



UNIVERSIDADE ESTADUAL DO CEARÁ
CENTRO DE CIÊNCIAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

MARCOS VINÍCIUS DE FREITAS BORGES

**CLOUD RESTRICTION SOLVER: A REFACTORING-BASED APPROACH TO
MIGRATE APPLICATIONS TO THE CLOUD**

FORTALEZA – CEARÁ

2017

MARCOS VINÍCIUS DE FREITAS BORGES

CLOUD RESTRICTION SOLVER: A REFACTORING-BASED APPROACH TO MIGRATE
APPLICATIONS TO THE CLOUD

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Orientador: Prof. PhD. Paulo Henrique Mendes Maia

FORTALEZA – CEARÁ

2017

Dados Internacionais de Catalogação na Publicação

Universidade Estadual do Ceará

Sistema de Bibliotecas

Borges, Marcos Vinicius de Freitas.

Cloud restriction solver: a refactoring-based approach to migrate applications to the cloud [recurso eletrônico] / Marcos Vinicius de Freitas Borges. - 2017.

1 CD-ROM: il.; 4 ¼ pol.

CD-ROM contendo o arquivo no formato PDF do trabalho acadêmico com 67 folhas, acondicionado em caixa de DVD Slim (19 x 14 cm x 7 mm).

Dissertação (mestrado acadêmico) - Universidade Estadual do Ceará, Centro de Ciências e Tecnologia, Mestrado Acadêmico em Ciência da Computação, Fortaleza, 2017.

Área de concentração: Ciência da Computação.

Orientação: Prof. Dr. Paulo Henrique Mendes Maia.

1. Evolução do software. 2. Migração para nuvem. 3. Refatoração. I. Título.

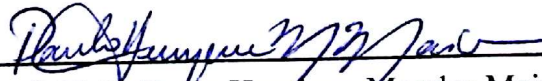
MARCOS VINÍCIUS DE FREITAS BORGES

CLOUD RESTRICTION SOLVER: A REFACTORING-BASED APPROACH TO MIGRATE
APPLICATIONS TO THE CLOUD

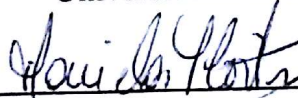
Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Aprovada em: 26/04/2017

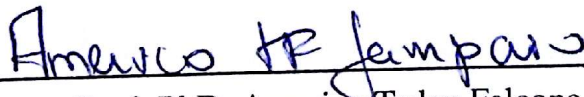
BANCA EXAMINADORA



Prof. PhD. Paulo Henrique Mendes Maia (Orientador)
Universidade Estadual do Ceará – UECE



Prof. Dra. Mariela Inés Cortés
Universidade Estadual do Ceará – UECE



Prof. PhD. Americo Tadeu Falcone Sampaio
Universidade de Fortaleza - UNIFOR

Dedico este trabalho a DEUS e a minha família por serem a base dos motivos para eu enfrentar e superar os mais diversos obstáculos, e por serem a grande verdade na minha vida.

AGRADECIMENTOS

Agradeço primeiramente a DEUS, por estar sempre presente em minha vida, por me dar fé para que este trabalho fosse concluído, por me ajudar a obter as respostas essenciais em momentos cruciais tanto durante o período do curso, quanto na vida.

Agradeço à minha família, representada pela minha avó Margarida, minha mãe Jesus, meu pai Salvador, minhas irmãs Angélica e Jéssica, por nunca deixarem de acreditar em mim e por sempre me darem forças para seguir em frente e vencer os mais diversos obstáculos durante essa caminhada.

Agradeço a minha noiva Márcia Cristina, por me ajudar nos momentos difíceis e complexos, por ter paciência nos momentos que não pude estar presente, por ouvir diariamente os mesmos assuntos repetitivos, e pelo incentivo e apoio para chegar a esta vitória.

Agradeço ao meu colega de pesquisa Erick Barros pela ajuda imprescindível durante esta caminhada. Agradeço também aos meus colegas de mestrado Anderson Couto, Flávio, Marcelo e Robson, pelos mais diversos momentos vividos durante o mestrado. Valeu pessoal, foi muito bom conhecer vocês, espero que essa amizade se mantenha mesmo pela inevitável distância.

Agradeço ao meu orientador Paulo Henrique, um exemplo de professor responsável, inteligente, dedicado, presente e disponível, que foi de fundamental importância para que eu conseguisse chegar ao resultado deste trabalho. Suas correções, questionamentos, debates e auxílios, influenciaram-me a superar minhas próprias expectativas e barreiras. Hoje posso dizer com muito orgulho a qualquer pessoa que tive um professor excepcional. Espero passar esse conhecimento, de qualidade, adiante. Mais uma vez, muito obrigado professor!

Agradeço à CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) pela concessão da bolsa durante todo o período de realização do mestrado.

Agradeço ao Mestrado Acadêmico em Ciência da Computação Universidade Estadual do Ceará (UECE), que por meio de professores altamente capacitados contribuiu de uma forma muito relevante para a minha aquisição de novos conhecimentos.

“O sucesso nasce do querer, da determinação e persistência em se chegar a um objetivo. Mesmo não atingindo o alvo, quem busca e vence obstáculos, no mínimo fará coisas admiráveis.”

(José de Alencar)

RESUMO

A migração de sistemas legados para o modelo Platform as a Service (PaaS) oferece vários benefícios, mas também traz novos desafios, como lidar com as restrições impostas pelo provedor de serviços. Além disso, fatores como tempo, treinamento e as extensas atividades de reengenharia tornam o processo de migração demorado e propenso a erros. Apesar de existirem várias técnicas para a migração parcial ou total de aplicações legadas para a nuvem, apenas algumas abordam especificamente a resolução dessas restrições. Este trabalho propõe uma nova abordagem semi-automática, chamada Cloud Restriction Solver (CRS), para a migração de aplicações para um ambiente PaaS evitando as restrições dessa nuvem através de refatorações definidas pelo usuário. A abordagem, que promove o reuso de *software* e é independente da nuvem, consiste principalmente de duas fases: identificação de restrições, que identifica os trechos de código que violam as restrições da plataforma PaaS escolhida e a execução da refatoração, que altera esses trechos por serviços equivalentes habilitados em nuvem. As fases são apoiadas por *engines* abertas e extensíveis, CRSAnalyzer e CRSRefactor, que constituem o *framework* CRS que implementa a abordagem. A aplicabilidade da abordagem CRS é feita através da ferramenta CRS4GAE (gerada pelo *framework* CRS baseado no PaaS Google App Engine (GAE)) em três aplicações web Java, que foram migradas com sucesso para o GAE.

Palavras-chave: Evolução do software. Migração para nuvem. Refatoração.

ABSTRACT

The migration of legacy systems to the Platform as a Service (PaaS) model provides several benefits, but also brings new challenges, such as dealing with the restrictions imposed by the service provider. In addition, factors such as time, training and the extensive reengineering activities make the migration process time consuming and error prone. Although there exist several techniques for the partial or total migration of legacy applications to the cloud, only a few specifically address the resolution of these constraints. This work proposes a novel semi-automatic approach, called Cloud Restriction Solver (CRS), for migrating applications to a PaaS environment that avoids the cloud restrictions through user-defined refactorings. The approach, which fosters software reuse and is cloud-independent, consists of two phases: *restriction identification*, identifies the pieces of code that violate the restrictions of the chosen PaaS platform, and *refactoring execution*, changes those pieces by equivalent cloud-enabled services. The phases are supported by open and extensible engines, CRSAnalyzer and CRSRefactor, which constitute the CRS framework that implements the approach. The applicability of the CRS approach is done through the CRS4GAE tool (generated by the CRS framework based on Google App Engine (GAE)) in three Java web applications, which have been migrated successfully to the GAE.

Keywords: Software evolution. Cloud migration. Refactoring.

LIST OF ILLUSTRATIONS

Figure 1 – Service Models	21
Figure 2 – Aspects in choosing a cloud model	23
Figure 3 – Cloud Reference Migration Model’s Processes	25
Figure 4 – 5-Phased Cloud Migration Model	25
Figure 5 – Two migration strategies to the cloud, cloud hosting and cloudification.	28
Figure 6 – Parse tree for listing 1	30
Figure 7 – The proposed approach overview	36
Figure 8 – CRSAnalyzer - Architecture.	38
Figure 9 – CRSAnalyzer - Execution flow	40
Figure 10 – CRSRefactor - Architecture	42
Figure 11 – Communication Class - Architecture	42
Figure 12 – CRSRefactor - Execution flow	43
Figure 13 – Google App Engine platform architecture	46
Figure 14 – <i>StackDriver</i> error report	47
Figure 15 – Using CRSAnalyzer in the application project.	56
Figure 16 – Report of violation of restricted classes of JAX-RS.	56
Figure 17 – Choose CRSRefactor refactorings.	57
Figure 18 – Report of violation of restricted classes of Servlet with Thread.	58
Figure 19 – Using CRSAnalyzer in the Pebble application project.	60
Figure 20 – Restrictions detected by classes.	60
Figure 21 – New abstract target and communication classes.	60

LIST OF TABLES

Table 1 – Challenges in migration of applications to the cloud	23
Table 2 – Correspondence between detection type and ASTNode type.	39
Table 3 – Restricted classes and its respective refactorings	48

LIST OF SOURCE CODES

Source Code 1 – Code excerpt - Simple java code example	29
Source Code 2 – Code excerpt - GAE	49
Source Code 3 – Code excerpt - GooglePaaSProvider	49
Source Code 4 – Code excerpt - CRSAnalyzer	49
Source Code 5 – Code excerpt - CRS4GAEFileRefactoring	50
Source Code 6 – Code excerpt - CRSFile	52
Source Code 7 – Code excerpt - CRS4GAEFile	52
Source Code 8 – Excerpt from the original code - JAX-RS	55
Source Code 9 – Refactored code excerpt - JAX-RS	55
Source Code 10 – Excerpt from original code - Java with Servlet	58
Source Code 11 – Refactored code excerpt - Java with Servlet	58

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
AST	Abstract Syntax Tree
Cloud-RMM	Cloud Reference Migration Model
CRS	Cloud Restriction Solver
CRS4GAE	Cloud Restriction Solver for Google App Engine
GAE	Google App Engine
IaaS	Infrastructure as a Service
ICT	Information and Communications Technology
IDE	Integrated Development Environment
IT	Information Technology
JSON	JavaScript Object Notation
NoSQL	Not Only Structured Query Language
PaaS	Platform as a Service
PHP	Hypertext Preprocessor
SaaS	Software as a Service
SDK	Software Development Kit
SLA	Service Level Agreement
SQL	Structured Query Language

SUMMARY

1	INTRODUCTION	15
1.1	STRUCTURE OF THE DISSERTATION	17
2	OBJECTIVES	18
2.1	GENERAL	18
2.2	SPECIFIC	18
3	THEORETICAL BACKGROUND	19
3.1	CLOUD COMPUTING	19
3.1.1	Features	19
3.1.2	Service Models	20
3.1.3	Deployment Models	22
3.2	MIGRATION OF APPLICATIONS TO THE CLOUD	22
3.2.1	Classification of migration studies	24
3.2.2	Service Level Migration	25
3.2.3	Migration Strategies	26
3.3	REFACTORING	27
3.4	CHAPTER SUMMARY	30
4	RELATED WORK	31
4.1	CLOUD MIGRATION APPROACHES	31
4.1.1	Manual Approaches	31
4.1.2	(Semi)Automatic Approaches	32
4.2	REFACTORING RECOMMENDATION	33
4.3	CHAPTER SUMMARY	34
5	CLOUD RESTRICTION SOLVER	35
5.1	OVERVIEW	35
5.2	CRS FRAMEWORK	36
5.2.1	Identification Engine	37
5.2.2	Refactoring Engine	39
5.3	CRS INSTANTIATION	43
5.4	CHAPTER SUMMARY	44
6	CRS4GAE	45

6.1	GOOGLE APP ENGINE	45
6.1.1	Restrictions	46
6.2	INSTANTIATION OF CRS FOR GAE	47
6.2.1	Identification Engine	48
6.2.2	Refactoring Engine	50
6.3	CHAPTER SUMMARY	53
7	USAGE EXAMPLE	54
7.1	JAX-RS FILE UPLOAD	54
7.2	SERVLET WITH THREAD	57
7.3	PEBBLE	59
7.4	CHAPTER SUMMARY	61
8	CONCLUSION	62
8.1	BENEFITS	62
8.2	LIMITATIONS	63
8.3	FUTURE WORK	63
	REFERENCES	64

1 INTRODUCTION

According to Buyya et al. (2009), the significant evolution of the Information and Communications Technology (ICT) in the last half of the 20th century, computing can be considered the fifth basic utility need by the population, as well as the other four indispensable services: water, electricity, gas and telephony. From that vision, several paradigms have been proposed among which the cloud computing, a technology trend of the 21st century that leads to an evolutionary path to provide better responses to current and future ICT requirements, can be highlighted (BUYYYA et al., 2009)(PUTHAL et al., 2015).

Cloud computing has the potential to transform part of the Information Technology (IT) industry, making software even more attractive as a service (ARMBRUST et al., 2010). It differs from traditional paradigms, since it brings certain benefits such as: high scalability and availability; different levels of services to customers, even outside the cloud; payment for use, which makes it oriented to the scaling economy; its services can be dynamically configured and delivered on demand (FOSTER et al., 2008)(SOSINSKY, 2010), and provided to external customers over the internet (MELL; GRANCE, 2011).

The services provided by the cloud, ranging from hardware to software, are specific to each vendor and can be summarized in the SPI model: *Software as a Service* (SaaS), the highest service level closer to the end-user that provides applications on demand, generally in multi-tenant¹ to the clients through the Internet; *Platform as a Service* (PaaS), which provides a platform with resources such as programming languages, tools, and frameworks in order to support the software development process (RIMAL et al., 2009)(PUTHAL et al., 2015); and *Infrastructure as a Service* (IaaS), the lowest level that provides for the client infrastructure technology like virtual machines, operating systems, network and other fundamental resources (RIMAL et al., 2009)(SOSINSKY, 2010).

In this realm, many IT companies are considering the cloud as an excellent opportunity to their business, mainly due to the possibility of creating new applications entirely based on that new technology (BUYYYA et al., 2009)(TRAN et al., 2011) or migrating legacy systems to a cloud platform (ZHAO; ZHOU, 2014)(MAENHAUT et al., 2016). According to (GARTNER, 2017), the public cloud global market will reach approximately US\$ 247 billion still in 2017, representing an expansion of 18% over the total raised in 2016. This trend may continue increasing until 2020 since more companies shall move their current systems to the

¹ A *software* instance that can be shared with two or more clients (MARSTON et al., 2011)

cloud.

There are three types of migration of legacy systems, considering the cloud service models (ZHAO; ZHOU, 2014): (i) migration to IaaS, in which the application is moved to a virtual machine that loads the software and its components, such as databases; (ii) migration to PaaS, in which the application has to be refactored according to the target platform to conform with its restrictions; and (iii) migration to SaaS, in which the legacy application undergoes a reengineering process or replacement of their services to similar ones available in the cloud. Among those types, the migration to PaaS is the one that offers more technical challenges, since a great effort on understanding the involved technologies and cost estimation is required, as well as the existence of limitations and restrictions imposed by the cloud service provider (MOHAGHEGHI; SÆTHER, 2011)(VU; ASAL, 2012), such as file writing, thread execution and socket access. This work focuses on that kind of migration.

From the developer's point of view, migrating legacy applications to a PaaS provider manually is a time-consuming and error-prone task, since it can demand special training in the chosen PaaS platform API and may need changes in the application source code to overcome the cloud environment restrictions (KWON; TILEVICH, 2014). To tackle that, refactoring emerges as an interesting and possible solution, since it relies on a series of small code transformations that maintain the observable system behavior (FOWLER; BECK, 1999). According to (KWON; TILEVICH, 2014), refactorings can be used to facilitate the transition of an application to the cloud because the application semantics is preserved and its features in general do not change, even though it now runs in another environment.

Although there are several techniques to migrate partially or totally legacy applications to the cloud (VU; ASAL, 2012; TRAN et al., 2011; MAENHAUT et al., 2013; MAENHAUT et al., 2016; COSTA et al., 2015; FREY; HASSELBRING, 2011; PRABHAKARAN, 2014; VASCONCELOS et al., 2015), even using refactoring (KWON; TILEVICH, 2014), just a few focus in the PaaS migration, particularly addressing the cloud provider restrictions. Such solutions are important since they help the user to identify the pieces of code that will not run in the cloud and automatically change them to behaviour equivalent ones, thus easing and boosting the migration process.

This work proposes a novel semi-automatic approach, called Cloud Restriction Solver (CRS), to aid the migration of legacy applications to a PaaS environment that avoids the possible cloud restrictions by using user-defined refactorings. The approach, which is cloud-independent,

consists of two phases: *restriction identification*, which detects the pieces of code that violate the target PaaS restrictions, and *refactoring execution*, in which the developer implements the refactorings that replace that pieces of code by cloud-equivalent services. The phases are supported by open and extensible engines, *CRSAnalyzer* and *CRSRefactor*, respectively. These engines constitute the CRS framework, which implements the approach.

1.1 STRUCTURE OF THE DISSERTATION

The dissertation is structured in six chapters, including the Introduction, which are described below.

Chapter 2, the objectives of this work are defined.

Chapter 3, named Theoretical Background, presents the main concepts used in this work, which are: cloud computing (the ground context of the dissertation), migration of applications to the cloud (the specific context of the study) and refactoring (proposed technique applied to the specific context).

Chapter 4, Related Work, discusses the main work found in the literature related to this dissertation. Those papers have been divided in two categories: cloud migration processes (manual and semi/automatic) and refactoring recommendation.

Chapter 5 describes in detail the approach of this work: Cloud Restriction Solver. These details demonstrate the overview of the approach, the framework that implements it, along with its intrinsic aspects (identification and refactoring engines), and a general instantiation process to be applied to a given cloud target.

Chapter 6, CRS4GAE, shows the instantiation of CRS to the Google App Engine (GAE) cloud. In that chapter it is highlighted how to extract the GAE information (restrictions) that serve as a basis for the use of the CRS framework, which will generate the CRS4GAE tool made up of the identification and refactoring engines based on the GAE environment.

In chapter 7, Usage Example, the applicability of CRS4GAE is demonstrated by describing the migration of three java web applications using the CRS4GAE tool. The main results are highlighted, and the links of the applications migrated to cloud and their codes are made available.

Finally, Chapter 8 presents the conclusions of this work, its contributions, limitations and possible future work.

2 OBJECTIVES

2.1 GENERAL

The main objective of this work is to define a semi-automatic approach for migrating applications to a given cloud PaaS platform, by detecting in the application code possible violations of the cloud constraints and performing their corrections through refactorings.

2.2 SPECIFIC

To achieve the general objective of this work, the following specific objectives have been defined:

- a) Define the approach's general workflow.
- b) Create a framework that implements the phases of identification and refactoring of the CRS approach.
- c) Provide a general step-by-step instantiation process of the CRS framework for any cloud
- d) Instantiate the CRS framework for a specific cloud to generate a tool that analyzes and refactors application codes that violate constraints on that cloud.
- e) Demonstrate the applicability of the CRS approach through that tool generated in three java web applications.

3 THEORETICAL BACKGROUND

This chapter presents the main concepts that give the basis for the approach of this work and is divided into three sections: (i) cloud computing main concepts and features, (ii) migration of applications to the cloud, and (iii) refactoring.

3.1 CLOUD COMPUTING

Cloud computing represents a fundamental change in the way the IT services are being invented, developed, deployed and maintained (MARSTON et al., 2011). The use of cloud computing brings potential benefits to organizations in general (primarily for small businesses), including greater flexibility and efficiency, in addition to the cost reduction of its service delivery, reducing the traditional support requirements and the fixed financial expenses commitment.

Since Google's chief executive, Eric Schmidt, expressed the term "Cloud Computing" in 2006, a number of IT organizations started to have cloud-based projects. However, the definitions of the term were quite scattered and divergent (FOWLER; WORTHEN, 2009). To demonstrate this, Vaquero et al. (2008) conducted a study in which they pointed out 22 definitions of cloud computing. Despite the expressive number, it is still possible to find more definitions about that term.

Among the main definitions are those of authors (FOSTER et al., 2008) and (MELL; GRANCE, 2011), whose ideas can be summarized in the following concept: Cloud Computing is a paradigm that aims to allow the access, in a suitable way and by demand, to a shared pool of configurable computational resources (network, servers, storage, applications, services) that can be quickly supplied and retrieved with the minimum effort from external clients through the internet.

According to Mell & Grance (2011), cloud computing is still composed of essential features, service models, and deployment models.

3.1.1 Features

The characteristics of the cloud can be treated as the crucial aspects that differentiate it from the more traditional paradigms. Those aspects are benefits constituted by a set of attractive technologies such as (MELL; GRANCE, 2011):

- **On-demand self-service:** a customer can automatically provision computing resources

without the need for interaction with each service provider.

- **Broad network access:** availability of resources that can be accessed by platform-independent devices (tablets, mobile phones, workstations) via internet.
- **Resource pooling:** the provider's physical and virtual computing resources are organized to serve multiple clients using a multi-tenant model. These features (such as storage, processing, bandwidth) are location-independent for the consumer, and can be dynamically allocated and deallocated according to their needs.
- **Rapid elasticity:** resources can be provisioned and released, sometimes automatically, so that the application is always in accordance with the customer demand. From the customer point of view, these features appear to be limitless and can be ordered in any quantity at any time.
- **Measured service:** cloud resources can be monitored, optimised and controlled through a measurement at some level of abstraction. So there is the guarantee of transparency for both providers of services to requesting clients in relation to the contracted services.

In addition to those essential characteristics, other benefits can also be highlighted, such as: (i) low costs and facility to adhere to the model, since the initial investments are greatly reduced, allowing any company to have a potential high growth; (ii) reliability, providing load balancing and the ability to handle failures by automatic replacement of unavailable services; (iii) maintenance and upgrade, since the customer can request upgrade at any time, which is usually based on the latest versions of the services.

3.1.2 Service Models

The service models consist of the SPI model (SaaS, PaaS, and IaaS) described in Chapter 1. Figure 1 illustrates the services model in more details. In the SaaS model, access is accomplished through a client interface, usually a web browser. In that model the client has no control and cannot discover the underlying software structure (operating systems, servers, network). However, the problems that happen in this structure are responsibility of the supplier (SOSINSKY, 2010; PUTHAL et al., 2015). Among the main examples of SaaS there can be mentioned: word processor (Google Docs¹), music streaming (Spotify²), disk storage (Dropbox³,

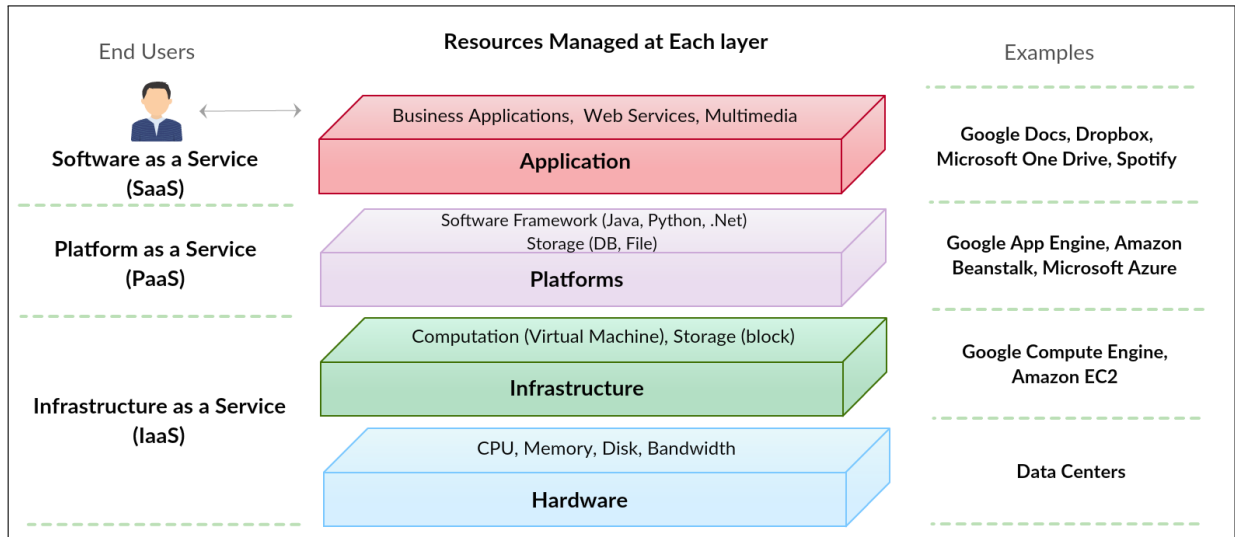
¹ available at <<https://www.google.com/docs/about/>>

² available at <<https://www.spotify.com/>>

³ available at <<https://www.dropbox.com/>>

Microsoft OneDrive⁴).

Figure 1 – Service Models



Source: Adapted from Zhang et al. (2010).

PaaS aims to provide to developers a platform that includes systems and environments that are committed with the entire lifecycle of software development, in addition to its hosting (RIMAL et al., 2009). At that level of service, the customer does not worry about managing the underlying hardware layers, being only responsible for the application's settings and deployment (PUTHAL et al., 2015). However, in that environment there are many limitations and restrictions that may vary like programming languages, databases, third-party components among others (VU; ASAL, 2012). Some of the main PaaS environments are: Google App Engine⁵, Amazon Elastic Beanstalk⁶, IBM Blue Mix⁷ and Microsoft Azure⁸.

Finally, the IaaS model provides infrastructure resources that the customer may need (SOSINSKY, 2010). At this level of service, the client has control of most of those features and also of some restricted parts of the administration system. With those privileges it is possible to deploy and run any software, since that environment does not impose restrictions on applications (PUTHAL et al., 2015). Among the main infrastructure providers are: Compute

⁴ available at <<https://onedrive.live.com/>>

⁵ available at <<https://appengine.google.com/>>

⁶ available at <<https://aws.amazon.com/pt/elasticbeanstalk/>>

⁷ available at <<https://console.ng.bluemix.net/>>

⁸ available at <<https://azure.microsoft.com/>>

Engine⁹, Amazon EC2¹⁰, GoGrid¹¹, Linode¹².

3.1.3 Deployment Models

Deployment models refer to the management and localization of the cloud infrastructure, as well as how its resources are shared and viewed (SOSINSKY, 2010). Jadeja & Modi (2012) and Mell & Grance (2011) classify these models into four types:

- **Public:** consists of the standard model of cloud computing, in which a service provider offers resources, such as storage and applications, to the general public via the web. However, these clouds are less secure and susceptible to malicious attacks compared to others.
- **Private:** unlike the public cloud, it allows access for users who manage the resources and applications within a proprietary organization. The main advantage is to manage the security, control and maintenance of the data.
- **Community:** in this model there is shared cloud infrastructure between various organizations that have common interests (safety requirements, jurisdiction, policies).
- **Hybrid:** composition of two or more distinct clouds (private, community and public) that remain as single entities, but which remain united by the standardization that enables the portability of applications and access to data.

3.2 MIGRATION OF APPLICATIONS TO THE CLOUD

Given the many existing cloud models (services and deployment), some aspects such as control, flexibility, capacity and level of abstraction should be considered when choosing a specific cloud environment (see Figure 2) to allocate a certain software. Once the cloud model is chosen, the software to be allocated in the cloud can be developed from scratch or adapted. The first option falls under the term *cloud adoption*, which considers that the whole process of software development has to be based on the current features and technologies of the cloud. The second one (adopted in this work) refers to the *cloud migration*, in which existing software present in the most diverse organizations can be adapted to the cloud (MENDONÇA, 2014).

Migration of applications to the cloud still involves several challenges that, according

⁹ available at <<https://cloud.google.com/compute/>>

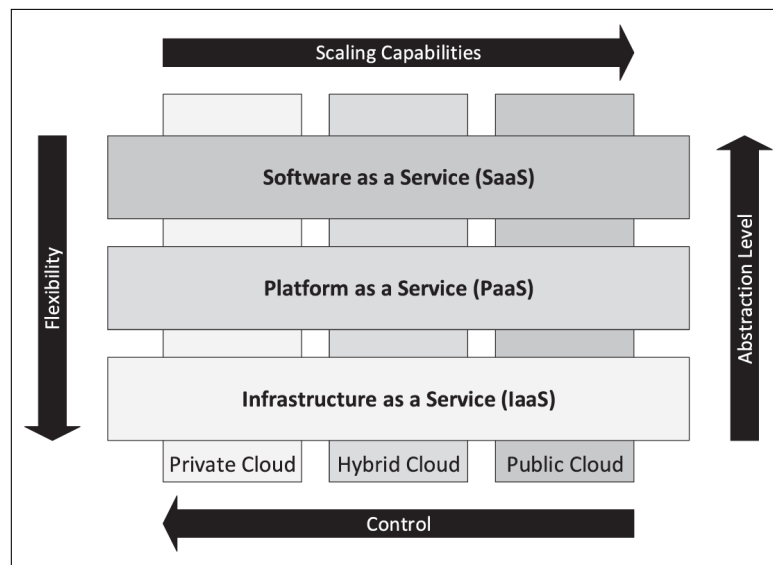
¹⁰ available at <<https://aws.amazon.com/ec2/>>

¹¹ available at <<https://www.datapipe.com/gogrid/>>

¹² available at <<https://www.linode.com>>

to Rai et al. (2013), can be classified into business and technical factors, shown in Table 1. Among them, IT training can be highlighted, which usually addresses upgrading IT professionals’ skills in architecture, implementation and development of cloud applications. Additionally, Scandurra et al. (2015) report that another challenge to be considered is the vendor lock-in - tying a technology coming from a vendor’s particular service or product - with respect to portability and interoperability of applications in general in the cloud, which directly interferes in the selection of other clouds.

Figure 2 – Aspects in choosing a cloud model



Source: (MAENHAUT et al., 2016).

Table 1 – Challenges in migration of applications to the cloud

Migration Challenges	Description
Business Factors	Costs
	Existing investments in IT
	Data security
	Regulations
	Provisioning
Technical Factors	Existing infrastructure
	Architecture
	Complexity
	Network and support
	IT skills
	Service Level Agreement (SLA)

Source: (RAI et al., 2013).

Thus, during the migration process, IT professionals may find problems regarding

non-compliance between applications and cloud platforms. As pointed out by Frey et al. (2013), not every cloud environment is suitable for a software system in the sense that it can impose constraints on the applications they host. Mendonça (2014) reports that after a migration to the cloud, a component must be able to resolve its dependencies on interactions with external services. In this way, developers must adapt or change the affected components such that they will comply with the cloud through the use of services offered in this new environment.

3.2.1 Classification of migration studies

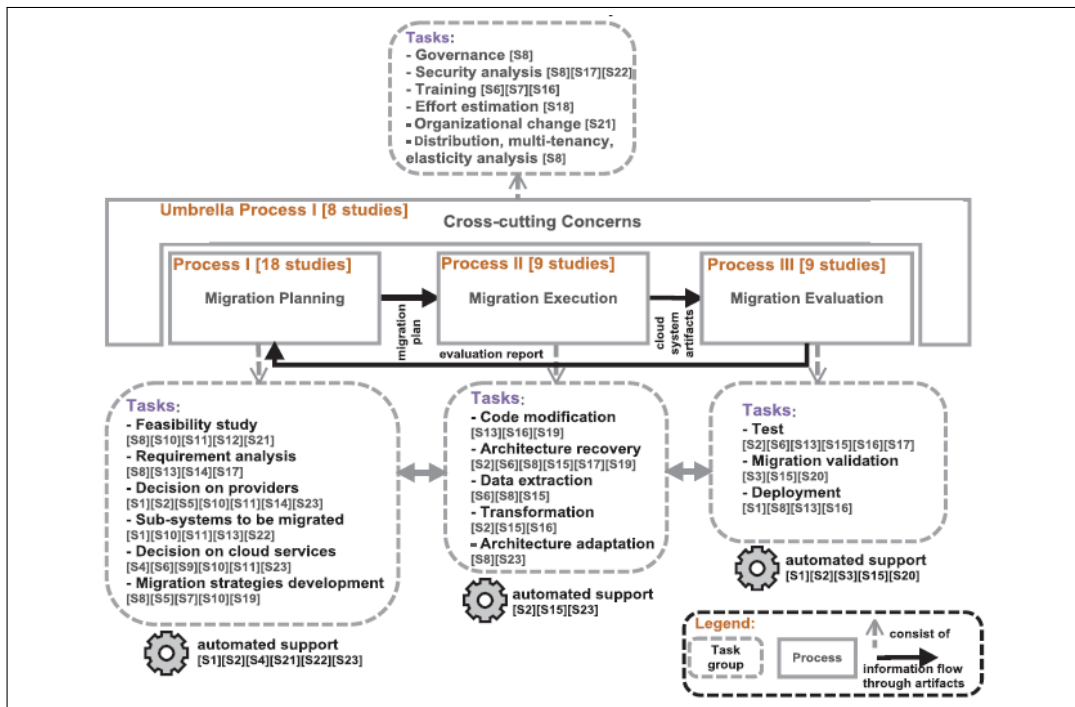
Jamshidi et al. (2013) and Rai et al. (2015) conducted systematic reviews in order to classify the main studies in the area of migration to the cloud and to guide the research on this topic.

In the review of Jamshidi et al. (2013), the authors classified the studies into a reference model called Cloud Reference Migration Model (Cloud-RMM), dividing them into four processes that can be seen in Figure 3. The first process (I) consists of a *Migration Planning*, which covers feasibility studies, requirements analysis, decisions about the cloud provider, which parts of the system will be migrated, the cloud services that will be used, and the migration strategy. The second one (II) deals with the *Migration Execution* and involves tasks such as data extraction, architecture readjustment, application adaptation to the cloud, code modifications and transformations at both conceptual and physical levels of the legacy application. The third process (III), *Migration Evaluation*, includes testing, validation and deployment of migrated systems. Finally, the fourth process (IV) deals with *Cross-cutting Concerns*, which regards IT governance, security analysis, effort estimation, and organizational change analysis.

In the systematic review of Rai et al. (2015), the authors classify the studies in a conceptual model called *5-Phased Cloud Migration Model*, illustrated in Figure 4. These five phases (feasibility analysis, requirements analysis and migration planning, migration execution, testing and migration validation and fifth, monitoring and maintenance) deal with the subjects of the Cloud-RMM model's processes.

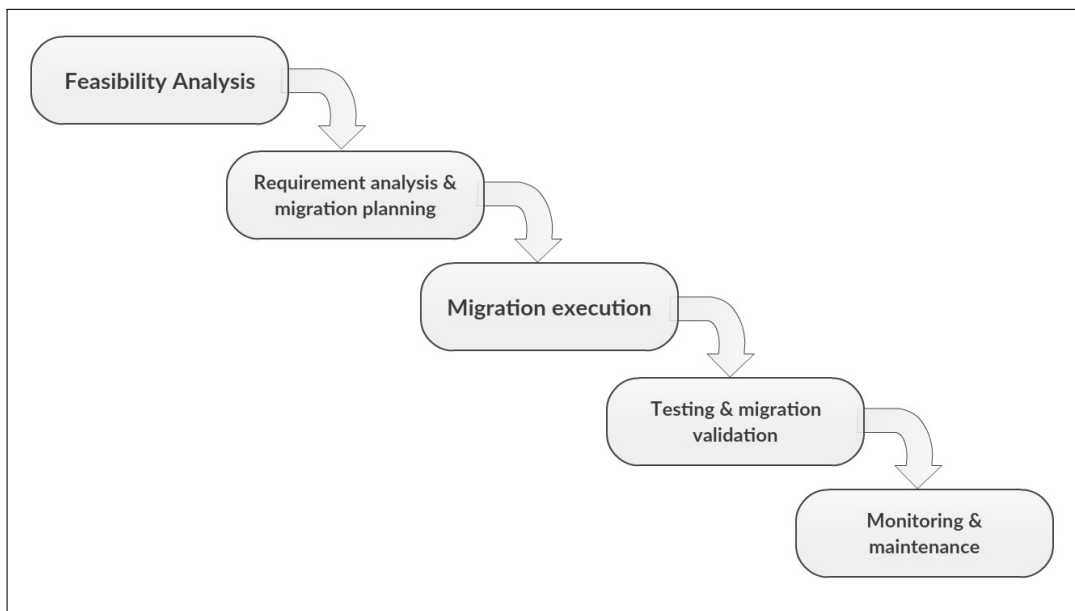
This work fits into the migration execution in both reviews, more precisely in the code modification.

Figure 3 – Cloud Reference Migration Model’s Processes



Source: (JAMSHIDI et al., 2013).

Figure 4 – 5-Phased Cloud Migration Model



Source: (RAI et al., 2015).

3.2.2 Service Level Migration

Zhang et al. (2010) and Zhao & Zhou (2014) have carried out a survey about the main strategies for cloud migration considering the three service models: IaaS, PaaS e SaaS.

In the first category, legacy systems are migrated to an IaaS service, but despite its

good cost benefit, the migration is not likely to take full advantage of the cloud's capabilities. Generally that category is easier and more used by companies. In the second category, systems are migrated to a PaaS service usually through code modification or transformation techniques. However, the adaptation may lead to problems due to the changes in the system code. This is the most complicated migration type because as there is adaptation of the application to the target cloud, it brings serious disadvantages such as *vendor lock-in* tying, risk in the transition of the application and, finally, the qualification of the IT professionals (cited previously, section 3.2). Finally, the migration to SaaS aims to perform a complete or partial replacement of a system by existing cloud services. Depending on how the software is migrated, there may or may not be reengineering efforts, redesign, and service generation.

This work addresses the migration of applications to the PaaS.

3.2.3 Migration Strategies

Despite the benefits of the cloud, some systems can not be migrated to this type of environment due to their specific characteristics (eg, critical systems). Others can be deployed natively to the cloud (*cloud-native*), and there are still those that need adaptations to be able to run in the cloud (*cloud-enabled*) (ANDRIKOPOULOS et al., 2013).

Andrikopoulos et al. (2013) have proposed the following categories to classify the types of migration to the cloud considering the different adaptation possibilities:

1. *Replacement*: this type of migration replaces one or more legacy components by services in the cloud. This is the least invasive of all types and requires data or business layers to be migrated to a cloud service. The migration process is accomplished by reconfiguring the application components to remove incompatibilities, thereby enabling them to use the cloud features.
2. *Partial Migration*: in this process it is performed only the partial migration of the application components to the cloud, which means that other parts of the application still remain in the original non-cloud environment.
3. *Whole stack migration*: here the entire application is encapsulated in one or more virtual machines, which are already running in the cloud. This process is also non-invasive in the sense that only few changes need to be made in the application to perform the migration.
4. *Cloudify*: Another example of whole stack migration, in which the application is turned into a cloud-enabled system through the composition of cloud services.

In the same direction, Mendonça (2014) defines two strategies for migration: *cloud hosting* and *cloudification*. In the former, which refers to hosting components in the cloud, there are four solutions: (i) *rebinding*, which is the simplest way to deploy an object in the cloud by simply having the original target component being compatible with the cloud component and being able to resolve external service dependencies after it has been migrated; (ii) *service adaptation*, which is required when there is incompatibility or constraints imposed by the cloud regarding the communication of the migrated component with the external services, so service adapters need to be used, one in the external component and the other in the cloud, to allow communication; (iii) *service conversion*, which is a more flexible alternative to service adaptation, but in this case, instead of using adapters, developers intrusively implement the necessary adaptation in the components so that they use the new cloud-compliant communication service; (iv) *compensation*, is a way to change the target component (after migration) so that it can compensate for any component that can not interact with it, because there is no practical way to do so.

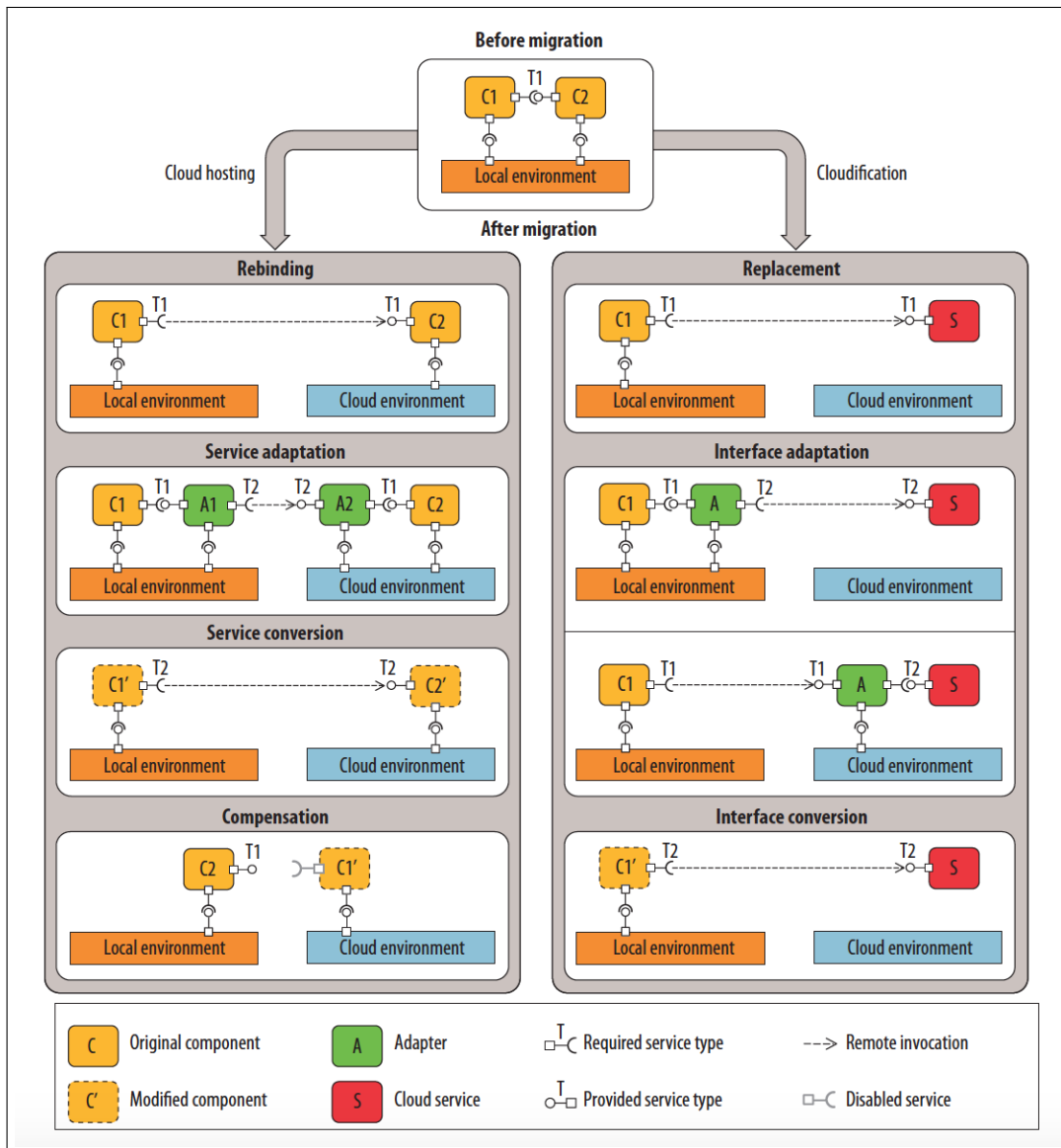
In the *cloudification* strategy, which aims to replace some components of the application by their cloud counterparts with similar services, there are three solutions: (i) *replacement*, which is the simplest form, but the target component and the target cloud service must have identical or fully compatible interfaces; (ii) *interface adaptation*, when the target component has similar functionality to the cloud service, but with an incompatible interface, the solution would be to reconfigure the components by using adapters (client-side or cloud-side) for the correct use of that cloud service; (iii) *interface conversion*, which is similar to the previous one, with the proviso that instead of adapters, developers implement the necessary interface directly into the source code of the target component, so that it natively accesses the cloud service.

In this work, it is used the *cloudification* strategy, more specifically the interface conversion solution, by directly modifying the component interface using the Adapter design pattern (GAMMA, 1995) to access the PaaS cloud similar service.

3.3 REFACTORING

According to Pressman & Maxim (2016), the software is subject to change mainly due to corrections of errors, new features requested by users, and adaptations to new environments. Considering the cloud as that environment, a solution to this environmental adaptation is the use of refactorings. In (KWON; TILEVICH, 2014), the authors advocate the use of refactorings to

Figure 5 – Two migration strategies to the cloud, cloud hosting and cloudification.



Source: (MENDONÇA, 2014).

ease the migration of an application to the cloud, since the application behaviour is preserved, even though some features are modified to rely on cloud-based services.

Refactoring, as defined by Fowler & Beck (1999), is a change made in the internal structure of a software in order to make it easier to understand and less costly to modify. Its essence consists of performing a series of small transformations without altering the observable behavior of the system. It can be classified in two main groups: *primitive* refactorings, an atomic modification that cannot be split into two or more small refactorings and for which a set of preconditions is defined to preserve the behaviour, and *composite* refactorings, composed by a sequence of primitive refactorings in which the preconditions of all refactorings need to be

satisfied (SAADEH et al., 2009). In (FOWLER; BECK, 1999) the authors present an extensive catalog of refactorings, from where can be highlighted:

- Rename attribute/method/class: this refactoring can be applied when the name of an attribute, method, or class does not reveal its real purpose. The solution is simply renaming the artifact to improve the understanding.
- Move method: when a class uses more features of another class than of its own, this refactoring is applied to move the method from one class to the other. This action also decreases the coupling between the classes.
- Extract method: this refactoring is used when a code fragment can be grouped into a well-named method in order to provide better understanding. This technique can also be applied in methods that are too long and need to be grained.
- Extract superclass: it can be applied when there exists more than one class with similar features, since this often leads to repeated code. The solution here is to create a superclass and move the common features to it.
- Pull up method: when there are two subclasses that have methods with identical results, this refactoring can be used to move the method to a superclass, thus removing the duplicated behaviour.

The most appropriated way to apply the code refactorings is using an automated tool. Fowler & Beck (1999) report that the purpose of such tool is the acceleration of the refactoring process in software, since developers would not waste time in retesting the software at each code change. Fowler & Beck (1999) still emphasizes that a refactoring tool should be simple, fast, easy to use and preserve the behavior of the system. One of the main ways to analyze and transform the codes is considering it as a *tree*.

In the *tree*, the application code is represented by nodes generated by a parser¹³, thus easing the search of source code snippets (FOWLER; BECK, 1999). An example can be seen in listing 1 (lines 1 to 3), which is a java code excerpt that aims to display a “Hello World”. The representation of this listing in the form of tree can be viewed in Figure 6.

Source Code 1 – Code excerpt - Simple java code example

```

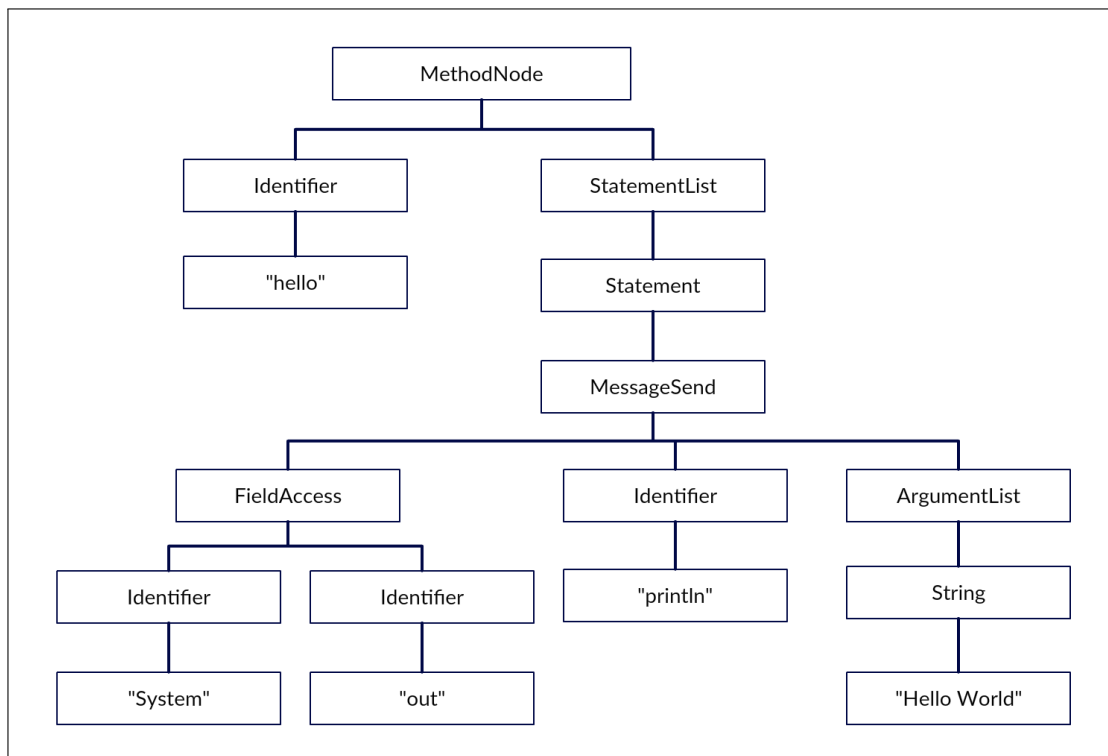
1 public void hello() {
2     System.out.println("Hello World");
3 }

```

¹³ Representation of the inherent syntactic structure of a program (AHO; ULLMAN, 1972)

Beyond the search, the tree also allows to create new nodes (PaaS cloud-based codes), to remove nodes (codes that violate restrictions in the cloud), and update nodes (codes that violate restrictions by new cloud creators), which is of fundamental importance for the approach of this work that evolves the software to other cloud environments.

Figure 6 – Parse tree for listing 1



Source: (FOWLER; BECK, 1999).

3.4 CHAPTER SUMMARY

This chapter presented the three main concepts used in this work. Regarding the first one, cloud computing, its main concepts, features and models (service ans deployment) have been described. The second one, migration of applications to the cloud, showed the main challenges, the classification of the studies in this topic and the main strategies used to enable applications to the cloud. Finally, in the last discussed concept, refactoring, main concepts and examples have been presented.

In the next chapter the main work related to this dissertation are discussed.

4 RELATED WORK

In this chapter the main related work are discussed. They are divided in two categories: techniques for migrating legacy applications to the cloud and refactoring recommendation.

4.1 CLOUD MIGRATION APPROACHES

Cloud migration can be divided in manual and (semi)automatic approaches. The former concerns from the feasibility planning to the actual migration of applications and legacy databases to the cloud (VU; ASAL, 2012; TRAN et al., 2011; MAENHAUT et al., 2013; MAENHAUT et al., 2016; COSTA et al., 2015), while the latter is related to tools that support and execute the application migration (FREY; HASSELBRING, 2011; KWON; TILEVICH, 2014; PRABHAKARAN, 2014; VASCONCELOS et al., 2015). As the proposed approach of this work also encloses a manual and an automatic phase, some work from those two migration process types are discussed in the following sections.

4.1.1 Manual Approaches

The authors in (VU; ASAL, 2012) present some aspects (restrictions, limitations, requirements) that have to be considered in the questions inherent to migration of legacy applications to the cloud. More specifically, in that work it is demonstrated a compatibility checklist (languages, components, and databases) for those applications. Finally, they describe a manual migration process of three applications, one of them developed in Java and the others in Python, for the Google App Engine (GAE) aiming at evaluating the feasibility, costs, solutions and efforts of the migration.

In the same direction, the work in (TRAN et al., 2011) proposes a taxonomy for the tasks performed in the migration to the cloud. The authors report their experience and technical effort during the migration of a .NET application for the Windows Azure platform, and also determine the costs of those tasks. During the process, the authors describe the necessary steps both to migrate that application to the cloud and to add multi-tenancy to it. The evolution of the work is presented in (MAENHAUT et al., 2016) and follows the same logic of the previous one, but in a more generic way covering another application and more hybrid and public cloud environments. In that new work there are also some considerations about the costs, eminent risks and benefits of the migration process.

In (COSTA et al., 2015) there is an experience report about the partial migration of two legacy applications in which its local relational database has been migrated to a NoSQL database in the cloud (Amazon DynamoDB) that aimed to solve performance problems. The applications, one web and the other one a standalone, have been adapted to use cloud services through aspect-oriented programming and the Groovy metalanguage feature.

Different from the other work, in the manual phase of the CRS, the developer has to identify the restricted classes for the target PaaS and proposes refactorings to avoid those restrictions. Hence, it is the developer's responsibility to make sure that the migration is feasible and worthwhile and, if so, he/she can proceed to the next phase.

4.1.2 (Semi)Automatic Approaches

CloudMig (FREY; HASSELBRING, 2011) is a model-driven semi-automatic approach that aims at assisting engineers in the process of migrating legacy systems to the cloud, reducing the complexity of alignment between those systems and the target environment at the level of IaaS and PaaS. More specifically, the approach generates models of the legacy system in order to compare them to the models of the target cloud environment and their restrictions. Nonetheless, it does not perform the migration, but only alerts for the possible restrictions that can occur regarding a cloud profile. Moreover, the available tool does not work properly. Similarly, the identification engine of this work proposes such alerts by specifying the errors and classifying them according to the restriction found.

The authors in (VASCONCELOS et al., 2015) propose a non-intrusive automatic approach based on events called Cloud Detours. That approach establishes a set of detours that can be manipulated through libraries (e.g, I/O and database detours) by developers such that the features of on-premise legacy applications can be replaced by compatible cloud services without making any changes in the original application's source code by using aspect-oriented programming. Although the proposed approach is intrusive, i.e, it directly modifies the application code, it requires less effort and simpler computational tools to perform the migration. Moreover, in (VASCONCELOS et al., 2015), the decision about which features to migrate is taken empirically by the developer, while that task is supported by a restriction identification tool in the CRS approach.

A process for the migration of JEE web applications that use relational databases to the Google App Engine platform is presented in (PRABHAKARAN, 2014). The process uses

two tools: the Java Source Code Analyzer and the Database Migration Tool, which migrates the data from relational database to the Cloud Datastore. Different from the proposal of this work, that work does not address code modification to enable the application to run in the cloud, but rather focuses on migrating the database data to its equivalent service in the GAE.

In (KWON; TILEVICH, 2014) it is proposed a semi-automatic and intrusive approach that uses refactoring techniques to help developers in the process of partially migrating on-premise applications to the cloud. The approach, called Cloud Refactoring, is implemented using the Eclipse IDE and contains two engines: recommendation and refactoring. Similar to the work of this dissertation, Cloud Refactoring enables replacing features of on-premise applications to equivalent ones in the cloud. However, this dissertation focuses on features that violate restrictions imposed by a cloud environment. This means that, by using approach of this work, the developer can identify in advance which features will not work in the cloud and then concentrate on solving those problems by implementing specific refactorings.

4.2 REFACTORING RECOMMENDATION

Several work that deal with refactoring recommendation can be found in the literature. A recommendation system whose main purpose is to provide refactoring guidelines to aid developers to remove architectural erosion in the software is presented in (TERRA et al., 2012). The authors claim that the recommendations made by other tools are too generic, while their approach gives more details about the process. In (SZŐKE et al., 2015; FOKAEFS et al., 2011) are presented tools, available as Eclipse plugins, that allow the identification of code smells and automatically apply the necessary refactorings. This dissertation is related to those work in the sense that it also recommends the necessary refactoring but when a given restriction is detected in the code. Additionally, it is also distributed as an Eclipse plugin to facilitate its use.

Other tools in the literature have more specific approaches to select the recommended refactorings. In (MKAOUER et al., 2014) it is proposed an approach to suggest refactorings to developers based on their feedback and code changes that uses a multi-objective evolutionary algorithm. An approach to identify Extract Method refactoring opportunities that uses a ranking function to classify the generated candidates is presented in (SILVA et al., 2014). This dissertation differs from those ones in the sense that the refactorings are selected according to the cloud restrictions found in the application source code. Furthermore, CRS not only recommends the refactoring, but also is able to apply it to the code.

4.3 CHAPTER SUMMARY

This chapter presented the main work found in the literature related to this dissertation. These work were divided into: approaches for migrating applications to the cloud (either manual or semi-automated); and refactoring recommendation. It could be seen that some other pieces of work also deal with refactoring applications to the cloud, but none of them apply the modifications in order to avoid the cloud platform restrictions, which could impede the system to run in the cloud.

The next chapter will present the approach proposed in this work, Cloud Restriction Solver (CRS), as well as its main details and intrinsic aspects.

5 CLOUD RESTRICTION SOLVER

The migration of legacy applications to the cloud still has many challenges to be surpassed, particularly in the PaaS level, since there are limitations and restrictions in the environments offered by the cloud providers. In addition, technical perspective factors like time, training and extensive activities of reengineering, make harder the work of developers in charge for the migration process. The proposed approach takes those questions into account and the importance of emerging new techniques as highlighted in the last systematic reviews of the literature (JAMSHIDI et al., 2013)(RAI et al., 2015).

This work proposes a semi-automatic approach, called Cloud Restriction Solver (CRS), to tackle the problem of migrating legacy applications to the cloud by identifying and replacing the application pieces of code that violate restrictions on a given PaaS environment using user-defined refactorings. The approach, which is cloud-independent and, therefore, can be applied to different PaaSs, is implemented by a framework that contains engines that automate the identification of constraints and the code refactoring application processes. The engines's core architecture can be extended and manipulated by the developers according to their need. The next sections present: the overview of the CRS, a framework that implements it, and a step-by-step guide to its instantiation.

5.1 OVERVIEW

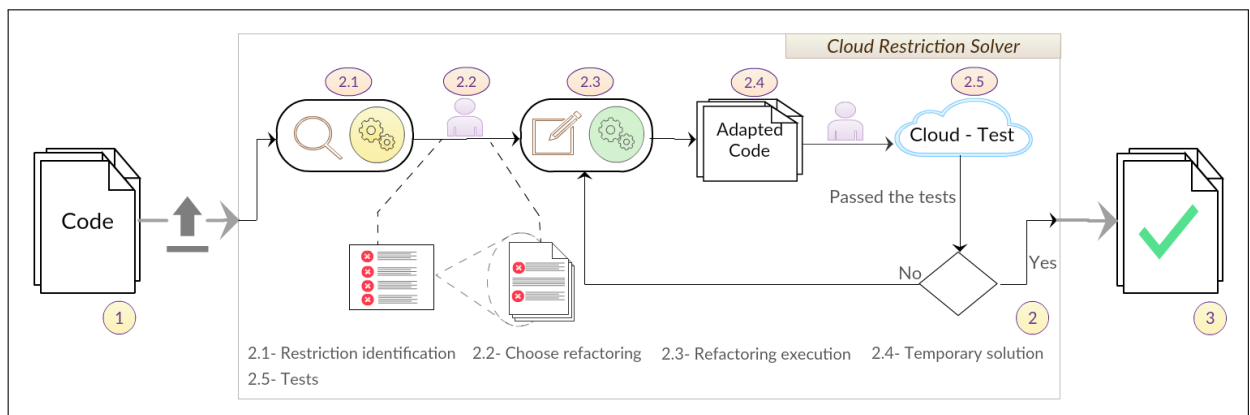
Figure 7 illustrates the proposed approach overview. It receives as input the source code of the legacy application that will be migrated (1), inspects the code to find potential violations of restrictions specific for the target PaaS provider, informs the found violations and its respective recommended refactorings to the user, who can apply the refactorings that modify the code with cloud-enabled similar services (2) and, finally, (3) outputs the refactored code that preserves the application behaviour but enables it to run on the chosen cloud PaaS platform.

The core of the approach (2) relies on two tool-supported phases: restriction identification and refactoring execution. The former (2.1) parses the inputted source code and displays a report containing the found violations, the files where the violations occur, and the possible refactorings to be applied. With this information, the user (the developer in charge of migrating the application) chooses the most appropriate refactoring (2.2) and executes it through the latter phase (2.3), which generates a temporary solution with the code adapted to use the cloud similar

services (2.4).

After migrating an application, it is necessary to perform unit or functional tests in the adapted system both to ensure that the code modifications have been successfully applied and to check whether the system behavior was actually preserved (2.5). Some PaaS providers offer, usually through its SDKs, local servers that simulate the cloud execution environment. If this technology is available, the test should start using the local server to simulate the application deployment in the cloud environment. Otherwise, it is necessary to execute the refactored application directly in the real cloud environment. If a refactoring fails and the modified application does not pass in the tests, it is necessary to go back to the refactoring engine and update the refactoring code (or create a new one). After that, the refactoring must be applied again to the source code until the application passes the migration tests.

Figure 7 – The proposed approach overview



Source: Elaborated by the author.

5.2 CRS FRAMEWORK

The proposed approach is implemented in terms of code through an open source framework that provides the implementation background needed to enable the user to specify the constrained functionality of the chosen PaaS. To this end, in this framework, there are two engines (identification and refactoring) that implement the two phases (2.1 and 2.3) of the CRS approach. The framework has been implemented in Java and is publicly available at BitBucket¹.

An aspect to be highlighted is that both engines transparently communicate via a JavaScript Object Notation (JSON) file, which is generated by the identification engine to tell its

¹ available at <<https://bitbucket.org/marcosborges1/crs>>

refactoring counterpart what code elements need to be modified.

In the following sections each engine is detailed, describing their core architecture and execution flows.

5.2.1 Identification Engine

Performing automatic identification of features that may violate restrictions in the target cloud is of the utmost importance since it provides developers agility by reducing the time spent in manually discovering those violations in the source code. In that sense, the best way to identify PaaS constraints in an application is to verify whether it uses restricted classes, which are classes that are not allowed to be executed in the chosen PaaS.

To perform the source code analysis, it has been created the *CRSAnalyzer* tool, which is based on *AutoRefactor*², a free and open source tool implemented in the Eclipse IDE to refactor Java code. Due to this, *CRSAnalyzer* is also available as a Eclipse plugin.

Although it is a refactoring engine, *AutoRefactor* has been chosen as the base for the identification engine since it uses the Abstract Syntax Tree (AST), which maps the application source code into a tree in which nodes represent code structures (ECLIPSE, 2006), and the Visitor³ design pattern (GAMMA, 1995) to query the tree nodes. Those elements (AST and the Visitor pattern) are very helpful in detecting the points of the code that violate the restrictions.

The identification mechanism proposed here relies on five types of detection that cover the behavioral possibilities of classes and interfaces: class instantiation, variable declaration, method declaration, class extension and interface implementation.

The *CRSAnalyzer* core architecture⁴ is depicted by Figure 8. Its main class is *CRSAnalyzer*, which scans the entire source code of the application looking for pieces of code that violate restrictions specific to the chosen PaaS. That class extends from *AbstractionAnalyzerRule* that, in turn, extends from *ASTVisitor* to visit the nodes, and implements the *AnalyzerRule* interface that defines methods that provide basic information, such as name (*getName()*) and description (*getDescription()*), used to process the tools as an Eclipse plugin.

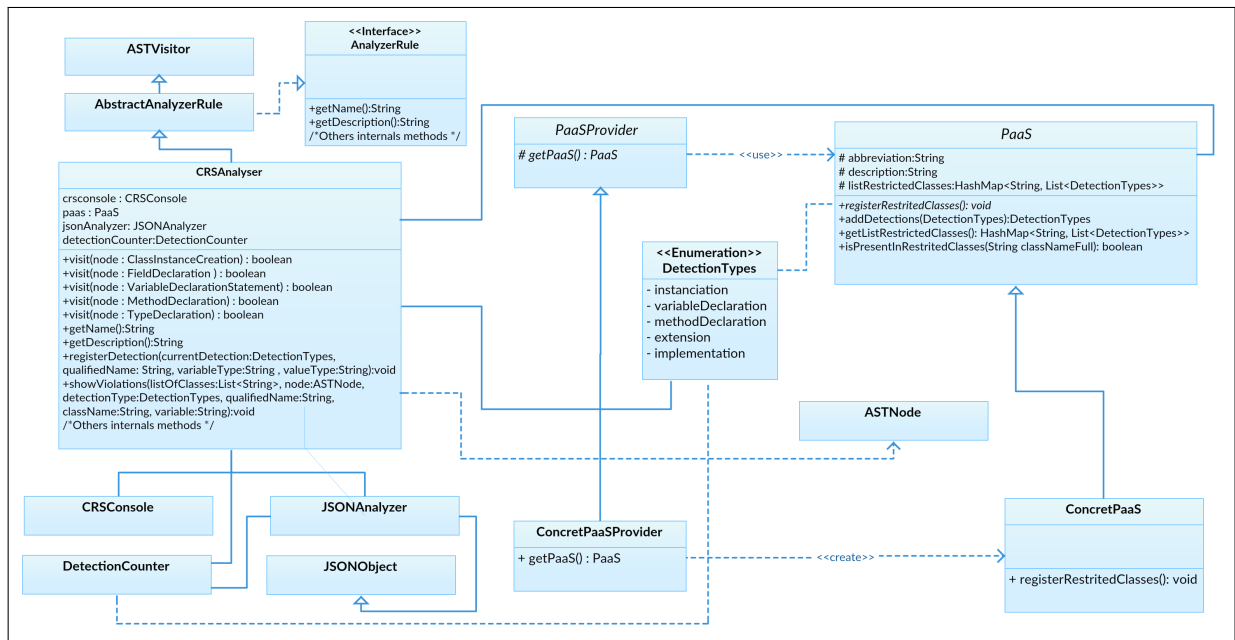
The five *visit* methods, overwritten by the *ASTVisitor*, receive as parameter the *ASTNode* type (an abstract class representing all types of node in an AST, such as classes,

² available at <<http://autorefactor.org>>

³ A pattern that allows the definition of a new operation over the object structure without modifying its class code

⁴ The diagram shows only the classes related to the identification process, but there are several others necessary to the communication with Eclipse that have been omitted for the sake of simplicity

Figure 8 – CRSanalyzer - Architecture.



Source: Elaborated by the author.

variables, methods, and conditional structure) of the desired information to visit specific nodes of an AST. These specific nodes are directly related to the five types of class detection and restricted interfaces (represented as the enumeration *DetectionType* in Figure 8) that should be analyzed in the application code. The relationship between the types of detection and the respective AST nodes can be seen in table 2.

Finally, the *registerDetection* method records all data (mostly detection types and restricted classes) in a JSON format, and the *showViolations* method shows the results of the violations as well as recommendation of the refactorings in the console for the developer.

Furthermore, the class *CRSanalyzer* is associated with four classes: (i) *CRSConsole*, (ii) *JSONAnalyzer*, (iii) *DetectionCounter*, (iv) *PaaS*. *CRSConsole* is responsible for displaying the results of the restriction detection in the Eclipse console, while *JSONAnalyzer* serves to register these results in a JSON format (used for intercommunication with the refactoring engine). Due to this, it extends the class that handles JSON files, the *JSONObject*⁵. *DetectionCounter*, based on the Singleton pattern (GAMMA, 1995), is used to count the occurrences of the restricted classes according to the the five types of detection previously mentioned and registers it in a JSON format using the *JSONAnalyzer* class.

The *PaaS* class represents the cloud platform for which the application will be

⁵ available at <<https://mvnrepository.com/artifact/com.googlecode.json-simple/json-simple/1.1.1>>

Table 2 – Correspondence between detection type and ASTNode type.

Detection Type	ASTNode type	Description
instantiation	ClassInstanceCreation	Nodes associated to the class instantiation
variableDeclaration	FieldDeclaration VariableDeclarationStatement	Nodes associated to the class attributes Nodes associated to the variables within methods
methodDeclaration	MethodDeclaration	Nodes associated to the method declarations
extension	TypeDeclaration	Nodes associated to the class extension
implementation	TypeDeclaration	Nodes associated to the interface implementation

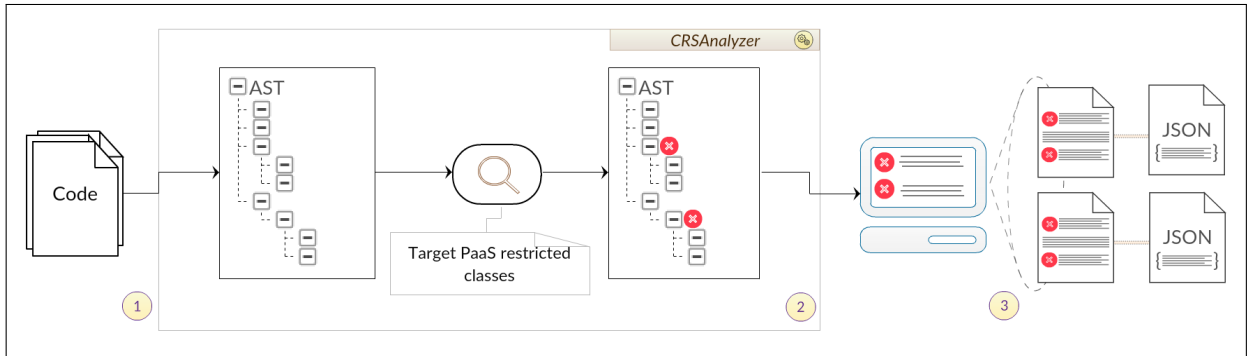
migrated and, therefore, contains the restrictions that will be analyzed by *CRSAnalyzer*. It is an abstract class that has 3 attributes and 4 methods. The attributes related to PaaS are: *abbreviation*, *description*, and *listRestrictedClass*, a hash map that associates the restricted classes to each type of possible detection of them in the code. The methods basically reference or cooperate with the *map* attribute. These methods are: *addDetectionsType()* (which adds detection types); *getListRestrictedClasses()* (which returns the *map*), *isPresentInRestritedClass()* (which checks if a restricted class is present on the *map*), and finally the *registerRestritedClasses()* abstract method that must be implemented by a *ConcretePaaS* class with the chosen PaaS restricted classes along with the types of detection for each of them. Furthermore, *ConcretePaaS* object is created in the *ConcretePaaSProvider* class, which extends from an abstract class *PaaSProvider* that represents the company that offers the selected PaaS. Note that those four classes implement the Factory Method pattern (GAMMA, 1995).

The internal execution flow of the *CRSAnalyzer* is depicted by Figure 9. From the original application code (1), a parser is carried out to generate the code Abstract Syntax Tree (AST), whose nodes (instantiated classes, variables and methods declarations, inherited classes and classes that implement interfaces) are analyzed according to the type of detection specified in each restricted class using the behavioral design pattern Visitor (2). If violations are found, the errors are reported in the IDE console area (3), which precisely indicates the lines of code in the application files that need to be refactored and recommends the appropriate refactoring. Finally, the tool generates a JSON file containing the information needed for the refactoring engine.

5.2.2 Refactoring Engine

Different from the traditional refactorings (FOWLER; BECK, 1999), which are defined to perform an action over a class element (e.g., rename class, remove attribute, extract method, move method), the primitive refactorings proposed in this work are meant to replace specific pieces of code that use restricted classes (identified by the identification engine) by

Figure 9 – CRSAnalyzer - Execution flow



Source: Elaborated by the author.

equivalent services provided by the PaaS, thus enabling the application to run in the cloud preserving the same behaviour.

To obtain these refactorings two implementation strategies have been considered. The first one consisted of modifying the application source code by removing the problematic pieces and introducing directly the equivalent cloud code. Although efficient, this solution had three main side effects: firstly, it turned the code highly coupled to the PaaS API, which may cause compatibility problems (or even breaking the application) in case the API evolves and some methods suffer changes; secondly, if the previous fact happens and the evolved class is spread out in many application files, it could be very expensive and time-consuming to update all affected files; finally, the code became more difficult to understand and, consequently, to maintain. Therefore, that strategy was discarded.

The chosen strategy consisted of introducing a communication class between the restricted class and the class that implements the similar service in the cloud. In fact, this solution implements the Adapter design pattern (GAMMA, 1995), so the communication class has the same interface (methods signatures) of the restricted class, but it reimplements them using the correct cloud services. This way, the refactoring replaces the pieces of code where the restricted class is used according to the five detection types previously described by the correspondent communication class. This is a simpler and more maintainable solution that should ease the implementation of new refactoring in the CRS approach.

To perform the modifications in the code, it has been implemented the *CRSRefactor* tool, which is also based on AutoRefactor and uses its main concepts: AST and the Visitor design pattern. The CRSRefactor core architecture can be seen in Figure 10. In this figure, each refactor class (*CRS4<Cloud><RestrictedClass>Refactoring*), which is specific to a restricted

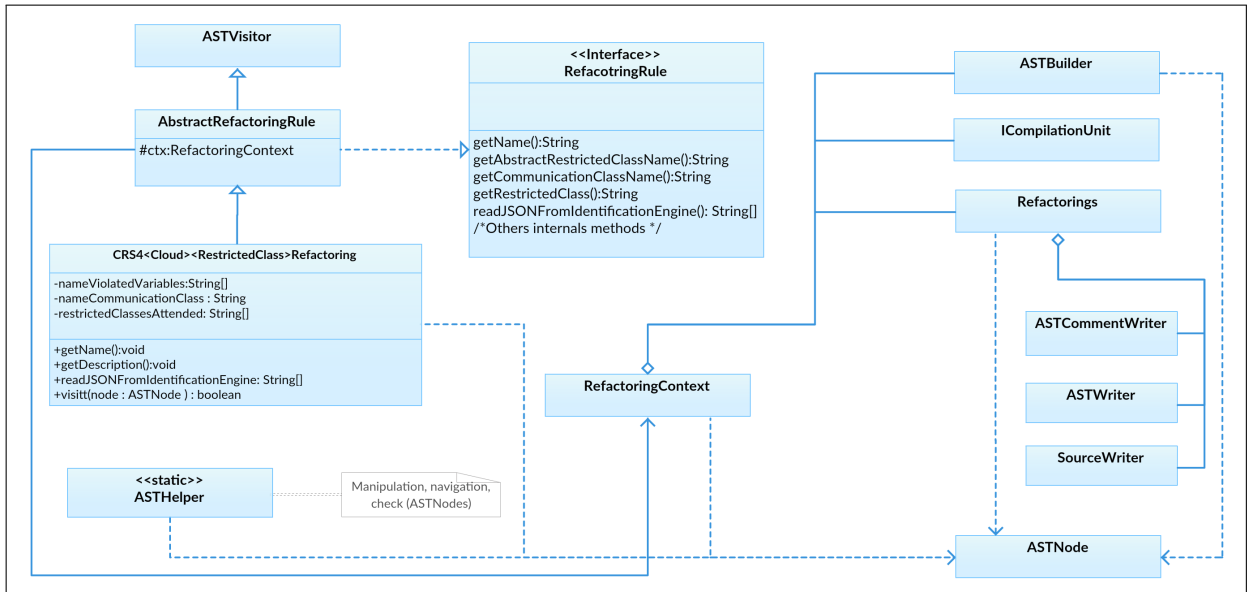
class in the cloud, must extend from *AbstractionRefactoringRule* (an abstract class that defines some internal settings for the refactoring processing). This last class extends from *ASTVisitor* and implements *RefactoringRule*, which is an interface that defines intrinsic methods for refactorings. Among these methods, can be highlighted: *getName()*, which indicates the name of the refactoring; *getAbstractRestrictedClassName()*, which indicates the restricted abstract class; *getCommunicationClassName()*, which references the communication class; *getRestrictedClass()*, which points to the restricted class of that refactoring; and *readJSONFromIdentificationEngine()*, which makes the refactoring engine reading the JSON file outputted by the CRSAnalyzer engine.

The code transformations occur when the specific nodes of the code AST are visited through the *visit* methods of the *ASTNode* class (some of them have been shown in table 2). For this, the *RefactoringContext* class can be used, since it aggregates: (i) specific classes for node creation in the AST (*ASTBuilder*); (ii) classes for updating, replacing and removing nodes in the AST (*Refactorings*), which accomplish this mainly through the *ASTRewrite* class (directly in the AST), *ASTCommentRewriter* (in the AST comments) and *SourceWriter* (in the source code itself); (iii) *ICompilationUnit*, a java compilation unit to which the previous items (i and ii) can be applied. The imports of the communication class are automatically called in this unit. Finally, to aid the code transformation, the static class *ASTHelper* can also be used, since it manipulates, navigates, and checks nodes in the AST. All those nodes created and updated in AST are based on the communication classes.

The communication classes are responsible for both implementing cloud services and adapting the refactorings in order to cover the possibilities (parameters and returns) of the existing data types and/or objects of the restricted class. To do that, the best found solution was to make them be constituted as *Maven* dependencies.

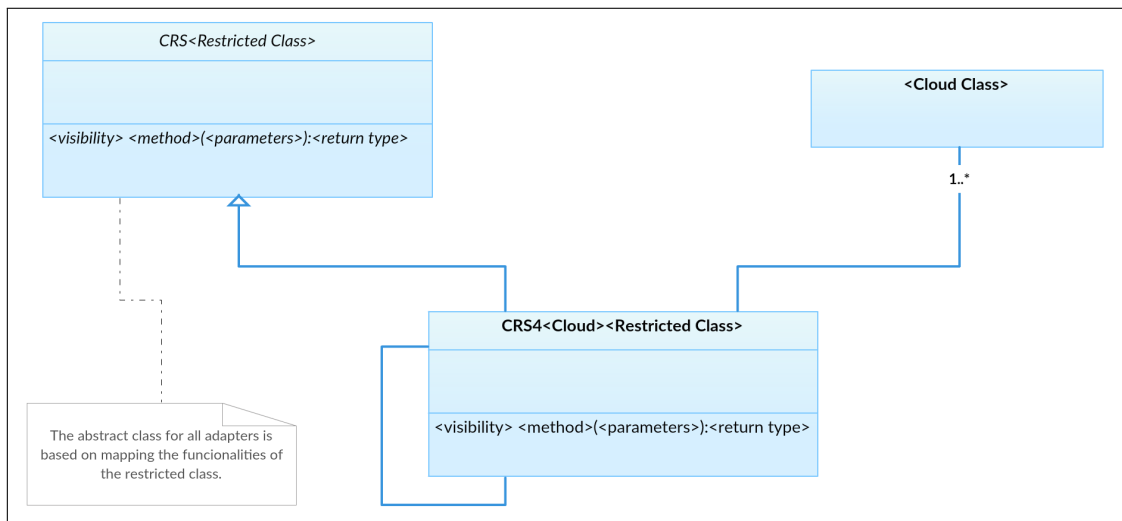
The architecture of the communication class, based on the Adapter pattern, can be seen in Figure 11, in which the communication class (*CRS4<Cloud><RestrictedClass>*), which is specific to a target cloud and a restricted class, must extend an abstract target class (*CRS<RestrictedClass>*) that contains the signatures of the methods of a restricted class. The communication class must rewrite those methods by adapting them based on classes implementing equivalent services in the target cloud (*<CloudClass>*). It is possible that a certain communication class is associated to another one mainly due to dependencies between restricted classes. It is important to note that, since the restricted abstract class is defined, it will serve as the basis for all adaptations from other clouds.

Figure 10 – CRSRefactor - Architecture



Source: Elaborated by the author.

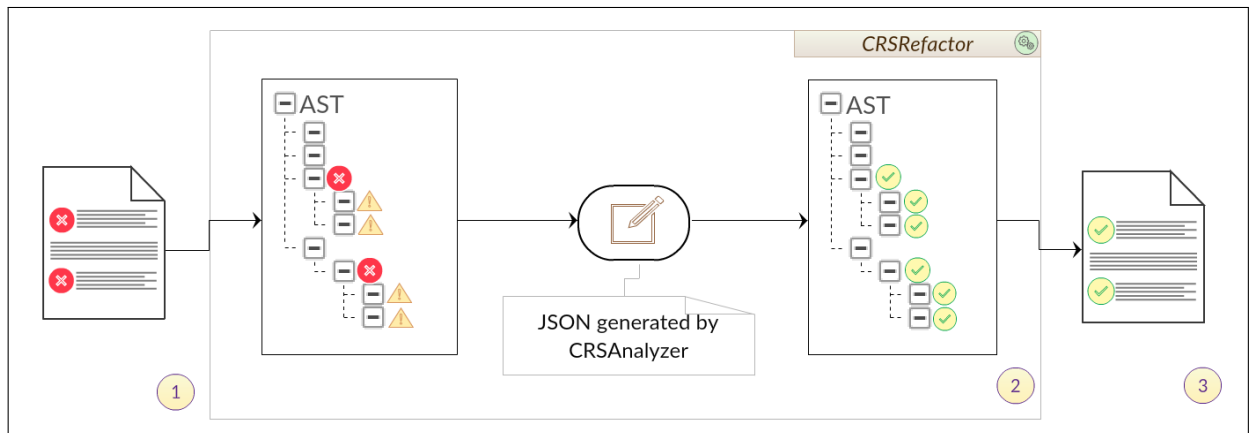
Figure 11 – Communication Class - Architecture



Source: Elaborated by the author.

The internal behavior of the refactoring engine can be seen in Figure 12. When a given refactoring is applied to a violated code (1), the engine firstly converts the code into its AST representation and then reads the output of the identification engine (data in JSON format) to locate the nodes and sub-nodes (objects, methods, and signatures of the class) that need to be modified. Each refactoring then updates the AST nodes by replacing them by nodes that represent similar services in the respective communication class (2). Finally, the AST is converted back into a text format, outputting the modified code that will be tested in the cloud PaaS.

Figure 12 – CRSRefactor - Execution flow



Source: Elaborated by the author.

5.3 CRS INSTANTIATION

If a developer is interested in instantiating the CRS approach to a specific PaaS, he/she has to perform the following steps:

- *Choose a target PaaS environment:* the first step consists of selecting the cloud provider and its PaaS environment for which the applications will be migrated.
- *Identify the PaaS restrictions and its respective restricted classes:* secondly, the developer needs to find the restrictions inherent to the chosen PaaS (maybe reading its documentation). After, it is necessary to identify the classes that violate each cloud restriction in the programming language of the applications to be migrated.
- *Specify a refactoring for each restricted class:* the developer has to map the methods of the restricted classes that need to be adapted to the cloud, create a refactoring class and give a refactoring name for each restricted class. In addition, he/she creates a communication class and an abstract target class for each refactoring.
- *Extend the CRSAnalyzer:* here the developer has to implement the restriction detection by extending the CRSAnalyzer classes, as described in Section 4.2.1.
- *Extend the CRSRefactor:* the developer has to extend the CRSRefactor classes to implement the specified refactorings and its communication classes and abstract target classes, as described in Section 4.2.2.

Finally, the developer has to compile the project to generate the new instance of the CRS as an Eclipse plugin. It is recommended to give to the new tool the name CRS4<PaaS>, indicating that it is an instance of the CRS approach for the specific PaaS.

5.4 CHAPTER SUMMARY

This chapter presented the approach of this work. It was demonstrated its general vision and its fundamental phases. In addition, was presented the CRS framework that implements the approach as well as the technical aspects of its identification and refactoring engines. Finally, a general process of instantiation of the CRS framework for PaaS clouds was highlighted.

In the next chapter it will be shown an instantiation of the CRS approach to the cloud Google App Engine, CRS4GAE.

6 CRS4GAE

This chapter describes the Cloud Restriction Solver for Google App Engine (CRS4GAE), an instantiation of the CRS approach designed to help the migration of applications to the Google's PaaS environment. Section 6.1 presents an overview about the Google App Engine, highlighting the restrictions imposed by the PaaS provider, while section 6.2 explains how CRS4GAE works, giving some details of the implementation of the engines.

6.1 GOOGLE APP ENGINE

Google App Engine (GAE) is the Google's cloud PaaS used to develop, in general, web and mobile applications. It was designed to deal with high request traffic rates, serving several users through the automatic scalability feature. It was selected as the target cloud environment of this work since it is a robust platform, has a great adoption in the market, and is commonly used by other work available in the literature (VU; ASAL, 2012; MAENHAUT et al., 2016; FREY; HASSELBRING, 2011; PRABHAKARAN, 2014). The platform, which is part of the *Google Cloud Platform*¹, provides development and execution environments (APIs and SDKs) to Java, Python, Go and PHP applications (SANDERSON, 2015). This work focus on Java web applications.

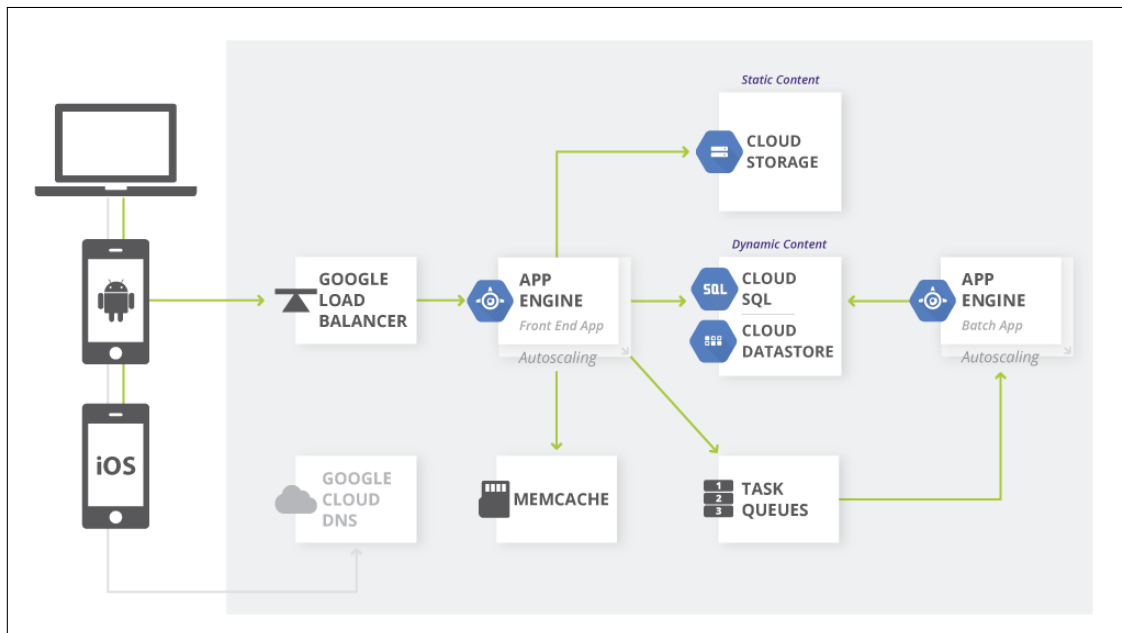
The simplified GAE architecture can be seen in Figure 13. Among the main services and structures available are: *Google Load Balancer*, which manages the load balancing of the applications; *Front End App*, responsible for redirecting requests for appropriate services; *Memcache*, which is the cache memory shared between instances of the GAE, generating high speed in the availability of the information on the server; and *Task Queues*, mechanism that provides redirection of long tasks to back-end servers, making front-end servers free for new user requests. In addition, GAE also has static and dynamic storage solutions. The former provides the file storage service called *Cloud Storage*, whereas the latter provides relational database services such as *Cloud SQL*, and non-relational NoSQL such as *Cloud Datastore* (GOOGLE, 2016).

The model for deploying a java application on GAE consists of 3 steps: (i) creation of the project in the Google Cloud Console²; (ii) testing of the application on the local server; (iii) deployment of application on GAE. The first step is crucial because it generates the project

¹ available at <<https://cloud.google.com/>>

² available at <<https://console.cloud.google.com/>>

Figure 13 – Google App Engine platform architecture



Source: (GOOGLE, 2016)

ID (unique) that serves to identify the application in the infrastructure of the GAE. The second is based on a web development console that inspects and simulates the application as if it were in the cloud, however on the local machine. Finally, the deployment step is to upload and test the application to the GAE cloud. This migrated application can be accessed as follows: <project-id>.appspot.com.

6.1.1 Restrictions

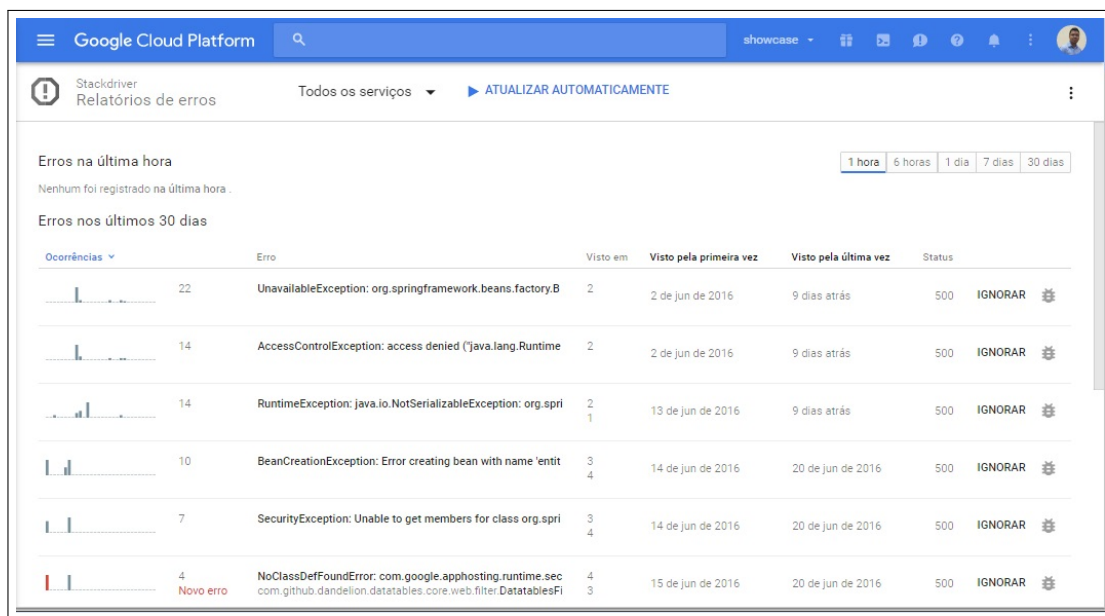
The Java applications deployed in the GAE are executed in an environment called *sandbox* (SANDERSON, 2015), which manages all interactions between the application and the cloud infrastructure. To ensure the performance and scalability, that environment imposes some restrictions to the application, such as:

- The application cannot create or modify files directly in the file system and is limited to read its own resource files.
- It is not allowed to an application to generate additional processes. The threads are not forbidden, but only work with the following condition: secondary threads are only executed until the end of the main thread, after that they are terminated.
- The application cannot see or access other applications or processes that are being executed in the same server.

- The requisitions to the application must respond in a maximum of 60 seconds in order to avoid overloading the web server.

GAE defines as a *whitelist* a set of classes that can be safely executed in its cloud environment. Analogously, in this work, classes that lie outside this *whitelist*, or that are in it but contain methods that cannot be executed in the cloud, are considered as restricted classes. Violations of such constraints usually occur at runtime through exceptions and are shown primarily in the local GAE (console on the local machine) and in the cloud GAE (logs and error reports from *Google StackDriver*³ seen in Figure 14). The error report in that figure shows the number of occurrences of each error, the exceptions triggered by it, the period in which they occurred, and the status of the server response. *Google StackDriver* is of paramount importance as it ensures that errors (exceptions posted) are displayed, since local GAE sometimes does not show them.

Figure 14 – StackDriver error report



Source: Elaborated by the author.

6.2 INSTANTIATION OF CRS FOR GAE

The instantiation of the CRS4GAE follows the steps presented in Section 4.3. The chosen PaaS is the GAE and, among its restrictions, has been selected two of them to deal with in this work: writing files and running threads. As the interest of this work is on migrating Java

³ Integrated solution for monitoring, logs and diagnostics of the *Cloud Platform*.

Table 3 – Restricted classes and its respective refactorings

Restriction	Class/package	Refactoring/(Class Refactoring)	Communication Class	Abstract target class
Writing Files	FileOutputStream/java.io	Adapt FileOutputStream to GAE (CRS4GAFileOutputStreamRefactoring)	CRS4GAFileOutputStream	CRSFileOutputStream
	FileWriter/java.io	Adapt FileWriter to GAE (CRS4GAFileWriterRefactoring)	CRS4GAFileWriter	CRSFileWriter
	File/java.io	Adapt File to GAE (CRS4GAFileRefactoring)	CRS4GAFile	CRSFile
Threads executions	Thread/java.lang	Adapt Thread to GAE (CRS4GAThreadRefactoring)	CRS4GAThread	CRSThread

applications, the restrictions affect the classes *java.io.File*, *java.io.FileWriter*, *java.io.FileOutputStream*, and *java.lang.Thread*.

Then, for each restricted class, has been defined a specific refactoring whose name follows the template “*Adapt <RestrictedClass> to GAE*” and its respective class was named as *CRS4GAE<RestrictedClass>Refactoring*. For instance, for the restricted class *File*, the refactoring created was named *Adapt File to GAE* and its class name was *CRS4GAFileRefactoring*.

Each refactoring has a communication class and an abstract target class that contains the method signatures of the original restricted class. For instance, for the restricted class *File*, it was created the classes *CRS4GAFile* and *CRSFile* as the communication and the abstract target classes, respectively. This entire map is presented in Table 3.

The steps regarding extending the tool will be better described in the next sections. The final step generated the CRS4GAE tool, which is available at Bitbucket⁴.

6.2.1 Identification Engine

As seen in section 5.2.1, the CRS identification engine can be implemented for several clouds and the main aspect to be considered by it is a concrete class of the *PaaS* (abstract class) type. Considering GAE this cloud, it is necessary to create a class that represents it (listing 2, line 2), putting its necessary information, such as abbreviation and description (lines 3 to 7). Additionally, in the implementation of the method *registerRestrictedClasses()*, the identified restricted classes that come from table 3 are registered, along with the types of detection in the applications for each one (lines 8 to 16).

Since the *GAE* class is defined, it must be called by its *PaaS* provider, in this case Google. So, it has been created the *GooglePaaSProvider* class that extends from *PaaSProvider* (listing 3, line 2) and implements the mandatory method *getPaaS()*, returning the respective *PaaS* of the company, in this case a *GAE* object (lines 3 to 5).

⁴ available at <<https://bitbucket.org/marcosborges1/crs4gae>>

After that, the chosen PaaS can be defined in the class *CRSAnalyzer* (listing 4 line 5), such that *CRSAnalyzer* checks its restricted classes (specified in the *registerRestrictedClasses()* method) and performs the analysis in the application source code looking for each type of restriction violation detection, generating the console report and the JSON files of the violated restrictions.

Source Code 2 – Code excerpt - GAE

```

1 //Omitted details
2 public class GAE extends PaaS {
3     public GAE() {
4         this.abbreviation = "GAE";
5         this.description = "Google App Engine";
6         listRestrictedClasses = new HashMap<String, List<DetectionTypes>>();
7     }
8     public void registerRestrictedClasses() {
9         listRestrictedClasses.put("java.io.FileWriter", addDetections(DetectionTypes.
10             variableDeclaration,
11                 DetectionTypes.methodDeclaration, DetectionTypes.extension,
12                 DetectionTypes.instantiation));
13         listRestrictedClasses.put("java.io.FileOutputStream", addDetections(
14             DetectionTypes.variableDeclaration,
15                 DetectionTypes.methodDeclaration, DetectionTypes.extension,
16                 DetectionTypes.instantiation));
17         listRestrictedClasses.put("java.io.File", addDetections(DetectionTypes.
18             variableDeclaration,
19                 DetectionTypes.methodDeclaration, DetectionTypes.extension,
20                 DetectionTypes.instantiation));
21         listRestrictedClasses.put("java.lang.Thread", addDetections(DetectionTypes.
22             instantiation));
23     }
24 }

```

Source Code 3 – Code excerpt - GooglePaaSProvider

```

1 //Omitted details
2 public class GooglePaaSProvider extends PaaSProvider {
3     public PaaS getPaaS() {
4         return new GAE();
5     }
6 }

```

Source Code 4 – Code excerpt - CRSAnalyzer

```

1 //Omitted details
2 public CRSAnalyzer() {
3     //Omitted details
4     PaaSProvider provider = new GooglePaaSProvider();
5     this.setPaaS(provider.getPaaS());
6     this.getPaaS().registerRestrictedClasses();
7     //Omitted details
8 }

```

6.2.2 Refactoring Engine

As seen in section 5.2.2, refactorings are rules that aim to adapt restricted class features to communication classes that implement services in the cloud. These refactorings are implemented through the *CRSRefactor* class and the communication classes are made available as Maven dependencies to ease the code modification during refactoring process. For the Google App Engine, the four refactorings and their respective created communication classes and abstract target classes can be seen in table 3.

To demonstrate how to implement a refactoring rule in the *CRSRefactor* tool, the *Adapt File to GAE* refactoring has been chosen. According to the class diagram of the *CRSRefactor*, shown in Figure 10, it has been created the class *CRS4GAEFileRefactoring* that extends from *AbstractRefactoringRule* (Listing 5, line 2) and implements the mandatory methods from the interface *RefactoringRule*, which contains the information necessary for the refactoring, such as its name, restricted abstract class, communication class, and original class that can not be performed completely in the cloud (lines 4 to 19). Finally, from lines 21 to 39, all AST nodes related to class instantiation are visited, and if that node belongs to a restricted class type (in this case, *java.io.File*), it is replaced (class *Refactorings*) by a newly created node (class *ASTBuilder*) of the respective communication class type (in this case, *CRS4GAEFile*).

Source Code 5 – Code excerpt - CRS4GAEFileRefactoring

```

1 //Omitted details
2 public class CRS4GAEFileRefactoring extends AbstractRefactoringRule {
3     //Omitted details
4     @Override
5     public String getName() {
6         return "Adapt File to GAE";
7     }
8     @Override
9     public String getAbstractRestrictedClassName() {

```

```

10     return "CRSFile";
11 }
12 @Override
13 public String getCommunicationClassName() {
14     return "CRS4GAEFile";
15 }
16 @Override
17 public String getRestrictedClass() {
18     return "java.io.File";
19 }
20
21 @Override
22 public boolean visit(ClassInstanceCreation node) {
23
24     final ITypeBinding typeBinding = node.getType().resolveBinding();
25     final List<Expression> arguments = ASTHelper.arguments(node);
26
27     // Tree-nodes creators (ASTBuilder) and refactorings (Refactorings)
28     ASTBuilder builder = this.ctx.getASTBuilder();
29     Refactorings refactorings = this.ctx.getRefactorings();
30
31     if (typeBinding != null && arguments.size() == 1) {
32         // Full class name, equivalent to import
33         final String fullNodeClassName = typeBinding.getQualifiedName();
34         if (this.getRestrictedClass().equals(fullNodeClassName)) {
35             // Replaces the old node with the new one from ASTBuilder
36             refactorings.replace(node,
37                 builder.createNewClass(this.getCommunicationClassName(), builder.
38                     copy(arguments.get(0))));
39         }
40         // Return Tree
41         return VISIT_SUBTREE;
42     }
43     //Omitted details
44 }

```

The *CRS4GAEFile* is the communication class of the refactoring class *CRS4GAEFileRefactoring*. Thus, based on the architecture of the communication class (Figure 11), it must extend a restricted abstract class *CRSFile* (which contains the restricted methods of the *java.io.File* class, shown in Listing 6), and then implement the methods of that class, adapting them based on equivalent services provided by the Google App Engine.

In Listing 7, the *CRS4GAEFile* communication class is created based on the restricted abstract class *CRSFile* (line 1), along with: (i) the associations of GAE classes (lines 4 and 5);

(ii) the possible constructors (lines 6 to 21); (iii) the implementation of the abstract methods, such as the *exists()* method (line 23 to 32).

Finally, the Maven dependency containing the GAE communication classes, based on table 3, is available for download⁵.

Source Code 6 – Code excerpt - CRSFile

```

1 public abstract class CRSFile {
2     //Omitted details
3     public abstract String getPath();
4     public abstract boolean exists();
5     public abstract String getAbsolutePath();
6     public abstract boolean mkdirs();
7     public abstract CRSFile[] listFiles();
8     public abstract boolean isDirectory();
9     public abstract String getName();
10    //Omitted details
11 }

```

Source Code 7 – Code excerpt - CRS4GAEFile

```

1 public class CRS4GAEFile extends CRSFile {
2
3     private GcsFilename gcsFileName;
4     private final static GcsService gcsService = GcsServiceFactory.createGcsService(
5         RetryParams.getDefaultInstance());
6
7     public CRS4GAEFile(String pathName) {
8         this.setGcsFileName(new GcsFilename(bucketName,
9             CRSUtils.isDirectory(pathName) ? CRSUtils.asDirectoryGAE(pathName) :
10                pathName));
11    }
12
13    public CRS4GAEFile(String parent, String child) {
14        this(CRSUtils.asDirectoryGAE(parent) + child);
15    }
16
17    public CRS4GAEFile(File parent, String child) {
18        this(CRSUtils.asDirectoryGAE(parent.getPath()) + child);
19    }
20
21    public CRS4GAEFile(CRS4GAEFile fileCloud, String name) {
22        this(fileCloud.getPath() + name);
23    }
24 }

```

⁵ available at <<https://mvnrepository.com/artifact/com.github.crs-tool/crs4gae/0.0.2>>

```
22
23     @Override
24     public boolean exists() {
25         boolean exists = false;
26         try {
27             exists = (getGcsService().getMetadata(this.getGcsFileName()) != null) ? true
28                 : false;
29         } catch (IOException e) {
30             e.printStackTrace();
31         }
32         return exists;
33     }
34 }
```

6.3 CHAPTER SUMMARY

In this chapter, the instantiation of the CRS approach to the cloud GAE was presented. Firstly, the GAE's general information and its main restrictions were highlighted. Subsequently, the two restrictions adopted in this work, written on the filesystem and threads, along with the implementation details of the CRS engines that handle them (identification and refactoring), were described.

In the next chapter a usage example of the tool CRS4GAE in three applications java is shown.

7 USAGE EXAMPLE

This chapter describes how three java web applications have been migrated to the GAE using the CRS4GAE tool. They have been selected since they contain features that violate some of the GAE constraints: writing files and threads.

The first two applications, *JAX-RS File Upload* and *Servlet with Thread*, are simple and have been used to show how the refactorings have been applied and modified the code. The last one, *Pebble*, is more complex and has been used to present quantitative results about the identification of restrictions and the impact of the refactorings in the code.

7.1 JAX-RS FILE UPLOAD

JAX-RS File Upload¹ is a JEE example application that illustrates the use of the Java API for RESTful Web Services (JAX-RS). The API is used to make file uploads to a specific directory from the local disk using the Jersey Multipart component.

The code excerpt presenting the file writing feature of that application can be seen in Listing 8. In that code, the *FileOutputStream* class is instantiated on lines 44 and 49, fact that violates the restriction of writing files in GAE. To find this code violation, the user can run the CRSAnalyzer tool by right-clicking on the project directory in Eclipse and choosing the option “CRSAnalyser” (illustrated by Figure 15), which performs the static analysis in the application’s project directory. As a result, the tool presents in the Eclipse console a report with the four violations found, as shown by Figure 16.

The report provides the following information for each violation: type of detection, detailed description of the reason for detection, file and line where the violation was found, and the recommended refactoring available at the CRSRefactor tool. For instance, the first violation in the report informs that a restricted class instantiation was detected, details that the problem was that the class *FileOutputStream* cannot be instantiated, points that the problem occurred in line 44 of the *UploadFileService.java*, and recommends the user to use the refactoring *Adapt FileOutputStream to GAE*.

With those information, the user can apply a refactoring by accessing the correct file in the project directory, right-clicking on it, choosing the option “CRSRefactor”, and after selecting “Choose refactorings”. A new screen is displayed containing the available refactorings

¹ available at <<http://www.mk Yong.com/wp-content/uploads/2011/07/JAX-RS-FileUpload-Jersey-Example.zip>>

from where the user can select the recommended refactoring by the CRSanalyzer tool. In the used example, he/she has selected the refactoring *Adapt FileOutputStream to GAE*. This process is depicted by Figure 17.

Finally, after all refactorings have been applied, the generated modified code is outputted and can be seen in Listing 9. Note that the refactoring basically used a helper class to provide the byte array (line 48) and changed the restricted classes by its respective communication classes (line 49). Now, the methods *write()*, *flush()*, and *close()* (lines 50, 51, and 52, respectively) execute the redefined services in the GAE.

The code in Listing 9 has been successfully tested both in the local environment and in the cloud. The application correctly uploaded files to the GAE administrator. The feature has been verified based on the navigation of web pages using the Selenium² tool and the PageObject³ pattern.

The code (refactored and tested) and the link for the JAX-RS File Upload application are available at, respectively, Bitbucket⁴ and in the GAE⁵.

Source Code 8 – Excerpt from the original code - JAX-RS

```

42 //Omitted details
43 try {
44     OutputStream out = new FileOutputStream(new File(uploadedFileLocation));
45     int read = 0;
46     byte[] bytes = new byte[1024];
47     out = new FileOutputStream(new File(uploadedFileLocation));
48     while ((read = uploadedInputStream.read(bytes)) != -1) {
49         out.write(bytes, 0, read);
50     }
51     out.flush();
52     out.close();
53 }
54 //Omitted details

```

Source Code 9 – Refactored code excerpt - JAX-RS

```

46 //Omitted details
47 try {
48     byte[] bytes = CRSUtils.convertFileToByteArray(uploadedInputStream);

```

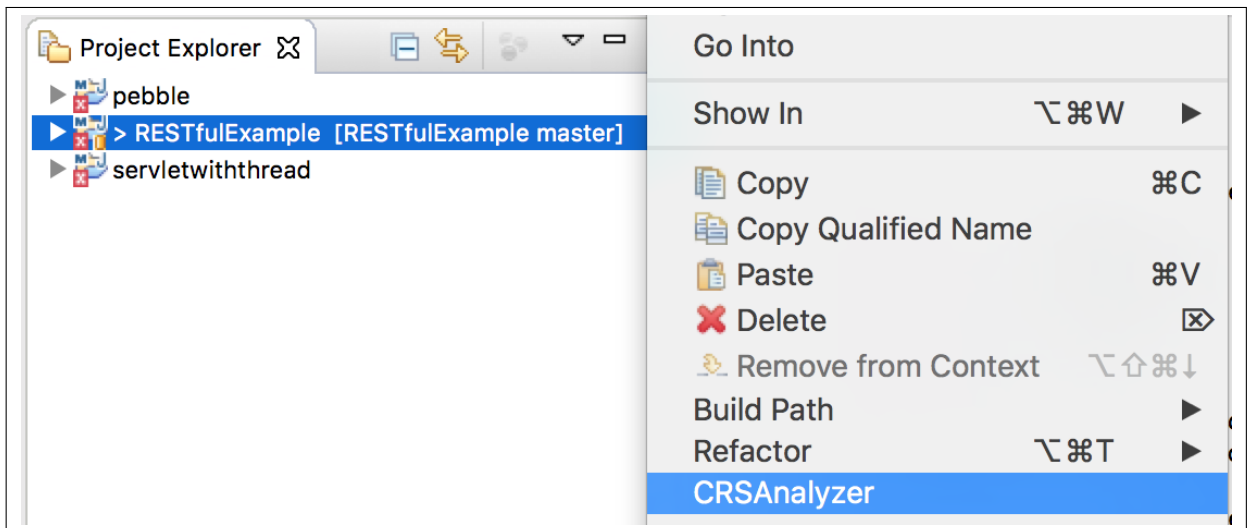
² available at <<http://www.seleniumhq.org>>

³ available at <<https://martinfowler.com/bliki/PageObject.html>>

⁴ available at <<https://bitbucket.org/marcosborges1/restfulexample>>

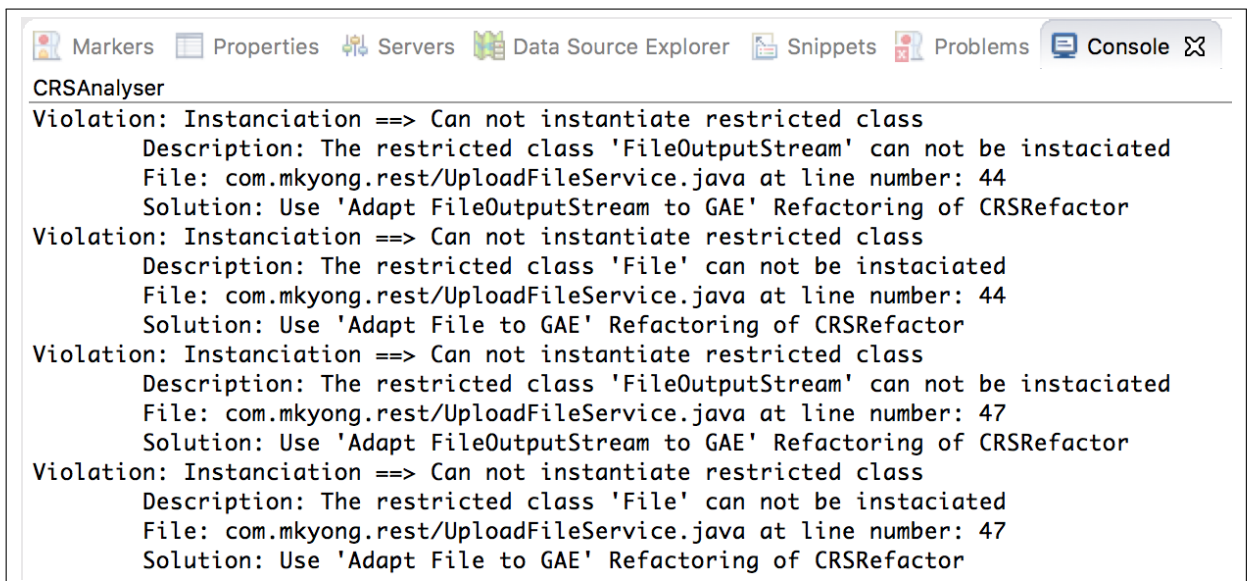
⁵ available at <<http://upteste-1247.appspot.com>>

Figure 15 – Using CRSAnalyzer in the application project.



Source: Elaborated by author.

Figure 16 – Report of violation of restricted classes of JAX-RS.



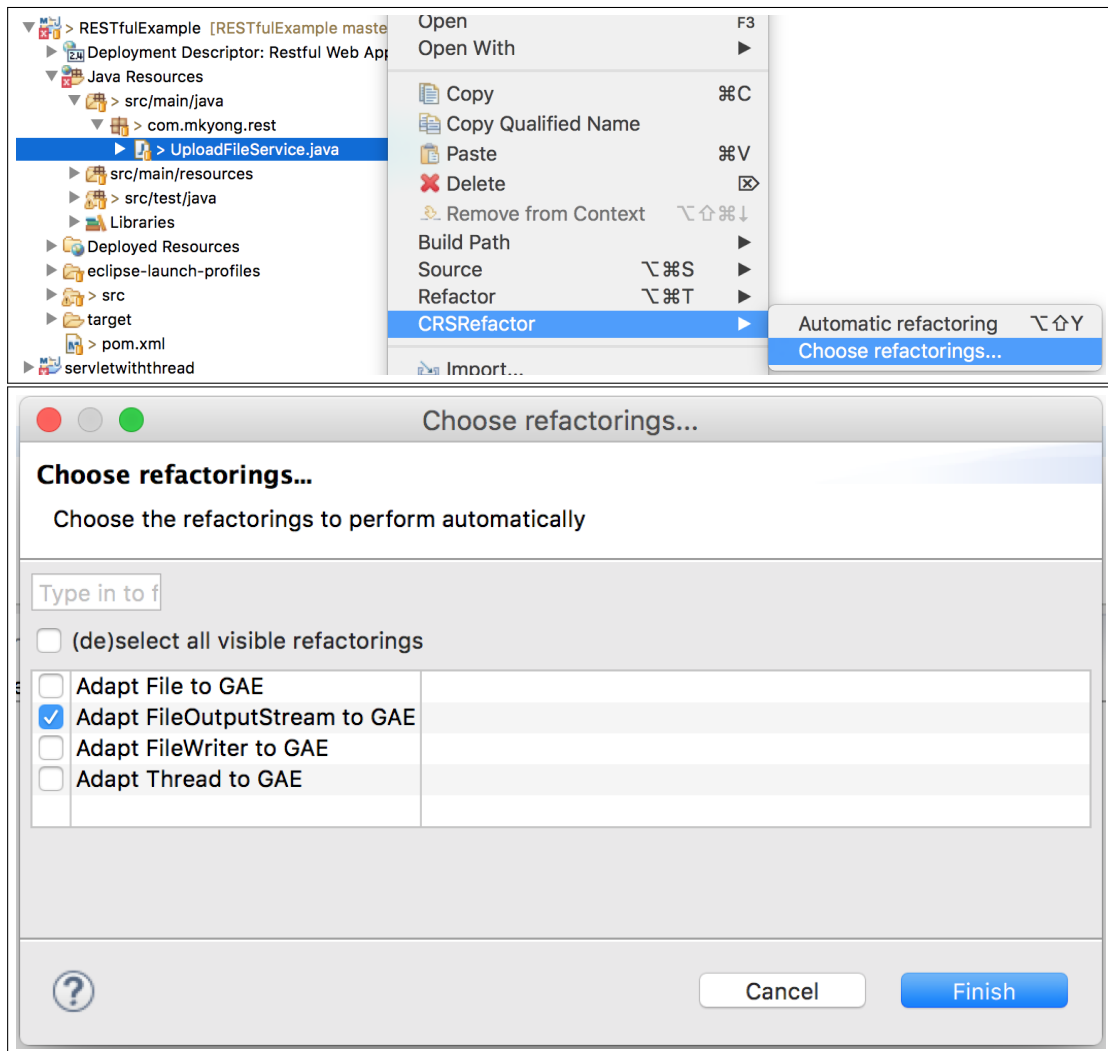
Source: Elaborated by author.

```

49     OutputStream out = new CRS4GAEFileOutputStream(new CRS4GAEFile(uploadedFileLocation))
        ;
50     out.write(bytes);
51     out.flush();
52     out.close();
53 }
54 //Omitted details

```

Figure 17 – Choose CRSRefactor refactorings.



Source: Elaborated by author.

7.2 SERVLET WITH THREAD

Servlet With Thread⁶ is an example application that basically works with threads in an servlet to enable the asynchronous communication in the web page.

The code excerpt in which threads are used in the application can be seen in Listing 10. In this code, the *Thread* class is instantiated on line 11, fact that violates the thread execution restriction in the GAE. Likewise the previous application, the CRSAnalyzer tool was used to find the code violations following the same steps. Figure 18 shows the violation report generated for this application that recommends the user to apply the *Adapt Thread to GAE* refactoring. By doing this using the CRSRefactor tool, it was obtained the refactored code shown in Listing 11. Given the simplicity of the original code (Listing 10), the modified code is very similar to the

⁶ available at <http://www.java2s.com/Tutorial/Java/0400__Servlet/ServletWithThread.htm>

original one, but with the proviso that it uses the communication class *CRS4GAETHread*.

The migrated application has also been successfully tested both in the local environment and in the cloud using Selenium and the PageObject pattern. The verification consisted of checking that running threads did not finish before the GAE main thread.

The code and the link for the migrated Servlet with Thread application are available at, respectively, Bitbucket⁷ and the GAE⁸.

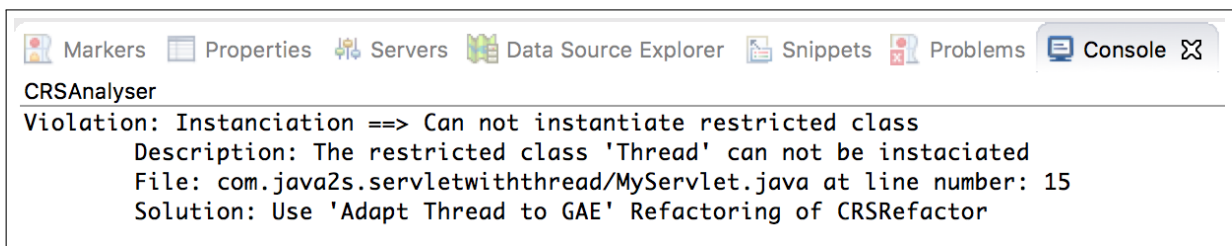
Source Code 10 – Excerpt from original code - Java with Servlet

```

11 //Omitted details
12 public void init() throws ServletException {
13     searcher = new Thread(this);
14     searcher.setPriority(Thread.MIN_PRIORITY);
15     searcher.start();
16 }
17 //Omitted details

```

Figure 18 – Report of violation of restricted classes of Servlet with Thread.



Source: Elaborated by author.

Source Code 11 – Refactored code excerpt - Java with Servlet

```

11 //Omitted details
12 public void init() throws ServletException {
13     searcher = new CRS4GAETHread(this);
14     searcher.start();
15 }
16 //Omitted details

```

⁷ available at <<https://bitbucket.org/marcosborges1/servletwiththread>>

⁸ available at <<http://servletwiththread.appspot.com>>

7.3 PEBBLE

Pebble⁹ is a lightweight, open source, Java EE blogging tool. Its content is stored as XML files on disk and served up dynamically without needing to install a database. The maintenance and administration are performed through a web browser. To run, the application needs at least Java 6 and Servlet 2.5.

The results of CRSAnalyzer can be seen in Figure 19. The leftmost side of the Figure shows that from the 685 Java application files, 46 contained references to restricted classes. In these 46 files, 362 violations (rightmost side) of restricted classes have been detected, which included the classes *File* (328 times), *FileOutputStream* (13 times), *FileWriter* (18 times), and *Thread* (3 times).

Figure 20 shows the violations for each restricted class and specific detection type. For the *File* class, for instance, there were found 159 variable declarations, 14 method declarations and 155 instantiations. For the *FileOutputStream* class, there were 4 variable declarations and 9 instantiations, and for the *FileWriter* class, 18 variable declarations. Finally the *Thread* class presented 3 instantiations.

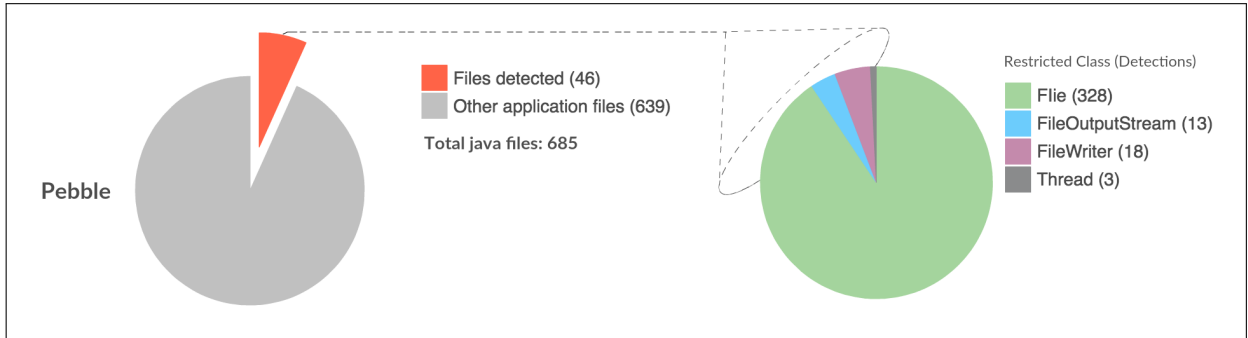
Figure 21 summarizes the modifications performed after the refactorings. The rightmost side shows the new abstract target classes and the new communication classes introduced in the application. The number of abstract target classes used in the application is tied to the number of variable declaration and method declaration detections of a restricted class, while the number of communication classes are related to the number of instantiation detections. For instance, the *FileOutputStream* class has 4 variable declarations and 9 instantiations, so its abstract target class and communication class will be used 4 and 9 times, respectively. Thus, it can be seen that summing up the use of the classes *CRSFile* (173), *CRS4GAEFile* (135), *CRSFileOutputStream* (4), *CRS4GAEFileOutputStream* (9), *CRS4GAEFileWriter* (18) and *CRS4GAETHread* (3), it was reached the total number of code modifications (362) performed based on the 4 refactorings available at CRSRefactor.

The migrated application has also been successfully tested in the local environment and in the cloud using its own tests. It was checked whether the files were created, written and read in the GAE administrator, even with the new communication classes that abstracted the storage directories.

⁹ available at <<https://sourceforge.net/projects/pebble/>>

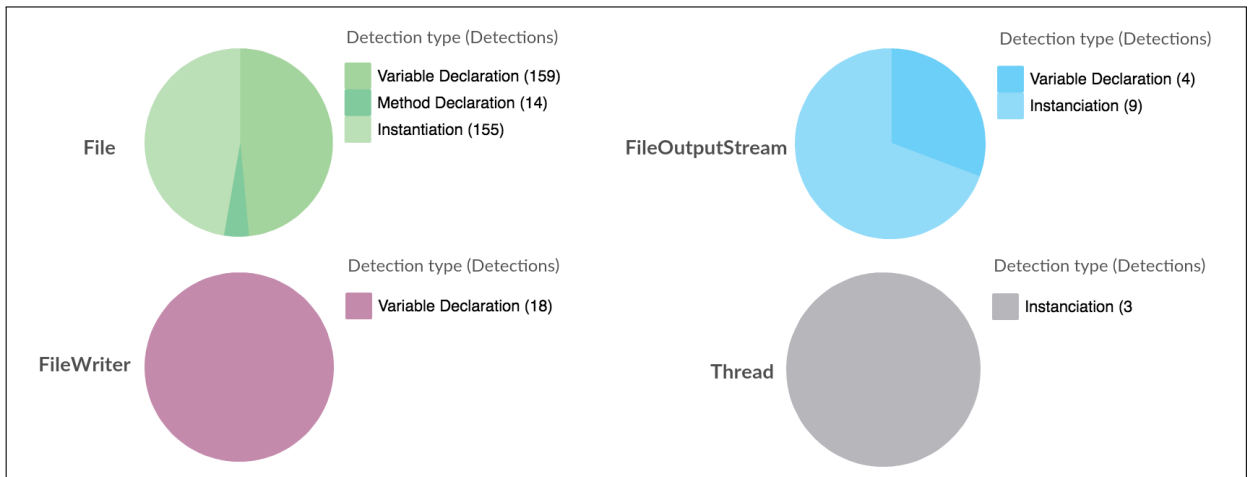
The code and the link for the migrated Pebble application are available at, respectively, Bitbucket¹⁰ and the GAE¹¹.

Figure 19 – Using CRSAnalyzer in the Pebble application project.



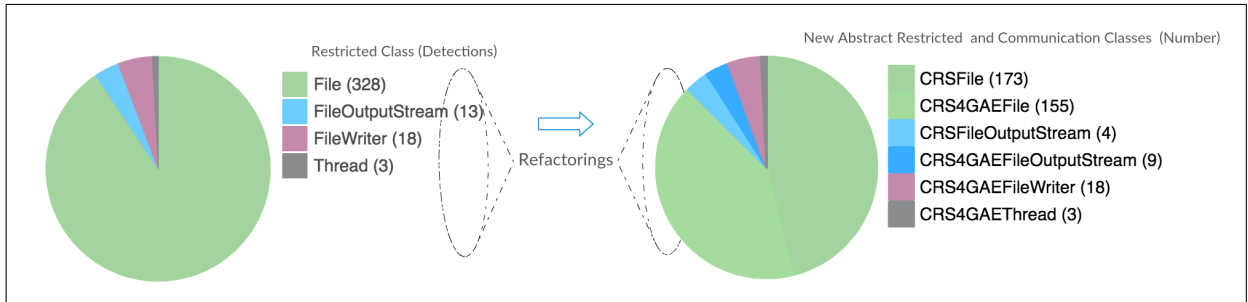
Source: Elaborated by author.

Figure 20 – Restrictions detected by classes.



Source: Elaborated by author.

Figure 21 – New abstract target and communication classes.



Source: Elaborated by author.

¹⁰ available at <<https://bitbucket.org/marcosborges1/pebble>>

¹¹ available at <<http://sincere-axon-148921.appspot.com>>

7.4 CHAPTER SUMMARY

This chapter demonstrated the applicability of the CRS4GAE tool in the migration to three Java web applications. Two applications were simple and useful to show how the code is changed after the refactoring, while the other more complex one was used to show how the restriction identification and refactoring application affect several code artifacts.

Finally, the next chapter shows the conclusion of this dissertation, as well as some considerations and future work.

8 CONCLUSION

This work presented Cloud Restriction Solver (CRS), a novel semi-automatic approach to aid the migration of legacy applications to PaaS environments by avoiding the platform restrictions through user-defined refactorings. The approach, which is cloud-independent, is composed of two main phases: the cloud restriction identification, which identifies possible piece of codes that violate restrictions in the cloud, and refactoring execution, which updates the identified restricted code by equivalent codes based on the cloud. The work also described the open source framework that implements the CRS approach, detailing the architecture of its two engines (CRSAnalyzer and CRSRefactor) used in the phases of the approach. In addition, a general framework instantiation process, which is able to be applied to any PaaS cloud that has constraints, was described. From that general process, a specific one was generated for the Google App Engine, which detailed the use of the engines of the framework. At the end of that process the tool CRS4GAE was generated and used to migrate successfully three Java web applications that originally violated some cloud restrictions.

8.1 BENEFITS

An important benefit of the proposed approach is that it fosters the software reuse in many aspects. Firstly, from the company's perspective, the approach enables the reuse of the application in the selected PaaS environment, thus eliminating the necessity of creating a new version for that cloud platform and, consequently, cutting costs. Secondly, a tool created by the instantiation of the CRS, like the CRS4GAE presented in this work, can be reused to migrate several applications.

Thirdly, considering the developer's point of view, both identification and refactoring engines are open source and can be extended with new restriction identification rules and refactoring operations, respectively, thus reducing the developer's effort for creating new automatic migration mechanisms and increasing the number of possible applications to be migrated. At last, but not least, the CRS itself reuses the AutoRefactor tool to create the identification and refactoring engines. Therefore, reuse is in the essence of the proposed approach.

8.2 LIMITATIONS

Although the results presented in the migration of the applications to the cloud are promising, there are still some limitations that have to be overcome, such as the fact that the identification engine analyzes only the application directory, which means that if the application uses third-part libraries, the migration can be affected. Another limitation to be highlighted is that refactorings, despite returning the data types and objects expected, may be subject to failure, and the developer has two possible alternatives: either manually correcting the application code or creating a rule in the refactoring engine for his/her need.

Finally, as the Eclipse provides two environments for the creation of plugins (developer and runtime workbenches), which require a considerable amount of memory when they are running, possibly machines with small computational power can not be used. Moreover, to test the implemented refactorings, it is necessary to run the generated plugin several times, which can be very costly on the first times, however this effort is reused in many parts of the application.

8.3 FUTURE WORK

As future work, it is planned to instantiate CRS to another PaaS providers and to migrate other applications to those cloud platforms. In addition, it is intended to improve the existing CRS4GAE tool, both in increasing the number of violation detection and refactoring implemented and in the search for new technical solutions. It is also intended to create other categories of primitive refactorings that privilege the renaming of variables and methods, thus making possible the creation of composite refactorings by gathering class adaptation and variable renaming, for instance. Finally, it is aimed at migrating bigger applications to evaluate the effort on using the proposed approach in that context.

REFERENCES

- AHO, A. V.; ULLMAN, J. D. **The theory of parsing, translation, and compiling**. New Jersey: Prentice-Hall, 1972.
- ANDRIKOPOULOS, V.; BINZ, T.; LEYMANN, F.; STRAUCH, S. How to adapt applications for the cloud environment. **Computing**, v. 95, n. 6, p. 493–535, Jun. 2013. Available at: <<https://link.springer.com/article/10.1007/s00607-012-0248-2>>. Access on: 18 Nov. 2016.
- ARMBRUST, M. et al. A view of cloud computing. **Communications of the ACM**, New York, v. 53, n. 4, p. 50–58, Apr. 2010. Available at: <<http://dl.acm.org/citation.cfm?id=1721672>>. Access on: 18 Nov. 2016.
- BUYYA, R. et al. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. **Future Generation computer systems**, Amsterdam, v. 25, n. 6, p. 599–616, Jun. 2009. Available at: <<http://dl.acm.org/citation.cfm?id=1529211>>. Access on: 13 Sep. 2016.
- COSTA, C. H. et al. Supporting partial database migration to the cloud using non-intrusive software adaptations: An experience report. In: EUROPEAN CONFERENCE ON SERVICE-ORIENTED AND CLOUD COMPUTING, 567., 2015, Barcelona. **Electronic proceedings...** Barcelona:Springer, 2015. p. 238-248. Available at: <https://link.springer.com/chapter/10.1007/978-3-319-33313-7_18>. Access on: 05 Dec. 2016.
- ECLIPSE. **Abstract Syntax Tree**. 2006. Available at: <http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/>. Access on: 18 Jul. 2016.
- FOKAEFS, M. et al. Jdeodorant: identification and application of extract class refactorings. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 33. , 2011, Honolulu. **Electronic proceedings...** Honolulu:ACM, 2011. p. 1037-1039. Available at: <<http://dl.acm.org/citation.cfm?id=1985989>>. Access on: 12 Feb. 2016.
- FOSTER, I. et al. Cloud computing and grid computing 360-degree compared. In: GRID COMPUTING ENVIRONMENTS WORKSHOP, 1. , 2008, Austin. **Electronic proceedings...** Austin:IEEE, 2008. p. 1-10. Available at: <<http://ieeexplore.ieee.org/document/4738445/>>. Access on: 10 Sep. 2016.
- FOWLER, G. A.; WORTHEN, B. The internet industry is on a cloud – whatever that may mean. **The Wall Street Journal**, New York, 26 Mar. 2009. Internet. Available at: <<http://www.wsj.com/articles/SB123802623665542725>>. Access on: 01 Jun. 2016.
- FOWLER, M.; BECK, K. **Refactoring: improving the design of existing code**. Boston: Addison-Wesley, 1999.
- FREY, S.; HASSELBRING, W. The cloudmig approach: Model-based migration of software systems to cloud-optimized applications. **International Journal on Advances in Software**, v. 4, n. 3 and 4, p. 342–353, Jan. 2011. Available at: <http://oceanrep.geomar.de/14431/1/soft_v4_n34_2011_8.pdf>. Access on: 12 Oct. 2016.
- FREY, S.; HASSELBRING, W.; SCHNOOR, B. Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. **Journal of Software: Evolution and Process**, v. 25, n. 10, p. 1089–1115, Oct. 2013. Available at: <<http://onlinelibrary.wiley.com/doi/10.1002/smr.582/abstract>>. Access on: 06 Sep. 2016.

GAMMA, E. **Design patterns**: elements of reusable object-oriented software. India: Pearson Education, 1995.

GARTNER. **Gartner says worldwide public cloud services market to grow 18 percent in 2017**. 2017. Available at: <<http://www.gartner.com/newsroom/id/3616417>>. Access on: 04 Apr. 2017.

GOOGLE. **Architecture**: web application on google app engine. 2016. Available at:<<https://cloud.google.com/solutions/architecture/webapp>>. Accessed: 2016-07-01.

JADEJA, Y.; MODI, K. Cloud computing-concepts, architecture and challenges. In: COMPUTING, ELECTRONICS AND ELECTRICAL TECHNOLOGIES (ICCEET), 2., 2012, Dehradun. **Electronic proceedings...** Dehradun:IEEE, 2012. p. 877-880. Available at: <<http://ieeexplore.ieee.org/document/6203873/>>. Access on: 16 Feb. 2016.

JAMSHIDI, P.; AHMAD, A.; PAHL, C. Cloud migration research: a systematic review. **IEEE Transactions on Cloud Computing**, v. 1, n. 2, p. 142–157, Oct. 2013. Available at: <<http://ieeexplore.ieee.org/document/6624108/>>. Access on: 08 Jan. 2016.

KWON, Y.-W.; TILEVICH, E. Cloud refactoring: automated transitioning to cloud-based services. **Automated Software Engineering**, v. 21, n. 3, p. 345–372, Oct. 2014. Available at: <<https://link.springer.com/article/10.1007/s10515-013-0136-9>>. Access on: 08 Sep. 2016.

MAENHAUT, P.-J. et al. Migrating medical communications software to a multi-tenant cloud environment. In: INTEGRATED NETWORK MANAGEMENT (IM), 14., 2013, Ghent. **Electronic proceedings...** Ghent:IEEE, 2013. p. 900-903. Available at: <<http://ieeexplore.ieee.org/document/6573107/>>. Access on: 04 Feb. 2016.

MAENHAUT, P.-J. et al. Migrating legacy software to the cloud: approach and verification by means of two medical software use cases. **Software: Practice and Experience**, v. 46, n. 1, p. 31–54, Jan. 2016. Available at: <<http://onlinelibrary.wiley.com/doi/10.1002/spe.2320/abstract>>. Access on: 08 Sep. 2016.

MARSTON, S. et al. Cloud computing—the business perspective. **Decision support systems**, v. 51, n. 1, p. 176–189, Feb. 2011. Available at: <<http://dl.acm.org/citation.cfm?id=1943810>>. Access on: 08 Sep. 2016.

MELL, P.; GRANCE, T. **The NIST definition of cloud computing**. 2011. Available at: <<http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf>>. Access on: 17 Jul. 2016.

MENDONÇA, N. C. Architectural options for cloud migration. **Computer**, v. 47, n. 8, p. 62–66, May. 2014. Available at: <<http://ieeexplore.ieee.org/document/6879750/>>. Access on: 05 Sep. 2016.

MKAOUER, M. W. et al. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, 29., 2014, Vasteras. **Electronic proceedings...** Vasteras:ACM, 2014. p. 331-336. Available at: <<http://dl.acm.org/citation.cfm?id=2642965>>. Access on: 20 Sep. 2016.

MOHAGHEGHI, P.; SÆTHER, T. Software engineering challenges for migration to the service cloud paradigm: Ongoing work in the remics project. In: IEEE WORLD CONGRESS ON

SERVICES, 3. , 2011, Washington. **Electronic proceedings...** Washington:IEEE, 2011. p. 507-514. Available at: <<http://ieeexplore.ieee.org/document/6012736/>>. Access on: 1 Feb. 2016.

PRABHAKARAN, P. **Efficiently migrating Java/JEE prototype application to Google App Engine PaaS Cloud**. 2014. 107 f. Dissertation (MSc in Cloud Computing) - School of Computing, National College of Ireland, Dublin, 2012. Available at: <<http://trap.ncirl.ie/1837/>>. Access on: 23 Nov. 2016.

PRESSMAN, R.; MAXIM, B. **Engenharia de software**. 8. ed. Porto Alegre: McGraw Hill Brasil, 2016.

PUTHAL, D. et al. Cloud computing features, issues, and challenges: a big picture. In: COMPUTATIONAL INTELLIGENCE AND NETWORKS (CINE), 8. , 2015, Bhubaneswar. **Electronic proceedings...** Bhubaneswar:IEEE, 2015. p. 116-123. Available at: <<http://ieeexplore.ieee.org/document/7053814/>>. Access on: 16 Mar. 2016.

RAI, R.; MEHFUZ, S.; SAHOO, G. Efficient migration of application to clouds: Analysis and comparison. **GSTF Journal on Computing (JoC)**, v. 3, n. 3, p. 58, Dec. 2013. Available at: <<http://dl6.globalstf.org/index.php/joc/article/download/507/524>>. Access on: 08 Sep. 2016.

RAI, R.; SAHOO, G.; MEHFUZ, S. Exploring the factors influencing the cloud computing adoption: a systematic study on cloud migration. **SpringerPlus**, v. 4, n. 1, p. 197, Apr. 2015. Available at: <<https://link.springer.com/article/10.1186/s40064-015-0962-2>>. Access on: 04 Sep. 2016.

RIMAL, B. P.; CHOI, E.; LUMB, I. A taxonomy and survey of cloud computing systems. In: INTERNATIONAL JOINT CONFERENCE ON INC, IMS AND IDC, 5., 2009, Washington. **Electronic proceedings...** Washington:ACM, 2009. p. 44-51. Available at: <<http://dl.acm.org/citation.cfm?id=1684085>>. Access on: 05 Dec. 2016.

SAADEH, E.; KOURIE, D.; BOAKE, A. Fine-grain transformations to refactor uml models. In: WARM UP WORKSHOP FOR ACM/IEEE ICSE, 6., 2009, Cape Town. **Electronic proceedings...** Cape Town:ACM, 2009. p. 45-51. Available at: <http://dl.acm.org/ft_gateway.cfm?id=1527048>. Access on: 20 Sep. 2016.

SANDERSON, D. **Programming google app engine with JAVA: build & run scalable java applications on google's infrastructure**. Sebastopol: O'Reilly Media, 2015.

SCANDURRA, P. et al. Challenges and assessment in migrating it legacy applications to the cloud. In: MAINTENANCE AND EVOLUTION OF SERVICE-ORIENTED AND CLOUD-BASED ENVIRONMENTS (MESOCA), 3., 2015, Bremen. **Electronic proceedings...** Bremen:IEEE, 2015. p. 7-14. Available at: <<http://ieeexplore.ieee.org/document/7328120/>>. Access on: 23 Feb. 2016.

SILVA, D.; TERRA, R.; VALENTE, M. T. Recommending automated extract method refactorings. In: INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION, 22., 2014, Hyderabad. **Electronic proceedings...** Hyderabad:ACM, 2014. p. 146-156. Available at: <<http://dl.acm.org/citation.cfm?doid=2597008.2597141>>. Access on: 21 Jun. 2016.

SOSINSKY, B. **Cloud computing bible**. Hoboken: John Wiley & Sons, 2010. v. 762.

SZÓKE, G. et al. Faultbuster: An automatic code smell refactoring toolset. In: SOURCE CODE ANALYSIS AND MANIPULATION (SCAM), 12., 2015, Bremen. **Electronic proceedings...** Bremen:IEEE, 2015. p. 253-258. Available at: <<http://ieeexplore.ieee.org/document/7335422/>>. Access on: 16 Feb. 2016.

TERRA, R. et al. Recommending refactorings to reverse software architecture erosion. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR), 16., 2012, Bremen. **Electronic proceedings...** Bremen:IEEE, 2012. p. 335-340. Available at: <<http://ieeexplore.ieee.org/document/6178900/>>. Access on: 08 Feb. 2016.

TRAN, V. et al. Application migration to cloud: a taxonomy of critical factors. In: INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING FOR CLOUD COMPUTING, 2., 2011, Honolulu. **Electronic proceedings...** Honolulu:ACM, 2011. p. 22-28. Available at: <<http://dl.acm.org/citation.cfm?id=1985505>>. Access on: 27 Dec. 2016.

VAQUERO, L. M. et al. A break in the clouds: towards a cloud definition. **ACM SIGCOMM Computer Communication Review**, New York, v. 39, n. 1, p. 50–55, Jan. 2008. Available at: <<http://dl.acm.org/citation.cfm?id=1496100>>. Access on: 04 Apr. 2016.

VASCONCELOS, M.; MENDONÇA, N. C.; MAIA, P. H. M. Cloud detours: A non-intrusive approach for automatic software adaptation to the cloud. In: . [S.l.: s.n.]. In: EUROPEAN CONFERENCE ON SERVICE-ORIENTED AND CLOUD COMPUTING, 9306. , 2015, Taormina. **Electronic proceedings...** Taormina:Springer, 2015. p. 181-195. Available at: <https://link.springer.com/chapter/10.1007/978-3-319-24072-5_13>. Access on: 12 Jun. 2016.

VU, Q. H.; ASAL, R. Legacy application migration to the cloud: Practicability and methodology. In: WORLD CONGRESS ON SERVICES, 8., 2012, Honolulu. **Electronic proceedings...** Honolulu:IEEE, 2012. p. 270-277. Available at: <<http://ieeexplore.ieee.org/document/6274061/>>. Access on: 04 Feb. 2016.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. **Journal of internet services and applications**, v. 1, n. 1, p. 7–18, Apr. 2010. Available at: <<https://link.springer.com/article/10.1007/s13174-010-0007-6>>. Access on: 01 Sep. 2016.

ZHAO, J.-F.; ZHOU, J.-T. Strategies and methods for cloud migration. **International Journal of Automation and Computing**, v. 11, n. 2, p. 143–152, Apr. 2014. Available at: <<https://link.springer.com/article/10.1007/s11633-014-0776-7>>. Access on: 05 Jun. 2016.