Avaliação de Projetos Arquiteturais Baseados em Busca

Erik Seyti Johansson[1], Willian Nalepa Oizumi[1]

[1]Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas – Instituto Federal do Paraná (IFPR) – Campus de Paranavaí
87703-536 – Paranavaí – PR– Brasil

erikseyti@gmail.com, wilian.oizumi@ifpr.edu.br

Abstract. *This paper presents the results of an exploratory study on search-based software architecture project. The employed methodology was an exploratory study involving 2 software projects. In this study, we used an approach -- called MOA4PLA – which generates optimal software architectures. From this work, it was possible to check the quality of the optimized architectures and to identify architectural problems in those architectures.*

Resumo. *Este artigo apresenta os resultados de uma pesquisa sobre projeto de arquitetura de software baseado em busca. A metodologia utilizada foi a realização de um estudo exploratório envolvendo 2 projetos de software. Neste estudo utilizou-se uma abordagem -- chamada MOA4PLA -- que gera soluções otimizadas para arquitetura de software. A partir deste estudo, foi possível verificar a qualidade das arquiteturas otimizadas, assim como identificar problemas arquiteturais presentes nessas arquiteturas.*

1. Introdução

Uma das principais funções existente dentro da engenharia de software é permitir ditar ou mesmo aconselhar as melhores práticas para as empresas de como organizar seus produtos de software assim como as equipes que as criam e as mantêm atualizados. Sendo assim, um dos aspectos de suma importância no desenvolvimento de um sistema é a Arquitetura de Software. A arquitetura de software abrange um conjunto de decisões significativas sobre a organização de um sistema [Booch 2004]. Caso a arquitetura seja mal projetada, poderá afetar futuras implementações ou mesmo manutenções do seu código fonte. Por isso, já foram propostas diferentes ferramentas e plataformas para ajudar a projetar e simplificar seu desenvolvimento, mas essas abordagens não impedem deslizes ou mesmo erros na hora de projetá-la. Como a evolução de uma arquitetura de software não é trivial, Colanzi (2014) propôs uma abordagem para otimização multiobjectivo que faz uso de algoritmos evolutivos para projetos de arquitetura chamada MO4PLA.

A abordagem MOA4PLA gera inúmeras arquiteturas de software a partir do uso de algoritmos genéticos. Entretanto, a abordagem MOA4PLA ainda não é capaz de identificar e remover Problemas Arquiteturais nos projetos gerados. Tais problemas são causados por decisões arquiteturais (podendo ou não ser intencionais) que afetam de forma negativa a qualidade do sistema. Existem catálogos documentando problemas arquiteturais genéricos [Garcia 2009] [Ganesh *et al.* 2013]. Todavia, na medida de nosso conhecimento, não existem heurísticas que permitam a identificação efetiva de problemas arquiteturais em projetos de arquitetura de software. Sendo assim, neste trabalho realizou-se uma avaliação em um conjunto de arquiteturas geradas por meio da

abordagem MOA4PLA, a fim de encontrar problemas arquiteturais nos projetos gerados. Com base nos resultados dessa análise, pretende-se, em trabalhos futuros, identificar estruturas recorrentes que podem servir de base para heurísticas de identificação automatizada de problemas arquiteturais.

O restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta os conceitos chaves deste trabalho. A Seção 3 contém detalhes sobre a configuração deste estudo. A Seção 4 apresenta os resultados obtidos a partir desta pesquisa. Por fim, a Seção 5 apresenta nossas conclusões finais.

2. Contextualização

Esta Seção apresenta uma breve descrição sobre os conceitos chave deste trabalho. O conceito de Arquitetura de Software é apresentado na Seção 2.1. A definição de Problema Arquitetural é apresentada na Seção 2.2. A Seção 2.3 descreve o conceito de Engenharia de Software Baseada em Busca. Por fim, a Seção 2.4 apresenta a abordagem MOA4PLA.

2.1. Arquitetura de Software

Projeto de Arquitetura de Software pode ser definido como sendo a organização do sistema todo entre componentes (ou subsistemas), interfaces e seus relacionamentos [Booch 2004]. O diagrama da Figura 1 apresenta a arquitetura do sistema Mobile Media [Oizumi *et al.* 2012]. Como se pode observar, a arquitetura do sistema Mobile Media estabelece a divisão dos componentes em duas camadas -- System Services e Business Services – de acordo com o papel de cada componente. Além disso, a arquitetura estabelece as interfaces pelas quais os componentes se comunicam como, por exemplo, `IAlbumMgt` e `IMediaMgt`.

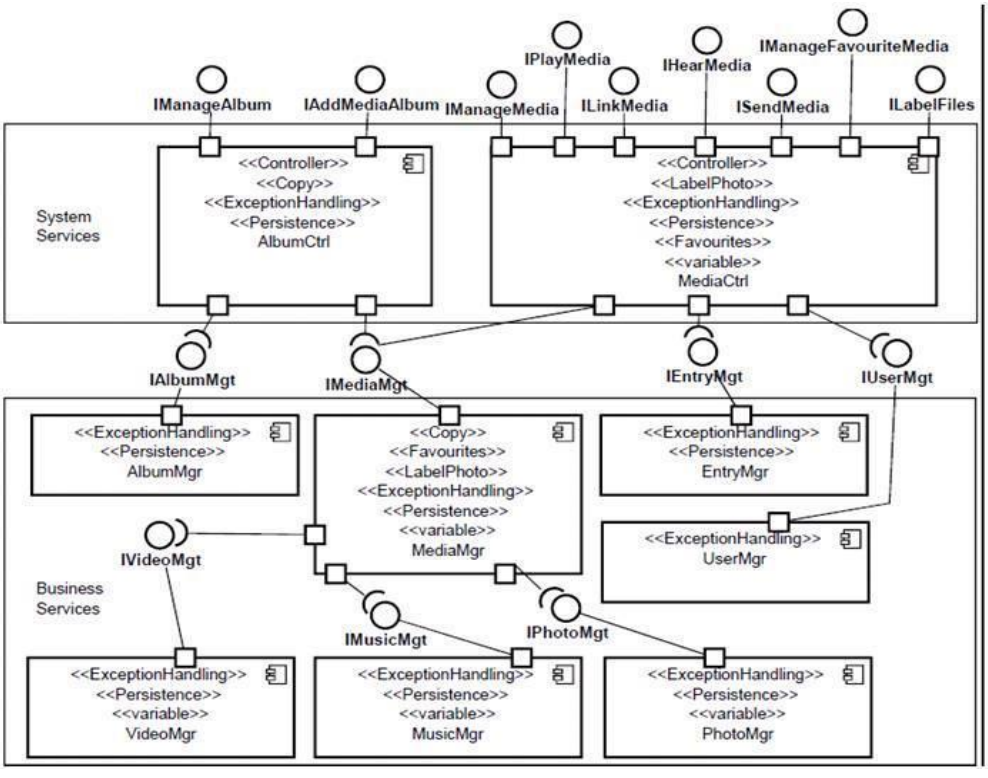

Figura 1. Arquitetura de Software do Sistema Mobile Media [Oizumi et al. 2012].

2.2. Problemas em Arquitetura de Software

Para adaptar softwares a diferentes plataformas, integrar funções, variáveis externas ou mesmo evoluir, deve-se ter cuidado quanto ao seu gerenciamento e sua execução. De forma frequente é necessário o refatoramento do seu código fonte, para que essas novas funcionalidades possam trabalhar de forma adequada e integrada às demais funções do sistema. A maneira mais eficaz de verificar a eficiência dentro de diferentes aspectos no sistema (usabilidade, testabilidade, inteligibilidade, reusabilidade entre outros) após implementação é identificar os códigos classificados como *Bad Smells* [Fowler 1999].

Bad Smells são implementações na sua arquitetura que afetam de forma negativa alguma das seguintes características em um sistema: entendimento pleno do sistema por parte da equipe, sua testabilidade, sua extensibilidade, seu reuso e principalmente sua manutenção futura. Eles são normalmente construtores presentes no nível de implementação do sistema como: atributos, classes, parâmetros, instanciamento de objetos ou códigos duplicados. Esse tipo de problema é decorrente de uma péssima integração ou entendimento de recursos externos como: conectores, componentes externos que deveriam ser delegados aos construtores, entre outros. Um exemplo de *Bad Smell* é *Long Method* – um método muito longo (em linhas de código) e complexo [Fowler 1999].

Em sistemas construídos de forma industrial podem ser percebidos fragmentos de elementos recorrentes do Projeto de Arquitetura de Software que afetam negativamente sua manutenção. Tais estruturas são semelhantes aos *Bad Smells* e são chamadas de Problemas Arquiteturais. Os Problemas Arquiteturais são classificados como (normalmente não de forma intencional) decisões arquiteturais que afetam de forma negativa a qualidade do sistema. Exemplos de problemas arquiteturais são as Dependências Cíclicas. Uma dependência cíclica acontece quando existe um relacionamento entre dois ou mais componentes que, direta ou indiretamente, dependem um do outro para funcionar corretamente. Tal característica é indesejada, pois torna os componentes altamente acoplados, prejudicando a manutenção do sistema.

Uma das causas para problemas arquiteturais é a aplicações de soluções de projeto em um contexto impróprio, junção de diversas partes de diferentes projetos provocando funções ou mesmo comportamentos de forma indesejada. Porém esses tipos de aplicação podem ter seus benefícios como melhoria na performance ou mesmo no reuso, pelo preço do ciclo de vida de suas funções e propriedades. Dessa forma os Problemas Arquiteturais envolvem diretamente uma troca (podendo ou não ser vantajosa) entre suas propriedades, e por isso os programadores devem decidir que ações devem tomar para ter maior proveito do contexto da aplicação. Essa correção deve ocorrer de forma que altere sua estrutura interna, mas sem alterar a estrutura externa.

2.3. Engenharia de Software Baseada em Busca

Algoritmos de busca são algoritmos que se utilizando de artefatos de um software como entrada buscam encontrar uma solução a partir da geração de várias possíveis soluções para determinado problema. No contexto de Engenharia de Software, a criação ou evolução de uma arquitetura de software pode gerar vários problemas se não for bem projetada, gerando muitas vezes assim os problemas arquiteturais que podem afetar diretamente na qualidade final do produto de software.

Dessa forma, com a utilização dos algoritmos de busca pode-se gerar a melhor solução para a arquitetura de um sistema, por meio de se achar uma combinação de propriedades mais adequada. Por isso, foi proposto por Colanzi (2014) o MOA4PLA para gerar essas arquiteturas de forma automática utilizando algoritmos baseados em busca.

2.4. MOA4PLA

Um importante aspecto do desenvolvimento de software é a arquitetura, que abrange um conjunto de decisões significativas sobre a organização de um sistema de software [Féderle, 2014], incluindo uma seleção de elementos arquiteturais que compõem seu sistema e suas interfaces. Assim como qualquer estrutura complexa, um software deverá ser projetado e desenvolvido sobre uma fundação sólida. Pois caso seja mal projetado pode ocasionar em graves problemas futuros como: instabilidade, não comprimento de requisitos de negócio, dificuldade para manutenção e implementação, dificuldades com os escalonamentos, entre outros.

Ferramentas e plataformas modernas já foram propostas para simplificar tarefas complexas no seu desenvolvimento. Entretanto, esses recursos não substituem a necessidade de projetá-lo cuidadosamente, tomando como base os requisitos e cenários onde o software atuará. Além disso, a evolução de uma arquitetura de software se mostrou muito mais complexa do que se poderia imaginar. Considerando esse fato, Colanzi (2014) apresentou uma abordagem para otimização multiobjectivo que faz uso de algoritmos evolutivos para projetos arquiteturais de software chamada MOA4PLA.

Possuindo como principal objetivo avaliar e melhorar projetos referentes a sua arquitetura de software em termos de princípios básicos de projeto, modularização de características e extensibilidade dos projetos criados a partir dela. Com isso produz um conjunto de soluções que apresenta a melhor troca de vantagens e desvantagens na sua implementação (conhecidas pelo termo inglês *trade-off*) entre as várias características e elementos arquiteturais do projeto a serem otimizados. Para isso é dado como entrada uma representação do projeto por meio de um diagrama de classes, que sofrerá efeitos do processo de busca, e assim obtendo um conjunto de soluções. Para que uma dada arquitetura seja passível de otimização. Essa representação é feita em base em um metamodelo.

Dentro de um único problema arquitetural pode-se ter inúmeras soluções que podem ser utilizadas para resolvê-lo. Cada uma dessas soluções possui seu próprio *trade-off*. Para cada uma das soluções apresenta-se uma abordagem para resolução do problema. No entanto o software MO4PLA gera inúmeras arquiteturas, sem que seja possível verificar a real qualidade das arquiteturas geradas. Este trabalho propõe analisar essas arquiteturas geradas pelo software e avaliá-las.

3. Configuração do Estudo

Nesta Seção são apresentadas as configurações definidas para a realização desta pesquisa. Na Seção 3.1 são apontados os objetivos deste estudo. A Seção 3.2 descreve os procedimentos definidos para a realização dos experimentos. Na Seção 3.3 são apresentados os sistemas utilizados nos experimentos. Por fim, na Seção 3.4 são indicados as ferramentas utilizadas na pesquisa.

3.1. Objetivos

Projetos de arquitetura de software gerados por meio de algoritmos de busca podem apresentar problemas arquiteturais graves. Tais problemas não podem ser localizados e removidos pelos algoritmos de busca. Isso ocorre porque, na medida de nosso conhecimento, os algoritmos de busca não estão preparados para encontrar tais problemas. Sendo assim, este trabalho tem como foco encontrar exemplos de problemas arquiteturais recorrentes em projetos arquiteturais baseados em busca, ou seja, projetos arquiteturais gerados por meio de algoritmos de busca. Dessa forma, pretende-se obter subsídios para definir heurísticas para a identificação automatizada de problemas arquiteturais em projetos de software. Para isso, este trabalho tem como objetivos específicos:

- Revisão de literatura sobre: arquitetura de software, problemas arquiteturais, linha de produto de software, algoritmos de otimização multiobjetivo e projeto arquitetural baseado em busca.
- Estudo sobre o domínio e a arquitetura dos sistemas Arcade Game Maker e Mobile Media.
- Avaliação manual de projetos arquiteturais baseados em busca.
- Análise dos resultados obtidos.

3.2. Sistemas Avaliados

Afim da realização desta pesquisa, foram escolhidos 2 tipos de projetos arquiteturais diferentes para este estudo, sendo desenvolvido a pesquisa em cima das arquiteturas Arcade Game Maker e Mobile Media. O Arcade Game Maker (AGM) é uma família de produtos de jogos voltada para uso pedagógico criada pelo Software Engineering Institute [SEI 2016]. O Mobile Media é sistema para dispositivos móveis, cujo objetivo é o gerenciamento de mídias, tais como fotos e vídeos [Oizumi et al. 2012].

3.3. Procedimentos

Cada uma das arquiteturas analisadas neste estudo foi definida através de diagramas no formato XML. Onde os formatos visuais destes diagramas se assemelha a um diagrama de classe, contendo respectivamente em suas arquiteturas apenas as classes e seus respectivos métodos e atributos.

A Figura 2 apresenta um trecho presente dentro da arquitetura AGM2, podendo ser verificado a partir dela o relacionamento entre as classes, todos seus atributos e métodos e identificar se esta classe é uma classe de interface ou não do projeto (identificado neste exemplo pelo <<interface, play>> acima do nome da classe).

Com as arquiteturas em mãos é possível assim fazer uma análise qualitativa e visual cujo objetivo é identificar os problemas existentes dentro dessas arquiteturas. Por exemplo, na arquitetura AGM2 foi encontrado dentro da classe *GameBoard* o problema de *Large Class*, problema resultante de quando uma classe possui um número excessivo de atributos e/ou métodos em código fonte. Onde neste exemplo a classe *GameBoard* apresenta uma quantidade grande no número de métodos.

Para cada uma das arquiteturas analisadas foi criado um relatório para registrar todos os problemas achados nas arquiteturas. Sendo separada a organização do relatório por classe identificada com algum problema arquitetural. Para cada classe representada no relatório é mostrado: o nome da classe, o problema que a classe apresenta uma definição do problema, o meio para fazer a refatoração da classe e uma imagem que representa mostra as informações na classe (nome da classe, atributos e métodos).

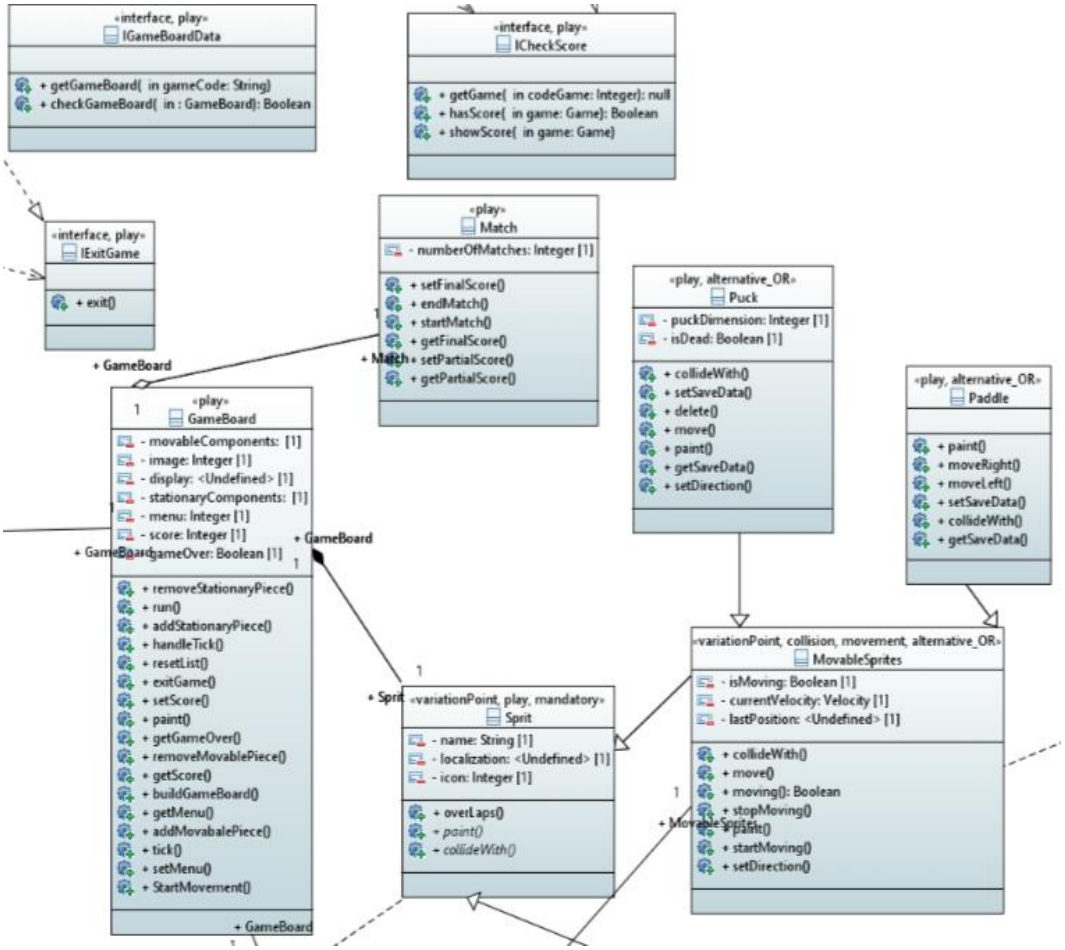

Figura 2. Exemplo de um trecho da Arquitetura AGM 2, podendo ser visualizados em cada uma das classes, o nome da classe, seus atributos e seus respectivos métodos.

3.4 Materiais da Pesquisa

Este estudo buscou identificar os diferentes tipos de problemas arquiteturais a partir de 6 diferentes arquiteturas de software disponíveis para este estudo, 3 delas referentes as arquiteturas da AGM e 3 delas referentes as arquiteturas do Mobile Media.

As arquiteturas AGM são um grupo de softwares que possuem todo o mesmo objetivo, uma tentativa de reaproveitamento da mesma arquitetura base como forma de experimentação e aprendizado desses tipos de conceitos. Para geração de games do estilo arcade utilizando-se do mesmo grupo de componentes entre si, porém podendo-se alterar caso necessário o valor dos seus atributos, número de atributos e a quantidade de métodos a serem utilizados no próximo game a ser desenvolvido. Possuindo até o momento 4 jogos desenvolvidos (*bowling*, *Pong*, *Brickles* e *Pinball*).

Já as arquiteturas Mobile Media foram desenvolvidas como uma solução para criação e manipulação dos diferentes tipos de mídias existentes para rodar em um ambiente mobile. Servindo atualmente como forma de estudo acadêmico por não haver mais a necessidade do desenvolvimento deste tipo de aplicação nos celulares modernos.

A fim de realizar o estudo das arquiteturas, todas elas são apresentadas no formato XML para serem analisadas com a ferramenta Papyrus, presente na loja virtual de extensões do Eclipse. Sendo a ferramenta Papyrus um componente desenvolvido pelo Eclipse Corporation [Eclipse Foundation 2015] para produzir um ambiente de

escala industrial para visualização e edição de forma gráfica de todos os diagramas pertencentes ao UML2.

Figura 3. Classe Player da arquitetura AGM 2.

4. Resultados

Nesta seção são apresentados os resultados obtidos a partir da realização desta pesquisa. Na Secção 4.1 é mostrado o catálogo criado para a realização da análise e a identificação de todos os problemas arquiteturais estudados nesta pesquisa. Na Seção 4.2 são passados os resultados obtidos a partir das análises das arquiteturas e possíveis soluções para cada um dos problemas encontrados.

4.1 Catálogo

A fim de proporcionar uma melhor análise para os possíveis problemas arquiteturais presente nas arquiteturas analisadas foi gerado um catálogo contendo 26 diferentes problemas arquiteturais encontrados na literatura e a partir do catálogo é apresentado se este problema será ou não analisados neste estudo. A classificação dos problemas no catalogo os divide em 3 tipos: os problemas que realmente serão analisados, os problemas que serão cogitados a análise deles e os problemas que não serão analisados

em detrimento de limitações presentes nas arquiteturas analisadas. A Tabela 1 apresenta os problemas encontrados dentro dessas categorias.

Nas seções a seguir, são apresentados todos os problemas encontrados a partir da revisão bibliográfica, suas definições e a maneira como podem ser encontrados e analisados em diferentes arquiteturas de software.

Tabela 1. Problemas arquiteturais analisados neste trabalho.

Classificação	Problemas Arquiteturais
Problemas a Serem Analisados	*Link Overload* *Concern Overload* *Unused Interface* *Large Class* *Primitive Obsession* *Duplicated Code* *Duplicate Component Functionality* *Unused Brick* *Sloppy Delegation* *Dependency Cycle* *Connector Envy*
Problemas a Serem Cogitados para a Análise	*Unplanned and Uncontrolled Solution* *Lazy Class* *Scattered Functionality* *Alternative Classes with Different Interfaces* *Connector Chain Overload*
Problemas a Não Serem Analisados	*OddBall Solution* *Switch Statements* *Indecent Exposure* *Conditional Complexity* *Brick Functionality Overload* *Long Method* *Scattered Parasitic Functionality* *Ambiguous Interface* *Combinatorial Explosion* *duplicateExtraneous Connector*

4.1.1. Link Overload

Descrito como sendo um problema arquitetural no qual um dos componentes possui muitas dependências a outros componentes do sistema. Afetando diretamente o cuidado na separação das funcionalidades dentro de um sistema e também na manutenção desses componentes [Garcia 2014]. Podendo ser facilmente percebida em uma análise ao identificar uma classe que possui uma quantidade ínfima de atributos ou mesmo métodos próprios, mas está totalmente interligada por meio de dependências a outros demais componentes.

4.1.2. Large Class

Fowler (1999) indica que caso a classe possui muitas variáveis em seu código fonte é provável que esteja sendo atribuídas funções demais a ela. Isso se deve porque o problema de *Large Class* normalmente representa uma classe que contém responsabilidades demais atribuídas a elas. Sua identificação é vista quando uma classe possui um número excessivo tanto de atributos como métodos que acabam por prejudicar futuramente a sua manutenção ou mesmo a inserção de novas

funcionalidades a ela. Para fazer a refatoração desta classe é necessário dividi-la entre várias outras classes do sistema. Onde cada uma dessas novas classes será responsável por umas das responsabilidades que esta classe anterior possuía, a fim de permitir um melhor entendimento do código por parte da equipe e uma melhoria também na manutenção futura dessas funcionalidades.

4.1.3. Primitive Obsession

Os tipos primitivos, que incluem os booleanos, *strings*, inteiros, *doubles*, vetores e outros elementos de baixo nível em uma linguagem são genéricos por que muitos usuários a utilizam. As classes em contrapartida podem ser tão específicas quanto necessário, dado que você as cria para propósitos específicos. Descrita por Fowler (1999) como sendo um problema presente em códigos que utilizam um número excessivo de tipos primitivos ao invés da utilização de conceitos mais avançados e recomendados como as classes da orientação a objeto.

4.1.4. Duplicated Code

Dentre os problemas ele é o mais difundido e possui um forte efeito sobre o software. Podendo ser apresentado de forma tanto implícita como sutil [Kerievsky 2004]. Na forma implícita existe um código idêntico, já na sutil existem estruturas ou passos do processamento aparentemente diferentes, mas são essencialmente idênticos entre si. Sua identificação está presente em classes ou mesmo funcionalidades que apresentam algum tipo de duplicação não necessária (e normalmente não intencional) dentro de um mesmo sistema.

4.1.5. Unused Interface

Resultante de uma interface dentro de um sistema que não possui ligamentos com as demais classes do sistema. Este problema é verificado quando uma determinada classe de interface é criada no sistema, mas não aparenta estar sendo utilizada por nenhuma outra classe [Garcia 2014]. Um dos motivos mais prováveis para isso pode ser o resultado de mudanças realizadas no escopo ou mesmo no andamento do projeto ou resultado de uma refatoração mal feita no sistema.

4.1.6. Concern OverLoad

Ocorre no momento em que um mesmo componente (seja um método, classe ou conector) é responsável por muitas funcionalidades de uma vez só. Este tipo de problema afeta não só a qualidade do seu sistema, mas também quebra o princípio de responsabilidade única [Garcia 2014]. Indicando que cada componente dentro de um sistema deverá possuir apenas uma única determinada responsabilidade para si.

A maneira de identificar este problema acaba sendo uma análise mais sugestiva sobre quais seriam os papéis e funcionalidades que um determinado componente apresenta dentro do sistema.

4.1.7. Duplicate Component Functionality

Ocorre quando é apresentado um componente que possui as mesmas funcionalidades de outro já existente no sistema. Esse tipo de problema se mostra nocivo para a qualidade do produto por apresentar duplicações em seu código fonte [Garcia 2014]. É encontrada no sistema ao identificar um componente que realiza a mesma funcionalidade que outro

componente, mudando-se desde o nome do componente ou apresentando até uma outra implementação mas cujos resultados se mostram iguais ou muito similares.

4.1.8. Dependency Cycle

Ocorre quando uns grupos de componentes se ligam entre si como se fosse um ciclo, na qual uma mudança ocorrida em um determinado componente do grupo afeta diretamente os demais componentes presentes neste grupo [Garcia 2014]. É percebido no ponto em que 2 ou mais componentes ligam-se entre si na forma de uma dependência um do outro. Afeta principalmente a manutenção do sistema, pois afeta diretamente o comportamento dos componentes que dependem deste determinado componente.

4.1.9. Unused Brick

Ocorre quando todas as interfaces de um determinado grupo de componentes não são utilizadas de forma nenhuma. Similar a problema de *Unused Interface*, resultando em um sistema mais complicado de fornecer manutenção graças a esta complexidade desnecessária [Garcia 2014]. Sua detecção é apresentada em um grupo de interfaces que não apresentam nenhuma ligação ou dependência com outra componente do sistema, sendo bem visível ao analisar diagramas como o diagrama de classes do projeto.

4.1.10. Sloppy Delegation

Ocorre quando um componente delega uma quantidade pequena de outras funcionalidades a outro componente, sendo que ele poderia muito bem ter delegado as demais funcionalidades a si mesmo [Garcia 2014]. É mostrado dentro de um sistema um componente relativamente pequeno ou mesmo simples na qual não existe a necessidade para que ela seja feita de forma separada a outra componente dentro do sistema.

4.1.11. Lazy Class

Descrita por Fowler (1999) como sendo: “Uma classe que não está fazendo o suficiente para pagar por si própria deve ser eliminada”. Resultado normalmente de estruturações erradas na hora de criar classes dentro de um sistema. São apresentados por componentes que normalmente estão ligados a outro componente por meio de uma dependência, mas não apresentam a necessidade de estar separado a outro componente dentro do sistema.

4.1.12. Long Method

Representado por Fowler (1999), como sendo prejudiciais se comparados a vários métodos curtos ou pequenos, tendo como principal desvantagem o compartilhamento da lógica. Dois métodos diferentes que apresentem esse problema podem apresentar códigos duplicados, que caso fossem divididos entre pequenos métodos poderiam ser achados maneiras de compartilhar da mesma lógica entre eles. Este problema infelizmente não foi possível ser feita sua análise, por não possuir acesso ao código fonte das arquiteturas que serão analisadas. Pois com apenas a arquitetura fechada em um formato de um diagrama UML, não é possível verificar e analisar o tamanho do método a ser verificado.

4.1.13. Ambiguous Interface

São interfaces que geram apenas um único meio de acesso aos diferentes componentes e funcionalidades do sistema [Garcia 2010]. Isso ocorre justamente quando as interações ou acessos as funções não estão explicitamente modelados e apresentados na interface.

Ocorre principalmente em sistemas que utilizam *ints* ou mesmo *strings* como entrada para acesso às funções do sistema. Este problema infelizmente não foi possível ser feita sua análise, por não possuir acesso ao código fonte das arquiteturas que serão analisadas. Pois foi apenas disponível de uma arquitetura fechada em um formato de um diagrama UML.

4.1.14. Unplanned and Uncontrolled Solution

Acontece quando códigos e/ou dados utilizados para implementar uma responsabilidade tornam-se espalhados entre diversas classes [Kerievsky 2004]. Esse problema é normalmente resultante da adição de uma nova funcionalidade ao sistema rapidamente, sem gastar tempo suficiente simplificando ou mesmo consolidando o projeto para melhor acomodar a nova funcionalidade. Percebido na funcionalidade de um componente que necessita de muitas importações ou dependências de atributos de outras componentes para ser feita ou executada.

4.1.15. Alternative Classes with Different Interfaces

Este problema de codificação retratado por Fowler (1999) ocorre quando as interfaces de duas classes são diferentes, mas suas classes são muito parecidas. Se você conseguir encontrar essas similaridades entre as classes, poderá refatorar com frequências ambas para fazer que compartilhem a mesma interface. Caso não possua acesso ao código fonte é possível utilizar um *Adapter* unificando suas interfaces e direcionando cada nova requisição ou importação dessa interface para a interface adequada.

4.1.16. Connector Chain OverLoad

Ocorre quando existe um grande *link* de conexão entre diversos conectores, que excede o número máximo de conectores que deveriam estar sendo conectados entre si para uma única funcionalidade dentro de um sistema [Garcia 2014]. Este tipo de problema é principalmente nocivo a manutenção do sistema, pois é muito provável ocorrer incompatibilidades entre os diferentes conectores presentes dentro de um mesmo link.

4.1.17. Indecent Exposure

Este problema ocorre quando métodos ou classes que não deveriam ser visíveis aos usuários são publicamente visíveis a eles [Parnas 1974]. Expor tal código significa que os clientes conhecem código que não é importante ou que é apenas indiretamente importante. Isso contribui para aumentar a complexidade do projeto. Esse tipo de problema não pode ser analisado neste trabalho, pois seria necessário acesso ao código fonte do projeto para assim analisar quais classes ou métodos que seriam relevantes para serem visualizadas pelos seus usuários, assim como definir a relevância ou não desse tipo de código a ser implementado ao usuário.

4.1.18. Switch Statements

As *Switch Statements* são classificados como sendo uma cadeia muito longo de decisões presentes (na maioria dos casos) dentro de métodos de uma classe para servir como maneiras de validação destas classes [Kerievsky 2004]. Não se mostrando nocivas

normalmente, porém podem se tornar caso elas sejam muito rígidas ou complicadas para permitir aos seus desenvolvedores fazerem manutenção do seu código.

Não foi possível ser feita a análise desse problema arquitetural, por não possuir acesso ao código fonte da arquitetura. Pois neste tipo de problema busca-se encontrar uma lista longa de decisões presentes em algum método nas classes.

4.1.19. OddBall Solution

Ocorre quando um problema é solucionado de uma maneira ao longo do sistema, e novamente é solucionado este problema, porém de outra forma, sendo assim umas delas uma solução não usual ou uma solução inconsistente [Kerievsky 2004]. A presença deste problema indica uma sutil duplicação de código.

Não foi possível ser feita a análise desse problema arquitetural, por não possuir acesso ao código fonte da arquitetura. Pois não é possível analisar como os métodos e classes são codificados, pois este tipo de problema buscar soluções codificados de forma diferente, porém tendo o mesmo resultado.

4.1.20. Combinatorial Explosion

Esse problema se baseia em uma sutil forma de duplicação. Existindo quando inúmeros trechos de código fazem o mesmo tipo de ação utilizando-se de diferentes tipos ou quantidades de dados ou objetos [Kerievsky 2004]. Por exemplo, imagine que existe uma quantidade numerosa de métodos em uma classe para realização de consultas. Onde cada método realiza uma consulta a partir de um número de condições e dados específicos. Desta forma quanto mais específica são essas consultas para as quais deverão ser oferecidas o suporte, mais métodos de consulta deverão ser criados. Desta forma você terá uma infinidade de métodos para a realização dessas consultas. Tendo também assim uma linguagem de consulta implícita.

Não foi possível ser feita a análise desse problema arquitetural, por não possuir acesso ao código fonte da arquitetura. Pois este problema se baseia na existência numerosa de métodos (sendo na maioria dos casos muito específicos) sem que haja realmente a necessidade de tantos métodos diferentes presentes em uma única classe.

4.1.21. Conditional Complexity

Uma lógica condicional no início ela é simples e possui fácil entendimento por conter poucas linhas de código [Kerievsky 2004]. Infelizmente ela raramente envelhece bem. Por exemplo, você implementa diversas novas funcionalidades e subitamente sua lógica condicional torna-se extensa e complicada.

Não foi possível ser feita a análise desse problema arquitetural, por não possuir acesso ao código fonte da arquitetura. Neste tipo de problema é necessário verificar e analisar as diferentes decisões presentes dentro de um método em uma classe do sistema e a necessidade ou não deste número de decisões presentes neste método.

4.1.22. Extraneous Connector

Ocorre quando 2 conectores de diferentes tipos são usados para interligar um par de componentes [Garcia 2010]. Não foi possível ser feita a análise deste problema, pois para ser realizado seria necessário seu código fonte para ser analisado as tecnologias utilizadas e quais seriam os tipos de conectores utilizados neles.

4.1.23. Connector Envy

Ocorre quando uma funcionalidade é implementada no sistema ao invés de delegá-la a um conector [Garcia 2010]. Conectores são classes ou mesmo diferentes sistemas que servem de intermédio entre as classes, para desde tratamento de uma determinada entrada para gerar uma nova saída, ou mesmo para interligação ou facilitação na hora da utilização de novas tecnologias. Esse tipo de prática interfere justamente no entendimento e no reuso dessas funcionalidades. Quando delegadas a um conector é mais fácil não só o entendimento da funcionalidade como também o reuso entre diferentes aplicações, porque estas funcionalidades estarão separadas do restante do código.

4.1.24. Scattered Functionality

Este tipo de problema acontece quando uma funcionalidade ou componente está espalhada entre diferentes classes do sistema. Por não estar alocado em apenas um único componente, esta funcionalidade irá apresentar uma maior dificuldade principalmente na questão de manutenção [Garcia 2010]. Sendo necessário alterar muito mais seu código fonte para em uma futura manutenção (principalmente resultante das dependências das classes e do código duplicado) do que se estivesse inicialmente alocado em um único componente esta funcionalidade. Um dos exemplos disso está na atribuição de métodos para realização das ações de um CRUD espalhadas entre vários métodos ao invés de uma classe específica (DAO).

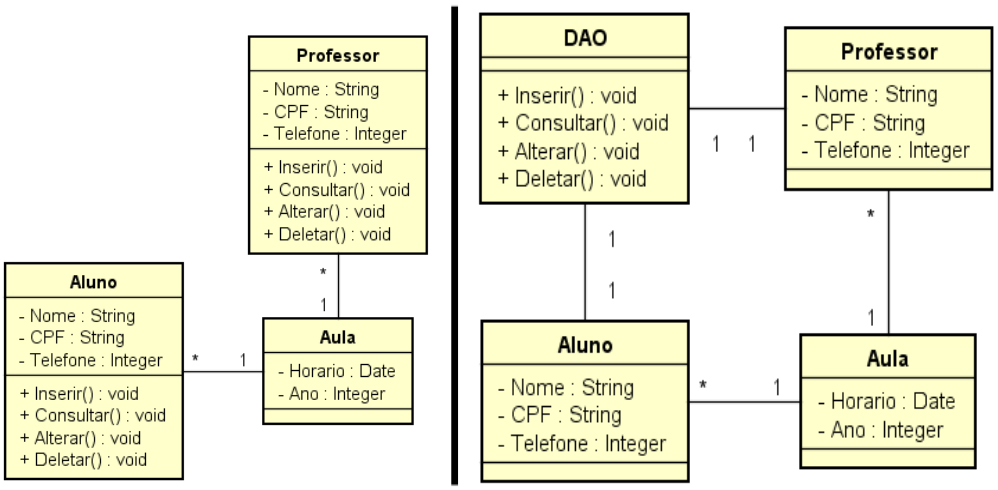

Figura 4. Exemplo de um problema de *Scattered Functionality*, ao qual as 3 classes criadas inicialmente (Aluno, Aula e Professor) são divididas entre outras 4 classes a fim de definir melhor as responsabilidades de cada um das diferentes classes.

4.1.25. Brick Functionality Overload

Ocorre quando um componente apresenta um número excessivo de funcionalidades presentes em seu código fonte [Garcia 2014]. Esse tipo de problema agride diretamente o princípio de separação e isolamento dos diferentes tipos de preocupações que cada um dos componentes devem comportar.

Não foi possível ser feita a análise desse problema arquitetural, por não possuir acesso ao código fonte da arquitetura. Pois este problema é causado quando um determinado componente apresenta um número grande de funcionalidades descritas

dentro do seu código fonte, não sendo possível ser identificado apenas pelo tamanho dos seus métodos ou mesmo na sua quantidade, mas a maneira e o comportamento destes métodos e demais componentes dentro deste componente.

4.1.26. Scattered Parasitic Functionality

Descrito como um sistema ao quais múltiplos componentes são responsáveis por realizar as mesmas funcionalidades importantes do sistema, enquanto outros demais componentes são responsáveis por adicionar ou mesmo modificar esses outros componentes presentes nessas múltiplas classes [Garcia 2014].

Não foi possível ser feita a análise deste problema arquitetural, por não possuir acesso ao código fonte da arquitetura. Pois para ser analisado seria necessário entender como os métodos funcionam, além de verificar suas respectivas entradas e saídas para assim perceber ou mesmo descobrir se houve uma alteração ou modificação de um componente para ser utilizado em outras demais classes. Sendo o ideal as classes se modificarem para se adequar a esses componentes e não o inverso.

4.2. Avaliação de Problemas Arquiteturais

Conforme descrito na Seção 3.2, neste estudo realizou-se um experimento para a identificação e avaliação de problemas arquiteturais nos sistemas Arcade Game Maker e Mobile Media. Para registro de cada uma das diferentes arquiteturas e os seus problemas encontrados foi criado um documento para expor os resultados desta pesquisa. Em cada um destes documentos é mostrado o problema encontrado, sua definição, o meio para ser feito a refatoração deste problema e a imagem do problema encontrado. Todas os relatórios das análises realizadas neste estudo estão em um apêndice online deste trabalho[1]. As seções a seguir apresentam, de forma resumida, as principais observações coletadas durante a avaliação dos problemas arquiteturais.

4.2.1. Arquitetura AGM Original

Nesta arquitetura foram encontrados três tipos de problemas diferentes: *Concern OverLoad, Unused Interface* e *Large Class*.

Unused Interface: este problema ocorre quando uma classe de interface é criada no sistema, mas ela não está sendo utilizada em lugar nenhum. Sendo o principal motivo para esse problema, mudanças existentes no escopo do projeto. Este problema foi encontrado em 2 classes de interface no sistema: *Point* e *Display*. Para realizar a refatoração deste problema, verifique primeiro se a classe de interface ou alguns dos seus métodos serão utilizados em alguma implementação futura do sistema. Caso não sejam exclua as classes do projeto.

Large Class: este problema ocorre no momento em que uma classe possui um número excessivo de métodos e/ou atributos no corpo da classe. Sendo identificados 2 classe com esse problema: GameBoard e Player. Na classe GameBoard apresenta um número excessivo de métodos, já a classe Player apresenta um número excessivo de atributos no corpo de sua classe. Para fazer a refatoração desta classe é necessário

[1]Pasta no Google Drive com os relatórios das análises das arquiteturas realizados neste estudo. Disponível em: https://drive.google.com/open?id=0B-WdRhZAccyFcXZVNklIUHpnYkU

dividi-la entre várias outras classes. Onde cada uma das classes será responsável por uma das responsabilidades que esta classe anterior possuía.

Concern Overload: ocorre quando um mesmo componente (podendo ser uma classe, método ou conector) é responsável por muitas funcionalidades de uma vez só. Este tipo de problema afeta não só a qualidade do seu sistema, mas também quebra o princípio de responsabilidade única. A maneira de realizar sua refatoração é identificar os papéis e funcionalidades que determinado componente tenta alcançar. Caso essas funcionalidades estejam muito intrínsecas entre si e dificulte sua divisão. Quebra esse único componente entre vários, e para continuar sua unidade entre eles, e realize chamadas entre esses componentes.

4.2.2. Arquitetura AGM 2

Nesta arquitetura foram encontrados três tipos de problemas diferentes na arquitetura: *Link OverLoad*, *Unused Interface* e *Large Class*.

Link Overload: Este problema foi encontrado dentro da classe GameCtrl, por apresentar apenas 1 único atributo próprio dentro de sua classe, sendo necessário a dependência e implementação de 2 outras classes de interface para que seja possível realizar alguma ação dentro de seu sistema. Ela afeta diretamente a qualidade do software como um todo, pois neste exemplo não é necessário ter sido criada esta classe GameCtrl, sendo muito mais aconselhável as funcionalidades atribuídas nas classes de interface serem implementadas nas classes que ocorrem a dependência da classe GameCtrl.

Unused Interface: Este problema ocorreu em diferentes classes de interface existentes dentro do sistema. Este problema ocorre no momento em que uma classe de interface é criada no sistema, porém ele não está sendo utilizada em nenhum lugar. O principal motivo para isso são mudanças presentes no escopo do projeto. Para realizar a refatoração, primeiro verifique se esta classe vai ser ou não utilizada futuramente ou se alguns dos métodos presentes na classe serão utilizados futuramente. Caso apenas um dos métodos será utilizado, exclua os demais métodos do sistema. Caso a classe não seja utilizada de forma nenhuma, exclua a classe de interface.

Large Class: Este problema é recorrente em classes que possui um número excessivo de atributos e/ou métodos no corpo de sua classe. Comprometendo assim a manutenção desta classe para tanto adição como alteração e/ou manutenção de uma funcionalidade nova no sistema. Nesta arquitetura em específico foram identificados 2 exemplos de classes com este problema arquitetural, a classe *Player* e a classe *GameBoard*. A classe Player apresenta um número excessivo de atributos no seu código fonte, já a classe *GameBoard* apresenta um número excessivo de métodos. A melhor solução está na separação das responsabilidades antes atribuídas nas classes *Player* e *GameBoard* entre várias outras classes para assim melhorar a legibilidade de seu código fonte e permitir uma manutenção menos trabalhosa.

4.2.3. Arquitetura AGM 3

Nesta arquitetura foram encontrados dois tipos de problemas diferentes em sua arquitetura: *Unused Interface* e *Sloppy Delegation*.

Unused Interface: Este problema ocorre quando é criado no sistema uma classe de interface, porém ela não está sendo utilizada em nenhum lugar do projeto. Este tipo

de problema está presente em inúmeras classes dentro do sistema, sendo resultado normalmente de mudanças presentes no escopo do projeto. Verifique se esta classe de interface será utilizada futuramente no projeto. Caso não seja, a exclua do projeto.

Sloppy Delegation: Este problema ocorre quando um componente dentro do sistema delega uma quantidade ínfima de funcionalidades (sendo métodos ou atributos) a outro componente, sendo que ela poderia muito bem ter delegado essas funções para si próprio. Este é um problema relacionado diretamente às responsabilidades que uma classe deve possuir para ser relevante dentro do sistema. Ao qual a classe com uma quantidade ínfima de funcionalidades não apresenta uma necessidade de estar separada a outra classe. Sendo neste caso o aconselhável a transferências destes atributos e métodos para a classe que faz esta dependência com ela, e após isso excluí-la do projeto. Foi encontrado 2 classes que apresentam este problema: Class1749, já a outra é uma classe de interface, porém ela apresenta apenas um único método em código fonte e esta se relacionando com apenas 1 única classe (AnimationLoopMgr).

4.2.4. Arquitetura Mobile Media 1

Nesta arquitetura foram encontrados dois tipos de problemas diferentes em sua arquitetura: *Unused Interface* e *Large Class*.

Unused Interface: Este problema ocorre no momento em que uma classe de interface é criada, mas ela não está sendo utilizada em nenhum lugar do sistema. Ocorre normalmente devido a mudanças presentes no escopo do projeto. Estando presente em inúmeras classes dentro do sistema. Para fazer a refatoração desta classe é necessário verificar se esta classe de interface será ou não utilizada futuramente. Caso não seja, a exclua do projeto.

Large Class: Resultado de uma classe que possui um número excessivo de métodos e/ou atributos em código fonte. Comprometendo assim a manutenção desta classe para tanto adição como alteração e/ou manutenção de uma funcionalidade nova no sistema. Foi identificado nesta arquitetura a classe de interface *IManageMedia* possuir este problema arquitetural. Ao qual esta classe possui um número excessivo de métodos em seu código fonte, e por ser uma classe de interface não apresenta nenhum atributo em sua classe. A maneira mais aconselhável de realizar a refatoração desta classe é dividi-la em entre várias classes, sendo que cada uma delas fica responsável por uma ou mais responsabilidades que a classe *IManageMedia* possuía.

4.2.5. Arquitetura Mobile Media 2

Nesta arquitetura foram encontrados dois tipos de problemas diferentes em sua arquitetura: *Unused Interface* e *Lazy Class*.

Unused Interface: Este problema ocorre no momento em que uma classe de interface é criada, mas ela não está sendo utilizada em nenhum lugar do sistema. Ocorre normalmente devido a mudanças presentes dentro do escopo do projeto. Tendo este problema aparecido em diferentes classes dentro desta arquitetura. A solução para realizar a refatoração é verificar se estas arquiteturas realmente não serão utilizadas no projeto, caso não sejam mesmo utilizadas, exclua elas do projeto.

Lazy Class: Este problema é representado por um componente no sistema que possui uma dependência a outro componente, mas não possui a necessidade de estar separado a esse primeiro. Sendo uma classe que não apresenta funcionalidades que

justifique sua separação da classe principal. Sua refatoração é feita extraindo os métodos e atributos na classe que apresenta este problema para a classe que ela faz a dependência e após isso exclua a classe com o problema de *Lazy Class*.

4.2.6. Arquitetura Mobile Media 3

Nesta arquitetura foi encontrado apenas um tipo de problema em sua arquitetura: *Unused Interface*.

Unused Interface: Este problema ocorre no momento em que uma classe de interface é criada no projeto, mas não está sendo utilizada em lugar nenhum. Ocorre normalmente devido a mudanças presentes no escopo do projeto. Tendo este problema aparecido em diferentes classes dentro desta arquitetura. A solução para realizar a refatoração desta classe é primeiro, verifique se esta classe poderá ser ou não utilizada futuramente no sistema. Caso não seja utilizado, exclua esta classe do projeto.

5. Conclusão

Este estudo buscou realizar uma análise qualitativa das arquiteturas geradas pelo software MO4PLA, a fim de encontrar, analisar e se possível propor heurísticas para resolução dos problemas encontrados. Sendo realizado as análises das arquiteturas, a partir da ferramenta de criação e manipulação fornecida pelo Eclipse: Papyrus.

Onde esta ferramenta possibilita a visualização e manipulação de todos os diagramas pertencentes ao UML2. Sendo as arquiteturas cedidas para este estudo no formato XML. Para cada classe que apresentasse um problema arquitetural, foi apresentado: o nome da classe que apresenta o problema, o tipo do problema arquitetural, a definição deste problema e um meio para gerar realizar a refatoração desta classe. Sendo encontrado um total de 6 problemas existentes nas arquiteturas analisadas: *Unused Interface, Lazy Class, Large Class, Sloppy Delegation, Link Overload e Concern Overload.*

5.1. Heurística Proposta

A partir deste trabalho foi possível identificar uma possível heurística para a solução do problema de *Unused Interface*, problema resultante de uma classe de interface criada no projeto, mas que não está sendo utilizada por nenhuma outra classe. Este tipo de problema é muito difícil de ser analisado dentro do código fonte de um projeto de software, por não apresentar nenhum tipo de aviso ou alerta que esta classe não apresenta um relacionamento com nenhuma outra classe.

Porém este tipo de problema é facilmente verificado e analisado nos diagramas realizados neste estudo, pois a partir deles é possível ver todos os relacionamentos e dependências que uma classe possui. Sendo uma classe de interface que apresenta o problema de *Unused Interface* fica isolada das demais classes do sistema. Abaixo estão representadas duas classes de interface retiradas dos diagramas em formato XML para o estudo. A qual a primeira classe por não se relacionar a nenhuma outra classe do projeto, acaba a isolando das demais classes do projeto, possuindo assim o problema de Unused Interface por esta característica. Já a segunda figura apresenta uma classe de interface, *IAlbumGUI*, sendo utilizada pela classe *AlbumGUI*, configurando um caso de uma classe de interface que não apresenta o problema de Unused Interface.

A proposta deste trabalho de heurística para solução deste problema está no momento de codificação do código fonte do projeto. No caso de importações de demais

classe dentro do projeto é possível perceber se essa importação está ou não sendo utilizada, pois ela apresenta um risco amarelado em baixo da importação (no caso da IDE Eclipse), sendo facilmente percebido pelo programador a falta de uso desta importação. Para classe de interface neste caso seria apresentado um esquema parecido. Enquanto esta classe não estiver sendo utilizada por nenhuma outra classe, deverá ser feito um risco abaixo da classe de interface nos pacotes de código fonte do projeto. Na figura abaixo são representadas 2 classes de interface presentes nas arquiteturas deste estudo. Sendo a primeira, por não se relacionar a nenhuma outra classe, possuir o problema arquitetural *Unused Interface*. Já a segunda classe de interface possui um relacionamento na classe AlbumGUI, configurando uma classe de interface que não apresenta este tipo de problema.

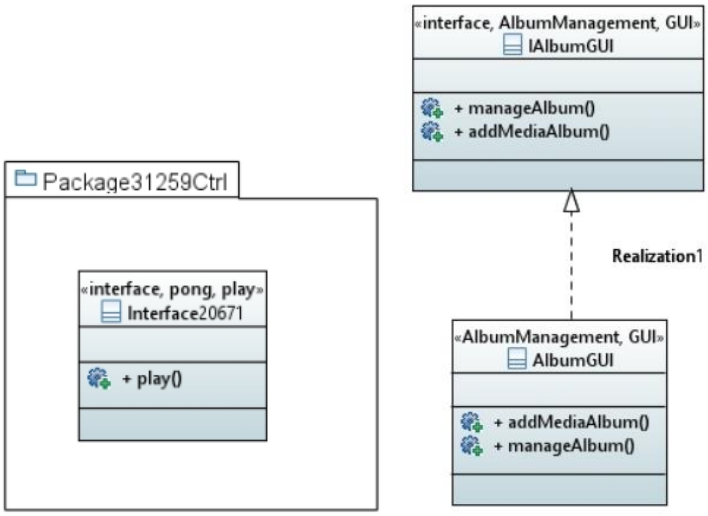

Figura 5. Representação de 2 classe de interface do projeto, sendo que a primeira delas apresenta o problema de *Unused Interface*, já a segunda não.

5.2. Divulgação e Trabalhos Futuros

Resultados parciais deste trabalho foram publicados e apresentados no I Congresso Internacional de Ciência, Tecnologia e Inovação da Unipar [Johansson, 2016]. Os resultados finais deste trabalho serão submetidos para publicação na forma de artigo completo.

Para a realização de trabalhos futuros seria interessante a utilização e estudo de outros projetos arquiteturais, para assim fazer uma espécie de parâmetro de comparação da qualidade geral das arquiteturas de software presentes na comunidade científica. Outro desdobramento existente para este trabalho seria os estudos dos *Design Smells*, problemas decorrentes de implementação de solução para o design das páginas do sistema, que afetam não só o design da página como na manutenção da mesma. Assim como um estudo mais aprofundado de demais problemas existentes nas arquiteturas para serem acrescentados no catálogo de problemas arquiteturais, para ser realizada uma análise mais completa e detalhada de diferentes problemas presentes nas arquiteturas estudadas.

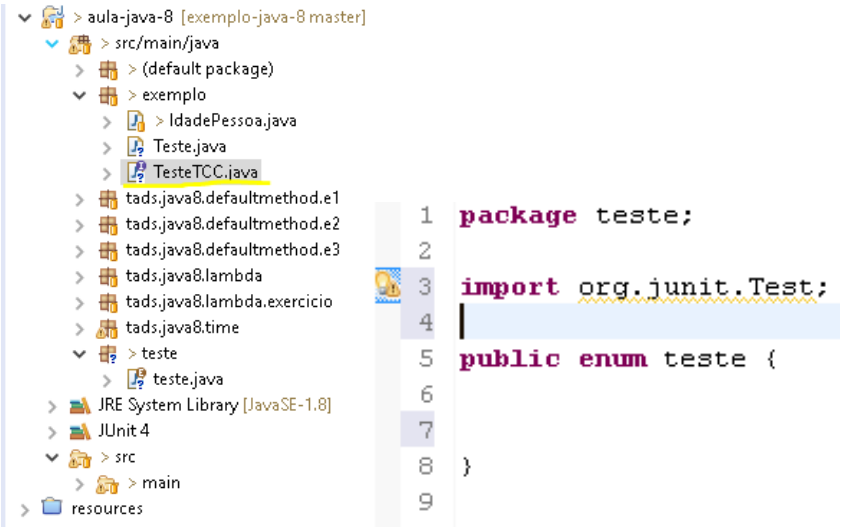

Figura 6. Exemplo de uma possível solução por meio de heurística.

Referências

Booch, Grady. *et al.* (2014) "Object-Oriented Analysis and Design with Applications", 3. ed., Addison Wesley, Redwood City, CA, EUA.

Colanzi, Thelma Elita. (2014) "Uma abordagem de otimização multiobjetivo para projeto arquitetural de linha de produto de software", http://dspace.c3sl.ufpr.br:8080/dspace/bitstream/handle/1884/36061/R%20-%20T%20-%20THELMA%20ELITA%20COLANZI.pdf?sequence=1, Março.

Ganesh, S. G., Sharma, T. e Suryanarayana, G. (2013) "Towards a Principle-based Classification of Structural Design Smells", Journal of Object Technology, 12(2), 1-1.

Garcia, Joshua, *et al.* (2010) "Identifying architectural bad smells", http://softarch.usc.edu/~josh/pubs/csmr_2009.pdf, Março.

Féderle, Édipo Luis. (2014) "Uma ferramenta de apoio ao projeto arquitetural de linha de produto de software baseado em busca", 130 fls, Dissertação (Mestrado em Informática)- Universidade Federal do Paraná, Curitiba.

Fowler, Martin. (1999) "Refactoring: Improving the Design of Existing Code", Addison-Wesley Professional.

Johansson, E. S.; Kamakawa, H. T.; Oliveira, F. W. C.; Flôr, D. E.; Zavan, A. R.; Oizumi, W. N. (2016) "Um Estudo Exploratório Sobre a Identificação de Problemas em Arquiteturas de Software". I Congresso Internacional de Ciência, Tecnologia e Inovação da Unipar.

Oizumi, Willian Nalepa, *et al.* (2012) "On the proactive design of product-line architectures with aspects: an exploratory study", In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, p. 273-278.

OMG. (2016) "XML Metadata Interchange (XMI)", http://www.omg.org/spec/XMI/, Maio.

Rumbaugh, James, Jacobson, Ivar, e Booch, Grady. (2004) "Unified Modeling Language Reference Manual", The. Pearson Higher Education.

SEI. (2016) “Arcade Game Maker Pedagogical Product Line”, http://www.sei.cmu.edu/productlines/ppl/, Maio.

Parnas, David (1972) “On the Criteria To Be Used in Decomposing Systems into Modules", Wikipedia:Communications of the ACM286.

Garcia, Joshua, *et al.* (2014) “A unified Framework for Studying Architectural Decay of Software Systems”, Dissertação (Doutorado em Ciência da Computação) – Universidade do Sul da California, California.

Eclipse Foundation (2015) ”Papyrus Modeling Environment”, https://eclipse.org/papyrus/, Dezembro