Pi_a(R)$ if a is a key for relation R, $\mathrm{V}(R, a) = \mathrm{T}(R)$ [9].

4.1.5.1 Estimating the Size of a Selection

When we perform a selection, we generally reduce the number of tuples, although the sizes of tuples remain the same. In the simplest kind of selection, where an attribute is equated to a constant, there is a way to estimate the size of the result if we know the number of different values the attribute has. Given the following selection operation on an equality condition: $\sigma_{a=c}(R)$

Having c as a constant and a as an attribute of R, an estimate would be [11]:

$$T(\sigma_{a=c}(R)) = \frac{T(R)}{V(R,a)}$$

A conjunctive selection is a selection of the form [9]: $\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(R)$

We can estimate the result size of such a selection by estimating the size of the selection for each θ_i. The probability that a tuple in the relation satisfies selection condition θ_i is $\frac{T(\sigma_{\theta_i})}{T(R)}$ and is called the selectivity of the selection σ_{θ_i}. Assuming that the conditions

are independent of each other, the probability that a tuple satisfies all the conditions is the product of all these probabilities, as follows:

$$T(\sigma_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n}(R)) = T(R) \times \frac{T(\sigma_{\theta_1}) \times T(\sigma_{\theta_2}) \times \cdots \times T(\sigma_{\theta_n})}{T(R)^n}$$

4.1.5.2 Estimating the Size of a Join

A **natural join** of two relations pairs tuples that share the same values for the attributes that are common to both tables. Considering a natural join of two relations R and S that share a single attribute a:

1. If R and S have disjoint sets of values for the common attribute, the size of the query result is:
$$T(R \bowtie S) = 0$$

2. If a is a key in R and a foreign key in S, then each tuple of S joins with exactly one tuple of R and thus the size of the final query result is:
$$T(R \bowtie S) = T(S)$$

3. If almost all tuples of R and S could have the same value for a, therefore pairing almost every tuple of R with each tuple of S, the estimated query size is:
$$T(R \bowtie S) = T(R) \times T(S)$$

Having $a_1, a_2, ..., a_n$ as the set of common attributes between R and S, the size of the query result is:

$$T(R \bowtie S) = T(R) \times T(S) \times \prod_{i=1}^{n} \frac{1}{max(V(R, a_i), V(S, a_i))}$$

Regarding other types of joins:

1. For an equijoin the number of tuples in the result can be computed like the natural join, after accounting for the change in variable names.

2. For a cartesian product (cross join) the number of tuples in the result is the product of the number of tuples in the relations involved, as follows:
$$T(R \times S) = T(R) \times T(S)$$

3. For theta-joins the number of tuples in the result can be estimated as if it is a selection following a product.
$$T(R_1 \bowtie_\theta R_2) = T(\sigma_\theta(R_1 \times R_2))$$

For multiple joins the estimate we can assume, as the upper bound value, it being the product of the number of tuples in each relation. In this estimate we use $\bowtie_\theta$ but it could be any type of join, as follows:

$$T(R_1 \bowtie_\theta R_2 \bowtie_\theta \ldots \bowtie_\theta R_n) = T(R_1) \times T(R_2) \times \cdots \times T(R_n)$$

Then, for each attribute a appearing at least twice, we can divide by all but the least of the $V(R,a)$'s. Likewise, we can estimate the number of values that will remain for attribute a after the join. By the "preservation-of-value-sets assumption", the estimate is the least of these $V(R,a)$'s [11].

4.1.5.3 Estimating the Size of a Projection

The estimated number of tuples of a projection of the form $\Pi_a(R)$ will be equal to $V(R,a)$, since the projection eliminates duplicates [9]. If a is the primary key of R, and thus, and no duplicate tuples are allowed in the primary key, the projection operator will leave the number of tuples in the result unchanged, as follows:

$$T(\Pi_a(R)) = V(R,a) \leq T(R)$$

Usually, tuples shrink during a projection, as some attributes are excluded. However, the projection allows for the creation of new attributes as combinations of attributes, and in these cases, the projection can increase the size of the relation but not the number of tuples.

4.1.5.4 Estimating the Size of a Grouping and Aggregation

The number of tuples returned by the group by operation is the same as the number of groups [11]. The lower bound would be one group in the result or as many groups as there are tuples in the relational algebra expression. Considering the general formula for the group by operation including aggregation functions:

$$g_1, g_2, \ldots, g_k \mathcal{G} F_1(a_1), F_2(a_2), \ldots, F_i(a_i)(E)$$

Having E as a relational algebra expression and $g_1, g_2, \ldots, g_k$ as the attributes on which to group by, an estimate would be:

$$T(g_1, g_2, ..., g_k \mathcal{G} F_1(a_1), F_2(a_2), ..., F_i(a_i)(E)) = V(E, [g_1, g_2, \ldots, g_k])$$

In this estimate we consider that the value of $V(E, [g_1, g_2, \ldots, g_k])$ is obtainable.

4.2 Refactoring

Refactoring is the designation given to both "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its

observable behavior" and also to "restructure software by applying a series of refactorings" [1].

When refactoring, the code's intentions should remain the same, it should do exactly what it did before. This does not mean it will work exactly in the same way (as performance characteristics might change) but nothing should change that the user should care about.

Software design and architecture used to be fixed and completed before writing the code. Refactoring has changed this perspective and allows to significantly alter the software architecture that has been running for years, therefore it is possible to say that "refactoring can improve the design of existing code" [12].

This section starts by describing refactoring concerning other important aspects of software and what drives us to perform it. Next, we present the work methodologies to perform refactoring by presenting different relevant patterns and some of the most relevant refactoring techniques regarding this work will be explained. Finally, we give an overview of Duplicated Code and how it is currently detected in OutSystems.

4.2.1 Refactoring and Architecture Design

The impact of refactoring on architecture is how it can be used to form a well-designed code base that can respond gracefully to changing needs. It is very difficult to finish the architecture before coding because the requirements for the software may not be finished or well understood. One approach to deal with future changes is to make the software flexible but anticipating the change can slow down the process of reacting to the change [12].

Refactoring permits a different strategy, by focusing on building software that solves the current requirements with the best design possible. As the understanding of users' needs changes, refactoring makes it possible to adapt the architecture to these demands without increasing complexity. This approach is commonly known as Incremental Design and is one of the practices used in Extreme Programming. It allows developers to keep investing in the design of the system even after implementation and in proportion to the needs of the system [13].

4.2.2 Refactoring and Performance

Refactoring is very similar to performance optimization, as both involve carrying out code manipulations that do not change the overall functionality of the program. The difference between refactoring and performance optimizations lies in the purpose. Refactoring is done to make the code "easier to understand and cheaper to modify"(which might make the program faster or slower), performance optimization concerns are only to make the program faster (which might lead to code that is harder to understand) [1].

A common concern with refactoring is the effect it has on the performance of a program. Making the software easier to understand often leads to changes that will cause

the program to run slower. These refactoring steps are still necessary because even if they make the program slower, they also make the software more amenable to performance tuning. On the other hand, the performance of software usually depends on just a few parts of the code, so most of the changes introduced in refactoring won't make an appreciable difference [12].

4.2.3 Testing

The first foundation for *hand refactoring* is to have self-testing code. It is necessary to have a suite of automated tests to be run regularly so that if there was an error made while programming, some test will fail. These tests will help find the mistakes that a developer will inevitably do, the larger the program, the more likely the affected changes will cause some undesirable effect in the software. If refactoring is done in small steps, it is easy to find where and when the bug was introduced. These tests should be self-checking to avoid spending time hand-checking values from the test against the desired values [1].

Automated Refactoring allows the introduction of modifications to the code base that are provenly correct. But even with automated refactoring tools, some of the refactorings will still need checking via a test suite. Even without refactoring, writing good tests increases the effectiveness of a programmer. Because programmers end up spending most of their time debugging, a suite of tests is a powerful bug detector that reduces the time it takes to find bugs [12].

4.2.4 Advantages

A poorly designed system is hard to change because it is difficult to understand what needs to be changed and how these changes will affect the existing code. This difficulty will also make the developer more prone to make mistakes and may end up introducing bugs. This process tends to be slower than simply working with a well-structured program. The aforementioned reasons, highlight the importance of the following conclusion. When it is necessary to add a feature to a program, if the code is not structured conveniently, refactoring the program first will make it easier to add the feature. In this section, we present the many reasons to make refactoring a priority [1, 12].

4.2.4.1 Improves the Design of Software

Software architecture tends to decay as changes in code are introduced without regard for maintaining structure. The harder it is to see the design in the code, the harder it is to preserve it, and the more rapidly it decays. Regular refactoring will aid in avoiding this loss of structure and help improve software design.

4.2.4.2 Makes Software Easier to Understand

The importance of making software easier to understand comes from the necessity of having different users (or even the same user at a different time) reading the source code produced and eventually having to change it. When developing we often only think about getting the program to work and making code easier to understand will naturally affect this rhythm. Nonetheless, refactoring helps make the code more readable, as developing often leads to code that works but is not ideally structured. Because one easily forgets what was the intention of the code produced, the moment it was written, these considerations should be put into the code.

> "Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
>
> — Martin Fowler

4.2.4.3 Helps In Finding Bugs

Refactoring helps at being more effective at writing robust code. When refactoring, there is time spent understanding the code. By putting that understanding into the code and by clarifying the structure of the program, it is much easier to find bugs.

4.2.4.4 Helps In Programming Faster

Refactoring does not only improve quality (better internal design, readability and reduces bugs) but also makes it easier to find how and where to make the necessary changes to add a new feature in software with a good internal design. Good modularity allows one to only need to understand a small subset of the code base to make a change. If the code is clear, the probability of introducing bugs or having to spend a lot of time debugging drops.

4.2.5 Disadvantages

Like all good engineering practices, refactoring is a valuable technique, but it comes with its tradeoffs and there is a need to know when and where to apply it. In this section, we will get into detail about some of the problems that emerge because of refactoring [12].

4.2.5.1 Slowing Down New Features

Refactoring intends to make the development of new features faster and not the contrary. The whole purpose of refactoring is to make developers program faster, producing more value with less effort. The tradeoff between the time it takes to add the new feature and the time needed to refactor should pose as a guideline.

It is dangerous to justify refactoring in terms of “clean code” or as a “good engineering practice” because the point of refactoring is not to show “pretty code” but it is purely

economic. Refactor makes the developer faster at developing new features and at fixing bugs. The **economic benefits of refactoring** should always be the driving factor.

4.2.5.2 Introducing Bugs

One of the most important characteristics of refactoring is that it does not change the observable behavior of the program. Even by taking the greatest care in each refactoring step, when refactoring manually mistakes can still be introduced. This explains why some are concerned that refactoring carries too much risk of introducing bugs. This step back can be solved by having a solid set of tests. If these tests are run quickly and often, finding bugs is extremely easy.

Another way of dealing with this problem is by using an environment with **automated refactorings** that can be trusted without running tests. Automated refactoring was a revolutionary introduction to refactoring by enabling the performance of refactorings with a lot of confidence.

4.2.6 Automated Refactoring

Automated Refactoring is "perhaps the biggest change to refactoring in the last decade or so is the availability of tools that support automated refactoring"(Fowler, 2018). The first tool to appear with automated refactoring functionalities was the **Smalltalk Refactoring Browser** [50]. Since then, the availability of automated refactorings in environments like the IntelliJ IDEA [51] or Eclipse [52] have allowed doing refactorings like renaming variables, methods, or classes in a most simplistic way. Automated tools often go further by suggesting changes in reaction to others, such as prompting to change comments after renaming a variable.

To do refactoring properly, an automated tool has to operate on the **syntax tree of the code**, not only on the text. Manipulating the syntax tree is much more reliable to preserve what the code is doing. Therefore, at the moment, most refactoring capabilities are part of powerful IDEs, because these use the syntax tree not just for refactoring but also for code navigation [12].

On the other hand, many refactorings are made much safer when applied in a language with **static typing**. For example, when renaming a method of a class that exists with the same name in another class, without static typing, the tool will find it difficult to tell whether any call to that method is intended for each of the classes.

4.2.7 Code Smells

To understand when to refactor it is useful to look for certain structures in the code that suggest the possibility of refactoring. This is not a precise criterion but an indication that there might be a problem in the code that can be solved by refactoring [12]. These structures go by different names including code smells, bad smells, and **anti-patterns**.

Code smells often lead to bugs, runtime errors, and software maintenance difficulties. Consequently, they should be systematically prevented and fixed all along the software lifecycle.

Below we present a list with some code smells. This list is not exhaustive and the problems detected may not look exactly like the ones referenced here, this list serves as a guide for what to start looking for or what might be possible to do to solve it [1, 12]. We also provide a description of the relationship between the code smells and patterns analyzed with the Architecture Dashboard tool.

1. **Duplicated Code** consists of a repetition of the same code structure in more than one place in the program. A refactoring will present itself in finding a way to unify the structures. Having duplicated code can become a problem for long-lived programs as any changes to the logic of that code must be performed everywhere it is present. It is also necessary to ensure that everything was updated consistently.

   Fixing: A duplicated code problem can consist of the same expression in two methods of the same class, this problem can be solved by applying Extract Function and then invoking the code from both places. Similar code (and not identical) may be solved by using Slide Statements to arrange the code in a way that similar items are all together for extraction.

   OutSystems context: This pattern can also be identified in OutSystems Logic. The Visual Language allows for building logic flows that are comprised of nodes, such as actions and data-related elements. Equivalent flows can be found in OutSystems' applications as explained in Section 4.4.2.

2. **Long Function** proves to be a problem because the longer a function is, the more difficult it is to understand it. When broken down into small functions it is important to give them good names so that there is almost no need to check the function's body to understand it.

   Fixing: Most of the time, it is simply needed to use Extract Function to shorten the function, by finding parts of the function that make sense together. If the function has many temporary variables, it is useful to use Replace Temp With Query to avoid passing as parameters. Long lists of parameters can be reduced using *Introduce Parameter Object* [12] and *Preserve Whole Object* [12].

   OutSystems context: The equivalent to this pattern has been identified by Dijkink as a *Long Undocumented Flow* [46] and affects the maintainability of applications [49].

3. **Global Data** is a problem that leads to many bugs because this data can be modified from anywhere in the code base and there's no mechanism to find the code that changed it. The solution passes by wrapping this global data by a function and trying to see where it is modified and controlling its access. Finally, the scope of the

data should be limited as much as possible by moving it within a class or module where only that module's code can see it.

Fixing: When confronted with this problem, the first step is to use *Encapsulate Variable* [12], so that by having it wrapped by a function, it is possible to visualize where it is modified to control its access. The final step is to limit the scope of the variable as much as possible by moving it to a class or module.

OutSystems context: *Public Entities aren't read-only* is a pattern identified in the OutSystems environment that affects the architecture of applications. This pattern allows for operations to be performed by any consumer on entity actions 2.3, which may lead to bugs as any consumer may perform inconsistent and even destructive changes [46]. It strongly relates to the code smell above, as both concern the lack of control over information, when it is out of limited scope.

4. **Large Class** consists of a class, with a lot of fields, that probably is doing a lot of computation that can be split into smaller units. If a class does not use all its fields all of the time, choosing a useful subset of these fields may lead to a separate class. As for a class with too much code, the problem with duplicated code may eventually rise. The solution is to eliminate redundancy in the class itself by finding the code in common and sharing it in only one place.

   Fixing: A Large Class problem can be solved with *Extract Class* [12] by grouping a set of variables that are logically intertwined.

   OutSystems context: The *Monolithic mobile UI module* pattern and the *Monolithic Service Module* pattern affect the architecture of Outsystems' applications [46]. These patterns are the counterpart to the code smell above when applied to the OutSystems code (as OutSystems is not an Object-Oriented programming language). This relation has been identified by Dijkink [49].

4.2.8 Refactoring Mechanisms

In this section, we present some commonly used refactoring mechanisms which are worthwhile to name and describe in the context of this thesis. These mechanisms are based on Fowler's catalog of refactorings [1, 12].

4.2.8.1 Extract Function

Motivation: Some guidelines for extracting code are based on its length or the possibility of code reuse. For Fowler, the biggest motivation appears when it is necessary to provide clarification on what the code is doing. This clarification can be given by extracting the code and naming it after its purpose.

Mechanics: 1. Copy the code to be extracted from the source function into the new target function; 2. Scan the extracted code for references to any local variables that will

not be in scope for the extracted function and pass them as parameters. If a variable is only used inside the extracted code but is declared outside, move the declaration into the extracted code; 3. Scan the extracted code for variables that are assigned. If it is only one, treat the extracted code as a query and assign the result to the variable concerned. Otherwise, it might be better to abandon the procedure and consider Replace Temp With Query or *Split Variable* [12] before revisiting the extraction; 4. Compile; 5. Replace the extracted code in the source function with a call to the target function; 6. Test.

Example: Listings I.1 and I.2 are Javascript code examples of the before and after execution of three Extract Function operations. Extracting the function *recordDueDate* requires passing the *invoice* variable as a parameter because it will be out of scope. Extracting the function *calculateOwing* also requires *invoice* to be passed as a parameter. Additionally, this function has to return the result of the computation which is assigned to the *owing* variable in the original function. This variable was also turned into a constant because it will not be reassigned. Lastly, the *printDetails* function requires both variables, *invoice* and *owing*, to be passed as parameters.

4.2.8.2 Slide Statements

Motivation: Code is easier to understand when pieces of code that are related to each other are shown together. If there are several lines of code accessing the same data structure, they should appear rather than mixed with unrelated code.

Mechanics: 1. Identify the target position to move the fragment to. Proceed if changing the statements will not change the program's behavior. 2. Cut the fragment from the source and paste it into the target position; 3. Test.

Note: Regarding the Duplicated Code Smell and the detection of near-duplicates, applying this mechanism to the code might easily lead to side effects. Changing the order of operations in code to achieve similarity can change the behavior of the code. The Slide Statements operation is an arduous task, especially when prompted to be resolved and validated with automated refactoring mechanisms.

4.2.8.3 Replace Temp With Query

Motivation: When splitting a large function, turning variables into their own functions makes it easier to extract parts of the function without needing to pass variables as parameters. Using functions instead of variables also contributes to avoiding duplication of the calculation logic in similar functions, which can then be extracted and reused.

Mechanics: 1. Verify that the variable is entirely determined before it is used and that the code that calculates it does not yield a different value when it is used. Make the variable read-only if possible. 2. Test. 3. Extract the assignment of the variable into a function. 4. Ensure that the extracted function is free of side effects. Otherwise, use *Separate Query from Modifier* [12]. 5. Test Again. 6. Use *Inline Variable* [12] to remove the temporary variable.

4.3 Graphs

In this section, we describe the necessary concepts for a better understanding of the problems involving graphs that will be presented throughout this work.

4.3.1 Definitions

A **graph** $G = (V, E)$ consists of a set of elements V called **vertices** and a set of elements E called **edges**. In this work, we use the same notation for graphs as in the work by Terra-Neves *et al.* [6]. Each edge joins two vertices and is denoted by specifying its two vertices, e.g. (u, v). In a graph, two or more edges joining the same pair of vertices are **multiple edges** and an edge joining a vertex to itself is a **loop** [14]. A **simple graph** is a graph with no multiple edges or loops. G is a **finite graph** if the sets V and E are both finite, and a **null graph** if they are both null. We use the symbol $\emptyset$ to denote a null graph [15].

The vertices v and w of a graph are **adjacent vertices** if they are joined by an edge e [14]. The vertices v and w are defined as **incident** with the edge e, and the edge e is incident with the vertices v and w. The **degree** of a vertex v is the number of edges incident with v. Each loop is counted twice. A graph is **completely determined** when its vertices and edges are known. Two graphs are the same if they have the same vertices and edges. Two graphs G and H are **isomorphic** if H can be obtained by relabelling the vertices of G, i.e. if there is a one-one correspondence between the vertices of G and those of H, such that the number of edges joining each pair of vertices in G is equal to the number of edges joining the corresponding pair of vertices in H. A **sub-graph** S of a graph G is a graph all of whose vertices are vertices of G and all of whose edges are edges of G.

A **walk** of length k in a graph is a succession of k edges of the form $uv, vw, wx, \ldots, yz$ and is referred to as a walk between u and z. It is not required for all the edges or vertices in a walk to be different. A **path** is a walk in which all the edges and vertices are different. A graph is **connected** if there is a path between each pair of vertices. Every **disconnected** graph can be split up into a number of connected sub-graphs, named **components**. A **closed walk** in a graph is a succession of edges of the form $uv, vw, wx, \ldots, zu$ that starts and ends at the same vertex u. A **cycle** is a closed walk in which all the edges are different and all the intermediate vertices are different.

4.3.2 Digraphs

A **digraph** $G = (V, E)$ consists of a set of elements called V vertices and a set of elements E called arcs [14]. Arcs are edges with a distinguishable origin and destination, each **arc** joins two vertices in a specified direction and the arc is denoted by specifying its two vertices in order, e.g. in a digraph (u, v) differs from (v, u).

Two or more arcs joining the same pair of vertices in the same direction are **multiple arcs** and an arc joining a vertex to itself is a **loop**. Just like in an undirected graph, a

digraph with no multiple arcs or loops is a **simple digraph**. In a digraph, the **out-degree** of a vertex v is the number of arcs incident from v and the **in-degree** of v is the number of arcs incident to v.

The digraph analogs of **adjacency**, **incidence**, and **isomorphism** are similar to the corresponding definitions for graphs, except that we take account the direction of the arcs. Similarly, the terms **walk**, **path** and **cycle** also apply to digraphs.

A digraph is **connected** if its underlying graph is a connected graph otherwise, it is disconnected. A digraph is **strongly connected** if there is a path between each pair of vertices. Similarly, every disconnected graph can be split up into a number of connected sub-graphs, named **components**.

4.4 Duplicated Code

An important aspect of improving design is to eliminate duplicated code. Reducing the amount of code by eliminating its' repetition, does not make the program run faster but it makes a big difference when modifying the code. The more code there is to understand, the harder it is to modify it correctly. By eliminating duplication we ensure that the code is not repetitive resulting in better readability and maintainability, which is the essence of good design [12].

Software clones are often introduced due to copying code that already solves a similar problem somewhere in the code base [16] and adjusting it to the current context. The term "clone" is often used to identify duplicated code [17]. However, this "shortcut" to reusing an existing design carries the risk of reducing code readability, maintainability [16] and promoting bug propagation [5].

Removing duplication associated with the clones results in a single copy of the originally duplicated sections of code, which leads to increasing modularity [5]. Furthermore, this activity limits the possibility of carrying bugs to the repetitive code snippets, which contributes to code robustness. In conclusion, avoiding repetition is a concept that leads to good design [18].

4.4.1 Code Clone Types

Code fragments can be similar due to the similarity of their program text or due to their functionalities [19]. Based on the kind of similarity between clones, they can be classified into different types.

Textual Similarity clones can be classified as follows:

- **Type I.** The code fragments are identical except for whitespace, layout, and comments.
- **Type II.** The code fragments structure/ syntax is identical except for variations in identifiers, literals, types, layout, and comments.

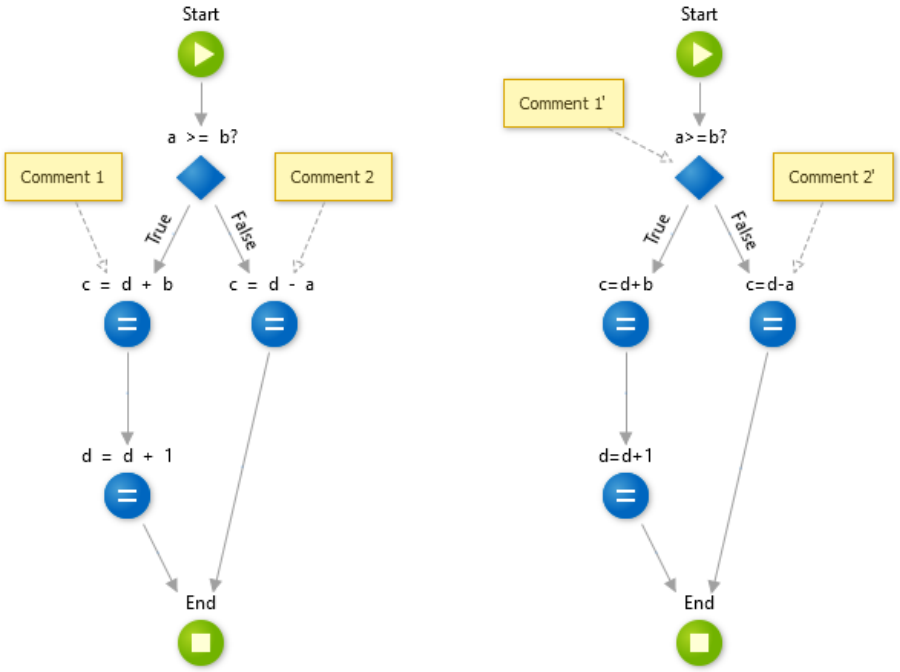


Figure 4.1: Type I Duplicate Code Fragments in OutSystems

- **Type III.** The code fragments are identical but can have additional/ missing statements in addition to variations in identifiers, literals, types, layout, and comments.

Functional Similarity clones occur if the functionalities of the two code fragments are identical or similar, leading to the following classification:

- **Type IV.** The code fragments perform the same computation but implemented through different syntactic variants.

The analytical complexity and sophistication in detecting clones increases from Type I through Type IV and is not related to the automation of this process, we will get into detail about each type of clone with examples of duplicated flows in OutSystems. In this analysis, the examples for duplicated code fragments in A Survey on Software Clone Detection Research [19] were translated/ amended into OutSystems code.

4.4.1.1 Type I

Type I clones are also known as **exact clones** and are often due to "copy-paste" code. The copied code fragment is the same as the original except for some possible variations in whitespace (blanks, newlines, tabs, etc.), comments, and/or layouts. Figure 4.1 shows an example of Type I duplicated flows in OutSystems. These two flows are textually similar, as each element has an exact correspondence in the other flow, after removing the whitespace and comments.

Type I clones may also include **changes in layout** for languages where curly brackets ("{" and "}") can be positioned to enclose groups of statements and define the scope. Thus,

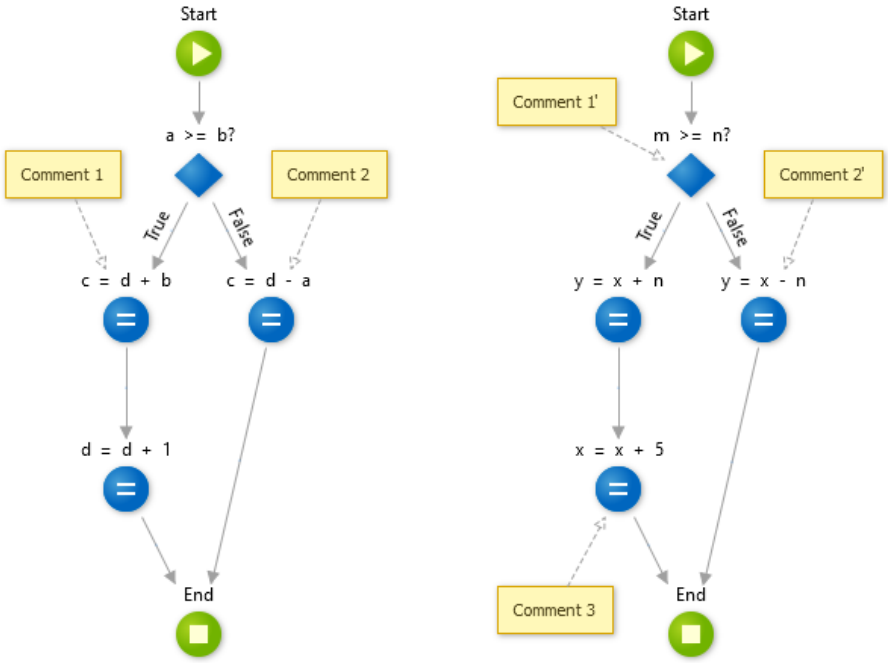


Figure 4.2: Type II Duplicate Code Fragments in OutSystems

a typical "line-by-line" technique may fail to detect such clones that vary in layout. This is not a problem in OutSystems due to the inherent structure of its' visual language. There can be changes in the placement of elements throughout the canvas but these do not affect duplicated code detection.

4.4.1.2 Type II

A Type II clone is a code fragment that is the same as the original except for some possible variations in the corresponding names of identifiers (name of variables, constants, class, methods, etc.), types, layout, and comments. Figure 4.2 shows an example of Type II duplicated flows in OutSystems. Although the two flows change a lot in variable names and value assignments, the syntactic structure is still similar in both flows as the elements' structure is essentially the same in the duplicated flows.

4.4.1.3 Type III

In Type III clones, statements of the copied fragment can be changed, added, and/or deleted. Figure 4.3 shows an example of Type III duplicated flows in OutSystems. The two flows are textually similar but in addition to differences in comments, assignments, and variable names, one extra statement was inserted in the flow. Without the added assignment ("e = 1"), these duplicated flows would be considered Type II clones.

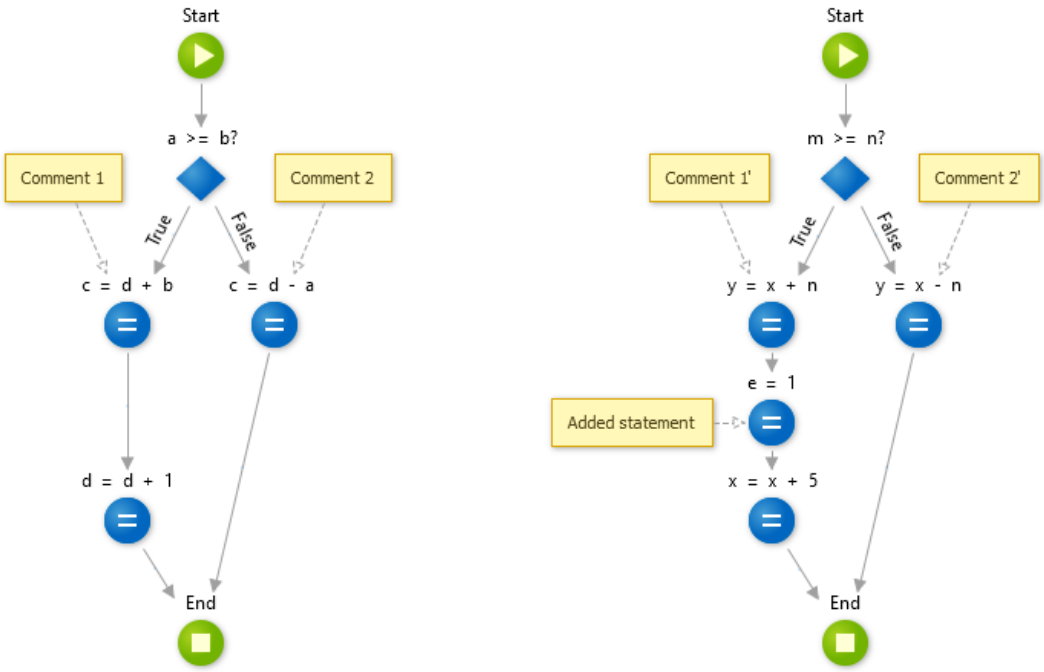


Figure 4.3: Type III Duplicate Code Fragments in OutSystems

4.4.1.4 Type IV

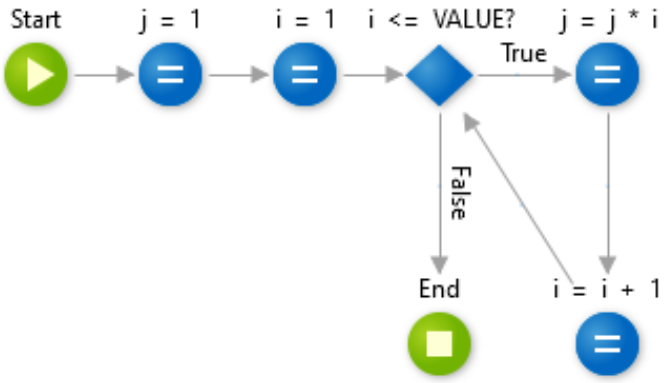
Type IV clones result from the **semantic similarity** between code fragments. In this type of clone, the cloned fragment is not necessarily copied from the original. An example of when this type of clone could emerge is when two different programmers implement the same kind of logic and the code fragments end up similar in their functionality but not necessarily textually similar.

Functional similarity reflects the degree to which the fragments act alike, i.e., captures similar functional properties, and similarity assessment methods rely on matching of pre/post-conditions. If the clones have similar pre and post conditions they are considered semantic clones.

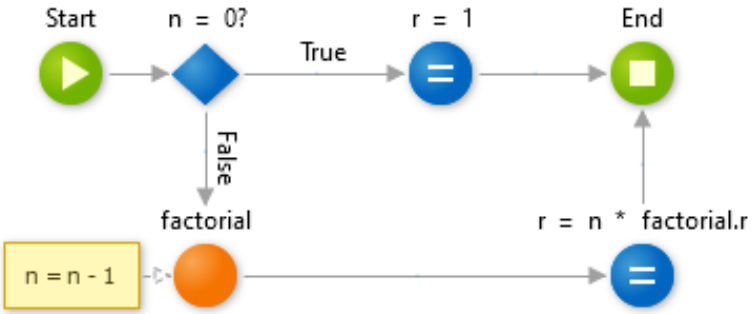
Figure 4.4 shows an example of Type IV duplicated flows in OutSystems. In the first flow (Figure 4.4a), the final value of the output parameter "j" is the result of the factorial value of the input parameter "VALUE" and is computed iteratively. However, in the second flow (Figure 4.4b) the factorial value of input parameter "n" is computed recursively and the final value is stored in output parameter "r". From the semantics point of view, both flows are similar in their functionality and considered Type IV semantic clones even though there are no lexical/ syntactic/ structural similarities between the elements of the duplicated flows.

4.4.2 Detection in Visual Programming Languages

Duplicated Code affects the maintainability and readability of evolving software. It also comes with the risk of propagating bugs since that a certain bug will be present in every



(a) Iterative Factorial Flow



(b) Recursive Factorial Flow

Figure 4.4: Type IV Duplicate Code Fragments in OutSystems

copy made of a given code snippet. Having such a strong implication on good software design, Terra-Neves *et al.* developed a **duplicated code detector** that takes leverages on the visual facet of SAPs to highlight the duplicated code patterns [6].

The functional requirements for this detector included that the duplicated structure needed to be visually highlighted to the user, so that the duplicates were easier to analyze and understand, and thus the duplicated code detector returns the mappings of flow nodes to the duplicated code pattern nodes. This duplicated code detector also needed to be highly efficient and scalable, optimizing operation costs, as it had to be integrated into Architecture Dashboard and perform static analyses for hundreds of OutSystems code bases every 12 hours.

The proposed duplicated code detector for OutSystems addresses these issues by iteratively mining **Maximum Common Sub-graphs** (MCS) of graph representations of OutSystems code. In OutSystems, logic is implemented through logic flows. The approach uses **MaxSAT encoding** [20] for finding a single maximal duplicated code pattern and the pattern mining algorithm uses a lazy greedy approach with further optimizations for achieving the same results in less time.

4.4.2.1 Maximum Common Sub-graph

Logic flows are represented as graphs, thus a duplicated code pattern is a **common sub-graph** that occurs across multiple flows. The largest common pattern in those flows corresponds to an Maximum Common Sub-graph (MCS). Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be a pair of graphs with labeled nodes/edges. In this work, it is assumed that graphs are directed. Next, we describe the notation used to define an MCS [6]:

- $L(v)$ denotes the label of some node v
- V_i^l denotes the subset of nodes $v \in V_i$ such that $L(v) = l$
- $L(u, v)$ denotes the label of some edge (u, v)
- E_i^l the subset of edges $(u, v) \in E_i$ such that $L(u, v) = l$
- $L_{comb}(u, v)$ is equal to $(L(u), L(u, v), L(v))$ and denotes the combined label of (u, v)
- $E_i^{comb/l}$ denotes the subset of edges $(u, v) \in E_i$ such that $L_{comb}(u, v) = l$
- $L_{comb}(E_i)$ is used to denote the set of combined labels that occur in E_i

A graph $G_C = (V_C, E_C)$ is a **common sub-graph** of G_1 and G_2 if there exist mappings $f_1 : V_C \to V_1$ and $f_2 : V_C \to V_2$ such that $L(v) = L(f_1(v)) = L(f_2(v))$ for all $v \in V_C$ and $L(u, v) = L(f_1(u), f_1(v)) = L(f_2(u), f_2(v))$ for all $(u, v) \in E_C$. G_C is said to be an MCS if and only if no common sub-graph $G'_C = (V'_C, E'_C)$ of G_1 and G_1 exists containing more nodes or edges than G_C, i.e. such that $|V'_C| > |V_C|$ or $|E'_C| > |E_C|$. Given a node $v \in V_i$, $v \in V_C$ is used to denote that there exists $v' \in V_C$ such that v' is mapped to v, i.e. $f_i(v') = v$. Similarly, given $(u, v) \in E_i$, $(u, v) \in E_C$ is used to denote that there exists $(u', v') \in E_C$ such that $f_i(u') = u$ and $f_i(v') = v$.

4.4.2.2 MaxSAT Encoding

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be a pair of graphs with labeled nodes and edges. The **MaxSAT formulation** extracts an MCS by mapping the nodes of graph G_2 into the nodes of the graph to compare G_1. To specify these constraints, Terra-Neves *et al.* used node and edge labels and parameterized them with the node/edge with their types as well as the necessary attributes to match the duplicates. In this approach, the labels were tuned with the goal of maximizing the detection of Type III Duplicates that share the same graph structure. The inclusion of isolated nodes is forbidden because such nodes are not desirable for the duplicated code pattern mining use case. The following constraints apply [6]:

- **Inclusion clauses:** A node $v \in V_1$ is in the MCS if and only if at least one node in V_2 is mapped to v.
- **One-to-one clauses:** At most one node in V_2 can be mapped to each node $v \in V_1$.

- **Function property clauses:** Each node $v' \in V_2$ cannot be mapped to more than one node in V_1.
- **Label consistency clauses:** A node $v' \in V_2$ cannot be mapped to $v \in V_1$ if v and v' do not share the same label.
- **Control-flow consistency clauses:** Consider some edge $(u, v) \in E_1$ and a pair of nodes $u', v' \in V_2$. If u' and v' are mapped to u and v respectively, and (u', v') is not an edge of G_2 or does not share the same label as (u, v), then (u, v) cannot be in the MCS.
- **No spurious edge clauses:** An edge $(u, v) \in E_1$ can be part of the MCS only if both u and v are as well.
- **No isolated node clauses:** A node $v \in V_1$ can be part of the MCS only if at least one of its incoming/outgoing edges is in the MCS.

4.4.2.3 Refactor Weight

For mining duplicated code patterns the algorithm of picks pairs of graphs with the highest priority, according to some custom priority function [6]. For the duplicated code use case, the priority function is based on the notion of **refactor weight** of a graph. The algorithm receives as input a minimum refactor weight threshold β and discards graphs with a refactor weight lower than β.

Given a graph $G = (V, E)$, each node $v \in V$ has an associated refactor weight w_v, that depends on its type and on the kind of operations it performs. For all nodes, except Instruction nodes that correspond to database accesses, a refactor weight of 1 is considered. The weight of the remaining nodes is given by the respective number of database tables, filter conditions, and sort conditions. For all edges $(u, v) \in E$, a refactor weight of 1 is also considered.

Let $G_{W1} = (E_{W1}, VW1), G_{W2} = (V_{W2}, E_{W2}), \ldots, G_{Wp} = (V_{Wp}, V_{Wp})$ denote the p **weakly connected components** of G. A weakly connected component G_{Wi} is a **maximal subgraph** of G such that, for all node pairs $u, v \in V_{Wi}$, v is reachable from u in the undirected counterpart of G. The refactor weight w_G of G is given by the maximum weight across G's components. The weight of each component is given by the sum of its nodes and edges refactor weight.

5

Related Work

The development of automated refactoring techniques is the goal of this work. This chapter provides an overview of multiple solutions and tools that are already implemented for automated refactoring. Baqais and Alshayeb [21] conducted a thorough literature review of papers that suggest, propose, or implement an automated refactoring process and classified them according to several quality measures. This work was carefully analyzed during the elaboration of this thesis. We have categorized our research according to the method followed in each study to either propose the detection of refactoring opportunities or the correction of problems using refactoring techniques. The most relevant findings are presented and discussed in the following sections.

5.1 Clone Management

Many authors have analyzed and studied the problem of having software clones in the code base. These are code snippets that are strongly similar to each other and are equivalent to Duplicated Code, the bad smell captured by Fowler [1].

Research by Wang and Godfrey is based on the observation that even with the current availability of tools that detect code clones with high accuracy and scalability, managing the clones remains challenging. Regarding clone management, fixing the clones involves applying refactoring techniques, namely merging the code snippets into one, while preserving the original functionality. This process reduces the risk of introducing or propagating bugs as well as, improve the maintainability of the code and readability. In **Recommending Clones for Refactoring Using Design, Context, and History** [16] the authors propose an automated approach to recommend clones for refactoring taking into account benefits, costs, and risks and by training a decision tree-based classifier. In this work, the *iClones* tool is used for clone detection, which is a token based-clone detector. The detection includes clones with minor differences in identifiers, literals and types, white spaces, and gaps among clone fragments. Before refactoring a developer considers the costs and risks of the operations to be performed. The developer may also weigh the benefits of performing the refactoring. In this work, clone refactoring recommendations

is considered a classification problem. A **supervised machine learning** approach is used to process features of clones and to capture the essence of the developers' analysis. Therefore, a **decision tree-based classifier** is trained to learn from features of both refactored and "unrefactored" clone instances found in clone evolution history. The dimensions for feature selection are the following: **1.** the cloning relationship (e.g. whether clone fragments are located in the same file); **2.** the cloned code snippets (e.g. proportion of method invocation statements of a cloned code snippet); **3.** the context of cloned code snippets (e.g. lines of code of methods that contain the cloned code snippets).

Regarding the identification and choice of the refactoring clones, the authors generate candidate sets of clones based on a combination of metrics associated with refactoring and **manual inspection** is required to filter out clones that are not refactoring instances (false positives). This work presents an interesting solution because due to the amount of duplicated code found in OutSystems applications, and even though the code detector can discard duplicates with a refactor weight lower than a given threshold, it might be necessary to further analyze the refactoring costs and benefits before refactoring code clones so that the duplicated snippets are proven worthwhile to refactor.

Tairas and Gray developed ***CeDAR*** (Clone Detection, Analysis, and Refactoring) [22] an Eclipse [52] plug-in for Java that incorporates the results of various clone detection tools and displays the clone properties within the IDE. With this tool, it is possible to perform simultaneous refactoring of the clones to remove the duplicated code associated, instead of performing multiple refactoring steps separately on the same clone group. The improvements in this work consist in the information about clones from the detection phase serving as input to the refactoring engine as well as *CeDAR* evaluating clone properties that are provided to the user (such as the parameterized elements of the clones) when a group of clones is selected to be refactored. This information is useful to determine which identifiers need to be passed to the new method if an Extract Method refactoring was performed.

In **Increasing clone maintenance support by unifying clone detection and refactoring activities** [5] the authors have identified the gap between the process of clone detection and refactoring, this is also the current situation at OutSystems, and this thesis addresses this issue. The process of removing duplicated code can be considered in three phases: detection, analysis, and refactoring [5]. Tairas and Gray utilize *CeDAR* [22] to streamline the clone maintenance process, allowing for programmers to delegate the code structure changes to a refactoring engine, which reduces the number of errors that may occur when compared to changing the code manually. Figure 5.1a depicts the process of using the Extract Method refactoring within Eclipse. The Abstract syntax tree (AST) of the selected code is determined and, using subtree matching on the AST of the class where the code is located, matching sub-trees are identified. In this scenario, the programmer does not know before selecting the code to refactor if there are clones of the specified code. Figure 5.1b incorporates the results from a clone detection tool that are displayed and accessible within the IDE. The *CeDAR* plugin displays the location of

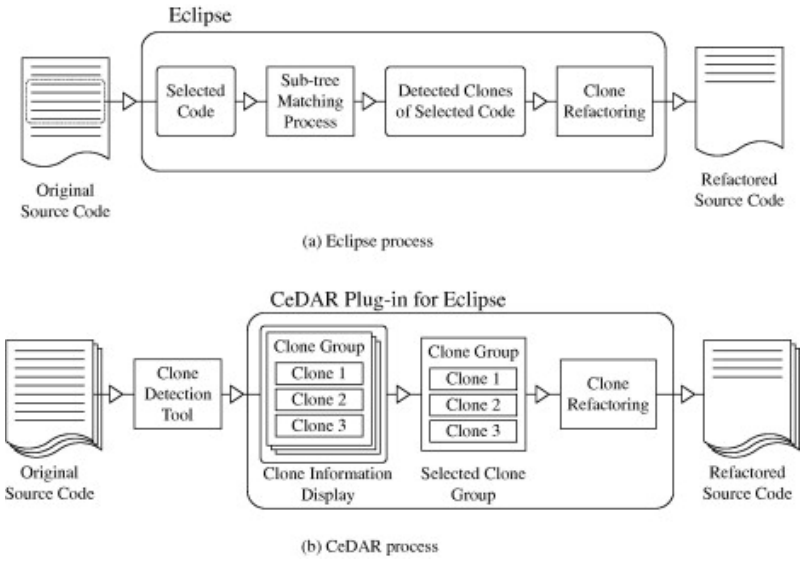


Figure 5.1: Clone maintenance processes in [5]

the clones and allows connection directly to the source code associated. Therefore, the developer acknowledges the clones before initiating the refactoring. Additionally, the parameterized differences between the clones are represented.

Although the streamline of the clone maintenance process is desired and relevant for this thesis, in [5], the developers still have to instrumentalize the refactoring and not only decide on refactoring the presented clones. As previously mentioned, this cannot be the solution for OutSystems as the goal of this work is to automate the process so that OutSystems developers do not have to make the changes for resolving technical debt issues caused by anti-patterns.

5.2 Search-based Refactoring

The need for an automatic refactoring approach is also present when it comes to databases. Related work on this subject includes the ***MIGRATOR*** [23] prototype tool for automatically migrating database programs to a new schema by Wang *et al.*. The necessary evolution of database applications leads to schema refactorings, which involve changes to the database schema, intended to improve the design and/or performance of the application without changing its semantics. This task is nontrivial and error-prone because changes to the database schema often require re-implementing parts of the database program to make the program logic consistent with the underlying schema. The database transactions in the original program P are then performed over a schema S so given the new schema S' it is necessary to generate a new version of the program P' that operates over S' and so that P and P' are semantically equivalent. The methodology for automatically **Synthesizing Database Programs for Schema Refactoring** is subdivided into three tasks identified and briefly described as follows:

- **Value Correspondence Generation** consists of lazily enumerating possible value correspondences (VC) between the source and target schemas (S and S' respectively) in decreasing order of likelihood using a partial weighted MaxSAT encoding (MaxSAT is a generalization of the boolean satisfiability problem and aims to determine the maximum number of clauses that can be satisfied by an assignment of truth values [20]).

- **Sketch Generation** is the procedure to generate a sketch Ω that represents all programs that may be equivalent to P under a given value correspondence ϕ. This phase starts with *join correspondence* done by mapping each join chain used in P to a set of possible join chains over the target schema. A join correspondence (J , J') is valid with respect to ϕ if ϕ can map all attributes used in J to attributes in J'.

- **Sketch Solver** performs a symbolic search (using SAT) over the space of programs encoded by the sketch Ω and then subsequently check for equivalence. If the two programs are not equivalent, minimum failing inputs are employed to minimize the search space. This is achieved by identifying programs that are incorrect for the same reasons as P' which in practice involves checking for function invocations where the query results in P differ from those of P'. **Sketch Completion** is achieved if given a sketch Ω and the original program P, it is possible to find an instantiation P' of Ω such that P' is equivalent to P.

This work proves to be relevant in the context of this thesis as program synthesis is a technique that can generally be applied for finding programs that satisfy a given specification. The symbolic search performed is especially interesting because of the possibility of minimizing the search space by discarding programs where the results after refactoring are not the same. Furthermore, we can consider the target program as the specification for the search phase, making this specification implicit. Raychev *et al.* presented a work [24] that uses program syntheses on the refactoring domain and is in compliance with the assessment made above.

In **Improving multi-objective code-smells correction using development history** [25] the authors use the development history collected from existing software projects to propose refactorings taking into account the similarity with past situations. The general idea is to maximize and encourage the use of new refactorings that are similar to those applied to different software projects in similar contexts. Ouni *et al.* propose a multi-objective optimization-based approach to find meaningful sequences of refactoring. The fitness function comprises the following objectives: **1.** minimize the number of code-smells; **2.** maximize the use of development history and **3.** preserve the construct semantics (how code elements are semantically grouped and connected).

The non-dominated sorting genetic algorithm (NSGA-II) is used to find the best trade-offs between the goals mentioned above. The basic idea of **NSGA-II** is to make a population of candidate solutions evolve toward the near-optimal solution to solve a

multi-objective optimization problem [25]. A non-dominated solution provides a suitable compromise between all the objectives mentioned above, without degrading any of them.

The search-based process takes as inputs: **1.** the source code, **2.** code smells detection rules, **3.** a set of refactoring operations, and **4.** a call graph for the whole program. A solution consists of a sequence of refactoring operations that should be applied to improve the quality of the input code. In their approach, they have considered the following code-smells [25]:

- **Blob:** One large class that monopolizes the behavior of a system, while other classes primarily encapsulate data.
- **Dataclass:** A class that contains only data and performs no processing on the data.
- **Spaghetti code:** Code with a complex and tangled control structure.
- **Functional decomposition:** A class designed to perform a single function.

The performance of this refactoring approach is evaluated by accessing if it could generate meaningful sequences of refactorings that fix code smells while preserving the construct semantics of the original design and reusing as much as possible the development history applied to similar contexts.

The related work mentioned above is used to introduce the use of genetic algorithms (GAs), as these are the most commonly used algorithms for automatic refactoring found in the study by Baqais and Alshayeb [21]. The solution presented by Ouni *et al.* is an interesting approach to find sequences of refactoring steps. Nonetheless, instruction on how to implement the automatization of these steps was not presented.

5.3 Other Methods

In **Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures** [26], Bavota *et al.* propose an *Extract Class* refactoring method, a technique to split a class with many responsibilities into different classes, based on graph theory and that exploits structural and semantic relationships between methods. This approach takes a class, identified by a software engineer, as a candidate for refactoring, which is parsed to extract a weighted graph representation. A **Max-Flow Min-Cut** algorithm is used to obtain a partition of the original graph (class) in two subsets of nodes (methods). The partition is obtained by cutting the minimum number of edges with a low weight. The process results in two sub-graphs that can be used to build two new classes that have cohesion higher than the original class and without increasing too much coupling. The attributes of the original class are distributed among the extracted classes according to how they are used by methods in the new classes. The selected **graph algorithm** allows splitting a class into two classes while ensuring that the number of

dependencies between the two extracted classes is low (because of the min-cut). The operations are likely performed on classes with low cohesion, splitting into two classes helps increase the cohesion of the original class and produces a low increment of coupling.

Bavota *et al.* defined a **semi-automatic approach**, as the extracted classes are analyzed by developers who can accept the proposed refactoring or change it by moving methods and attributes from one class to another. The developer can analyze the cohesion of the two new classes and possibly re-apply the operation. This is an interesting solution for allowing governance over automated refactorings in OutSystems as well as displaying the changes in technical debt that result in resolving an anti-pattern automatically.

Ganea *et al.* [27] developed ***INCODE***, an Eclipse [52] **plugin** for continuous quality assessment and code inspections of Java systems. *INCODE* also assists developers in performing refactoring operations. It provides concrete refactoring advice on how to solve a particular problem or how an instance can be corrected, by taking into account the entire context of dependencies of a class/method. Furthermore, if the plugin detects that in the given context, a predefined Eclipse refactoring, or a composed restructuring (defined by the plugin) can be applied, the suggested code transformation can be **triggered directly** from the *INCODE* interface and performed by the IDE. This work manifests a step forward for minimizing the gap between the detection of anti-patterns and their resolution with automated refactoring techniques.

5.4 Final Discussion

In this chapter different solutions for automated refactoring were presented and analyzed to understand the different approaches in the current state of the art. With the emergence of new methods to automate the software refactoring process to reduce the time and effort required for refactoring [21], automated refactoring proves to be a field at the vanguard. In our studies, we have observed an increasing interest in applying search-based methods for detecting refactoring opportunities [28, 29]; for finding the operations needed to achieve a desired refactored program [23], and the for suggestion of refactoring opportunities [25, 30]. Several solutions to aid in clone management have also been addressed in the literature. We have found that most state-of-the-art solutions tend to exemplarily describe the detection of automated refactoring opportunities but lack in giving clear solutions on how to solve the detections in an automated way. Although there are already several tools and plugins integrated into IDEs (such as Eclipse [52]), to the best of our knowledge, the use of automated refactoring techniques in the context of SAPs is not yet explored.

6

Technical Approach

In this chapter, we present context information about the solution delivered by this dissertation as well as our approach for tackling the problem identified in Problem Statement. We start by presenting the architecture for our automated refactoring tool. Then, we present the set of anti-patterns that were selected to be tackled in this work. Next, we present the execution pipeline of the proposed solution. Lastly, we describe the representation of the programs that we will refactor.

6.1 Solution Architecture

As explored in the previous chapters, the OutSystems refactoring process comprises inefficient and challenging tasks. This process begins when a developer observes a warning from Architecture Dashboard of an anti-pattern that has been found in the code-base and suggests that the developer solves this finding by hand. This process becomes inefficient when we consider the number of findings that Architecture Dashboard detects daily. In the OutSystems context, there is no tool or mechanism to guide and ease the refactoring process. Taking this into account, the proposed solution is going to cover the automated refactoring process in OutSystems.

The solution expected as a result of this dissertation automatically refactors a set of selected high-impact anti-patterns that contribute to high technical debt. For this purpose, we developed a prototype tool with automated refactoring techniques for OutSystems logic. The logical architecture of the proposed solution is illustrated in Figure 6.1. The solution is based on a unidirectional transformation of the source program given as input of the prototype tool as well as the detection mechanisms to find the anti-patterns in those flows. The prototype tool has two components, the detection of subcases within the findings given by the detection component and the Automated Refactoring component that executes the refactorings. The Subcase detection component operates on the results of the Detection component that detects anti-patterns in the source program, whilst the Automated Refactoring component operates on snippets of the program that match subcases of the detected anti-patterns. The Target Program is the result of this process and

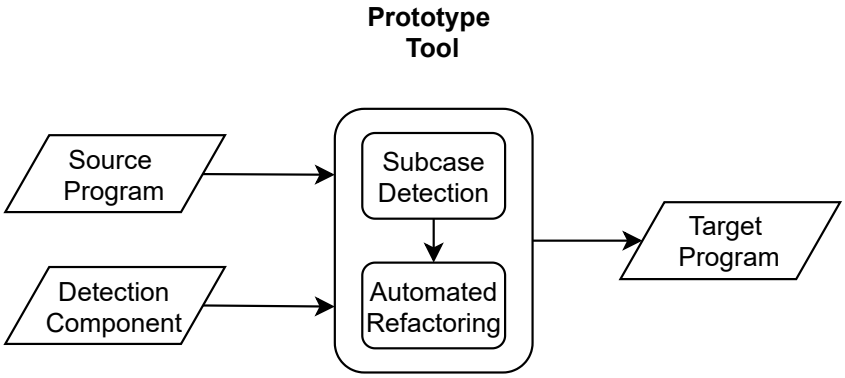


Figure 6.1: Logical Solution Architecture

is a program where the anti-patterns have been refactored. In the thesis context, the Source Program comprises the logic flows where the detection component will look for anti-patterns and the Target Program comprises the refactored OutSystems logic flows. The solution architecture proposed is designed to be generic as possible and not dependent on a specific programming language. This way, in the future it will be easy to extend with the refactoring of other anti-patterns and generalize for other technologies.

6.2 Anti-Pattern Selection

Considering the high technical debt problem and its identification by the OutSystems' technical debt monitoring tool and that this work is being developed in an industrial environment. It was important to take into consideration all of the project's stakeholder's insights on the existing inefficiencies and their suggestions considering the real-world problems related to the subject. Table 6.1 displays the top ten patterns with the most findings in Architecture Dashboard according to their respective categories as of 10/01/2021. An internal analysis, conducted at OutSystems on a large set of real-world code bases, has observed that the amount of duplication is at least 0.7% in the analyzed benchmarks and reaches as high as 39% in the worst case [6].

After studying and analyzing the complete set of anti-patterns detected by Architecture Dashboard [46] and with the aforementioned stakeholder's experience to better understand their cause, we aimed to establish which patterns would have the most significant impact and were more urgent to solve. Taking all the above into consideration, the selected patterns for analysis are the following: 1. Unlimited Records in Aggregate and 2. Duplicated Code.

6.3 Proposed Solution

In the first phase, and for each anti-pattern, we defined rules for syntactic transformations that solve the patterns correctly. Next, we refactor the patterns according to these

Table 6.1: Top ten patterns with the most findings in Architecture Dashboard

Category	**Pattern**	**Findings**
Maintainability	Missing description on public element	3,122,423
Performance	Unlimited records in Aggregate	1,373,897
	Inline CSS style	1,057,730
	Large Local Variable in ViewState	599,299
Security	Avoid Anonymous/Registered access Screens	540,455
Maintainability	Long undocumented flow	405,665
Performance	Long Server Requests Timeout	248,827
	Unlimited records in SQL query	221,281
	Image widgets without width	154,987
	Query data in ViewState	131,204

rules. As for the case of Duplicated Code, performing the Extract to Action operation in OutSystems (equivalent to Extract Function) and reusing the newly created action solves the problem of Type I Duplicates. Lastly, we evaluate each refactored pattern according to the number of detections/ refactorings performed and to the elapsed time of the algorithms.

The proof of concept consists of an offline tool that detects and refactors the anti-patterns currently present in a snapshot of OutSystems code bases. The execution process of the proposed solution is illustrated in Figure 6.2. The execution pipeline starts with the input program being analyzed, followed by the detection of the anti-patterns currently integrated, either by running algorithms developed by us or from previous work that is taken as input of the prototype tool (see Figure 6.1). Then, if the anti-patterns are found, we check if they match any subcase that we have identified within this work, if so the input program is automatically refactored in the next step and is presented at the result of our process.

In the future, the tool could be integrated into Architecture Dashboard receiving as input the already detected anti-patterns and automatically refactoring them. It may be interesting to provide the developer with a warning link to the code and a button that would perform the automated refactoring. Moreover, the developer would have the possibility to visualize the changes made to the code base before accepting them.

6.4 Logic Flows

In OutSystems, logic is implemented through logic flows. These flows can correspond to Client Actions, Server Actions, Screen Actions, Preparation Flows, etc. A Preparation Flow is a flow that runs during the loading of a screen to retrieve the data that is going to be displayed in it. In our work we use the same definition/notation for logic flows as in previous work by Terra-Neves *et al.* [6].

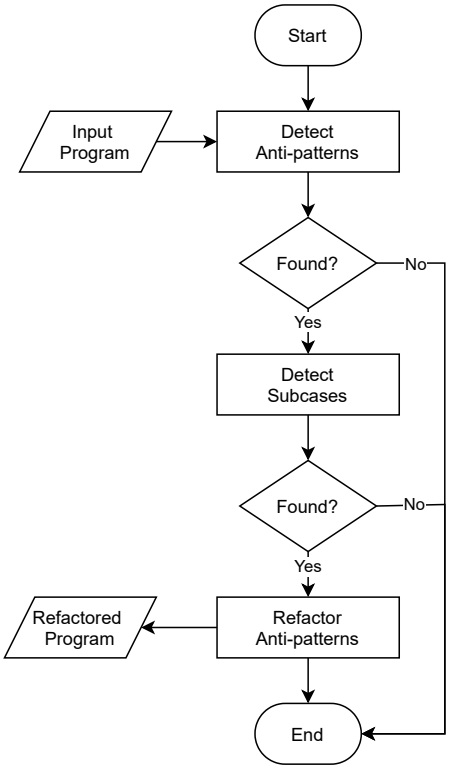


Figure 6.2: Prototype Tool Execution Process

A **logic flow** is a directed weakly connected graph $G = (V, E)$ where the type of each node in V is one of the following: Start, End, Instruction, ForEach, If, or Switch. The type of each edge in E is one of the following: Connector, True, False, Cycle, Condition, or Otherwise. We refer to the outgoing edges of a node as branches. G satisfies the following properties:

- G does not contain self-loops or parallel edges.
- V contains only one Start node v and no edge $(u', v') \in E$ exists such that $v = v'$, i.e. no branch exists in E for v.
- Given an End node $v \in V$, no branch exists in E for v and there exists at least one edge $(u', v') \in E$ such that $v = v'$, i.e. there exists at least one edge in E for v.
- A Start or Instruction node $u \in V$ has exactly one Connector branch $(u, v) \in E$.
- An If node $u \in V$ has exactly one True branch $(u, v) \in E$ and one False branch $(u, v') \in E$.
- A ForEach node $u \in V$ has exactly one Connector branch $(u, v) \in E$ and one Cycle branch $(u', v') \in E$ such that there exists a path from u to itself through (u', v').
- A Switch node $u \in V$ has at least one Condition branch $(u, v) \in E$ and exactly one Otherwise branch $(u', v') \in E$.

The logic flow is similar to the control flow graph of a program written in a traditional programming language. The execution of the logic flow starts in its Start node and ends in one of its End nodes. Logic flows can have input and output parameters. The nodes/edges can have different attributes besides their types, as follows:

- An **If node** contains a Boolean expression to be evaluated in execution. If the expression is evaluated to true it continues through its True branch otherwise, it continues through its False branch. An If node can be used to create cycles over the value of the given Boolean expression, as there can exist more than one branch towards the node (e.g. when the expression of an If node $v \in V$ is evaluated to true and as it continues through its True branch to another node $u \in V$, if there exists $(u, v') \in E$ such that $v = v'$, there exists a cycle around the If node v while its expression evaluates to true).

- A **ForEach node** contains reference to a variable of an iterable type (e.g. list), and optionally, the start index to iterate through that list and the maximum number of iterations to be performed. While the variable can be iterated (i.e. the last position is not reached or the maximum number of iterations, if given, is not surpassed) the execution continues through its Cycle branch, and this condition is evaluated again when the path from the node to itself circles black. When the variable can no longer be iterated, it continues through its Connector branch.

- A Condition branch of a **Switch node** contains a Boolean expression to be evaluated in execution. If the expression is evaluated to true, then the execution continues through that branch otherwise, it continues through the evaluation of the next branch. Condition branches have a pre-specified order of evaluation. If none of the branches of the Switch node evaluates to true, the execution continues through its Otherwise branch.

Instruction nodes can be of various kinds, such as Assign nodes for variable assignments, Aggregate or SQL nodes for database accesses, Execution nodes for calls to other logic flows, *etc.* These nodes have different attributes according to their types, some of the relevant Instruction nodes in our analysis are defined as follows:

- An **Assign node** can be used to assign values to as many variables as required. For each of these assignments, it contains an expression for the variable to be assigned and an expression for its value.

- An **Aggregate node** contains reference to its sources (the Entities), the joins it performs over the sources, the expressions for the sorts it performs, the expressions for the filters it applies to the data (e.g. select only columns where the value is above x), the expressions for attributes it calculates (e.g. repeat the attribute of a column but multiplied by x), the expressions for the attributes being grouped by (e.g. group by the id of table store), the expressions for attributes it aggregates (e.g. given the

group by of the id of the store, sum the revenue of the store) and the expression for its Max Records value (if given).

- An **Execution Action node** contains reference to the action to be executed (the logic flow) and to the arguments required to execute that action. Each argument contains a reference to the expression to use as value and a reference to the input parameter of the action that its value will fill.

The kind of expressions in the logic flow that can be evaluated in If nodes, attributed in Assign nodes, *etc.*, are the following:

- **Basic**, for an expression with a value of a basic type (e.g. Boolean, Text, Integer, Date, *etc*).
- **Operation**, unary if performed over only one expression (e.g. the negation of a Basic Boolean expression) or binary if performed over two expressions (e.g. the sum of two Basic Integer expressions).
- **Identifier**, for an expression that refers an attribute of a node/ structure (e.g. the input parameter of an Execution node).
- **Compound Identifier**, for an expression that refers: 1) another node/ structure in the flow and 2) an Identifier that specifies an attribute of that node/ structure (e.g. an Execution node and an Identifier for one of its input parameters, or an Aggregate node and an Identifier for its Count runtime property).
- **Call**, for an expression that is the result of a function (e.g. the concatenation of two Text expressions).

In our analysis, we consider two types of graph representations for flows: **logic flow** and **extended flow**. A logic flow in OutSystems contains only the nodes and edges that are visible in the Logic layer of the OutSystems' IDE.

An **extended flow** is a directed weakly connected graph $G' = (V', E')$ that contains additional nodes and edges for connecting the nodes in V to their attributes and expressions. We use the extended flow representation of the graph to find paths between the nodes in V that emerge from accessing/modifying the same attributes, that here are represented by nodes/edges. Since the types for each node in V' and each edge in E' belong to an extensive set, we will define the relevant types as needed, throughout the explanation of the detection/refactoring of the anti-patterns.

7

Unlimited Records Anti-pattern

In this chapter, we start by defining the Unlimited Records Anti-pattern and providing further detail on the Aggregate's structure in OutSystems. Then, we present our approach for detecting and solving two subcases of the Unlimited Records anti-pattern: the Filter by Id Subcase and the Count Subcase. Lastly, the results of an experimental evaluation showing the merits of the proposed techniques are explained followed by a discussion of the limitations to our approach.

7.1 Definition

Unlimited Records is an anti-pattern found in OutSystems applications which occurs whenever queries are unbounded, i.e. the maximum number of records to be fetched is not explicitly limited, resulting in the return of all records and contributing to higher technical debt. When defining an Aggregate in OutSystems, it is possible to limit the maximum number of records to be returned by the Aggregate by defining the **Max Records** property. If there are limitations to the number of records that are fetched by a query, it is considered to be a good practice to define the Max Records property accordingly, to optimize the query execution time [53]. It is also possible to visualize the Executed SQL property which contains the SQL statement produced from the Aggregate. This query is changed to match the given definitions, including the limitation imposed by defining the Max Records property.

One of the anti-patterns returned by the Architecture Dashboard tool is the Unlimited Records anti-pattern. These findings can easily escalate and become time-consuming to fix, as each application usually has several Aggregates (737,443 Aggregates in total across 45,405 modules). Furthermore, there are scenarios where the Max Records limit was not defined because the query returns only one record by construction (e.g. get by primary key). These are still flagged by the Architecture Dashboard but could be easily fixed just by setting the Max Records property to one. The profusion of false positives in this analysis hinders the handling of truly problematic scenarios. Such scenarios should be solved automatically so that the developer can focus on the less trivial scenarios and that

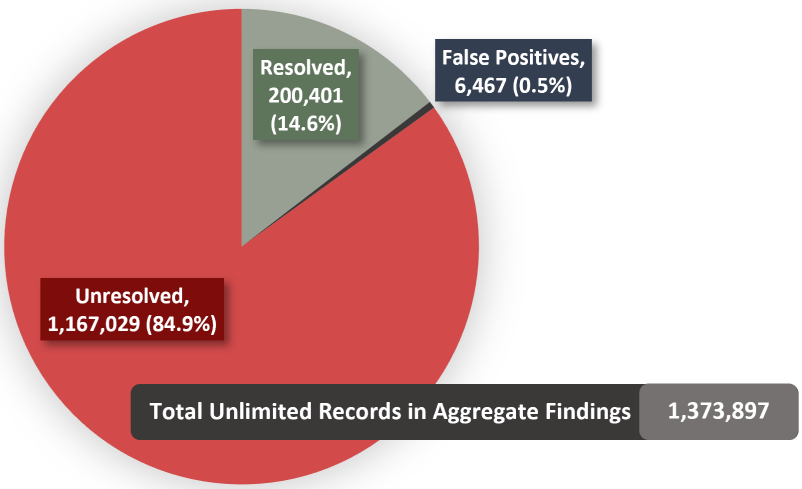


Figure 7.1: Unlimited Records In Aggregate Pie Chart

actually require their attention. Thus, we focus on the identification of such spurious warnings and automatically fixing some straightforward occurrences, thus contributing to a more effective software evolution process.

The total number of real-world findings reported in the OutSystems analysis tools, as of January 2021, consists of +1,3M unbounded queries as displayed in Figure 7.1. The state-of-the-art solution reports and recommends that developers solve these findings by hand. However, up to this day, developers only solved 15% (≈ 200K) of the reports.

7.2 Aggregate's Structure

The Aggregate's structure was proposed in previous work by Seco *et al.* [4] which introduced the CombineSources node to allow the use of nested collections and the orchestration of several remotely located data sources. This work is also the base for the OutSystems data layer, allowing for gradual construction of queries and with immediate feedback to the developers. Figure 7.2 illustrates the Aggregate's structure. Considering the extended representation of the graph, an Aggregate can be directly connected to a Source (if it only has one source) or to a GroupBy node (if it performs any group bys/ has any aggregated attributes) or to a CombineSources node (if it performs any joins or filters, or contains any calculated attributes). The Max Records value is set if there is an edge from the Aggregate to an Expression, whose type is MaxRecords. We detect the Unlimited Records anti-pattern by evaluating the preceding condition.

Aggregates have a **Count runtime property** that specifies the total number of records that match the query criteria. Changing the Max Records value does not change this property, as it consists of an extra query for counting all the records. Usually, there is no

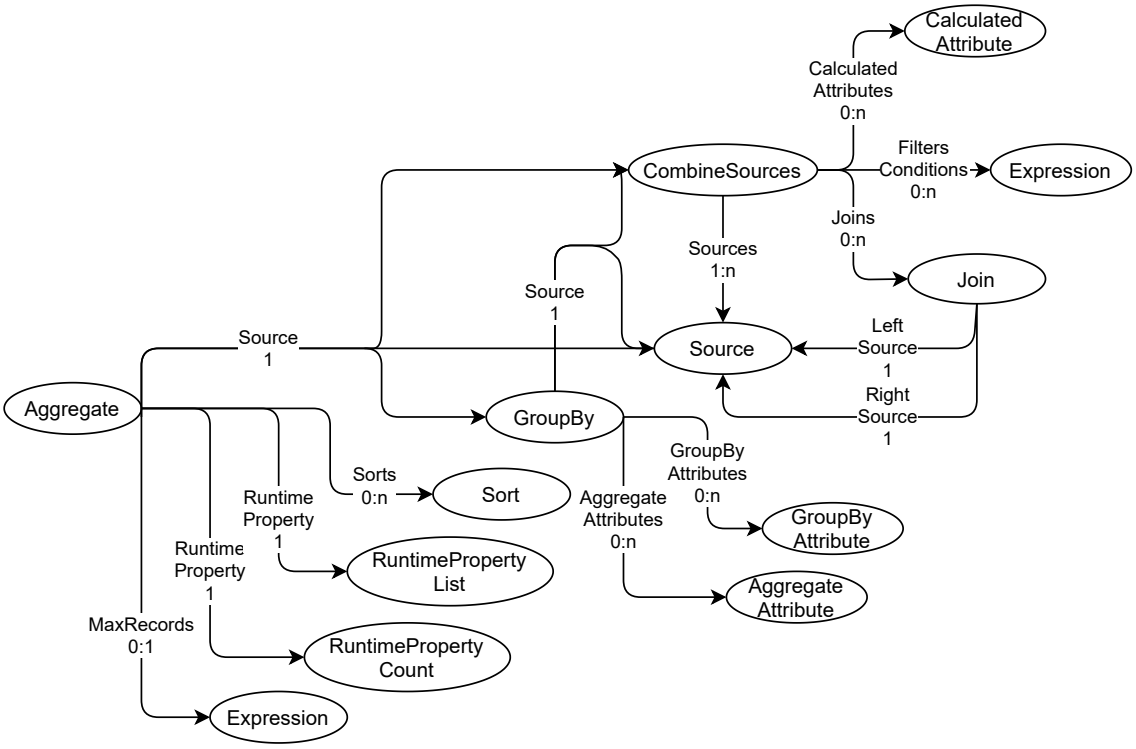


Figure 7.2: Graph structure example of an Aggregate

need to display all the records on a single screen, and thus it is not necessary to fetch all of these tuples from the database. The tuples that are fetched can be accessed by referencing the Aggregate's **List runtime property**. SQL knowledge is not required to use Aggregates, hence concerns such as query time optimization, strongly correlated to the amount of data fetched from the database, are usually overlooked by OutSystems users. Limiting the number of records retrieved, when possible, will improve the screen loading time, making applications faster and more responsive, which is a top priority for OutSystems users.

7.3 Filter By Id Subcase

When defining an Aggregate in OutSystems, the developer can define: sources (tables) for the data to be fetched, joins between sources, filters that are equivalent to where conditions in SQL, etc. The Filter By Id subcase consists of an Aggregate that has one or more filters, and at least one of them is an equality condition involving the primary key of one of the sources. Considering that an Aggregate has sources that correspond to relational databases and that each source (table) has a primary key and may have foreign keys to other sources (tables). If we were to define a SQL query as shown in Query 1), with a `SELECT` clause on only one table `R` and with a `WHERE` clause checking if the primary key `id` is equal to some value `c`. Then, at most one record would satisfy the where condition otherwise, the attribute would not be a primary key. We can extend this rule to a query

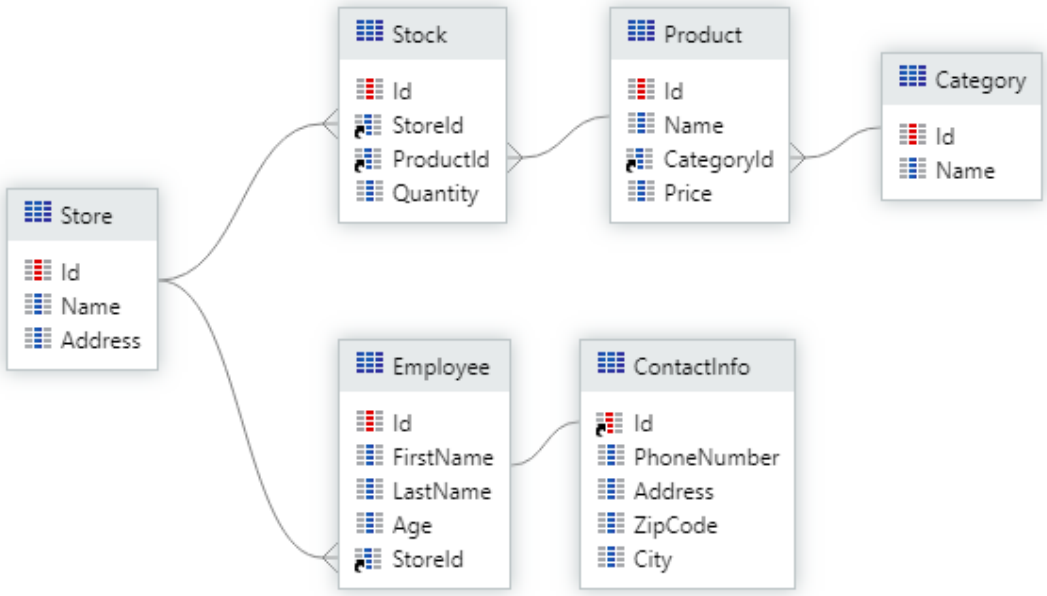


Figure 7.3: Example of an ERD representing a simplified Store

selecting more than one table, as long as enough filters are applied to ensure that at most one record would satisfy the query.

Query 1) `SELECT * FROM R WHERE R.id = c;`

The primary identification of this subcase came from one of the project's stakeholder's insights, that besides already having identified the Unlimited Records anti-pattern as a predominant anti-pattern, also distinguished the most common case that was being flagged, it being the Filter By Id Subcase. In our analysis, we observed that the Filter By Id subcase is fairly frequent in practice, contributing to 28% of the occurrences of the Unlimited Records anti-pattern. The goal of this analysis is to be able to limit the Max Records property, by setting it to one and therefore solving these straightforward occurrences.

This section presents our approach for solving the problem of finding Aggregates that match the Filter By Id subcase of the Unlimited Records anti-pattern among a given set of aggregates $A_1, A_2, ..., A_n$ and how to fix them. We start by presenting a concrete example of Filter By Id subcase and the sound rules following Relational Databases query size rules that make an Aggregate fall into this subcase in Section 7.3.1, followed by an example in SQL and OutSystems and a description of the detection algorithm. In Section 7.3.2 we present the solution and the refactoring algorithm for the Filter By Id subcase.

Figure 7.4: Filter By Id Service Studio

7.3.1 Detection

For detecting the Filter By Id Subcase we consider the extended representation of the flow to check if the expressions for the Aggregate's filters reference the primary keys of the sources and to analyze the data model, namely the relations between the Aggregate's sources. In this chapter, when giving examples of each subcase we will refer to the data model displayed in the Entity Relationship Diagram (ERD) in Figure 7.3 representing a simplified Store. Figure 7.4 illustrates a logic flow in OutSystems for getting the name of the category of a given product. To verify the necessary conditions, we look at the representation in Figure 7.5 that shows a simplified example of the extended flow representation of the same logic flow, used for assigning to a local variable ProductCategory, the name of the Category that a given Product belongs to. Considering two database tables Product and Category and that Product has an attribute CategoryId, i.e. the foreign key reference to the primary key of Category, then each Product has only one Category. If the Aggregate definition contains an inner join over tables Product and Category and the join condition is the equality of the foreign key CategoryId with Id, the primary key of the Category. Then each Product will match exactly one Category and the size of the results after the join would be the same size as the Product table. If we further define a filter that checks if Id, the primary key of Product, is equal to some local variable ProductId, the local variable will match with at most one Product otherwise, it would not be a primary key. When assigning the name of the Category to the local variable ProductCategory, the List property of the Aggregate would have at most one row matching the Product with the Id in the local variable ProductId, together with the information of its Category. This

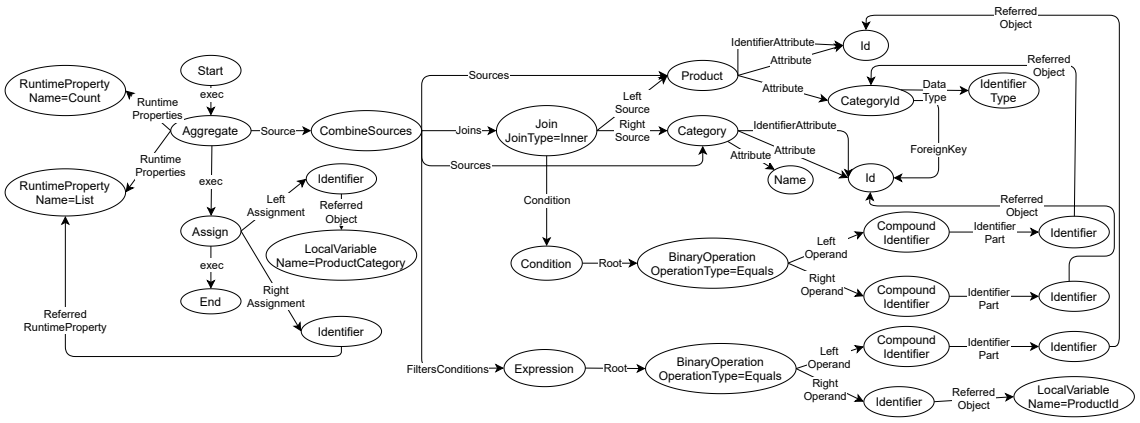


Figure 7.5: Graph structure example of an Aggregate used to filter by primary key

example matches the Filter By Id subcase of the Unlimited Records anti-pattern.

To detect that an Unlimited Records in Aggregated occurrence falls into the Filter By Id subcase, we start by defining sound rules that follow Relational Databases query size rules. Then, we propose an algorithm that evaluates these rules given a set of Aggregates and that, if these are respected, classifies them as Filter By Id subcase occurrences.

7.3.1.1 Rules

The following rules are considered for estimating the number of tuples resulting from a relational algebra expression over n relations:

- **One Relation.** Consider a relation R ($n = 1$), if a filter by id is performed over the primary key a of R, then the query will return at most a single record. Having c as a constant, and since there exist T(R) distinct values for a in R, the estimate for the number of tuples in the query result is:

$$T(\sigma_{R.a=c}(R)) = \frac{T(R)}{V(R,a)} = \frac{T(R)}{T(R)} = 1 \tag{7.1}$$

- **N Relations.** Consider relations $R_1, R_2, \ldots, R_n$ ($n > 1$), joined in pairs using any type of join ($\bowtie$, $\rtimes$, $\ltimes$, $\overline{\ltimes}$ and $\times$) and that the same relation may be joined more than once. In the case of theta-joins, the join condition can be combined with any other condition $C_1, C_2, \ldots, C_n$, as long as it uses the conjunction operator ($\wedge$), as follows:

$$R_1 \bowtie_{C1 \wedge C_2 \wedge \cdots \wedge C_n} (R_2)$$

  If we can estimate that every result relation E_i being joined will have at most a single record, and because in the worst-case estimate the size of intermediate relations are multiplied (see Section 4.1.5.2), the estimate for the number of tuples in the query result is:

$$T(E_1 \bowtie_\theta E_2 \bowtie_\theta \ldots \bowtie_\theta E_n) = 1 \times 1 \times \cdots \times 1 = 1 \tag{7.2}$$

For every join performed between two relations one of the following rules need to be satisfied:

- **One-To-One Relationship.** Consider two relations R and S such that the primary key a of R is both a foreign key and the primary key of S. Given any join, except the cross join, over a, due to a being both the primary key of R and of S and a foreign key in S (one-to-one relationship), $T(S) = T(R)$ and the number of tuples returned by the join is:

$$T(R \bowtie_{R.a=S.a} (S)) = T(S) = T(R)$$

  If a filter by id is performed over the primary key of any of the tables R or S, then the query will return at most a single record. Having c as a constant, and since there exist T(R) distinct values for a in R, the estimate for the number of tuples in the query result is:

$$T(\sigma_{R.a=c}(R \bowtie_{R.a=S.a} (S))) = \frac{T(R)}{V(R,a)} = \frac{T(R)}{T(R)} = 1 \tag{7.3}$$

- **One-To-Many Relationship.** Consider two relations R and S such that the primary key a of R is a foreign key of S (one-to-many relationship) and b is the primary key of S. Given any join, except the cross join, over a, due to a being the primary key of R and a foreign key in S, the size of the join is:

$$T(R \bowtie_{R.a=S.a} (S)) = T(S)$$

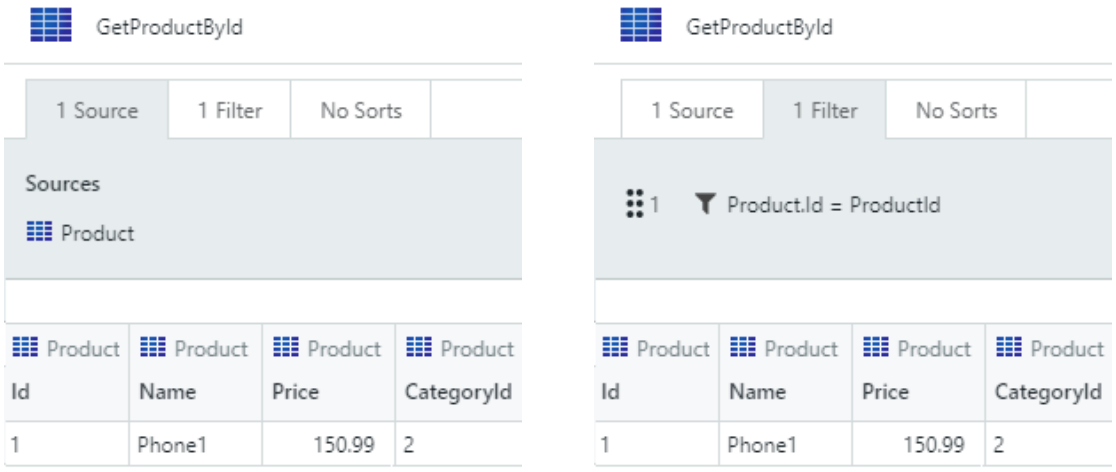
  If a filter by id is performed over b, then the query will return at most a single record. Having c as a constant, and since there exist T(S) distinct values for b in S, the estimate for the number of tuples in the query result is:

$$T(\sigma_{S.b=c}(R \bowtie_{R.a=S.a} (S))) = \frac{T(S)}{V(S,b)} = \frac{T(S)}{T(S)} = 1 \tag{7.4}$$

- **Other relationships.** Consider two relations R and S such that a is the primary key of R, and b is the primary keys of S. Given any of the following cases: 1) R and S are joined with a cross join ($\times$) or 2) R and S are joined with any join, except the cross join, and the join condition does not relate a primary key with a foreign key or 3) R and S have no foreign keys to each other, if a filter by id is performed over the primary keys of both tables being joined, then the query will return a single record. Given a cross join between R and S, the query size estimation for the cross join operation would be:

$$T(R \times S) = T(R) \times T(S)$$

  If two filters by id are performed, over a and over b, then the query will return at most a single record. Having c and d as constants, and since there exist T(R)



(a) Product Source

(b) Filter By Id over Product

Figure 7.6: Filter By Id on one Source Example in OutSystems

distinct values for a in R and $T(S)$ distinct values for b in S. The estimate for the number of tuples in the query result is:

$$T(\sigma_{R.s=c \wedge S.b=d}(R \times S)) = \frac{T(R)}{V(R,s)} \times \frac{T(S)}{V(S,b)} = \frac{T(R)}{T(R)} \times \frac{T(S)}{T(S)} = 1 \times 1 = 1 \tag{7.5}$$

- **Filters.** Consider m selections $F_1, F_2, \ldots, F_m$ ($m \geq 0$) over E a result relation of an expression that performs operations over n relations ($n > 0$). The number of tuples returned by the expression will be at most the number of tuples in E as the selection operand only restricts the data (Section 4.1.5.1). Since we are considering arbitrary filters and not only filters on equality conditions, the estimate for the number of tuples in the query result is:

  $$T(\sigma_{F_1,F_2,\ldots,F_m}(E)) = T(E)$$

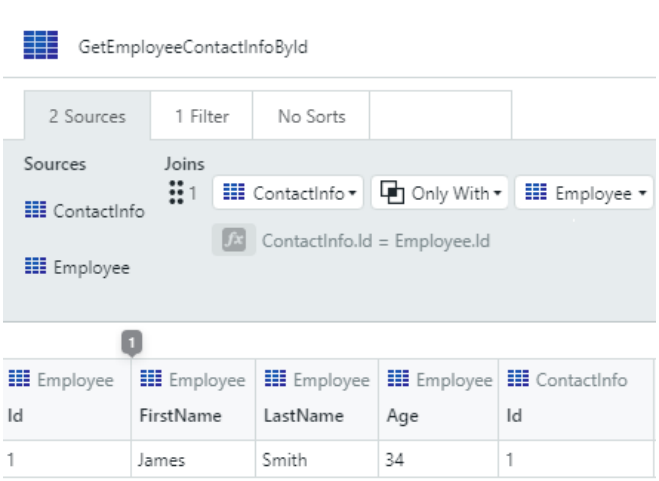
  If we can estimate that the number of records in E will be at most a single record, and because if the filters are combined through the conjunction operator ($\wedge$), then the query will return at most a single record.

  $$T(\sigma_{F_1,F_2,\ldots,F_m}(E)) = T(E) = 1 \tag{7.6}$$

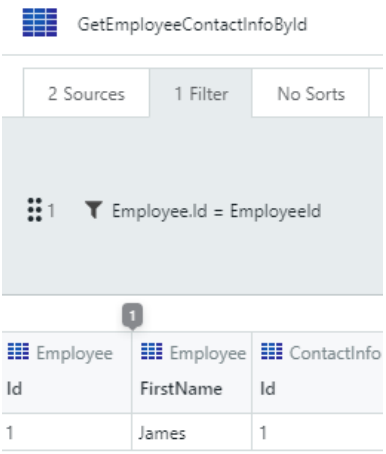
7.3.1.2 SQL and OutSystems Example

Considering each of the rules defined in Section 7.3.1.1 in Relational Algebra, we will now give concrete examples of scenarios in OutSystems and SQL where these rules apply.

Figure 7.6 illustrates an Aggregate in OutSystems for getting a product by its primary key ID. We can verify that only one source is being queried (Figure 7.6a), the PRODUCT table in the ERD (Figure 7.3), and in that the only filter being applied is a filter by id



(a) Join of Employee with ContactInfo



(b) Filter By Id over Employee or ContactInfo

Figure 7.7: Join of two Sources with a One-To-One Relationship Example in OutSystems

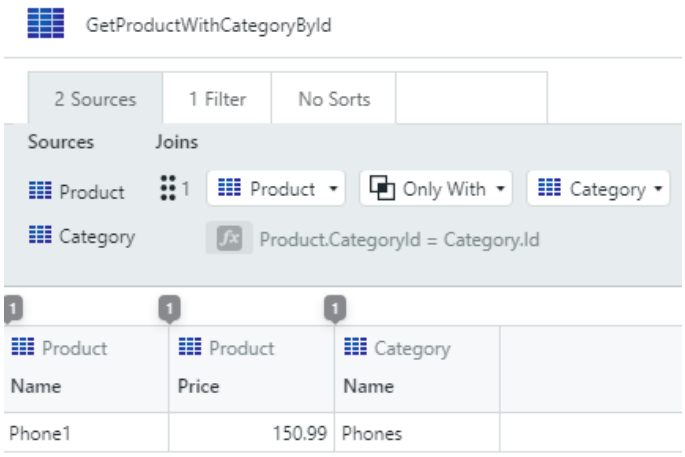
(Figure 7.6b), matching the ID of PRODUCT with a local variable PRODUCTID (Rule 7.1). Lastly, we can verify that only one row is returned for this definition. Query 2) matches this Aggregate's definition.

Query 2)
```
SELECT * FROM Product
WHERE Product.Id = ProductId;
```

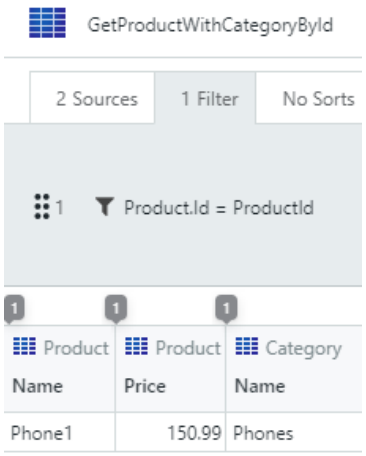
Figure 7.7 illustrates an Aggregate in OutSystems for getting the contact information of an employee. We can verify that two sources are being joined (Figure 7.7a), the CONTACTINFO table and the EMPLOYEE table. Additionally, the join condition matches the *id* of CONTACTINFO with the *id* of EMPLOYEE (Figure 7.3). These two tables share a one-to-one relationship, one row of table EMPLOYEE matches one row of table CONTACTINFO as they share the same primary key. Thus, when we perform a filter by id matching the ID of EMPLOYEE with a local variable EMPLOYEEID (Figure 7.7b), we can verify that only one row is returned (Rule 7.3). Query 3) matches this Aggregate's definition.

Query 3)
```
SELECT * FROM ContactInfo INNER JOIN Employee
ON ContactInfo.Id = Employee.Id
WHERE Employee.Id = EmployeeId;
```

Figure 7.8 illustrates an Aggregate in OutSystems for getting the category of a given product. We can verify that two sources are being joined (Figure 7.8a), the PRODUCT table and the CATEGORY table, additionally the join condition matches the *id* of PRODUCT with the *id* of CATEGORY (Figure 7.3). These two tables share a one-to-many relationship, since one row of table PRODUCT matches one row of table CATEGORY and one category can be matched by many products, the size of joining these two tables will be at most

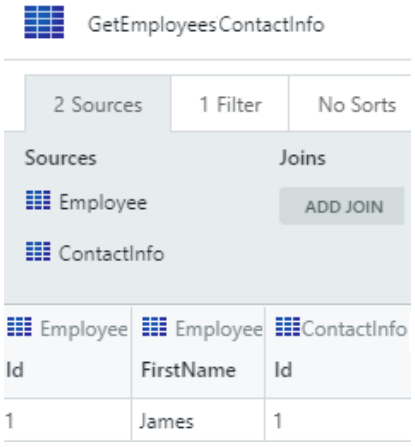


(a) Join of Product with Category

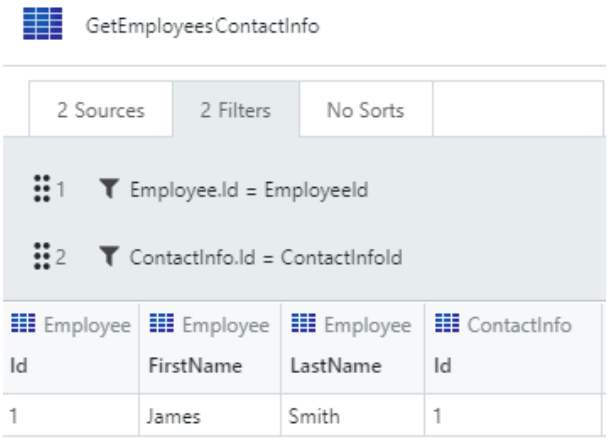


(b) Filter By Id over Product

Figure 7.8: Join of two Sources with a One-To-Many Relationship Example in OutSystems



(a) Cross Join of Employee with Contact Info



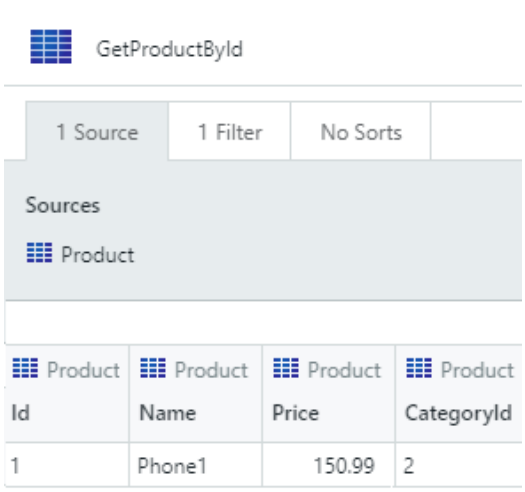
(b) Filters By Id over Employee and Contact Info

Figure 7.9: Cross Join of two Sources with two Filters By Id Example in OutSystems

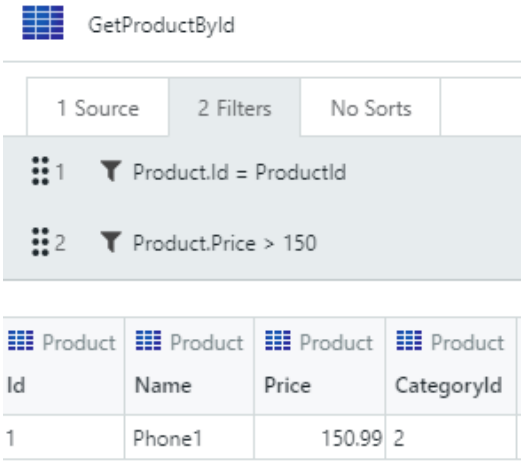
the size of table Product. Thus, when we perform a filter by id matching the id of Product with a local variable ProductId (Figure 7.8b), we can verify that only one row is returned (Rule 7.4). Query 4) matches this Aggregate's definition.

Query 4)
```
SELECT * FROM Product INNER JOIN Category
ON Product.CategoryId = Category.Id
WHERE Product.Id = ProductId;
```

Figure 7.9 illustrates an Aggregate in OutSystems for getting the contact information of an employee. We can verify that two sources are being joined with a cross join (Figure 7.9a), the Employee table and the ContactInfo table, as this is a cross join no join



(a) Product Source



(b) Two Filters: Filter By Product Id and Filter By Price

Figure 7.10: More than one Filter Example in OutSystems

condition is defined even though the tables share a primary key. Figure 7.9b shows how performing two filters by id, one matching the ID of EMPLOYEE with a local variable EMPLOYEEID and the other matching the ID of CONTACTINFO with a local variable CONTACTINFOID, restricts the data leading to a result of only one row (Rule 7.5). Query 5) matches this Aggregate's definition.

Query 5)
```
SELECT * FROM ContactInfo, Employee
WHERE Employee.Id = EmployeeId
AND ContactInfo.Id = ContactInfoId;
```

Figure 7.10 illustrates an Aggregate in OutSystems for getting a product by its primary key ID if its price is above 150. We can verify the source being queried (Figure 7.10a), the PRODUCT table, and that two filters are being applied (Figure 7.10b), namely: one filter by id, matching the ID of PRODUCT with a local variable PRODUCTID and a filter for selecting the product if its $Price > 150$ (Rule 7.6). Lastly, we can verify that only one row is returned for this definition. Query 6) matches this Aggregate.

Query 6)
```
SELECT * FROM Product
WHERE Product.Id = ProductId AND Product.Price > 150;
```

7.3.1.3 Algorithm

Algorithm 1 detects Aggregates used to filter by primary key. It receives as input a set of n Aggregate nodes $A_1, A_2, ..., A_n$ and returns a set R of Aggregate nodes that match the Filter By Id subcase. For the n Aggregate nodes to analyze, it checks if the number of

Algorithm 1: Detecting Aggregates used to filter by primary key algorithm.

Input: $A_1, A_2, \ldots, A_n$

```
1  R ← ∅;
2  for i ← 1 to n do
3      if |Filters(A_i)| > 0 then
4          if |Sources(A_i)| = 1 then
5              s ← Pop(Sources(A_i));
6              foreach f ∈ Filters(A_i) do
7                  if IsFilterById(f, s) then
8                      R ← R ∪ A_i;
9                      break
10         else
11             v ← True;
12             foreach j ∈ Joins(A_i) do
13                 vj ← False ;
14                 ls, rs ← LeftSource(j), RightSource(j);
15                 if ¬ IsCrossJoin(j) then
16                     if HasForeignKey(ls, rs) then
17                         vj ←IsSafeJoin(ls, rs, Condition(j), Filters(A_i));
18                     if ¬vj ∧ HasForeignKey(rs, ls) then
19                         vj ←IsSafeJoin(rs, ls, Condition(j), Filters(A_i));
20                 if ¬vj then
21                     l, r ← False, False;
22                     foreach f ∈ Filters(A_i) do
23                         if ¬l then
24                             l ← IsFilterById(f, ls)
25                         if ¬r then
26                             r ← IsFilterById(f, rs)
27                     vj ← l ∧ r;
28                 if ¬vj then
29                     v ← False;
30                     break
31             if v then
32                 R ← R ∪ A_i;
33 return R
```

filters is at least one (line 3) otherwise, there is no filter by id and this Aggregate does not match the subcase. If the Aggregate A_i only has one source (line 4), for each of the filters in A_i (line 6), it calls the auxiliary function IsFilterById (line 7). The auxiliary function IsFilterById receives as parameters a filter f and a source s, and returns true if f is a filter by id, i.e. it is an equality condition with the left operand being the primary key of s. If f contains a conjunction operator, it returns true if either the left operand or the right operand of the condition are filters by id, so the function IsFilterById is called recursively for every conjunction in the filter. If f is a filter by id, A_i matches the Filter By Id subcase because only one filter by id is needed for the one source (Rule 7.1)

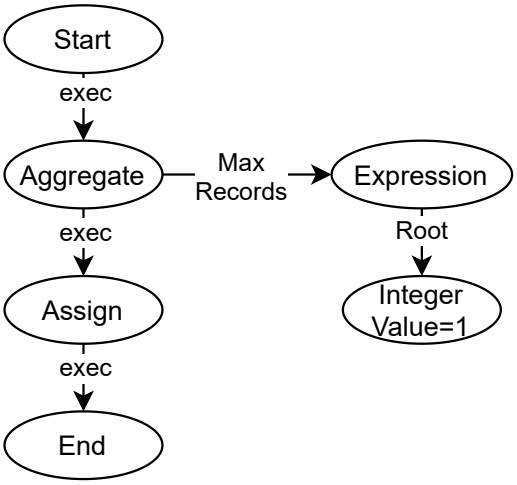
Function 1: IsSafeJoin(*sfk*, *spk*, *c*, *F*)

```
1  v ← false;
2  fk ← ForeignKey(sfk, spk);
3  if IsSafeCondition(c, PrimaryKey(spk), fk) then
4      if PrimaryKey(spk) = PrimaryKey(sfk) then
           // One-To-One
5          foreach f in F do
6              v ← IsFilterById(f, sfk) ∨ IsFilterById(f, spk);
7              if v then
8                  break
9      else
           // One-To-Many
10         foreach f in F do
11             v ← IsFilterById(f, sfk);
12             if v then
13                 break
14 return v;
```

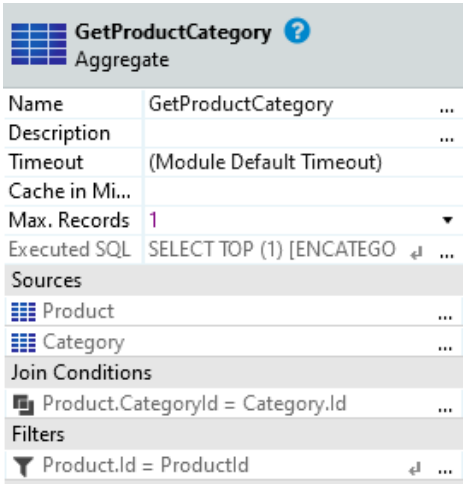
and separate filters in Aggregates are combined using conjunctions (Rule 7.6).

Otherwise, if A_i has more than one source, for each of the joins j defined in the Aggregate (line 12) it will check if j is a safe join, i.e. the Aggregate has the necessary filters by id (one or two depending on the join definition) for that join's result to be at most one record. If the join is not a cross join (line 15), it checks if the left source *ls* has a foreign key to the right source *rs* or the opposite, since if there is a foreign key, there is a possibility that this join only needs one filter by id and so it calls function IsSafeJoin (line 17).

The auxiliary function IsSafeJoin receives as parameters the source that has the foreign key *sfk*, the source with the primary key *spk*, the join condition *c* and the Aggregate's filters *F*, and returns a value *v* that is true if: 1) the foreign key *fk* is also the primary key of *sfk* and one filter by id is found on the primary key of source with the foreign key *sfk* or on the source with the primary key *spk* (Rule 7.3), or 2) one filter by id is found on the primary key of the source with the foreign key *sfk* (Rule 7.4). Initially, it checks if *c* is a safe join condition (line 3), i.e. it is an equality condition on the foreign key *fk* with the key of *spk*. Next, it checks if the foreign key *fk* is equal to the primary key of the source *sfk* (line 4), if so, the relationship between the sources is a one-to-one relationship as they share the same primary key, thus the filter by id can be performed on either one of the sources (Rule 7.3). Thus, it iterates through each filter f in F (line 5) and checks if f is a filter by id for any of the sources *sfk* or *spk* (line 6). If a filter by id is found, the iteration stops due to 1) only one filter by id is needed (Rule 7.3) and 2) separate filters are combined with the conjunction operator (Rule 7.1). Otherwise, if the foreign key *fk* is not the primary key of the source with the foreign key *sfk*, the relationship between the sources is a one-to-many relationship and thus the filter by id has to be performed on



(a) Graph structure example of an Aggregate used to filter by primary key after refactoring



(b) OutSystems example of an Aggregate with Max Records set to 1

Figure 7.11: Aggregate with Max Records set to 1

the "many" side, i.e. on the source with the foreign key (Rule 7.4). Similarly, it iterates through each filter f in F (line 10), but only checks if f is a filter by id for sfk (line 11), the source with the foreign key. If a filter by id is found, it stops the iteration due to: 1) only one filter by id is needed (Rule 7.4) and 2) separate filters are combined with the conjunction operator (Rule 7.6).

In Algorithm 1, if vj is false (line 20), i.e. the join is not valid, one of the following cases is present: 1) the join is a cross join or 2) there are no foreign keys relating the tables or 3) the join condition was not on the equality of the foreign key with the primary key of the sources, and thus the Aggregate needs two filters by id over the primary key of each source (Rule 7.5). Thus, it iterates through each filter f in A_i and, if the two filters have not been found, checks if f is a filter by id for either of the sources being joined ls or rs (lines 24 and 26). The join is valid if both filters are found (line 27). If the join is not valid, the iteration through the joins can stop as all the joins needed to be valid (Rule 7.2).

7.3.2 Refactoring

Automatically refactoring the Filter By Id subcase consists of setting the Max Records value to 1. This refactoring arguably preserves the behavior of the program as the soundness of the rules can be checked in a straightforward fashion (see Section 7.3.1). This process varies depending on the SAP logic. Figure 7.11a shows the graph representation of fixing the Filter By Id subcase by setting the Max Records to 1 in OutSystems following the example in Figure 7.5. As the rest of the graph remains unchanged, it is omitted for simplification purposes. Figure 7.11b shows the result of this refactoring in the OutSystems environment.

7.4 Count Subcase

Aggregates can be used to discover how many records match a certain query. The Count subcase occurs when the developer defines an Aggregate with as many sources, joins, and filters as necessary, but then only uses the Count property of the Aggregate, never accessing the resulting list returned by the query. To match the Count subcase this Aggregate can not be referenced in a Preparation Flow since these flows are used to load screen data and thus the Aggregate's results are implicitly accessed. Figure 7.12 illustrates a logic flow defined for getting the number of products that match a given category. During runtime, after executing a query to get all the Products that match this condition, when the variable NProducts is assigned the Count value of the Aggregate, another query will run to count these records. In this flow, the list returned from the Aggregate's definition is never referenced because we are only interested in finding out the number of products that match the category and not in reasoning about the products' data.

The identification of this subcase came from analyzing the structure/ attributes of the Aggregate as well as inspecting real-world code bases written in OutSystems and observing their logic flows. Although not as common, ≈1% of the occurrences of the Unlimited Records anti-pattern, this subcase proved to be relevant to refactor due to it leading to a visible improvement in the code's structure.

This section presents our approach for solving the problem of finding Aggregates that match the Count subcase of the Unlimited Records anti-pattern among a given set of aggregates $A_1, A_2, ..., A_n$ and how to fix them. We start by presenting an example of the Count subcase of the Unlimited Records anti-pattern and the respective algorithm for detecting Aggregates used to count the number of records that match the query in Section 7.4.1. Lastly, the automated refactoring algorithm for the Count subcase is described in Section 7.4.2.

7.4.1 Detection

The process of detecting the Count Subcase depends on the programming language. For detecting the Count Subcase in logic flows, we have to consider the extended representation of the flow to verify if the nodes in a flow reference the Aggregate's runtimes properties. Figure 7.13 shows an example of the extended representation of the logic flow presented in Figure 7.12 used for assigning the number of products that belong to a given Category to a local variable NProducts. In this example, the filter is defined as the equality of Id, the primary key of the source Category, with a local variable CategoryId, and we expect that the local variable will match with all the products that belong to that category. When assigning to NProducts the result of the Count property, the program executes the same query defined in the Aggregate but this time for counting the records instead of returning them. The result of the query performed by the Count

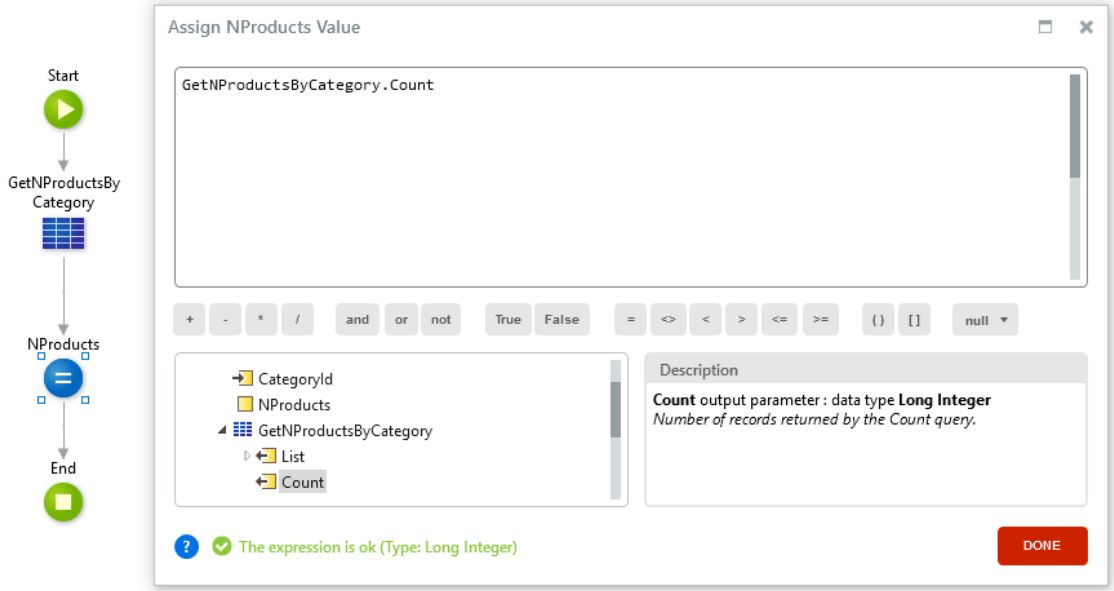


Figure 7.12: Count Runtime Property of an Aggregate

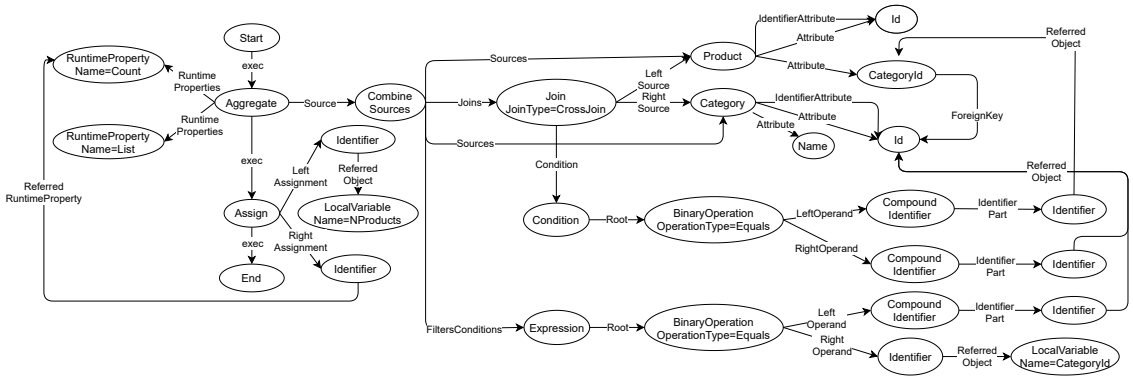


Figure 7.13: Graph structure example of an Aggregate used to count the number of records that match the query

property will have only one record: the number of products that belong to that category. This example matches the Count subcase of the Unlimited Records anti-pattern.

Algorithm 2 detects Aggregates used to count the number of records that match the query. It receives as input a set of n Aggregate nodes $A_1, A_2, ..., A_n$ and returns a set R of Aggregate nodes that match the Count subcase. It starts by iterating through the predecessors of the Aggregate (line 10), and then their predecessors recursively. The iteration stops if the Aggregate is referenced in a Preparation Flow or when the algorithm reaches the root of the graph. This process is done to analyze all direct and indirect predecessors of A_i to guarantee that A_i can be refactored safely. Next, it iterates through the Aggregate's successors (line 18) and checks if the successor is one of the Aggregate's runtime properties (line 19). The references to a runtime property will correspond to edges starting in any node and leading to the runtime property through reference edges.

Algorithm 2: Detecting Aggregates used to count the number of records that match the query algorithm.

Input: $A_1, A_2, \ldots, A_n$

```
1  R ← ∅;
2  for i ← 1 to n do
3  |  N, V ← {A_i}, ∅;
4  |  v ← True;
5  |  while v ∧ |N| > 0 do
6  |  |  n ← Pop(N);
7  |  |  if n ∉ V then
8  |  |  |  V ← V ∪ n ;
9  |  |  |  P ← Predecessors(n);
10 |  |  |  foreach p in P do
11 |  |  |  |  if IsPreparation(p) then
12 |  |  |  |  |  v ← False;
13 |  |  |  |  |  break
14 |  |  |  |  else
15 |  |  |  |  |  N ← N ∪ p ;
16 |  if v then
17 |  |  c ← False;
18 |  |  foreach s in Successors(A_i) do
19 |  |  |  if IsRuntimeProperty(s) then
20 |  |  |  |  foreach ie in InEdges(s) do
21 |  |  |  |  |  if IsReferred(ie) then
22 |  |  |  |  |  |  if IsList(s) then
23 |  |  |  |  |  |  |  v ← False;
24 |  |  |  |  |  |  |  break
25 |  |  |  |  |  |  else if IsCount(s) then
26 |  |  |  |  |  |  |  c ← True;
27 |  |  if v ∧ c then
28 |  |  |  R ← R ∪ A_i;
29 return R
```

Thus, the algorithm iterates through the "in edges" of the runtime property node (line 20) and checks if the edge corresponds to a reference (line 21). If the runtime property that was referenced was the List property (line 22), then the Aggregate's records are accessed and this Aggregate can not be refactored as a Count subcase. Otherwise, if the runtime property that was referenced was the Count property (line 25), then there is a possibility that this Aggregate corresponds to the Count subcase as long as the other conditions are satisfied. If v and c are both true at the end of the algorithm (line 27), then the Aggregate was not referenced in a Preparation Flow, the List property was not referenced and the Count property was referenced at least once, thus A_i matches the Count subcase.

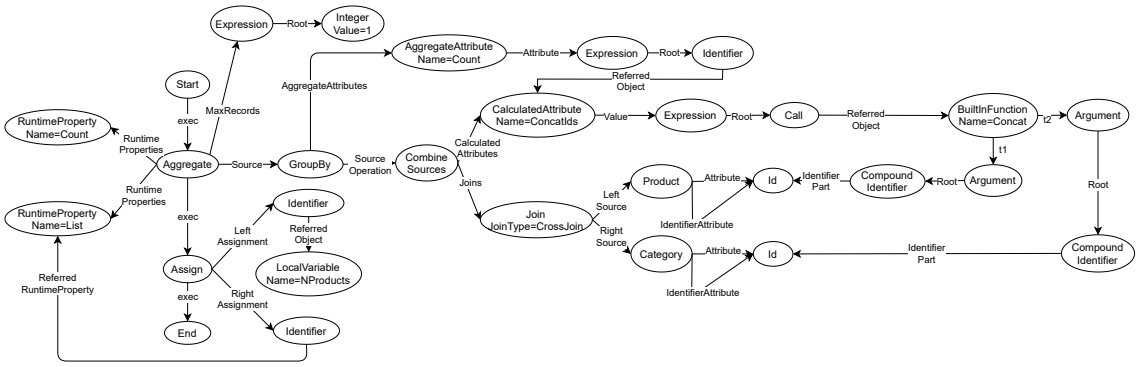
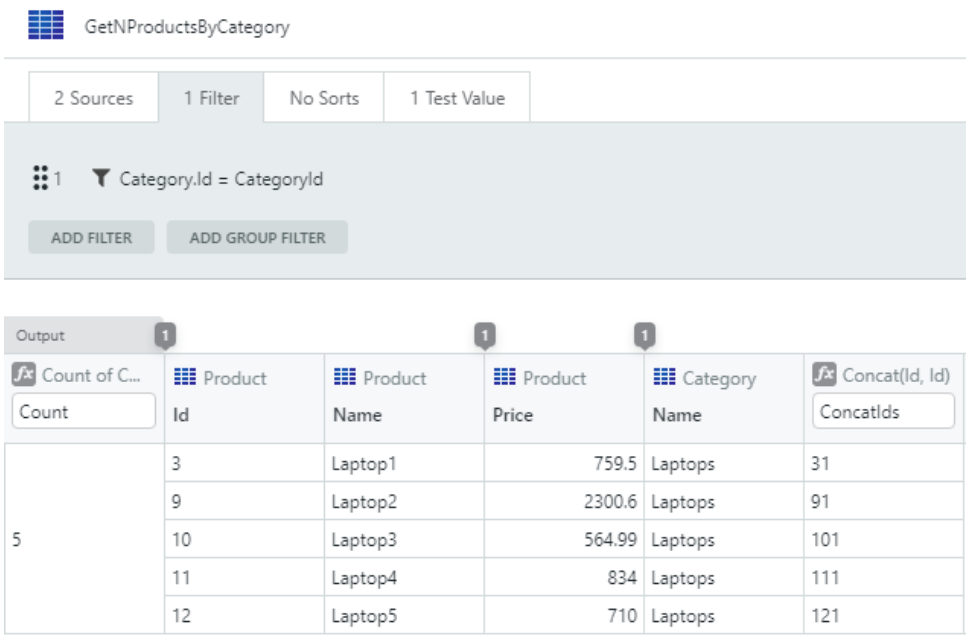


Figure 7.14: Graph structure example of an Aggregate used to count the number of records that match the query after refactoring

7.4.2 Refactoring

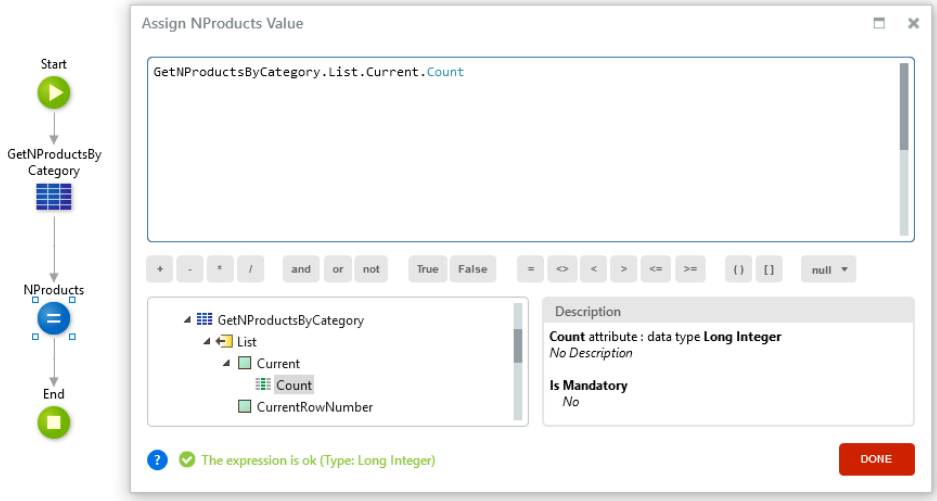
Our solution for automatically refactoring the Count subcase consists in replacing the references to the Count property with references to an additional aggregated attribute for counting, defined in the Aggregate. Thus, leading to only one query being performed instead of n queries, corresponding to the n accesses in a flow to the Count property. However, this process in OutSystems is not as straightforward, since Aggregates do not have mechanisms for 1) counting all rows (`COUNT(*)` or `COUNT(1)`), 2) counting more than one attribute (`COUNT(a,b)`), and 3) counting Group By attributes (`COUNT(b)... GROUP BY b`). Hence, for counting all rows, we add a calculated attribute to the query consisting of the concatenation of the primary keys of all the sources being queried and count the distinct values of this attribute instead of doing a count over all the attributes. Aggregates can have several joins, the join type that leads to the most results is the cross join. As shown in Section 4.1.3, for two tables R and S, the number of tuples resulting from the cross join operation is equal to the product between the number of tuples of the first table and the second table ($T(R) \times T(S)$), because each tuple in R (each distinct key of R) is paired with each tuple S (each distinct key of S). Thus, if we create a calculated attribute as the concatenation of the primary key of R with the primary key of S, we have a different value for this attribute per row as each distinct value for the primary key of R was paired with each distinct value for the primary key of S. These conditions can be extended to any type of join and as many joins as defined in the Aggregate. The number and type of filters performed do not affect the calculated attribute, as filters only restrict the data, leaving the size and the number of tuples in a query as at most the same as before applying them.

Figure 7.14 shows the transformations performed to the OutSystems extended flow following the example in Figure 7.13. In this example, besides adding the Max Records value the same way as performed for the Filter By Id subcase in Section 7.3.2, it is necessary to add the calculated attribute ConcatIds as the concatenation of the Ids of the Product and the Category tables. Next, the Count aggregate attribute is defined over



Count of C...	Product Id	Product Name	Product Price	Category Name	Concat(Id, Id)
Count					ConcatIds
5	3	Laptop1	759.5	Laptops	31
	9	Laptop2	2300.6	Laptops	91
	10	Laptop3	564.99	Laptops	101
	11	Laptop4	834	Laptops	111
	12	Laptop5	710	Laptops	121

(a) Aggregate's added attributes: Count and ConcatIds



(b) Replace extra query with access to the Count attribute

Figure 7.15: Count Subcase Refactoring in OutSystems

this calculated attribute. In Figure 7.15a it is possible to verify that the output of the Aggregate will only be the COUNT attribute after adding the two attributes in OutSystems. Finally, the right side of the assignment to NPRODUCTS is replaced to use the Count attribute present in the List property returned by the Aggregate instead of the Count property. Figure 7.15b illustrates the process in OutSystems of replacing the value side of the assignment. These transformations lead to only one query being performed and only one record is returned by the query. Limitations to this process are discussed in Section 7.6.

7.4.2.1 Rules

The following rules are considered for estimating the number of tuples resulting from a relational algebra expression over n relations:

- **One Relation.** Consider a relation R ($n = 1$) and the primary key a of R. Let E be the result relation of performing operations over R. If we define a COUNT function over a, because the COUNT produces the number of values in a column (Section 4.1.3.8) and because the selected column is a primary key, and so is unique for every single tuple, there exist T(R) distinct values for a in R. Thus, the COUNT value will hold exactly the number of tuples of relation R. Because no group by attributes were defined (nor other aggregated attributes), the COUNT computes one single result for the whole group (all the tuples of relation R). If we project only c, the result of the COUNT computation, as the projection operator leaves the number of tuples in the result unchanged, the query will return a single record.

$$T(\Pi_c(\mathcal{G}\ \mathrm{COUNT}(a) \rightarrow c\ (E))) = 1 \tag{7.7}$$

- **N Relations.** Consider n relations $R_1, R_2, \ldots, R_n$ ($n > 1$), and that $a_1, a_2, \ldots, a_n$ are the primary keys for each of the n relations, respectively. The relations can be joined in pairs using any type of join ($\bowtie$, $\overset{\theta}{\bowtie}$, $\overset{\theta}{\bowtie}$, $\overset{\theta}{\bowtie}$ and $\times$) and the join condition can be a combination of conditions $C_1, C_2, \ldots, C_n$. The estimate for the size of the result of joining the n relations will be at most the product of the number of tuples in each relation (see Section 4.1.5.2), as follows:

$$T(R_1 \overset{\theta}{\bowtie} R_2 \overset{\theta}{\bowtie} \ldots \overset{\theta}{\bowtie} R_n) = T(R_1) \times T(R_2) \times \cdots \times T(R_n)$$

  To distinguish the tuples in the relations we resort to the primary keys that can uniquely identify them in a relation (Section 4.1.1.2). Let E be the result relation of performing operations over the n relations, if we define a COUNT function over $a_1, a_2, \ldots, a_n$, because the COUNT produces the number of value combinations for the given columns (Section 4.1.3.8) and because the selected columns are primary keys, and so they uniquely identify every single tuple, there exist $T(R_1 \times R_2 \times \cdots \times R_n)$ distinct values in the result relations. Thus, no matter the number of joins performed the COUNT value will hold exactly the number of tuples in the result relation. Because no group by attributes were defined (nor other aggregated attributes), the COUNT computes one single result for the whole group (all the tuples of result relation E). If we project only c, the result of the COUNT computation, as the projection operator leaves the number of tuples in the result unchanged, the query will return a single record, as follows:

$$T(\Pi_c(\mathcal{G}\ \mathrm{COUNT}(a_1, a_2, \ldots, a_n) \rightarrow c\ (E))) = 1$$

Algorithm 3: Refactoring Aggregates used to count the number of records that match the query algorithm.

Input: $A_1, A_2, \ldots, A_n$

```
1  for i ← 1 to n do
2      S, T ← Sources(A_i), AggregateAttributes(A_i);
3      if |GroupBys(A_i)| = 0 ∧ |T| = 0 then
4          c ← null;
5          if |S| = 1 then
6              c ←AddCount(PrimaryKey(Pop(S)), A_i);
7          else
8              k ← ϵ;
9              foreach s ∈ S do
10                 k ← k· PrimaryKey(s);
11             c ← AddCount(k, A_i);
12         ReplaceCountReferences(c, A_i);
13         AddMaxRecords(1, A_i);
```

If instead of defining the COUNT function over the primary key attributes $a_1, a_2, \ldots, a_n$ we define an extended projection (Section 4.1.3.10) with a calculated attribute resulting from the concatenation of $a_1, a_2, \ldots, a_n$, the primary keys of all the sources being joined, we will produce a new column c with a value for every row (leaving the number of tuples intact even though the size of the relation increased). Since each expression has as many different tuples as the unique keys in the result relation, the COUNT function produces the number of value combinations for the given columns (Section 4.1.3.8) and the because new column k is the concatenation ("$\cdot$") of the primary keys, they still uniquely identify every single tuple, there exist $T(R_1 \times R_2 \times \cdots \times R_n)$ distinct values in the result relations. If we project only c, the result of the COUNT computation, as the projection operator leaves the number of tuples in the result unchanged, the query will return a single record, as follows:

$$T(\Pi_c(\mathcal{G}\ \mathrm{COUNT}(k) \rightarrow c\ (\Pi_{a_1 \cdot a_2 \cdot \ldots \cdot a_n \rightarrow k}(E)))) = 1 \tag{7.8}$$

7.4.2.2 Algorithm

Algorithm 3 is used for automatically refactoring the Count subcase. It receives as input a set of n Aggregate nodes $A_1, A_2, ..., A_n$ and for each Aggregate A_i, the algorithm checks if the Aggregate does not perform any group bys and does not have any aggregated attributes (line 3). If the Aggregate only has one source (line 5), it adds a count attribute c as the count of the primary key of the only source (line 6), which will hold exactly the number of tuples in the result relation (Rule 7.7). Otherwise, for each source S defined in the Aggregate, it concatenates the key k with the primary key of the current source S (line 10). After the iteration, it sets c as the count of the concatenated key (line 11), that will match the number of tuples in the result relation (Rule 7.8). Lastly, it replaces the

Table 7.1: Missing Max Records statistics.

Parameter	Total	%
Aggregates	737,443	-
Missing Max Records	514,826	69.8%

Table 7.2: Unlimited Records statistics matching subcase.

Parameter	Total	%
Aggregates	737,443	-
Filter By Id	234,564	31.8%
Count	4,532	0.6%

references of the Count runtime property with references to the count attribute *c* (line 12) and adds the Max Records edge to the Aggregate setting this value to one (line 13).

7.5 Experimental Evaluation

In this section, we evaluate the performance of the algorithms for detecting and automatically refactoring the Filter By Id and Count subcases of the Unlimited Records anti-pattern proposed in Section 7.3 and Section 7.4, respectively. All experiments were run on a Windows 10 Enterprise instance with 16 GB of RAM and an Intel® Core™ i5-8350U CPU@1.70GHz. The algorithms were executed on benchmark sets of graphs from a random sample of 500 real-world code bases written in OutSystems. The code bases were sampled from the total of 1481 code bases that existed at data collection time. Only benchmarks for which there exists at least one Aggregate are considered in the evaluation. In this analysis, two performance indicators are considered: the percentage of missing max records findings by subcase that can be solved with our algorithms and, the detection and refactoring time, i.e. the elapsed time since the start of the algorithm until termination.

Even though the Unlimited Records anti-pattern is already detected in Architecture Dashboard, because this is an offline proof of concept tool, we evaluated the presence of the anti-pattern in our sample by checking a simple rule to collect the necessary statistics without running it through Architecture Dashboard. As mentioned earlier, detecting an Unlimited Records occurrence is equivalent to checking if the Max Records value is set. In our analysis, we refer to Unlimited Records anti-pattern occurrences as Missing Max Records occurrences as the two concepts are akin. Statistics regarding the number of Aggregate nodes and Missing Max Records occurrences are summarized in Table 7.1. In the analyzed sample, 69.8% of Aggregates were Missing Max Records and each one of them leads to an unbounded query.

Table 7.2 presents the number of Aggregates that match each of the two subcases found in the benchmark sets, regardless of the value of their Max Records property. These statistics show that 31.8% of Aggregates match the Filter by Id subcase and that 0.6% of Aggregates match the Count subcase.

Table 7.3 shows that, for the Filter By Id subcase, 60.2% of the findings are missing the Max Records property and this is a substantial amount of findings to be resolved with our automated refactoring techniques. Table 7.4 is similar to the previous table except

Table 7.3: Filter By Id subcase statistics.

Parameter	Total	%
Filter By Id	234,564	-
Missing Max Records	141,324	60.2%
Max Records != 1	5,232	2.2%

Table 7.4: Count subcase statistics.

Parameter	Total	%
Count	4,532	-
Missing Max Records	3,981	87.8%
Max Records != 1	253	5.6%

Table 7.5: Maximum and total elapsed time for executing algorithms.

Algorithm	Max	Total
Detecting Missing Max Records	< 0s	10s
Detecting Filter By Id	5s	2m18s
Detecting Count	4s	58s
Refactoring Filter By Id	29s	5m18s
Refactoring Count	< 0s	1s
Benchmark Analysis	33m14s	13h59m0s

Table 7.6: Unlimited Records per subcase and Missing Max Records statistics.

Parameter	Min	$\rho = 0.25$	$\rho = 0.5$	$\rho = 0.75$	$\rho = 0.90$	$\rho = 0.95$	$\rho = 0.99$	Max
Missing Max Records	0.0%	57.5%	76.2%	89.1%	95.1%	98.7%	100.0%	100.0%
Filter By Id	0.0%	17.5%	24.8%	31.8%	39.7%	44.8%	59.5%	100.0%
Count	0.0%	0.0%	0.2%	0.8%	2.2%	3.7%	7.9%	10.0%

for the fact that it shows information regarding the Count subcase. Even though it is not as common for an Aggregate to fall into the Count subcase, 0.6% of Aggregates, the number of findings missing the Max Records property, 87.8% of occurrences, justifies the relevance of refactoring this subcase as this proves that developers find it difficult to reason about the number of records the Aggregate returns.

The percentage value for the 'Max Records != 1' parameter shows the number of findings that have the Max Records set to a value other than one. For the Filter By Id case only a small number of findings match this condition, 2.2% (Table 7.3). We can calculate that, for the Filter By Id subcase, a considerable amount of findings, 37.5%, have the Max Records property set to the correct value (based on our rules). Even though, only a small number of findings matching the Count subcase have the Max Records property set, the percentage of findings with the Max Records set to a value other than one, 5.6% (Table 7.4), is still lower than the number of findings set to the value we would expect, 6.6% ('= 1').

Table 7.5 shows the elapsed time for this experimental evaluation. It is possible to see that the time spent in detecting and refactoring the subcases of the Unlimited Records anti-pattern is considerably fast, less than 9 minutes, considering that the elapsed time for the total benchmark analysis amounts to 13 hours due to the process of loading and caching the data.

Table 7.6 shows the number of Missing Max Records per Unlimited Records subcase. These statistics represent the percentage of Missing Max Records that can be solved through our automated refactoring techniques. The $\rho = x$ columns show the result from

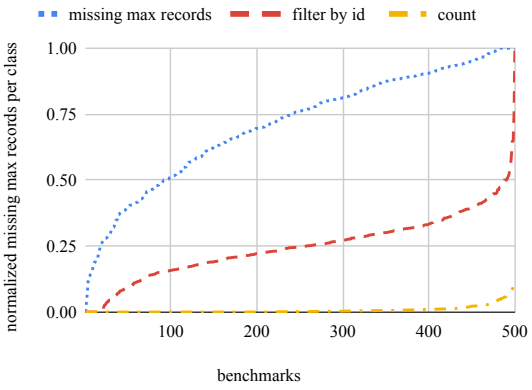


Figure 7.16: Normalized distribution of Missing Max Records per subcase for different values of the index's ρ parameter

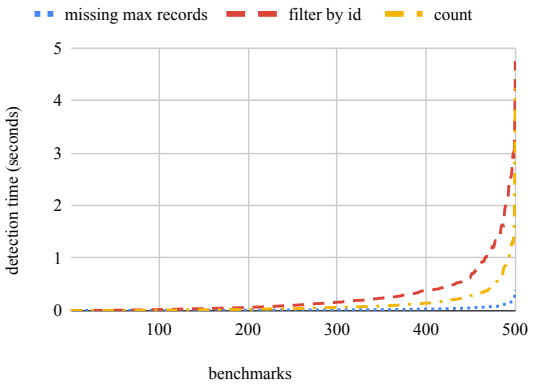


Figure 7.17: Distribution of detection time, in seconds, for detecting Missing Max Records per subcase

Table 7.7: Automated Refactoring statistics.

Parameter	**Total**	**% From Subcase**	**% From Total**
Refactored Filter By Id	140,175	99.2%	27.2%
Refactored Count	2,390	60.0%	0.5%
Total	142,565	98.1%	27.7%

the analysis of the x percentage of the benchmarks for each parameter. For example, a value of 24.8% in the $\rho = 0.5$ column of the 'Filter By Id' parameter represents that, in 50% of the benchmarks, 24.8% or less of the Aggregates match the Filter By Id subcase. A value of 0% in the 'Min' column represents benchmarks where every Aggregate matching that subcase has the Max Records property defined. Similarly, a value of 100% in the 'Max' column represents benchmarks where every Aggregate matching that subcase is missing the Max Records property.

Figure 7.16 shows a distribution plot comparing the percentage of Unlimited Records anti-pattern occurrences, Aggregates that are missing max records, with the percentage of these occurrences matching the identified subcases. We can see that a relevant amount of missing max records occurences are covered by the Filter By Id, subcase proving the relevance of its refactoring. Figure 7.17 shows a distribution plot comparing the times the detecting times of Missing Max Records detections per subcase. For a given subcase, each (x, y) point indicates that, for x benchmarks, the detection time of that subcase is at most y. For example, the $(400, 0.4)$ point in the line that corresponds to detecting the Filter By Id subcase indicates that 400 of the benchmarks are processed in 0.4 seconds or less by the Filter By Id detection algorithm. Overall, we can see that the detection process takes less than 5 seconds per subcase for each benchmark.

Table 7.7 presents statistics concerning the cases that were automatically refactored, taking into account the limitations presented in Section 7.6. For the Filter By Id subcase,

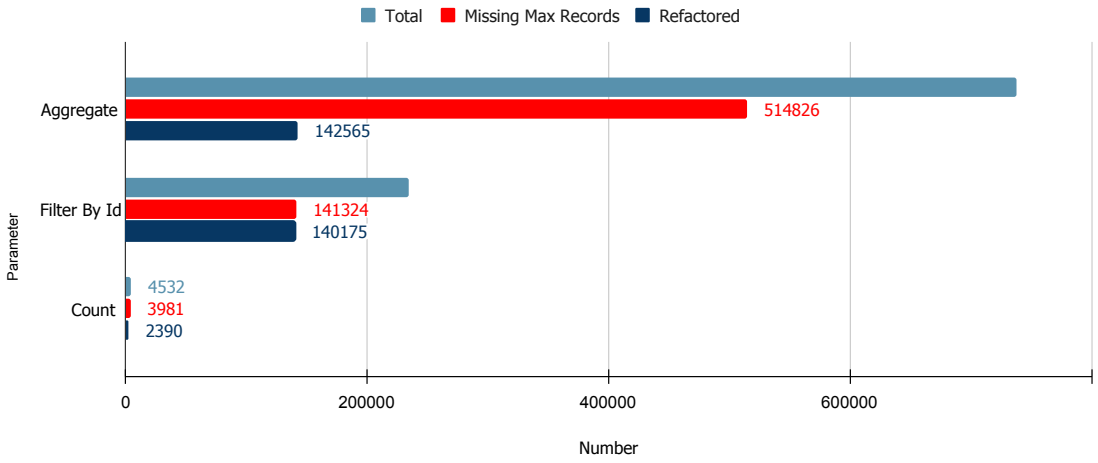


Figure 7.18: Number of Missing Max Records Detected and Refactored Per Parameter

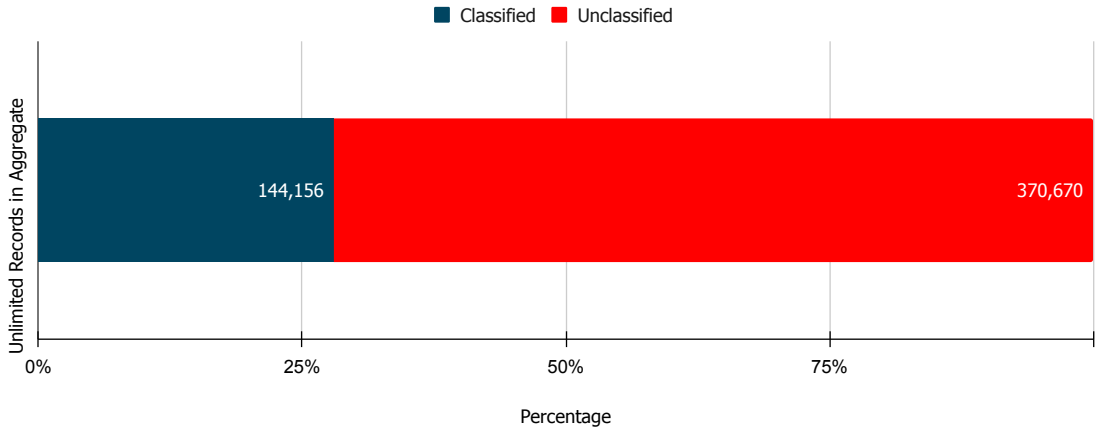


Figure 7.19: Percentage Of Unlimited Records Findings Classified/ Unclassified

99.2% of the findings were automatically refactored, having only dismissed the interception cases, which amounts to 27.2% of the total Missing Max Records. As for the Count subcase, 60% were automatically refactored, dismissing both the interception cases and the invalid cases, which amounts to 0.5% of the total Missing Max Records. Lastly, it is shown that 27.7% of the total Missing Max Records found in the analyzed benchmarks were solved with our automated refactoring techniques. Figure 7.18 illustrates these refactoring efforts.

Lastly, Figure 7.18 illustrates the ratio between the number of Aggregates Missing Max Records (Unlimited Records) that we were able to classify in our work versus the amount that remains unclassified. We managed to classify 28% of Missing Max Records occurences (144,156 Aggregates in total). We do not count the interception cases as classified Unlimited Records. As 72% of the Unlimited Records occurrences remain unclassified, some analysis can still be done to continue to identify different subcases in future work.

7.6 Limitations and Discussion

While executing the algorithm for detecting Aggregates that match the Filter By Id subcase, some Aggregates were considered invalid as the information about their sources and respective keys is not accessible and therefore cannot be evaluated by the algorithm. This amounts to a total of 2368 invalid Aggregates present in the analyzed benchmarks. Nonetheless, these Aggregates are included in our evaluation.

After executing both the algorithms for detecting Aggregates that match the Filter By Id subcase and the Count subcase we computed the interception of these results. Developers can define a flow with an Aggregate that filters by primary key and that accesses the Count property to find out how many records were returned. This situation is different from the previous subcases since it logically corresponds to checking if the list returned was empty or not, as the Count property value will always be zero or one. A total of 1149 interception cases were found, which correspond to ≈29% of the Count subcase occurrences and to ≈0.8% of the Filter By Id subcase occurrences. We left this resolution for future work deciding to further analyze these occurrences before refactoring.

Our approach for automatically refactoring the Count subcase considers that the SAP does not have a mechanism for 1) counting all rows, 2) counting more than one attribute, and 3) counting group by attributes; as is the case in OutSystems. However, the proposed detection algorithm for the Count Subcase does not capture these constraints. Thus, our refactoring algorithm evaluates these conditions to discard occurrences of the Count subcase that we cannot refactor. Another option would be to sacrifice the generality of the detecting algorithm by adding rules that would enforce these constraints. We chose the generality of the subcase detection over the generality of the refactoring as the latter strongly depends on the programming language. For counting all rows, Algorithm 3 adds a calculated attribute to the query consisting of the concatenation of the primary keys of all the sources being queried. Although the time for creating this extra attribute for counting is considered in our analysis, the performance of this step in run time is not yet evaluated as this is a prototype tool. This evaluation will have to be considered in future work. Nonetheless, if the SAP has mechanisms for counting all the attributes in the query, the algorithm can simply create a count attribute over all the attributes, thus counting all the rows that match the query. Regarding the problem of counting group by attributes, this limitation could not be overcome in OutSystems leading to verification of no group bys in line 3 of the same algorithm. Additionally, Aggregates that have aggregated attributes previous to adding the count attribute are also considered invalid for refactoring, as it is not possible to predict that these queries would also return at most one record when considering that group bys compute one result per group for each aggregate attribute. Although presented as a limitation, we consider that it would not be of much interest for developers to define other aggregated attributes and then only count the number of rows. These limitations, together with the interception cases, lead to the fact that it was only possible to refactor 2390 (60%) out of the 3981 count cases,

dismissing a total of 1149 (≈29%) interception cases and a total of 442 (≈11%) invalid cases.

In our analysis besides collecting statistics on how many findings of the Unlimited Records anti-pattern could be automatically refactored with our algorithms, we also collected statistics on the Filter By Id and Only Use Count subcases whose Max Records was set to a value different than one. These findings are not being solved automatically because we do not know the reasons that led the developers to set the Max Records to those values. However, we have developed algorithms for fixing these findings and, once we have a more complete study, we can apply them to solving this form of technical debt. Solving this issue will be left for future work.

Our main contribution is the definition of rules for finding different subcases within the Unlimited Records anti-pattern that occurs, even though it might not be detected, in both SAPs and programs that query data and do not mandatorily limit the number of records returned. The general anti-pattern detection and the refactoring solution are adaptable, in principle, even to textual programs such as SQL.

The immediate alternatives to our approach are 1) to define the max records property by construction, which presents no choice to users and is not backward compatible, or 2) to extend the current anti-pattern detecting system with the presented rules and still signaling the cases where the refactoring cannot be automatic. The latter seems to be the more natural evolution of our work.

8

Duplicated Code Anti-pattern

The Duplicated Code Anti-pattern was identified in OutSystems as the most common anti-pattern, reaching as high as 39% in some code-bases [6]. Terra-Neves *et al.* proposed a duplicated code pattern mining algorithm that leverages the visual structure of VPLs to not only detect duplicated code but also highlight the duplicated code patterns.

In this chapter, we start by explaining how we parameterized the labels used in the duplicated code detector to detect Type I Duplicates. Then, we explain how we processed the detections in order to find a maximal refactorable sub-graph for each pattern. Next, we explain the steps followed to perform automated refactoring on the duplicated logic flows. Lastly, we will present how we evaluated our approach and its limitations.

8.1 Detection

For detecting duplicates, we used a duplicated code detector for OutSystems proposed by Terra-Neves *et al.* [6] that maximizes the detection of Type III Duplicates, where near-misses are allowed for node expressions as long as the graph structure of the duplicated part is the same.

The type of duplicates detected in this approach depends on the **node and edge labels**. The edge labels are set to their respective types in the logic flows. On the other hand, node labels can be complex depending on their type, e.g. an If node label has to consider the kind of condition being checked in addition to the respective variable types and function calls.

The first step in our analysis before detecting duplicated sub-flows was to define the kind of duplicates we were looking for. We can customize the detection by parameterizing the node and edge labels that will be used to match the graphs. The inclusion of different node attributes further refines the search, leading to the detection of Type I Duplicates. According to each type of node/edge, different attributes need to be included and evaluated. For Type I Duplicates, we include every attribute that defines each node/edge except for display names (used to denominate nodes in the flow and having no impact on the execution and evaluation of expressions). As an example, for an Aggregate to be

a duplicate of Type I, all the joins, sources, filters, calculated attributes, sorts, group bys, and aggregated attributes need to be an exact match.

At the root of comparing each type of node is the comparison of the expressions that these nodes reference in the logic flow. For Type I Duplicates, the expression needs to be the same, and so we follow the following rules for each expression kind:

- **Basic:** the value and type need to be the same in the two logic flows (e.g. both the values for the two expressions reference the Basic type Boolean value true).
- **Operation:** the value and type of the expressions referred to in the operation are the same in the two logic flows (e.g. for a Unary Operation the negation of a Basic Boolean expression is the same as another if the Boolean expression value is the same).
- **Identifier**: the type and value of the attribute being referred need to be the same in the two logic flows (e.g. both the local variables being compared are called x and have value y).
- **Compound Identifier:** both these rules need to be followed: 1) the attributes of the node/ structure referred to in the expression have the same values and types and 2) the Identifier specifies an attribute of that node with the same value and type (e.g. a Compound Identifier referring an Aggregate node is the same as another if 1) the sources referred by the Aggregate are the same, 2) it performs joins over the same sources, 3) the values for the sorts/filters/other attributes are the same, and at last 4) the Identifier refers the same attribute/property of the Aggregate).
- **Call**: the function being called needs to be the same in the two logic flows, and for each parameter of the function the value of the expression for that argument also needs to be the same (e.g. a call to a function over two Text expressions is the same as another, if the function is the same (*concatenation*) and the values attributed to those two Text expressions are the same in both duplicates).

Lastly, the definition of Type I Duplicates considers that there can be differences in whitespace, layout, and comments. These requirements are met as follows:

- Differences in **Whitespace** can only occur in the definition of expressions. Thus, for each expression, we remove the whitespace occurrences in the expressions, including blanks, newlines, tabs, *etc.*, before comparison.
- Differences in **Comments** can occur by adding different Comment nodes to a flow and changing the string value that they represent. Thus, Comment nodes are entirely discarded in the detection of duplicates.
- Differences in **Layout** in OutSystems can occur when positioning the different nodes of a logic flow throughout the canvas. Thus, the position ('X' and 'Y' coordinates) of

nodes in the canvas is not included in the node labels to be compared. Additionally, the duplicated code detector only identifies clones whose graph structure of the duplicated part is the same.

8.2 Duplicated Patterns Processing

The result of running the duplicated code detector is a set of **tree hierarchies of duplicated patterns** (sub-graphs of the logic flows) with an indication of the logic flows, nodes and edges where the duplication occurs. Let G and G' be duplicated code patterns that occur across the logic flows in sets F and F' respectively. Assuming that, at some point during its execution, the algorithm extracts an MCS G_C for G and G', then G_C is a possibly smaller pattern that occurs across the flows in $F \cup F'$. The tree hierarchy contains an internal node for G_C with two children nodes for G and G'. Analogously, children of G would represent possibly larger patterns that occur in subsets of F [6].

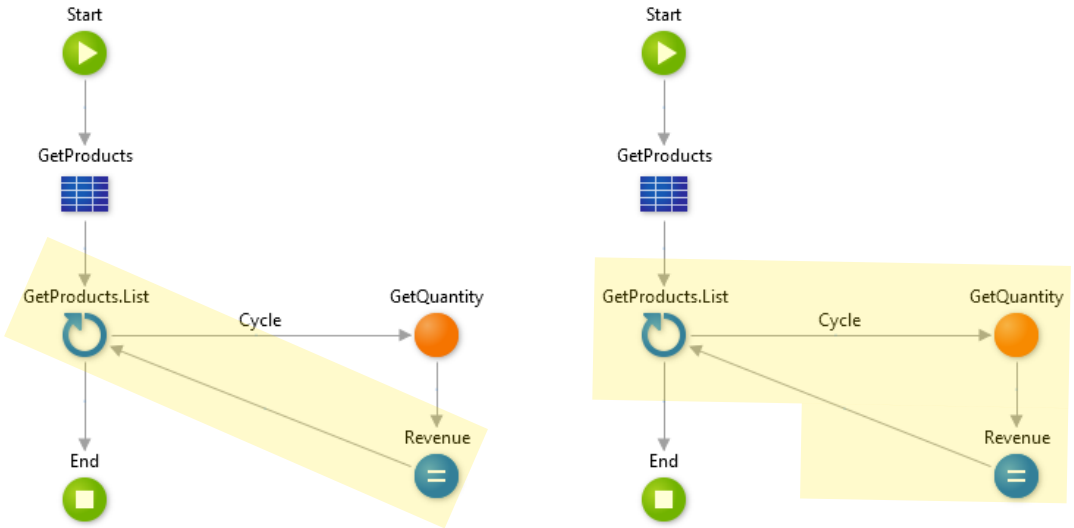
For each set of patterns, we chose the pattern with the highest refactor value from the flattened representation of the tree hierarchy (as a list). The highest refactor weight for a graph G is computed as described in Section 4.4.2.3. The **refactor value estimate** for graph G with refactor weight w_G and that occurs in n logic flows, is exactly the same as the total number of nodes and edges that can be removed when refactoring the set of logic flows where G occurs (taking into account that not all nodes have a refactor weight of 1) and is given by:

$$w_G \times (n-1) - n$$

Start and End nodes are discarded in the MaxSAT formulation phase, as they must appear in every flow and therefore cannot be refactored to a separate flow. Even though some post-processing is done to eliminate flows and nodes that cannot be refactored, our analysis found that some of the duplicated patterns could not be refactored in a straightforward way. Next, we will explain such cases that need some extra processing before executing the extract function mechanism. In the following examples consider edges whose labels are empty as having type Connector, since this is the default label.

8.2.1 Analysis

Consider the logic flow in Figure 8.1a where only one part of the cycle (coming from the For Each node) is included in the duplicated pattern. This pattern cannot be refactored because there are two outgoing edges outside the pattern, the Cycle edge and the Connector edge of the For Each node. If we were to refactor this pattern, there would be no way of including the Execute Action node branched from the cycle, since we would not have access to it in a separate logic flow. Next, consider the logic flow in Figure 8.1b besides noticing that the whole cycle is included in the duplicated pattern, we can verify that there is only one branch going from a node in the pattern to the nodes of the remaining



(a) Unrefactorable Pattern (b) Refactorable Pattern

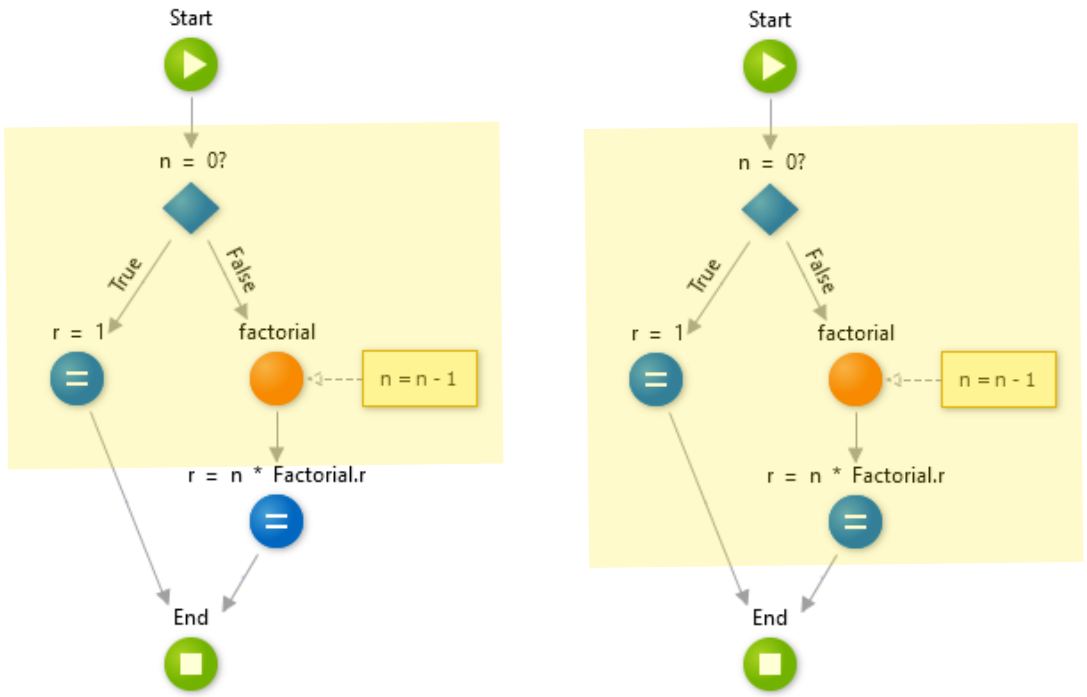
Figure 8.1: Duplicated patterns including For Each nodes

logic flow, we can generalize this condition by saying that **all branches whose destination is outside of the pattern go to the same node**, thus this pattern can be refactored.

Additionally, another important condition can be extracted from the logic flow in Figure 8.1a. There are two incoming edges from outside the pattern, the Cycle edge and the Connector edge of the Assign node. If we were to refactor this pattern, since all logic flows begin in a Start node with only one branch, there would be no way of including both the For Each node and the Assign node as Connector branches of the Start node. Next, consider the logic flow in Figure 8.1b, we can verify that there is only one branch going from a node outside the pattern into nodes in the pattern, the Connector branch whose destination is the For Each node, we can generalize this condition by saying that **all branches from outside the pattern into nodes in the pattern go to the same node**, thus this pattern can be refactored.

A different example can be seen for If nodes, consider the logic flow in Figure 8.2a where the True branch of the If node is completely refactorable (until the End node), whilst the False branch is not (missing one Assign node). Similarly, if we were to refactor this pattern, there would be no way of including the Assign node branched following the False branch since we would not have access to it in a separate logic flow. Next, consider the logic flow in Figure 8.2b besides noticing that both branches are completely included in the duplicated pattern, we can verify that even though two branches are going from nodes in the pattern to the nodes of the remaining logic flow, **all branches whose destination is outside of the pattern go to the same node**, thus this pattern can be refactored.

Following the previous example, consider the logic flow in Figure 8.3a where both



(a) Unrefactorable Pattern

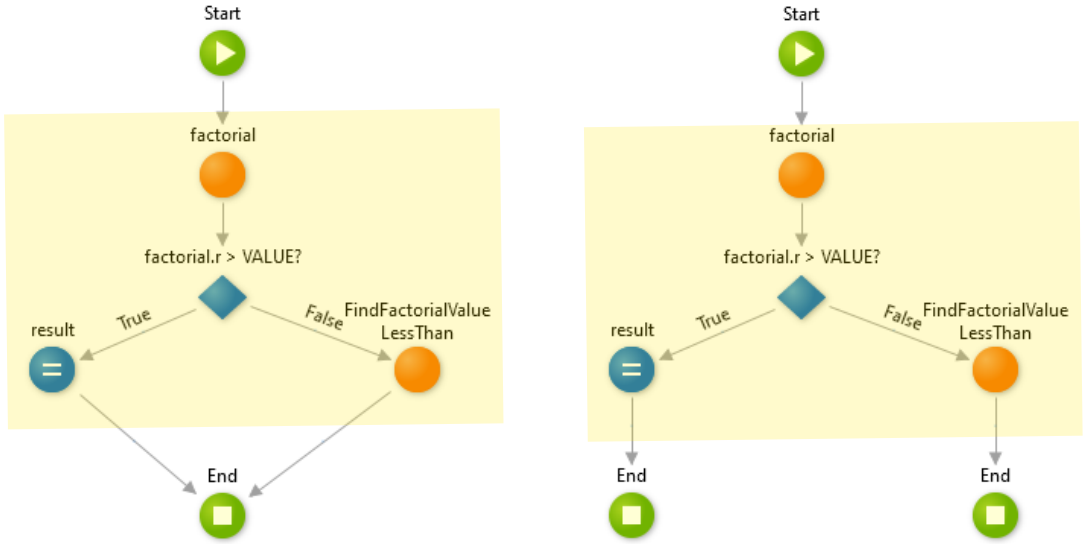
(b) Refactorable Pattern

Figure 8.2: Duplicated patterns including If nodes

branches of the If node are completely included in the duplicated pattern. As seen in the previous example, since all branches with destination outside of the pattern go to the same node, this pattern can be refactored. Next, consider an equivalent logic flow in Figure 8.3b, where the nodes following the True and False branches, branch to different End nodes. Although in this example the refactoring rule where all branches whose destination is outside of the pattern go to the same node does not hold, a different rule applies since all destinations are End nodes. We can imagine a refactoring of this pattern since both these End nodes can be added in the separate logic flow (or replaced with a single End node), in the current logic flow they can be replaced with a single End node when the pattern is refactored. Thus, **all patterns whose branches' destination outside of the pattern go to nodes of type End node**, can be refactored.

8.2.2 Rules

We will now formally present the rules that can be extracted from the examples above. Let G be a logic flow, defined as a directed weakly connected graph $G = (V, E)$. Let P be a sub-graph of G, that defines the duplicated pattern. P is defined as $P = (V_p, E_p)$ and where $V_p \subset V$ and $E_p \subset E$, and for any edge $(u, v) \in E_p$, $u, v \in V_p$. P is never the same as G because Start and End nodes cannot be refactored and neither can their connecting edges. We start by giving two useful definitions:



(a) Pattern with edges outside the pattern branching to the same node

(b) Pattern with edges outside the pattern branching to End nodes

Figure 8.3: Similar refactorable patterns

- **Root.** A node $v \in V$ is a root if there is no edge $(u, v') \in E$ where $v = v'$, i.e. there is no edge in E towards v.
- **Leaf.** A node $u \in V$ is a leaf if there is no edge $(u', v) \in E$ where $u = u'$, i.e. there is no branch in E for u.

Start nodes are roots since they have no edges incident to the node. On the other hand, End nodes are leaves since there in no edge in the logic flow that starts in an End node. The pattern mining algorithm has a no spurious edge constraint, i.e. an edge is only in E_p if both its nodes are in V_p. Thus, if the edges are only included if both nodes are in the pattern, and since there is a mandatory Start node in the logic flow G that cannot be included in the pattern, then there will always be an incoming edge into the pattern from a node in outside the pattern (Start nodes have exactly one Connector branch), that cannot be included in E_p that makes its destination a root of the pattern. This rule is formalized as follows:

- **At least one root in the pattern.** There is at least one node $v \in V_p$ that is the root of the pattern.

$$\exists r \in V_p \forall (u, v) \in E_p \Rightarrow v \neq r \tag{8.1}$$

A similar situation happens for End nodes, where there is at least one End node in the logic flow G and since these cannot be included in the pattern, then there will always be at least one branch from a node in the pattern to a node in G (End nodes have at least one

Function 2: GetUnrefactorableNodes(G,P)

Input: G, P

```
1  N ← ∅;
2  I ← InEdges(Nodes(P)) \ Edges(P);
3  if |I| > 1 then
4  |   if ¬AllEquals(InNodes(I)) then
5  |   |   N ← OutNodes(I);
6  if N = ∅ then
7  |   O ← OutEdges(Nodes(P)) \ Edges(P);
8  |   if |O| > 1 then
9  |   |   if ¬AllEquals(OutNodes(O)) ∨ ¬AllEndNodes(OutNodes(O)) then
10 |   |   |   N ← OutNodes(O);
11 return N;
```

incoming edge) that cannot be included in E_p. Additionally, the root of the pattern and the node with a branch to a node outside the pattern (if there is only one) cannot be the same node since the no isolated node constraint prohibits it, because a node is only in V_p if there is at least one edge in E_p to another node in V_p, i.e. the pattern has at least two nodes.

Finally, we can present the rules that the duplicated pattern P needs to respect before refactoring:

- **All incoming edges to the only root.** There is one node $r \in V_p$ that is the root of the pattern, and for all edges $(u,v) \in E$ where $u \notin V_p$ and $v \in V_p$, r is equal to v, i.e. there is at least one incoming edge from an origin outside the pattern and all incoming edges branch to the same destination in the pattern, the root of the pattern.

$$\exists r \in V_p \forall (u,v) \in E, u \notin V_p, v \in V_p \Rightarrow v = r \tag{8.2}$$

- **All outgoing edges to the same destination or to End node type.** There is one node $d \in V$, and for all edges $(u,v) \in E$ where $u \in V_p$ and $v \notin V_p$, d is equal to v or v has type End node, i.e. there is at least one branch to a destination outside the pattern, and all branches from nodes in the pattern to nodes outside the pattern go to the same node or to a node with type End node.

$$\begin{gathered}(\exists d \in V \forall (u,v) \in E, u \in V_p, v \notin V_p \Rightarrow v = d) \\ \vee \\ (\forall (u,v) \in E, u \in V_p, v \notin V_p \Rightarrow IsEndNode(v))\end{gathered} \tag{8.3}$$

8.2.3 Algorithm

Function GetUnrefactorableNodes receives as input a logic flow G and P, a subgraph of G containing only the duplicated pattern, and returns N the set of nodes that

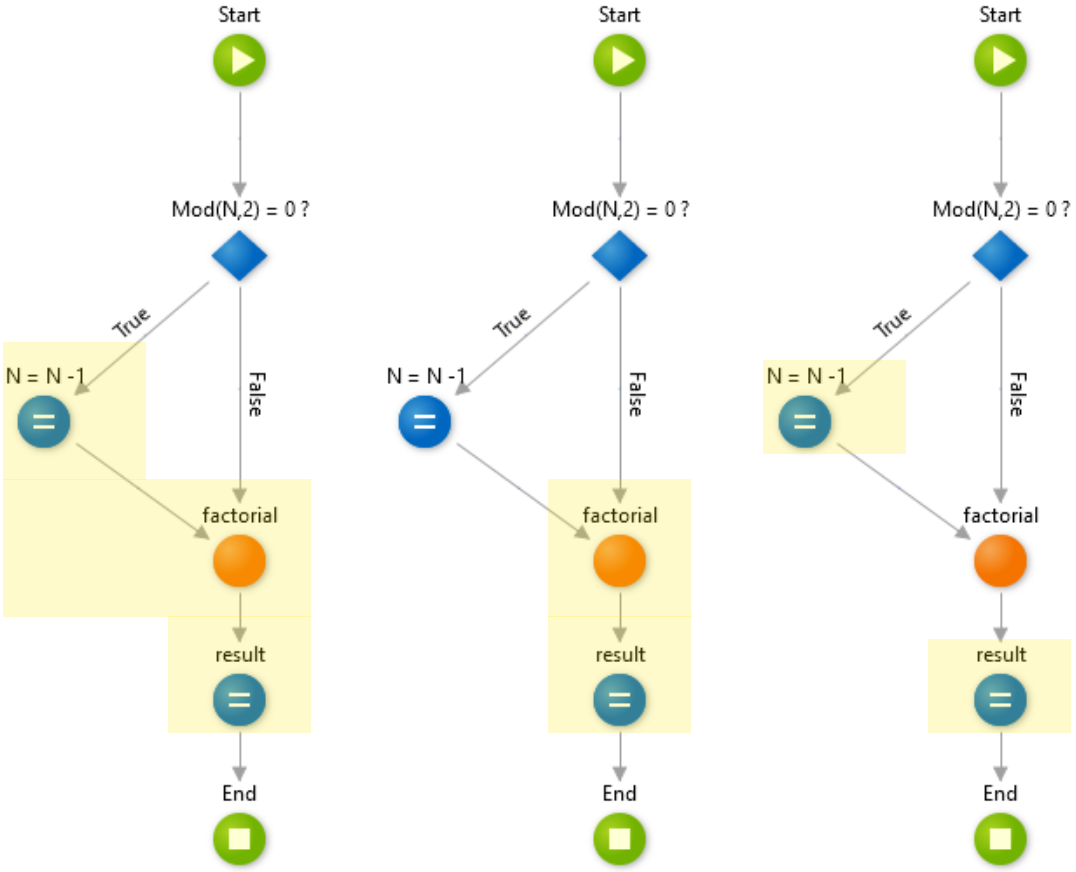
Algorithm 4: Maximal refactorable sub-graph algorithm.

Input: G, P, N

```
1  S, m ← ∅, β;
2  foreach n ∈ N do
3  |  S_t ← DiGraph(Nodes(P) \ n, Edges(P) \ (InEdges(n) ∪ OutEdges(n)));
4  |  if |Nodes(S_t) > 1| ∧ RefactorValue(S_t) > m then
5  |  |  U ← GetUnrefactorableNodes(G, S_t);
6  |  |  if U ≠ ∅ then
7  |  |  |  S_t ← MaximalSubgraph(G, S_t, U);
8  |  |  if S_t ≠ ∅ then
9  |  |  |  S ← S_t;
10 |  |  |  m ← RefactorValue(S_t);
11 return S;
```

cannot be refactored in a straightforward way, i.e. the multiple roots of the pattern that break Rule 8.2 or the multiple nodes outside the pattern that break Rule 8.3. The set of nodes is empty if all the nodes in the pattern can be refactored in a straightforward way. First, we initialize I as the set of all branches into nodes of S and remove those that are edges of S (line 2). This leaves us with all the branches from G to P. If I has more than one edge (line 3), since there is at least one because the pattern has at least one root (Rule 8.1), we check if all the origin nodes are the same (line 4), since this pattern can only be refactorable if all its incoming edges from outside the pattern have the same origin (Rule 8.2). If this condition does not hold, we set N as the roots of the pattern that cannot be refactored together (line 5). If N is still empty (line 6), i.e. the pattern has only one root, we initialize O as the set of all branches from nodes of S and remove those that are edges of S (line 7). If O has more than one edge, since there is at least one because the pattern has at least one outgoing edge outside the pattern, we check if all the destination nodes are the same or if they are all End nodes (line 8), since this pattern can only be refactorable if all its branches to nodes outside the pattern have either the same destination or an End node as destination (Rule 8.3). Similarly, if this condition does not hold, we set N as the leaves of the pattern that cannot be refactored together (line 10).

We need to find a maximal sub-graph of P that respects the constraints in Section 8.2.2, along with the constraints for the pattern miner, by removing one or more nodes that cannot be refactored together. Algorithm 4 receives as input: G a logic flow, P a sub-graph of G (containing only the duplicated pattern), and N a set of nodes to be removed to find a maximal sub-graph that respects the constraints, and returns S a maximal sub-graph of P that is refactorable or an empty graph. For each node in N, we try to find a valid pattern S_t which has all the nodes of P except for n and all the edges of P except for the incident edges and branches of n (line 3). Next, we compute the refactor weight of S_t and we check that S_t has more than one node (no isolated nodes) and that the refactor weight is higher than the maximum refactor weight found m (line 4), which has to be higher than



(a) Unrefactorable Pattern (b) Refactorable Pattern (c) Unrefactorable Pattern

Figure 8.4: Maximal refactorable sub-graph example

the minimum refactor weight threshold β. Then, we check if the sub-graph S_t of P is still not refactorable by calling the function GetUnrefactorableNodes that checks the rules in Section 8.2.2 and returns U the set of nodes that cannot be refactored together. If U is not empty, we recursively call the algorithm but now with U as the set of nodes to remove (line 7). If the algorithm returns a sub-graph S_t that in not a *null graph* (line 8), it will have at least one node and the refactor weight will be higher than the maximum found, we found a graph that can be refactorable and with the highest refactor weight. Lastly, we set S as the maximal sub-graph to be returned (line 9) and m as the highest refactor weight found (line 10) and the iteration continues.

An example of applying the algorithm is shown in Figure 8.4. The logic flow in Figure 8.4a does not respect Rule 8.2 since the highlighted duplicated pattern has two roots, and thus the Assign node ($N = N - 1$) and the Execute Action node (FACTORIAL), cannot be refactored together. Algorithm 4 iteratively tests sub-graphs by removing nodes from the set of nodes that could not be refactored together, Figure 8.4b shows one of these iterations after removing the Assign node ($N = N - 1$). We can verify that all the edges coming from the logic flow into the highlighted duplicated pattern go to the same node

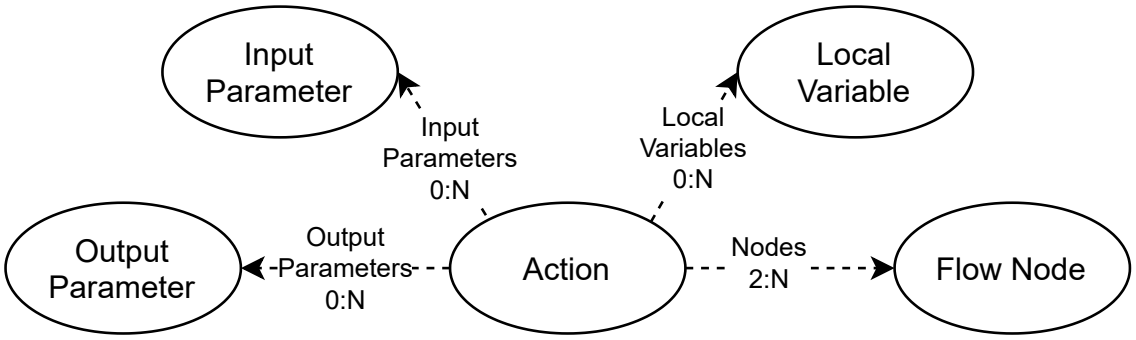


Figure 8.5: Action extended representation

(Rule 8.2), thus the pattern only has one root and since all the edges from the only node with outgoing edges outside the pattern (Assign RESULT) go to the same node, this sub-graph can be refactored. A second iteration of the algorithm would remove the Execute Action node instead and the result would leave us with both rules being broken instead of one since the two incoming edges from the logic flow into the highlighted pattern go to separate nodes (Rule 8.2) and the two branches from the highlighted pattern into the logic flow also go to separate nodes that are not of type End (Rule 8.3). Since the pattern is still not refactorable, the next iterations would leave the pattern with only one node. Algorithm 4 before adding the sub-graph as a possible maximal sub-graph, checks if the pattern has more than one node (line 4) otherwise, it is considered invalid since such graphs do not respect the no isolated node constraint.

8.3 Refactoring

We consider the refactoring of Type I Duplicates as refactoring a subcase within the Duplicated Code anti-pattern. Once we have a refactorable duplicated pattern with the highest refactor value and a set of logic flows where the duplicated pattern occurs, we can start refactoring the flows. First, we consider the extended representation of an action as shown in Figure 8.6. An action is a logic flow that, just like a function in a regular programming language, has:

- **Input parameters:** the parameters used to pass values into functions/ logic flows (can be none).
- **Output parameters:** the parameters(s) returned from the functions/ logic flows (the set is empty when the function's return type is VOID).
- **Local variables:** the variables whose scope is within the function/ logic flow.
- **Nodes:** the statements of the function or for the case of logic flows, the statements represented as nodes in the flow.

8.3.1 Rules

We follow the extract function refactoring mechanism steps but taking into account that the pieces of code to refactor are logic flows.

1. **Copy the duplicated pattern into a new logic flow.** We add a new logic flow (action) with one Start node and one End node. Then, we connect the nodes of the duplicated pattern to the new logic flow nodes and set them as the action's nodes.

2. **Pass references to local variables as input parameters.** We scan the duplicated pattern for references to any local variables and pass them as input parameters of the action. If a local variable is only used inside the new logic flow, we add a local variable to the new action and remove it from the other logic flows.

3. **Pass references in the duplicated pattern as parameters.** We scan the duplicated pattern for references to variables that are assigned and their values. In the logic flow, these variables can be input parameters, output parameters, the result of previous Instruction nodes or Aggregate nodes that are in the scope of the logic flow. We pass the referenced input parameters and the referenced results of the nodes defined before the duplicated pattern (displayed as "Before Nodes" in Figure 8.6) as input parameters of the new logic flow. If the output parameters of the original logic flows are assigned in the duplicated pattern, we add them as output parameters of the new logic flow. Lastly, we replace the assignments in the duplicated pattern with the new variables.

4. **Pass references in the original flow as parameters.** We scan the original logic flow for references to the result of Instruction nodes in the duplicated pattern. If any node that comes after the duplicated pattern (displayed as "After Nodes" in Figure 8.6) accesses the result of an Instruction node in the duplicated pattern in OutSystems we add this result as an output parameter of the new logic flow and replace the references in the original flow with this new variable.

5. **Replace the duplicated pattern in the original logic flows with a call to the new logic flow.** We add an Execute Action node as shown in Figure 8.7 with reference to the new action and reconnect the original action. Additionally, we have to add the arguments for the input parameters of the new action and set the values accordingly. If the input parameter of the new action was originally an input parameter, we simply set the argument to take the value of the original input parameter. If the input parameter was a local variable or the result of an Instruction node, we pass this value accordingly. Lastly, we add an Assign node when needed to assign to the original output parameters the return values of the output parameters that are assigned in the duplicated pattern.

A similar process is executed by the extract to action functionality in OutSystems.

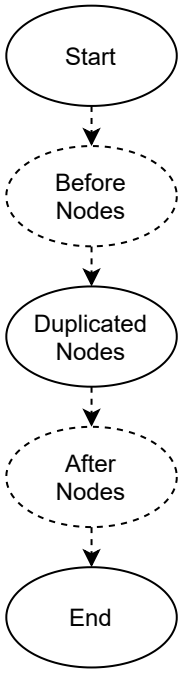


Figure 8.6: Action flow extended representation

8.3.2 Algorithm

The maximal sub-graph S of P returned by algorithm 4, respects the constraints in Section 8.2.2 and has the highest refactor value of its sub-graphs. S is computed according to a logic flow G that we want to refactor. Nonetheless, we always have at least two logic flows to refactor, or we would not have a duplicated pattern between flows. Thus, the resulting sub-graph S depends on the graph structure of the logic flow passed to the algorithm and where we are looking for maximal sub-graphs. With this in mind, we have to compare the maximal sub-graphs given by the duplicated pattern for each flow G. Additionally, the result of the duplicated code detector is a tree hierarchy of duplicated patterns, thus when we chose a refactorable pattern from the set comprised of the root graph (or its highest refactorable value sub-graph according to each flow G) and of its children (or a highest refactorable value sub-graph per child according to each flow G), we have to take into account that the number of logic flows that can be refactored with this graph may have decreased. Thus, we recalculate the refactor value for each graph with the new number of logic flows. The graph to be refactored will be the one with the highest refactorable value that respects the constraints in Section 8.2.2.

Algorithm 5 receives the tree hierarchy of duplicated patterns T and the logic flows $G_1, G_2, \ldots, G_n$ where the patterns occur (all patterns of T do not necessarily occur in every G_i or even have the same subset of occurrences) and returns a new logic flow A that encapsulates the selected duplicated pattern and the set R with logic flows that we were able to refactor and where the duplicated pattern is no longer present. It starts by calling auxiliary function TreeMaximalSubgraph (line 1) that for each of the patterns in

Algorithm 5: Refactoring logic flows given a tree hierarchy of duplicated patterns.

Input: $G_1, G_2, \ldots, G_n, T$

```
1 P, R ← TreeMaximalSubgraph(G_1, G_2, ..., G_n, T);
2 if P ≠ ∅ then
3 |   A ← AddAction(P);
4 |   AddParameters(A, P);
5 |   foreach r ∈ R do
6 |   |   r ← RefactorGraph(A, P, r);
7 return A, R
```

the flattened tree and according to each G_i calculates the maximal sub-graph with the highest refactor value from the given patterns. The highest refactor value sub-graph P can correspond to one of the patterns or to a maximal sub-graph of one of the patterns and R is the set of graphs where the pattern was identified and can be refactored. The function ultimately calls Algorithm 4 and Function GetUnrefactorableNodes to find this maximal sub-graph. If a valid pattern that matches at least two logic flow with no isolated nodes and with a refactor weight higher than β cannot be found, the *null graph* is returned instead of the maximal sub-graph. If a pattern was found (line 2), the call to auxiliary function AddAction performs step 1 by copying P to a separate logic flow and making the necessary connections. Then, the call to auxiliary function AddParameters performs steps 2 to 4 by adding the necessary input parameters, local variables and output parameters to the new logic flow (line 4). Considering the case of adding local variables, the function checks if the local variables accessed in the pattern are used before and after the duplicated pattern, and then 1) if the local variable is only used in the pattern, it is removed from the original logic flows and added as a local variable of the new action, 2) if the local variable is used before the pattern, it is added as an input parameter, 3) if the local variable is used after the pattern, it is added as an output parameter, 4) if the local variable is used both before and after the pattern, it needs to be assigned at the end of the new action to carry its value to the original logic flows since the value received has been modified, and 5) any output parameters of the new action will need to be assigned in the original logic flows right after calling the new action, and the local variables that became output parameters are no exception. After adding the action's parameters in algorithm 5, for each of the graphs in R, where the pattern can be refactored, we perform step 5 by calling auxiliary function RefactorGraph (line 6) and replacing the duplicated pattern within each graph with a call to the new logic flow A. In this function, we add both the arguments values for each parameter and the necessary assignments after the call to the new action for the output parameters and the local variables that remained in the original flows.

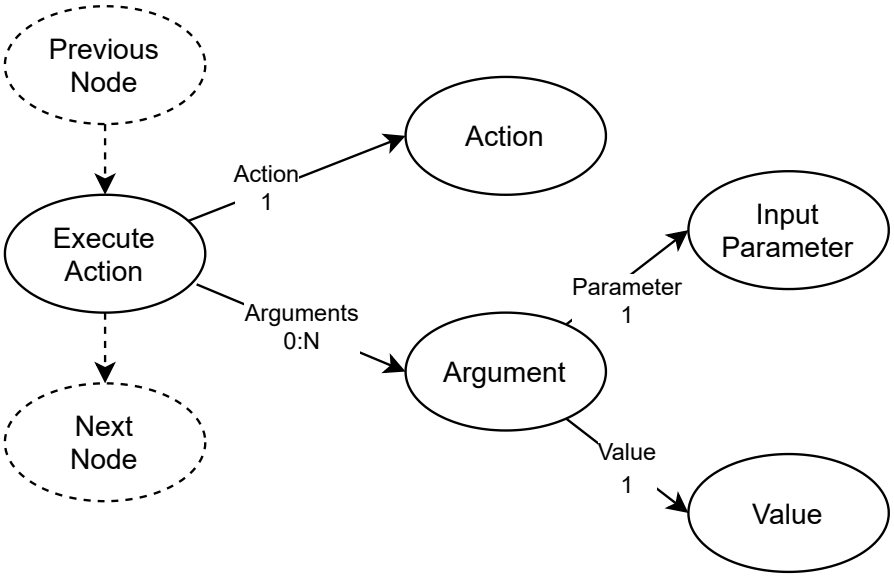


Figure 8.7: Execute action extended representation

8.4 Experimental Evaluation

In this section, we evaluate the performance of the algorithms for finding a maximal refactorable sub-graph of a duplicated pattern and for automatically refactoring Type I Duplicates of the Duplicated Code anti-pattern proposed in Section 8.2.3 and Section 8.3.2, respectively. All experiments were run on a Windows 10 Enterprise instance with 16 GB of RAM and an Intel® Core™ i5-8350U CPU@1.70GHz. The algorithms were executed on benchmark sets of graphs from a random sample of 500 real-world code bases written in OutSystems. The code bases were sampled from the total of 1221 code bases that existed at data collection time. Only benchmarks for which there exists at least one logic flow are considered in the evaluation. This results in a final collection of 497 benchmarks. In this analysis, two performance indicators are considered: the duplicated refactor weight fixed with our refactoring algorithm and, the detection and refactoring time, i.e. the elapsed time since the start of the algorithm until termination. The same node/edge may appear multiple times across different patterns, but the respective refactor weight is counted only once.

In our analysis, to obtain only the action flows in which we are interested in detecting duplicates, we use filters defined and implemented by the AI team that discard unwanted flows, such as flows that are part of the OutSystems environment, flows that correspond to system or template flows, flows that are too small to be refactorable, Preparation flows, *etc*. Additionally, a significant amount of flows were considered invalid as the information about their nodes necessary to build the node labels for matching graphs is not accessible or incomplete and therefore cannot be evaluated by our algorithms. Flows with a refactor weight lower than the threshold 5 (β) are also discarded. Before running the mining algorithms, nodes that cannot be refactored to a separate logic flow, such as Start and

Table 8.1: Number of flows statistics.

Parameter	Min	Median	Max
Flows	35	1,642	20,255
Flows considered	1	87	1,215
Flows with duplicated code	0	20	539
Flows with duplicated code refactored	0	19	475

Table 8.2: Maximum and total elapsed time for executing algorithms.

Algorithm	Max	Total
Mining time	2s	1m16s
Finding pattern time	6h31m18s	52h34m20s
Refactoring time	7m56s	2h25m13s
Benchmark Analysis	6h43m5s	120hh40m38s

End nodes are discarded. Some post-processing is done to discard certain patterns with If and Switch nodes, since these can only be refactored if at least one of their branches is also part of the duplicated code pattern. Nonetheless, the total number of logic flows, nodes and edges considered for an environment in our statistics is independent from these filters so that our estimates are accurate. Table 8.1 presents statistics regarding the number of flows found in the analyzed benchmarks. "Flows considered" corresponds to the flows that are not discarded before mining duplicates. "Flows with duplicated code" represent the flows where duplicated code has been detected and thus the flows where we will apply automated refactoring.

Table 8.2 shows the elapsed time for this experimental evaluation. It is possible to see that almost no time is spent in detecting the Duplicated Code anti-pattern, being this algorithm the result of a work focused on scalability [6]. Note that the elapsed time for executing our algorithms for "Finding" a valid pattern and for "Refactoring" the pattern include the time spent in copying the flow representations since we operate in a copy of the original data. Additionally, the elapsed time for the "Refactoring' algorithm also takes into account the time spent building logic flow representations of the refactored flows, since we execute the refactoring on extended representations of the flows but still wanted to leverage the visual facet of SAPs that was already maintained in the detection of the flows. Thus, we are able to do a side by side comparison of the detected pattern and its logic flows with its refactoring representation, for each pattern tree refactored. The high values for the elapsed time of our algorithms justify the need for a future revision of their implementations, in order to improve their performance and scalability, besides the possibility of running these experiences in a different setting than the enterprise instance mentioned above.

Table 8.3 shows the amount of duplication found in the benchmark sets for Type I duplicates. The $\rho = x$ columns show the result from the analysis of the x percentage of the benchmarks for each parameter. For example, a value of 1.2% in the $\rho = 0.5$ column

Table 8.3: Duplicated code of Type I found and refactored per benchmark set statistics.

Parameter	**Min**	$\rho = 0.25$	$\rho = 0.5$	$\rho = 0.75$	$\rho = 0.9$	$\rho = 0.95$	$\rho = 0.99$	**Max**	**Total**
Flows with duplicated code	0%	0.5%	1.2%	2.0%	2.9%	3.5%	5.4%	10.1%	–
Duplicated nodes found	0%	0.2%	0.7%	1.0%	1.6%	2.0%	2.8%	5.1%	–
Duplicated weight found	0	44	134	363	619	872	1451	3178	132695
Flows refactored	0%	0.2%	0.9%	1.6%	2.4%	3.1%	4.9%	9.3%	–
Duplicated nodes refactored	0%	0.1%	0.5%	0.8%	1.2%	1.6%	2.7%	4.6%	–
Duplicated weight refactored	0	24	88	250	478	610	1152	2537	94822

Table 8.4: Aggregated refactoring values statistics.

Parameter	**Max**	**Total**	**%**
Duplicated weight found	3,178	132,695	-
Duplicated weight refactored	2,537	94,822	71%
Flows with duplicated code found	539	19,862	-
Flows with duplicated code refactored	475	15,253	77%
Duplicated nodes found	1,804	73,997	-
Duplicated nodes refactored	1,481	53,523	72%
Pattern trees found	95	3,390	-
Patterns trees refactored	54	2,144	63%

of the 'Flows with duplicated code' parameter indicates that, in 50% of the benchmarks, 1.2% or less of the flows are found to contain duplicated code of Type I. Note that these percentages consider the full universe of flows/nodes present in these benchmarks before pre-processing. There are benchmarks where no duplication of Type I was found ("Min"=0%) and thus, the duplicated weight found in those benchmarks was equal to zero. If we consider only benchmarks where there exists duplication, the minimum duplicated weight value is 14. The last lines in the table represent the percentage of Type I duplicates that can be solved through our automated refactoring techniques out of the total duplication found in the benchmarks.

Table 8.4 shows the refactoring performed in the benchmark sets out of the total duplication detected. For the case of Type I Duplicates, we found a total refactor weight of 132,695 and we managed to solve 71% of this refactor weight. The refactor weight that was not solved corresponds to the patterns of the tree hierarchy that the finding pattern algorithm did not choose for refactoring, since we only chose one pattern per tree. Additionally, when the pattern with the highest refactor value cannot be refactored, we have to find a maximal refactorable sub-graph and end up losing the refactor weight that corresponds to the node and edges that are eliminated from the pattern. Lastly, if no maximal refactorable sub-graph for the tree hierarchy is found, with a minimum refactor weight equal to the threshold β, the total refactor weight of that pattern tree is lost since it cannot be refactored. On the other hand, these constraints have less impact on the number of flows, since most of the time, a maximal refactorable sub-graph of the pattern trees can be found and we are able to refactor the flows (solving 63% of the patterns corresponds to solving 77% of the flows).

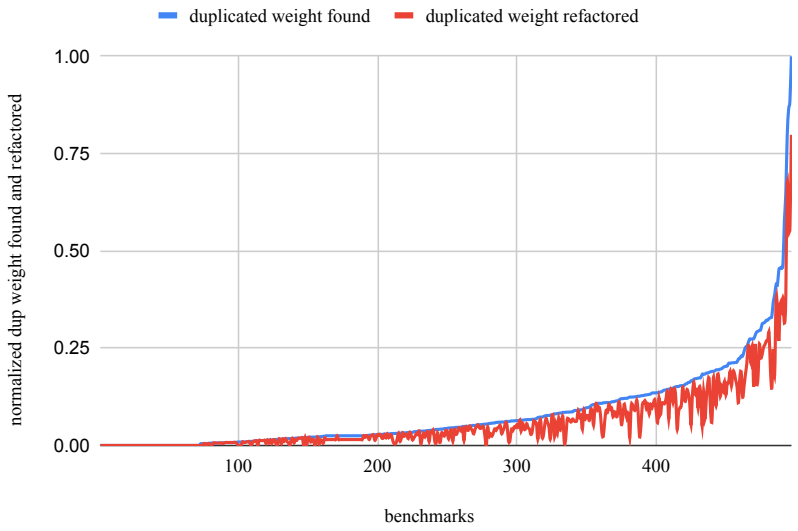


Figure 8.8: Normalized duplicated refactor weight found/ refactored distribution per benchmark.

Figure 8.8 shows a distribution plot of duplicated refactor weight found and respective refactoring per benchmark. These values are normalized against the largest duplicated refactor weight value found in all of the analyzed benchmarks and the plot is presented in ascending order of the duplicated refactor weight found. For a given benchmark x, if the y coordinate in the plot for the duplicated weight found matches the y coordinate for the duplicated weight refactored that, then the duplicated refactor weight found was completely refactored for that benchmark. The accentuated negative dips correspond to benchmarks where we were not able to refactor the total duplicated refactor weight found, and are due to benchmarks where a maximal sub-graph for each set of detected duplicated patterns was not found as explained above. The small negative variations correspond to benchmarks where we lost refactor weight by only selecting one pattern for refactoring or by finding a smaller sub-graph that can be refactored straightforwardly.

8.5 Limitations and Discussion

In this work, we present results from an evaluation of 500 real-world code bases written in OutSystems. Since the effectiveness of the developed techniques was evaluated in the context of the OutSystems ecosystem, and since the infrastructure does not have test banks defined for its code bases, that would allow for continuous integration, it would be unfeasible to validate the refactoring operations automatically. In the future, when the OutSystems platform has mechanisms for automated tests, our results can be automatically validated. An alternate validation could be performed if there existed an equivalence checker for OutSystems, so that our refactorings could be formally verified in an automated way.

We have selected 10% of the refactored patterns for manual validation (214 patterns out of the 2,144 refactored patterns). To the best of our knowledge, the refactorings performed are correct and the program's behavior is expected to be maintained. The precision and recall of the proposed approach strongly depend on the quality of the node and edge labels. Our focus in this dissertation is the establishment of rules that allow for the refactoring to be performed and not in the validation of the refactoring operations that could easily be fixed in the case of a bug. Note that we have refactored Type I Duplicates, which correspond to exact clones, and that this is a prototype tool running offline.

Our detection and refactoring capabilities are limited to Type I clones and thus our results have a low impact in the total duplication. Extending our automated refactoring techniques to be applied on Type II and Type III Duplicates that maintain the graph structure of the duplicated part will be left for future work. Additionally, our approach only refactors one pattern per tree hierarchy and in some cases, it might be possible to refactor more than one pattern detected in the logic flows, e.g. sub-flows of the pattern without common nodes and edges.

Some efforts can still be applied on improving the performance of the algorithms proposed in this work. The elapsed time for finding a maximal sub-graph and refactoring the duplicates can be reduced, since our focus was not on algorithm scalability but mainly on refactoring the highest possible value, thus ultimately contributing to lowering technical debt. These improvements will be left for future work.

9

Conclusion And Future Work

In this chapter, we present the concluding remarks of this dissertation. Finally, we suggest proposals for future work indicated by our research.

9.1 Conclusions

Our work is motivated by the high technical debt problem found in SAPs. Applications built with a short-term mindset end up consuming a large portion of a company's resources, time, and energy. These resources are spent maintaining and rewriting defective code that contributes to high technical debt instead of focusing on the development of new ideas [49].

In this dissertation, to mitigate the above challenge, we presented an approach to automatically refactor a set of relevant high-impact anti-patterns that contribute to high technical debt. The high technical debt problem can be subdivided into smaller problems, each consisting of resolving a single identified anti-pattern. The problems are independent and we were able to progressively resolve, test, and evaluate our solution. This allowed for an iterative resolution with independent deliveries where each step consists of the automated resolution of an anti-pattern and each improvement increases the final solution value.

The first anti-pattern for which we developed automated refactoring techniques was the Unlimited Records Anti-pattern. The profusion of this anti-pattern in OutSystems proves to be a real concern, as 7 out of 10 Aggregates are missing the Max Records property, each contributing to an occurrence of this anti-pattern. These findings incur a negative impact on software maintenance and evolution, being the state-of-the-art solution having developers fixing them by hand when notified by Architecture Dashboard. When considering the quantity of these findings as each application usually has several Aggregates (737,443 Aggregates in total across 45,405 modules), this can become extremely time-consuming. If we also consider the background necessary to estimate the size of queries, the fixing of these findings can prove to be an arduous task. Scenarios such as queries returning only one record by construction (e.g. get by primary key) hinder the

handling of true problems. Furthermore, for the identified Count subcase, the problem also negatively impacts the performance of applications, increasing screen loading time since 1) it corresponds to an extra query and 2) all the records that match the Aggregate definition are returned when only one was necessary. Hence, we proposed automated refactoring techniques for detecting and automatically refactoring two subcases of the Unlimited Records anti-pattern and that effectively solve 28% of all occurrences.

The Duplicated Code Anti-pattern was identified in OutSystems as the most common anti-pattern, reaching as high as 39% in some code-bases. As this pattern is found in such a large part of the OutSystems code base, it is understandable why it was included in the initial set of anti-patterns refactored with automated techniques. In our work, we have identified that the amount of duplication of Type I in OutSystems can reach as high as 5.1% and we were able to refactor as high as 4.6%.

In this work, we were able to refactor subcases of two high-impact anti-patterns that result in the elimination of findings that contribute to technical debt. Since this is a prototype tool we cannot evaluate that the warnings for the refactored anti-patterns have disappeared in Architecture Dashboard and thus lowering technical debt in the active code base. Nonetheless, since we are working on a snapshot of the data consisting of real-word code bases and since our automated prototype tool has solved a considerable amount of findings, if these refactoring techniques were to be implemented on the running version of the code, we are positive that the technical debt would decrease. Thus, we have shown that it is possible to lower technical debt in SAPs by implementing automated refactoring techniques that solve the anti-patterns causing high technical debt.

9.2 Future Work

An interesting topic for future work is the definition of refactoring rules and algorithms for other relevant anti-patterns and adding them to the conjoint prototype tool since the high technical debt problem can be subdivided into smaller problems, each consisting of resolving a single identified anti-pattern.

In terms of research, a particular challenge would be to explore ways of providing a guided but automated refactoring experience to the user, allowing for visualization of the refactoring changes and refactoring automatically on approval. This research can benefit from the visual representations of refactored logic flows presented in this dissertation for the refactoring of Type I Duplicates.

Future work can also include widening the set of subcases to automatically refactor within each anti-pattern. An example of a subcase to explore, considering the Unlimited Records anti-pattern, is the Empty Subcase that occurs in flows that execute unnecessary and possibly complex queries to only check if the result is empty. As 72% of the Unlimited Records occurrences remain unclassified, some analysis can still be done to continue to identify different subcases. Regarding the Duplicated Code anti-pattern, we can extend

our automated refactoring techniques to be applied to Type II and Type III Duplicates that maintain the graph structure of the duplicated part.

Finally, the current refactoring implementation relies on the OutSystems concepts for the extended representation of logic flows. Future research could examine if this implementation could be decoupled from these concepts.

Bibliography

[1] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999. ISBN: 978-0-201-48567-7. URL: http://martinfowler.com/books/refactoring.html (cit. on pp. xiii, 1, 2, 31, 32, 35, 36, 46, 113).

[2] T. Mens and T. Tourwé. "A survey of software refactoring". In: *IEEE Transactions on software engineering* 30.2 (2004), pp. 126–139 (cit. on p. 2).

[3] A. Ouni et al. "Maintainability defects detection and correction: a multi-objective approach". In: *Autom. Softw. Eng.* 20.1 (2013), pp. 47–79. DOI: 10.1007/s10515-011-0098-8. URL: https://doi.org/10.1007/s10515-011-0098-8 (cit. on p. 2).

[4] J. C. Seco, H. Lourenço, and P. Ferreira. "A common data manipulation language for nested data in heterogeneous environments". In: *Proceedings of the 15th Symposium on Database Programming Languages*. 2015, pp. 11–20 (cit. on pp. 2, 59).

[5] R. Tairas and J. Gray. "Increasing clone maintenance support by unifying clone detection and refactoring activities". In: *Inf. Softw. Technol.* 54.12 (2012), pp. 1297–1307. DOI: 10.1016/j.infsof.2012.06.011. URL: https://doi.org/10.1016/j.infsof.2012.06.011 (cit. on pp. 2, 39, 47, 48).

[6] M. Terra-Neves et al. "Duplicated code pattern mining in visual programming languages". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 1348–1359 (cit. on pp. 3, 38, 43–45, 53, 54, 85, 87, 99).

[7] I. P. Fernandes, M. Terra-Neves, and J. C. Seco. "Automated Refactoring of Unbounded Queries in Software Automation Platforms". In: *Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '21. Fukuoka, Japan, 2021 (to appear) (cit. on p. 4).

[8] P. Kruchten, R. L. Nord, and I. Ozkaya. "Technical Debt: From Metaphor to Theory and Practice". In: *IEEE Software* 29.6 (2012), pp. 18–21. DOI: 10.1109/MS.2012.167 (cit. on p. 16).

[9] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020. ISBN: 9780078022159. URL: https://www.db-book.com/db7/index.html (cit. on pp. 20, 22, 23, 28, 30).

[10] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983. ISBN: 0-914894-42-0. URL: http://web.cecs.pdx.edu/\%7Emaier/TheoryBook/TRD.html (cit. on p. 22).

[11] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009, pp. I–XXVI, 1–1203. ISBN: 978-0-13-187325-4 (cit. on pp. 22, 23, 27, 28, 30).

[12] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Signature Series. Addison-Wesley, 2019. ISBN: 978-0134757599. URL: http://martinfowler.com/books/refactoring.html (cit. on pp. 31–37, 39).

[13] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN: 0321278658 (cit. on p. 31).

[14] J. Aldous, S. Best, and R. Wilson. *Graphs and Applications: An Introductory Approach*. The Open University. Springer London, 2003. ISBN: 9781852332594. URL: https://books.google.pt/books?id=1qRvTI_oWUAC (cit. on p. 38).

[15] W. T. Tutte. "On the Problem of Decomposing a Graph into n Connected Factors". In: *Journal of the London Mathematical Society* s1-36.1 (Jan. 1961), pp. 221–230. ISSN: 0024-6107. DOI: 10.1112/jlms/s1-36.1.221. eprint: https://academic.oup.com/jlms/article-pdf/s1-36/1/221/2467215/s1-36-1-221.pdf. URL: https://doi.org/10.1112/jlms/s1-36.1.221 (cit. on p. 38).

[16] W. Wang and M. W. Godfrey. "Recommending Clones for Refactoring Using Design, Context, and History". In: *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 331–340. DOI: 10.1109/ICSME.2014.55. URL: https://doi.org/10.1109/ICSME.2014.55 (cit. on pp. 39, 46).

[17] F. Arcelli Fontana et al. "Software clone detection and refactoring". In: *International Scholarly Research Notices* 2013 (2013) (cit. on p. 39).

[18] M. Fowler. "Avoiding repetition [software design]". In: *IEEE Software* 18.1 (2001), pp. 97–99 (cit. on p. 39).

[19] C. K. Roy and J. R. Cordy. "A survey on software clone detection research". In: *Queen's School of Computing TR* 541.115 (2007), pp. 64–68 (cit. on pp. 39, 40).

[20] C. Ansótegui, M. L. Bonet, and J. Levy. "SAT-based MaxSAT algorithms". In: *Artificial Intelligence* 196 (2013), pp. 77–105 (cit. on pp. 43, 49).

[21] A. A. B. Baqais and M. Alshayeb. "Automatic software refactoring: a systematic literature review". In: *Softw. Qual. J.* 28.2 (2020), pp. 459–502. DOI: 10.1007/s11219-019-09477-y. URL: https://doi.org/10.1007/s11219-019-09477-y (cit. on pp. 46, 50, 51).

[22] R. Tairas and J. Gray. "Get to know your clones with CeDAR". In: *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. Ed. by S. Arora and G. T. Leavens. ACM, 2009, pp. 817–818. DOI: 10.1145/1639950.1640030. URL: https://doi.org/10.1145/1639950.1640030 (cit. on p. 47).

[23] Y. Wang et al. "Synthesizing database programs for schema refactoring". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, pp. 286–300 (cit. on pp. 48, 51).

[24] V. Raychev et al. "Refactoring with synthesis". In: *ACM SIGPLAN Notices* 48.10 (2013), pp. 339–354 (cit. on p. 49).

[25] A. Ouni et al. "Improving multi-objective code-smells correction using development history". In: *J. Syst. Softw.* 105 (2015), pp. 18–39. DOI: 10.1016/j.jss.2015.03.040. URL: https://doi.org/10.1016/j.jss.2015.03.040 (cit. on pp. 49–51).

[26] G. Bavota, A. D. Lucia, and R. Oliveto. "Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures". In: *J. Syst. Softw.* 84.3 (2011), pp. 397–414. DOI: 10.1016/j.jss.2010.11.918. URL: https://doi.org/10.1016/j.jss.2010.11.918 (cit. on p. 50).

[27] G. Ganea, I. Verebi, and R. Marinescu. "Continuous quality assessment with inCode". In: *Sci. Comput. Program.* 134 (2017), pp. 19–36. DOI: 10.1016/j.scico.2015.02.007. URL: https://doi.org/10.1016/j.scico.2015.02.007 (cit. on p. 51).

[28] M. Kessentini et al. "Search-Based Design Defects Detection by Example". In: *Fundamental Approaches to Software Engineering - 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by D. Giannakopoulou and F. Orejas. Vol. 6603. Lecture Notes in Computer Science. Springer, 2011, pp. 401–415. DOI: 10.1007/978-3-642-19811-3_28. URL: https://doi.org/10.1007/978-3-642-19811-3_28 (cit. on p. 51).

[29] A. ben Fadhel et al. "Search-based detection of high-level model changes". In: *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 2012, pp. 212–221. DOI: 10.1109/ICSM.2012.6405274. URL: https://doi.org/10.1109/ICSM.2012.6405274 (cit. on p. 51).

[30] M. W. Mkaouer et al. "Recommendation system for software refactoring using innovization and interactive dynamic optimization". In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. Ed. by I. Crnkovic, M. Chechik, and P. Grünbacher. ACM, 2014, pp. 331–336. DOI: 10.1145/2642937.2642965. URL: https://doi.org/10.1145/2642937.2642965 (cit. on p. 51).

Webography

[31] M. Henriques. *Introducing the "2020 Digital Experiences Summit: Real-World Stories to Help your Digital Customer Experience Transformation"*. https://www.outsystems.com/blog/posts/digital-experiences-summit/. 2021. (Visited on 02/23/2021) (cit. on p. 1).

[32] F. Alexander. *What Is Low Code*. https://www.outsystems.com/blog/posts/what-is-low-code/. 2021. (Visited on 02/12/2021) (cit. on p. 1).

[33] M. Fowler. *Technical Debt*. https://martinfowler.com/bliki/TechnicalDebt.html. 2019. (Visited on 02/23/2021) (cit. on pp. 1, 16).

[34] OutSystems. *Setting Up OutSystems*. https://success.outsystems.com/Documentation/11/Setting_Up_OutSystems/. 2020. (Visited on 02/03/2020) (cit. on p. 7).

[35] OutSystems. *Service Studio Overview*. https://success.outsystems.com/Documentation/11/Getting_started/Service_Studio_Overview/. 2020. (Visited on 02/03/2021) (cit. on p. 8).

[36] OutSystems. *Entities*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Data/Data_Modeling/Entities. 2020. (Visited on 09/08/2021) (cit. on p. 9).

[37] OutSystems. *Database Constraints*. https://success.outsystems.com/Documentation/11/Reference/OutSystems_Language/Data/Database_Reference/Database_Constraints. 2020. (Visited on 09/08/2021) (cit. on p. 9).

[38] OutSystems. *Aggregate a Column into a Single Value*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Data/Query_Data/Aggregate_a_Column_into_a_Single_Value. 2021. (Visited on 09/27/2021) (cit. on p. 10).

[39] OutSystems. *Calculate Values from Grouped Data*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Data/Query_Data/Calculate_Values_from_Grouped_Data. 2021. (Visited on 09/27/2021) (cit. on p. 10).

[40] OutSystems. *Create a Calculated Attribute in an Aggregate*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Use_Data/Query_Data/Create_a_Calculated_Attribute_in_an_Aggregate. 2021. (Visited on 09/27/2021) (cit. on p. 12).

[41] OutSystems. *Reuse and Refactor*. https://success.outsystems.com/Documentation/11/Developing_an_Application/Reuse_and_Refactor. 2020. (Visited on 09/24/2021) (cit. on p. 12).

[42] OutSystems. *Does OutSystems allow the reuse of code?* https://www.outsystems.com/evaluation-guide/does-outsystems-allow-the-reuse-of-code/. (Visited on 01/27/2021) (cit. on p. 13).

[43] P. Sebastião. *The New Architecture Dashboard: Diagnosis And Treatment of Technical Debt*. https://www.outsystems.com/blog/posts/architecture-dashboard/. 2020. (Visited on 12/17/2020) (cit. on pp. 13, 15).

[44] OutSystems. *Architecture Dashboard*. https://www.outsystems.com/platform/architecture-dashboard/. (Visited on 12/17/2020) (cit. on p. 14).

[45] OutSystems. *Introduction to Architecture Dashboard*. https://success.outsystems.com/Documentation/Architecture_Dashboard/Introduction_to_Architecture_Dashboard/. 2021. (Visited on 01/15/2021) (cit. on pp. 14, 17).

[46] OutSystems. *Code Analysis Patterns*. https://success.outsystems.com/Documentation/Architecture_Dashboard/Code_Patterns/. 2020. (Visited on 12/17/2020) (cit. on pp. 15, 17, 35, 36, 53).

[47] OutSystems. *OutSystems Platform Best Practices*. https://success.outsystems.com/Documentation/Best_Practices/Development/OutSystems_Platform_Best_Practices. 2021. (Visited on 02/25/2021) (cit. on p. 15).

[48] OutSystems. *How to use Architecture Dashboard*. https://success.outsystems.com/Documentation/Architecture_Dashboard/How_to_use_Architecture_Dashboard/. 2020. (Visited on 02/13/2021) (cit. on pp. 15, 17).

[49] R. Kellond. *What Is Technical Debt and How to Manage It*. https://www.outsystems.com/blog/posts/technical-debt/. 2020. (Visited on 02/13/2021) (cit. on pp. 16, 35, 36, 103).

[50] S. R. Browser. *The official website of the Smalltalk Refactoring Browser*. https://refactory.com/refactoring-browser/. (Visited on 02/08/2021) (cit. on p. 34).

[51] I. IDEA. *The official website of the IntelliJ IDEA*. https://www.jetbrains.com/idea/. (Visited on 02/08/2021) (cit. on p. 34).

[52] Eclipse. *The official website of Eclipse*. https://www.eclipse.org/. (Visited on 02/08/2021) (cit. on pp. 34, 47, 51).

[53] OutSystems. *Performance Best Practices - Queries*. https://success.outsystems.com/Documentation/Best_Practices/Performance_and_Monitoring/Performance_Best_Practices_-_Queries/. 2020. (Visited on 02/06/2021) (cit. on p. 58).

I

Refactoring Mechanisms Examples

Listing I.1: Code Before Extract Function (Amended From [1])

```
function printOwing(invoice) {
  // record due date
  const today = Clock.today;
  invoice.dueDate = new Date(
    today.getFullYear(),
        today.getMonth(),
        today.getDate() + 30);

  let owing = 0;
  // calculate owing
  for (const o of invoice.orders)
        owing += o.amount;

  // print details
  console.log(`name:${invoice.client}`);
  console.log(`amount:${owing}`);
  console.log(`due:${invoice.dueDate
    .toLocaleDateString()}`);
}
```

Listing I.2: Code After Extract Function (Amended From [1])

```
function recordDueDate(invoice) {
  const today = Clock.today;
  invoice.dueDate = new Date(
        today.getFullYear(),
        today.getMonth(),
        today.getDate() + 30);
}
function calculateOwing(invoice) {
  let result = 0;
  for (const o of invoice.orders)
        result += o.amount;
  return result;
}
function printDetails(invoice, owing) {
  console.log(`name:${invoice.client}`);
  console.log(`amount:${owing}`);
  console.log(`due:${invoice.dueDate
    .toLocaleDateString()}`);
}
function printOwing(invoice) {
  recordDueDate(invoice);
  const owing = calculateOwing(invoice);
  printDetails(invoice, owing);
}
```