Um Framework para Recuperação Arquitetural Independente de Plataforma

Luis Paulo Torres de Oliveira e Sandro Santos de Andrade

Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Curso de Tecnologia em Análise e Desenvolvimento de Sistemas
Rua Emídio Santos, s/n - Barbalho - Salvador - Bahia

Resumo

Compreender como um *software* foi desenvolvido e como ele opera é uma tarefa fundamental para a Engenharia de *Software*. Sistemas no mundo todo passam por constantes processos de manutenção e evolução, tarefas que exigem do profissional um bom entendimento da arquitetura do *software* para serem feitas de forma correta e eficiente. Neste contexto, as técnicas de recuperação arquitetural apresentam caminhos rápidos para se ter conhecimento necessário sobre os artefatos arquiteturais que compõem um sistema. Entretanto, diante da variedade de abordagens existentes, um arquiteto de *software* deveria encontrar um meio simples de desenvolver, utilizar e readaptar técnicas de recuperação arquitetural para facilitar e agilizar a sua tarefa, bem como escolher o modo adequado de representar a arquitetura recuperada da mesma forma. Este trabalho apresenta um *framework* de recuperação de multiartefatos arquiteturais, com o qual o arquiteto poderá controlar, desenvolver e configurar todo o processo de recuperação arquitetural da forma que lhe for mais apropriada.

Palavras-chave: Recuperação Arquitetural, Engenharia Reversa, *Application Framework*, Arquitetura de *Software*, Engenharia de *Software*.

1. Introdução

O entendimento da arquitetura de um sistema de *software* é uma tarefa que se torna cada vez mais fundamental em todas as etapas do seu ciclo de vida. Atividades de manutenção e evolução permeiam a maior parte de todo o tempo de vida dos sistemas legados ou de grande porte, tornando-se mais árduas e custosas na ausência de um planejamento pré-definido. Portanto, uma boa compreensão do *software* garante que a qualidade seja mantida durante suas fases de evolução, bem como orienta e facilita todo o trabalho feito sobre ele pela equipe de desenvolvimento [1].

Neste contexto, uma série de iniciativas na academia e na indústria têm como foco o desenvolvimento de métodos flexíveis e sofisticados para recuperação de arquiteturas a partir do conjunto disponível de artefatos de *software* [2].

Além disso, estas informações devem ser representadas de forma compreensível por quaisquer pessoas envolvidas no projeto de um *software*, independente do seu nível de conhecimento técnico. A esta atividade de recuperação de informações pertinentes à arquitetura de um *software*, dá-se o nome de recuperação arquitetural.

Uma abordagem de recuperação arquitetural [3] define os aspectos arquiteturais a serem recuperados e as técnicas a serem utilizadas na obtenção destas informações. Ela também pode ser útil para gerar representações mais precisas de uma arquitetura de *software*.

Esta representação deve manter o máximo de fidelidade com as arquiteturas descritivas do sistema analisado, como a estrutura de código-fonte do sistema, por exemplo, garantindo maior entendimento e suporte para futuras mudanças.

A diversidade destas abordagens de recuperação arquitetural tem crescido de forma substancial, pois uma arquitetura de *software* pode ser recuperada de diversas maneiras. Essa grande variação ocorre porque cada forma de recuperação depende da abordagem utilizada, das características do sistema analisado ou das necessidades dos *stakeholders*. Dentre o conjunto de elementos que são considerados na utilização de uma abordagem estão:

1. Informações estruturais que deseja-se obter, como classes, interfaces, relacionamentos entre classes, uso de bibliotecas, número de invocações, etc.;

2. Informações comportamentais, como os fluxos de dados ou de controle;
3. As formas de representação, que variam desde o uso de linguagens informais, como textos ou gráficos, passando pelas semiformais, como a UML (*Unified Modeling Language*) [4] até as mais formais, como as ADLs (*Architecture Description Languages*) [5];
4. O nível de abstração da representação, podendo ser rico em informações mais específicas sobre o *software* (informações de baixo nível) ou apenas conter informações superficiais e resumidas (informações de alto nível);
5. Os tipos de visões arquiteturais que serão apresentadas a depender das informações que deverão ter maior ênfase na análise (visão de classes, visão de estados, fluxo de dados, camadas, etc.).

A variedade de abordagens usadas na recuperação arquitetural traz aos engenheiros de *software* inúmeras possibilidades de obter as informações necessárias para manutenção e evolução de sistemas. Porém, dentre tantas técnicas e ferramentas de extração de arquitetura desenvolvidas e disponibilizadas pela comunidade, como saber qual delas poderá atender melhor as necessidades de seus usuários? Como garantir que uma determinada abordagem possa gerar os melhores resultados dentre tantas outras alternativas disponíveis?

Para responder estas questões, pesquisadores da área de Engenharia de *Software* têm se empenhado em estudos que analisam as abordagens mais conhecidas e utilizadas na indústria e na comunidade acadêmica. [2] [3]

Dentro de critérios preestabelecidos, seguidos por experimentos e testes de validação, é possível classificar o conjunto de técnicas pelo grau de precisão, conformidade ou corretude das representações geradas pelas mesmas. [1] A princípio, estes estudos comparativos podem ser úteis para orientar os engenheiros de *software* sobre qual pode ser a abordagem que mais atenderá as suas necessidades.

Entretanto, é necessário ter algum cuidado na utilização de referências teóricas para realizar esta escolha. Embora existam estudos voltados para esta questão, não há um consenso geral sobre qual técnica pode ser a mais adequada. Isto ocorre porque é difícil encontrar uma abordagem que seja útil independente do sistema, de quais informações devem ser extraídas e de como elas devem ser representadas.

Então, diante de tantas alternativas para se recuperar a arquitetura de um *software*, qual delas mais poderia atender as demandas dos *stakeholders*? Considerando a variedade de abordagens existentes, sejam elas bem conhecidas nos meios industrial e/ou acadêmico; ou mesmo as mais recentes, pouco conhecidas, mas bem referenciadas, é trivial que esta questão seja levantada.

Como solução, este trabalho propõe um *framework* onde o próprio usuário pode ter um controle maior sobre o processo de recuperação arquitetural. Sendo assim, ele se torna capaz de utilizar e customizar mecanismos existentes para este tipo de tarefa, ou mesmo criar novos mecanismos.

Visando facilitar a implementação de todo o processo, o *framework* oferece um conjunto de *interfaces* que correspondem às diferentes fases da recuperação arquitetural. Além disso, a flexibilidade provida por esta divisão de fases permite que as implementações de cada uma das fases sejam substituídas, estendidas ou reutilizadas em novas combinações demandando pouca ou nenhuma alteração no código já existente.

Assim, através do uso do *framework*, é possível escolher, desenvolver e testar rapidamente muitas abordagens usadas para recuperar informações arquiteturais de um *software*. Estes artefatos podem variar a depender da motivação da recuperação arquitetural, assim como da plataforma de desenvolvimento na qual o sistema alvo foi desenvolvido.

Para lidar com estas variações, o *framework* permite ao usuário adaptar o processo de recuperação com as restrições definidas pelas características da plataforma. Por fim, a forma na qual a arquitetura será representada pode variar a depender do grau e do tipo de informações que são recuperadas, variação esta que, como será visto a seguir, também é suportada pelo *framework*.

Este trabalho esta organizado da seguinte forma: A sessão 2 explica, de forma mais detalhada, o que é a recuperação arquitetural e as principais vantagens alcançadas pela sua utilização em sistemas de *software*. Adicionalmente, a sessão ainda aponta os principais problemas originados da variedade de técnicas de recuperação arquitetural existente. A sessão 3 fornece uma descrição do *framework* proposto para o desenvolvimento de técnicas de recuperação arquitetural independente de plataforma, apontando algumas vantagens por ele oferecidas e suas principais características. A sessão 4 descreve o processo de desenvolvimento de uma técnica de recuperação arquitetural com a utilização do *framework*, fornecendo detalhes dos aspectos de implementação. A sessão 5 demonstra como o *framework* foi avaliado e testado, bem como descreve as principais observações a respeito de sua viabilidade. A sessão 6 apresenta os trabalhos correlatos, explicando o diferencial encontrado na utilização do *framework* para recuperação arquitetural. Por fim, a sessão 7 apresenta as conclusões feitas após o término

deste trabalho, as expectativas e propostas de trabalhos futuros entorno do *framework* desenvolvido.

2. Recuperação Arquitetural

A arquitetura de *software*, por sua definição, é um conjunto formado pelas principais decisões de projeto tomadas durante todo o seu processo de engenharia do *software*. Por isso, pode-se dizer que todo o *software*, por menor que seja, tem a sua própria arquitetura, já que reúne um conjunto de informações a respeito de sua criação e de seu funcionamento.

Cada item ou conceito que seja inserido em um sistema em fase de desenvolvimento pode ser considerado como uma decisão arquitetural, que induz ao *software* uma série de qualidades e restrições que o caracterizam. Para se criar uma boa arquitetura, portanto, o arquiteto de *software* deve ter um conhecimento amplo sobre tudo o que envolverá a criação de um sistema. Este conhecimento abrange desde as pessoas envolvidas no projeto, passando pelos conhecimentos técnicos até as informações de outras tecnologias e os domínios para os quais a aplicação será voltada.

É fundamental que a Engenharia do *Software* seja centrada para a construção de uma boa arquitetura (Engenharia Orientada a Arquitetura), independente da fase em que se encontre. Esta decisão garante base adequada para realizar mudanças, manutenções, evoluções e até mesmo garantir a reutilização de artefatos em outros sistema. Além disso, garante aos envolvidos no projeto um maior controle sobre as informações técnicas e estruturais do *software*, assim como a facilidade de gerenciamento e na comunicação sobre o projeto. [6]

Com o decorrer do tempo, as práticas de Engenharia de *Software* centradas na arquitetura se tornaram cada vez mais frequentes no desenvolvimento de sistemas de grande porte. *Softwares* criados com o intuito de terem ciclos de vida longos e de permanecerem em operação devem contar com as vantagens deste processo. Entretanto, nem todos os sistemas existentes puderam contar com estes benefícios, sendo mantidos e evoluídos sem um planejamento adequado.

A consequência disso é a aparição de uma série de desvios que degradam toda a sua estrutura em relação àquela planejada na fase de projeto. Estas inconsistências são conhecidas como erosões arquiteturais [1], e se forem numerosas, são responsáveis por dificultar consequentes processos de alteração no *software*, comprometendo o seu ciclo de vida.

Para que este problema não ocorra, sempre será necessário compreender a arquitetura do sistema por meio da atualização de sua documentação. Mas o que acontece com muitos sistemas legados existentes, é que existe pouca ou nenhuma fonte de informação que possibilite a sua compreensão plena, o que os torna muito mais complicados de manter e evoluir. Para resolver este problema, uma engenharia reversa no sistema é feita para recuperar assim todas as informações e decisões arquiteturais possíveis do *software*.

Estes artefatos, por sua vez, devem ser organizados em representações que sejam compreensíveis aos membros do projeto, garantindo assim uma maior integridade conceitual do sistema. Este processo é conhecido como recuperação arquitetural [3], e sua principal finalidade é mapear as decisões arquiteturais de um *software* da forma mais fiel possível. Desta forma, os *stakeholders* (pessoas envolvidas) adquirem uma base sólida para orientá-los em tarefas futuras de manutenção e evolução do sistema.

Existe uma variedade de abordagens que são utilizadas para realizar a recuperação da arquitetura do *software*. Cada uma com mecanismo peculiar para recuperar conjuntos de informações específicas, bem como representá-las em diferentes formas e níveis de abstração.

As abordagens mais simples, por exemplo, utilizam técnicas de reflexão para recuperar a arquitetura através do código-fonte ou de sua documentação, criando diagramas estruturais para demonstrar as classes e seus inter-relacionamentos. Outras informações da estrutura do sistema, como interfaces, heranças, composições, usos de bibliotecas e de COTS (*Commercial Off-The-Shelf*) também podem ser captadas a depender dos algoritmos usados.

Abordagens mais complexas, por sua vez, garantem informações mais detalhadas sobre o comportamento do sistema, além de adquirir detalhes sobre padrões e estilos arquiteturais que o compõem. Informações mais detalhadas e implícitas no *software*, como comportamento de processos, padrões de projeto, requisitos funcionais e não-funcionais, são obtidas por meio das abordagens baseadas em padrões (*Graph Pattern-Matching*) [7, 8].

Em contrapartida, as informações mais triviais, como métricas, classes e seus inter-relacionamentos, são recuperadas por abordagens mais simples, baseadas em Clusterização Hierárquica [2]. Um amplo conjunto de diferentes métricas para avaliação de similaridade está atualmente disponível na literatura [2]. Na sessão a seguir, será explicado como o *framework* proposto neste trabalho auxiliou a utilização destas abordagens.

Um dos grandes problemas na área de recuperação arquitetural é a inexistência de teorias e *"first-principles"* que sedimentem uma avaliação comparativa mais precisa entre métodos de recuperação arquitetural. Traba-

lhos recentes [9, 10] tentam minimizar este problema através da definição de métodos parcialmente automatizados de recuperação.

3. Um Framework para Recuperação Multiartefato de Arquiteturas de Software

O *framework* proposto por este trabalho é um componente da ferramenta DuSE-MT [1] [11] – *software* livre desenvolvido em Qt que oferece um ambiente para definição, criação, manipulação e representação de modelos arquiteturais. Seus mecanismos voltados para a análise arquitetural fazem uso de linguagens de modelagem existentes, como a UML (*Unified Modeling Language*) ou a MOF (*Meta-Object Facility*).

A ferramenta ainda possui uma linguagem específica, a DuSE, e faz uso da linguagem de programação JavaScript para a montagem e gerenciamento de diagramas. Além disso, sua arquitetura baseada em *plugins* permite que as funcionalidades sejam estendidas para a realização de qualquer outra tarefa de manipulação, recuperação e projeto arquitetural.

Como um *plugin* integrante do DuSE-MT, o *framework* proposto por este trabalho oferece um mecanismo para recuperar múltiplos artefatos arquiteturais de *software* independente de qualquer plataforma de desenvolvimento. Junto ao processo de recuperação, o *plugin* também provê meios para variar a representação destas informações em diversos modelos e níveis de abstração.

Suas três principais funcionalidades são definidas por interfaces que devem ser implementadas, a fim de fornecer instâncias com os mecanismos para cada a tarefa integrante da recuperação arquitetural. O diagrama da Figura 1 apresenta as quatro interfaces, que serão detalhadas abaixo.

Conforme apresentado na Figura 1, o *framework* define uma *engine* genérica para recuperação arquitetural através da definição de quatro *interfaces* que viabilizam a implementação dos *hot-spots* indicados. Tais *interfaces* permitem a instanciação do *framework* de modo a recuperar arquiteturas de *softwares* desenvolvidos em uma linguagem de programação específica (ex: C++), utilizando um algoritmo particular de recuperação (ex: ACDC [12] ou ARC [13]) e descrevendo as arquiteturas recuperadas através de uma notação de modelagem específica (ex: UML, ADLs ou gráficos informais [5]).

`IArchitectureRecoveryBackend` é a interface responsável por prover mecanismos para recuperar as arquiteturas em diferentes ambientes de desenvolvimento. Linguagens de programação, como C++, Java e PHP por exemplo, costumam ser diferenciadas pelas suas sintaxes, estruturas, bibliotecas, *framework*s de desenvolvimento utilizados e outros fatores. Devido a este fato, o processo de recuperação arquitetural deve ser capaz de se adaptar para capturar as informações dos *software*s implementados em qualquer uma destas linguagens.

Além disso, para cada plataforma citada, existem diversas formas de se recuperar suas informações, diferenciando-se pela fonte (código-fonte, documentação, etc.) ou pelo uso de alguma ferramenta ou *script*. Portanto, esta interface é fundamental para que as recuperações arquiteturais sejam feitas independente das restrições de plataforma ou do uso de alguma ferramenta automatizada.

`IArchitectureRecoveryAlgorithm` provê os algoritmos de recuperação arquitetural, diferenciados pelas fontes de informações utilizadas e como elas são processadas para que os artefatos arquiteturais sejam recuperados. Como dito anteriormente, os métodos de recuperação arquitetural podem ser baseados em clusterização ou em detecção de padrões, podendo assim utilizar conjuntos de informações distintas em determinado grau.

Dentre estas informações utilizadas pelos algoritmos, estão as estruturas, comportamentos, estados, métricas, fluxos de controles e de dados das entidades do sistema. Isso significa dizer que, para cada tipo de informação extraída de um *software*, podem haver diferentes formas de processamento para gerar formas variáveis de resultados. Ao prover implementações de diferentes algoritmos de recuperação arquitetural, esta interface garante que diferentes demandas de tipos de informações arquiteturais sejam atendidas de forma flexível.

Conectores de *software* [14] desempenham um papel crucial no projeto e análise de arquiteturas. Eles mediam a interação entre componentes e são fundamentais para a satisfação de certos requisitos não-funcionais. Dentre os conectores mais utilizados, destacam-se o *Procedure Call* – caracterizado por uma comunicação acoplada, síncrona e binária entre componentes – e o *Event* – geralmente utilizado para comunicação desacoplada, assíncrona e do tipo muitos-para-muitos.

`IConnectorCodeSnippet` permite a definição de expressões regulares que analisam o código-fonte em busca de trechos que descrevem a utilização de um conector específico. O objetivo é flexibilizar a detecção de conectores não previstos e viabilizar a sua representação como elementos arquiteturais de primeira classe

`IArchitectureRecoveryNotation` é a interface

[1] http://duse.sf.net

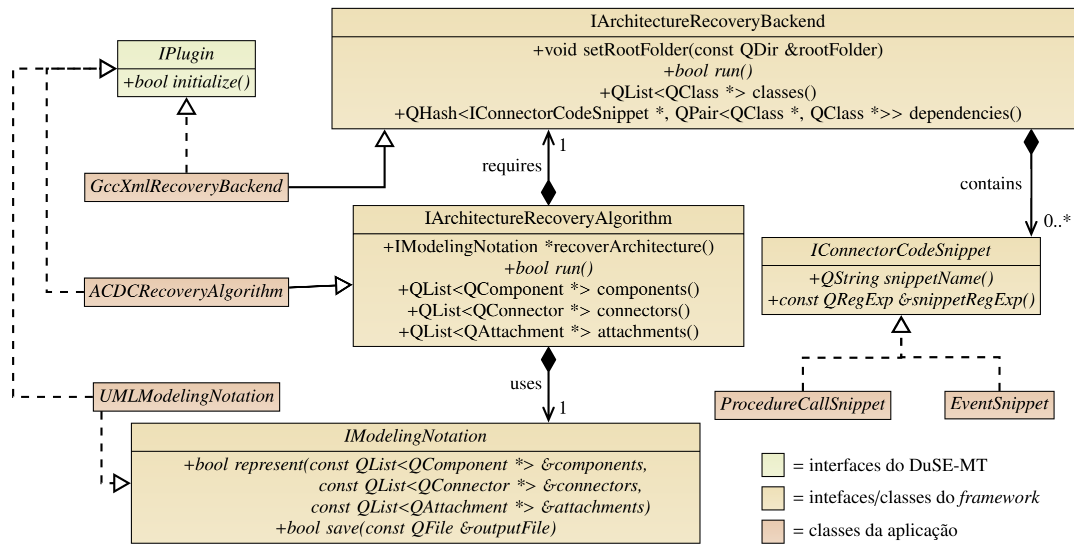

Figura 1: Interfaces do DuSE-MT, interfaces/classes definidas pelo *framework* e exemplo de classes implementando os *hot-spots* requeridos.

que provê diversas formas de representação dos artefatos arquiteturais recuperados. As notações utilizadas variam desde as mais informais, como os textos e gráficos, até as notações mais formais e detalhadas, como as ADL (*Architecture Description Language*). Notações informais permitem melhores descrições dos aspectos arquiteturais mais importantes, ao passo que as mais formais fornecem detalhes mais técnicos e completos de uma arquitetura.

Notações padronizadas, como a UML e as ADL, podem ser suportadas em outros *software*s para realizar análises e representações arquiteturais mais específicas. Outra alternativa bastante conveniente é o uso de tabelas de métricas para uma representação. Seja como for, esta interface deve ser capaz de variar facilmente entre quaisquer uma destas formas de descrever arquiteturas.

Todas as implementações concretas de *hot-spots* foram desenvolvidas como *plugins* do DuSE-MT e, portanto, implementam a interface `IPlugin` definida pela ferramenta. Este aspecto é importante para permitir a fácil instalação de novos algoritmos, *backends* e notações, bem como a configuração *on-the-fly* das implementações a serem utilizadas em uma determinada recuperação.

4. Instanciando o Framework

Para criar a primeira forma concreta de uso para este *framework*, foi feita uma implementação específica para cada uma das três interfaces. O objetivo foi prover uma primeira abordagem completa de recuperação arquitetural para sistemas escritos na linguagem C++, e então submetê-la a testes e validação.

O *backend* utilizado foi a ferramenta GCC-XML [15], que armazena todas as informações de cada arquivo de cabeçalho C++ em arquivos XML separados. A conveniência de se ter todas as informações em XML (*eXtensible Markup Language*) se dá ao fato de que a biblioteca do Qt provê meios práticos para a sua manipulação.

O Qt - que não foi utilizado somente para implementar este *framework*, mas também para toda a ferramenta DuSE - já possui uma API (*Application Programming Interface*) voltada para esta tarefa. Com isso, a implementação e uso do *backend* tornam-se mais práticos, de modo que todas as informações recuperadas possam ser armazenadas em objetos `QObject`, e então usadas em algum dos algoritmos de recuperação.

O algoritmo escolhido para o trabalho foi o ACDC (*Algorithm for Comprehension-Driven Clustering*) [12]. Esta abordagem utiliza a clusterização para gerar informações do sistema através da análise das interdependências entre as suas classes.

O objetivo desta análise é identificar no *software* a presença de subsistemas (*clusters*) a partir das classes que possuem muitas dependências em relação às outras. Para realizar esta tarefa, o ACDC considera todo o sistema como um grafo, onde as classes são tratadas como os nós do grafo e as dependências são tratadas como arestas que interligam os nós.

Através deste mapeamento, é possível que o al-

goritmo encontre regiões no grafo que correspondem aos *clusters* do sistema que está sendo analisado. A identificação destes subsistemas é feita através da busca pelas classes que possuem mais relações de dependências, sendo estas consideradas os nós dominadores de cada subgrafo encontrado. Naturalmente, os subsistemas recebem o mesmo nome de seus respectivos nós dominadores para a representação.

O principal motivo para a escolha do ACDC para instanciar o *framework* foi devido a sua forma simplificada e eficiente de realizar a recuperação arquitetural com representações de alto nível de abstração. Embora existam formas mais complexas e que geram representações mais detalhas do *software* analisado, seus resultados tiveram as melhores taxas de precisão em comparação com os demais algoritmos utilizados atualmente. [3] Além disso, o seu código está disponível gratuitamente para estudo e reutilização, o que permitiu a sua aplicação para os fins deste trabalho.

Para a recuperação de conectores, foram escolhidas as estratégias de *code snippets* para detectar a presença de conectores do tipo *Procedure Call* e *Event* [14] pelo fato de serem mais frequentes em aplicações desenvolvidas em C++/Qt. Quanto a forma de representar a arquitetura do *software*, foi criado um diagrama de componentes através do uso da UML.

Uma representação em UML pode não ser tão formal quanto a de uma ADL, mas ainda assim é muito útil devido a sua simplicidade. Além disso, a notação pode ser suportada em outras análises automatizadas, e é bastante compreensível.

Na Figura 2, é possível visualizar o *workflow* genérico de todo o processo de recuperação arquitetural que foi proposto neste trabalho. Basicamente, os elementos que constituem as informações de entrada e saída (inseridas nos blocos retangulares) são processados entre as três etapas do processo (inseridas nos blocos elípticos).

Ainda na figura, a numeração presente constitui na ordem de execução das atividades onde: 1) O *backend* de recuperação extrai as informações do *software* de seus artefatos (*codebase*); 2) O algoritmo de recuperação arquitetural processa as informações do *software* para encontrar as informações arquiteturais deste; 3) O módulo de representação arquitetural gera o modelo arquitetural a partir das informações arquiteturais encontradas.

4.1. Processo inicial de integração

Antes da implementação do processo de recuperação arquitetural, foi desenvolvido também dois mecanismo

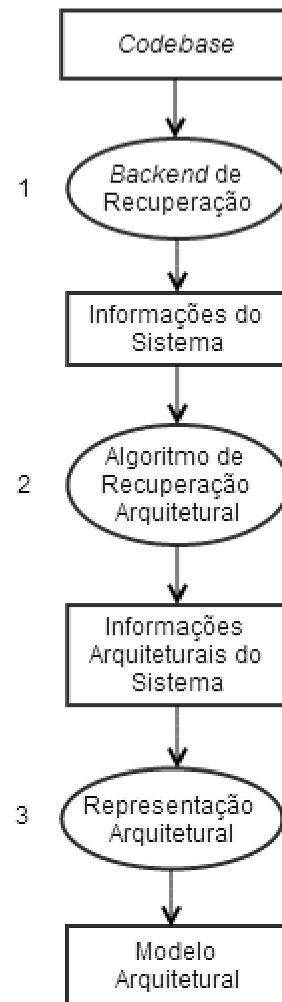

Figura 2: Fluxo de atividades genérico definido pelo processo de recuperação arquitetural apresentado no *framework*. Os elementos numerados e presentes nas elipses constituem nas etapas do processo, que utiliza ou gera as informações rotuladas em cada bloco retangular da figura.

intermediários para auxiliar o usuário no momento da escolha e execução de processos já criados e disponíveis. Tais mecanismos possuem papéis distintos, mas trabalham em conjunto para simplificar a configuração da recuperação arquitetural como um todo.

Usufruindo das vantagens oferecidas pela arquitetura baseada em *plugins* do DuSE-MT, ambos foram tratados como *plugins* diferentes. Quanto aos seus papéis, enquanto o primeiro é responsável pela gerência dos demais *plugins* das etapas de recuperação arquitetural, o segundo se encarrega da representação gráfica utilizada para a escolha do processo.

No momento em que é executado, o DuSE-MT passa por um processo de inicialização onde todos os *plugins* do são instanciados e armazenados na memória da máquina. Assim, os *plugins* que foram desenvolvidos para o *framework* também são inicializados, já que o *framework* funciona a partir da própria ferramenta.

Considerando que a quantidade de *plugins* do *framework* relacionados às etapas dos processos de recuperação arquitetural implementados irá crescer com o passar do tempo, surgiu a necessidade de criar um meio eficaz para armazená-los e recuperá-los com facilidade. Para suprir esta necessidade, foi reutilizada a mesma solução desenvolvida para o armazenamento e recuperação de todos os componentes gráficos e *plugins* pertencentes ao DuSE-MT.

A solução constitui na utilização do padrão de projeto *Singleton* no qual é utilizada uma instância única de uma classe que fornece um ponto de acesso à todas as instâncias de *plugins* e componentes gráficos. Este padrão de projeto foi utilizado pelo DuSE-MT para implementar o *ICore*, uma classe que armazena em seu estado as listas de componentes gráficos e de *plugins* da ferramenta. Estas duas listas são recuperadas através dos métodos estáticos assessores *uiController()* e *pluginController()*, respectivamente.

Deste modo, um objeto cliente pode solicitar operações de qualquer uma das instâncias armazenadas pelo *Singleton* sem a necessidade de criar muitas instâncias em seu código - aumentando o desempenho da aplicação - além de usufruir da flexibilidade que é provida por meio do acesso dinâmico das instâncias armazenadas. Portanto, é possível concluir que este mecanismo representa o núcleo (*Core*) da aplicação DuSE-MT.

Para atender as necessidades do *framework*, foi criada a classe `ArchitectureRecoveryCore`, que pode ser considerado como o núcleo do *framework* e uma parte integrante do DuSE-MT. Esta classe também foi feita aos moldes do padrão de projeto *Singleton*, sendo o ponto único de armazenamento e acesso dos *plugins* de *backend* de recuperação, de algoritmo de recuperação, de notação de modelagem e de *code snippets* para a recuperação de conectores.

Como já foi explicado, cada um dos *plugins* do *framework* são implementações de uma destas quatro *interfaces*. Portanto, `ArchitectureRecoveryCore` tem seu estado constituído por quatro listas que responsáveis pela separação e armazenamento de suas instâncias. Além disso, a classe contém os métodos assessores, responsáveis pela recuperação de cada uma das listas contendo os *plugins* já implementados e disponíveis.

O segundo mecanismo criado para auxiliar o usuário na escolha e execução de um processo de recuperação arquitetural exibe uma *interface* gráfica de configuração. Definida pela classe `ArchitectureRecoveryPanel`, a *interface* é composta por três componentes *combo box* que possibilitam a especificação das etapas do processo de recuperação arquitetural. Para carregar cada uma das *combo boxes* exibidas, a instância de `ArchitectureRecoveryPanel` solicita a `ArchitectureRecoveryCore` o acesso às listas das implementações existentes para três das etapas do processo.

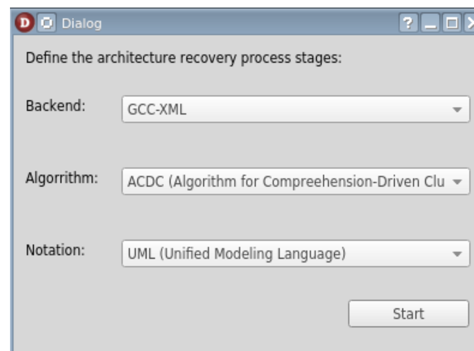

Figura 3: Painel de configuração do processo de arquitetura de software.

Na Figura 3, é possível visualizar o painel criado com a finalidade de auxiliar a configuração e execução do processo de recuperação arquitetural. Através dos componentes de *combo box*, onde estão listados os conjuntos de *plugins* disponíveis para cada uma das etapas do processo, o usuário torna-se apto a configurar o modo como a recuperação arquitetural será feita.

Tendo escolhido o *backend* de recuperação (*label Backend*), o algoritmo de recuperação (*label Algorithm*) e a notação de modelagem utilizada (*label Notation*), o botão *Start* inicialmente desabilitado torna-se habilitado para que seja pressionado. Uma vez pressionado, uma chamada de evento é emitida para dar início ao processo de recuperação.

4.2. O backend de recuperação baseado no GCC-XML

O GCC-XML [15] é uma extensão do compilador g++ que produz uma descrição XML de um programa C++/Qt a partir das representações utilizadas internamente pelo compilador. Tais representações em XML são úteis, pois são fáceis de serem lidas e manipuladas, se comparadas ao desenvolvimento completo de *parsers* para a linguagem C++.

Conforme apresentado na Figura 1, a classe de *plugin* de recuperação de *backend* utiliza o GCC-XML

para exportar representações XML de todos os arquivos de código-fonte encontrados a partir do diretório *_rootFolder*. Os mecanismos para manipulação de XML, presentes no *toolkit* Qt, são então utilizados para a obtenção dos dados das classes que compõem o sistema.

Na implementação do *hot-spot* (Figura 4), foram criados diferentes *containers* para armazenar os conjuntos de informações representadas pelas *tags* dos arquivos XML gerados e que são necessárias para identificar todas as dependências presentes no sistema. No trecho de código 1, é possível visualizar uma versão simplificada de um trecho dos arquivo XML que são gerados pela ferramenta.

```
1  <Class id="_1" name="Employee" bases="" />
2  <Class id="_2" name="Computer" bases="" />
3  <Class id="_3" name="Notebook" bases="" />
4  <Class id="_4" name="Programmer" >
5     <Base type="_1" access="public" />
6  </Class>
7  <Method id="_5" name="setComputer" context="_1
      " >
8     <Argument name="comp" type="_8" />
9  </Method>
10 <Field id="_6" name="computer" type="_8"
      context="_1" />
11 <Field id="_7" name="notebook" type="_3"
      context="_2" />
12 <PointerType id="_8" type="_3" />
```

Listing 1: Demonstração do trecho de um arquivo gerado pelo GCC-XML.

Primeiro, a instância de recuperação de *backend* (`GccXmlArchitectureRecoveryBackendPlugin`) solicita que o usuário selecione o diretório raiz do código-fonte do projeto por meio de uma *interface* de navegação de diretórios. Este diretório será armazenado no campo *_rootFolder* da classe por meio da operação *setRootFolder()*. Quando este valor é carregado, a execução do *backend* de recuperação torna-se possível, ocorrendo quase que por inteiro dentro do método *run()*.

Dentro deste método, a operação *runGccXml()* é a primeira a ser chamada. Ela será responsável por utilizar a uma instância da classe `XmlFileManager` para executar o comando GCC-XML no diretório *_rootFolder* e em todos os seus sub-diretórios por meio da operação *generateXmlFile()*.

Neste ponto, vale ressaltar que assim como o compilador g++, o GCC-XML também realiza uma verificação de sintaxe no código da aplicação antes de armazenar as suas informações em arquivos XML. Portanto, caso haja algum erro de compilação do sistema, os arquivos XML não serão gerados. A depender da quantidade de arquivos de cabeçalho do sistema, o GCC-XML pode apresentar um tempo de resposta alto ou baixo, mas sempre superior ao tempo de compilação do g++, já que o tempo de criação dos arquivos XML também passa a ser considerado.

Quando um arquivo XML é criado, ele não somente inclui as informações da classe que foi implementada no arquivo de código que analisou. São incluídas também todas as informações das classes que foram incluídas no código por meio da diretiva de pré-compilação `#include`. Com isso, o tamanho dos arquivos torna-se maior na medida em que novas classes são inseridas. Diante disso, um estudo foi feito para tentar corrigir este problema, mas foi descoberto que esta é a forma esperada e permanente de se gerar os arquivos XML.

Caso a execução do GCC-XML tenha sido realizada com sucesso, o *plugin* cria uma lista contendo todos caminhos completos de todos os arquivos XML que foram gerados. Esta lista então é utilizada no método *recoverDependencyRelations()* para que a instância se recorde de todos os arquivos cujos dados devem ser coletados.

Para cada arquivo XML examinado, é criada uma instância temporária da classe `XmlFileReader`. É nesta classe que estão todos os algoritmos que irão extrair os diferentes conjuntos de informações presentes nos arquivos. Estas informações, por sua vez, estão armazenadas em atributos das *tags* que representam entidades de código C++.

Para cada uma das *tags* analisadas, são criados *containers* semelhantes àqueles contidos na classe do *plugin*. Eles são responsáveis por prover acesso instantâneo aos valores que serão essenciais para a descoberta das relações de dependência entre as classes.

As entidades de código que são definidas pelas *tags* (Trecho 1) são as seguintes: *i)* as *tags Class* definem as classes que fazem parte do código cujas informações foram extraídas; *ii)* as *tags Base*, definem as superclasses de uma classe, por isso elas aparecem aninhadas nas *tags Class* das subclasses encontradas no sistema; *iii)* as *tags Method* definem os métodos utilizados pelas classes do sistema; *iv)* as *tags Argument*, aninhadas nas *tags Method*, definem os argumentos destes métodos; *v)* as *tags Field* definem os campos das classes; *vi)* e as *tags PointerType* definem os ponteiros que são utilizados no código.

A instância da classe `XmlFileReader` realiza a

extração destas informações através do método *fillTagContainers()*, utilizando as operações *openXmlFile()* e *closeXmlFile()* da classe `XmlFileManager` para abrir e fechar os arquivos XML.

Durante a execução do método, todos os *containers* - semelhantes àqueles contidos na classe `GccXmlArchitectureRecoveryBackendPlugin` - são preenchidos realizando-se uma única leitura para cada arquivo XML examinado.

A leitura é feita através de uma instância da classe `QXmlStreamReader`, provida pela biblioteca do Qt. Através de métodos assessores implementados em `XmlFileReader`, o *plugin* é capaz de acessar todos os *containers* que precisa e carregar os seus próprios *containers*.

Feito isso, a instância do *plugin* invoca o método *recoverDependencyRelations()*, onde ocorre o processo de busca pelas interdependências entre as classes do sistema analisado. Estas interdependências são encontradas após a realização das seguintes etapas:

Identificação das Classes: da *tag Class*, são recuperados os nomes das classes utilizadas no sistema junto aos seus respectivos valores identificadores (`id`), que são utilizados para relacionar as *tags* de classe com todas as demais *tags*. Na implementação, esta tarefa é feita pelo método `fillTagContainers()`, que armazena os dados de cada *tag* no *container _fileClasses*.

Identificação de Dependências por Herança: um dos casos de dependência entre classes ocorre quando uma classe herda atributos e operações de outras classes. Para descobrir se uma classe é derivada de outras classes, é necessário verificar se a sua *tag Class* possui *tags Base* aninhadas, conforme ocorre com a classe `Programmer` do trecho 1.

Identificando a existência de superclasses de uma classe por meio destas *tags*, a aplicação compara cada o valor do atributo `type` com os valores de `id` de outras *tags Class*, conseguindo encontrar as superclasses correspondentes quando estes valores são iguais. Na implementação, esta tarefa é feita pelo método *baseAdjacencies()*.

Identificação de Dependências por Agregação ou Composição: outra forma de definir uma relação de dependência é verificar se a classe agrega ou é composta por objetos de outras classes. Para identificar se uma classe possui algum campo, basta verificar se o valor do atributo `id` de sua *tag Class* é igual ao valor do atributo `context` de uma ou mais *tags Field* presentes no arquivo. Para identificar quais classes fazem parte deste conjunto de campos, por sua vez, deve-se levar em conta duas abordagens diferentes.

Na linguagem de programação C++, a ocorrência de agregação e composição podem aparecer de duas formas: Por definição de referências ou por definição de ponteiros como atributos de classe. No caso da identificação dos tipos das referências, é feita apenas uma comparação direta entre os valores de `type` nas *tags Field* com os valores de `id` das *tags Class* presentes no arquivo. No caso da identificação dos tipos de ponteiros, os valores de `type` devem ser comparados primeiramente com os valores de `id` das *tags PointerType*.

Finalmente, a aplicação compara os valores de `type` destas *tags* com os valores de `id` das *tags Class* do arquivo, encontrando assim os tipos de ponteiro definidos como atributos de uma classe. Na Figura **??** é possível verificar que o campo `computer` se trata de um ponteiro, enquanto o campo `notebook` é identificado como uma referência. Na implementação, esta tarefa é feita pelo método *fieldAdjacencies()*.

Identificação de Dependências por Parâmetros de Métodos: o último caso de dependência considerado pela aplicação criada para o *framework* é definido quando uma classe define objetos de outras classes como parâmetros em alguma de suas operações. Conforme o exemplo do trecho 1, os métodos de classe são definidos pela *tag Method*, enquanto os atributos destes métodos estão definidos em *tags Argument* aninhadas nas primeiras.

Através do atributo `context`, é possível identificar a classe que define este método, bastando comparar o seu valor com o valor dos atributos `id` das *tags Class* do arquivo. Quanto aos tipos de parâmetros aceitos pelo método, é necessário realizar o mesmo processo feito para identificar os tipos de campo definidos em uma classe, já que eles podem ser definidos tanto como referências quanto como ponteiros. Na implementação, esta tarefa é feita pelo método *argumentsAdjacencies()*.

Além dos *containers* de `QMap` criados para armazenar o conjuntos de valores dos atributos presentes em cada uma das *tags* dos arquivos XML, um *container QMultiMap* foi definido como atributo da classe do *plugin* para ser preenchido com todas as relações de dependência descobertas durante a análise.

Este tipo de tabela define uma relação de um-para-muitos entre as chaves e seus valores, considerando que uma única classe pode ter mais de uma dependência. Por fim, o `QMultiMap` preenchido com todas as informações de interdependência entre as classes esta pronto para ser recuperado pelo seu método assessor *getDependencyRelations()*.

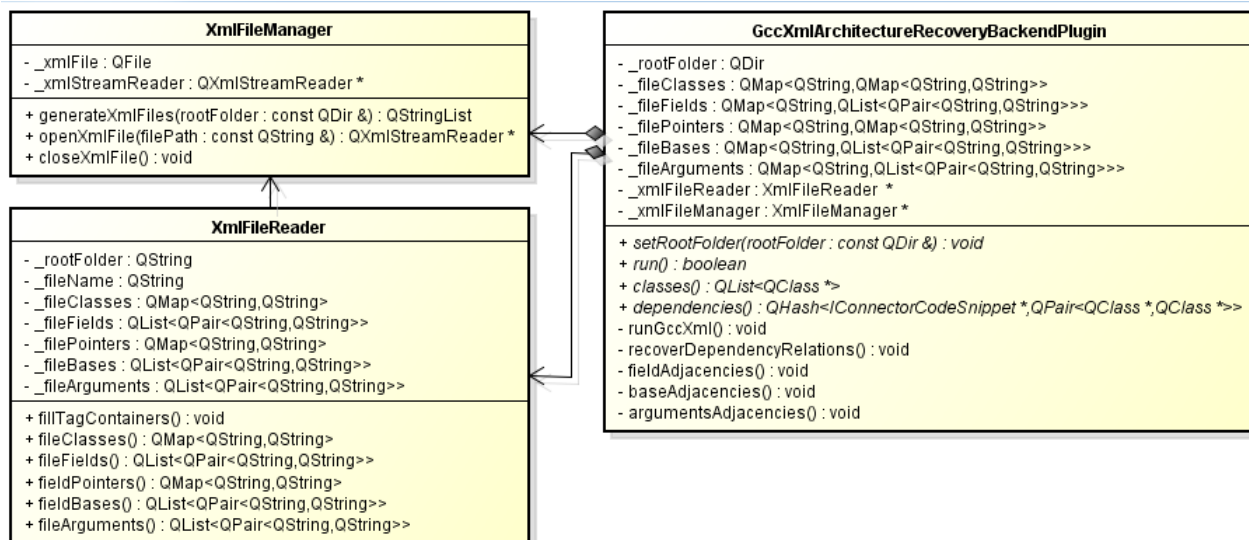

Figura 4: Implementação do *plugin* de *backend* de recuperação para sistemas desenvolvidos em C++, utilizando a ferramenta GCC-XML.

```
Programmer     Employee
Programmer     Computer
Programmer     Notebook
```

Listing 2: Demonstração de um trecho do resultado do processo de identificação de dependências entre as classes do sistema.

Vale ressaltar que esta foi apenas uma forma simplificada de utilização de todos os recursos oferecidos pelo *hot-spot* de recuperação de *backend*. A utilização de um método assessor para recuperar as relações de dependências ao invés o método *dependencies()* provê economia de esforços.

Se esta medida não fosse adotada, o *container* `QMultiMap` preenchido deveria ser desmembrado em diversos *containers* `QPair` para depois ser recuperado pelo *plugin* do algoritmo de recuperação arquitetural. E seguida, os *containers* de `QPair` deveriam ser estruturados em um novo `QMultiMap`, o que demandaria mais esforço de implementação para o desenvolvedor.

Sendo assim, o *framework* deve garantir flexibilidade na implementação do *hot-spot*, pois nem todos os métodos disponibilizados pela API são capazes de definir a forma mais eficaz e eficiente de executar tarefas próprias da técnica de recuperação. As tarefas mais básicas do *plugin* de *backend* de recuperação são armazenar todas as classes e os conectores identificados no sistema, mas como foi visto no caso da implementação do ACDC, estas informações nem sempre são desejadas.

A descoberta destes conectores é realizada através da estratégia de *code snippets*, que define expressões regulares utilizadas para identificar a forma de comunicação entre instâncias das classes da aplicação. Estas operações devem ser realizadas na implementação do método abstrato *run()* e seu papel é popular as estruturas de dados retornadas pelos métodos *classes()* e *dependencies()*.

Duas instâncias de `IConnectorCodeSnippet` trabalham em conjunto com a classe `GccXmlRecoveryBackendPlugin`: `ProcedureCallSnippet` e `EventSnippet`. Tais *snippets* são responsáveis pela definição de expressões regulares que identificam invocações síncronas convencionais de métodos e utilização do mecanismo de *signals/slots* do Qt (implementação do conector *Event*), respectivamente.

Apesar do *framework* ter sido projetado para suportar a descoberta de conectores, bem como as estratégias de *code snippets* não foram implementadas, mas postergadas para trabalhos futuros.

4.3. *O algoritmo de recuperação arquitetural ACDC*

O ACDC [12] é um algoritmo de recuperação arquitetural que integra detecção de padrões comumente adotados em recuperações manuais às operações básicas de *clusterização* de entidades. O objetivo é produzir

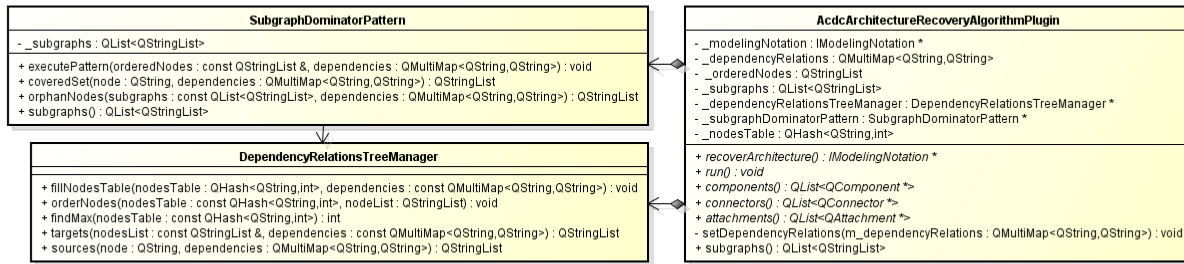

Figura 5: Implementação do *plugin* do algoritmo de recuperação arquitetural ACDC.

clusters que não abstraiam demasiadamente a estrutura interna do *software* e que utilizem rótulos significativos, facilitando a compreensão das representações geradas independente do nível de conhecimento técnico possuído pelo observador.

O algoritmo é composto por dois estágios principais: *i)* aplicação dos padrões de detecção para geração da decomposição que representa a arquitetura; e *ii)* alocação de entidades órfãs a *clusters* já existentes. O algoritmo define critérios para detecção dos seguintes padrões: *source file clusters* (aglomeração de entidades presentes em um mesmo arquivo de código-fonte), *body-header conglomeration* (aglomeração de declarações e definições de entidades), *leaf collection and support library identification* (entidades com número de dependentes maior que 20) e *ordered and limited subgraph domination* (uso de conjuntos dominantes de grafos para detecção de sub-sistemas).

A adoção de órfãos é uma técnica de clusterização incremental, utilizada no ACDC para alocar entidades não clusterizadas na primeira fase a algum *cluster* já existente. Quando mais de um *cluster* preexistente são igualmente propensos à adoção de uma determinada entidade, um novo *cluster* é criado.

O ACDC apresenta três características interessantes. Primeiro, a cada novo subsistema (*cluster*) criado pelo ACDC, um rótulo significativo é a ele atribuído, por exemplo, o nome do nó dominador do conjunto dominado encontrado. Segundo, a cardinalidade dos *clusters* encontrados é sempre limitada (não mais que 20 entidades), de modo a não abstrair demasiadamente a estrutura da aplicação. Por fim, a utilização dos padrões produz recuperações mais facilmente compreensíveis pois refletem as operações comuns de recuperação executadas manualmente por arquitetos.

A implementação da instância da *interface IArchitectureRecoveryAlgorithm* foi baseada no código original do ACDC, que foi desenvolvido na linguagem de programação Java. Após um breve período de estudo e de análise do código-fonte do algoritmo, foram identificados os trechos pertinentes às principais tarefas do algoritmo, como o processo de descoberta dos subgrafos dominadores e a adoção de órfãos.

Em seguida, ambas foram reescritas para a linguagem C++ de modo que gerassem os mesmos resultados da sua versão implementada em Java. Para facilitar as tarefas que são executadas pelo algoritmo, os recursos do Qt foram utilizados principalmente para a leitura e escrita dos arquivos de entrada e saída além do armazenamento e manipulação dos dados por meio dos *containers* oferecidos.

A execução da tarefa desempenhada pelo instância *plugin* do algoritmo, pertencente à classe `AcdcArchitectureRecoveryAlgorithmPlugin`, tem o seu início quando as informações *backend* são recuperadas da instância da classe `GccXmlArchitectureRecoveryBackendPlugin`.

Sendo assim, o primeiro passo da instância do algoritmo é carregar o seu estado com estas informações de *backend* para realizar a clusterização. O conteúdo destas informações nada mais é que um *container* `QMultiMap` que define todas as relações de dependência encontradas no sistema que deve ser analisado, sendo portanto fundamental para o funcionamento do algoritmo.

Na versão Java do algoritmo, o processamento das dependências tem como base três padrões de interdependências. O primeiro padrão, o *Body-Header*, não possui relevância alguma para a versão implementada no *framework*. Este fato ocorre porque, ao contrário da versão Java do algoritmo, a versão do *framework* não acessa diretamente os arquivos de cabeçalho e de corpo de métodos do C++. Ao invés disso, esta tarefa é delegada para a instância de *backend* de recuperação, que tem a sua tarefa facilitada graças ao uso do GCC-XML.

Neste ponto, é possível observar a capacidade customização do processo de recuperação arquitetural, uma vez que somente as partes mais relevantes da implementação do algoritmo em Java foi adaptada para fins mais específicos quando reimplementada em C++.

O segundo e mais importante padrão utilizado pelo ACDC é o *Subgraph Dominator*, pois é através dele que são gerados os *clusters* do sistema analisado. Partindo do princípio deste padrão, os conjuntos de classes e de suas relações de dependência dentro do *software* são tratados respectivamente como nós e arestas dentro de um grafo. O objetivo do algoritmo é descobrir nestes grafos a presença de subgrafos que sejam correspondentes aos subsistemas encontrados em um sistema.

Esta tarefa é realizada através da análise das relações de dependência entre as classes, relações estas que foram definidas no objeto `QMultiMap` que foi carregado na instância da classe no momento de sua inicialização. Nestas relações, a quantidade de dependências que uma classe possui é chamada de grau de saída, ao passo que a quantidade de dependentes que a classe possui é chamada de grau de entrada. O ACDC utiliza de ambos os valores para descobrir os subgrafos e seus respectivos nós dominadores.

A Figura **??** demonstra, em pseudoalgoritmo, como o processo de descoberta de subgrafos dominadores é executado. O ACDC desempenha seu papel de forma que o sistema seja fragmentado em *clusters* de tamanhos aproximados. Para tanto, ele realiza duas tarefas antes de executar esta atividade. A primeira e principal tarefa é a ordenação crescente dos nós pelo seu valor de grau de saída).

A análise de cada nó em ordem crescente é importante, pois evita que *clusters* menores sejam gerados na representação caso o nó analisado seja o dominador de um pequeno subgrafo. Subgrafos menores podem fazer parte de subgrafos maiores descobertos posteriormente, devendo portanto serem desconsiderados como *clusters* individuais nestes casos.

Na implementação do algoritmo, o método *fillNodesTable()* é utilizado para criar um *container* `QHash` que é útil para se ter um acesso rápido ao grau de saída de cada uma das classes do sistema. A instância da classe `DependencyRelationsTreeManager` tem como objetivo gerenciar e fornecer informações importantes sobre o *container _dependencyRelations*.

A tabela `QHash` que foi preenchida pelo seu método é então utilizada pelo método *orderNodes()* para que a ordenação seja feita. Com a ajuda do método privado `findMax()`, é possível encontrar a classe com maior grau de saída, e a partir dela, encontra-se as demais em ordem decrescente por meio de seu algoritmo.

A segunda tarefa desempenhada pelo ACDC é a limitação da cardinalidade dos *clusters*, isto é, a quantidade de classes que o compõe. Cardinalidades maiores do que 20, por exemplo, podem produzir *clusters* muito grandes principalmente se o sistema analisado for de grande porte.

Da mesma forma, permitir cardinalidades com apenas um ou dois elementos causa a geração de *clusters* minúsculos. Em ambos os casos, a representação do sistema torna-se menos compreensível e gerenciável do que deveria. Por estes motivos, o ACDC oferece ao usuário a possibilidade de delimitar o tamanho máximo e mínimo de cardinalidade dos *clusters*.

Apesar disso, este passo é opcional e não impede que o restante do algoritmo seja processado para a geração do resultado final. Assim, estes valores podem ser configurados ou não para garantir a geração de representações adequadas do sistema analisado independente de seu tamanho. Na implementação do *plugin*, este passo foi descartado, já que o objetivo era ter uma versão simplificada ao ponto de se tornar funcional para o *framework*.

```
1  orderEntities(entityMultiHash)
2
3  foreach(entity from entityMultiHash)
4     entityList = coveredSet(entity)
5
6     if(entityList not contains 1 item)
7        subgraphList.add(entityList)
8     endif
9  endfor
10
11 List<Item> coveredSet(item) {
12    List list, covered, falseOnes
13
14    do
15       list.add(item)
16       covered.add(targets(item))
17
18       do
19          List both;
20          both.addList(covered)
21          both.addList(list)
22
23          foreach(bothItem from both)
24             if(both not contains sources(
                   bothItem))
25                falseOnes.add(bothItem)
26             endif
27          endfor
28       while(covered.remove(falseOnes))
29    while(list.add(covered))
30 }
```

Listing 3: Pseudoalgoritmo do processo de descoberta de subgrafos dominadores.

O processo de descoberta de subgrafos é iniciado a

partir da análise de cada um dos nós por ordem crescente de grau de saída (linha 1). Primeiramente, quatro diferentes listas são criadas para atender as necessidades do algoritmo (linhas 12 e 19). A lista recebe o nome de (`list`), e é criada a princípio para armazenar o nó analisado. Contudo, caso um subgrafo seja descoberto, ela armazena também todos os nós dominados pelo nó analisado.

As demais listas são utilizadas no processo de descoberta de um subgrafo. A lista (`covered`) é criada para armazenar todas dependências dos nós contidos em `list`. A terceira lista (`falseOnes`) é criada para armazenar nós que não fazem parte do subgrafo e a lista (`both`) armazena temporariamente os nós que estão tanto em `list` quanto em `both`.

Com estas listas, o algoritmo deve buscar todos os nós que dependem dos nós da lista `both` e verificar se eles também fazem parte da lista `both` (linha 24). Se fizerem, significa que estes nós também fazem parte do subgrafo que está sendo descoberto, caso contrário, serão adicionados em `falseOnes` e excluídos da análise. Este processo se repete até que todos os nós de `falseOnes` sejam encontrados.

Caso todos os nós de `covered` sejam encontrados em `falseOnes`, ao fim do segundo *loop* `do-while` não haverá subgrafos para o nó analisado, pois `covered` estará vazia. Caso contrário, os nós que estiverem em `covered` serão adicionados em `list` e uma nova lista de `covered` será criada a partir da versão atualizada de `list`, para que todo o processo se repita.

Independente da quantidade de repetições realizadas, o algoritmo termina a sua execução quando todos os nós que fazem parte do subgrafo são encontrados e quando não há mais nenhum nó falso para ser descoberto.

Ao fim de cada análise separada dos nós, os subgrafos descobertos são adicionados separadamente em uma tabela *multihash* onde a chave é o nó dominador e os valores são os nós dominados. Neste ponto do algoritmo, os *clusters* obtidos a partir dos subgrafos podem ser representados sem a continuidade do algoritmo. Quando o processo de descoberta do subgrafo é encerrado, todos os nós que não fazem parte dos conjuntos dos dominados ou dos dominadores são armazenados em um *cluster* a parte denominado `orphanContainer`.

Este *cluster* não será excluído da representação, embora todos os órfãos nele contido possam ser incrementados nos subgrafos encontrados a partir do processo de doação de órfãos, que constitui a parte final do algoritmo.

No implementação do algoritmo, os processos de descoberta tanto dos nós dominadores quanto dos órfãos foram implementados na classe `SubgraphDominatorPattern` por meio dos métodos *executePatterns()* e *orphanNodes()*. Os métodos são chamados através de uma instância agregada à classe do *plugin* e são utilizados em conjunto com as operações da classe `DependencyRelationsTreeManager`.

Durante o processo de descoberta de subgrafos, esta última desempenha duas funções importantes. A primeira é a de recuperar todas as dependências de uma classe através do método *targets()*. A segunda é a recuperação de uma lista contendo as classes dependentes de uma das classes presente nas relações, feita através do método *sources()*.

Ao fim do método *run()* que é executado pela instância da classe do *plugin*, cada subgrafo é armazenado em uma `QStringList`, onde o primeiro elemento será sempre o nome do nó dominador (ou `orphanContainer.ss`, no caso do conjunto de órfãos) seguido pelos nomes de todos os nós dominados. Todas as listas que são criadas, por sua vez, são armazenadas em um objeto de `QList<QStringList>` para serem utilizadas como entrada na execução do processo de modelagem da arquitetura recuperada.

Na versão Java do ACDC, a representação do ACDC era feita de forma textual por meio da criação de um arquivo. O conteúdo deste arquivo lista todas as relações de dependência entre os nós dominadores e seus nós dominados seguindo o mesmo modelo de dependente-dependência demostrado no trecho 3. No caso do código implementado em C++, a criação deste arquivo foi descartada em prol de uma representação mais refinada, explicada mais a frente.

Os métodos *components()*, *connectors()* e *attachments()* representam as formas mais genéricas de recuperação de informações de uma arquitetura de *software*. Entretanto, somente o método *attachments()* atende as necessidades do algoritmo de recuperação arquitetural ACDC, que busca saber apenas as informações de dependências entre as classes do sistema. Através destes métodos, o *plugin* de notação de modelagem tem acesso às informações que deverá representar.

4.4. Suportando a notação de modelagem UML

Após a execução de um algoritmo de recuperação, os componentes, conectores e a topologia da arquitetura recuperada estão disponíveis através dos métodos *components()*, *connectors()* e *attachments()*. Entretanto, é necessário representar a arquitetura através do uso de alguma notação de modelagem.

Para viabilizar a geração de modelos arquiteturais descritos em UML, a classe `UmlModelingNotation`

UmlModelingNotation

- _clusters : QList<QUmlPackage *>
- _clustersStringList : QList<QStringList>

+ represent(components : const QList<QComponent *>, connectors : const QList<QConnector *>, attachments : const QList<QAttachment *>) : boolean
+ save(outputFile : const QFile &) : boolean
+ setClustersStringList(m_clusterList : QList<QStringList>) : void

Figura 6: Implementação do *plugin* da notação de modelagem UML.

(apresentada na Figura 1) implementa a interface `IModelingNotation` e utiliza os recursos do módulo *QtModeling* – parte integrante do DuSE-MT – para criar e exportar um diagrama de componentes que represente a arquitetura recuperada. O *QtModeling* implementa a versão 2.4.1 dos metamodelos das linguagens UML e MOF, bem como as operações de serialização de modelos no padrão XMI (*XML Metadata Interchange*).

Para isso, cada nó resultante do algoritmo de recuperação é instanciado como um *package* UML. Dependências UML, anotadas com um estereótipo que descreve um tipo particular de conector, são utilizadas para representar interações entre componentes. O modelo UML é então serializado no formato XMI e pode então ser visualizado no DuSE-MT ou em qualquer outra ferramenta em conformidade com a especificação XMI.

Na implementação feita a partir da *interface* de notação de modelagem (*IModelingNotation*), a lista encadeada de subgrafos que é gerada pelo ACDC é carregada pelo método configurador *setClustersStringList()* para ser acessada. As informações presentes neste *container* são extraídas e armazenadas em uma instância da classe `QUmlPackage`. A partir da leitura da lista, a aplicação cria um diagrama UML através desta classe para representar os *cluster* como *packages*, rotulando-os com os mesmos nomes dos nós dominadores encontrados pelo ACDC.

Em seguida, os nós dominados são representados como *sub-packages* dentro de seus respectivos *packages*, formando o conjunto de componentes que compõem os *clusters* do sistema. Uma vez criado este modelo, ele será armazenado em um arquivo de formato XMI para ser utilizado na geração da representação. No caso deste *plugin*, o arquivo XMI já é carregado pelo DuSE-MT ao final do método *represent()*, embora seja possível somente salvá-lo no sistema pelo método *save()*.

A depender do arquivo que é gerado pelos *plugins* de notação de modelagem, ele pode ser aberto por meio de qualquer ferramenta que suporte o seu formato. No caso do DuSE-MT, o método *openModel()* pertencente ao *projectController* da aplicação pode ser acessado para que o carregamento do XMI ocorra de forma instantânea.

Para tanto, basta que o caminho completo do arquivo seja passado como parâmetro. Caso a representação seja dependente de alguma outra ferramenta, esta pode ser chamada para carregar o arquivo através de um objeto de `QProcess`, disponibilizado pela biblioteca do Qt.

O carregamento da representação é o último passo do processo de recuperação arquitetural, quando o modelo arquitetural está pronto para auxiliar na compreensão e na execução das tarefas que o arquiteto ou os desenvolvedores desejam desempenhar sobre o sistema.

5. Validação

Visto que este trabalho não tem como objetivo a proposta de novas técnicas para recuperação arquitetural, a avaliação do *framework* proposto busca analisar três aspectos fundamentais: *i)* adequação dos *hot-spots* às variações atualmente existentes; *ii)* facilidade de compreensão e de instanciação do *framework*, por desenvolvedores; *iii)* a eficácia da execução do processo de um processo de recuperação arquitetural que foi implementado através do *framework*. A importância de se realizar esta análise se dá pela possibilidade de se levantar as principais vantagens trazidas pela utilização do *framework*, e pela descoberta dos principais aspectos que devem ser reexaminados, refinados ou corrigidos.

5.1. Adequação dos hot-spots às variações de técnicas existentes

Em relação ao primeiro aspecto, realizou-se um estudo para verificação da adequação da API definida no *framework* à implementação dos principais algoritmos de recuperação arquitetural, *backends* para obtenção de artefatos e notações de modelagem. O objetivo foi verificar se a API não é restrita demais a ponto de inviabilizar a utilização de certos algoritmos de recuperação, ou ampla demais a ponto de não trazer benefícios à produtividade de desenvolvimento.

Para realizar tal verificação, seis dos diversos algoritmos conhecidos e analisados pela literatura (ACDC, WCA [16], LIMBO [17], Bunch [18], MoJo [19] e ARC [13]) foram estudados de forma que fosse possível compreender seus mecanismos. Com base na leitura dos códigos-fonte disponíveis – ou mesmo dos trabalhos científicos que apresentem informações mais detalhadas sobre tais técnicas – foi possível concluir que todos podem ser facilmente suportados.

No caso dos algoritmos que foram estudados através de seus códigos-fonte – e mesmo durante a instanciação do *framework* que foi demonstrada neste trabalho – foi possível verificar um importante fator que auxilia na compreensão de tal facilidade. Suas implementações na maioria das vezes já possuem algum mecanismo ou para recuperar as informações dos sistemas a serem analisados ou para exibir a representação da arquitetura recuperada.

Assim, implementar o algoritmo de recuperação arquitetural pode se tornar uma tarefa mais fácil caso ao desconsiderar a implementação de ambos os mecanismos que o acompanha. Ainda assim, a flexibilidade provida pelo *framework* permite que qualquer um dos dois sejam reutilizados na implementação de *backend* de recuperação ou da notação de modelagem.

Um contraponto que foi identificado, entretanto, foi o fato de que as APIs de cada um dos *hot-spots* definidos ainda são genéricas demais para simplificar a implementação de um processo de recuperação arquitetural. Esta questão pode ser observada através da análise das classes de *plugins* que foram implementadas. No caso dos *backends* de recuperação, por exemplo, pode-se ter mais de um meio de se extrair as informações do código, seja por código-fonte ou a partir da documentação do sistema.

Além disso, a utilização de uma ferramenta – ou para auxiliar, ou para automatizar completamente o processo de recuperação de informações – pode ser uma possível alternativa, como foi visto no caso da utilização do GCC-XML para a instanciação do *framework*. Quanto aos algoritmos de recuperação arquitetural, sabe-se que podem ser identificados ou como algoritmos de clusterização, ou como *graph pattern matching*, sendo que para cada um deles podem ser divididos em subprocessos definidos pelo *hot-spot* de recuperação arquitetural.

Sendo assim, uma possível melhoria para solucionar esta questão é a definição de subcategorias das três etapas do processo de recuperação arquitetural, onde *template methods* disponibilizariam uma infraestrutura que melhoraria a produtividade na implementação de técnicas particulares (*hot-spots* de menor granularidade).

5.2. *Facilidade de compreensão e de instanciação do framework*

Apesar do *framework* apresentar *interfaces* que definem e separam as três principais variações de um processo de recuperação arquitetural, é possível que haja dificuldade para entender o seu *workflow* detalhado. Os métodos que são definidos pela *interface* nem sempre podem servir para a implementação de alguma destas variações.

Um exemplo que pode ser analisado para compreender esta questão com mais clareza está na forma como o algoritmo ACDC foi implementado. Por sua natureza, o algoritmo não considera detalhes sobre componentes e nem dos conectores que definem seus inter-relacionamentos. Desta forma, os métodos *components()* e *connectors()* se tornam inutilizados caso seja feita uma implementação fiel a versão original do algoritmo.

Um outro caso onde a compreensão e instanciação do *framework* podem ter suas complexidades aumentadas é originado da necessidade de implementar muitos métodos que já poderiam ser oferecidos pelo *framework*. Dentre alguns dos exemplos de implementações que poderiam existir estão a automatização de busca por código-fonte através da navegação de árvores de diretórios, ou mesmo uma abstração para a leitura de arquivos de código-fonte ou de documentação.

A possibilidade de reutilização destes mecanismos mais triviais no processo de recuperação arquitetural facilitaria ainda mais a implementação do *framework*, delegando ao desenvolvedor apenas a responsabilidade de implementar os procedimentos mais específicos adotados pelas técnicas.

Independente se o processo de recuperação arquitetural descarta a utilização de alguns métodos ou demanda a implementação de alguma tarefa que poderia ser automatizada, problemas na compreensão e na instanciação do *framework* são bastante reduzidos por meio de uma documentação bem elaborada.

Descrições detalhadas sobre cada método definido pelos *hot-spots*, bem como os possíveis *workflows* que podem ser executados por um processo de recuperação arquitetural, são definitivos para auxiliar tanto na compreensão quanto na instanciação. Com isto, a produtividade na criação das técnicas de recuperação torna-se mais elevada, reduzindo esforços maiores que seriam feitos para compreender o *framework* apenas pelo estudo de sua API.

5.3. Eficácia da execução do processo de recuperação arquitetural implementado

Para verificar se o *framework* proposto realmente oferece flexibilidade e um bom funcionamento dos processos de recuperação arquitetural neles integrados, foi realizado um teste com a implementação do ACDC desenvolvida. Vale ressaltar que a implementação dos *plugins* de recuperação de *backend* e do algoritmo de recuperação não foram refinadas ou completas o suficiente para gerarem bons resultados de desempenho ou de precisão na representação. Contanto, os objetivos principais não são estes.

A principal motivação para a execução de um teste de execução é verificar se o próprio *framework* suporta a sua execução sobre um sistema que possui muitas classes separadas em muitos subdiretórios A principio, esta validação desconsiderou testes em sistemas que utilizam alguma outra biblioteca que não seja a padrão do C++. Nestes casos, a execução do GCC-XML precisa ser incluída no processo de compilação do próprio sistema, através da configuração do *Makefile* (arquivo que define as diretivas de compilação de um *software* feito em C++).

Portanto, o sistema alvo escolhido foi o módulo *core* da biblioteca do Qt 5, já que este apresenta um grande número de classes bem dispersas em sua árvore de diretório além de serem implementadas em C++ puro. Através do painel de configuração e execução do processo de recuperação arquitetural (Figura 3), a iniciação da tarefa torna-se intuitiva ao usuário.

Como esperado, o painel carregou corretamente os *plugins* já desenvolvidos de forma dinâmica, bem como realizou a execução do processo de forma estável e bem-sucedida. Foi verificado que a arquitetura baseada em *plugins* provida pelo DuSE permitiu uma comunicação rápida entre as instâncias que foram implementadas.

Por não ter sido implementado da forma mais ótima possível, o *plugin* de *backend* recuperação foi o que levou maior tempo para terminar sua execução. O GCC-XML foi executado com sucesso em todas as classes, e os arquivos XML gerados foram todos acessados para encontrar 230 relações de dependências presentes. A partir deste ponto, todo o processo discorreu de forma rápida.

Este resultado demonstra de forma clara que o *framework* é altamente eficaz na execução das técnicas de recuperação arquitetural por ele suportadas. Se uma instanciação do *framework* que foi feita com o propósito de demandar maiores esforços de processamento da aplicação foi bem sucedida, então uma instanciação feita de forma ótima seria capaz de realizar a mesma tarefa sem problemas.

Figura 7: Representação do arquivo XMI do DuSE-MT.

6. Trabalhos Correlatos

Uma série de esforços para recuperação arquitetural podem ser encontrados na literatura. [20] Apresentam uma abordagem para recuperação arquitetural baseada na utilização de *Latent Semantic Indexing* (LSI) e clusterização via *k-means*. Adicionalmente, eles definem mecanismos para adição e remoção de componentes na representação LSI (*fold-in/fold-out*), reduzindo o custo computacional necessário para a recuperação.

[21] propõem uma abordagem para recuperação arquitetural que é utilizada como um guia para processos subsequentes de reengenharia de sistemas legados. Técnicas para análise de dependência entre módulos são utilizadas para a obtenção de recuperações que apresentam componentes com alta coesão, baixo acoplamento e interfaces de comunicação mínimas.

[22] apresentam uma abordagem para recuperação arquitetural que considera a presença das deficiências de projeto mais relevantes. O método proposto apresenta os benefícios alcançados caso as deficiências detectadas sejam removidas, como base para a execução de um processo de reengenharia Os componentes mais relevantes e as deficiências de projeto são detectados através do uso de métricas que avaliam incompatibilidade de inter-

faces e organização lógica de classes em pacotes.

[23] apresentam uma abordagem híbrida que combina técnicas de clusterização e detecção de padrões para realização de recuperação arquitetural. A abordagem utiliza ainda a detecção e remoção de *bad smells* para melhorar os resultados das operações de clusterização.

Um *plugin* para o Eclipse – denominado MARPLE – suportando a detecção de *design patterns* e recuperação de arquiteturas é apresentado em [24]. O *plugin* baseia-se na utilização de métricas e elementos básicos extraídos da árvore sintática abstrata que representa o código-fonte. A detecção de *design patterns* é realizada através da identificação de subcomponentes que indicam a presença de um *design pattern* particular. Experimentos para detecção de *Abstract Factories* e geração de visões arquiteturais são também apresentados no artigo.

[25] apresenta um modelo e uma ferramenta para recuperação arquitetural, baseados na utilização de padrões arquiteturais definidos pelo usuário e de técnicas de *graph pattern matching*. Experimentos avaliando a complexidade de tempo e espaço da recuperação, bem como a precisão dos artefatos recuperados são citados no artigo.

[6] apresentam uma técnica para recuperação arquitetural baseada na obtenção de dados comportamentais de um *software* em execução. Adicionalmente, modificação em *run-time* podem ser efetuadas no sistema através da manipulação direta do modelo recuperado. Os autores utilizam uma plataforma de *middleware* com capacidades de reflexão computacional para implementar a recuperação e a adaptação dinâmica. Experimentos com o PKUAS – um servidor de aplicação baseado no J2EE – são apresentados ao final.

O *framework* apresentado neste artigo difere dos trabalhos acima citados em uma série de aspectos. Primeiro, a infraestrutura aqui proposta permite a recuperação de arquiteturas sem uma dependência direta com uma determinada plataforma de desenvolvimento, algoritmo de recuperação ou notação de modelagem. Segundo, a solução dá ênfase à recuperação e representação de conectores de *software* como entidades arquiteturais de primeira classe.

Tal característica é importante em sistemas cujo atendimento de requisitos de escalabilidade, tolerância a falhas e segurança é mérito majoritariamente do uso de conectores sofisticados. Por fim, o *framework* aqui proposto pode constituir plataforma eficiente para comparação de técnicas de recuperação e/ou definição de *benchmarks* a serem utilizados em futuras pesquisas.

7. Conclusões e Trabalhos Futuros

Este artigo apresentou o projeto e implementação de um *framework* para recuperação arquitetural independente de plataforma, algoritmo de recuperação e notação de modelagem das arquiteturas recuperadas. Um exemplo de instanciação do *framework* para recuperação de arquiteturas de sistemas implementados em C++ foi também apresentado.

Foi feita também uma análise de sua API para verificar a sua capacidade de oferecer uma boa compreensão e suporte para implementações produtivas dos algoritmos de recuperação arquitetural existentes. Além disso, um teste de execução permitiu verificar se o *framework* não apresentaria falhas durante a realização da tarefa de recuperação arquitetural.

A grande flexibilidade, extensibilidade e desempenho providos pelo *framework* desenvolvido permite que desenvolvedores e arquitetos consigam implementar e reutilizar as técnicas de recuperação arquitetural que desejam. Uma única técnica pode ser reimplementada para apresentar diversas variações, permitindo que a recuperação seja feita em qualquer plataforma, por meio de qualquer algoritmo ou representada em diversas notações.

Por meio de um painel de configuração, todas os processos e variações estão ao alcance do usuário de forma simples e intuitiva. Por fim, não existe limites para a de *plugins* que podem ser suportados pela aplicação, permitindo que diversas implementações diferentes do mesmo.

O *framework* apresentou viabilidade para executar qualquer processo de recuperação arquitetural com eficácia e eficiência. Com isso, espera-se que ele seja capaz de auxiliar quaisquer especialistas que desejem utilizar o DuSE-MT para recuperar e manipular as arquiteturas de *software*.

Os trabalhos futuros possíveis vão desde a parte descritiva de toda a API fornecida pelo *framework* até o próprio refino e extensão desta. A importância de se ter uma boa documentação para um *framework* auxilia o desenvolvedor a projetar melhor o seu processo de recuperação arquitetural utilizando o máximo dos recursos oferecidos.

Estender a API que já é oferecida pelo *framework* também pode aumentar a facilidade no desenvolvimento das técnicas de recuperação arquitetural independente de sua categoria. Isto ocorre porque os *hot-spots* mais detalhados definem métodos específicos para cada tarefa que um algoritmo ou um *backend* de recuperação pode executar, por exemplo.

A automatização de algumas tarefas triviais na

recuperação de arquitetura de *software* também é uma tarefa que deve ser pensada. Reutilizar métodos de navegação de subdiretórios do sistema ou métodos de leitura e escrita de arquivos permitiriam que os desenvolvedores pudessem se concentrar somente na implementação das principais tarefas desempenhadas pelo processo de recuperação arquitetural. Sendo assim, tal medida tornaria mais rápido o processo de implementação das possíveis técnicas.

Outra forma de automatização de tarefas a se considerar diz respeito às resoluções de inconsistências presentes nas representações arquiteturais geradas. Enquanto algumas técnicas de recuperação arquitetural já possuem embutidas em suas implementações este tipo de mecanismo, seria interessante desenvolver uma forma de refinar modelos arquiteturais num contexto mais genérico. Deste modo, a utilização do *framework* concederá aos seus usuários maiores garantidas de representações mais consistentes com o sistema analisado.

Possíveis trabalhos futuros também incluem a definição de mecanismos mais sofisticados para detecção de conectores *composite*, projeto de *hot-spots* de menor granularidade para facilitar a implementação de algoritmos específicos de recuperação e melhor suporte a algoritmos de clusterização através da disponibilização prévia de métricas de similaridade.

Referências

[1] R. N. Taylor, N. Medvidovic, E. M. Dashofy, Software Architecture: Foundations, Theory, and Practice, Wiley Publishing, 2009.

[2] O. Maqbool, H. Babri, Hierarchical clustering for software architecture recovery, IEEE Trans. on Software Eng. 33 (11) (2007) 759–780. doi:10.1109/TSE.2007.70732.
URL http://dx.doi.org/10.1109/TSE.2007.70732

[3] J. Garcia, I. Ivkovic, N. Medvidovic, A comparative analysis of software architecture recovery techniques, in: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th Intl. Conference on, IEEE, 2013, pp. 486–496.

[4] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, J. E. Robbins, Modeling software architectures in the unified modeling language, ACM Transactions on Software Engineering and Methodology (TOSEM) 11 (1) (2002) 2–57. doi:10.1145/504087.504088.
URL http://doi.acm.org/10.1145/504087.504088

[5] N. Medvidovic, R. N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on Software Engineering 26 (1) (2000) 70–93. doi:10.1109/32.825767.
URL http://dx.doi.org/10.1109/32.825767

[6] G. Huang, H. Mei, F.-Q. Yang, Runtime recovery and manipulation of software architecture of component-based systems, Automated Software Engg. 13 (2) (2006) 257–281. doi:10.1007/s10515-006-7738-4.
URL http://dx.doi.org/10.1007/s10515-006-7738-4

[7] K. Sartipi, Software architecture recovery based on pattern matching, in: Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, IEEE, 2003, pp. 293–296.

[8] D. Conte, P. Foggia, C. Sansone, M. Vento, Thirty years of graph matching in pattern recognition, IJPRAI 18 (3) (2004) 265–298.

[9] J. Garcia, I. Krka, C. Mattmann, N. Medvidovic, Obtaining ground-truth software architectures, in: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013, pp. 901–910.

[10] J. Garcia, I. Krka, N. Medvidovic, C. Douglas, A framework for obtaining the ground-truth in architectural recovery, in: Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on, IEEE, 2012, pp. 292–296.

[11] S. S. Andrade, R. J. d. A. Macêdo, A search-based approach for architectural design of feedback control concerns in self-adaptive systems, in: Proceedings of the 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2013, IEEE, Philadelphia, PA, USA, 2013.

[12] V. Tzerpos, Comprehension-driven software clustering, Ph.D. thesis, University of Toronto (2001).

[13] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, Y. Cai, Enhancing architectural recovery using concerns, in: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 552–555. doi:10.1109/ASE.2011.6100123.
URL http://dx.doi.org/10.1109/ASE.2011.6100123

[14] N. R. Mehta, N. Medvidovic, S. Phadke, Towards a taxonomy of software connectors, in: Proceedings of the 22Nd International Conference on Software Engineering, ICSE '00, ACM, New York, NY, USA, 2000, pp. 178–187. doi:10.1145/337180.337201.
URL http://doi.acm.org/10.1145/337180.337201

[15] B. K. King, GCC-XML, http://gccxml.github.io/. Acesso: 27/03/2014, 2014.

[16] M. Chatterjee, S. Das, D. Turgut, An on-demand weighted clustering algorithm (wca) for ad hoc networks, in: Global Telecommunications Conference, 2000. GLOBECOM '00. IEEE, Vol. 3, 2000, pp. 1697–1701 vol.3. doi:10.1109/GLOCOM.2000.891926.

[17] P. Andritsos, P. Tsaparas, R. J. Miller, K. C. Sevcik, Limbo: A scalable algorithm to cluster categorical data (2003).

[18] S. Mancoridis, B. S. Mitchell, Y. Chen, E. R. Gansner, Bunch: A clustering tool for the recovery and maintenance of software system structures, in: In Proceedings; IEEE International Conference on Software Maintenance, IEEE Computer Society Press, 1999, p. pages.

[19] V. Tzerpos, R. Holt, Mojo: a distance metric for software clusterings, in: Reverse Engineering, 1999. Proceedings. Sixth Working Conference on, 1999, pp. 187–193. doi:10.1109/WCRE.1999.806959.

[20] M. Risi, G. Scanniello, G. Tortora, Using fold-in and fold-out in the architecture recovery of software systems, Formal Asp. Comput. 24 (3) (2012) 307–330.

[21] L. Wu, Y. Feng, H. Yan, Software reengineering with architecture decomposition, in: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC '07, ACM, New York, NY, USA, 2007, pp. 1489–1493. doi:10.1145/1244002.1244320.
URL http://doi.acm.org/10.1145/1244002.1244320

[22] M. C. Platenius, M. von Detten, S. Becker, Archimetrix: Improved software architecture recovery in the presence of design deficiencies, 2011 15th European Conference on Software Maintenance and Reengineering 0 (2012) 255–

264. doi:http://doi.ieeecomputersociety.org/10.1109/CSMR.2012.33.

[23] M. von Detten, S. Becker, Combining clustering and pattern detection for the reengineering of component-based software systems, in: Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – ISARCS, QoSA-ISARCS '11, ACM, New York, NY, USA, 2011, pp. 23–32. doi:10.1145/2000259.2000265.
URL http://doi.acm.org/10.1145/2000259.2000265

[24] F. A. Fontana, M. Zanoni, A tool for design pattern detection and software architecture reconstruction, Inf. Sci. 181 (7) (2011) 1306–1324. doi:10.1016/j.ins.2010.12.002.
URL http://dx.doi.org/10.1016/j.ins.2010.12.002

[25] K. Sartipi, Software architecture recovery based on pattern matching, in: ICSM, IEEE Computer Society, 2003, pp. 293–.