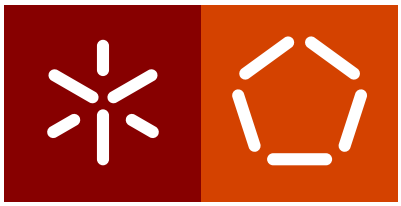
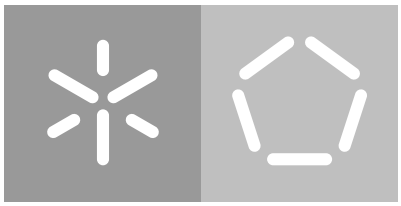


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago Fernando Santos Braga Fernandes

Suporte para Refatorização Automática de Lógica de Negócio baseada em Modelos

Julho de 2017



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Tiago Fernando Santos Braga Fernandes

Suporte para Refatorização Automática de Lógica de Negócio baseada em Modelos

Dissertação de Mestrado
Mestrado em Engenharia Informática

Dissertação supervisionada por
António Nestor Ribeiro
Orientador na empresa
David Varela Nunes

Julho de 2017

AGRADECIMENTOS

Apresentar publicamente esta dissertação de mestrado é uma enorme responsabilidade e apenas foi possível graças à colaboração incondicional de quem a apoiou. Não teria chegado a este ponto sem a ajuda e a orientação de pessoas extraordinárias, a quem, reconhecido, agradeço publicamente:

- Ao Professor António Nestor Ribeiro, meu orientador de mestrado, por coordenar esta dissertação, pela experiência e *know-how* que a tornaram possível, pela confiança depositada, pelo *feedback* atento e pelas sugestões e estímulos permanentes;

- Ao David Nunes, *Head of Product Architecture* na OutSystems, meu co-orientador de mestrado, pela definição do tema, pelo acompanhamento em todo o processo, por balizar o contexto e objetivos da dissertação e fazer com que esta chegasse a bom porto. Pelas conversas construtivas e pelas inesquecíveis sessões de *brainstorming* — obrigado por me teres deixado experienciar o modo extraordinário como abordas e solucionas os problemas. Além de tudo, obrigado pelo tempo dispendido, que foi fundamental para o meu trabalho;

- Ao Hugo Lourenço e ao Luiz Santos, ambos integrados em equipas de *R&D* que, apesar de não orientarem formalmente esta dissertação, dispenderam de parte do seu tempo pessoal em conversas construtivas para me encorajarem na elaboração da dissertação e para me guiarem na plataforma OutSystems;

- Ao Departamento de Informática da Universidade do Minho, por me formar com uma educação sólida que contribuiu, sem dúvida, para a minha formação profissional;

- À OutSystems, por apostar na minha dissertação de mestrado, por me proporcionar um ambiente acolhedor e por me dar a conhecer tantas pessoas extraordinárias. Estou, sem dúvida, rodeado pelos melhores;

- A todos aqueles com quem me relacionei e com quem fiz amizades, nas equipas de Braga, em especial nas equipas *Collaboration Team* e *Integration Team*, pelos momentos de descontração e lazer que todos fomentaram e pelas atividades extra-laborais que partilhamos juntos — os nossos *Status Update* são inesquecíveis;

- Aos meus pais, pela educação e pelo apoio que me deram e por fazerem de mim a pessoa que sou hoje, e ao meu irmão, pelo apoio e pelas cumplicidades partilhadas;

- Aos meus amigos, por me acompanharem nos momentos mais divertidos e nos mais difíceis. E à Inês, minha namorada, pelo apoio incondicional, pelo amparo e por acreditar sempre no meu sucesso;

- A todos aqueles que, apesar de não estarem nomeados nesta lista, me ajudaram e acompanharam nesta viagem.

ABSTRACT

Software's structure profoundly affects its development and maintenance costs. Poor software's structure may lead to well-known design flaws, such as large modules or long methods.

A possible approach to reduce a module's complexity is the Extract Method refactoring technique. This technique allows the decomposition of a large and complex method into smaller and simpler ones, while reducing the original method's size and improving its readability and comprehension.

Nowadays, it's almost mandatory that *Integrated Development Environments (IDEs)* support this and other refactoring techniques. Despite the wide availability of the extract method operation on IDEs, the identification of portions of code that are worthwhile to refactor still relies mostly on developer knowledge and expertise.

Thus, the purpose of this dissertation is to empower the OutSystems platform with a system that is able to analyse modules complexity and automatically suggest Extract Method refactoring opportunities.

Keywords: Refactoring, Program Slicing, Code Complexity Metrics, Low-code, OutSystems, Graph Theory

RESUMO

A estrutura das aplicações de *software* afeta significativamente os seus custos de desenvolvimento e de manutenção. Uma fraca estruturação do código pode levar a problemas de *design*, como é o exemplo de módulos de grande dimensão, ou *Long Methods*.

Uma das abordagens para reduzir a complexidade de módulos é a aplicação da técnica de refatorização *Extract Method*. Esta técnica permite extrair comportamento de um método complexo em métodos mais simples, reduzindo o tamanho do método original e aprimorando a sua leitura e compreensão.

Hoje em dia, é quase imperativo que os IDEs suportem esta e outras técnicas de refatorização. Apesar dos IDEs disponibilizarem a operação de extração de métodos, a identificação de secções do código potencialmente úteis para refatorizar continua a ser uma operação que depende do conhecimento e experiência do programador.

Assim, o objetivo desta dissertação é prover a plataforma OutSystems de um sistema capaz de analisar a complexidade de módulos e automaticamente sugerir oportunidades de refatorização *Extract Method*.

Palavras-chave: Refatorização, *Program Slicing*, Métricas de Complexidade, *Low-code*, OutSystems, Teoria de Grafos

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

SIGLAS E ACRÓNIMOS

ABC Assignments, Branches, Conditionals.
ASCMM Automated Source Code Maintainability Measure.

CC Cyclomatic Complexity.
CCC Coupled Cyclomatic Complexity.
CFG Control Flow Graph.

DSL Domain-Specific Language.

IDE Integrated Development Environment.

LOC Lines of Code.

OMG Object Management Group.

PDG Program Dependence Graph.

RAD Rapid Application Development.
RAW Read After Write.
REST Representational State Transfer.

SDG System Dependence Graph.
SIG Software Improvement Group.
SOAP Simple Object Access Protocol.

TCC Total Cyclomatic Complexity.
TI Tecnologias da Informação.

UI User Interface.

WAR Write After Read.
WAW Write After Write.

1

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

A dimensão, complexidade e incoerência de um sistema de *software* exigem custos acrescidos de manutenção e investimento excessivo de tempo para procedimentos de implementação e de gestão. Múltiplos estudos apontam causas que limitam o funcionamento e evolução de qualquer sistema de *software*: Gill e Kemerer [GK91] referiram os constrangimentos provocados pelo elevado grau de complexidade; Banker et al. [BDKZ93] mostraram as dificuldades causadas pelos sistemas de grande dimensão; Meyers e Binkley [MB07] analisaram os problemas causados pela falta de coesão dos sistemas.

Durante o ciclo de vida de *software*, este é constantemente alterado. Essas alterações são originadas pela correção de defeitos, introdução de novas funcionalidades, adoção de novas arquiteturas, práticas ou equipas de desenvolvimento. Todas estas alterações, ao longo do tempo, podem conduzir a módulos complexos, pouco elegantes e incompreensíveis.

Nesse contexto, a refatorização, enquanto atividade fundamental no desenvolvimento de *software*, assume um papel determinante. Quando corretamente aplicada, permite o restabelecimento do equilíbrio e coerência do projeto, contribuindo para a melhoria da qualidade do sistema e para a facilidade da sua compreensão.

A refatorização define-se pela execução de alterações sistemáticas na estrutura interna do *software* para facilitar a sua compreensão, sem alterar o comportamento observado [FBB+00], com o objectivo de melhorar a qualidade do *software* existente. A refatorização tem como objetivo tornar o código mais simples de compreender, por exemplo, através da extração de blocos de código para métodos mais simples (*Extract Method*), ou através da remoção de código duplicado (*Duplicated Code*).

Num produto de *software*, a identificação de oportunidades de refatorização depende diretamente da complexidade desse produto, que pode dificultar a sua leitura e compreensão. Naturalmente, quanto mais complexa for uma solução, mais difícil é de a compreender e mais agravado é o custo de a manter. Com o objetivo de aprimorar a clareza do código e reduzir os custos de manutenção, é necessário analisar se um módulo é de difícil compre-

ensão e leitura. No entanto, esta análise de qualidade e posterior intervenção no sistema são ações realizadas, tipicamente, por peritos na área.

A OutSystems apresenta uma alternativa aos modelos tradicionais de desenvolvimento de *software*. Através da sua plataforma proprietária (OutSystems Platform) e de uma *Domain-Specific Language (DSL)* visual, permite o desenvolvimento de aplicações de grande complexidade de forma simples. A plataforma OutSystems é composta por um conjunto de funcionalidades que permitem que os programadores se foquem na parte funcional das aplicações, e se abstraiam de detalhes de implementação, o que resulta num aumento de produtividade e qualidade do *software* desenvolvido. Apesar das notórias melhorias proporcionadas pela plataforma OutSystems, a componente funcional do programa, dependente da programador, pode conter alguns problemas que afetam negativamente a qualidade do programa. Assim, é necessária a aplicação de técnicas de refatorização para manter o equilíbrio e a coerência dos projetos de *software*.

O IDE da OutSystems, tal como outros IDEs estabelecidos na indústria, dispõe de primitivas de refatorização como o *Extract Method*. Contudo, a identificação de oportunidades de refatorização mantém-se um processo dependente da intervenção humana. Assim, é imperativa a identificação automática de oportunidades de refatorização, num contexto em que o IDE supervisione e sugira alterações no código.

1.2 Motivação e Objetivos

A aplicação de técnicas de refatorização é fundamental para controlar a complexidade de *software*. No entanto, a refatorização é uma atividade que necessita de alterar o código e, se não for cuidadosa, pode originar *bugs*, atrasando o desenvolvimento. Além disso, a refatorização apresenta ainda maior grau de risco se for praticada sem método ou de forma desorganizada.

A aplicação de técnicas de refatorização é uma operação de elevado risco [FBB+oo], o que pode levar a que os programadores se mostrem relutantes à sua prática. Tal comportamento é compreensível devido a vários fatores: à falta de domínio das técnicas de refatorização; ao facto de o benefício obtido se revelar apenas a longo prazo; devido a ser uma atividade complementar ao desenvolvimento de funcionalidades; por se identificar como uma operação de elevado risco. De igual modo, os projetos desenvolvidos na plataforma OutSystems podem beneficiar da aplicação de algumas técnicas de refatorização, dada a existência de módulos complexos e de grande dimensão (*Long Methods*). Segundo Fowler et al. [FBB+oo], uma das técnicas mais utilizadas para reduzir a complexidade de módulos é a primitiva *Extract Method*. Esta carateriza-se pela extração de código de uma rotina para uma sub-rotina, com o objetivo de reduzir a sua complexidade.

Como foi referido, a refatorização é uma operação de risco e pode contrair prejuízos para a equipa de desenvolvimento. Assim, uma refatorização aplicada externa e exclusivamente pelo programador deve ser auxiliada pelo próprio IDE, através da exposição de sugestões e respetivos benefícios. O auxílio do IDE nos processos de identificação e aplicação automáticas de técnicas de refatorização garante uma intervenção metódica, que nem sempre é garantida por um perito na área. Além disso, o facto de ser o IDE a sugerir e a alterar o *software*, torna todo o processo de refatorização mais rápido e eficaz. Assim, é essencial a aplicação metódica de técnicas de refatorização, de forma automática, para que o sistema mantenha a qualidade e integridade esperada.

Presentemente, apesar da evolução no setor da refatorização, comprovada, aliás, pela vasta bibliografia disponível, a identificação de oportunidades de extração de comportamento é ainda escassa e pouco suportada.

Tal como outras organizações, a OutSystems entende que a refatorização é um processo que permite manter o equilíbrio e coerência dos projetos. A identificação de oportunidades de refatorização e posterior intervenção permite que os programadores, na plataforma OutSystems, estruturem melhor as suas aplicações, sem correr riscos de alterar o comportamento da lógica de negócio do projeto.

Assim, o objetivo e motivação desta dissertação é explorar um conjunto de metodologias que promovam sugestões automáticas de extração de módulos, auxiliando os programadores na manutenção do *software* produzido na plataforma OutSystems. Nesse sentido, os objetivos a serem concretizados com esta dissertação são os seguintes:

- Validar as abordagens de cálculo de métricas de complexidade de módulos para identificar os módulos de grande dimensão e complexidade (*Long Methods*), que potencialmente beneficiariam de *Extract Method*.
- Adaptar e validar as abordagens de identificação de oportunidades de *Extract Method* e compreender as que melhor se enquadram no contexto da plataforma OutSystems, com o objetivo de reduzir a ocorrência de *Long Methods*.
- Definir um conjunto de métricas que permitam aferir a qualidade dos módulos sugeridos.

A expectativa é que esta ferramenta possa, futuramente, ser disponibilizada aos programadores que utilizam a plataforma OutSystems. Com essa disponibilização pretende-se diminuir o custo de refatorização e possibilitar que os programadores OutSystems menos experientes possam aumentar o nível e qualidade de código das suas aplicações através de refatorizações eficazes.

1.3 METODOLOGIA DE PESQUISA

Com o intuito de cumprir os objetivos assinalados na secção 1.2 (Motivação e Objetivos) e conduzir esta dissertação no devido rigor, elencaram-se algumas tarefas que delinearam o percurso desta dissertação. Nesse sentido, identificou-se o problema descrito na secção 1.1 (Contextualização), que levou ao estudo do trabalho relacionado no capítulo 3 (TRABALHO RELACIONADO). De seguida, com recurso à bibliografia estudada, elaborou-se uma proposta de solução, posteriormente implementada e avaliada.

As etapas que orientaram esta dissertação foram, progressivamente, as seguintes:

1. Caraterização do domínio do problema e identificação de oportunidade na aplicação do mesmo no contexto empresarial. Esta etapa permitiu definir o âmbito da dissertação e balizar os objetivos que a mesma pretende resolver.
2. Estudo acerca do trabalho desenvolvido no âmbito da identificação e sugestão automática de oportunidades de refatorização de *software*.
3. Proposta de uma solução inicial para validar o trabalho relacionado com o contexto empresarial.
4. Implementação da solução de acordo com os requisitos e objetivos da empresa que acolheu esta dissertação (OutSystems).
5. Análise da implementação, avaliação e discussão de resultados, apontando possíveis caminhos de investigação subsequentes.

1.4 PROPOSTA DE SOLUÇÃO

Esta secção descreve genérica e brevemente a solução proposta, possibilitando uma descrição mais detalhada e minuciosa no capítulo 4 (DETALHES DA SOLUÇÃO).

A plataforma da OutSystems permite desenvolver aplicações numa linguagem baseada num modelo visual de elevada abstração. Os ambientes de desenvolvimento nesta plataforma são de fácil interpretação e alteração. Contudo, à semelhança de outras linguagens de programação, se as intervenções neles não forem cuidadosas, potenciam a ocorrência de sistemas de grande complexidade.

Com o intuito de controlar a evolução destes ambientes, propõe-se uma abordagem automática que permita reduzir a complexidade de ações de lógica de negócio. Para tal, a proposta de solução analisa estaticamente as dependências de controlo e de dados do programa, considerando todos os caminhos possíveis que o mesmo possa seguir, para garantir que a refatorização executada preserva o comportamento funcional do programa [Wei81].

A abordagem escolhida [Maro1] divide o programa em secções, a partir das quais são propostas sugestões para reduzir a complexidade do programa. Desta forma, potencia-se um maior número de alterações no código. Com base nas possíveis refatorizações, o programador seleciona aquela que melhor se enquadra no contexto da lógica do programa.

Assim, uma possível proposta enumera-se, genericamente, nos seguintes passos:

1. Seleção dos módulos que apresentam maior grau de complexidade.
2. Identificação de regiões do programa, a partir das quais é correto extrair a computação de variáveis.
3. Análise das dependências entre os nodos do programa, de forma a identificar quais os elementos de dependem da execução de outros.
4. Identificação de sub-rotinas capazes de efetuar o cálculo das variáveis nas regiões identificadas, com recurso à análise de dependências de controlo e de dados dos nodos a extrair.
5. Sugestão das sub-rotinas que melhor reduzem a complexidade original do programa, sem alterar o seu comportamento inicial.
6. Aplicação da sub-rotina escolhida pelo programador, removendo a computação de uma variável e reduzindo a complexidade do módulo original.

1.4.1 *Requisitos da Solução Proposta*

De forma a adaptar a solução ao ambiente de produção da OutSystems, foram identificados alguns requisitos que devem ser respeitados na solução proposta, que se apresentam de seguida:

- As sugestões propostas devem ser devidamente avaliadas, no sentido de melhorar a legibilidade e compreensão do sistema, e ordenadas por ordem decrescente no ganho obtido que elas representam.
- Deve ser aplicável a ambientes de desenvolvimento reais, dotados de cenários de elevada complexidade.

1.5 ESTRUTURA DO DOCUMENTO

O presente relatório está organizado em 6 capítulos:

O CAPÍTULO 1 introduz o tópico desta dissertação através da exposição do problema que se pretende resolver e esclarecer a relevância da refatorização. Este capítulo termina com a exposição da estrutura do documento.

O CAPÍTULO 2 descreve o contexto empresarial da OutSystems, a partir do qual esta dissertação é conduzida.

O CAPÍTULO 3 expõe o estudo realizado por diversos autores, no âmbito da identificação de oportunidades de refatorização e análise de complexidade de módulos.

O CAPÍTULO 4 descreve, detalhadamente, os processos de implementação da solução.

O CAPÍTULO 5 apresenta um conjunto de módulos, desenvolvidos no seio da empresa OutSystems, que são utilizados como caso de estudo. Os módulos são avaliados no sentido de analisar a capacidade de reduzir a complexidade de programas desenvolvidos na plataforma OutSystems.

O CAPÍTULO 6 conclui o documento e apresenta indicações a ser futuramente estudadas e implementadas.

2

CONTEXTO EMPRESARIAL

O presente capítulo expõe e carateriza o contexto empresarial no qual esta dissertação foi conduzida. Este capítulo tem como principal objetivo clarificar o ambiente de desenvolvimento em OutSystems, e explicar alguns detalhes relevantes, que contribuem para uma melhor compreensão dos exemplos descritos ao longo do documento.

2.1 A EMPRESA OUTSYSTEMS

A empresa OutSystems, fundada em 2001, na cidade de Lisboa, é uma empresa de *software* internacional, atualmente sediada em Atlanta, Estados Unidos da América. Tem como principal foco reduzir custos associados ao desenvolvimento de *software*, com recurso à sua plataforma *low-code* de desenvolvimento rápido (*RAD*). A OutSystems está presente, mundialmente, em mais de 30 países e emprega mais de 400 funcionários nas áreas de engenharia, *marketing*, vendas, etc.

2.2 A PLATAFORMA OUTSYSTEMS

A plataforma OutSystems (Figura 1) permite desenvolver aplicações *web* e *mobile* numa linguagem baseada num modelo visual de abstração elevada e divide-se em dois componentes principais: *Development Environment* (composto por *Service Studio* e *Integration Studio*) e *Platform Server*. A modelação a um nível de abstração mais elevado, permite que os programadores não tenham necessidade de se preocupar com detalhes de implementação e de publicação, resultando em melhorias no tempo de entrega do produto (*time to market*) e respetiva qualidade.

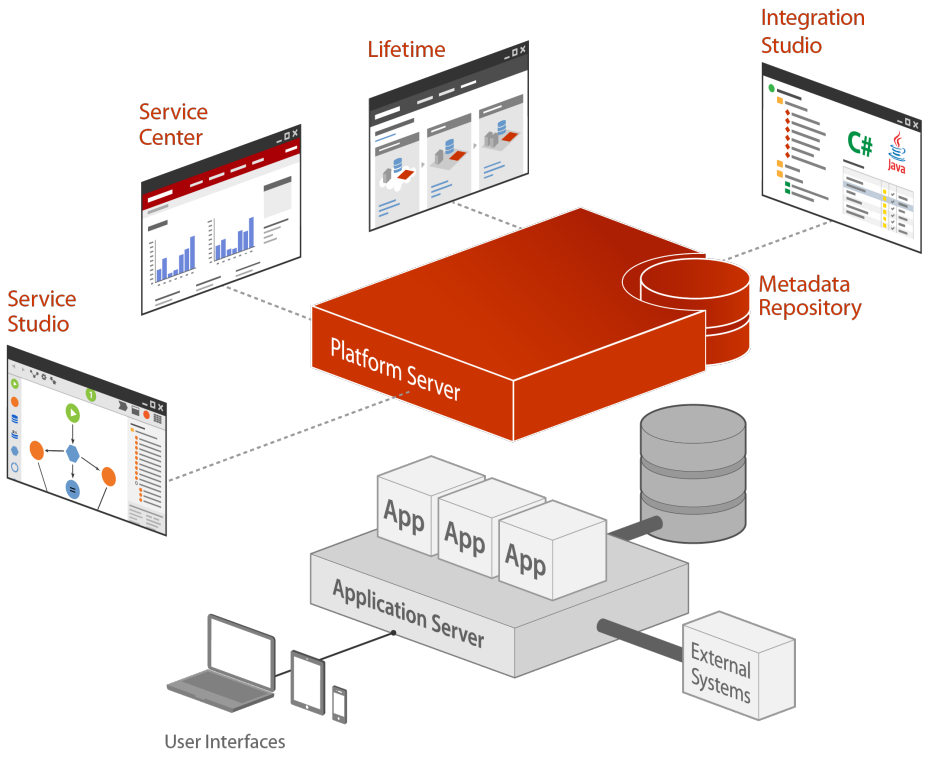


Figura 1.: Plataforma OutSystems

2.2.1 *Service Studio*

O *Service Studio* (Figura 2) é o IDE para a DSL visual da plataforma OutSystems e permite definir processos de negócio, *interfaces* de utilizador, componentes de lógica de negócio, modelos de base de dados e ainda construir e utilizar *web services*, em *Simple Object Access Protocol (SOAP)* e *Representational State Transfer (REST)*, agendar a execução de procedimentos, etc.

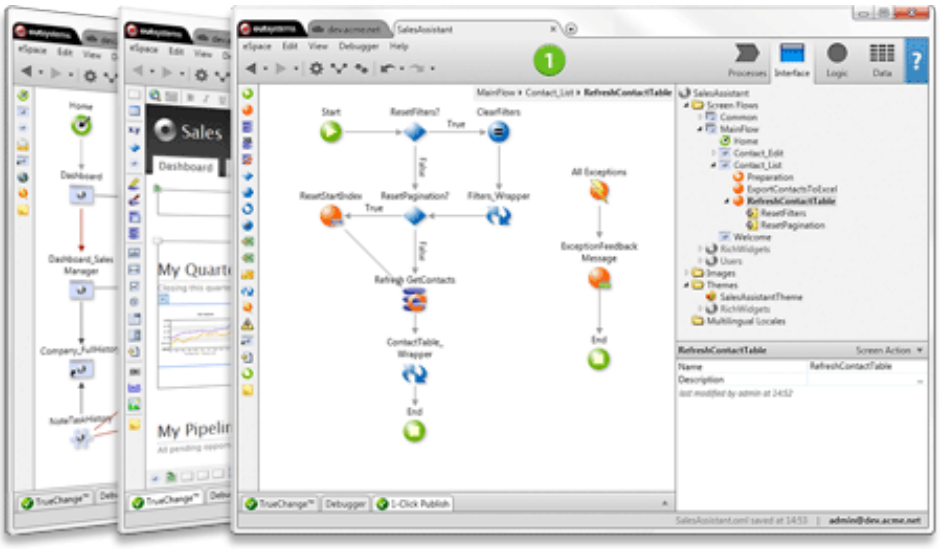


Figura 2.: Ambiente de desenvolvimento no IDE *Service Studio*

2.2.1.1 *Sintaxe da Linguagem*

O *Service Studio* manipula uma DSL visual composta por um conjunto de símbolos, dos quais se apresentam os mais relevantes:

Start - ícone de início, que indica onde o fluxo de negócio inicia.

End - ícone de fim, que indica onde o fluxo de negócio termina.

Server Action - ícone de chamada de uma *server action*, que executa tarefas do lado do servidor.

Aggregate - ícone de agregação, que permite facilmente consultar dados na base base de dados.

SQL - ícone de *SQL*, que permite consultar dados na base de dados, através de *queries* definidas em *SQL*.

If - ícone condicional, que permite executar um determinado fluxo, com base numa condição.

Switch - ícone condicional, que permite agregar um conjunto de múltiplos *Ifs*, e executar um determinado fluxo, com base na definição de múltiplas condições.

For Each - ícone de ciclo, que permite iterar sobre os elementos de uma lista (dada a abstração oferecida pela *DSL*, é possível construir ciclos independentes de listas, com recurso aos dois predicados de controlo anteriormente referidos).

Assign - ícone de atribuição, que atribui um valor a uma variável. Este nodo pode conter múltiplas atribuições (*Assignments*), evitando a criação de múltiplos nodos *Assign* consecutivos.

Comment - ícone de comentário, exibido em ambiente de desenvolvimento (pode ser útil como lembrete).

2.2.2 *Platform Server*

O *Platform Server* é um conjunto de componentes e serviços responsáveis pela transformação das aplicações OutSystems em código nativo e pela compilação do código gerado, ou C# ou Java, dependendo da *stack* tecnológica utilizada.

Em suma, os programadores utilizam o IDE *Service Studio* (Figura 2) para construir as suas aplicações e, quando escolhem publicar a sua aplicação, o *Service Studio* envia a definição do módulo para o *Platform Server* que, por sua vez, gera o código correspondente e publica-o num servidor aplicacional.

3

TRABALHO RELACIONADO

O presente capítulo expõe o trabalho relacionado com a identificação de módulos de grande complexidade e a refatorização automática de programas, áreas relevantes para o tema abordado nesta dissertação.

3.1 TÉCNICAS DE REFATORIZAÇÃO

O livro de Fowler et al. [FBB+00], suporte bibliográfico fundamental desta secção, no que diz respeito ao processo de refatorização, explicou a sua importância, clarificou o modo de identificar *code smells*, catalogou e explicitou as técnicas e processos da sua aplicação e referiu cuidados a ter durante esse processo.

Este trabalho permite concluir que a estruturação do *software* tem um impacto significativo no seu desenvolvimento e manutenção. O desenvolvimento de *software* enquadra-se num ambiente de constante mudança, o que leva a que a estrutura do sistema se afaste da arquitetura inicial. A refatorização de *software* apresenta-se como uma abordagem para melhorar a estrutura e possibilitar uma melhor compreensão do sistema.

A refatorização é uma técnica que permite melhorar o *design* do código, a partir de alterações sistemáticas que não modifiquem o seu comportamento externo [FBB+00]. A execução de alterações simples é benéfica no âmbito da refatorização, pois reduz a probabilidade de introdução de *bugs*.

A refatorização é, habitualmente, motivada pela identificação de *bad smells*[1], ou seja, uma má estruturação de código, por exemplo, um método ou classe de grande dimensão (*Long Method*), código duplicado (*Duplicated Code*) ou um método com demasiados parâmetros de *input* (*Long Parameter List*). A remoção destes *code smells* torna o código mais simples de compreender e alterar.

As técnicas de refatorização podem ser divididas, entre outras, nas seguintes categorias:

- técnicas de extração, que permitem isolar componentes simples a partir de componentes mais complexos (e.g., *Extract Method*, *Extract Class*, etc.);

1 Também conhecido por *code smell*

- técnicas de abstração, que permitem melhorar a estruturação do código (e.g., *Encapsulate Field*, *Strategy pattern*, *Polymorphism*, etc.);
- técnicas de invocação, que permitem simplificar as chamadas (i.e., invocações) de métodos (e.g., *Add Parameter*, *Remove Parameter*, *Parameterize Method*, etc.).

A refatorização possibilita, assim, uma melhor manutenção do sistema, uma vez que, tornando o código mais percetível, facilita a correção de *bugs*. Por outro lado, segundo Fowler et al. [FBB+00], a refatorização agiliza os processos de desenvolvimento do projeto, através da adoção de *design patterns*, tornando o sistema mais flexível.

3.2 PROGRAM SLICING

Uma significativa porção da literatura relacionada com a refatorização assistida estuda o conceito de *program slicing*, ou extração de comportamento. *Program slicing* é uma técnica que permite simplificar programas, originando sub-rotinas secundárias que isolam parte do código, sem afetar o comportamento original. *Slicing* foi inicialmente concebido para facilitar os processos de *debugging* [Wei81], mas rapidamente se tornou útil noutros setores do ciclo de vida de desenvolvimento de *software*. A título de exemplos, verificou-se relevante em ambientes de teste, cálculo de métricas, análise e compreensão, manutenção e paralelismo [Tip95, HH01, TC11].

De acordo com Weiser [Wei81], uma *slice* consiste nas instruções do programa que podem afetar o valor de uma variável v num ponto de interesse p. O par (p, v) é referido como *slice criterion*. Resumidamente, as *slices* são identificadas pelo conjunto de instruções que afetam direta ou indiretamente o valor de uma variável v, baseando-se na análise das dependências de controlo e de dados de um programa. Após a definição de *program slicing* [Wei81], diversas abordagens foram estudadas e propostas, apresentando-se algumas delas ao longo desta secção.

Considerando a definição original de Weiser [Wei81], *static slicing* consiste em todas as instruções que podem afetar o valor de uma variável v. Em *static slicing*, o *slicing criterion* é definido por $C = (p, v)$. *Static slicing* está diretamente associado à computação total de uma variável (*complete computation slice*) e que se refere a todas as instruções que alterem o valor de uma variável num método. Ainda relativamente à informação em tempo de execução do programa, *dynamic slicing* [KL88] tem em consideração um valor específico das variáveis para uma execução do programa, originando, assim, *slices* mais precisas, no contexto do valor dessas variáveis. Em *dynamic slicing*, o *slicing criterion* no programa P, dado um *input* de x é, então, definido por $C = (x, I^q, V)$, onde I é uma instrução na posição q e V é um subconjunto das variáveis de P. Em termos comparativos, as *slices* estáticas são tipicamente mais extensas, mas têm em consideração todas as possíveis execuções e

caminhos do programa. As *slices* dinâmicas são, por sua vez, menos extensas, mas apenas atendem a um determinado valor das suas variáveis. Com o intuito de preencher esta lacuna, *conditioned slicing* [CCL98] permite calcular *slices* e extrair comportamento de acordo com o valor dos predicados de controlo do programa (i.e., nodos de controlo).

Relativamente à direção de fluxo do programa, *backward slicing* entende que uma *slice* contém todas as instruções e predicados de controlo que possam afetar o valor de uma variável num determinado ponto de interesse, enquanto *forward slicing* [BC85] entende que uma *slice* contém todas as instruções e predicados de controlo que podem ser afetados por uma variável, num determinado ponto de interesse. Para qualquer *slicing criterion*, uma *slice* pode ser calculada com recurso a qualquer um dos fluxos. Uma *forward slice* é calculada ignorando as instruções que não podem ser afetadas pelo *slicing criterion*, enquanto uma *backward slice* é calculada ignorando as instruções que não podem afetar o *slicing criterion*.

Relativamente à preservação de sintaxe do programa, *syntax-preserving slicing* pretende simplificar um programa através da remoção de instruções e predicados de controlo sem os modificar. Por outro lado, *amorphous slicing* [HD97] calcula *slices* utilizando transformações sintáticas para simplificar o programa, e.g., alterando os seus predicados de controlo.

Em relação ao alcance das *slices*, *intraprocedural slicing* utiliza uma única rotina para gerar *slices*, enquanto que *interprocedural slicing* [HRB88] gera *slices* que podem tem em consideração múltiplos procedimentos.

Ettinger [Ett07] agrupa a extração de *slices* em duas categorias: metodologias que extraem *slices* baseadas na indicação de instruções pelo utilizador (*arbitrary method extraction*) e metodologias que extraem o comportamento de uma variável.

A primeira proposta de decomposição de um método foi apresentada por Gallagher e Lyle [GL91], onde foi introduzido o conceito de *decomposition slice*, que engloba todas as instruções que alteram, direta ou indiretamente, o valor de uma variável. Um exemplo da proposta pode ser avaliado pela Figura 3 - (b), onde se deduz a dependência entre variáveis do programa da Figura 3 - (a).

Segundo Gallagher e Lyle [GL91], duas *decomposition slices* $S(v)$ e $S(w)$ são consideradas independentes se a sua interseção for um subconjunto vazio, $S(v) \cap S(w) = \emptyset$. Uma *decomposition slice* $S(v)$ é considerada fortemente dependente de $S(w)$, se $S(v)$ for um subconjunto de $S(w)$, $S(v) \subset S(w)$. A título de exemplo, $S(c)$ é fortemente dependente das restantes *decomposition slices* e $S(inword)$ é fortemente dependente de $S(nw)$, pois $S(nw) = \{10, 22\} \cup S(inword)$, onde $S(inword) = \{8, 15, 16, 20, 21\} \cup S(c)$ e $S(c) = \{12, 13, 24\}$. Como pode ser observado pela Figura 3 - (c), as *slices* podem ter instruções partilhadas. Tonella [Ton03] introduziu o conceito de *weak inferences* para representar instruções partilhadas pelas *slices*, i.e., instruções indispensáveis. As instruções indispensáveis de uma *slice* são instruções que pertencem a uma *slice*, mas não devem ser removidas do método original, preservando o comportamento inicial do método.

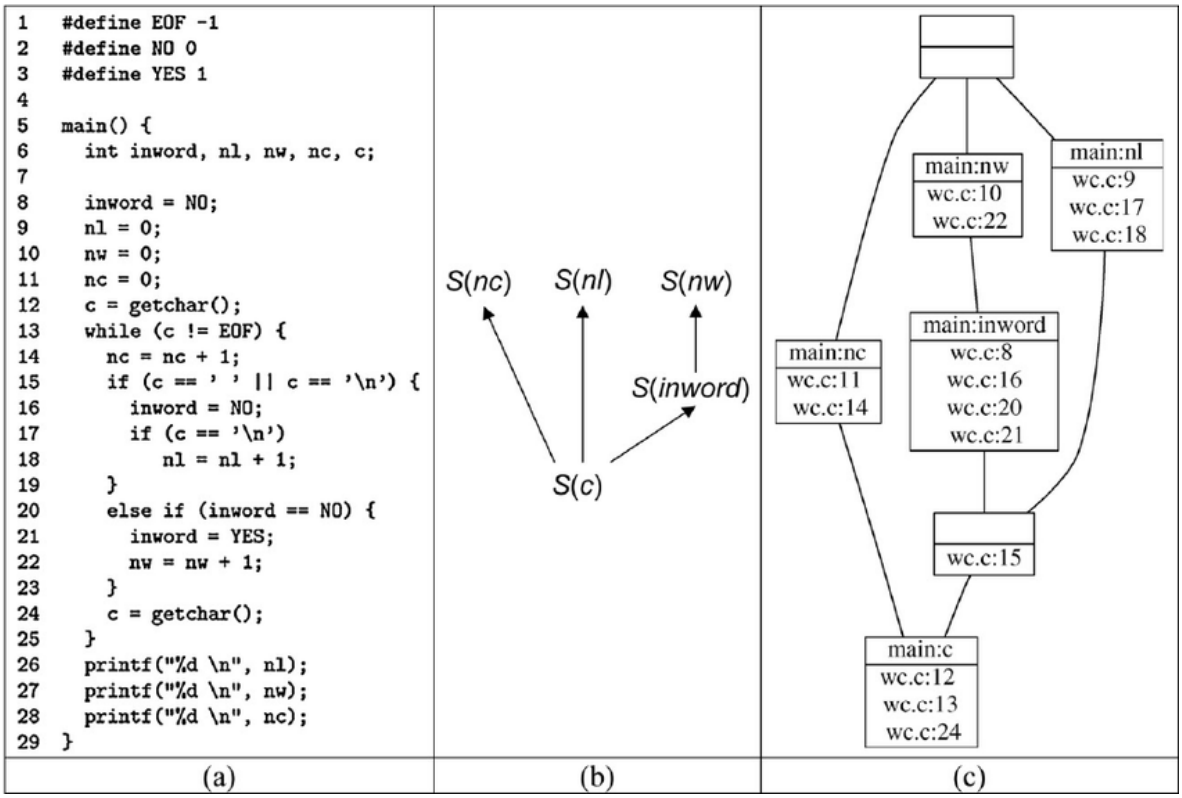


Figura 3.: (a) Programa de contagem de palavras. (b) Reticulado de *decomposition slices* de acordo com Gallagher e Lyle [GL91]. (c) Reticulado concetual de *decomposition slices* proposto por Tonella [Ton03].

As abordagens referidas permitem a extração da computação de uma variável, originando um novo método. Contudo, todo o método é tomado em consideração, produzindo para cada *slicing criterion* uma, e só uma, *slice*.

Com o intuito de produzir mais do que uma *slice* para um dado *slicing criterion* e evitar utilizar todo o método como região, Maruyama [Mar01] introduziu o conceito de *block-based region*. Este conceito permite definir um conjunto de regiões, dado um *slicing criterion*, definidas por fronteiras até às quais é semanticamente correto aplicar uma extração de comportamento. Assim, no contexto proposto por Maruyama [Mar01], ao contrário das propostas que o antecederam, é possível produzir múltiplas *slices*, dado que é possível construir múltiplas regiões em torno de um *slicing criterion*.

Tsantalis e Chatzigeorgiou [TC09] apresentaram uma abordagem que automaticamente identifica oportunidades de refatorização, relacionadas com a operação *Move Method*, e propõem um algoritmo que calcula a distância entre as entidades de um sistema (i.e., atributos e métodos) e classes, calculando um conjunto de oportunidades de refatorização, baseando-se na métrica *Entity Placement* que quantifica o posicionamento das entidades das classes.

Tsantalis e Chatzigeorgiou [TC11] apresentaram uma abordagem que permite automaticamente identificar oportunidades de *Extract Method* relacionadas com a computação total de uma variável (*complete computation slice*) e com as instruções que afetam o estado de um objeto (*object state slice*). Para tal, basearam-se na proposta inicial de *block-based slicing*, apresentada por Maruyama [Mar01]. Com o intuito de compreender se as oportunidades de refatorização identificadas são válidas (semanticamente corretas), se não alteram o comportamento inicial do programa e se são funcionalmente úteis, os autores apresentaram, também, um conjunto de regras para avaliar se as *slices* devem ser consideradas para extração. Os autores terminam o artigo com a apresentação de um *plug-in* [JDe10], desenvolvido para o IDE Eclipse, que permite identificar oportunidades de refatorização e aplicar técnicas de extração de comportamento na linguagem de programação Java.

Ferrante et al. [FOW87] introduziram o *Program Dependence Graph (PDG)* que tem como objetivo representar as dependências de dados e controlo entre as operações de um procedimento. Os nodos do PDG representam as instruções do procedimento, e cada qual define um conjunto de variáveis modificadas por si e um conjunto de variáveis por si utilizadas. Uma dependência do nodo p para o nodo q, devido a uma variável v, denota que q define o valor de v, q recorre ao valor de v, e não existe um nodo p' entre p e q que define v. Mais tarde, Horwitz et al. [HRB88] introduziram o *System Dependence Graph (SDG)* para representar interações entre os múltiplos PDGs de um programa, possibilitando, como foi acima referido, *interprocedural slicing*.

No contexto dos paradigmas orientados a objetos, a designação *God Classes* refere os módulos de grande dimensão, complexos ou incompreensíveis. Fokaefs et al. [FTSC11] apresentam uma abordagem que pretende identificar automaticamente oportunidades de refatorização, nomeadamente a extração de classes mais coesas a partir de classes mais complexas, apresentando uma extensão ao plugin JDeodorant [JDe10]. O algoritmo agrupa as entidades de uma classe (i.e., atributos e métodos) em *clusters* para compreender quais fazem sentido manter-se um conjunto. Assim, o algoritmo identifica zonas que podem ser extraídas como classes secundárias. Esta abordagem, orientada a objetos, não se encontra no mesmo espectro de estudo que o ambiente OutSystems, contudo, revela-se importante na construção de *clusters*. Uma possível abordagem análoga, em OutSystems, seria estudar um conjunto de abordagens que permitam agrupar variáveis de uma ação e extraí-las, em conjunto, para uma ação mais simples.

3.3 Métricas de Complexidade

A análise de complexidade pode ser dividida em duas abordagens: análise quantitativa e análise estrutural. As métricas quantitativas têm em consideração os aspetos quantitativos e palpáveis do programa, ou seja, estudam, genericamente, a dimensão do mesmo.

A métrica *Lines of Code (LOC)*, tem como objetivo contar o número de linhas do programa e aferir se este deve ser alvo de alterações. Esta é uma métrica frequentemente utilizada no desenvolvimento de projetos, dada a sua simplicidade de compreensão e aplicação. A *Software Improvement Group (SIG)* defende que esta métrica é relevante para manter a qualidade do *software* e sugere um número máximo de 15 linhas de código por função [VRvdL+16]. Segundo os autores, o reduzido tamanho das funções contribui positivamente para uma análise mais eficaz, testes mais curtos e objetivos e uma reutilização de código mais incidente.

Halstead [Hal77] propôs uma métrica que classifica um programa P numa coleção de operadores e operandos, para calcular o número distinto de operadores ($n1$), o número distinto de operandos ($n2$), o número total de operadores ($N1$) e o número total de operandos ($N2$). A partir desta proposta, o autor apresenta um conjunto de métricas de cálculo para validar a qualidade do programa.

Weiser [Wei81] defende que algumas métricas baseadas em *slices* podem ser particularmente úteis no contexto do cálculo de complexidade, nomeadamente, *coverage*, *overlap*, *clustering*, *parallelism* e *tightness*. *Coverage* compara a dimensão das *slices* com a dimensão do programa, propondo que a existência de muitas *slices* pequenas determinam múltiplas funcionalidades. *Overlap* calcula o número de instruções que apenas existem numa determinada *slice*, e um valor alto de *overlap* pode indicar que o código é muito interdependente. *Clustering* calcula o grau de coesão das *slices*, indicando uma forte ou fraca dependência entre módulos. *Parallelism* determina o número de *slices* com poucas instruções em comum, o que pode sugerir a execução de *slices* em paralelo, acelerando significativamente o tempo de execução do programa. *Tightness* calcula o número de instruções presentes em todas as *slices* face à dimensão do programa, que pode indicar a existência de *slices* que são utilizadas em simultâneo.

Ott e Thuss [OT93] formalizaram um conjunto de métricas quantitativas que permitem estimar a coesão de módulos, um estudo apoiado nas métricas propostas por Weiser [Wei81]. Estas são baseadas no cálculo dos perfis das *slices*, um conceito introduzido pelos autores em [OT89], que revela a disposição das *slices* no módulo.

Em 2000, Fitzpatrick [Fitoo] explica como se aplica a métrica *Assignments, Branches, Conditionals (ABC)*, cujo objetivo é calcular o tamanho de um programa face ao número de atribuições, bifurcações e condições.

Além das métricas referidas, é também relevante mencionar métricas quantitativas apoiadas em grafos [HM11].

As métricas estruturais, por sua vez, têm em consideração a estrutura do programa e defendem que a complexidade do mesmo depende do número de caminhos que ele possa tomar e do fluxo de dados que transporta.

McCabe [McC76] explica como a teoria de grafos se aplica aos conceitos de programação. O autor defende que a complexidade não está diretamente relacionada com a dimensão do programa, mas sim com o número de caminhos que o programa possa tomar. Assim, o autor introduz uma medida de complexidade de controlo de fluxo, *Cyclomatic Complexity (CC)*, baseada na teoria de grafos, que pretende representar um programa com recurso a um conjunto de nodos, arcos e atributos quantificáveis.

Os nodos de um grafo de controlo de fluxo, *Control Flow Graph (CFG)* [All70], correspondem a grupos de comandos indivisíveis de um programa. Esses nodos são ligados por um arco, se o segundo nodo for executado imediatamente após o primeiro, sem existência de bifurcações intermédias.

A complexidade ciclomática (CC) de um CFG é definida como o número de caminhos linearmente independentes. O número ciclomático $V(G)$ de um grafo G com n nodos, e arcos e p componentes (subgrafos) ligados é dado pela Fórmula 1.

$$V(G) = e - n + 2p \tag{1}$$

McCabe [McC76] apresenta, ainda, algumas propriedades de um CFG, uma das quais sustenta que a complexidade não depende do tamanho físico do código fonte, mas sim do número de predicados de controlo. Por outro lado, sugere um valor limite superior de complexidade ciclomática (10 unidades), a partir do qual o módulo deve ser reestruturado (operação corresponde à refatorização). Este limite superior não é, contudo, consensual, pois deve ter em consideração o contexto da sua aplicação. Por exemplo, o *Object Management Group (OMG)* estipula um valor limite para a complexidade ciclomática em 20 unidades na sua especificação de Medição Automática de Manutenção de Código Fonte, *Automated Source Code Maintainability Measure (ASCMM)*, [Obj16]. A SIG defende que esta métrica é relevante para manter a qualidade e manutenção do *software* e sugere, ainda, um valor máximo de complexidade ciclomática de 4 unidades [VRvdL+16]. Segundo os autores, o reduzido número de caminhos que uma função possa tomar potencia uma melhor alteração e manutenção do *software*, e conduz à produção de testes mais incisivos. A métrica proposta por McCabe é, até à data, muito utilizada pela indústria, devido à sua simplicidade e eficácia. Está presente em especificações de qualidade de código OMG, e modelos de qualidade [HKV07] estabelecidos por líderes da indústria na área de qualidade de *software*.

Myers [Mye77] verificou que a complexidade ciclomática de McCabe é uma métrica que calcula a complexidade de um programa, apesar de produzir situações de anomalia que envolvem predicados de controlo com múltiplas condições. A anomalia pode ser identificada na Figura 4.

Figura 4.: Anomalia identificada no cálculo da Complexidade Ciclomática

```
A: IF X=0 THEN ...
      ELSE ...
B: IF X=0 AND Y>1 THEN ...
               ELSE ...
C: IF X=0 THEN ...
      IF Y>1 THEN ...
             ELSE ...
      ELSE ...
```

A premissa de Myers [Mye77] é que, na Figura 4, a instrução C apresenta maior nível de complexidade que B e B apresenta maior nível de complexidade que A, ou seja, $A < B < C$. Contudo, segundo a métrica de McCabe [McC76], as instruções têm uma complexidade ciclomática de, respetivamente, $V(A) = 2$, $V(B) = 2$ e $V(C) = 3$, invalidando a premissa acima referida $2 \not< 2 < 3$. Myers [Mye77] sugere que a complexidade de um módulo deve ser medida com base num intervalo, e não num valor discreto (Fórmula 2).

$$V(G) = [CC, CC + h] \tag{2}$$

Na Fórmula 2, h é o número de condições individuais ($n - 1$ para n argumentos), ou seja, $V(A) = [2, 2]$, $V(B) = [2, 3]$ e $V(C) = [3, 3]$.

Madi et al. [MZK13] pretendem reforçar a importância do cálculo da complexidade ciclomática de um módulo, enquanto métrica fundamental no *design* de *software*. Os autores propõem uma métrica (Total Cyclomatic Complexity), derivada da complexidade ciclomática apresentada por McCabe [McC76], que permita calcular o valor de CC de um módulo com base no somatório das suas funções (Fórmula 3).

$$TCC(M) = \sum_{i=1}^{n} CC(f_i) + \sum_{i=1}^{m} CC(fc_i) - (n + m) + 1 \tag{3}$$

Na Fórmula 3, $f_i \in F(M)$ é o conjunto de funções do módulo M e $fc_i \in Fc(M)$ é o conjunto de funções executadas por M.

Os autores sublinham a relevância que o acoplamento dos módulos contribui para um *design* indesejado e formulam uma métrica que permita calcular o valor de CC considerando o fluxo de dados existente entre as funções de um módulo (Coupled Cyclomatic Complexity). Dado que o modo da interação entre módulos pode ter impacto na complexidade do sistema, os autores introduzem uma variável α para representar a sua complexidade

intra-modular. Assim, segundo os autores, a Fórmula 4 permite efetuar o cálculo da complexidade inter-modular e intra-modular.

$$CCC = CC + \sum_{i=1}^{n} \alpha_i (CCC(fc_i) - n) \tag{4}$$

O valor de α da Fórmula 4 está relacionado com o nível de coesão do sistema e definido na Tabela 1.

Tabela 1.: Valor de *coupling* de α

Valor de α	**Tipo de *coupling***	**Descrição**
1	*Data*	Um módulo envia um valor escalar ou um array por parâmetro para outro módulo
2	*Stamp*	Um módulo envia uma estrutura por parâmetro para outro módulo
3	*Control*	Um módulo envia um valor por parâmetro para outro módulo e esse valor é utilizado para controlar a lógica do segundo
4	*Global*	Dois módulos utilizam uma variável global
5	*Content*	Um módulo utiliza o código interno de outro módulo

3.4 FERRAMENTAS DE PROGRAM SLICING

No sentido de delinear o modo de desenvolvimento do produto adjacente a esta dissertação, foram estudadas e avaliadas algumas ferramentas de *slicing* disponíveis no mercado. Com este estudo pretendeu-se avaliar o modo como as ferramentas apresentam as oportunidades de refatorização ao utilizador.

O JDeodorant [JDe10] é um *plug-in open-source* desenvolvido para o IDE Eclipse, e dispõe de uma *User Interface (UI)* que permite identificar e resolver alguns problemas de *design* de código Java: *Feature Envy*, *Type Checking*, *Long Method*, *God Class* e *Duplicated Code*. Entre os existentes, destaca-se a identificação do *code smell Long Method* e respetiva resolução, através da aplicação da técnica de refatorização *Extract Method*. A técnica de *slicing* utilizada pelo *plug-in* é o *block-based slicing*, apresentada em [Maro1]. O JDeodorant é o resultado de uma pesquisa realizada pelos departamentos de Engenharia de *Software* da Universidade Concordia, Canadá, e de Informática Aplicada da Universidade da Macedónia, Grécia.

A Figura 5 ilustra um *screenshot* de como o JDeodorant apresenta uma possível refatorização de *Extract Method*. O *plugin* apresenta as *slices* calculadas, e respetiva informação,

ordenadas pelo seu grau de duplicação e ilustra as instruções que podem ser removidas (cor verde) das que serão duplicadas no método original e no método extraído, após extração da variável (cor vermelha).

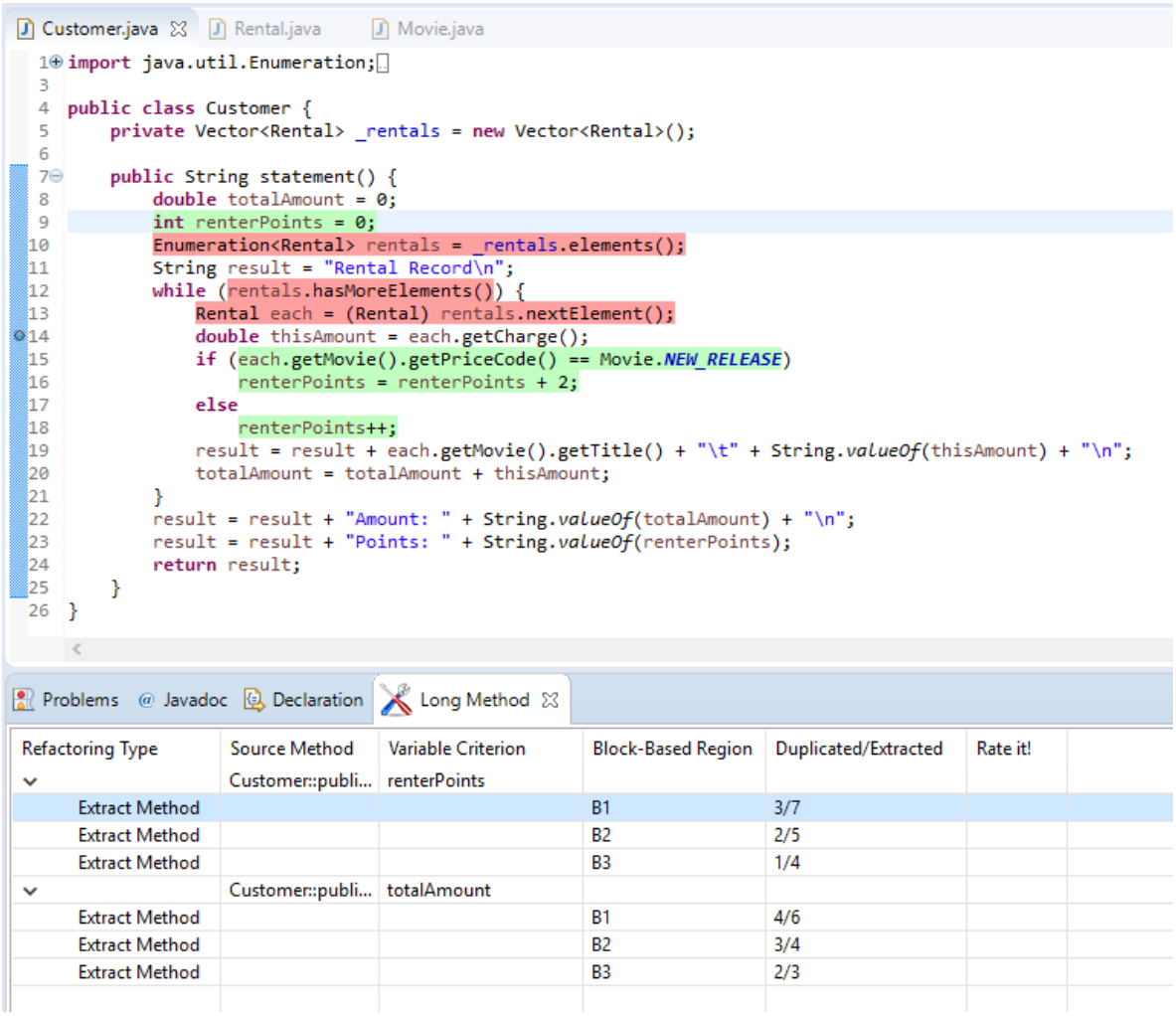


Figura 5.: *Screenshot* de *block-based slicing* com JDeodorant

O WALA [WAL] é uma infraestrutura que permite analisar estaticamente as dependências de controlo e de dados de sistemas em código Java e JavaScript. É executável a partir do IDE Eclipse e da linha de comandos. O WALA foi inicialmente desenvolvido como parte do projeto de pesquisa DOMO no centro IBM T.J. Watson Research Center, e entregue à comunidade de *open-source* de *software* em 2006. Desde então, surgiram diversas ferramentas apoiadas no WALA[2].

O Frama-C [Fra] é uma ferramenta *open-source* de análise estática de código C, que permite efetuar *program slicing*, explorar o fluxo de dados do programa, entre outros (Frama-C

2 https://github.com/wala/WALA/wiki/WALA-Based-Tools — visitado a 28 de julho de 2017

Plugins). À semelhança do JDeodorant, este dispõe de uma UI que destaca as instruções que podem ser afetadas por um determinado *slicing criterion* (Figura 6). Contudo, esta ferramenta necessita que o programador indique o *slicing criterion* a partir do qual são construídas as *slices*.

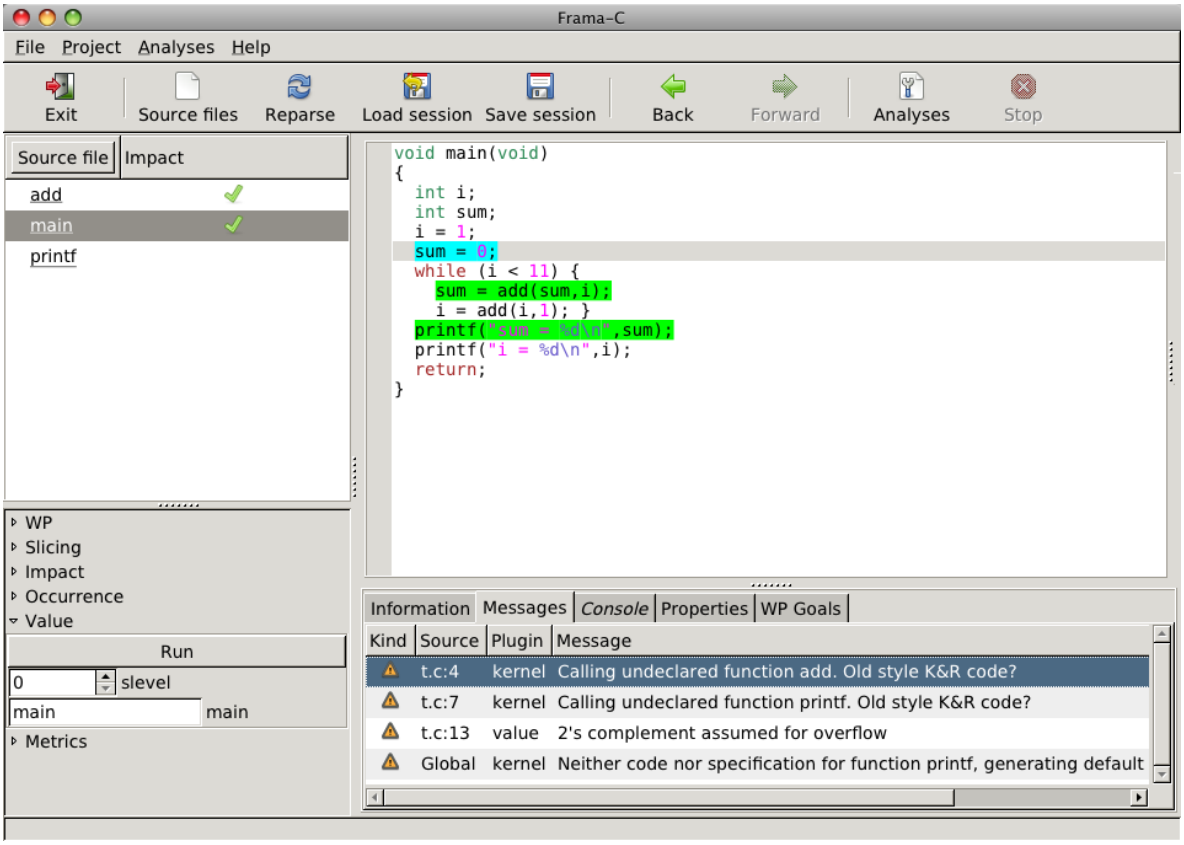


Figura 6.: *Screenshot* de *forward slicing* com Frama-C

O CodeSurfer [Cod] é um *software* comercial que suporta *backward slicing* e *forward slicing* em código C e C++, a partir da análise de dependências de controlo e de dados do sistema. Esta ferramenta divergiu de uma implementação inicial conhecida por The Wisconsin Program-Slicing Tool [Wis].

4

DETALHES DA SOLUÇÃO

O presente capítulo descreve, detalhadamente, a solução proposta, tal como foi inicialmente apresentada na secção 1.4. Este capítulo formaliza os algoritmos, em pseudo-código, que compõem a implementação da proposta de solução.

A solução desenvolvida pretende, em primeira instância, avaliar a complexidade de módulos, por forma a averiguar a necessidade da sua refatorização. Esta análise permite identificar ações propícias à refatorização, ou seja, ações cuja complexidade é relevante e pode ser reduzida. Em segunda instância, a solução pretende reduzir o grau de complexidade de módulos, através da aplicação da técnica de refatorização *Extract Action*[1].

A complexidade de módulos é calculada com base no número de nodos (LOC) e no valor da complexidade ciclomática (CC, Fórmula 1). Segundo a SIG, estas são duas métricas relevantes para analisar a qualidade e o grau de manutenção do *software* [VRvdL+16]. Ambas as métricas são utilizadas para avaliar se uma ação deve ser refatorizada, e ajudam a delinear o conjunto de ações que produzem melhores oportunidades de refatorização.

No que diz respeito à abordagem de *slicing* para decompor programas, foi implementada a técnica *block-based slicing*, proposta por Maruyama [Maro1]. Esta potencia um maior número de refatorizações, uma vez que promove a extração de variáveis a partir das várias regiões identificadas no código. Neste contexto, a técnica é, então, utilizada para extrair parte da lógica de negócio de uma ação para a tornar menos complexa.

Para demonstrar o algoritmo implementado, é utilizada uma adaptação de Tsantalis e Chatzigeorgiou [TC11] de um exemplo de refatorização apresentado por Fowler et al. [FBB+00] (Figura 7). A adaptação do exemplo de refatorização encontra-se na Figura 35, em Java. Em causa, trata-se de um programa que calcula os pontos de um cliente e as suas dívidas numa loja de aluguer de vídeos. O código da Figura 7 é sujeito a *block-based slicing* para identificar partes do código que podem ser extraídas para uma ação secundária.

1 Técnica análoga a *Extract Method*

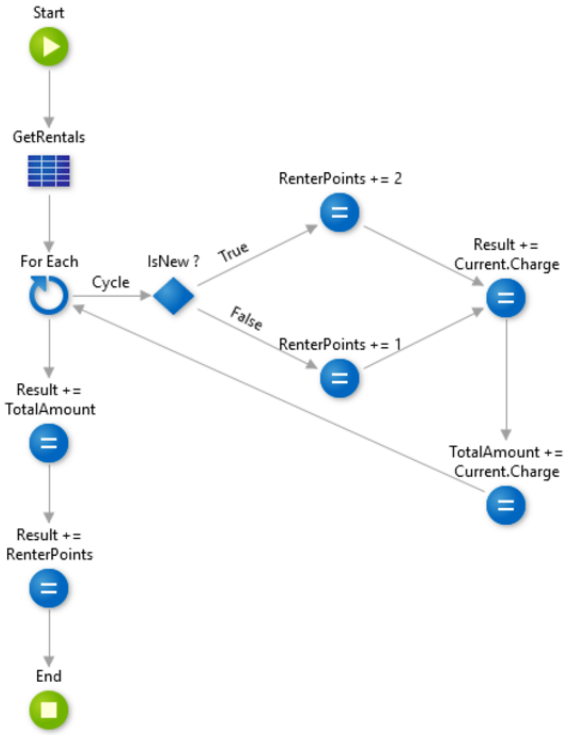


Figura 7.: Ação *Statement* em OutSystems

4.1 Análise de Complexidade

Como foi referido no início do capítulo, a análise de complexidade é relevante para o algoritmo, pois permite filtrar as ações que apresentam maior grau de complexidade daquelas que são mais simples de analisar.

Dada a elevada amplitude da complexidade dos módulos estudados, tornou-se imperativa a seleção das ações que apresentam maior grau de complexidade. Assim, apenas são avaliadas as ações que mostrarem ter uma complexidade relevante, pois são aquelas que potencialmente melhor a reduzem, após a extração de código.

A análise da complexidade dos módulos recorre a uma métrica quantitativa e a outra estrutural (ver secção 3.3). Recorre, por um lado, ao cálculo do número de nodos (LOC) das ações. A SIG considera esta como uma métrica relevante para manter a qualidade e manutenção dos sistemas de *Tecnologias da Informação (TI)* [VRvdL+16]. Por outro lado, esta análise recorre, também, ao cálculo da métrica de complexidade estrutural proposta por McCabe [McC76] ($CC = e - n + 2p$, n nodos, e arcos e p subgrafos). O valor da complexidade ciclomática de um módulo pode, também, ser calculada com recurso ao número dos seus predicados de controlo [MZK13], i.e., bifurcações. O Algoritmo 1 e o Algoritmo 2 representam o cálculo da complexidade de uma ação, em pseudo-código.

Algoritmo 1 Cálculo do Número de Nodos (LOC)

```
1:  procedure LOC(root)
2:      nodes ← 0
3:      Add root to queue
4:      Mark root as visited
5:      while queue is not empty do
6:          current ← Dequeue(queue)
7:          nodes ← nodes + 1
8:          for each edge ∈ Edges(current) do
9:              if Target(edge) is not visited then
10:                 Add Target(edge) to queue
11:                 Mark Target(edge) as visited
        return nodes
```

Algoritmo 2 Cálculo da Complexidade Ciclomática (CC)

```
1:  procedure CC(root)
2:      nodes ← 0
3:      edges ← 0
4:      Add root to queue
5:      Mark root as visited
6:      while queue is not empty do
7:          current ← Dequeue(queue)
8:          nodes ← nodes + 1
9:          for each edge ∈ Edges(current) do
10:             edges ← edges + 1
11:             if Target(edge) is not visited then
12:                 Add Target(edge) to queue
13:                 Mark Target(edge) as visited
        return nodes − edges + 2
```

4.2 BLOCK-BASED SLICING

Como foi referido no início do capítulo, a abordagem escolhida para decompor programas é a técnica *block-based slicing* proposta por Maruyama [Maro1]. A implementação apresenta as caraterísticas (i) *backward slicing*, (ii) *static slicing*, (iii) *intraprocedural slicing*, (iv) *syntax preserving slicing* e (v) *block-based slicing*. Com recurso a esta técnica, torna-se exequível a aplicação de Extract Action, a técnica de refatorização implementada no contexto desta dissertação.

As técnicas de *slicing* tradicionais [Wei81] utilizam a função na sua completude como região, enquanto que técnicas de *slicing* baseadas em blocos [Maro1] definem regiões na ação, a partir das quais é igualmente praticável a extração de comportamento. Esta abordagem

explora o programa por regiões (*block-based regions*), com o objetivo de balizar a expansão das *slices*. Assim, é possível produzir, pelo menos, uma *slice* que compute cada variável v para cada região R do programa.

4.2.1 *Determinação de Basic Blocks*

Um *basic block* é uma sequência de nodos consecutivos no CFG sem bifurcações [Maro1]. O nodo líder de um *basic block B* é 1) o primeiro nodo, 2) um nodo *join* (i.e., nodos com mais do que uma aresta de entrada) ou 3) nodos que sejam diretamente consecutivos a nodos de bifurcação (i.e., nodos consecutivos aos nodos *If*, *For Each* e *Switch*). Para cada nodo líder, um *basic block B* consiste no nodo líder e todos os nodos que o seguem até ao próximo nodo líder. Os nodos *Start* e *End*, apresentados no ponto 2.2.1.1, não pertencem a nenhum *basic block*.

A Figura 8 ilustra os *basic blocks* que compõem a ação apresentada na Figura 7.

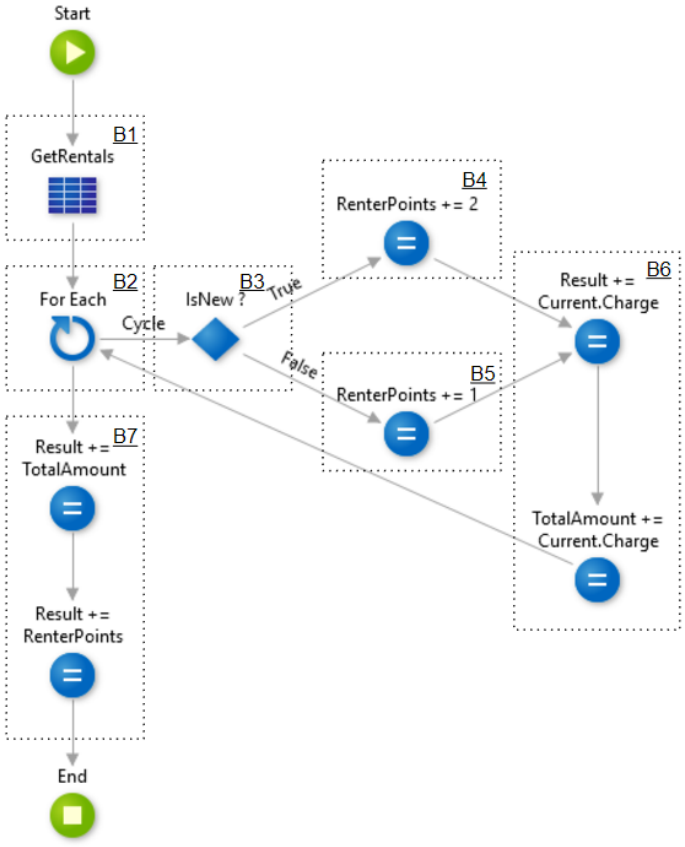


Figura 8.: *Basic blocks* que constituem a ação da Figura 7

O Algoritmo 3 representa a determinação dos *basic blocks* de um CFG, em pseudo-código.

Algoritmo 3 Determinação de *Basic Blocks*

```
1:  procedure BasicBlocks(root)
2:      Add root to queue
3:      while queue is not empty do
4:          current ← Dequeue(queue)
5:          mark current as visited
6:          if current is root ∥ current is join ∥ ∃ p ∈ Pred(current): p is branch node then
7:              BasicBlock(current) ← new BasicBlock
8:          for each s ∈ Succ(current) do
9:              if s is not visited then
10:                 BasicBlock(s) ← BasicBlock(current)
11:                 Add s to queue
```

4.2.2 *Determinação de Reachable Blocks*

Um *basic block* B_i alcança B_j se existir pelo menos um caminho no CFG entre B_i e B_j, sem percorrer arestas cíclicas (i.e., *loopback edges*), e, então, B_j pertence ao alcance de B_i, $Reach(B_i)$ [Mar01]. Verifica-se pela definição descrita e pela análise da Figura 8 que:

$$Reach(B_1) = \{B_1, B_2, B_3, B_4, B_5, B_6, B_7\}$$

$$Reach(B_2) = \{B_2, B_3, B_4, B_5, B_6, B_7\}$$

$$Reach(B_3) = \{B_3, B_4, B_5, B_6\}$$

$$Reach(B_4) = \{B_4, B_6\}$$

$$Reach(B_5) = \{B_5, B_6\}$$

$$Reach(B_6) = \{B_6\}$$

$$Reach(B_7) = \{B_7\}$$

O Algoritmo 4 representa a determinação dos *reachable blocks* de um *basic block* B_i, em pseudo-código.

Algoritmo 4 Determinação de *Reachable Blocks*

```
1:  procedure Reach(blocks)
2:      for each block ∈ blocks do
3:          for each ancestor ∈ Ancestors(Leader(block)) do
4:              Add block to Reach(BasicBlock(ancestor))
```

4.2.3 *Determinação de Dominated Blocks*

Segundo Maruyama [Maro1], os *dominated blocks* do *basic block* B_i, $Dom(B_i)$, são os *basic blocks* dominados pelo nodo j, onde j é o nodo que domina diretamente os nodos de B_i. Por definição, o nodo de entrada na função (*Start*) domina todos os nodos. O cálculo de $Dom(B_i)$ recorre à análise das dependências de controlo existentes no PDG (Figura 9).

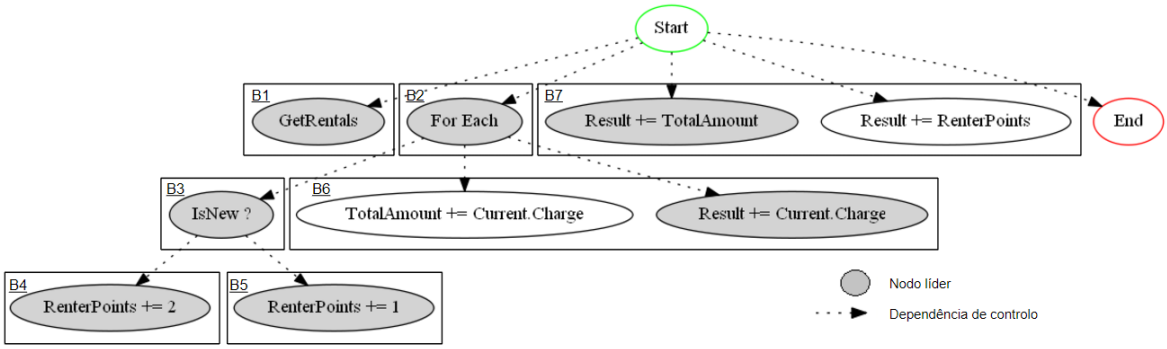


Figura 9.: Dependências de controlo existentes no PDG da ação *Statement*

Verifica-se pela definição descrita e pela análise da Figura 9 que:

$$Dom(B_1) = Dom(B_2) = Dom(B_7) = \{B_1, B_2, B_3, B_4, B_5, B_6, B_7\}$$
$$Dom(B_3) = Dom(B_6) = \{B_3, B_4, B_5, B_6\}$$
$$Dom(B_4) = Dom(B_5) = \{B_4, B_5\}$$

O Algoritmo 5 representa a computação de blocos dominantes, que recorre à definição das dependências de controlo do PDG.

Algoritmo 5 Determinação de *Dominated Blocks*

```
1: procedure DominatedBlocks(blocks)
2:     for each block ∈ blocks do
3:         incomingControl ← First(IncomingControlDependencies(Leader(block)))
4:         sourceControl ← Source(incomingControl)
5:         Dom(block) ← BasicBlocks(AllControlDependentNodes(sourceControl))
```

A função "AllControlDependentNodes", utilizada pelo Algoritmo 5, pretende determinar, recursiva e cumulativamente, os nodos j que dependem da execução de um nodo i (Algoritmo 6).

Algoritmo 6 Determinação de todos os nodos dependentes de controlo

```
1: procedure AllControlDependentNodes(dominant)
2:     result ← {}
3:     for each controlDependency ∈ OutgoingControlDependencies(dominant) do
4:         Add Target(controlDependency) to result
5:         result ← result ∪ AllControlDependentNodes(Target(controlDependency))
       return result
```

Tal como ilustrado na Figura 9, as dependências de controlo no PDG traduzem dependências entre nodos de bifurcação (i.e., nodos *If*, *For Each* e *Switch*), ou o nodo de início (i.e., nodo *Start*), e os nodos aninhados j, i.e., nodos cuja execução depende de i.

O cálculo das dependências de controlo $p \rightarrow_c q$ do PDG recorre à definição de *dominators* introduzida por Reese Prosser [Pro59]. Segundo o autor, uma instrução i domina j se todos os caminhos da função, desde o início até ao fim, que contêm a instrução j também contêm a instrução i. Segundo o autor, uma instrução i domina j se todos os caminhos da função, desde o início até ao fim, que contêm a instrução j também contêm a instrução i. O Algoritmo 7 representa a determinação dos nodos dominantes de um nodo, em pseudo-código.

Algoritmo 7 Determinação de *Dominators*

```
1: procedure Dominators(nodes)
2:     Dom(nodes[0]) ← nodes[0]
3:     for each n ∈ nodes − {n} do
4:         Dom(n) ← nodes
5:     while changes in any Dom(n) do
6:         for each n ∈ nodes − {n} do
7:             Dom(n) ← ( ⋂_{p∈Preds(n)} Dom(p) ) ∪ {n}
```

O Algoritmo 7 foi apresentado por Aho et al. [ALSU06] para determinar os nodos dominantes. A Figura 10 ilustra a *Dominator Tree* [LT79], gerada a partir da definição acima referida, permite aferir quais são os *immediate dominators* de i, $idom(i)$, i.e., os nodos que dominam diretamente a execução de outros.

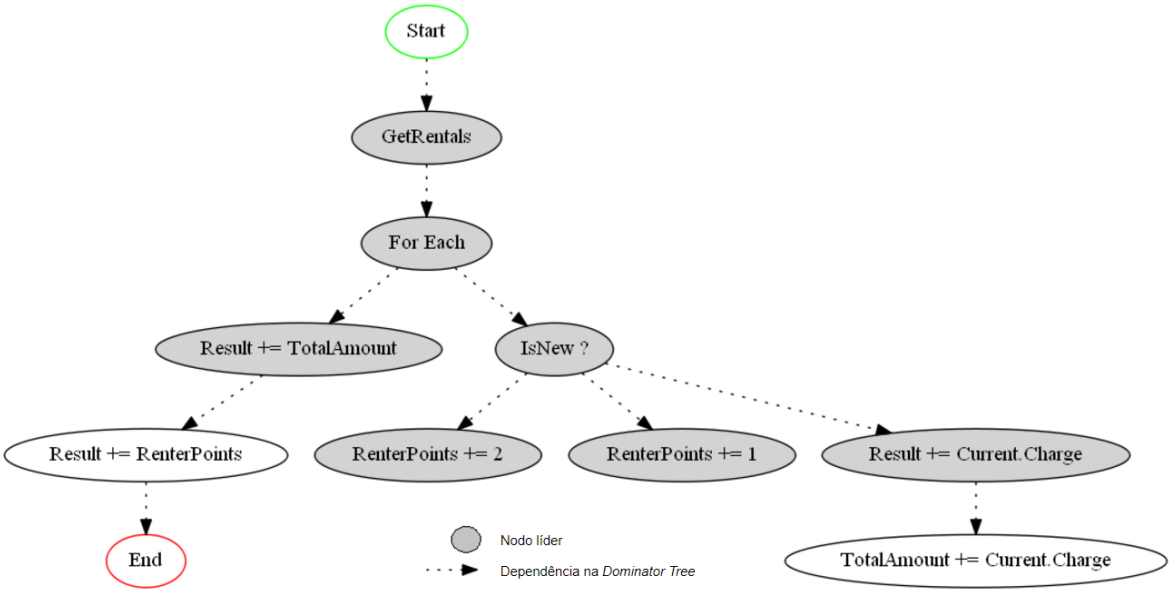


Figura 10.: *Dominator Tree* da ação *Statement*

Contudo, a definição de $idom(i)$ não é suficiente para aferir as dependências de controlo do PDG, pois, por um lado, segundo a definição, apenas existem dependências de controlo entre nodos de tipo *branch* ou *Start* para outros. Por outro lado, nem todas as dependências na *Dominator Tree* se traduzem em dependências de controlo. Por exemplo, apesar do nodo *"Is New ?"* ser o *immediate dominator* do nodo *"Result += Current.Charge"*, não existe uma dependência de controlo entre eles, pois todos os caminhos possíveis desde o nodo *"Is New ?"* contêm o nodo *"Result += Current.Charge"*. Assim, foi necessário alterar os nodos dominantes, como apresentado pelo Algoritmo 8.

Algoritmo 8 Colapso da *Dominator Tree*

```
1:  procedure CollapseDominatorTree(root)
2:      Add root to queue
3:      while queue is not empty do
4:          current ← Dequeue(queue)
5:          while parentChanged do
6:              for each child ∈ Children(current) do
7:                  Add child to queue
8:                  if current ≠ root & IsTrueDependency(current, child) = False then
9:                      ChangeParent(child, Parent(current))
10:                     parentChanged ← True
```

O Algoritmo 9 representa a função *"isTrueDependency"* utilizada pelo Algoritmo 8.

Algoritmo 9 Determinação de dependências de controlo

```
1: procedure IsTrueDependency(source, target)
2:     if source = target then return False
3:     if source is End then return True
4:     Mark source as visited
5:     result ← False
6:     for each flow in OutgoingFlows(source) do
7:         if Target(flow) is not visited then
8:             result ← result || IsTrueDependency(Target(flow), target)
       return result
```

A Figura 11 representa a *Dominator Tree* que, após a aplicação do Algoritmo 8, ilustra as dependências de controlo do PDG.

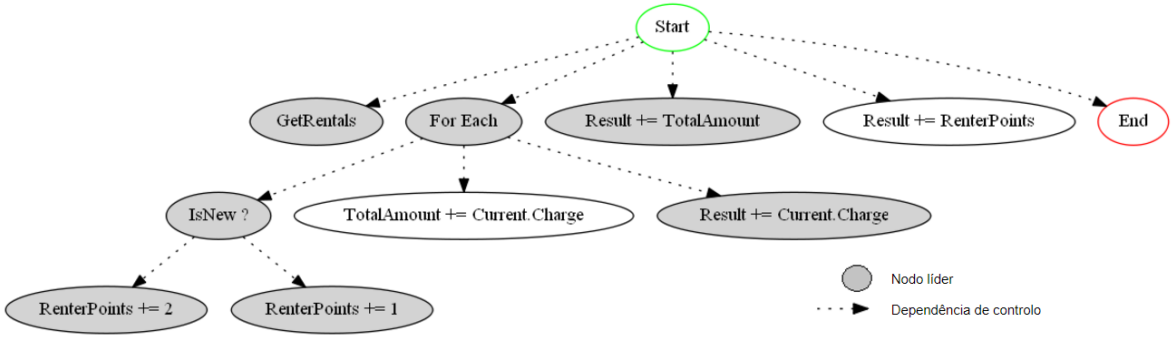


Figura 11.: Estado final da *Dominator Tree* da ação *Statement*

4.2.4 *Determinação de Boundary Blocks*

Maruyama [Maro1] utiliza a definição de *reachable* e *dominated blocks* para introduzir o conceito de *boundary blocks*. Para cada *basic block* B_i presente na ação, calcula-se $Reach(B_i) \cap Dom(B_i)$ e B_i é *boundary block* de B_j, $B_j \in Reach(B_i) \cap Dom(B_i)$.

De seguida, apresentam-se os *boundary blocks* para todos os nodos de cada *basic block*. Note-se que, por definição, os *boundary blocks* definem-se por $Blocks(n)$, $n \in N(m)$. Contudo, dado que todos os nodos do mesmo *basic block* B_i apresentam o mesmo conjunto de *boundary blocks*, $Blocks(n)$ simplifica-se em $Blocks(B_i)$.

$$Blocks(B_1) = \{B_1\}$$

$$Blocks(B_2) = \{B_1, B_2\}$$

$$Blocks(B_3) = \{B_1, B_2, B_3\}$$

$$Blocks(B_4) = \{B_1, B_2, B_3, B_4\}$$

$$Blocks(B_5) = \{B_1, B_2, B_3, B_5\}$$
$$Blocks(B_6) = \{B_1, B_2, B_3, B_6\}$$
$$Blocks(B_7) = \{B_1, B_2, B_7\}$$

O Algoritmo 10 representa a determinação dos *boundary blocks*, em pseudo-código.

Algoritmo 10 Determinação de *Boundary Blocks*

```
1: procedure BLOCKS(blocks)
2:     for each block ∈ blocks do
3:         for each intersection ∈ Reach(block) ∩ Dom(block) do
4:             Add block to Blocks(intersection)
```

4.2.5 *Determinação de Block-based Regions*

Maruyama [Maro1] define uma *block-based region*, $R(B_n)$, como sendo os nodos alcançáveis a partir de um *basic block* identificado como sendo um *boundary block* B_n, i.e., nodos que pertençam a $Reach(B_n)$ (Fórmula 5). A partir das múltiplas regiões $R(B_n)$ são exequíveis n extrações de uma variável v, possibilitando a criação de *block-based slices*.

$$R(B_n) = \{p \in N(m) \mid p \in C \wedge C \in Reach(B_n)\} \tag{5}$$

4.2.6 *Determinação de Block-based Slices*

Uma *block-based slice* é uma *slice* estática gerada a partir de uma *block-based region*, sem necessitar de utilizar toda a função como região [Maro1]. A *block-based slice* é o conjunto dos nodos da região R que definem a variável v e os nodos da região R que contêm dependências de controlo ou de dados (*Read After Write (RAW)* que podem ser transportadas por ciclos) para os mesmos, i.e., nodos que podem afetar o comportamento de v. As arestas que compõe a *slice* são todas as dependências de controlo ($\rightarrow_c$), dependências de dados independentes de ciclo ($\rightarrow_{li}$) e algumas dependências de dados transportadas por ciclos ($\rightarrow_{lc(l)}$) em R (Fórmula 6).

$$\begin{aligned} E_B(R) = &\{p \rightarrow_c q \in E(G) \mid p, q \in R\} \\ &\cup \{p \rightarrow_{li} q \in E(G) \mid p, q \in R\} \\ &\cup \{p \rightarrow_{lc(l)} q \in E(G) \mid l, p, q \in R\} \end{aligned} \tag{6}$$

Na Fórmula 6, $E(G)$ denota todas as arestas do PDG G. A Fórmula 7 representa uma *block-based slice* S_B produzida pelo *slicing criterion* $C = (n, v)$, nodo n, variável v e *boundary block* $B_n \in Blocks(n)$.

$$S_B(n, v, B_n) = \{m \in N(G) \mid m \rightarrow^* n \in E_B(R(B_n))\}, v \in Def(n)$$

$$\begin{aligned} S_B(n, v, B_n) &= \bigcup_{m \in M(n,v,B_n)} S_B(m, v, B_n), \\ M(n, v, B_n) &= \{m \in N(G) \mid m \rightarrow^v n \in E_B(R(B_n))\}, v \in Use(n) \end{aligned} \tag{7}$$

4.2.6.1 *Determinação dos Nodos da Slice*

Como foi referido, os nodos da *slice* são todos os nodos que podem afetar um *slicing criterion*. Ou seja, pertencem à *slice* todos os nodos que podem afetar o valor da variável v: nodos que definem v na região R, nodos, em R, que utilizam v e nodos, em R, com dependências de dados ou de controlo para os referidos nodos. Assim, recorre-se a uma travessia *bottom-up* de todos os nodos que possam afetar o critério $C = (n, v)$, i.e., nodos da região R com dependências de controlo ou de dados para o nodo n, devido à variável v.

O Algoritmo 11 representa a determinação dos nodos da *block-based slice*, em pseudo-código. Note-se que os nodos utilizados no Algoritmo 11 pertencem a uma *block-based region* e que as arestas são dependências de controlo ou de dados existentes entre os mesmos nodos.

Algoritmo 11 Determinação dos nodos da *Block-based slice*

```
1: procedure SliceNodes(node, variable, nodes, edges)
2:     result ← {}
3:     if variable is null ∥ variable ∈ Def(node) then
4:         result ← result ∪ BackwardsTraversal(node, edges)
5:     else if variable ∈ Use(node) then
6:         for each defNode ∈ IncomingDataDependencies(node) do
7:             result ← result ∪ BackwardsTraversal(defNode, edges)
8:         result ← result ∪ BackwardsTraversal(node, edges)
       return result
```

O Algoritmo 12 representa a travessia utilizada para efetuar *slicing*. Por questões de estética na apresentação do algoritmo, as dependências de entrada de controlo e de dados foram abreviadas para "IncControlDeps" e "IncDataDeps", respetivamente.

Algoritmo 12 Travessia *bottom-up* do grafo

```
1:  procedure BackwardsTraversal(initial, edges)
2:      result ← {initial}
3:      Add initial to queue
4:      while queue is not empty do
5:          current ← Dequeue(queue)
6:          for each dependency ∈ IncControlDeps(current) ∪ IncDataDeps(current) do
7:              if dependency ∈ edges then
8:                  Add Source(dependency) to result
9:                  if Source(dependency) is not visited then
10:                     Add Source(dependency) to queue
11:                     Mark Source(dependency) as visited
        return result
```

De seguida, apresenta-se a especificação das funções utilizadas pelo Algoritmo 11 e pelo Algoritmo 12:

Use(**p**) é o conjunto de variáveis utilizadas pelo nodo p.

Def(**p**) é o conjunto de variáveis definidas pelo nodo p.

IncomingDataDependencies(**p**) é o conjunto de dependências de dados RAW de um nodo k para o nodo p.

IncomingControlDependencies(**p**) é o conjunto de dependências de controlo de um nodo k para o nodo p.

4.2.6.2 *Determinação dos Nodos Indispensáveis*

Nodos indispensáveis são nodos que pertencem à *slice*, mas não devem ser removidos da ação original após a extração da *slice*, garantindo a preservação do comportamento da ação. Ou seja, estes nodos são necessários na ação original e na ação extraída, pelo que devem ser duplicados [TC11]. Maruyama [Mar01] introduz o conceito de nodos indispensáveis no sentido de identificar nodos essenciais da ação original, numa determinada região R e *slicing criterion* $C = (n, v)$. À semelhança de [TC11], propõe-se a identificação dos nodos indispensáveis de todos os critérios $C = (n, v)$ em R. A Fórmula 8 representa os nodos indispensáveis, que são os nodos com dependências de controlo (I_{CD}) ou de dados (I_{DD}) pertencentes à *slice* e com dependências para nodos não pertencentes à *slice*.

$$
\begin{aligned}
&I_B(S_B, U_B, v, B) = I_{CD} \cup I_{DD},\\
&I_{CD} = \{q \in N(m) \mid (q \in S_B(p, u, B) \vee q = p) \wedge p \in N_{CD}(S_B, U_B) \wedge u \in Use(p)\},\\
&I_{DD} = \{q \in N(m) \mid q \in S_B(p, u, B) \wedge p \in N_{DD}(S_B, U_B, v) \wedge u \in Def(p)\}
\end{aligned}
\tag{8}
$$

A Fórmula 9 representa o cálculo dos nodos da *slice* com pelos menos uma dependência de controlo para nodos não pertencentes à *slice*, N_{CD}.

$$N_{CD}(S_B, U_B) = \{p \in N(m) \mid p \rightarrow_c q \in E(m) \wedge p \in S_B \wedge q \in U_B\} \tag{9}$$

A Fórmula 10 representa o cálculo dos nodos da *slice* com pelos menos uma dependência de dados para nodos não pertencentes à *slice*, N_{DD}.

$$N_{DD}(S_B, U_B, v) = \{p \in N(m) \mid p \rightarrow_d^u q \in E(m) \wedge u \neq v \wedge p \in S_B \wedge q \in U_B\} \tag{10}$$

As fórmulas acima referidas utilizam as seguintes definições:

N(m) é o conjunto de nodos do PDG.

E(m) é o conjunto de arestas (dependências) do PDG.

S_B é o conjunto dos nodos da *slice*.

U_B é o conjunto de nodos não pertencente à *slice* $U_B = N(m) \backslash S_B$.

Use(p) é o conjunto de variáveis utilizadas pelo nodo p.

Def(p) é o conjunto de variáveis definidas pelo nodo p.

N_{CD} é o conjunto de nodos em S_B que têm, pelo menos, uma dependência de controlo para um nodo de U_B.

N_{DD} é o conjunto de nodos em S_B que têm, pelo menos, uma dependência de dados (RAW) para um nodo de U_B.

O Algoritmo 13 representa o cálculo dos nodos indispensáveis devido a dependências de controlo, em pseudo-código. Por questões de estética na apresentação do algoritmo, as dependências de saída de controlo foram abreviadas para "OutControlDeps".

Algoritmo 13 Determinação de Nodos Indispensáveis devido a dependências de controlo

```
1: procedure IndispensableControlNodes(N(m), S_B)
2:     result ← {}
3:     N_CD ← DetermineNCD(N(m), S_B)
4:     for each node ∈ N_CD do
5:         for each v ∈ Use(node) do
6:             result ← result ∪ SliceNodes(node, v, S_B, Edges(S_B))
7:         if Use(node) = ∅ then
8:             result ← result ∪ SliceNodes(node, null, S_B, Edges(S_B))
       return result
```

O Algoritmo 14 representa o cálculo de N_{CD}, indicado na Fórmula 9 e utilizado pelo Algoritmo 13.

Algoritmo 14 Nodos de U_B com dependências de controlo para nodos de S_B

```
1: procedure DetermineNCD(N(m), S_B)
2:     N_CD ← {}
3:     U_B ← N(m)\S_B
4:     for each node ∈ S_B do
5:         for each dependency ∈ OutControlDeps(node) do
6:             if Target(dependency) ∈ U_B then
7:                 N_CD ← N_CD ∪ {node}
       return N_CD
```

De estrutura semelhante, o Algoritmo 15 representa o algoritmo de determinação dos nodos indispensáveis devido a dependências de dados, em pseudo-código. Por questões de estética na apresentação do algoritmo, as dependências de saída de dados foram abreviadas para "OutDataDeps".

Algoritmo 15 Determinação de Nodos Indispensáveis devido a dependências de dados

```
1: procedure IndispensableDataNodes(N(m), S_B)
2:     result ← {}
3:     N_DD ← DetermineNDD(N(m), S_B)
4:     for each node ∈ N_DD do
5:         for each v ∈ Def(node) do
6:             result ← result ∪ SliceNodes(node, v, S_B, Edges(S_B))
       return result
```

O Algoritmo 16 representa o cálculo de N_{DD}, indicado na Fórmula 10 e utilizado pelo Algoritmo 15.

Algoritmo 16 Nodos de U_B com dependências de dados para nodos de S_B

```
1: procedure DetermineNDD(N(m), S_B)
2:     N_DD ← {}
3:     U_B ← N(m)\S_B
4:     for each node ∈ S_B do
5:         for each dependency ∈ OutDataDeps(node) do
6:             if Target(dependency) ∈ U_B & Variable(dependency) <> Variable(S_B) then
7:                 N_DD ← N_DD ∪ {node}
       return N_DD
```

4.2.6.3 *Determinação dos Nodos Removíveis e Parâmetros de Input da Ação Gerada*

Segundo Maruyama [Maro1], os nodos removíveis da ação original são os nodos que não pertencem a $U_B \cup I_B$, i.e., nodos da *slice* que não são indispensáveis ($S_B \backslash I_B$).

Após a implementação dos passos acima referidos, que concretizam a construção de uma *slice*, torna-se fundamental determinar os parâmetros de *input* que a ação gerada

necessita para preservar a sua sintaxe. Os parâmetros de *input* correspondem às variáveis das dependências de dados existentes entre os nodos de U_B e os nodos da *slice* (S_B): $P(S_B, U_B) = \{u \in V(m) \mid p \rightarrow_d^u q \in E(m) \wedge p \in U_B \wedge q \in S_B\}$, onde $V(m)$ é o conjunto de variáveis locais e variáveis de *input* da ação m.

O Algoritmo 17 representa a determinação dos parâmetros de *input* da ação gerada, em pseudo-código.

Algoritmo 17 Determinação de parâmetros de *Input* da ação gerada

```
1: procedure InputParameters(N(m), S_B)
2:     result ← {}
3:     U_B ← N(m)\S_B
4:     for each node ∈ S_B do
5:         for each dependency ∈ IncDataDeps(node) do
6:             if Source(dependency) ∈ U_B then
7:                 result ← result ∪ {node}
       return result
```

4.3 Seleção das Melhores Oportunidades de Refatorização

No sentido de aumentar o grau de manutenção de um sistema, é necessário que as sugestões de refatorização melhorem a sua legibilidade e qualidade. Dado que o algoritmo deve assistir o programador na seleção das melhores oportunidades identificadas, então, é necessário respeitar os seguintes tópicos:

- As sugestões apresentadas não devem afetar o comportamento inicial do programa.
- As sugestões apresentadas devem apresentar um corpo de refatorização suficiente.
- As sugestões apresentadas devem ser ordenadas por ordem decrescente no ganho que produzam.

Como foi referido no início do capítulo, apenas são avaliadas as ações que mostrarem ter complexidade relevante, i.e., as ações cujo número de nodos (LOC) ou o número de caminhos possíveis (CC) seja superior aos valores *standard* definidos pela SIG [VRvdL+16]: 15 nodos e 4 unidades, respetivamente. Estes são os valores de *threshold* utilizados pelo algoritmo para aferir a utilidade de refatorização das ações, ou seja, se uma ação apresentar no mínimo 16 nodos ou 5 unidades de CC, então deve ser analisada para refatorização.

4.3.1 *Regras de Preservação do Comportamento das Slices*

No intuito de sugerir as *slices* que preservam o comportamento funcional para refatorização, é aplicado um conjunto de pré-condições, ou regras. Estas regras, apresentadas

por Tsantalis e Chatzigeorgiou [TC11], garantem que não são sugeridas oportunidades de refatorização que possam alterar o comportamento do programa devido a dependências de dados existentes entre a *slice* e o resto do programa ou que possam ser sugeridas refatorizações pouco relevantes.

4.3.1.1 *Preservação de Anti-Dependências*

Uma anti-dependência [KHoo], ou dependência *Write After Read (WAR)*, existe entre dois nodos p e q devido à variável v ($p \rightarrow_a^v q$), se p utiliza v ($v \in Use(p)$) e q define o valor de v ($v \in Def(q)$) e existir, pelo menos, um caminho no CFG que inicie em p e termine em q, sem que exista um nodo k intermédio que defina o valor de v (independentemente da existência de nodos k_i que utilizem o valor de v).

Ora, não deve existir uma anti-dependência, devido à variável v, na *block-based region* $R(B)$ entre um nodo p, que pertence à ação original após extração da *slice*, ($p \in U_B \cup I_B$), e um nodo q removível ($q \in S_B \backslash I_B$), sem a presença de uma dependência de dados RAW ($k \rightarrow_d^v p \mid v \in Def(k) \wedge v \in Use(p)$) entre um nodo k da *block-based region* $R(B)$ ($k \in R(B) \wedge k \in U_B \cup I_B$) e p devido à variável u [TC11]. Se a regra não se verificar, então a *slice* é inválida, pois pode alterar o comportamento inicial do programa, pelo que não deve ser apresentada. A Fórmula 11 formaliza a definição desta regra.

$$\begin{aligned} &\{p \rightarrow q \in A_B(R(B)) \mid p \in U_B \cup I_B \wedge q \in S_B \backslash I_B\} \setminus \\ &\{p \rightarrow q \in A_B(R(B)) \mid p \in U_B \cup I_B \wedge q \in S_B \backslash I_B \wedge \exists k \rightarrow_d^u p \mid k \in R(B) \wedge k \in U_B \cup I_B\} \\ &= \emptyset \end{aligned} \tag{11}$$

Na Fórmula 11, $A_B(R(B) = \{p \rightarrow_a^u q \mid p, q \in R(B)\})$ denota as anti-dependências da região $R(B)$, que podem ser transportadas por ciclos.

O Algoritmo 18 determina se a *slice* preserva anti-dependências, em pseudo-código. Por questões de estética na apresentação do algoritmo, as anti-dependências de saída e as dependências de entrada de dados foram abreviadas para "OutAntiDeps" e "IncDataDeps", respetivamente.

Algoritmo 18 Preservação de Anti-Dependências

```
1: procedure PreservesAntiDependencies(N(m), R(B), S_B, I_B)
2:     antiDeps ← {}
3:     antiDepsWithDataDeps ← {}
4:     remaining ← (N(m)\S_B) ∪ I_B
5:     for each node ∈ remaining do
6:         for each antiDep ∈ OutAntiDeps(node) do
7:             if antiDep ∈ Edges(R(B)) & Target(antiDep) ∈ S_B\I_B then
8:                 antiDeps ← antiDeps ∪ {antiDep}
9:             for each dataDep ∈ IncDataDeps(node) do
10:                if Source(dataDep) ∈ R(B) & Source(dataDep) ∈ remaining &
   Variable(dataDep) = Variable(antiDep) then
11:                    antiDepsWithDataDeps ← antiDepsWithDataDeps ∪ {antiDep}
       return antiDeps \ antiDepsWithDataDeps = ∅
```

4.3.1.2 *Preservação de Dependências de Output*

Uma dependência *output* [KHoo], ou dependência *Write After Write (WAW)*, existe entre dois nodos p e q devido à variável v ($p \rightarrow_o^v q$), se ambos os nodos definirem o valor de v ($v \in Def(p) \wedge v \in Def(q)$) e existir pelo menos um caminho no CFG que inicie em p e termine em q, sem existir um nodo k intermédio que defina o valor de v (independentemente da existência de nodos k_i que utilizem o valor de v).

Ora, não deve existir uma dependência *output*, devido à variável v, na *block-based region* $R(B)$ entre um nodo p, que pertence à ação original após extração da *slice*, ($p \in U_B \cup I_B$), e um nodo q removível ($q \in S_B \backslash I_B$) [TC11]. Se a regra não se verificar, então a *slice* é inválida, pois pode alterar o comportamento inicial do programa, pelo que não deve ser apresentada. A Fórmula 12 formaliza a definição desta regra.

$$\{p \rightarrow q \in O_B(R(B)) \mid p \in U_B \cup I_B \wedge q \in S_B \backslash I_B\} = \emptyset \tag{12}$$

Na Fórmula 12, $O_B(R(B) = \{p \rightarrow_o^u q \mid p, q \in R(B)\})$ denota as dependências *output* da região $R(B)$, que podem ser transportadas por ciclos.

O Algoritmo 19 determina se a *slice* preserva dependências *output*, em pseudo-código.

Algoritmo 19 Preservação de Dependências *Output*

```
1: procedure PreservesOutputDependencies(N(m), R(B), S_B, I_B)
2:     outDeps ← {}
3:     remaining ← (N(m)\S_B) ∪ I_B
4:     for each node ∈ remaining do
5:         for each outDep ∈ OutgoingOutputDeps(node) do
6:             if outDep ∈ Edges(R(B)) & Target(outDep) ∈ S_B\I_B then
7:                 outDeps ← outDeps ∪ {outDep}
       return outDeps = ∅
```

4.3.2 *Regras Funcionais das Slices*

Ao contrário das regras anteriormente apresentadas, as regras relacionadas com a funcionalidade das *slices* têm como objetivo determinar quais as oportunidades de refatorização mais proveitosas. Estas regras estão relacionadas com o tamanho das *slices*, a duplicação de nodos e com a variável *output* da ação original:

- O número de nodos extraídos da união das *slices* US_B deve ser superior ao número de nodos que alteram o valor de v na *slice* S_B. Assim, as *slices* devem poder extrair nodos além dos nodos que alteram o valor de v.
- O número de nodos extraídos da união das *slices* US_B deve ser inferior ao número total de nodos da ação original. Assim, excluem-se as oportunidades de refatorização que extraem a ação na sua completude.
- Os nodos que alteram o valor da variável v não devem ser todos indispensáveis (i.e., duplicados na ação original e na ação extraída). Assim, excluem-se as oportunidades de refatorização cuja computação da variável esteja de igual modo presente na ação original.
- A variável retornada pela ação original não deve ser extraída, evitando que a ação extraída seja semelhante à ação original.

Além das regras acima definidas, o algoritmo está preparado para receber um conjunto de valores de *threshold* definidos pelo programador para aumentar a qualidade das *slices*:

- O programador pode definir um número mínimo de nodos para a *slice* ser sugerida, para evitar a sugestão de *slices* demasiado curtas.
- O programador pode definir um número máximo de nodos para a *slice* ser sugerida, para evitar a sugestão de *slices* demasiado longas.

- O programador pode definir um número máximo de nodos duplicados na ação original e na ação extraída, para limitar a redundância das *slices* sugeridas.
- O programador pode definir um valor máximo da razão de nodos duplicados na ação original e na ação extraída, para limitar a redundância das *slices* sugeridas.

4.3.3 *Ordenação das Oportunidades de Refatorização*

Após filtrar as *slices* que validam as regras nas secções acima definidas daquelas que potencialmente afetam a legibilidade e qualidade do código, tornou-se necessária a sua apresentação ordenada pelo ganho que elas representam. Deste modo, as *slices* são ordenadas pela ordem dos seguintes tópicos:

1. As *slices* são agrupadas em grupos de *slices* que computam a mesma variável.
2. Estes grupos são ordenados de forma decrescente pelo grau de duplicação que as suas *slices* em média apresentam.
3. No caso do grau de duplicação dos grupos ser igual, estes são ordenados de forma decrescente pela média de nodos duplicados das suas *slices*.
4. No caso da média de nodos duplicados dos grupos ser igual, estes são ordenados de forma crescente pelo número máximo de nodos das *slices*.
5. No caso do número máximo de nodos dos grupos ser igual, estes são ordenados de forma crescente pela média do número de nodos das *slices*.
6. Caso nenhuma destas ordenações for suficiente para ordenar os grupos, então são ordenados de forma crescente pelo tamanho da *string* da variável.
7. Depois dos grupos ordenados, as *slices* são ordenadas de forma decrescente pelo número de nodos removíveis.
8. No caso do número de nodos das *slices* ser igual, estas são ordenadas de forma crescente pelo grau de duplicação que apresentam.

4.4 LIMITAÇÕES

O algoritmo de *block-based slicing* utiliza *block-based regions* como meio de balizar a expansão das *slices*. Esta estratégia permite a extração de *slices* em situações em que não seria exequível a utilização de toda a ação como região. Apesar disso, as regiões são definidas pela identificação de nodos líder (ver subsecção 4.2.1), o que pode nem sempre conduzir

às melhores *slices*. Coloca-se a possibilidade da exploração de outras técnicas para definir regiões, alertando que a exploração de todas as regiões possíveis conduziria a um custo computacional acrescido. A título de exemplo, deve-se evitar que os nodos de tipo *query* (e.g., nodos SQL e Aggregate) sejam duplicados (indispensáveis) para evitar a duplicação de consultas à base de dados. Assim, para evitar que estes sejam nodos indispensáveis, pode-se definir regiões que iniciem em nodos diretamente consecutivos a nodos do tipo *query*.

As ações de preparação têm funcionalidades distintas das restantes ações de lógica de negócio, pois são executadas antes do carregamento da página *web* associada, por forma a preparar o ambiente que esta necessita para ser executada. Entre as múltiplas funcionalidades, as ações de preparação geram variáveis partilhadas com os *widgets* da página *web* em questão, cujos nodos que as definem não devem ser removidos. A título de exemplos, os nodos *SQL* e *Aggregate* (ver ponto 2.2.1.1) criam variáveis implícitas para agilizar alguns processos de desenvolvimento, que podem ser utilizados pela página *web* e não podem, por esse motivo, ser removidos. Por este motivo, estas ações não são atualmente consideradas para refatorização, embora, a sua inclusão seja, essencialmente, um exercício de implementação.

Como explicado no ponto 2.2.1.1, as atribuições na plataforma OutSystems são feitas com recurso a nodos *Assign*, que podem conter múltiplas atribuições (i.e., *Assignments*). O algoritmo faz um pré-processamento interno para separar as múltiplas atribuições de um nodo *Assign* em múltiplos nodos *Assign* com um único *Assignment*. Ou seja, o algoritmo pode gerar *slices* que removem nodos que inicialmente não eram visíveis para o programador, i.e., nodos gerados que correspondem aos consecutivos *Assignments*. Apesar disso, sublinha-se que um nodo *Assign* com múltiplos *Assignments* representa complexidade computacional semelhante a múltiplos nodos *Assign* com um único *Assignment*.

4.5 Exemplo de Extract Action na ação Statement

Como se pode verificar pela Figura 7, a ação *Statement* é composta por um conjunto de variáveis, entre as quais se destacam as que são calculadas na ação: *RenterPoints*, *TotalAmount* e *Result*. Para efeitos de exemplificação, foi escolhida a variável *RenterPoints* para demonstrar o modo de funcionamento do algoritmo.

A Figura 12 destaca, circulados a negro, os nodos que definem o valor da variável *RenterPoints*.

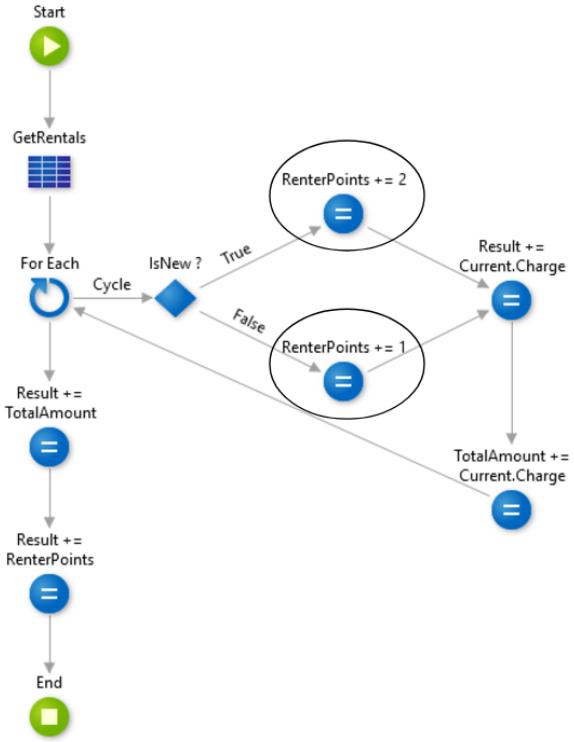


Figura 12.: Nodos que definem o valor de *RenterPoints* na ação *Statement*

Como explicado na subsecção 4.2.1 (Determinação de *Basic Blocks*), foram identificados 7 *basic blocks*, entre os quais B_4 e B_5 são os que contêm nodos que definem o valor de *RenterPoints*. A Figura 13 ilustra, a preenchido, os *basic blocks* que contêm os nodos que definem *RenterPoints* e, a tracejado, os restantes *basic blocks*.

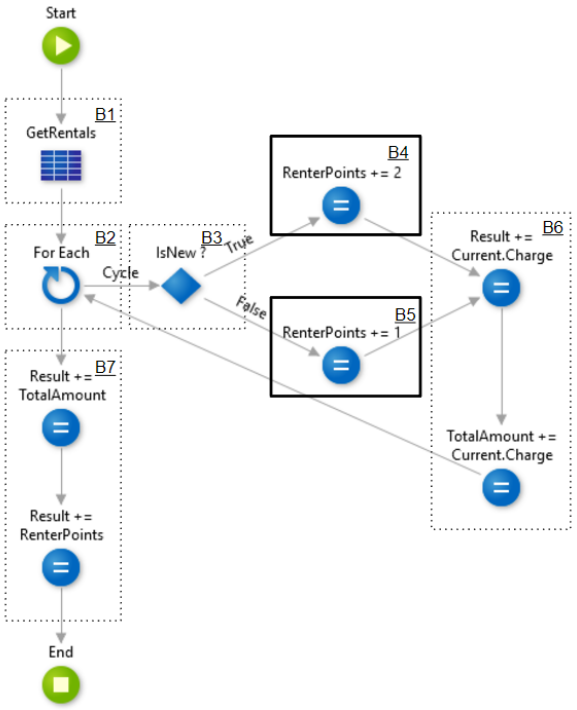


Figura 13.: Basic blocks que contêm os nodos que definem o valor de *RenterPoints* na ação *Statement*

Como explicado na subsecção 4.2.4 (Determinação de *Boundary Blocks*), os *boundary blocks* de B_4 e de B_5 são, respetivamente, $Blocks(B_4) = \{B_1, B_2, B_3, B_4\}$ e $Blocks(B_5) = \{B_1, B_2, B_3, B_5\}$.

Dado que o objetivo da técnica de *slicing* é extrair o cálculo de uma variável v no contexto de uma *block-based region*, então, é necessário identificar as regiões comuns aos nodos que definem v. Para tal, calcula-se a interseção dos *boundary blocks* identificados: $Blocks(B_4) \cap Blocks(B_5) = \{B_1, B_2, B_3\}$. É a partir destes *boundary blocks* que são calculadas as *block-based slices*, pois representam regiões comuns ao cálculo de *RenterPoints*. Assim, é possível extrair o comportamento de *RenterPoints* em 3 zonas distintas (B_1, B_2 *e* B_3), a partir dos nodos *"GetRentals"*, *"For Each"* e *"IsNew ?"*.

Para efeitos de demonstração do algoritmo, foi selecionado o *boundary block* B_2 como mediador da expansão da *slice*. Recorrendo ao Algoritmo 11, calculam-se, então as 2 *slices*, uma para cada nodo que define *RenterPoints*, na região do *boundary block* B_2. Note-se que, em situações normais, seriam calculadas 6 *slices* (dado terem sido identificados 3 *boundary blocks*).

As 2 *slices* calculadas identificam-se de seguida:

$$S_B("RenterPoints+=2", "RenterPoints", B_2) =$$
$$\{"ForEach", "IsNew?", "RenterPoints+=2"\}$$

$$S_B(\text{"}RenterPoints+=1\text{"}, \text{"}RenterPoints\text{"}, B_2) = \{\text{"}ForEach\text{"}, \text{"}IsNew?\text{"}, \text{"}RenterPoints+=1\text{"}\}$$

Para possibilitar a remoção de um maior número de nodos, o algoritmo implementado une as 2 *slices*. A união das *slices* é importante, pois permitiu tornar o nodo *"Is New ?"* num nodo removível, caso contrário, seria considerado um nodo indispensável. Assim, a Figura 14 ilustra a união das 2 *slices*. A região definida a tracejado é a *block-based region* do *boundary block* B_2 (ver subsecção 4.2.5). Os nodos destacados a verde são nodos que podem ser removidos da ação, enquanto que os nodos destacados a vermelho são nodos indispensáveis (ver ponto 4.2.6.2) que devem ser duplicados, após *Extract Action*.

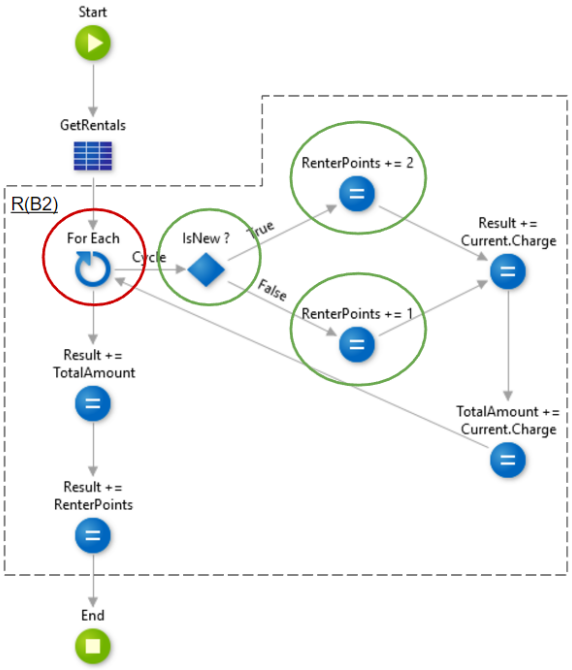


Figura 14.: Nodos pertencentes à união das *slices*

Assim, após a aplicação de *Extract Action*, é gerada uma nova ação, ilustrada na Figura 15, que calcula o valor de *RenterPoints*.

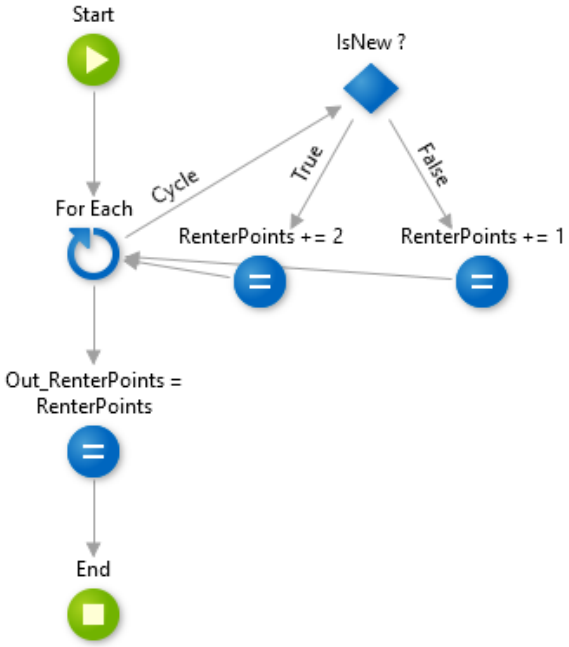


Figura 15.: Ação gerada, após *Extract Action*

Como se pode verificar na Figura 15, a ação gerada itera sobre a mesma lista da ação original mas, ao contrário desta, não tem um nodo *Aggregate*. Assim, como foi explicado anteriormente, é necessário adicionar um parâmetro de *input* para que a nova ação possa iterar sobre essa lista (ver ponto 4.2.6.3).

Após a extração da computação de *RenterPoints* para a ação gerada, é necessário remover os nodos removíveis da ação original e invocar a nova ação (Figura 16).

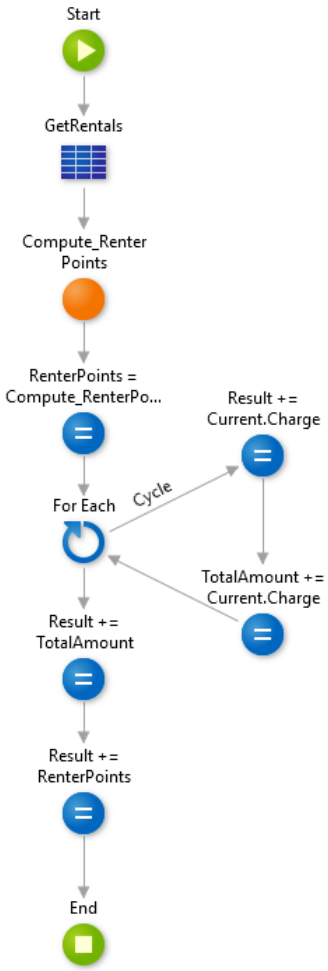


Figura 16.: Ação *Statement*, após *Extract Action*

5

CASO DE ESTUDO

O caso de estudo selecionado é composto por 9 módulos, desenvolvidos na plataforma OutSystems para uso interno da organização. Estes módulos foram desenvolvidos pela equipa de TI da OutSystems para fornecer aos seus colaboradores meios de análise de testes de regressão, suporte a processos de integração de novos colaboradores, listagem da informação de funcionários, gestão de prémios salariais e agendamento de férias.

Os 9 módulos referidos foram escolhidos por terem um conjunto de caraterísticas que são adversas para a qualidade de *software*. São aplicações de média complexidade, desenvolvidas nos anos de 2011 e 2014 e receberam, em média, contribuições de cerca de 20 profissionais. O desenvolvimento dos projetos iniciou em equipas especializadas, mas, no decorrer dos últimos anos, acolheram ambientes de manutenção dotados de equipas rotativas. Até à data, desde o arranque das fases de desenvolvimento de cada um dos projetos, as aplicações aumentaram de complexidade e divergiram notoriamente das propostas de solução iniciais.

As adversidades presentes nestes projetos são expectáveis num departamento de TI e são, por isso, um bom caso de estudo.

Os 9 módulos explorados contêm, no total, 1977 ações, das quais 1643 ações foram alvo de avaliação. Este subconjunto está relacionado com as limitações de implementação apresentadas na secção 4.4 (Limitações), i.e., as ações de preparação não são consideradas para refatorização.

Com o objetivo de identificar os módulos que apresentam maior necessidade de refatorização (maior complexidade), procedeu-se a uma avaliação baseada num conjunto de métricas. Conforme mencionado nos detalhes da implementação da secção 4.1 (Análise de Complexidade), as métricas LOC e CC foram as selecionadas para aferir o grau de complexidade de ações em OutSystems.

5.1 ANÁLISE DE LINES OF CODE (LOC)

Com esta avaliação, pretende-se compreender se a abordagem escolhida reduz, de facto, a complexidade dos programas, em particular perceber qual o potencial de redução de

complexidade medida em LOC. A Figura 17 representa a distribuição do número de nodos, que oscila entre 2 e 256 nodos. Existe uma clara amplitude no número de nodos das ações, que são, numa grande maioria, ações de tamanho reduzido.

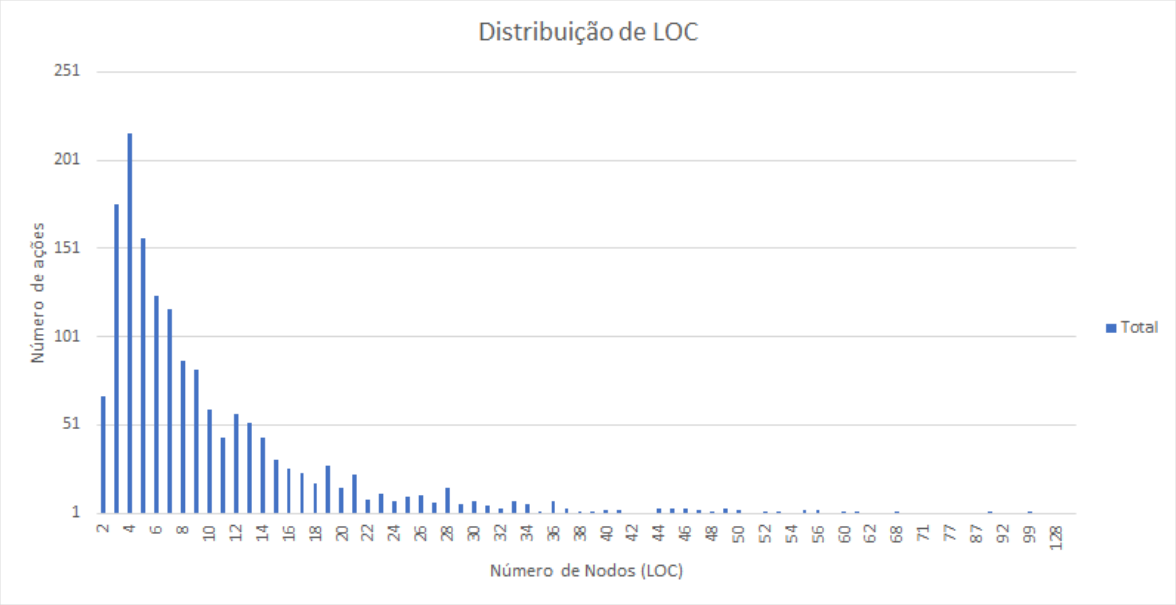


Figura 17.: Distribuição do Número de Nodos das ações

Dada a elevada quantidade de ações com reduzido número de nodos, que não contribuíram o suficiente para a redução da complexidade global dos módulos, tornou-se importante a seleção das ações mais relevantes para refatorização. Para tal, estabeleceu-se um valor limite (*threshold*) para aferir o grau de complexidade das ações.

Assim, o valor de *threshold* utilizado é o indicado em [VRvdL+16]: no máximo 15 nodos. Deste modo, as ações que tiverem 15 ou menos nodos são consideradas menos relevantes para refatorizar e não integram esta análise.

Do total de 1643 ações identificadas, 331 ações ($\approx$ 20% da totalidade) têm mais de 15 nodos. Dado o critério definido em [VRvdL+16], estas ações são as mais relevantes para aplicação de *Extract Action*. Cerca de 88% das ações acima do *threshold* de LOC (290 ações do total de 331 ações avaliadas) produziram, pelo menos, uma *slice* válida. Após refatorização, 49 ações ($\approx$ 17% das 290 ações que se situavam acima do valor de *threshold*) conseguiram reduzir o número de nodos abaixo do valor de *threshold* escolhido e, então, não necessitam de posterior refatorização, baseada na métrica LOC. As restantes 241 ações ($\approx$ 83% das 290 ações que se situavam acima do valor de *threshold*) reduziram de complexidade, mas mantiveram-se acima desse valor, após refatorização. A Tabela 2 resume os dados obtidos pela análise da refatorização realizada, face ao valor de *threshold* de 15 nodos, no máximo.

Tabela 2.: Resumo da avaliação medida em LOC

	Número de ações	**Percentagem (%)**
Número de ações acima do *threshold* de LOC	331	20%
Número de ações acima do *threshold* de LOC que produziram, pelo menos, uma *slice*	290	88%
Número de ações que se situam abaixo do *threshold* de LOC, após refatorização	49	17%
Número de ações que se situam acima do *threshold* de LOC, após refatorização	241	83%

Para compreender as propriedades das 49 ações que reduziram o número de nodos abaixo do valor de *threshold* de LOC, analisou-se o número de nodos que estas tinham antes de serem refatorizadas. Assim, a Figura 18 representa a distribuição do número de nodos das ações que outrora eram complexas, mas que, após uma aplicação de *Extract Action*, passaram a situar-se abaixo do valor de *threshold*.

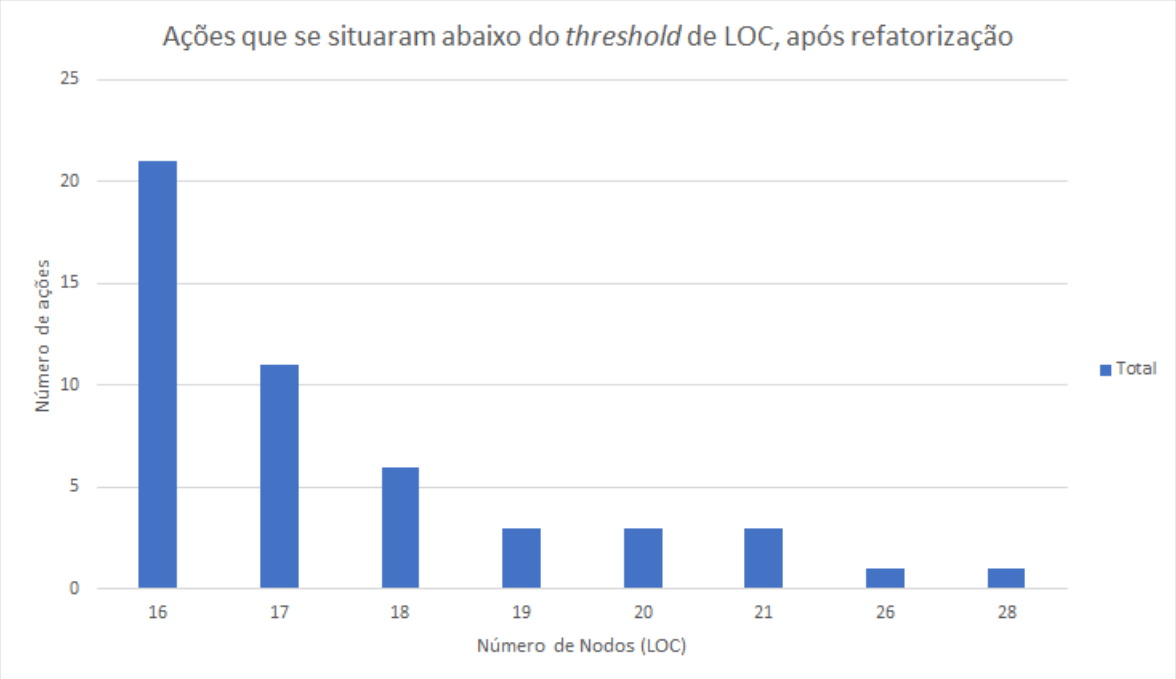


Figura 18.: Ações que se situaram abaixo do *threshold* de LOC, após *Extract Action*

A Figura 18 indica que as ações que eram complexas mas que, após *slicing*, conseguiram reduzir o número de nodos abaixo do valor de *threshold*, apresentam, maioritariamente, um valor de LOC compreendido entre 16 e 20 nodos. Isto leva a concluir que as ações mais propícias a deixarem de ser complexas após uma aplicação de *Extract Action*, são aquelas cujo número de nodos é relativamente próximo do valor de *threshold*.

De igual modo, analisou-se também o número de nodos das 241 ações que, apesar de reduzirem de complexidade, se mantiveram acima do valor de *threshold* de LOC. Assim, a Figura 19 representa a distribuição do número de nodos das ações que se mantêm complexas, após uma aplicação de *Extract Action*.

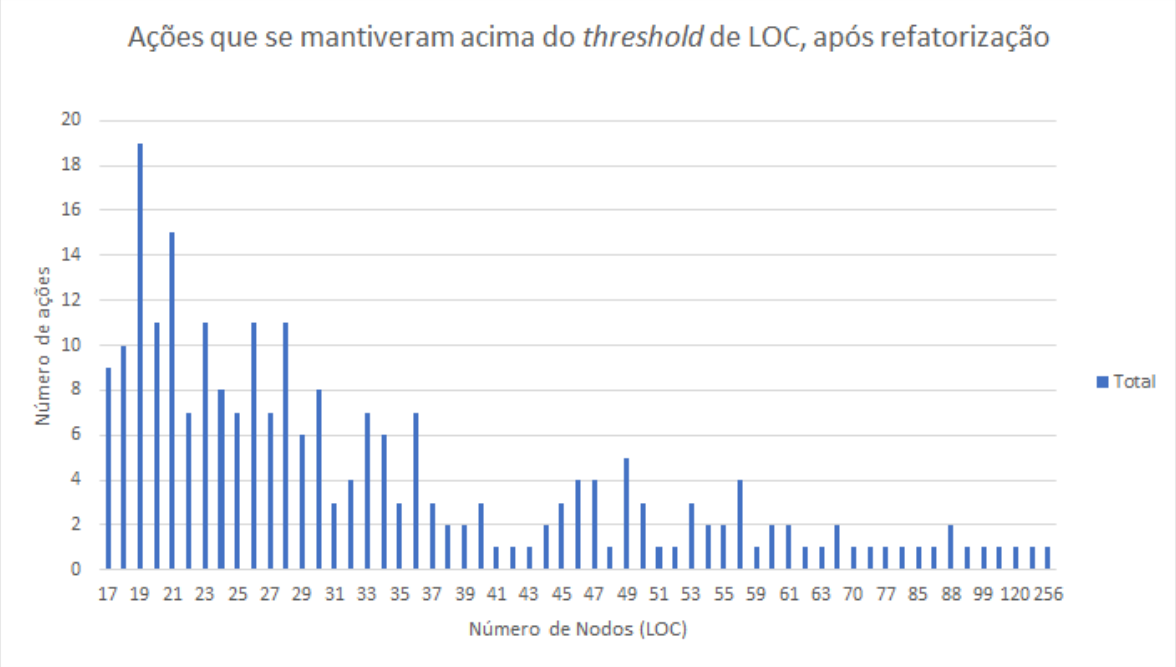


Figura 19.: Ações que se mantiveram acima do *threshold* de LOC, após *Extract Action*

A Figura 19 indica que as ações que se mantiveram complexas após refatorização, mas reduziram de complexidade, apresentam um número de nodos compreendido entre 17 e 256 nodos.

Como foi acima referido, após a aplicação de uma *Extract Action*, cerca de 241 ações reduziram de complexidade, mas mantiveram-se acima do valor de *threshold*, o que pode levar a concluir que uma aplicação única desta técnica pode ser insuficiente, em algumas situações. Das 241 ações que se mantêm acima do valor de *threshold*, existem 234 ações ($\approx$ 97%) que manifestaram capacidade de produzir, pelo menos, mais uma *slice* válida, ou seja, apresentam espaço para novas aplicações de *Extract Action*. A Figura 20 representa as ações que ainda têm espaço para novas refatorizações, após uma aplicação de *Extract Action*.

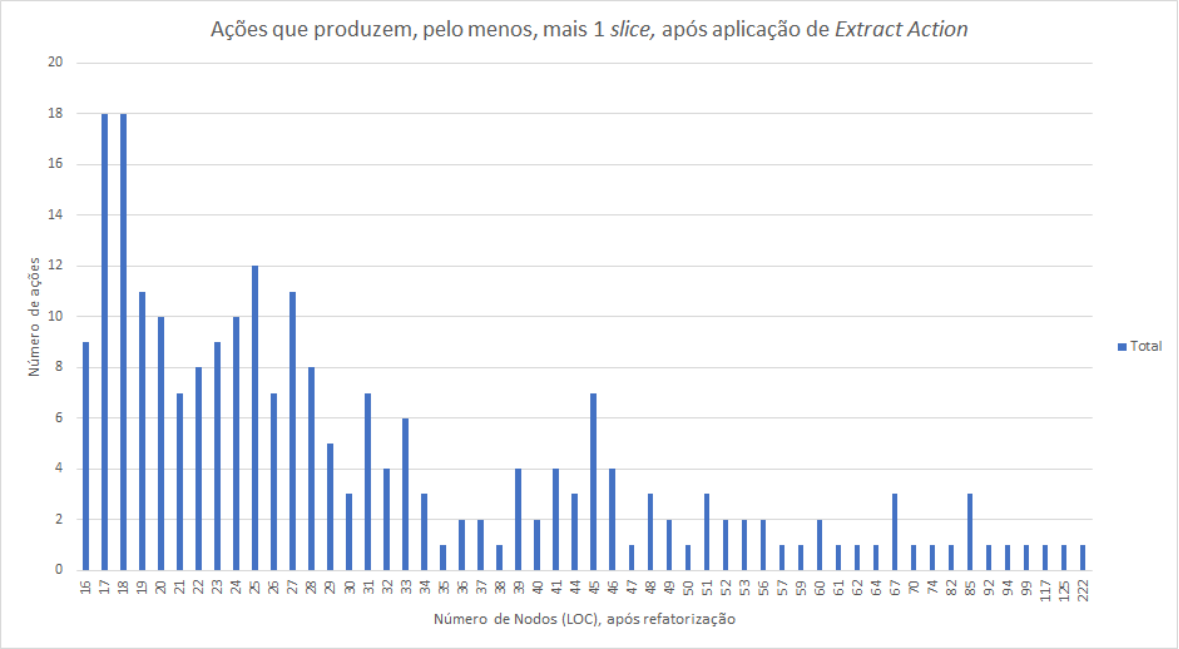


Figura 20.: Ações acima do *threshold* de LOC, após refatorização, que manifestaram, pelo menos, mais 1 *slice*

A Figura 20 representa a distribuição das ações que, após uma aplicação de *Extract Action*, ainda têm espaço para novas aplicações da mesma técnica. O número de nodos que estas apresentam, após uma aplicação da mesma técnica, oscila entre 16 e 222 nodos.

5.1.1 *Impacto da Abordagem de Slicing na Métrica Lines of Code*

Com o objetivo de estudar o impacto da refatorização *Extract Action* na redução da métrica LOC, conduziu-se um estudo acerca do ganho proporcionado pela técnica de *slicing*. A fim de remover algum ruído causado pelas oportunidades de refatorização menos relevantes, a exploração de dados atuou apenas nas melhores *slices* produzidas para cada ação, pois são essas que melhor reduzem a complexidade inicial da ação.

A Figura 21 representa o ganho médio absoluto de nodos removíveis pelas *slices*, face ao número de nodos que uma ação contém. O indicador de nodos removíveis é relevante, pois potencia a geração de ações com menos nodos, e a função de ganho é dada pela Fórmula 13.

$$G = \#\ Removable\ Nodes \tag{13}$$

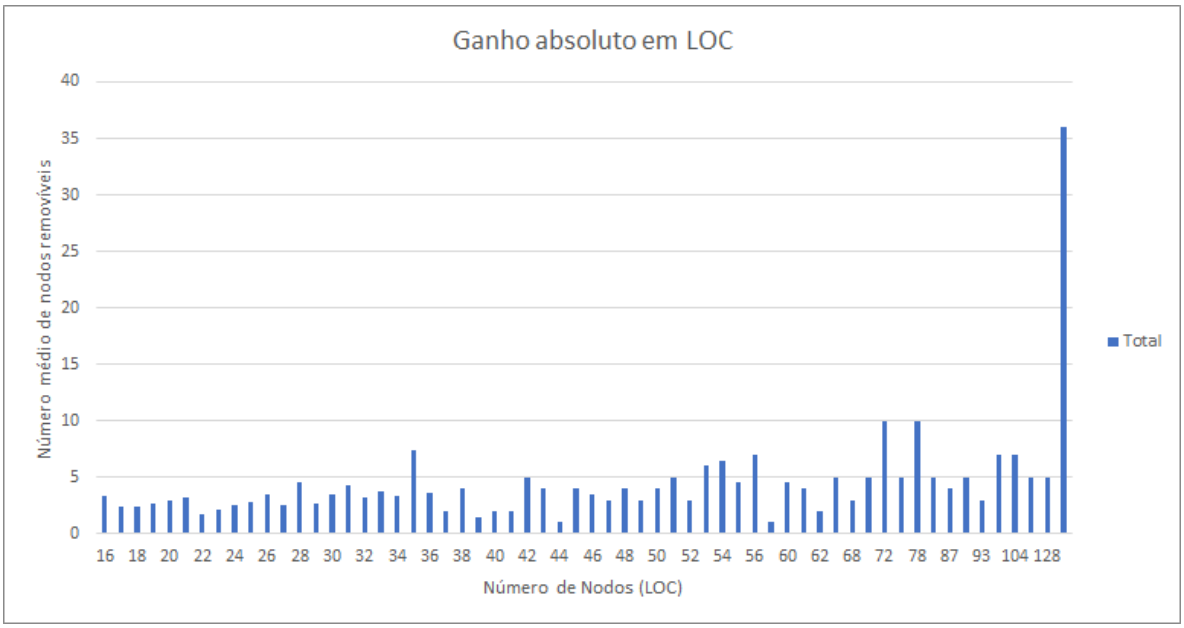


Figura 21.: Ganho médio absoluto medido em LOC

Pela análise da Figura 21, pode verificar-se que as ações de maior tamanho possibilitam a geração de *slices* capazes de remover um maior número de nodos, em comparação com ações curtas. A média de nodos removíveis apresentado pelas *slices* é de 5 nodos.

A Figura 22 representa o ganho médio relativo de nodos removíveis, face ao tamanho inicial de uma ação, cuja função de ganho é dada pela Fórmula 14:

$$G = \frac{\#\ Removable\ Nodes}{\#\ Initial\ Nodes} \tag{14}$$

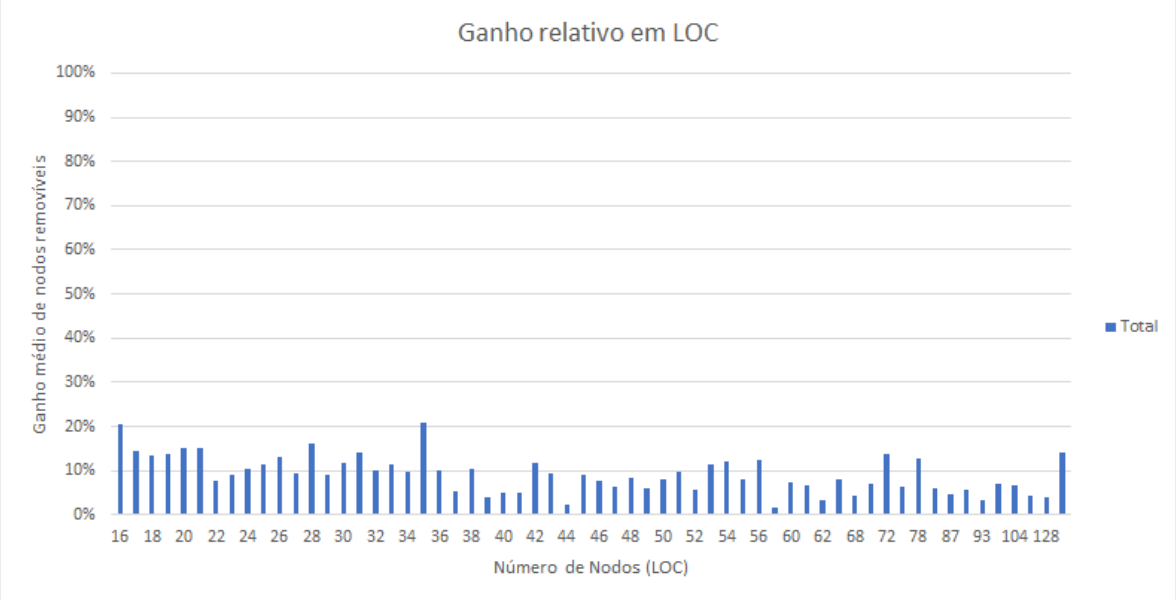


Figura 22.: Ganho médio relativo medido em LOC

O ganho representado pela Fórmula 14 é relevante, pois permite complementar a análise de ganho absoluto, dada pelo indicador utilizado pela Fórmula 13. Apesar de uma ação com um elevado número de nodos potenciar uma maior remoção de nodos, pode originar numa pequena redução percentual nesse valor. Isto leva a crer que pode ser útil uma aplicação complementar de *Extract Action*, com o objetivo de melhorar os indicadores de ganho absoluto e relativo dos nodos removíveis.

5.2 Análise de Cyclomatic Complexity (CC)

Com esta avaliação, pretende-se compreender se a abordagem escolhida reduz, de facto, a complexidade dos programas, em particular perceber qual o potencial de redução de complexidade medida em CC. A Figura 23 representa a distribuição da complexidade ciclomática, que oscila entre 1 e 41 unidades, e as ações são, numa grande maioria, ações com um número de caminhos possíveis reduzido.

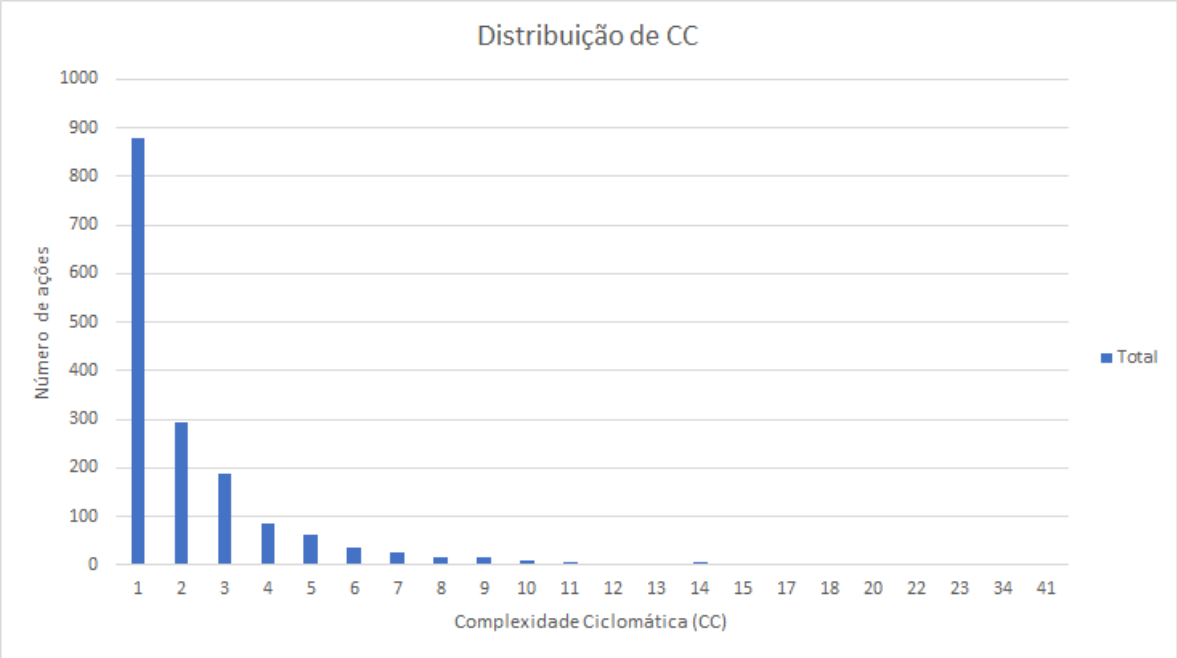


Figura 23.: Distribuição da Complexidade Ciclomática das ações

Dada a elevada quantidade de ações com reduzido valor de complexidade ciclomática, que não contribuíram o suficiente para a redução da complexidade global dos módulos, tornou-se importante a seleção das ações mais relevantes para refatorização. Para tal, estabeleceu-se um valor limite (*threshold*) para aferir o grau de complexidade das ações.

Assim, o valor de *threshold* utilizado é o indicado em [VRvdL+16]: no máximo 4 unidades de CC. As ações que tiverem uma valor de complexidade ciclomática igual ou inferior a 4 unidades são consideradas menos relevantes para refatorizar e não integram esta análise.

Do total de 1643 ações identificadas, 199 ações (≈ 12% da totalidade) têm um valor de complexidade ciclomática superior a 4 unidades. Dado o critério definido em [VRvdL+16], estas ações são as mais relevantes para aplicação de *Extract Action*. Cerca de 89% das ações acima do *threshold* de CC (178 ações do total de 199 ações avaliadas) produziram, pelo menos, uma *slice* válida. Após refatorização, 32 ações (≈ 18% das 178 ações que se situavam acima do valor de *threshold*) conseguiram reduzir o valor de CC abaixo do valor de *threshold* escolhido e, então, não necessitam de posterior refatorização, baseada na métrica CC. As restantes 146 ações (≈ 82% das 178 ações que se situavam acima do valor de *threshold*) reduziram de complexidade, mas mantiveram-se acima desse valor, após refatorização. A Tabela 3 resume os dados obtidos pela análise da refatorização realizada, face ao valor de *threshold* de 4 unidades de complexidade ciclomática, no máximo.

Tabela 3.: Resumo da avaliação medida em CC

	Número de ações	**Percentagem (%)**
Número de ações acima do *threshold* de CC	199	12%
Número de ações acima do *threshold* de CC que produziram, pelo menos, uma *slice*	178	89%
Número de ações que se situam abaixo do *threshold* de CC, após refatorização	32	18%
Número de ações que se situam acima do *threshold* de CC, após refatorização	146	82%

Para compreender as propriedades das 32 ações que reduziram o valor da complexidade ciclomática abaixo do valor de *threshold* de CC, analisou-se o valor desta métrica antes de serem refatorizadas. Assim, a Figura 24 representa a distribuição do número de nodos das ações que outrora eram complexas, mas que, após uma aplicação de *Extract Action*, passaram a situar-se abaixo do valor de *threshold*.

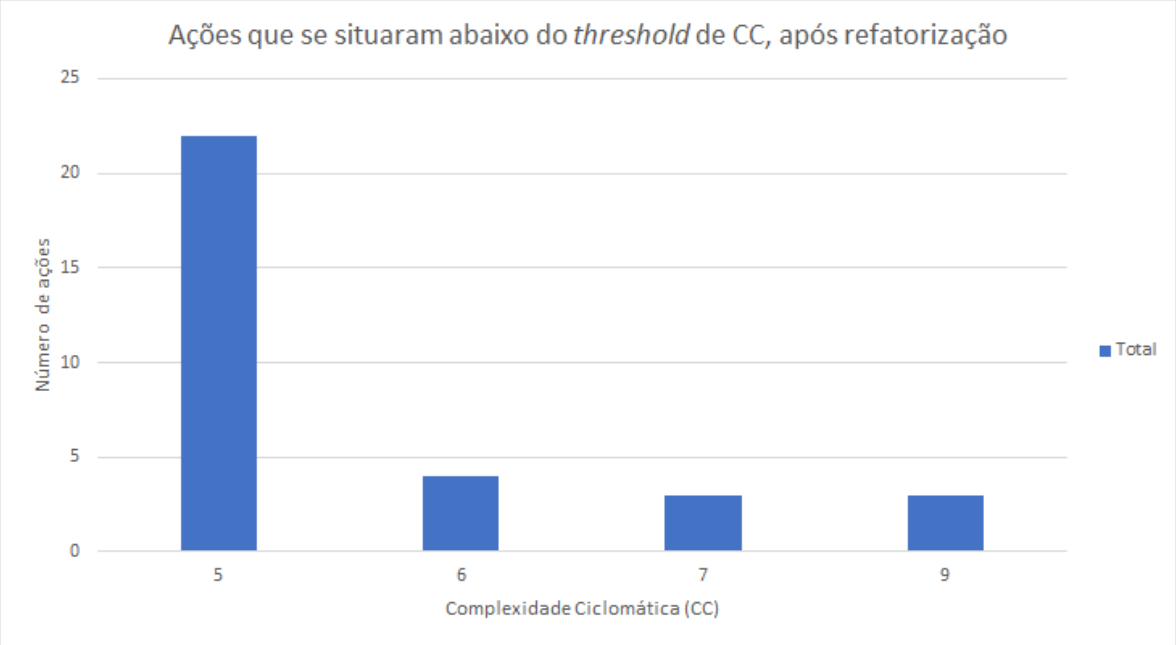


Figura 24.: Ações que se situaram abaixo do *threshold* de CC, após *Extract Action*

A Figura 24 indica que as ações que eram complexas mas que, após *slicing*, conseguiram reduzir o valor da sua complexidade ciclomática abaixo do valor de *threshold*, apresentam

um valor CC compreendido entre 5 e 9 unidades. Isto leva a concluir que as ações mais propícias a deixarem de ser complexas após uma aplicação de *Extract Action*, são aquelas cuja complexidade ciclomática é relativamente próxima ao valor de *threshold*.

De igual modo, analisou-se também o valor da complexidade ciclomática das 146 ações que, apesar de reduzirem de complexidade, se mantiveram acima do valor de *threshold* de CC. Assim, a Figura 25 representa a distribuição da complexidade ciclomática das ações que se mantêm complexas, após uma aplicação de *Extract Action*.

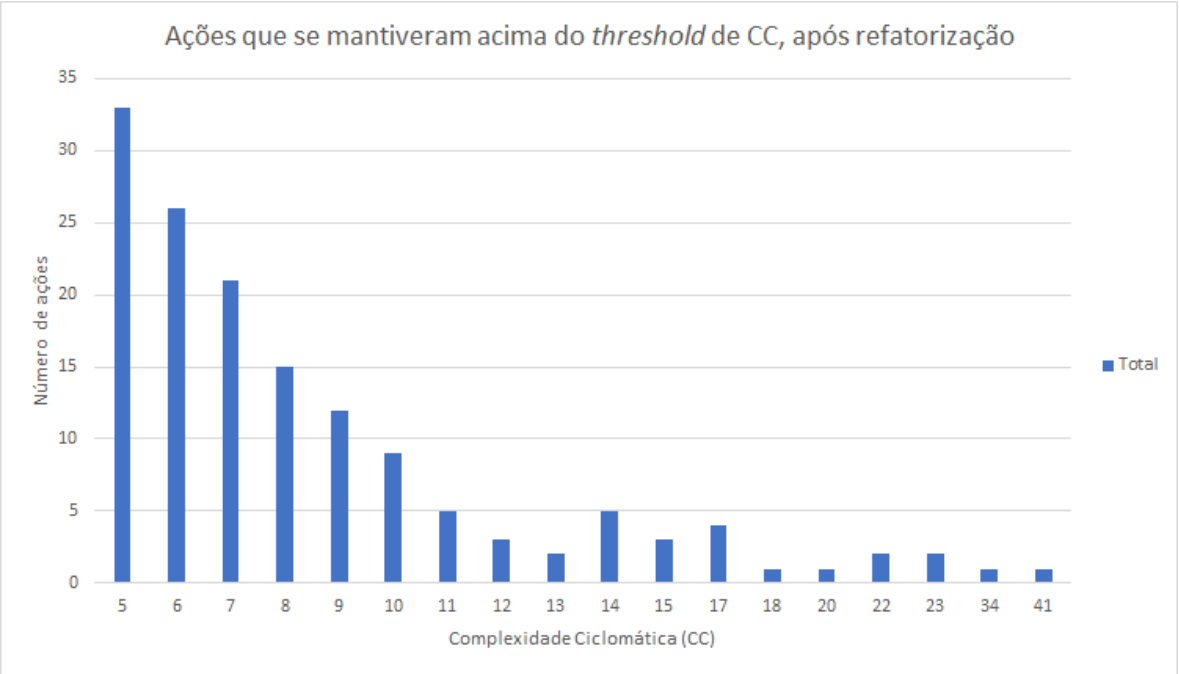


Figura 25.: Ações que se mantiveram acima do *threshold* de CC, após *Extract Action*

A Figura 25 indica que as ações que se mantiveram complexas após refatorização, mas reduziram de complexidade, apresentam um valor de CC compreendido entre 5 e 41 unidades.

Como foi acima referido, após a aplicação de uma *Extract Action*, cerca de 146 ações reduziram de complexidade, mas mantiveram-se acima do valor de *threshold*, o que pode levar a concluir que uma aplicação única desta técnica pode ser insuficiente, em algumas situações. Das 146 ações que se mantêm acima do valor de *threshold*, existem 115 ações ($\approx$ 79%) que manifestaram capacidade de produzir, pelo menos, mais uma *slice* válida, ou seja, apresentam espaço para novas aplicações de *Extract Action*. A Figura 26 representa as ações que ainda têm espaço para novas refatorizações, após uma aplicação de *Extract Action*.

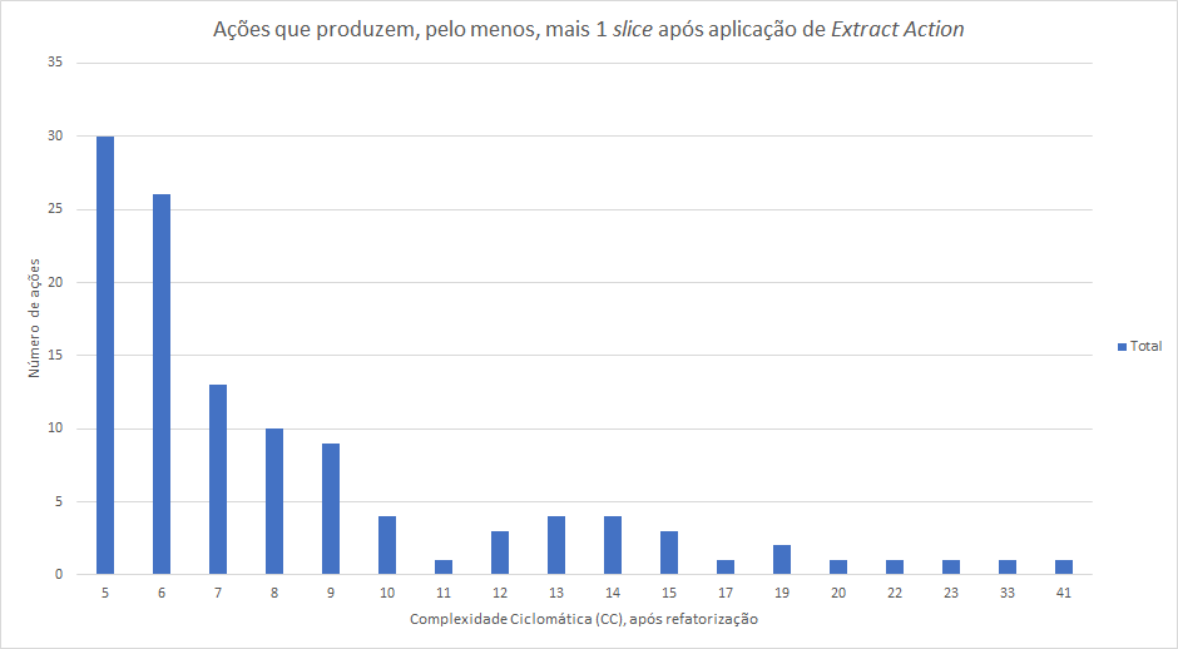


Figura 26.: Ações acima do *threshold* de CC, após refatorização, que manifestaram, pelo menos, mais 1 *slice*

A Figura 26 representa a distribuição das ações que, após uma aplicação de *Extract Action*, ainda têm espaço para novas aplicações da mesma técnica. O valor de CC que estas apresentam, após uma aplicação da mesma técnica, oscila entre 5 e 41 unidades.

5.2.1 *Impacto da Abordagem de Slicing na Métrica Cyclomatic Complexity*

Com o objetivo de estudar o impacto da refatorização *Extract Action* na redução da métrica CC, conduziu-se um estudo acerca do ganho proporcionado pela técnica de *slicing*. A fim de remover algum ruído causado pelas oportunidades de refatorização menos relevantes, a exploração de dados atuou apenas nas melhores *slices* produzidas para cada ação, pois são essas que melhor reduzem a complexidade inicial da ação.

A Figura 27 representa o ganho médio absoluto de nodos removíveis pelas *slices*, face ao número de nodos que uma ação contém. O indicador de nodos removíveis é relevante, pois potencia a geração de ações com menos nodos, e a função de ganho é dada pela Fórmula 15.

$$G = CC\ before\ extraction - CC\ after\ extraction \tag{15}$$

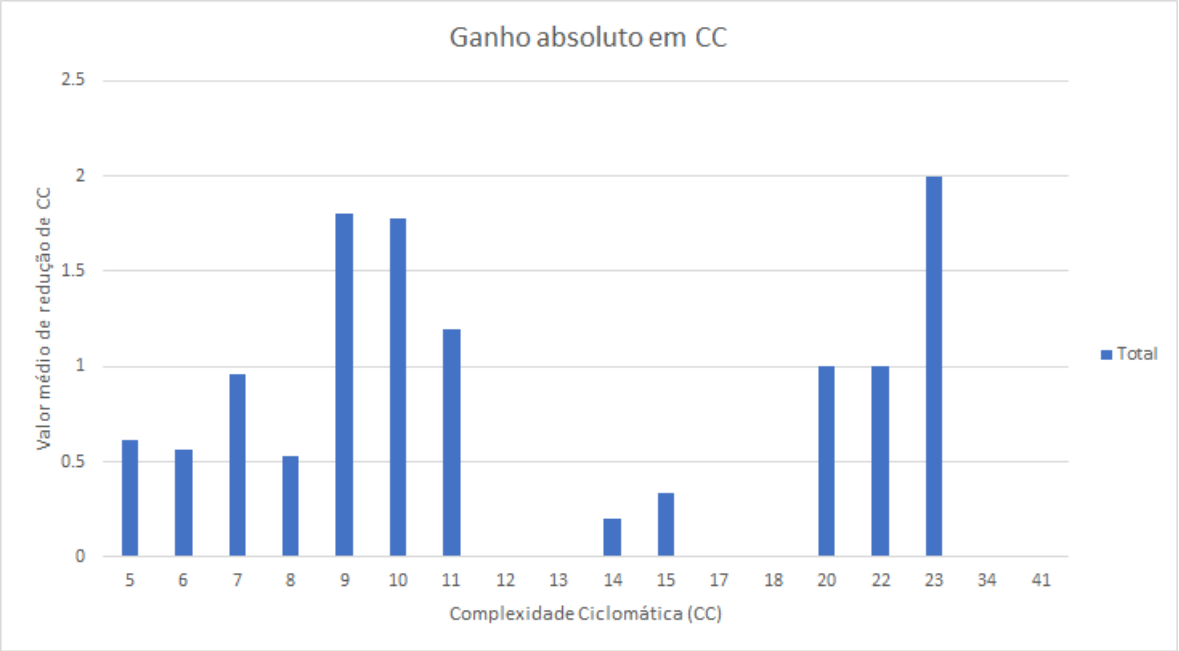


Figura 27.: Ganho médio absoluto medido em CC

A análise da Figura 27 permite verificar que as ações tendencialmente com maior número de bifurcações possibilitam a geração de *slices* capazes de melhor reduzir o seu valor, em comparação com ações com reduzido valor de CC. A média de redução de complexidade ciclomática é de 1 unidade.

A Figura 28 representa o ganho médio relativo no valor de CC, face ao valor inicial da complexidade ciclomática de uma ação, cuja função de ganho é dada pela Fórmula 16.

$$G = \frac{CC\ before\ extraction - CC\ after\ extraction}{CC\ before\ extraction} \tag{16}$$

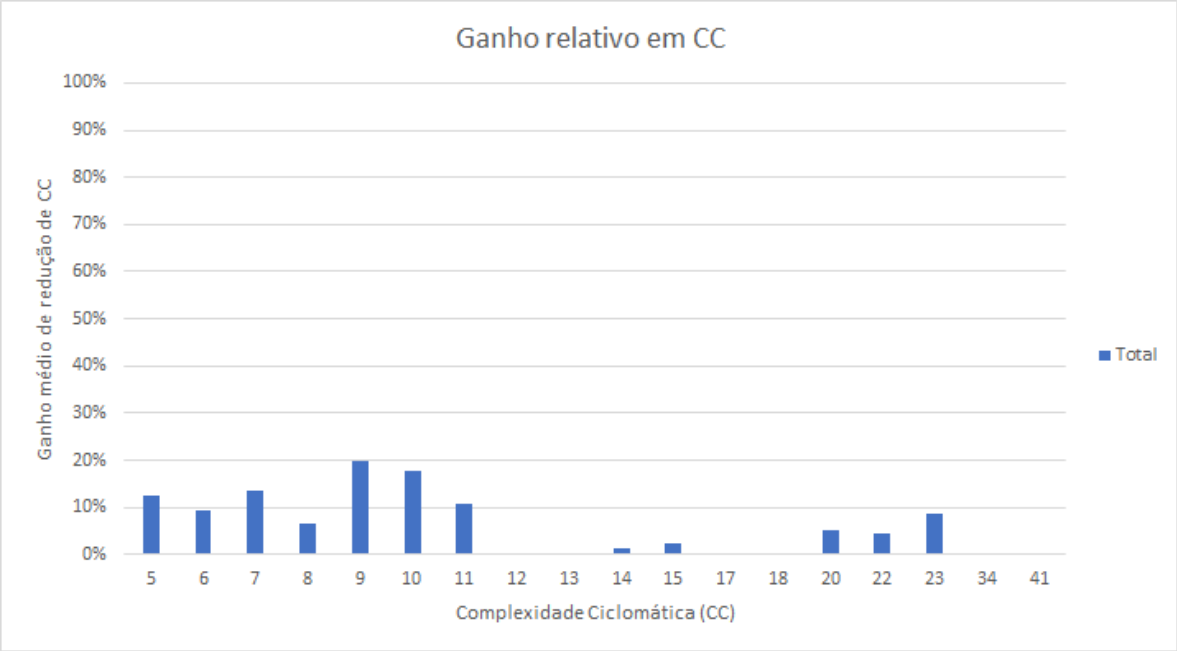


Figura 28.: Ganho médio relativo medido em CC

O ganho representado pela Fórmula 16 é relevante, pois permite complementar a análise de ganho absoluto, dada pelo indicador utilizado pela Fórmula 15. Apesar de uma ação com um elevado valor de complexidade ciclomática potenciar uma maior remoção de bifurcações, pode originar numa pequena redução percentual nesse valor. Isto leva a crer que pode ser útil a aplicação múltipla do *Extract Action*, com o objetivo de melhorar os indicadores de ganho absoluto e relativo da complexidade ciclomática.

5.3 Exemplos de Ações que Reduziram de Complexidade

Esta secção mostra o resultado da aplicação da técnica *block-based slicing* em algumas ações dos módulos analisados, reforçando assim a utilidade desta técnica na redução de complexidade de ações desenvolvidas na plataforma OutSystems.

A Figura 29 ilustra a ação "TrialEndNotification", que é composta por 17 nodos e apresenta 7 unidades de CC. A ação efetua uma *query* à base de dados para consultar quem são os mais recentes colaboradores a juntarem-se à equipa da OutSystems. De seguida, essa informação é aglomerada numa única variável "Users" para enviar num *email* aos responsáveis.

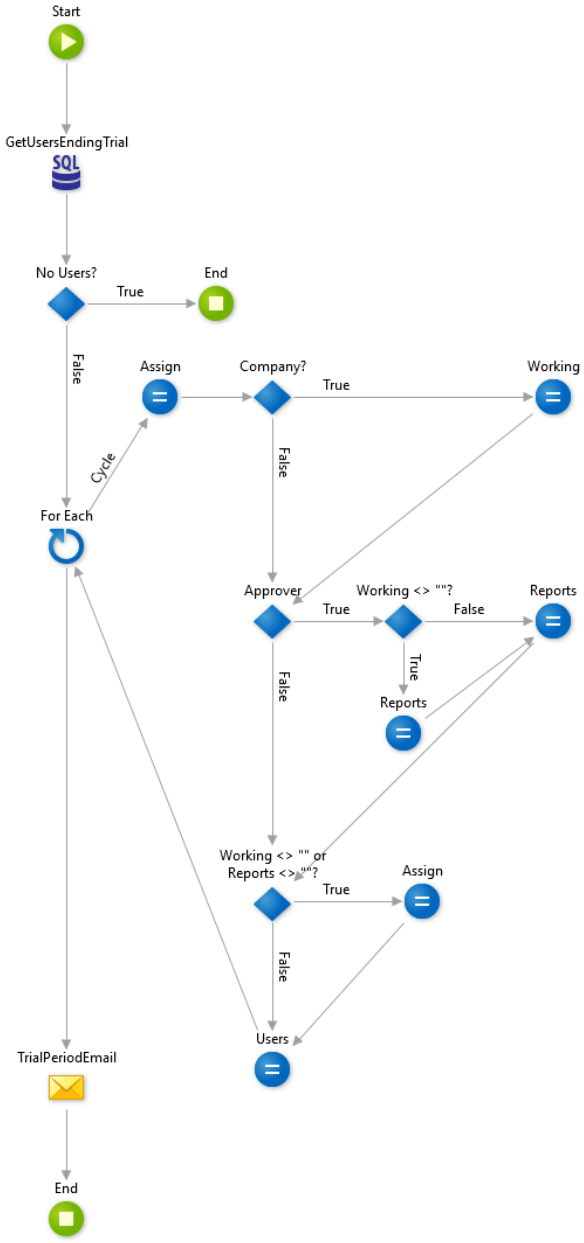


Figura 29.: Ação "TrialEndNotification", antes de aplicar *block-based slicing*

A melhor oportunidade de refatorização identificada (ver secção 4.3) na Figura 29 pelo algoritmo de *block-based slicing* representa uma *slice* para a computação da variável "Users" cuja *block-based region* inicia no nodo "For Each". Tal *slice* não seria possível se a abordagem não fosse baseada em regiões, i.e., se a abordagem escolhida não fosse *block-based*, então, para esta ação, seria necessário adicionar à *slice* os nodos anteriores ao nodo líder ("For Each") e duplicá-los nas duas ações.

A Figura 30 ilustra a ação "Compute_Users" gerada a partir dessa *slice*, que recebe por parâmetro a lista de colaboradores, calcula o valor de "Users" e retorna-o à ação original.

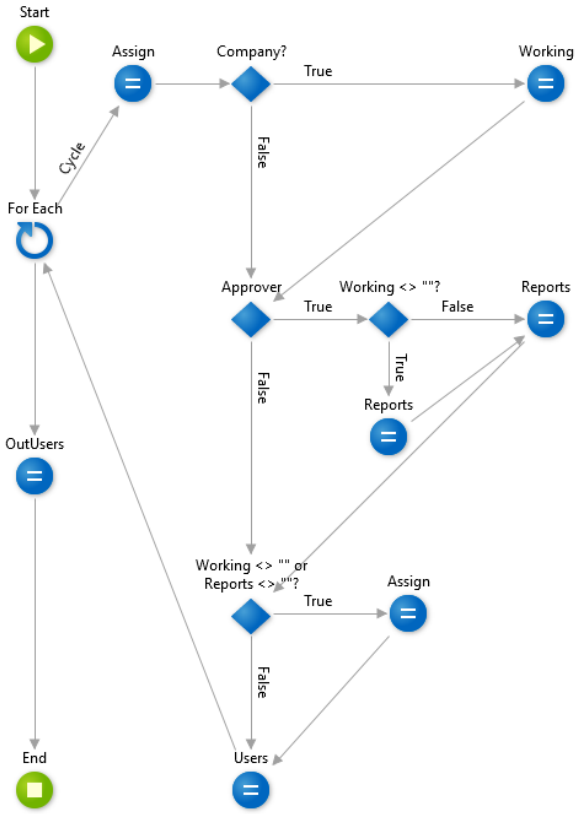


Figura 30.: Ação "Compute_Users" gerada a partir da ação "TrialEndNotification"

A Figura 31 ilustra a ação "TrialEndNotification" que, após refatorização, invoca a ação gerada "Compute_Users" da Figura 30 e reduz a sua complexidade de 17 nodos e 7 unidades de CC para 8 nodos e 2 unidades, respetivamente.

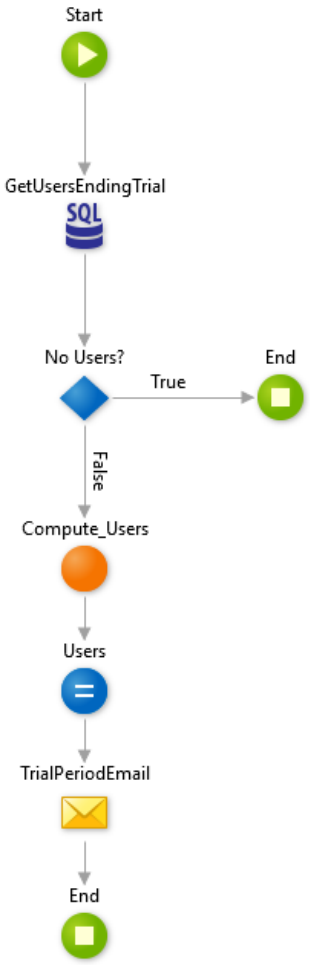


Figura 31.: Ação "TrialEndNotification", após aplicar *block-based slicing*

A Figura 32 ilustra a ação "TestSettingTestInstancesFullyMatchSettings", que é composta por 29 nodos e apresenta 10 unidades de CC. Esta ação, em primeiro lugar, pré-processa a variável passada por parâmetro "Instances" e efetua uma consulta à base de dados para obter um resultado que é utilizado pelo nodo *If* ligado ao nodo *Switch* pela condição *Otherwise* e pela *server action* "TestSetting_Contains" (ver ponto 2.2.1.1). Além disso, dependente da condição do nodo *Switch* que o fluxo seguir, a ação efetua uma consulta à base de dados para atribuir o valor da variável "VersionList".

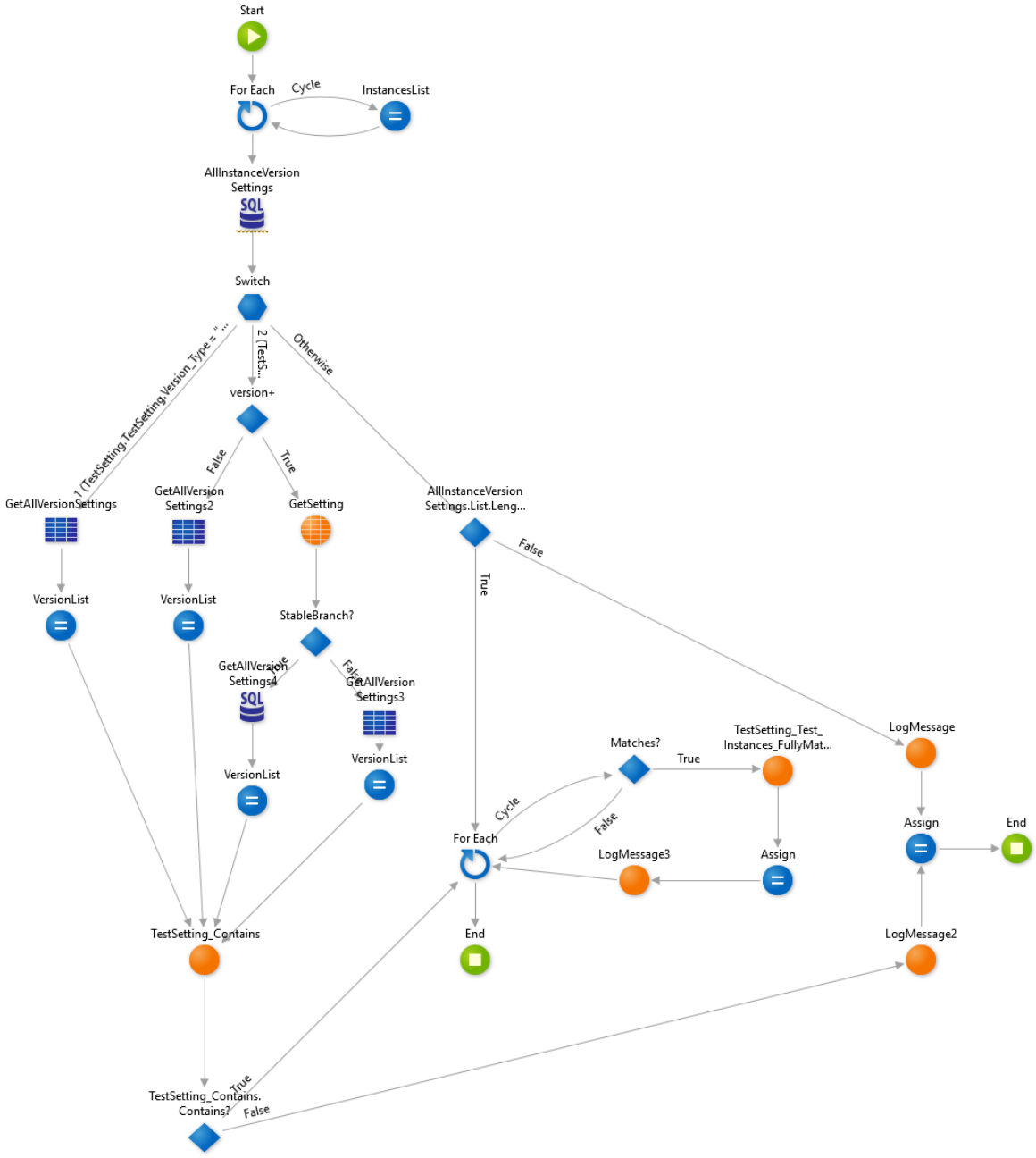


Figura 32.: Ação "TestSettingTestInstancesFullyMatchSettings", antes de aplicar *block-based slicing*

A melhor oportunidade de refatorização identificada na Figura 32 pelo algoritmo de *block-based slicing* representa uma *slice* para a computação da variável "VersionList" cuja *block-based region* inicia no primeiro nodo "For Each".

A Figura 33 ilustra a ação "Compute_VersionList" gerada a partir dessa *slice*, que recebe por parâmetro as variáveis "TestSetting" e "StableBranch" utilizadas pelos nodos "Switch", "version+" e "StableBranch?", calcula o valor de "VersionList" e retorna-o à ação original.

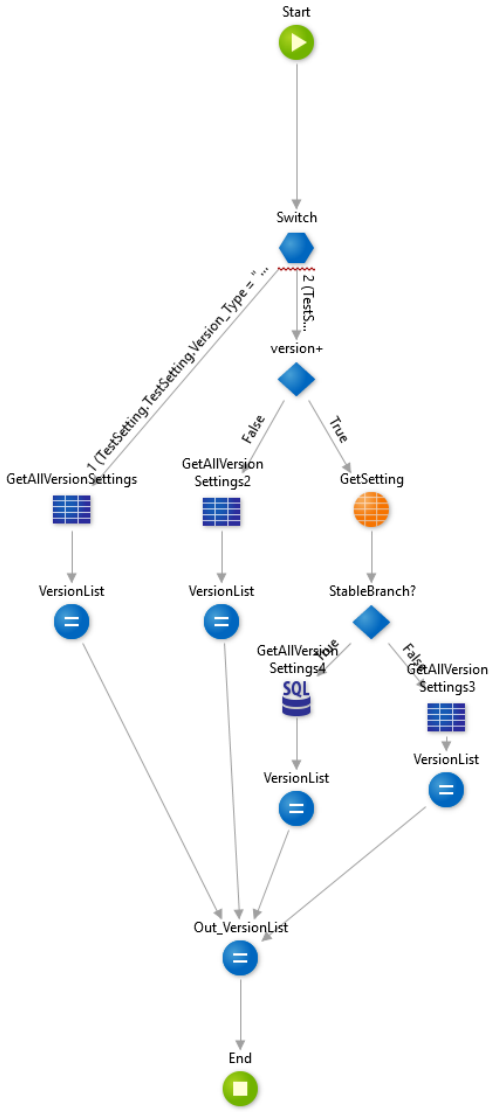


Figura 33.: Ação "Compute_VersionList" gerada a partir da ação "TrialEndNotification"

A Figura 34 ilustra a ação "TestSettingTestInstancesFullyMatchSettings", após refatorização. Esta invoca a ação gerada "Compute_VersionList" (Figura 33), reduzindo a sua complexidade de 29 nodos e 10 unidades de CC para 20 nodos e 7 unidades, respetivamente.

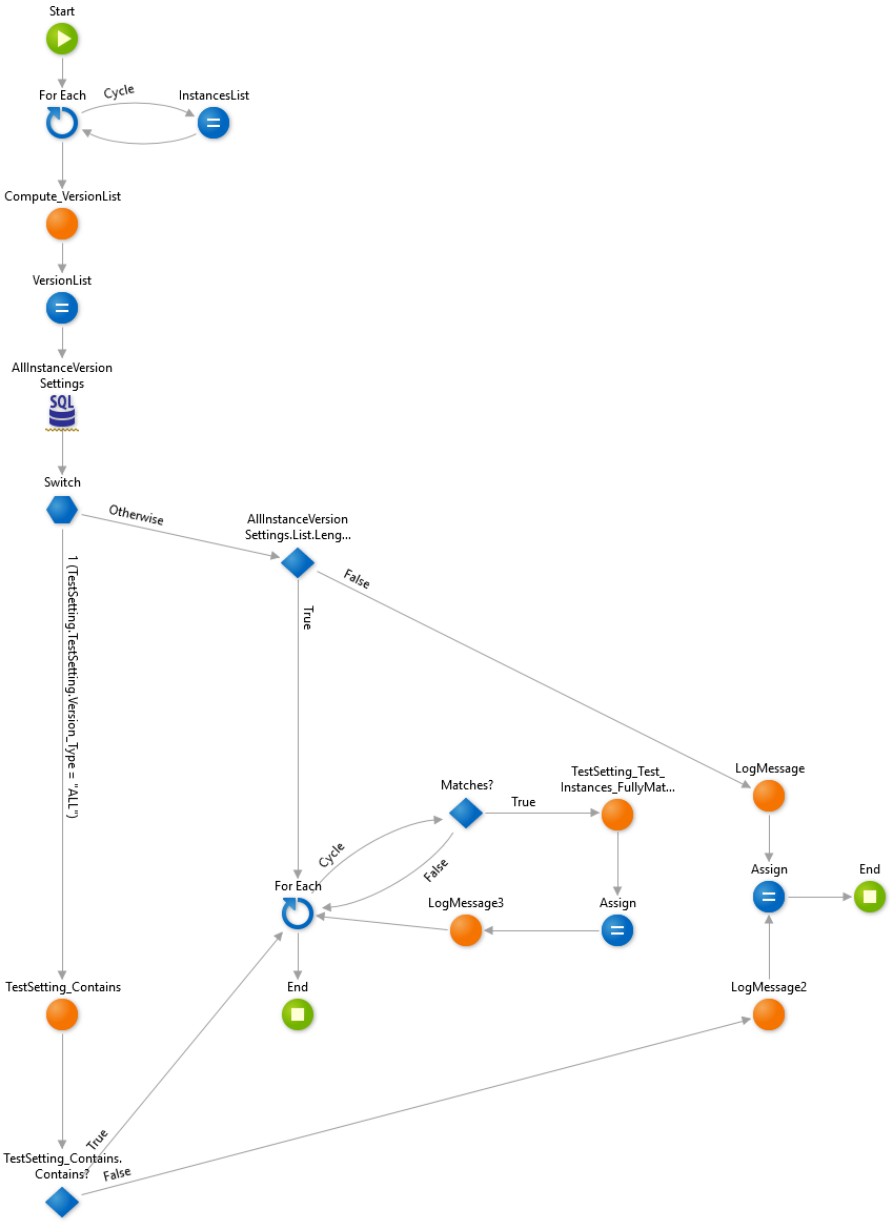


Figura 34.: Ação "TestSettingTestInstancesFullyMatchSettings", após aplicar *block-based slicing*

6

CONCLUSÃO

A fraca estruturação das aplicações de *software* afeta negativamente os seus custos de desenvolvimento e de manutenção, o que pode levar à ocorrência de *code smells*, designadamente módulos de elevada complexidade (*Long Actions*). Assim, o objetivo desta dissertação foi prover a plataforma OutSystems de um sistema capaz de analisar a complexidade de módulos e automaticamente sugerir oportunidades de refatorização *Extract Action*. Esta primitiva sugere a geração de ações que calculam o valor de variáveis, reduzindo o tamanho da ação original e aprimorando a sua leitura e compreensão.

A solução implementada combina o cálculo de métricas de complexidade com *program slicing* para apresentar as melhores oportunidades de refatorização de módulos desenvolvidos na plataforma OutSystems. Para tal, recorreu-se à análise das métricas LOC e CC como forma de aferir o grau de complexidade dos módulos. De seguida, recorreu-se a *program slicing* para reduzir a complexidade das ações, em particular utilizou-se a técnica *block-based slicing* para controlar a expansão das *slices* geradas. Além disso, a subsecção 4.3.1 apresenta um conjunto de regras que as *slices* produzidas devem respeitar, por forma a aferir a sua relevância e a sua capacidade de manter o comportamento funcional do programa.

O capítulo 5 (CASO DE ESTUDO) confirma a necessidade de ter uma ferramenta que automaticamente identifique melhorias no código OutSystems para manter o equilíbrio e coerência dos projetos. Assim, torna-se possível auxiliar o programador a refatorizar, através da sugestão das oportunidades de refatorização que melhor reduzem a complexidade dos módulos.

No decorrer desta dissertação, o autor submeteu o artigo científico "Support for Automatic Refactoring of Business Logic", que foi aceite como *regular paper* no *symposium* INFORUM 2017, em Portugal. Neste artigo, apresenta-se a abordagem desenvolvida como uma ferramenta adequada para a identificação de oportunidades de refatorização em módulos de grande complexidade.

6.1 PRINCIPAIS CONTRIBUIÇÕES

O capítulo 4 (DETALHES DA SOLUÇÃO) apresenta uma combinação de análise de complexidade de módulos com a sua refatorização. Por um lado, analisa a qualidade de módulos, com recurso ao cálculo de métricas de complexidade e, por outro, reduz o grau de complexidade de módulos, com recurso à técnica *block-based slicing*.

As ferramentas estudadas na secção 3.4 (Ferramentas de Program Slicing) têm como foco a refatorização assistida de programas desenvolvidos em linguagens de programação imperativa e orientada a objetos. Esta dissertação confirma a necessidade de que as primitivas de refatorização também são relevantes em DSLs *low-code*, como é o exemplo da plataforma OutSystems. Além disso, o capítulo 5 (CASO DE ESTUDO) comprova que o algoritmo desenvolvido nesta dissertação permite reduzir a complexidade de programas em OutSystems e concluir que os programas desenvolvidos em linguagens *low-code* também podem ser alvo de refatorização.

O capítulo 5 (CASO DE ESTUDO) expõe e analisa o leque de módulos identificados. A avaliação realizada recorreu à análise de aplicações cuja complexidade é semelhante a aplicações desenvolvidas num departamento de TI, podendo assim ser considerados bons exemplos para caso de estudo. Assim, sublinha-se que este estudo foi relevante, pois focou-se num leque de módulos reais, i.e., dotado de ações com uma boa diversidade de estruturação de código.

6.2 TRABALHO FUTURO

A secção 4.4 (Limitações) aponta a utilidade de definir *block-based regions* com recurso a metodologias diferentes da implementada. Assim, pretende-se investigar acerca de metodologias menos convencionais à identificação de nodos líderes para limitar a expansão das *slices*. Por exemplo, podem-se definir regiões que iniciem em nodos diretamente consecutivos a nodos de tipo *query* (e.g., nodos SQL e Aggregate), evitando que esses nodos participem na *slice*.

Além dos valores de *threshold* que o programador pode definir para aumentar a qualidade das *slices* (ver subsecção 4.3.2), pretende-se que o programador possa avaliar qualitativamente as sugestões de refatorização para complementar a avaliação quantitativa, acrescentando um fator subjetivo à análise, que pode ser tomado em consideração em futuras versões do algoritmo.

Como foi demonstrado no capítulo 5 (CASO DE ESTUDO), uma única aplicação de *Extract Action* teve impacto limitado na complexidade global dos módulos (17% $\sim$ 18% das ações deixaram de ser complexas). Apesar disso, a grande maioria das restantes ações demonstraram capacidade de produzir, pelo menos, mais uma *slice*. Isto permite admitir a possi-

bilidade da extração de variáveis de um modo iterativo e incremental. Assim, pretende-se que o algoritmo sugira ao programador múltiplas refatorizações sucessivas para que este retire maior proveito da aplicação de *Extract Action*.

A plataforma OutSystems permite que uma ação tenha múltiplos valores de *output*. Assim, pretende-se que o algoritmo suporte a extração de múltiplas variáveis, em simultâneo, para uma única sub-rotina. Esta opção é relevante em situações em que há um nível de interseção elevado em *slices* de variáveis distintas (ver Figura 36). Assim, nodos outrora indispensáveis (e.g., Figura 36, nodo *For Each* "Iterate OutSystems Version") poderiam ser removidos da ação original. Este nodo *For Each* deve ser duplicado, pois existem dependências de controlo desse nodo para outros nodos no ciclo que não pertencem à *slice*. Além disso, em situações semelhantes à secção A.2, deixa de ser necessária a geração de múltiplas sub-rotinas para remover o cálculo de múltiplas variáveis.

BIBLIOGRAFIA

[All70] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[BC85] Jean-franc Bergeretti and Bernard A. Carry. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7:37–61, 1985.

[BDKZ93] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36(11):81–94, November 1993.

[CCL98] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11–12):595 – 607, 1998.

[Cod] CodeSurfer | GrammaTech. https://www.grammatech.com/products/codesurfer.

[Ett07] Ran Ettinger. Refactoring via program slicing and sliding. In *IEEE International Conference on Software Maintenance, ICSM*, pages 505–506. IEEE, oct 2007.

[FBB+00] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. 2000.

[Fit00] Jerry Fitzpatrick. More c++ gems. chapter Applying the ABC Metric to C, C++, and Java, pages 245–264. Cambridge University Press, New York, NY, USA, 2000.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[Fra] Frama-C. https://frama-c.com/.

[FTSC11] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: Identification and application of extract class refactorings.

In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1037–1039, New York, NY, USA, 2011. ACM.

[GK91] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Trans. Softw. Eng.*, 17(12):1284–1288, December 1991.

[GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, August 1991.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[HD97] Mark Harman and Sebastian Danicic. Amorphous program slicing. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, WPC '97, pages 70–, Washington, DC, USA, 1997. IEEE Computer Society.

[HH01] Mark Harman and Robert Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[HKV07] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A Practical Model for Measuring Maintainability. *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 30–39, 2007.

[HM11] Javier Martín Hernández and Piet Van Mieghem. Classification of graph metrics. 2011.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 23(7):35–46, June 1988.

[JDe10] JDeodorant. https://marketplace.eclipse.org/content/jdeodorant, 2010.

[KH00] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 155–169, New York, NY, USA, 2000. ACM.

[KL88] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, October 1988.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979.

[Mar01] Katsuhisa Maruyama. Automated method-extraction refactoring by using block-based slicing. *SIGSOFT Softw. Eng. Notes*, 26(3):31–40, May 2001.

[MB07] Timothy M. Meyers and David Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Trans. Softw. Eng. Methodol.*, 17(1):2:1–2:27, December 2007.

[McC76] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[Mye77] Glenford J. Myers. An extension to the cyclomatic measure of program complexity. *SIGPLAN Not.*, 12(10):61–64, October 1977.

[MZK13] Ayman Madi, Oussama Kassem Zein, and Seifedine Kadry. On the improvement of cyclomatic complexity metric. *International Journal of Software Engineering and its Applications*, 7(2):67–82, 2013.

[Obj16] Object Management Group (OMG). *Automated Source Code Reliability Measure (ASCRM)*. 2016.

[OT89] Linda M. Ott and Jeffrey J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the 11th International Conference on Software Engineering*, ICSE '89, pages 198–204, New York, NY, USA, 1989. ACM.

[OT93] L.M. Ott and J.J. Thuss. Slice based metrics for estimating cohesion. In *[1993] Proceedings First International Software Metrics Symposium*, pages 71–81. IEEE Comput. Soc. Press, 1993.

[Pro59] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 133–138, New York, NY, USA, 1959. ACM.

[TC09] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Trans. Softw. Eng.*, 35(3):347–367, May 2009.

[TC11] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *J. Syst. Softw.*, 84(10):1757–1782, October 2011.

[Tip95] F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.

[Ton03] Paolo Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering*, 29(6):495–509, jun 2003.

[VRvdL+16] Joost Visser, Sylvan Rigal, Rob van der Leek, Pascal van Eck, and Gijs Wijnholds. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. O'Reilly Media, Inc., 1st edition, 2016.

[WAL] WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[Wis] The Wisconsin Program-Slicing Tool. http://research.cs.wisc.edu/wpis/slicing_tool/.

A

ANEXOS

A.1 MÉTODO STATEMENT

A Figura 35 representa um exemplo de refatorização apresentado por Fowler et al. [FBB+00], que calcula os pontos de um cliente e as suas dívidas numa loja de aluguer de vídeos.

```
public String statement() {
    double totalAmount = 0;
    int renterPoints = 0;
    Enumeration<Rental> rentals = _rentals.elements();
    String result = "Rental Record\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        double thisAmount = each.getCharge();
        if (each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
            renterPoints = renterPoints + 2;
        else
            renterPoints++;
        result = result + each.getMovie().getTitle() + "\t" + String.valueOf(thisAmount) + "\n";
        totalAmount = totalAmount + thisAmount;
    }

    result = result + "Amount: " + String.valueOf(totalAmount) + "\n";
    result = result + "Points: " + String.valueOf(renterPoints);
    return result;
}
```

Figura 35.: Método Statement

A.2 SERVER ACTION NEW_TESTSETTINGS_CONVTOTESTSETTING

A *server action* apresentada pelas figuras abaixo (Figura 36, Figura 37, Figura 38, Figura 39, Figura 40 e Figura 41) é composta por 87 nodos e tem um valor de complexidade ciclomática de 41 unidades. A *server action* é composta por 6 regiões principais: *"OutSystems Version"*, *"OutSystems Stack"*, *"OutSystems Edition"*, *"Application Servers"*, *"Database"* e *"Operating System"*. Dividir a ação em ações mais simples pode ser benéfico, pois reduz a complexidade da ação principal. Por exemplo, a aplicação de *Extract Action* reduz o tamanho e complexidade da ação principal, através da criação de ações mais simples, o que facilita a sua leitura e compreensão.

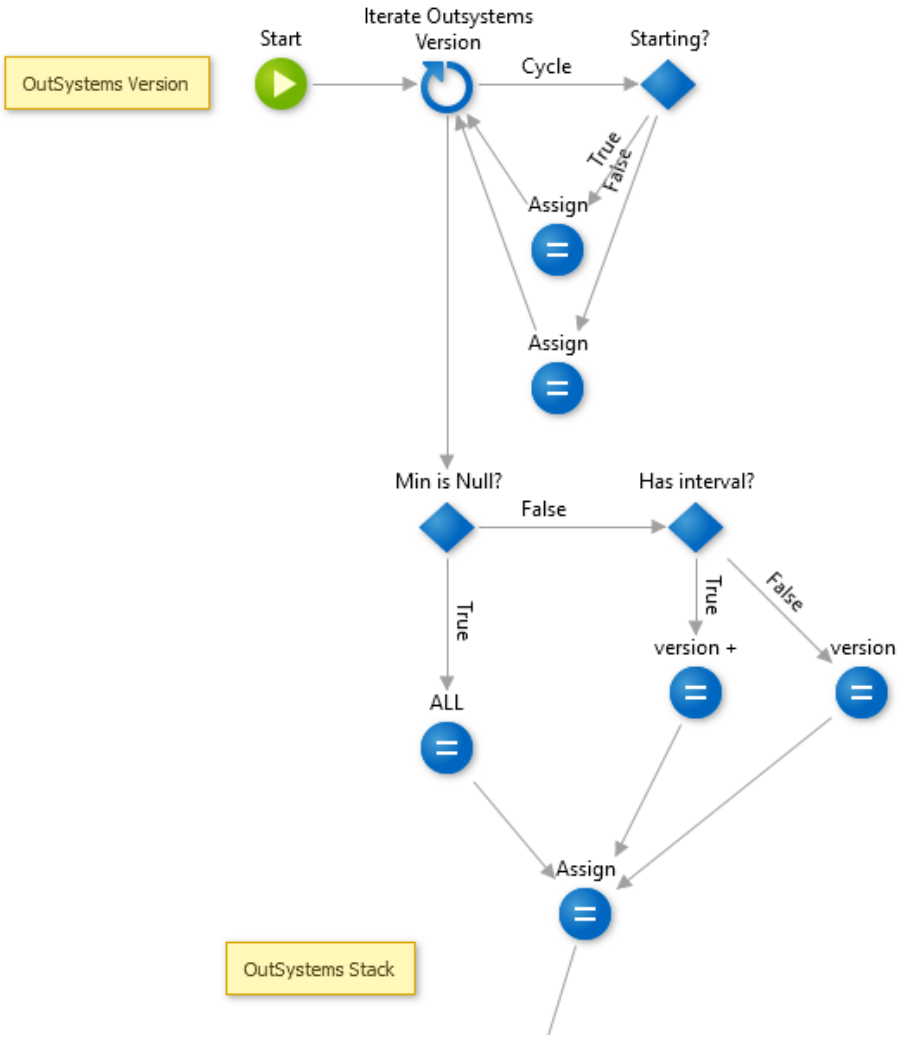


Figura 36.: Server action *New_TestSettings_ConvToTestSetting* (*OutSystems Version*)

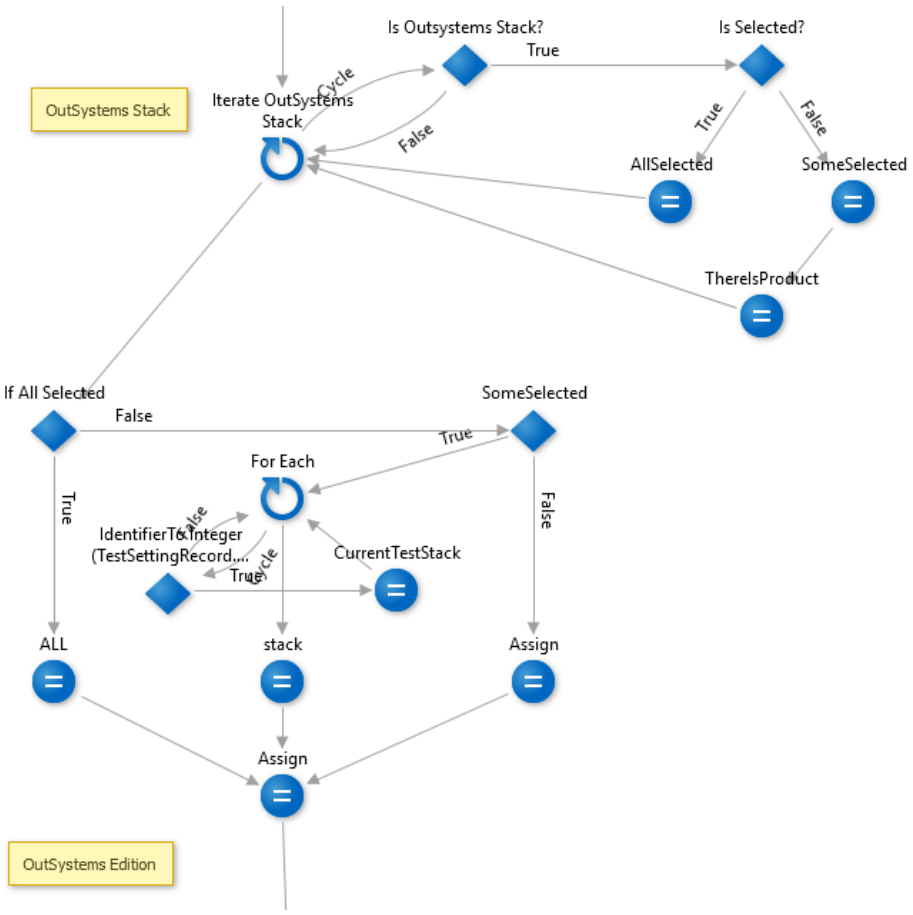


Figura 37.: Server action *New_TestSettings_ConvToTestSetting* (*OutSystems Stack*)

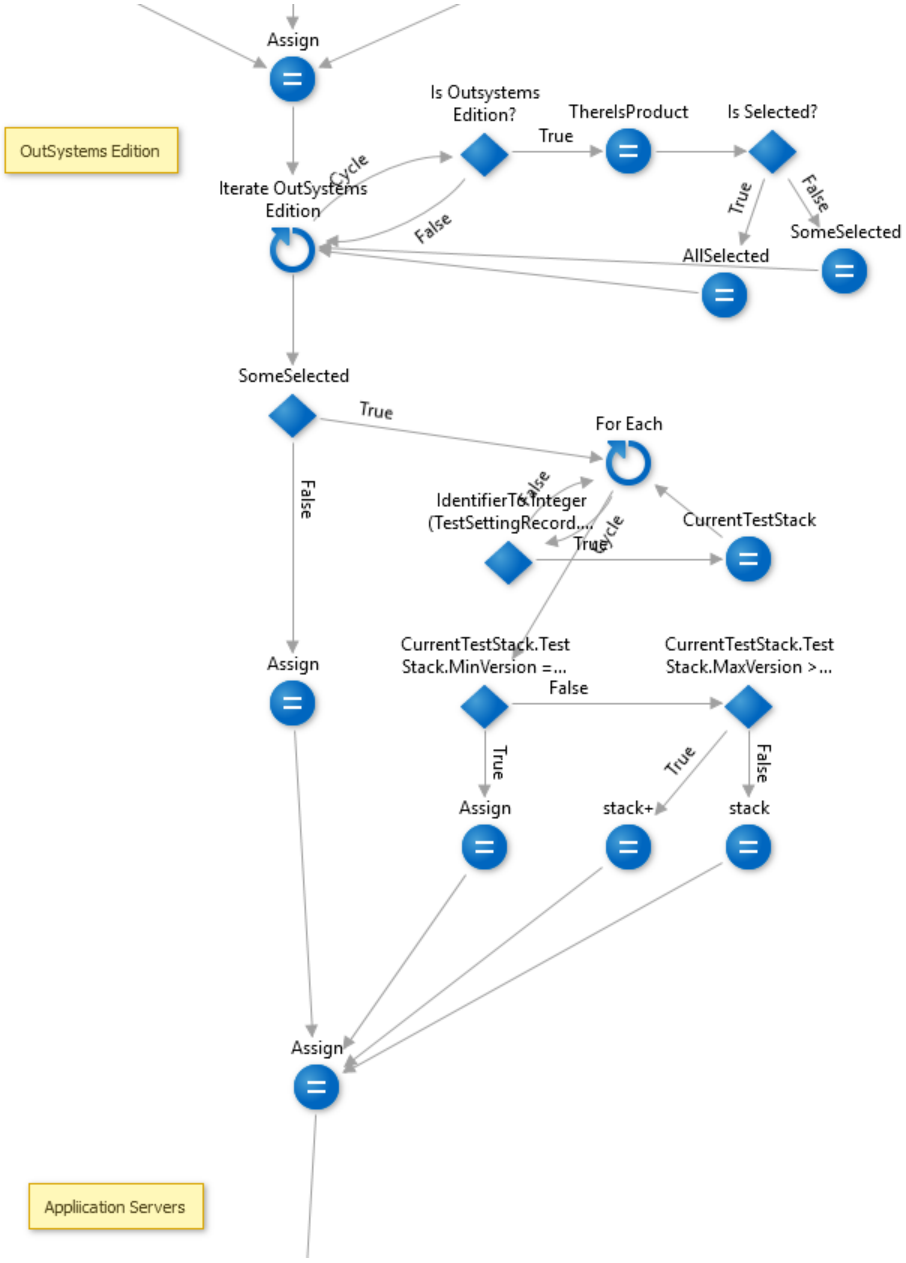


Figura 38.: Server action *New_TestSettings_ConvToTestSetting* (*OutSystems Edition*)

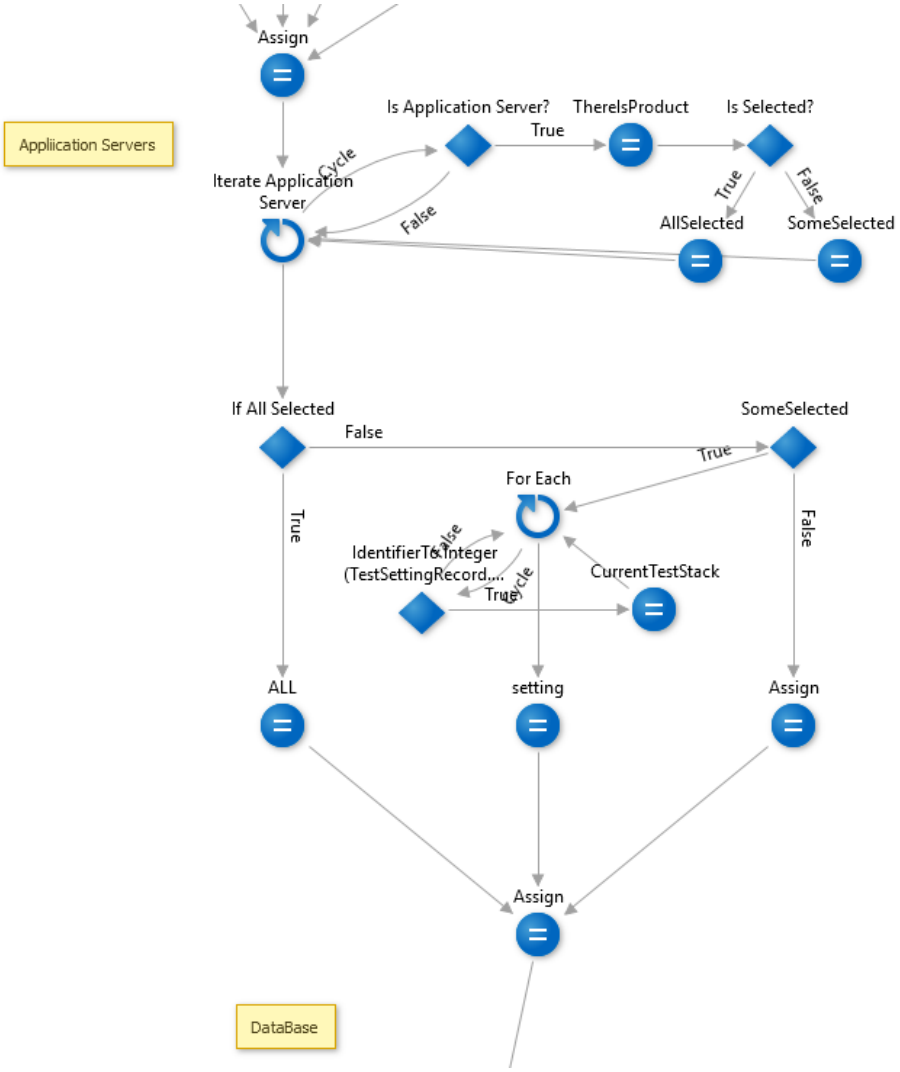


Figura 39.: Server action *New_TestSettings_ConvToTestSetting* (*OutSystems Application Servers*)

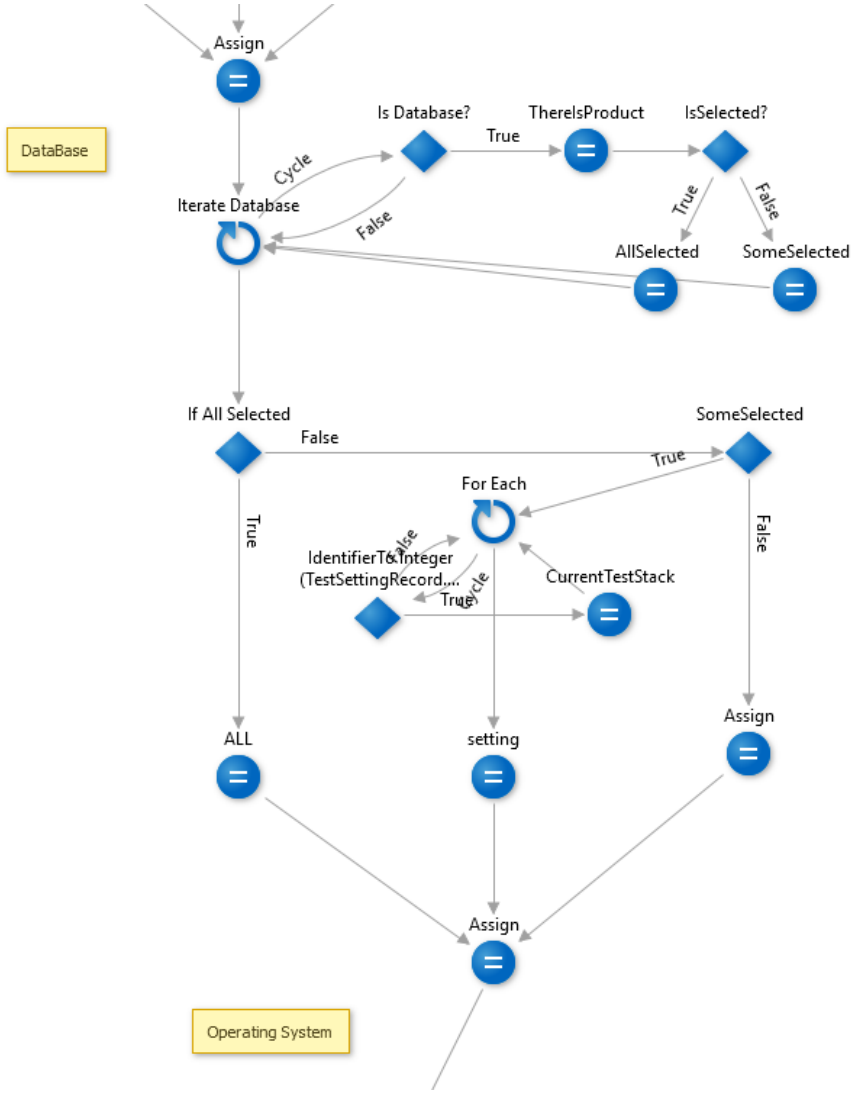


Figura 40.: Server action *New_TestSettings_ConvToTestSetting* (*OutSystems Database*)

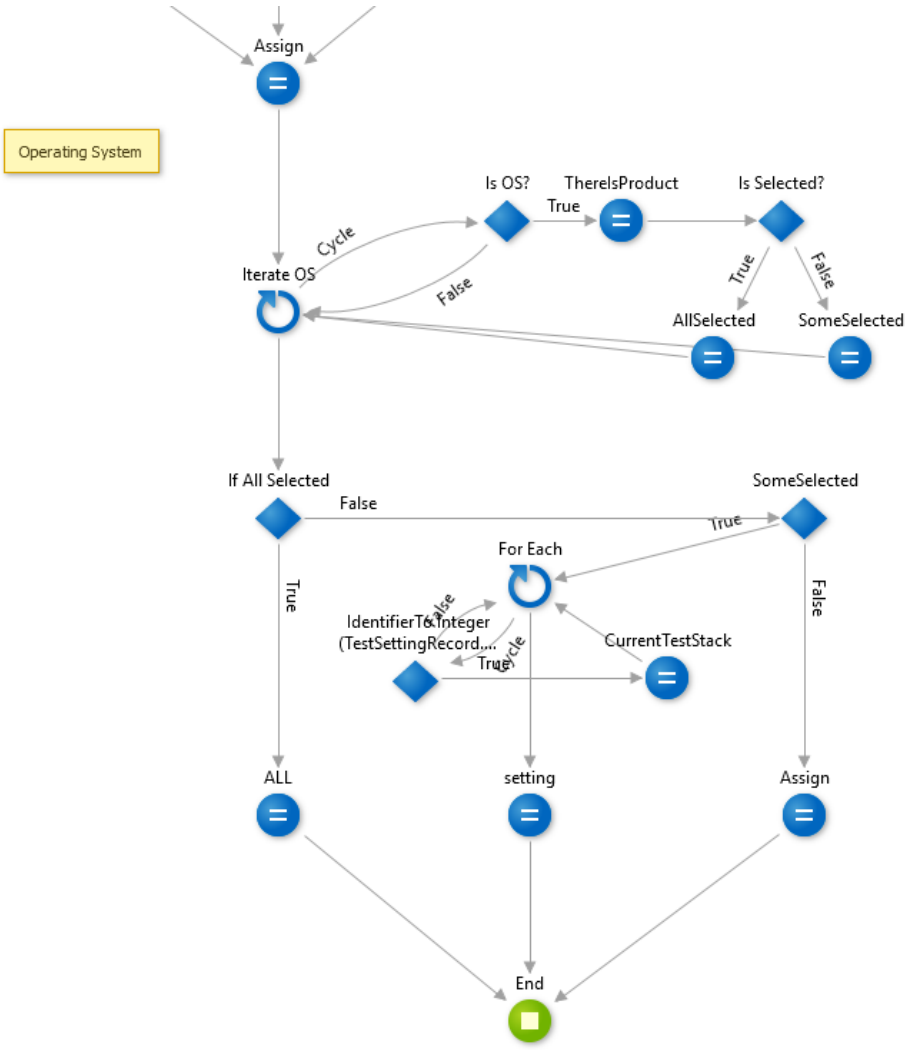


Figura 41.: Server action *New_TestSettings_ConvToTestSetting* (*OutSystems Operating System*)