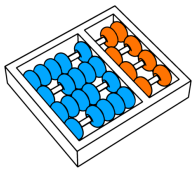
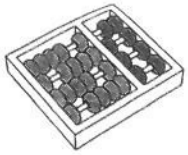


Andre Petris Esteve

“Evolução e recuperação de arquiteturas de software baseadas em componentes”

CAMPINAS
2013



Universidade Estadual de Campinas
Instituto de Computação

Andre Petris Esteve

"Evolução e recuperação de arquiteturas de software baseadas em componentes"

Orientador(a): **Prof. Dra. Cecília Mary Fisher Rubira**
Instituto de Computação - UNICAMP

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão da Dissertação apresentada à banca examinadora por Andre Petris Esteve, sob orientação de Prof. Dra. Cecília Mary Fisher Rubira
Instituto de Computação - UNICAMP.

Cecília Mary Fischer Rubira
Assinatura do Orientador(a)

CAMPINAS
2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

Esteve, Andre Petris, 1989-
Es85e Evolução e recuperação de arquiteturas de software baseadas em componentes / Andre Petris Esteve. – Campinas, SP : [s.n.], 2013.

Orientador: Cecília Mary Fischer Rubira.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Engenharia de software. 2. Software - Arquitetura. I. Rubira, Cecília Mary Fischer,1964-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Component based software architecture recovery and evolution
Palavras-chave em inglês:
Software engineering
Software - Architecture
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Cecília Mary Fischer Rubira [Orientador]
Patrick Henrique da Silva Brito
Ariadne Maria Brito Rizzoni Carvalho
Data de defesa: 13-12-2013
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 13 de dezembro de 2013, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Patrick Herique da Silva Brito
IC / UFAL

Profª. Drª. Ariadne Maria Brito Rizzoni Carvalho
IC / UNICAMP

Profª. Drª. Cecilia Mary Fischer Rubira
IC / UNICAMP

Abstract

Software systems must evolve so to fit people's needs. That is a natural process, in which people and societies change, new necessities arise and existing concepts are replaced by modern ones, thus forcing the software to fit into the new picture, adapting itself to the changes.

As software systems are modified, they age. *Software ageing* is the term given to the diminishing of a software's life spawn due to the inherently complexity of its creation, maintenance and understanding processes. As it evolves, the system tends to grow more intricate, its source code tends to become less expressive as it needs to be adapted to new requirements, run with a better performance or simply it has become so immense and complex that understanding it is, by itself, a challenging task.

Eventually, maintaining and supporting a software system will turn out to be even more expensive than rewriting it from scratch. From that point onwards, the original piece of software is abandoned, having its life spawn reached an end. The aged software is replaced by a new and modern implementation, one that fulfils the just placed requirements. However this brand new software piece will share the same fate as its predecessors, it will age and will eventually be replaced.

This work proposes a solution, composed by method and computer-aided tools, for managing software architecture evolution, which is a fundamental piece in the system's longevity. The solution's method aims to identify software architecture ageing problems ahead of time, so their impact can be adequately mitigated or even completely avoided, thus extending the software's life spawn.

So to allow the practical use of the method, as part of the proposed solution, a tool was implemented to automate most of the method's activities. Through automation, the tool is capable of reducing the human error associated to the processes execution, thus yielding high efficiency. By analyzing case studies, it is possible to verify that, when applied, the solution is capable to guide the software architect to uncover software ageing problems on the system under investigation. Through the computational aid offered by the solution, the architect is able to act upon the newly discovered issues, with undemanding time and effort, thus resolving or mitigating the problems that arise with software ageing.

Resumo

Os sistemas de software precisam evoluir para atender as necessidades das pessoas. Isso é natural, já que as pessoas, as sociedades mudam, novas necessidades surgem, conceitos alteram-se e o software deve adequar-se às mudanças.

Ao passo que alterações são feitas em sistemas de softwares, eles começam a envelhecer. O *envelhecimento de software* é o termo dado a redução da vida útil de um software devido à natureza da complexidade de seus processos de criação, manutenção e entendimento. Conforme um software é modificado, ele tende a se tornar mais complexo. Assim, ao precisar atender a novos requisitos seu código torna-se menos expressivo, isto é, menos legível e mais complicado de ser compreendido. Por haver se tornado tão imenso e intricado, com centenas de componentes e funcionalidades, entendê-lo passa a ser uma tarefa desafiadora.

A arquitetura de software é um conceito fundamental no desenvolvimento e evolução de software. E juntamente com o sistema de software, a arquitetura também sofre os efeitos do envelhecimento de software, conhecidos como *ersoão arquitetural e desvio arquitetural*. É de extrema importância o gerenciamento da evolução arquitetura de sofware, uma vez que ela representa a visão geral do sistema e é fundamentalmente responsável por requisitos de qualidade, como segurança e performance.

Eventualmente, manter e evoluir um sistema de software torna-se mais custoso que reescrevê-lo por completo. Atingido esse estado, o software original é abandonado e sua vida útil chega ao fim. Ele é substituído por uma nova implementação, mais moderna, que atende aos novos requisitos, mas que, eventualmente, também envelhecerá, e também será substituída por uma nova implementação.

Este trabalho propõe uma solução, composta por método e apoio ferramental, de gerenciamento da evolução de um sistema de software, especialmente, de sua arquitetura, que é peça fundamental na longevidade do sistema. O método proposto nesta solução objetiva identificar precocemente problemas de envelhecimento de software relacionados à arquitetura do sistema, permitindo ao arquiteto de software atuar sobre eles, de forma a mitigar ou eliminar seus impactos sobre a arquitetura do sistema, consequentemente, prolongando a vida útil do mesmo.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Este capítulo contextualiza o trabalho discutido nesta dissertação, apresentando a motivação que guiou seu desenvolvimento e o problema que se busca resolver. Também são apresentados os trabalhos relacionados, encontrados na literatura, que mais se aproximam deste, seja em objetivo ou método. Eles são examinados e suas contribuições e limitações analizadas, de forma a estabelescer em que este projeto baseia-se e o que ele busca aperfeiçoar. Ao final do capítulo, de forma a facilitar a leitura, o restante dos capítulos deste documento são brevemente descritos.

1.1 Contexto

O trabalho do engenheiro de software não termina quando o software é entregue ao cliente. Durante o uso do software defeitos são encontrados e precisam ser reparados, surgem demandas de novas funcionalidades a serem implementadas e atributos de qualidade que necessitam ser otimizados. As novas funcionalidades podem conflitar com requisitos anteriores, e as modificações podem ir contra premissas previamente estabelecidas. Fatores como esses contribuem para a redução da vida útil do software ao longo da sua evolução, uma vez que dificultam subsequentes manutenções até o momento em que se torna mais barato reescrever o software por completo a continuar a mantê-lo [60, 7].

É benéfico, e muitas vezes necessário, prolongar ao máximo a vida útil de um software. Isso se deve ao fato dos custos de desenvolvimento poderem ser diluídos no tempo de vida do sistema. Além disso, a descontinuação precoce do software traz custos indesejáveis de ter que se desenvolver ou adquirir um novo sistema. Desta forma, o intuito deste trabalho é prover uma abordagem para gerenciar a evolução de software de maneira a prolongar a vida útil do sistema.

O fenômeno da redução da vida útil do software é conhecido por diversos termos: (i)

erosão arquitetural [46], (ii) desvio arquitetural[1] [46] e (iii) envelhecimento de software[2] [45].

O termo erosão arquitetural foi cunhado por Perry e Wolf [46]. Seu significado está relacionado com alterações que violam a arquitetura do sistema (por exemplo, ao desrespeitar restrições de comunicação de um padrão arquitetural), aumentando a resistência do sistema a mudanças. Já o termo desvio arquitetural, definido pelos mesmos autores, está relacionado a mudancas não previstas na arquitetura que tendem a degradar sua consistência. Desta forma, após várias alterações, torna-se tão difícil evoluir o sistema que o custo de mantê-lo é maior que o de reescrevê-lo completamente. Neste caso, o software atinge o fim de sua vida útil, pois o custo de atender a novos requisitos é maior que o retorno obtido.

Argumenta-se que seria possível evitar a erosão arquitetural caso fosse utilizada uma abordagem como a engenharia dirigida por modelos[3] [63]. A engenharia dirigida por modelos prevê a descrição do software através de modelos focados no domínio e o uso de ferramentas para transformá-los em artefatos de implementação [50]. Assim para alterar o software é necessário alterar antes o modelo que o descreve, como por exemplo, um modelo arquitetural. Desta forma, as alterações da arquitetura seriam feitas conscientemente em um nível de abstração superior ao da implementação e o uso de ferramentas de transformações garantiriam a conformidade entre o modelo e a implementação.

Entretanto não é possível, na prática, impedir alterações diretas ao código-fonte [59, 60]. Entre as principais razões destacam-se restrições de orçamento e tempo, além da falta de suporte de ferramentas que auxiliem no processo de transformação dos modelos para código-fonte [14].

Ao passo que alterações diretas ao código-fonte ocorrem, questões como restrição de tempo, falta de documentação, rotação da equipe de desenvolvimento (e consequente perda de conhecimento das decisões de projeto dentro da equipe) levam os desenvolvedores a tomarem decisões sub-ótimas [59], que, por vezes, violam a arquitetura, contribuindo para sua erosão.

Há, portanto, uma necessidade prática em reconhecer e tratar problemas de erosão arquitetural o quanto antes durante a evolução do software. São para estes problemas que propomos uma solução, constituída por um método e ferramenta, fundados nos princípios de engenharia dirigida por modelos, que permitem aos desenvolvedores de software gerenciar a evolução de software ao reconhecer e atuar sobre os cenários de erosão arquitetural em um sistema baseado em componentes.

[1] Do inglês, *architectural drift*.
[2] Do inglês, *software aging*.
[3] Do inglês, *model driven engineering*.

1.2 Motivação e problema

"Como as pessoas, software envelhecem"[45]. Entre os várias aspectos do envelhecimento de software, destacam-se o envelhecimento da arquitetura de software. As formas em que a arquitetura pode envelhecer são classificadas em: (i) erosão arquitetural e (ii) desvio arquitetural [46]. Ambos os aspectos reduzem a vida útil do software, ao dificultar novos requisitos de serem incorporados ao sistema, ao reduzir a facilidade com que o sistema pode ser modificado, aumentando o custo de manutenção.

A identificação dos problemas de envelhecimento da arquitetura de software não é uma tarefa simples. Em geral, não existe um mapeamento claro entre os artefatos encontrados no código-fonte e os elementos definidos em diagramas arquiteturais [14]. Não existindo suporte explícito da linguagem de programação ou método utilizado durante o desenvolvimento, esse mapeamento tende a ser claro no início da implementação e a desvanecer conforme o sistema evolui.

Aliado à falta de suporte de linguagens de programação em estabelecer os limites entre elementos arquiteturais no código-fonte, a complexidade dos sistemas de software tornam o trabalho do arquiteto que deseja verificar a conformidade entre código-fonte e arquitetura extremamente árduo. Mesmo se o arquiteto conseguir executar a façanha de catalogar cada elemento arquitetural dentre as milhares de linhas de código-fonte, ele ainda precisará definir as variadas interações entre os elementos arquiteturais e se a implementação de cada um deles reflete real e corretamente a especificação arquitetural.

Se suficientemente determinado o arquiteto recuperar todas essas informações a partir do código-fonte, ele ainda terá o desafiador trabalho de comparar o modelo arquitetural com aquele que ele recuperou o partir do código-fonte. Após isso, ele precisará identificar as diferenças e julgar entre elas o que de fato é um problema de envelhecimento da arquitetura. Só com essa análise, ele poderá então decidir como atuar para sanar tal problema.

Depois de possivelmente alguns dias de trabalho, mesmo para os sistemas de complexidade mais modesta, o arquiteto encontrar-se-á com as mudanças de arquitetura necessárias para resolver as inconformidades que ele encontrou ao visitar o código-fonte. Entretanto, tais mudanças são aplicáveis à arquitetura e o arquiteto precisa, na verdade, que o código-fonte seja alterado. Agora o arquiteto precisa fazer o trabalho inverso e obter as mudanças do código-fonte que realizarão as alterações arquiteturais que ele deseja executar. Chegado este ponto, o arquiteto pode delegar o trabalho aos engenheiros de software que executarão as mudanças necessárias no código.

Infelizmente, além de extremamente trabalhoso, esse processo também é altamente suscetível a erro, devido à alta complexidade intrínseca de cada uma das tarefas associadas a ele. Considerando todos esses aspectos, a seção 1.4 descreve como esta pesquisa

propõe um método e ferramenta de apoio a fim de auxiliar o arquiteto de software a combater problemas de envelhecimento da arquitetura sem as limitações ou pontos fracos encontrados em outras propostas, descritas na seção 1.3.

1.3 Trabalhos relacionados

1.3.1 Abordagens de recuperação de arquitetura (Rasool et al. e Riva et al.)

Rasool et al. [47] apresentam uma abordagem de recuperação da arquitetura baseada tanto no código-fonte quanto em documentos de projeto e conhecimento do domínio. A recuperação da arquitetura baseada no código-fonte utiliza-se de análise sintática do código. Desta forma, esta solução tem por maior limitação a dependência da linguagem de programação usada e a necessidade do conhecimento do domínio da aplicação.

Riva et al. [48] descrevem uma solução de recuperação da arquitetura a fim de possibilitar a verificação de conformidade entre a implementação e a arquitetura definida em fase de projeto. A abordagem descrita baseia-se em recuperar a arquitetura através de regras heurísticas derivadas manualmente, baseadas no conhecimento do domínio e em detalhes de implementação. A principal limitação desta proposta está no fato de requerer intervenção manual para gerar as regras heurísticas e na necessidade de contínua intervenção para dar manutenção às regras conforme o sistema evolui.

Ambas as abordagens descritas anteriormente possuem a mesma limitação, ambas necessitam de intervenção manual do arquiteto e de conhecimento do domínio ou da implementação. Isso ocorre pelo fato desses trabalhos não se aproveitarem de nenhum modelo de implementação da arquitetura. A falta de tal modelo dificulta a recuperação da arquitetura, uma vez que existem inúmeras possibilidades de representar a arquitetura no nível do código-fonte. Sem um mapeamento explícito da arquitetura para a implementação é necessário o conhecimento do arquiteto sobre a tecnologia e a forma como ela é utilizada na implementação.

1.3.2 Zheng et al.

Zheng et al. [62, 63] definem uma nomenclatura para as abordagens de mapeamento e recuperação da arquitetura, que se classificam em duas categorias: (i) mapeamento arquitetura-implementação de uma via [4] ou (ii) mapeamento arquitetura-implementação de duas vias [5].

[4] Do inglês, *one way architecture-implementation mapping*.
[5] Do inglês, *two way architecture-implementation mapping*.

O conceito de *mapeamento arquitetura-implementação* consiste na abordagem de mapear artefatos arquiteturais (componentes, conectores e interfaces) em artefatos no nível do código-fonte. A partir deste mapeamento é possível tanto gerar código-fonte a partir de um modelo arquitetural quanto recuperar a arquitetura a partir do código-fonte.

O que diferencia o número de vias está no fato da abordagem adotada ser ou não capaz de realizar operações em ambas as direções, isto é, tanto gerar código, quanto recuperar a arquitetura. De acordo com Zheng et al. o mapeamento de uma via é capaz apenas de gerar código-fonte a partir de um modelo arquitetural, já o de duas vias é capaz, além da função anterior, de recuperar a arquitetura do sistema a partir do código-fonte.

A abordagem apresentada por Zheng et al. [62, 63] é nomeada *1.x-Way Architecture-Implementation Mapping* e consiste no mapeamento arquitetura-implementação de uma via, sendo capaz de gerar código-fonte a partir de um modelo arquitetural. A solução proposta para o problema de erosão arquitetural consiste em permitir a evolução da arquitetura apenas através da alteração do modelo arquitetural e então aplicando as alterações ao código-fonte através do auxílio de uma ferramenta de geração automática de código. Portanto, o desenvolvedor é impedido de alterar diretamente o código-fonte sem antes haver realizado uma análise da arquitetura.

A principal limitação dessa abordagem é que, como discutido na seção 2.2, nem sempre é possível evitar que a erosão arquitetural ocorra devido a uma série de fatores apresentados em [60] e discutidos no seção 1.1. Desta forma, impedir o desenvolvedor de alterar o código-fonte sem antes realizar uma análise detalhada do impacto arquitetural pode não ser uma restrição aceitável em todos os cenários de evolução do software. Restrições de tempo ou orçamento podem forçar a equipe de manutenção a ignorar o impedimento proposto por essa abordagem. Neste caso, ao implementarem alterações no sistema sem o auxílio da ferramenta proposta, haverá um desvio, não esperado pela ferramenta, entre o código-fonte e o modelo arquitetural[6]. Neste ponto é necessária a intervenção manual de um arquiteto de software para recuperar a arquitetura.

1.3.3 Lytra et al.

Lytra et al. [37], apresentam uma abordagem para tratar a vaporização do conhecimento arquitetural [32], isto é, a perda do conhecimento arquitetural, conjunto de informações e decisões que são relevantes para a concepção e construção de uma dada arquitetura.

Tal trabalho, propõe uma linguagem de transformação de conhecimento arquitetural para um modelo de componentes e conectores, seguindo as ideias fundamentais de engenharia dirigida por modelos. Com isso, a proposta procura atacar uma das formas de

[6]E portanto erosão arquitetural.

envelhecimento de software: a perda do conhecimento arquitetural e das decisões arquiteturais.

A premissa é que existem decisões arquiteturais recorrentes, como por exemplo, a implementação de padrões arquiteturais, tais como *proxies* e adaptadores. Tais decisões arquiteturais podem ser escritas utilizando a linguagem de transformação proposta pelos autores e então aplicada ao modelo arquitetural. Com isso, é reduzido o número de tarefas tediosas que o arquiteto necessita executar e também se é mantido o histórico de decisões arquiteturais aplicados a tal modelo.

Desta forma, tal trabalho está relacionado ao combate do envelhecimento de software, mas restrito à perda das decisões arquiteturais devido às alterações da arquitetura, realizadas pelo arquiteto. Entretanto, a abordagem foca-se somente em alterações controladas da arquitetura, realizadas pelo arquiteto através da linguagem proposta. Possuindo similares limitações da proposta de Zheng et al. (seção 1.3.2), quanto a inevitáveis alterações do código-fonte com pouca ou nenhuma consideração arquitetural.

1.3.4 Dimech et al.

Dimech et al. [18] propõe uma abordagem para verificar a conformidade da arquitetura implementada contra a arquitetura especificada durante o processo de desenvolvimento de software.

A proposta baseia-se em recuperar informações arquiteturais a partir do código-fonte e compará-las contra um modelo arquitetural do sistema, especificado pelo arquiteto de software. Após a análise, a ferramenta implementada pelos autores, reporta os conflitos encontrados. Desta forma, os desenvolvedores podem corrigir o código-fonte de forma a remover todas as inconsistências entre a implementação e a arquitetura especificada.

Deve-se notar, antes de mais nada, que o que os autores nomeiam de "componente" difere-se do conceito adotado neste trabalho. Para os autores, um componente arquitetural está no mesmo nível de abstração que uma classe; em sua proposta, eles utilizam diagramas de classes UML para representar a arquitetura do sistema. Desta forma, a etapa de recuperação arquitetural a partir do código-fonte tem como resultado um diagrama de classes em UML. Assim a verificação arquitetural também se dá no nível de classes.

Nesta dissertação, toma-se por um componente arquitetural uma unidade que pode ser implantada independentemente, com interfaces e dependências bem definidas [53]. Ainda que uma classe possa, em tais definições, ser considerada como um componente, usualmente tal unidade é muito menos granular, sendo composta por várias classes, quando a tecnologia de implementação segue o modelo orientado a objetos.

A proposta de Dimech et al. é extremamente interessante ao apontar as deficiências de implementação da arquitetura, mas não apresenta uma maneira automatizada de re-

solver tais problemas através de alteração no modelo arquitetural recuperado. Qualquer alteração deve ser feita manualmente pelos desenvolvedores.

Como será discutido durante o restante desta dissertação, a solução proposta por este trabalho procura trazer o gerenciamento da evolução do sistema para o nível da arquitetura, isto é, alterações e correções da arquitetura devem ser feitas utilizando-se modelos arquiteturais e não o código-fonte. Com o uso do ferramental que acompanha a solução, as adequações arquiteturais podem então ser mapeadas em mudanças no código-fonte.

1.4 Solução proposta

Como introduzido no Capítulo 1.1, a engenharia dirigida por modelos descreve um sistema de software através de modelos representativos do domínio [50]. Aliadas a isso, ferramentas devem ser usadas para transformar os modelos em artefatos de implementação. Com essas transformações, busca-se garantir que os artefatos de implementação estejam em conformidade com os modelos que descrevem o sistema de software. Desta forma, problemas como erosão e desvio arquitetural podem ser evitados, uma vez que não é permitida a inconsistência entre o modelo e a implementação.

Zheng et. al [63, 62] (seção 1.3.2) usam uma abordagem similar a descrita acima. Contudo como será detalhadamente discutido na seção 2.2, em cenários reais, restrições, como tempo e orçamento, demandam modificações diretas dos artefatos de implementação [60]. Isso torna esse tipo de abordagem difícil de ser aplicada de forma prática em ambientes reais.

Tendo em vista essa questão, o objetivo deste trabalho é propor uma solução, baseada nos fundamentos de engenharia dirigida por modelos, para auxiliar arquitetos e desenvolvedores a gerenciar a evolução de sistemas de software e combater problemas de seu envelhecimento, sem impor restrições quanto a mudanças diretas no código-fonte.

A solução constitui-se de um método para gerenciar e facilitar a evolução de sistemas de software baseados em componentes. Fundamentada na engenharia dirigida por modelos, propomos também uma ferramenta capaz de tornar prática e eficiente sua execução.

De maneira similar ao descrito em Riva et al. [48] e Rasool et al. [47], o método proposto prevê a recuperação da arquitetura do sistema de software. Desta forma, mesmo que haja alterações diretas no código-fonte, sem a devida atualização do modelo arquitetural, é possível recuperar o modelo da arquitetura implementada, compará-lo com o modelo da arquitetura especificada em fase de projeto e revelar inconsistências entre os modelos. Contudo, diferentemente das abordagens de ambos os autores, propomos o uso de um modelo de implementação da arquitetura (seção 2.5). O uso de tal modelo permite automatizar o processo de recuperação da arquitetura, além de possibilitar que correções

aplicadas ao modelo da arquitetura possam ser aplicadas diretamente ao código-fonte de maneira automatizada.

Em resumo, o método proposto prevê a recuperação de um modelo arquitetural a partir do código-fonte do sistema, bem como a geração de código automaticamente para aplicar, em nível de implementação, as alterações que sejam feitas no modelo arquitetural. Acompanhando o método na solução proposta, a ferramenta desenvolvida neste trabalho automatiza as atividades descritas no método, utilizando-se dos princípios da engenharia dirigida por modelos.

Deve-se notar que nem sempre é possível criar um modelo de geração automática de código sem nenhuma intervenção humana. Desta forma, as abordagens de geração automática de código buscam reduzir ao máximo a intervenção manual no processo e não necessariamente sua eliminação.

Este trabalho dá continuidade e integra-se com outros trabalhos do grupo de Engenharia de Software e Tolerância a Falhas do Instituto de Computação da Universidade Estadual de Campinas, nas áreas de arquitetura de software, ferramentas e evolução de sistemas baseados em componentes [56, 58].

1.5 Organização deste documento

O restante deste documento está organizado da seguinte forma: o Capítulo 2 apresenta alguns conceitos essenciais para a contextualização do trabalho; no Capítulo 3 é discutida a solução proposta, juntamente com a descrição das atividades do método que a compõe; no Capítulo 4 é apresentada a ferramenta implementada durante este projeto, responsável pela execução automatizada de várias atividades relacionadas a solução proposta; em seguida, no Capítulo 5 são realizados dois estudos de caso que buscam verificar a validade da solução adotada; por fim, no Capítulo 6 esta dissertação é concluída com um resumo, observações finais, propostas de trabalhos futuros e a descrição das contribuições alcançadas por este trabalho.

Capítulo 2

Fundamentos de arquitetura de software e arquitetura dirigida por modelos

Este capítulo apresenta os princípios que fundamentam este trabalho, necessários para a compreensão do restante desta dissertação, relativos às áreas de arquitetura de software, componentes de software e arquitetura dirigida por modelos.

2.1 Arquitetura de software

A arquitetura de software é uma visão estrutural de alto nível de abstração do sistema computacional. Ela é composta por **componentes** que se relacionam através de **conectores** [25]. Nos componentes são implementadas as funcionalidades, enquanto que os conectores são responsáveis por implementar protocolos de comunicação entre componentes e coordenar a execução de serviços que demandam mais de um componente do sistema.

A arquitetura de software apresenta uma série de benefícios para o desenvolvimento de sistemas complexos, os principais são: (i) a organização do sistema como uma composição de componentes lógicos; (ii) a antecipação da definição das estruturas de controle globais; (iii) a definição da forma de comunicação e composição dos elementos do projeto; e (iv) o auxílio na definição das funcionalidades de cada componente projetado. Ao prover uma visão estrutural, a arquitetura de software permite um entendimento e analise de alto nível sobre o sistema, tornando-se um artefato importante durante toda a vida útil do software [14].

Sommerville [51] apresenta outros benefícios trazidos pela arquitetura de software: (i)

facilita a comunicação entre os diferentes *stakeholders*[1] por apresentar uma visão de alto nível do sistema, (ii) permite o reúso em larga escala de funcionalidades ao restringi-las e mapeá-las em subsistemas e (iii) o estilo e a estrutura da arquitetura influenciam na determinação de requisitos não funcionais do sistema.

Uma propriedade arquitetural representa uma decisão de projeto relacionada a algum requisito não funcional do sistema, que quantifica determinados aspectos do seu comportamento, como confiabilidade, reusabilidade e modificabilidade [14, 24].

Dado que a arquitetura de software é tão benéfica e vital no ciclo de vida do software, é necessário que ela seja preservada tanto durante o período de desenvolvimento do software quanto após a entrega de sua primeira versão ao cliente. Assim é fundamental para o desenvolvimento de softwares complexos que existam ferramentas que possam apoiar o desenvolvimento centrado na arquitetura, evitando sua erosão e gerenciando sua evolução [14].

2.1.1 Recuperação de arquiteturas de software

O processo de recuperação de arquitetura de software consiste em extrair níveis superiores de abstração a partir de sistemas de software existentes [47]. Em geral, a tarefa é iniciada a partir do código-fonte do sistema e, através de um processo de engenharia reversa, busca-se obter um modelo de alto nível, como por exemplo um modelo de componentes e conectores do sistema [47, 52].

É essencial que se mantenha a conformidade entre a implementação e a modelagem da arquitetura especificada em nível de projeto, já que, como descrito na seção 2.2, a incapacidade de mantê-las conforme é a causa do aumento crescente dos custos de evolução do software. Também é descrito na mesma seção o fato de atualmente não ser possível manter durante toda a evolução do software a conformidade deste com sua arquitetura definida na fase de projeto.

Portanto, é preciso que existam alternativas para reverter cenários de erosão arquitetural avançada de um sistema. Como primeiro passo para se recuperar um sistema erodido, é necessário obter sua arquitetura subjacente. Desta forma, a recuperação da arquitetura de software é uma das ideias de fundamental importância utilizadas nesta pesquisa.

2.1.2 MDA: Arquitetura dirigida por modelos

Plataformas tecnológicas atualmente utilizadas para o desenvolvimento de software, como por exemplo J2EE [44] e .NET [39], possuem milhares de classes e métodos, com dependências intricadas e sutis efeitos colaterais que requerem aos desenvolvedores amplo

[1]Termo em inglês usado para denotar os interessados e envolvidos em um projeto de software.

conhecimento das plataformas e suas engrenagens. A engenharia dirigida por modelos é uma abordagem promissora para combater tal complexidade, expressando, através de modelos, conceitos de domínio do negócio, dificilmente capturados por tais plataformas [50].

Fundamentalmente, a engenharia dirigida por modelos baseia-se em [50]: (i) linguagens específicas de domínio, descritas através de metamodelos, que definem as relações entre os conceitos de negócio, detalhando a semântica e restrições relacionadas a tais conceitos; e (ii) ferramentas de transformação de modelos, que são capazes de analisar os modelos construídos e sintetizar diversos artefatos, como código-fonte, entradas de simulação, documentação de desenvolvimento ou mesmo um outro modelo.

Umas das incarnações da abordagem de engenharia dirigida por modelos, é a arquitetura dirigida por modelos [20], ou MDA[2], definida em 2000 pela OMG[3] [27]. A arquitetura dirigida por modelos é uma iniciativa da OMG de desenvolver padrões baseados na ideia de que a modelagem é fundamental para o desenvolvimento e manutenção de sistemas, e com isso proporcionar uma série de tecnologias e técnicas baseadas em tais padrões [38].

Com as tecnologias orientadas a objeto, um dos princípios básicos era a afirmação de que "tudo é um objeto". Com o MDA, o jargão torna-se "tudo é um modelo". Fazendo uma comparação com orientação a objetos, na qual uma instância é, de fato, instância de alguma classe, e uma classe pode herdar propriedades de uma terceira; com MDA um sistema é representado por um modelo, que por sua vez, é descrito por um metamodelo [8].

Em resumo, a ideia básica da engenharia dirigida por modelos é a representação dos sistemas por modelos claramente baseados no domínio do negócio e de que ferramentas automatizadas capazes de transformar um modelo em outro, até em artefatos dependentes de plataforma, que podem ser executados e prover as funcionalidades modeladas. Como será extensamente discutido no restante desta dissertação, a engenharia dirigida por modelos é claramente uma das bases na idealização da solução proposta.

2.2 Erosão arquitetural, desvio arquitetural e envelhecimento de software

O fenômeno da redução da vida útil do software é conhecido por diversos termos: (i) erosão arquitetural [46], (ii) desvio arquitetural[4] [46] e (iii) envelhecimento de software[5] [45].

[2]Do inglês, *model-driven architecture*.
[3]Object Management Group.
[4]Do inglês, *architectural drift*.
[5]Do inglês, *software aging*.

Ainda que alguns pesquisadores usem esses termos de forma intercambiada, existem diferenças em suas definições originais. Parnas [45] descreveu o envelhecimento de software como a redução da vida útil de uma peça de software por dois motivos relacionados:

1. Com o tempo, usuários tornam-se insatisfeitos e demandam novos requisitos. Caso eles não sejam implementados os usuários procurarão outro sistema tão logo que os benefícios de trocar de sistema superem os custos de manter o atual.

2. A alteração do software é feita por pessoas que não entendem a concepção original do projeto, implementando decisões contraditórias às iniciais. Após as alterações, faz-se necessário conhecer as ideias inicias do projeto mais aquelas adicionadas na mudança. Depois de várias alterações, torna-se tão difícil entender as regras implementadas que as mudanças começam a levar muito tempo e possibilitam a introdução de novas falhas. A partir daí, torna-se inviável manter o software (e portanto atender a novos requisitos).

Perry e Wolf [46] definem erosão arquitetural como violações da arquitetura de software, que tendem a aumentar a fragilidade do sistema, ou seja, um crescente aumento na resistência a mudança do software. Essas violações consistem em alterações que ignorem restrições ou premissas da arquitetura, como, por exemplo, o desrespeito a restrições de comunicação entre componentes arquiteturais.

Já o desvio arquitetural é devido à insensibilidade à arquitetura, causando inadaptabilidade, falta de coerência e clareza de forma, facilitando a ocorrência da erosão arquitetural [46], isto é, alterações que adicionam decisões de projeto não previstas na arquitetura, mas que também não afetam as premissas ou restrições iniciais.

Ambas definições de erosão arquitetural e envelhecimento de software convergem para o fato da crescente dificuldade em alterar o software devido às próprias alterações anteriores. Uma vez que o custo de manter o software torna-se maior do que o custo de reescrevê-lo, não é mais possível atender aos novos requisitos demandados pelos usuários e então o software atinge o fim de sua vida útil.

O envelhecimento de software pode ser traduzido como a recorrente necessidade de que os sistemas computacionais se adequem a novos requisitos aliada ao fato da incapacidade de se evoluir o software sem que ocorra erosão arquitetural. Portanto o envelhecimento do software ocorre devido à erosão arquitetural.

Desta forma, quanto mais longo for o período de vida do sistema de software, menor terá sido o custo de seu desenvolvimento relativamente ao tempo de uso do sistema. Certamente é benéfico que haja preocupação em evitar a erosão arquitetural desde do início do ciclo de vida do software.

Contudo, ainda que desejável, nem sempre é possível, durante a evolução do software, manter a conformidade entre a implementação e a arquitetura. As principais causas desse

problema são descritas em [60, 48]: (i) decisões de projeto perdidas durante a evolução do sistema (por exemplo, devido à falta de documentação), (ii) insensibilidade às decisões de projeto, (iii) restrições de tempo e/ou custo não permitem a resolução adequada de falhas, (iv) manutenção destinada a times inexperientes ou sem prévio conhecimento do sistema, (v) pressão dos gestores por resultados rápidos (podendo reduzir a qualidade do software produzido).

Acredita-se que o envelhecimento de software é inevitável [45, 59], ou seja, cedo ou tarde ocorrerão eventos que erodirão a arquitetura do sistema. Portanto é necessário que haja meios de recuperar um sistema erodido, de forma a evitar ao máximo a necessidade de ter que reescrevê-lo por completo. Essa é a motivação principal deste trabalho.

2.3 Desenvolvimento baseado em componentes

A noção de desenvolvimento baseado em componentes é fundada no principio da decomposição, muito usada em várias das ideias em engenharia de software. Um sistema complexo pode ser subdividido em partes menores, com funcionalidades e interfaces bem definidas. Uma análise de cada uma das partes, ou *componentes*, é então realizada para verificar como elas devem interagir para reconstruir o sistema original [10].

Um componente possui interfaces e funcionalidades providas bem definidas. Isso torna-os passíveis de reutilização quando certa funcionalidade é novamente desejada. Desta forma, há um aumento de produtividade, já que a reutilização de componentes evita a reimplementação da funcionalidade. Em mesmo sentido, há um crescimento da qualidade do software, pois é feito o reúso de peças de software já consolidadas e testadas. Deve-se ressaltar que isso não elimina as possíveis falhas de integração [10].

De acordo com Szyperski, um componente é uma unidade, sujeita a composição, com interfaces contratualmente bem especificadas, dependência de contexto explícita e que pode ser implantada independentemente [53]. As dependências explícitas são materializadas por *interfaces requiridas* enquanto que as funcionalidades implementadas são expostas através de *interfaces providas*.

Todo componente segue basicamente três princípios fundamentais, comuns à tecnologia de objetos [13]:

1. **Unificação de dados e operações:** um componente possui atributos e uma valoração para esses atributos (estado). A mudança de estado (alteração de valores de atributos) deve ser feita por operações internas ao componente. Uma vez que a dependência de dados e operações é internalizada no componente, ele torna-se mais coeso.

2. **Encapsulamento:** o cliente de um componente (aquele que se utiliza de suas interfaces providas) ignora como os dados são armazenados ou como as operações foram implementadas. Ele depende apenas da especificação do componente e não se importa com sua implementação. Essa separação de preocupações[6] é fundamental para tratar as dependências do software e reduzir o acoplamento.

3. **Identidade:** Independentemente de seu estado, cada componente possui uma identificação única.

É importante que não haja confusão entre os conceitos de *componente arquitetural* e *componente de implementação*. Enquanto o componente arquitetural é uma definição lógica e independente de tecnologia, o componente de implementação é uma materialização de um componente arquitetural em uma dada tecnologia, utilizando alguma linguagem de programação.

2.4 UML Components: um processo de desenvolvimento baseado em componentes

O método de desenvolvimento de software UML Components [13] foi idealizado para o desenvolvimento de sistemas baseados em componentes. Para simplificar o desenvolvimento, o método adota uma arquitetura pré-definida em cinco camadas, sendo enfatizadas duas camadas em especial: (i) camada de sistema, que contem os componentes que implementam os cenários específicos da aplicação (ou do sistema); e (ii) camada de negócio, que contem os componentes identificados a partir do domínio do negócio e por isso, são os principais candidatos a serem reutilizados em diferentes aplicações.

O método UML Components é dividido em fases, descritas brevemente a seguir.

O método inicia-se pela fase de especificação, na qual devem ser definidos os requisitos do sistema, usualmente levantados através da utilização de casos de uso. A partir dos requisitos e dos casos de uso, é construído o modelo conceitual de negócio, ou simplesmente, modelo de negócio. Este modelo representa as principais entidades que constituem o domínio a que o sistema está relacionado.

Na fase seguinte, as atividades tomadas focam-se na especificação dos componentes com grande grau de detalhe. Esta fase é a responsável pelo projeto da arquitetura do sistema, incluindo a identificação de componentes, conectores e configurações arquiteturais. O processo UML Components divide essa fase em três sub-fases: (i) identificação de componentes, responsável por identificar as interfaces e componentes arquiteturais; (ii) interação dos componentes, responsável por especificar as configurações arquiteturais e

[6]Do inglês, *separation of concerns*.

novas operações do sistema; e (iii) especificação final dos componentes, responsável por refatorar interfaces e componentes, antes de finalizar o projeto da arquitetura.

A fase seguinte é a de provisionamento, na qual são determinados a aquisição dos componentes, seja através de reúso de componentes internos a organização, compra de componentes de terceiros ou implementação. Na próxima fase, ocorre a montagem, os conectores especificados são implementados e os componentes adquiridos são incorporados, de forma a materializar a configuração arquitetural especificada anteriormente. O método, contudo, é vago quanto a execução das atividades de testes e implantação, que, de acordo com o processo, devem ser realizadas após a fase de montagem.

2.5 Modelo de implementação de componentes e o modelo COSMOS*

Existe uma lacuna semântica entre o modelo arquitetural de alto nível e a linguagem de programação utilizada para a sua implementação. Não existe, por parte destas últimas, um suporte explícito para o mapeamento dos artefatos arquiteturais em artefatos de implementação [14]. A falta desse mapeamento explícito facilita que desvios arquiteturais ocorram durante a implementação, e como consequência, o aparecimento de erosão arquitetural.

COSMOS*, *COmponent System MOdel for Software architectures* [7], é um modelo de implementação de arquitetura de software utilizando a tecnologia de orientação a objetos, independente de linguagem de programação [26].

A abordagem proposta pelo modelo COSMOS* não requer nenhum tipo de linguagem de maior nível além da linguagem orientada a objetos escolhida para a implementação, diferentemente de outros modelos, como o ArchJava [3].

O modelo baseia-se em algumas decisões fundamentais de projeto, das quais destacam-se [26]:

1. **Materialização da arquitetura:** componentes, conectores e interfaces possuem um mapeamento explícito para estruturas providas pelas linguagens de programação orientadas a objetos.

2. **Separação de preocupações não funcionais:** os componentes implementam os requisitos funcionais. É delegado aos conectores a responsabilidade de implementação de requisitos não funcionais, como segurança e distribuição.

[7] Modelo de sistema de componentes para arquiteturas de software, em inglês.

3. **Separação explícita entre especificação e implementação:** a especificação dos componentes é fisicamente separada de sua implementação, que é restrita a seus desenvolvedores. Com isso, há uma redução do acoplamento uma vez que o usuário do componente ignora sua implementação.

4. **Declaração explícita das dependências de um componente:** um componente pode apenas depender de funcionalidades declaradas por suas interfaces requeridas, o que permite que ele seja desenvolvido e implantado isoladamente.

O mapeamento explícito da arquitetura para a implementação permite o controle da evolução da *arquitetura implementada*, isto é, é possível verificar a ocorrência de mudanças arquiteturais comparando a arquitetura implementada com a arquitetura definida na fase de projeto. Através desse tipo de análise é possível verificar discrepâncias entre a implementação e a especificação, reconhecendo assim a existência de erosão arquitetural no sistema.

Deve-se ressaltar que o COSMOS* é um de vários modelos de implementação existentes e foi escolhido como modelo de implementação da arquitetura neste trabalho pelas características previamente mencionadas. Entretanto seria possível utilizar outro modelo de implementação que proporcionasse mapeamento unívoco entre os artefatos arquiteturais e os de implementação.

Capítulo 3

Meissa: método de apoio à evolução arquitetural

Meissa é uma das estrelas pertencentes a constelação de Orion e juntamente com outras duas estrelas, Bellatrix e Betelguese, elas formam o elmo da figura mítica, o Caçador, em cujo ápice Meissa está posicionada.

Meissa também é o nome dado à solução proposta neste trabalho, cujo foco é o gerenciamento da evolução arquitetural. O nome foi especialmente escolhido devido à relação deste trabalho e trabalhos anteriormente desenvolvidos pelo grupo de Engenharia de Software e Tolerância a Falhas do Instituto de Computação da Universidade Estadual de Campinas, destacando-se, entre elas, o ambiente Bellatrix [58], também nomeado segundo uma das estrelas da constelação de Orion[1].

O ambiente Bellatrix será discutido mais a frente, no próximo capítulo, quando o suporte ferramental que compõe a solução Meissa for apresentado. A solução Meissa é tanto composta pelo método Meissa quanto pela ferramenta Meissa, em conjunto, eles são capazes de auxiliar o arquiteto a identificar e encontrar problemas de envelhecimento de software que afetam ou se apresentam na arquitetura do mesmo.

Este capítulo foca-se na descrição do método e de suas atividades. A ferramenta será discutida a parte, no Capítulo 4.

3.1 Visão geral do método Meissa

Meissa composta por um método iterativo de apoio à evolução arquitetural. Sua aplicação independe do processo de desenvolvimento adotado, desde que seja possível aplicar as definições de arquitetura de software discutidas na seção 2.1.

[1]Aparentemente, o próximo trabalho do grupo de pesquisa tem grandes chances de se chamar Betelguese!

Ainda que seja independente de processo, existe grande afinidade entre o método proposto e processos iterativos de desenvolvimento de software. Em especial, a aplicação do método encaixa-se muito bem com o modelo espiral de desenvolvimento de software [9]. Neste modelo, o processo de desenvolvimento de software ocorre em uma espiral, tendo seu início no centro da mesma. Cada volta concluída na espiral é uma iteração do processo que segue as seguintes quatro fases: (i) definição de objetivos, (ii) avaliação e redução de riscos, (iii) desenvolvimento e validação, e (iv) planejamento.

É na terceira etapa, de desenvolvimento e validação, que o método melhor adequa-se, contribuindo em especial para a verificação e adequação da arquitetura implementada. Aplicando-se o método ao final de cada etapa de desenvolvimento irá reduzir as chances de erosão e desvio da arquitetura implementada, uma vez que ela estará sendo constantemente monitorada ao final das atividades que potencialmente podem degradá-la.

É claro que a equipe responsável pelo projeto deverá considerar os custos de aplicação do método e o orçamento disponível para as atividades de verificação e validação. Contudo, como será discutido mais à frente quando apresentarmos o suporte computacional para a aplicação do método, ficará claro que com o devido auxílio ferramental, o custo de aplicação do método é severamente reduzido, tornando sua aplicação extremamente interessante para a manutenção da conformidade entre arquitetura e implementação.

O método Meissa é similar a outros métodos centrados na arquitetura, como o ACDM[2] [36]. O foco de tais métodos está em utilizar a arquitetura como base do desenvolvimento de software, aproveitando-se dela para lidar com problemas técnicos advindos do desenvolvimento de software. A arquitetura é utilizada como elemento central para a comunicação entre os *stakeholders*, definir requisitos funcionais e de qualidade, identificar riscos técnicos e servir como base para o planejamento e execução de atividades [43].

O método baseia-se em quatro ideias fundamentais: (i) o desenvolvimento baseado em componentes (seção 2.3), (ii) modelos de implementação de componentes (seção 2.5), (iii) engenharia dirigida por modelos (seção 2.1.2) e (iv) recuperação da arquitetura de software (seção 2.1.1).

A seguir, definiremos as atividades do método Meissa e detalhamos como cada um dos elementos acima possibilita sua execução. As atividades do método são enumeradas abaixo:

ATIV. 1 Recuperação da arquitetura do sistema

ATIV. 2 Comparação da arquitetura recuperada contra a especificação da arquitetura[3]

[2]Do inglês, *Architecture-centric development method* ou método de desenvolvimento centrado na arquitetura.

[3]Entendemos por especificação da arquitetura o conjunto de artefatos produzidos durante as atividades de especificação e projeto de um sistema de software que definem como deverá ser sua arquitetura.

ATIV. 3 Adequação da arquitetura

ATIV. 4 Mapeamento das alterações arquiteturais para artefatos de implementação

ATIV. 5 Implementação das alterações arquiteturais

ATIV. 6 Versionamento dos modelos arquiteturais produzidos

A Figura 3.1 resume as atividades do método Meissa, relacionando a ordem em que elas ocorrem e quem é responsável por executá-las.

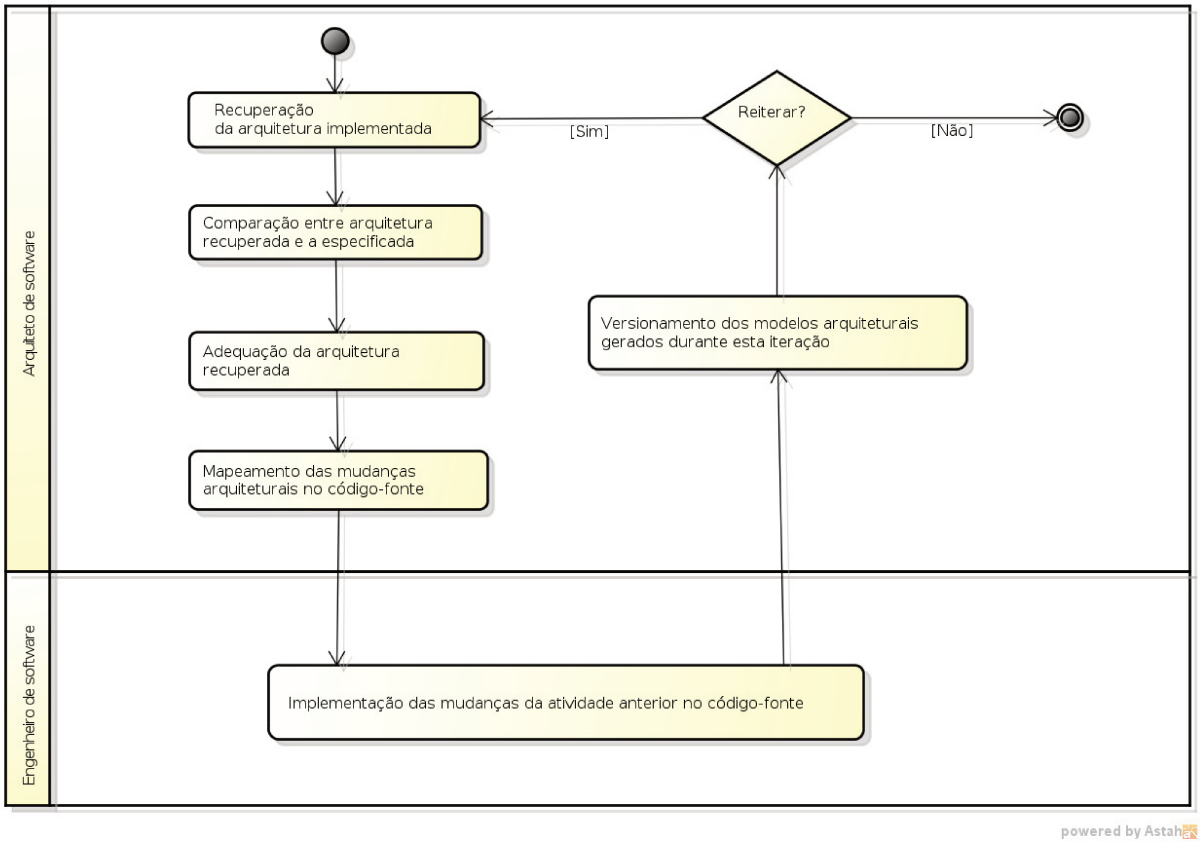


Figura 3.1: Diagrama de atividades do método Meissa.

Como discutido na seção 2.2, durante as fases de desenvolvimento e manutenção de um sistema de software, a arquitetura, como outros elementos do sistema, tende a degradar devido às alterações realizadas sobre o sistema. O método discutido nesta seção foi idealizado para auxiliar o arquiteto de software a identificar e atuar sobre problemas de envelhecimento de software relacionados à evolução da arquitetura de software. Para tanto o arquiteto deverá aplicá-lo durante as fases de desenvolvimento e evolução do sistema.

É possível resumir as atividades do método como **Recuperação**, **Comparação**, **Adequação**, **Mapeamento**, **Implementação** e **Versionamento**.

3.1.1 Definição da nomenclatura

Para o entendimento do restante deste capítulo, deve-se atentar à seguinte nomenclatura adotada: a arquitetura **atual** do sistema é a arquitetura **implementada**, ou seja, a arquitetura materializada pelo código-fonte. A arquitetura **original** é a arquitetura **especificada** pelo arquiteto de software durante as fases de especificação e projeto do sistema. Como a primeira atividade do método prevê a **recuperação** da arquitetura implementada, o termo arquitetura **recuperada** é intercambiadamente utilizado para se referir à arquitetura implementada do sistema (que foi, de alguma maneira, recuperada).

A arquitetura do sistema pode ser representada utilizando-se modelos[4], assim o **modelo arquitetural recuperado** é aquele que representa a arquitetura **recuperada** (ou implementada). De mesma forma, o **modelo arquitetural de especificação** é aquele que representa a arquitetura **especificada** pelo arquiteto. Devido ao fato do método Meissa trabalhar inerentemente com arquiteturas de software, por brevidade do texto, o termo *modelo* é, as vezes, utilizado em lugar do termo completo *modelo arquitetural*, subentendendo tratar-se da representação arquitetural. Quando se é referenciado um modelo que representa outro domínio que não o arquitetural, o texto explicitamente o distinguirá.

A Figura 3.2 a seguir reúne as atividades do método Meissa, explicitando os modelos de entrada e saída para cada uma delas.

3.1.2 Ativ. 1: Recuperação da arquitetura

A primeira atividade do método é a recuperação da arquitetura através da análise do código-fonte do sistema. Sua execução pode ser esquematizada nas seguintes tarefas.

1. Identificação dos componentes
2. Identificação dos conectores
3. Identificação das interfaces
4. Identificação das relações entre elementos arquiteturais
5. Criação de modelos representando os elementos arquiteturais e a arquitetura

[4]Como, por exemplo, o modelo UML [40].

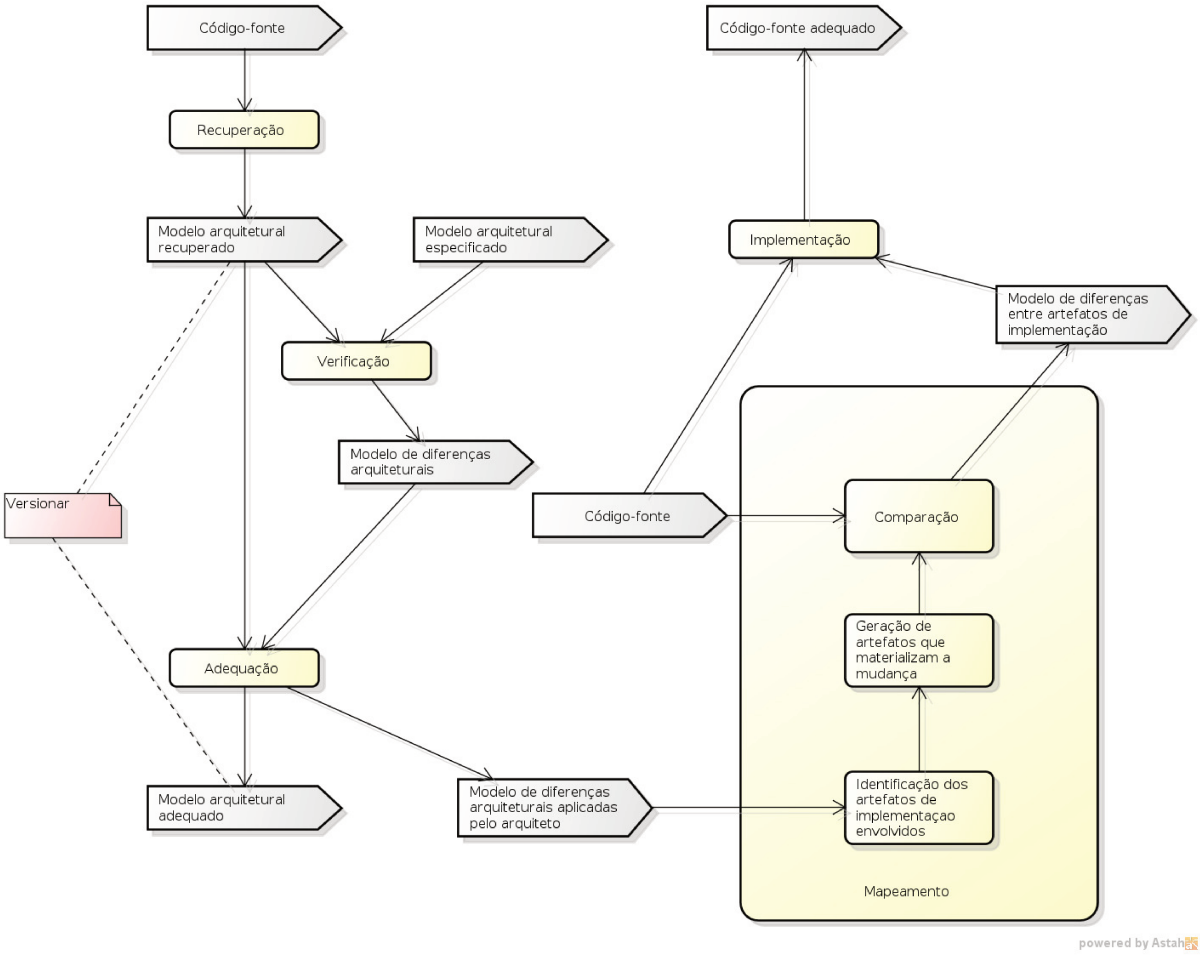


Figura 3.2: Diagrama detalhado de atividades do método Meissa.

A atividade de recuperação inicia-se com a inspeção do código-fonte. É necessário visitar os artefatos de implementação e identificar quais deles fazem partem da materialização da arquitetura. Isto é, que elementos do código fonte materializam cada componente, conector e interface arquitetural. Quais partes do código-fonte indicam relações entre os elementos arquiteturais, como, por exemplo, a existência de uma conexão entre dois componentes através de um conector arquitetural.

Como discutido nas Seções 2.1.1 e 1.3.1, a recuperação da arquitetura de um sistema arbitrário é uma tarefa extremamente árdua, imprecisa e dispendiosa. Contudo, ao utilizar um modelo de implementação de componentes (seção 2.5), essa atividade torna-se precisa, reprodutível e gerenciável e, com isso, o maior ganho: ela torna-se *passível de automação*.

Discutiremos na seção 3.2.1 os detalhes da automatização desta atividade.

3.1.3 Ativ. 2: Comparação entre arquitetura recuperada e especificada

Uma vez que a arquitetura implementada do sistema é recuperada e representada em um modelo arquitetural, torna-se então possível executar a atividade de comparação. Nesta atividade, busca-se analisar o modelo arquitetural recuperado contra o modelo arquitetural criado em fases anteriores, como na de especificação e projeto.

Esta atividade é composta das seguintes tarefas:

1. Identificação de componentes presentes em ambos modelos
2. Identificação de interfaces presentes em ambos modelos
3. Identificação de conectores presentes em ambos modelos
4. Identificação de diferenças entre elementos arquiteturais existentes em ambos modelos
5. Identificação de novos elementos arquiteturais no modelo arquitetural recuperado
6. Identificação de elementos não presentes no modelo arquitetural recuperado
7. Identificação de novas relações entre elementos arquiteturais
8. Identificação de relações entre elementos arquiteturais não presentes no modelo arquitetural recuperado

O arquiteto inicia a atividade de comparação por identificar os elementos arquiteturais comuns em ambos os modelos. Uma vez identificados, o arquiteto deve examiná-los

buscando diferenças entre os elementos no modelo arquitetural de especificação e os mesmos no modelo arquitetural recuperado. A análise de diferenças é discutida com mais detalhes na Seção 3.2.2, contudo, deve-se ressaltar, que faz parte da análise de diferenças identificar elementos arquiteturais presentes em apenas um dos modelos.

Como exemplo, no caso de um elemento estar presente apenas no modelo arquitetural de especificação, o arquiteto poderá ter encontrado uma falha de implementação. Um elemento, por exemplo, um componente arquitetural deixou de ser implementado e seus requisitos foram implementados como parte de algum outro componente (ou componentes), ou, em uma situação menos otimista, tais requisitos nunca foram considerados durante o desenvolvimento.

Grandes projetos de software podem conter dezenas de componentes arquiteturais, cada um deles, por sua vez, podem conter várias interfaces. Estas relacionam-se com outras interfaces de outros componentes, multiplicando o número de conectores arquiteturais existentes no modelo. A fim de reduzir a complexidade de tais modelos, é possível utilizar modelos arquiteturais com subcomponentes. Tais modelos abstraem um conjunto de componentes que interajam entre si com certa coesão por um só componente cujas interfaces expõe as interfaces dos subcomponentes.

Ainda assim, cada componente composto por subcomponentes possuirá um modelo arquitetural que inevitavelmente o arquiteto deverá analisar. Portanto, a atividade de comparação de modelos arquiteturais, como a atividade de recuperação anteriormente discutida, demanda do arquiteto um grande esforço. Discutiremos detalhadamente na Seção 3.2.2 como o processo de análise de diferenças pode ser automatizado, reduzindo o esforço necessário para examinar modelos arquiteturais.

3.1.4 Ativ. 3: Adequação da arquitetura recuperada

Possuindo o resultado da análise de diferenças, o arquiteto pode iniciar a atividade de adequação. O objetivo desta atividade é tornar nula a diferença entre os dois modelos (o modelo recuperado e o modelo de especificação). Para isso, tanto é válido alterar o modelo especificado ou o recuperado.

Alterar o modelo recuperado significa que a implementação precisa ser adequada, que de fato existe um problema de envelhecimento da arquitetura que precisa ser sanado. Já uma mudança no modelo de especificação significa que o arquiteto aceitou a solução adotada pelo engenheiro de software, ao implementar o sistema, e que a especificação da arquitetura necessita ser alterada.

Este último caso (alteração do modelo de especificação) não é em todo absurdo. A arquitetura especificada pode não atender a requisitos descobertos durante a fase de desenvolvimento, por exemplo, o que não é incomum no processo de desenvolvimento de

software. Novos requisitos ou alteração de requisitos durante o processo pode necessitar que alteração da arquitetura sejam feitas, a fim de melhor atendê-los.

Nada impede que durante a atividade de adequação, o arquiteto necessite alterar ambos os modelos a fim de eliminar as diferenças. É possível que certas diferenças sejam devidas a falhas de especificação da arquitetura, enquanto que outras a falhas de implementação.

A atividade de adequação compreende, para cada diferença encontrada na atividade de comparação, as seguintes tarefas:

1. Ponderação sobre a diferença entre os modelos

2. Decisão sobre o modelo a ser adequado

3. Aplicação das mudanças sobre o modelo escolhido

Para cada diferença encontrada durante a atividade anterior (comparação), o arquiteto deve realizar as alterações necessárias em um dos modelos. Cada iteração desta atividade deve resolver uma diferença, reduzindo o número de diferenças a serem resolvidas na próxima iteração.

Fica a critério do arquiteto quais diferenças devem ser priorizadas e resolvidas. Não existe imposição, quando da definição do método, no número de diferenças que devem ser resolvidas ou quantas iterações devem ser feitas nesta atividade. Devido às restrições de orçamento, tempo ou escopo, o arquiteto pode ver-se limitado a escolher um subconjunto das diferenças a resolver. Entretanto, o arquiteto deve evitar simplesmente incorporar as diferenças encontradas no modelo de especificação, sem a devida ponderação, pois, com isso, ele estará incorporando à arquitetura possíveis problemas de desvio e erosão arquiteturais, que com o tempo, degradarão a vida útil do sistema.

Na atividade de adequação é limitada a capacidade de automatização, pois ela demanda conhecimentos específicos do arquiteto não presentes nos modelos envolvidos no processo, como conhecimentos dos requisitos e do domínio. O arquiteto é o responsável por tomar as decisões de adequação arquitetural com base em seus conhecimentos do domínio, do sistema e dos requisitos.

Considerando tais aspectos, discutiremos na Seção 3.2.3 o suporte ferramental que o arquiteto possuirá para aplicar as adequações nos modelos arquiteturais.

3.1.5 Ativ. 4: Mapeamento das mudanças arquiteturais

A atividade de mapeamento tem como base as alterações realizadas pelo arquiteto na atividade anterior. As alterações executadas sobre o modelo arquitetural de especificação

não passam pela atividade de mapeamento. Elas apenas representam a aceitação do arquiteto em incorporar os desvios de implementação como parte da arquitetura do sistema. Desta forma, mudanças no modelo de especificação devem ser levadas em conta a fim de gerar uma nova versão do modelo de especificação da arquitetura do sistema.

Já as mudanças relacionadas ao modelo recuperado precisam ser mapeada em mudanças aplicáveis ao código-fonte do sistema. Tais alterações são necessárias para tornar a implementação do software concordante com o modelo arquitetural de especificação.

As tarefas desta atividade, para cada alteração do modelo de implementação obtidas na atividade anterior, estão listadas abaixo:

1. Identificação dos elementos arquiteturais diretamente afetados pela mudança
2. Identificação de elementos adicionados ou removidos ao modelo
3. Identificação de elementos afetados indiretamente pela mudança
4. Identificação dos artefatos de implementação relacionados à mudança
5. Geração dos artefatos de implementação que materializam o resultado da mudança
6. Obtenção das diferenças entre os artefatos de implementação existentes com os gerados na tarefa anterior
7. Aplicação das diferenças nos artefatos existentes

Elementos arquiteturais adicionados ou removidos ao modelo podem causar com que outros elementos sejam indiretamente afetados. Por exemplo, a remoção de um componente arquitetural irá implicar na remoção de conectores que a ele estejam relacionados. Similarmente, a alteração de uma interface de um componente pode forçar que alterações ocorram no conector que a relaciona com uma outra interface.

A atividade de mapeamento é simétrica à atividade de recuperação, apenas é diferenciada da anterior pelo sentido em que é realizada. Enquanto a recuperação irá gerar elementos arquiteturais baseados nos artefatos de implementação, o mapeamento irá gerar artefatos de implementação baseados nos elementos arquiteturais. Entretanto, a atividade de mapeamento pode torna-se mais complexa pois é necessário manter os artefatos de implementação existentes e apenas aplicar mudanças sobre eles. Isso introduz tarefas adicionais, não encontradas durante a atividade de recuperação.

Dada uma alteração no modelo arquitetural recuperado, esta atividade inicia-se por identificar os elementos arquiteturais afetados pela mudança, elementos adicionados e removidos, bem como elementos indiretamente afetados, como discutido anteriormente.

Uma vez identificados o conjunto de elementos arquiteturais envolvidos na mudança, é necessário identificar o conjunto de artefatos de implementação a eles relacionados.

Cada um desses conjuntos possuem domínios diferentes, o primeiro é composto por elementos de modelos arquiteturais, enquanto que o segundo são artefatos de implementação. Para ser possível obter as mudanças a serem aplicadas ao código-fonte, primeiramente obtém-se os artefatos de implementação que seriam necessários para materializar o resultado das mudanças aplicadas ao modelo arquitetural. Com isso, podemos então comparar os artefatos originais com os esperados pelo modelo arquitetural. A diferença entre eles, portanto, deve ser aplicada aos artefatos de implementação originais de forma a tornar o código-fonte concordante com o modelo arquitetural desejado.

De maneira análoga à recuperação, esta atividade demanda enorme esforço do arquiteto (e potencialmente dos engenheiros de softwares responsáveis pelo código a ser alterado). Na seção 3.2.4, discutiremos como esse processo pode ser automatizado e quais as limitações naturalmente inerentes a sua automatização.

3.1.6 Ativ. 5: Implementação das mudanças arquiteturais

Ao final da atividade de mapeamento, o arquiteto terá em mãos mais um modelo, um modelo de implementação. Esse modelo contempla as diferenças entre os artefatos de implementação existentes e os esperados, caso o código-fonte estivesse de acordo com o modelo arquitetural. Portanto, o trabalho a ser realizado durante a atividade de implementação é a geração e alteração de código-fonte, guiadas pelo modelo de implementação.

Quebrando em tarefas, esta atividade é dividida como abaixo, para cada artefato de implementação encontrado no modelo de implementação.

1. Adição, remoção ou alteração do artefato de implementação

2. Adequação do código-fonte envolvido na alteração efetuada

Dependendo das alterações realizadas no modelo arquitetural, é possível que artefatos de implementação sejam adicionados, removidos ou alterados. Após cada mudança realizada, deve verificar-se a necessidade de adaptação do código existente a fim de comportar tal mudança. Por exemplo, a remoção de uma interface pode materializar a remoção de uma classe no código-fonte, que por sua vez pode requerer que o código-fonte que utiliza essa classe (potencialmente em algum conector arquitetural) seja alterada de acordo.

Como discutido nas atividades de recuperação e mapeamento, o uso de um modelo de implementação de componentes é fundamental para tornar prática tais atividades. O mesmo aplica-se à atividade de implementação. Dependendo do modelo de implementação de componentes utilizado e das características de tal modelo, maior ou menor será o

trabalho de adequação do código-fonte após uma dada mudança. Isso se deve ao fato de modelos de implementação de componentes restringirem o acoplamento entre artefatos de implementação que materializam componentes distintos. Desta forma, um modelo de implementação de componentes bem especificado permitirá que alterações de artefatos do código-fonte afetem apenas o componente no qual se encontram.

De maneira similar a outras atividades, a implementação pode ser automatizada, contudo até certo ponto. Será discutido na Seção 3.2.5 as possibilidades de automatização, suas limitações e qual a intensidade necessária de intervenção dos engenheiros de software para garantir a corretude das mudanças.

3.1.7 Ativ. 6: Versionamento dos modelos arquiteturais

O versionamento, apesar de classificada como última atividade do método, poder ser considerada ortogonal às demais atividades. O objetivo desta atividade é oficializar a necessidade de manter registro das atividades realizadas durante o desenvolvimento e manutenção do sistema de software. A análise de versões anteriores dos modelos arquiteturais do sistema facilita o entendimento da evolução do software. Bem como ajuda a obter métricas para analisar a eficiência do desenvolvimento e a análise póstuma do projeto, permitindo à equipe entender quais fatores causaram alterações na arquitetura do sistema, sua intensidade e o impacto de tais mudanças sobre o sistema.

É de autonomia de cada equipe decidir quais modelos versionar e quando versioná-los. Como sugestão, para cada iteração do método, recomendamos manter os modelos resultantes das fases de recuperação e adequação. Tais modelo permitirão o tipo de análise mencionados anteriormente.

Modelos de outras fases podem ser versionados, contudo como discutiremos nas próximas seções, os demais modelos poderão ser obtidos de maneira automatizada a partir dos modelos anteriormente sugeridos.

3.2 Detalhamento do método Meissa

Nesta seção serão discutidas as possibilidades de automatização das várias etapas do método Meissa. Cada subseção detalha os requisitos que permitem a automatização possível e as etapas necessárias para tal. Bem como, quando apropriado, são descritas as limitações inerentes à automatização.

3.2.1 Ativ. 1: Recuperação da arquitetura

Como discutido na Seções 2.1.1 e 1.3.1, a recuperação da arquitetura de um sistema arbitrário é uma tarefa extremamente árdua, imprecisa e que demanda grande dedicação do arquiteto de software. Contudo, ao utilizar um modelo de implementação de componentes (Seção 2.5), essa atividade torna-se precisa, reprodutível e gerenciável e, com isso, ela torna-se *passível de automação*.

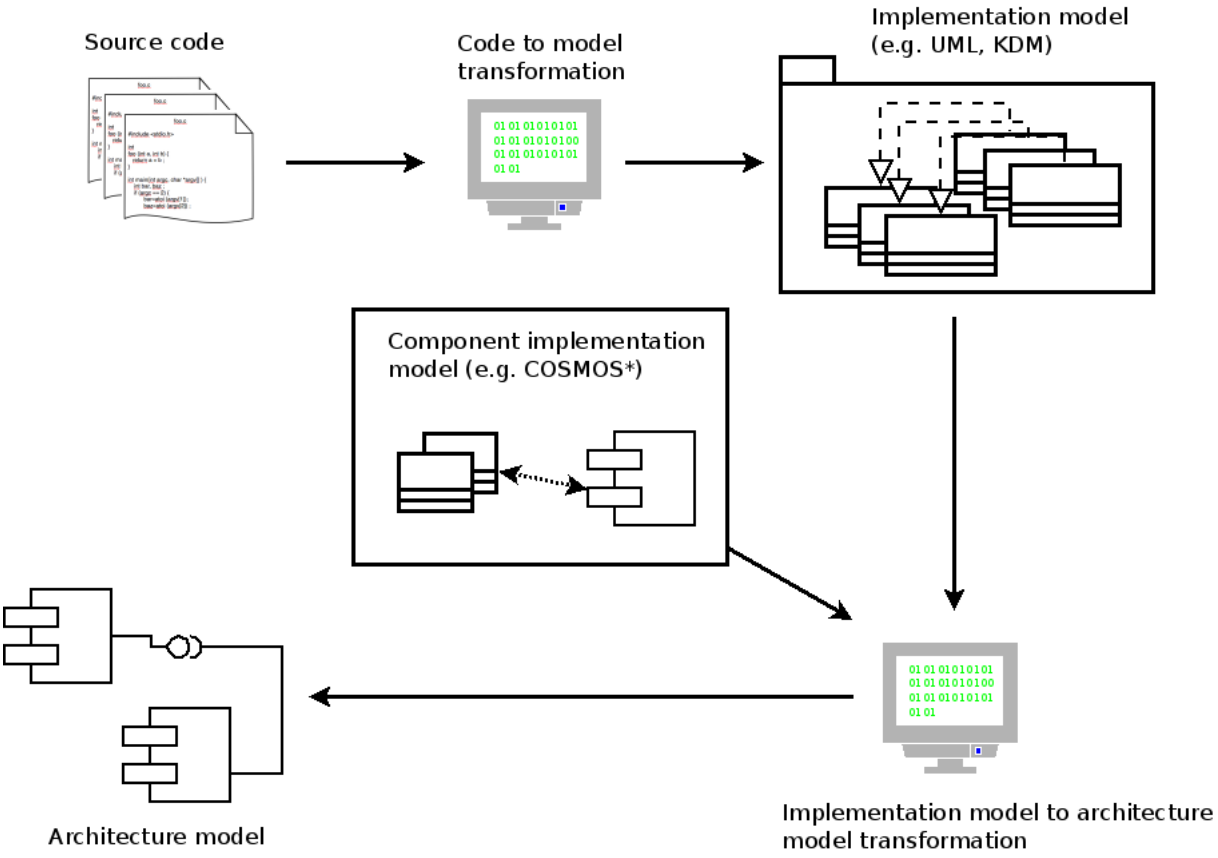


Figura 3.3: Diagrama representando as transformações de modelos para realizar a recuperação arquitetural baseando-se no código-fonte.

Esta atividade pode ser resumida no diagrama 3.3, que ilustra as transformações de modelos necessárias para a recuperação da arquitetura, com base no código-fonte do sistema e no modelo de implementação de componentes adotado. Durante o restante desta seção, será discutido em detalhes o processo de automatização da atividade de recuperação.

Um modelo bem estruturado de implementação de componentes introduzirá um mapeamento unívoco entre elementos arquiteturais e artefatos de implementação. Uma vez que tal modelo de componentes seja adotado, e o código-fonte do sistema seja implementado

seguindo suas premissas, é, portanto, possível aplicar o mapeamento em ambas direções. Isto é, é possível definir os artefatos de implementação relacionados a um certo elemento arquitetural, bem como apontar qual elemento arquitetural é implementado por um certo conjunto de artefatos de implementação.

Certamente é possível que um modelo de implementação de componentes não proveja um mapeamento unívoco entre a arquitetura e a implementação, como mostra a Figura 3.4. Isso significa que existem artefatos de implementação que materializam mais de um elemento arquitetural. Isso trará desvantagens à automatização do processo de recuperação da arquitetura, devido ao fato da análise do código-fonte tornar-se mais complexa para permitir a distinção das relações envolvidas entre o artefato de implementação e seus componentes arquiteturais envolvidos. Potencialmente, o fato de um artefato de implementação não poder ser mapeado a apenas um elemento arquitetural, impedirá a automatização da recuperação da arquitetura, necessitando a intervenção do arquiteto para distinguir as relações implícitas ao modelo de componentes.

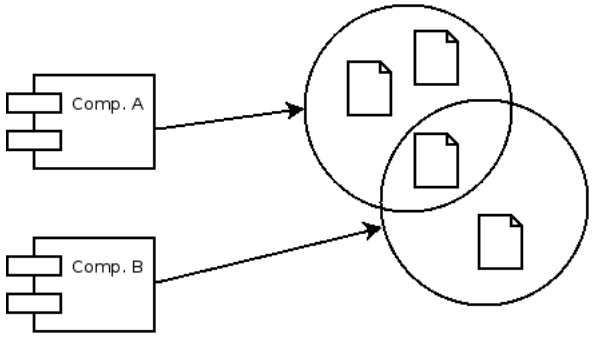


Figura 3.4: Mapeamento não unívoco entre elementos arquiteturais e artefatos de implementação.

O processo de automatização de recuperação arquitetural proposto é fundamentado em transformações de modelos [50]. O mapeamento entre elementos arquiteturais e artefatos de implementação definidos por um modelo de implementação de componentes pode ser esquematizado como uma transformação de um modelo arquitetural em um modelo mais próximo ao código-fonte, por exemplo, um modelo de classes, como o diagrama de classes em UML [40]. De mesma forma, o código-fonte pode ser visitado a fim de se gerar um modelo de mais alto nível nele baseado, como por exemplo, um outro modelo de classes.

Neste ponto, o domínio de ambos modelos é o mesmo e certas operações tornam-se possíveis, como por exemplo, sua comparação.

Por um lado, temos o modelo de classes gerado a partir do modelo de implementação de componentes. De fato, este é um metamodelo [49], uma vez que ele especifica como um outro modelo deve parecer-se. Por outro, temos o modelo representando o código-fonte

existente. Uma vez que pudermos reconhecer o padrão especificado pelo metamodelo neste segundo, teremos encontrado a implementação de um elemento arquitetural, isto é, teremos recuperado um elemento arquitetural a partir do código-fonte.

De forma simplista, pode-se resumir a recuperação de elementos arquiteturais como a execução de um algorítimo de *casamento de padrões*[5] [4] sobre o modelo de classes. A Figura 3.5 ilustra essa operação.

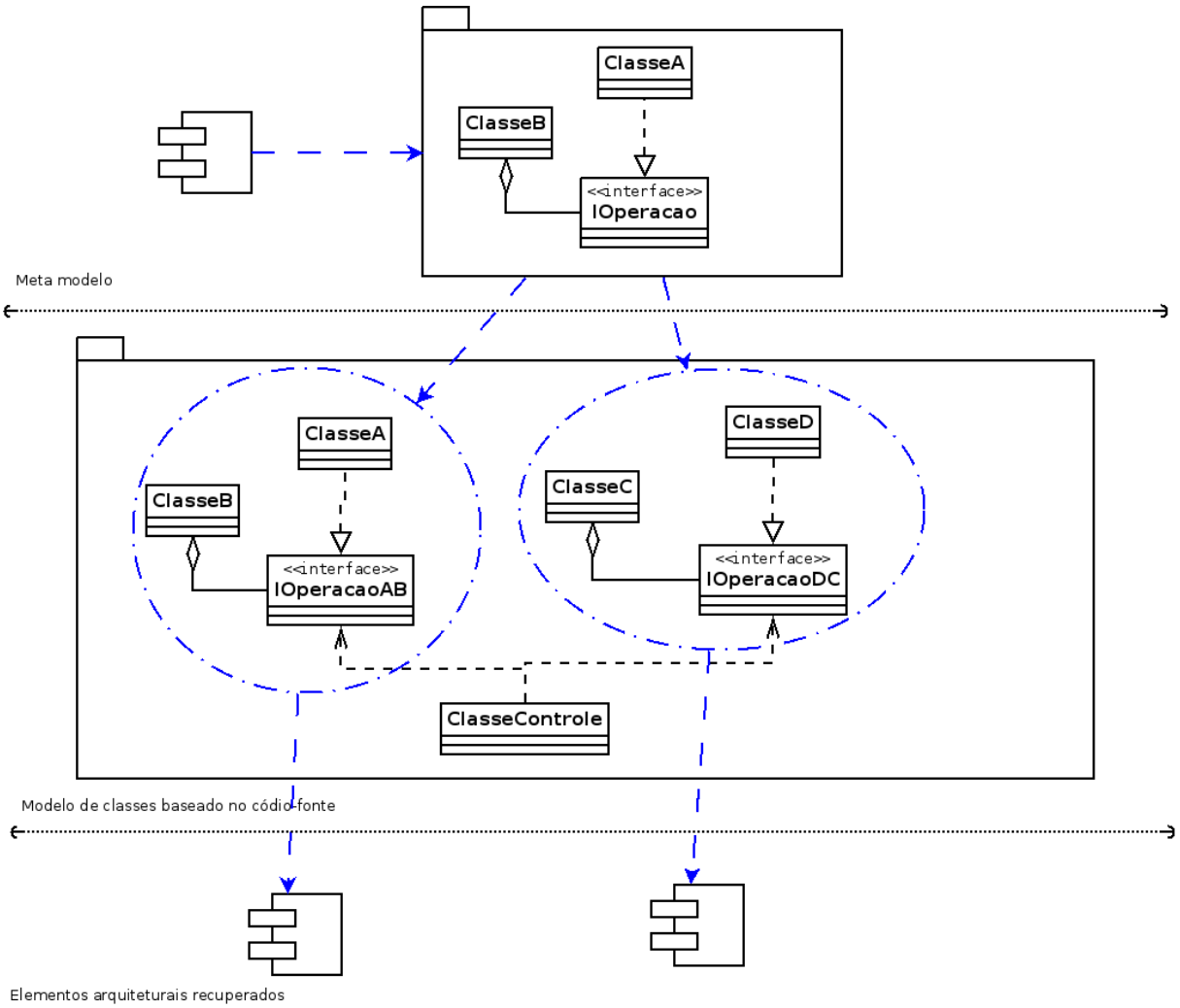


Figura 3.5: Exemplo de metamodelo e casamento de padrões executado no modelo de classes.

Assim como o modelo de implementação de componentes define como deve ocorrer a implementação de componentes arquiteturais no nível de implementação, tal modelo descreverá como deve ser materializada relações entre elementos arquiteturais. Portanto,

[5]Do inglês, *pattern-matching*.

recuperar o relacionamento de elementos arquiteturais tendo-se o código-fonte pode ser feito de maneira muito similar a recuperação dos elementos arquiteturais em si. A partir do modelo de implementação de componentes, pode-se obter um metamodelo descrevendo como a interação entre os elementos arquiteturais deve ocorrer no nível de implementação. De maneira similar a realizada durante a recuperação dos elementos arquiteturais, utilizando o modelo que representa o código-fonte e o metamodelo de relações entre os elementos, executa-se um algoritmo de *casamento de padrões* cujo resultado será uma coleção de artefatos de implementação que materializam as relações entre elementos arquiteturais.

Ao final destas duas etapas, é possível construir o modelo arquitetural recuperado, unindo-se os elementos arquiteturais recuperados e suas relações. Ao mesmo tempo, tem-se o mapeamento entre o modelo arquitetural e os artefatos de implementação. Sendo este o resultado da atividade de Recuperação, ele servirá de entrada para a atividade subsequente, na qual a identificação de diferenças entre modelos arquiteturais deverá ocorrer.

3.2.2 Ativ. 2: Comparação entre arquitetura recuperada e especificada

A atividade de comparação baseia-se em dois modelos, o modelo arquitetural recuperado (na atividade anterior) e o modelo arquitetural de especificação, como detalhado na Seção 3.1.3. O resultado desta atividade é um conjunto de diferenças entre os dois modelos. Para que seja possível automatizar este processo, é necessário caracterizar como os modelos são interpretados e como se define as diferenças entre dois elementos dos modelos.

Cada modelo arquitetural pode ser encarado como um grafo, cujos vértices são os elementos arquiteturais e as arestas, relacionamentos entre seus elementos. Assim, cada vértice poderá ser um componente, um conector ou interface arquitetural. E cada aresta representará o relacionamento entre uma interface e um conector arquitetural ou entre uma interface e um componente arquitetural. Qualquer outra aresta entre outros dois elementos arquiteturais é inválida, já que uma interface arquitetural só pode pertencer a um componente arquitetural e apenas conectores arquiteturais podem relacionar interfaces com componentes que as utilizem.

Uma vez que ambos os modelos arquiteturais sejam representados em forma de grafo, o passo seguinte na automação da atividade é compará-los. Dados dois vértices, um em cada grafo, eles representam o mesmo elemento arquitetural se: (i) ambos são de mesmo tipo (componente arquitetural, conector arquitetural, interface provida ou interface requerida) e (ii) são identificados em ambos os modelos por mesmo nome.

A fim de obter a diferença entre os modelos arquiteturais, precisamos obter os resultados para cada uma das tarefas que compõe a atividade de comparação, descritas

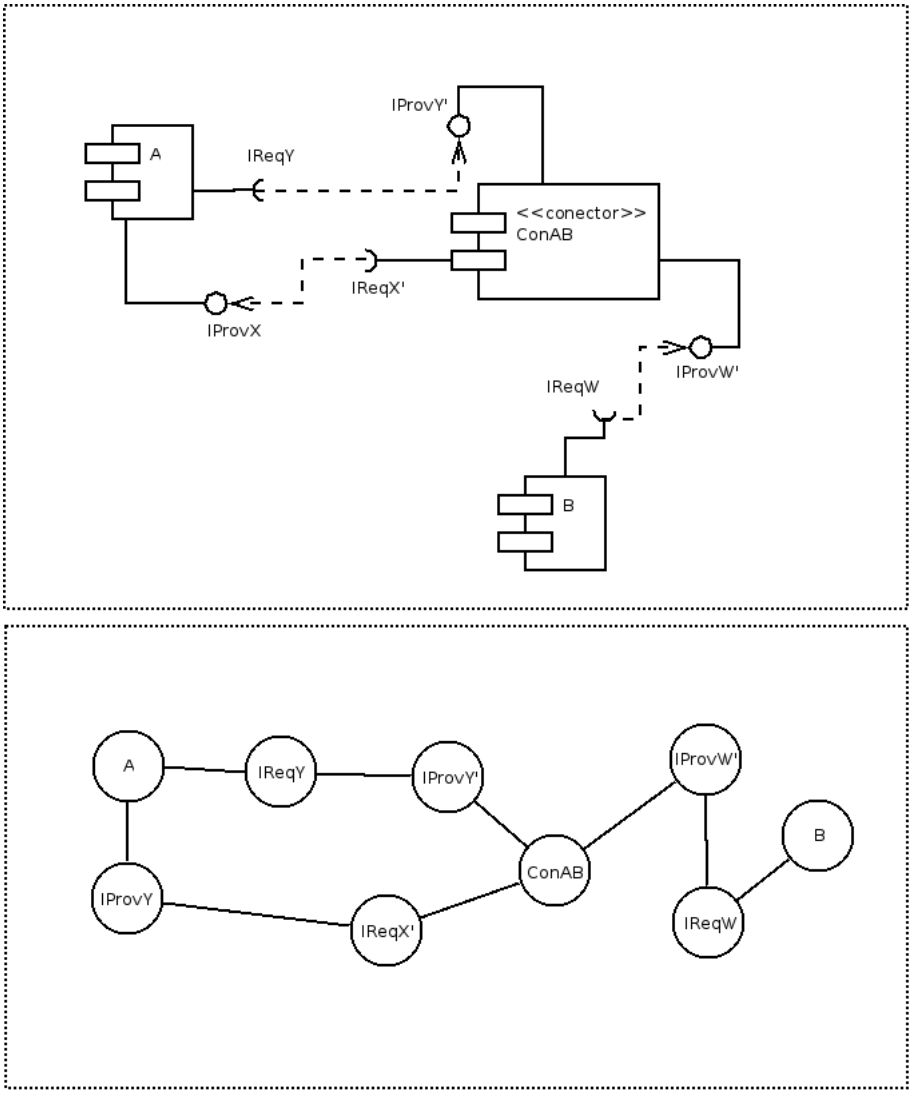


Figura 3.6: Exemplo de modelo arquitetural representando como um grafo.

previamente na seção 3.1.3 e repetidas abaixo. A fim de facilitar a esquematização de tais resultados, definimos a seguinte convenção:

R é o grafo obtido a partir do modelo arquitetural recuperado. E, o obtido a partir do modelo arquitetural especificado. $V_T(G)$ é o conjunto de vértices cujo tipo é T, por exemplo $V_{COMPONENTE}(R)$ é o conjunto de vértices cujo tipo representam componentes arquiteturais do grafo obtido a partir do modelo arquitetural recuperado. Por fim, $E(G)$ é o conjunto de arestas de um grafo G, o símbolo $\setminus$ representa a diferença de conjuntos e o símbolo $\bigcap$ a intersecção de conjuntos.

1. **Identificação de componentes presentes em ambos modelos:**

$V_{COMPONENTE}(R) \bigcap V_{COMPONENTE}(E)$

2. **Identificação de interfaces presentes em ambos modelos:**
   $V_{INTERFACE}(R) \bigcap V_{INTERFACE}(E)$

3. **Identificação de conectores presentes em ambos modelos:**
   $V_{CONECTOR}(R) \bigcap V_{CONECTOR}(E)$

4. **Identificação de novos elementos arquiteturais no modelo arquitetural recuperado:**
   $V(R) \setminus V(E)$

5. **Identificação de elementos não presentes no modelo arquitetural recuperado:**
   $V(E) \setminus V(R)$

6. **Identificação de novas relações entre elementos arquiteturais:**
   $E(R) \setminus E(E)$

7. **Identificação de relações entre elementos arquiteturais não presentes no modelo arquitetural recuperado:**
   $E(E) \setminus E(R)$

8. **Identificação de diferenças entre elementos arquiteturais existentes em ambos modelos:**

   Para os itens 1, 2 e 3, é necessário identificar se tais elementos arquiteturais, apesar de existirem tanto no modelo recuperado quanto no especificado, possuem alguma diferença, e se, de fato, representam o mesmo artefato com mesmas características em ambos os modelos. Para tanto, precisamos levar em consideração as especificidades de cada tipo de elemento arquitetural.

   (a) **Interfaces:** Duas interfaces serão iguais se compartilharem o mesmo identificador, pertencerem ao mesmo componente (ou conector) arquitetural, serem de mesmo tipo (interface provida ou requerida) e possuírem o mesmo conjunto de operações.

   (b) **Componentes:** Dois componentes serão iguais se compartilharem o mesmo identificador, proverem o mesmo conjunto de interfaces, requirirem o mesmo conjunto de interfaces e, em caso de possuírem subcomponentes, que suas arquiteturas internam sejam idênticas.

(c) **Conectores:** Utiliza-se o mesmo processo de comparação de componentes.

Em caso de comparação de componentes compostos, compará-los é comparar suas arquiteturas subjacentes, tornando a atividade de comparação recursiva. Isto é, a comparação da arquitetura recuperada de um componente composto contra sua arquitetura especificada, nada mais é que a execução de uma nova atividade de comparação, cujos modelos de entrada são: (i) a arquitetura recuperada e (ii) a especifica de tal componente composto.

3.2.3 Ativ. 3: Adequação da arquitetura recuperada

Foi mencionada, na Seção 3.1.4, a restrita possibilidade de automação desta atividade. Em resumo, durante a atividade de adequação, o arquiteto precisa decidir como modificar a arquitetura do sistema, seja ela a recuperada, a especificada, ou ambas.

O apoio computacional que pode ser provido ao arquiteto limita-se a ferramentas de desenho e edição de modelos, bem como ferramentas verificadoras, que ajudem o arquiteto a construir um modelo de acordo com as especificações da linguagem ou notação de modelagem utilizada.

3.2.4 Ativ. 4: Mapeamento das mudanças arquiteturais

A atividade de mapeamento e a de recuperação são as duas atividades do método que mais se beneficiam do apoio computacional. Sua automação é também uma das mais vantajosas ao método, pois reduzem drasticamente a complexidade das operações realizadas pelo arquiteto, a propensão de erro humano e o tempo demandado em sua execução.

Como mencionado na Seção 3.1.5, esta atividade é muito similar a atividade de recuperação, mas em sentido oposto. Inicia-se com o modelo arquitetural ajustado pelo arquiteto e o objetivo é atingir o código-fonte do sistema, com as modificações necessárias para refletir os ajustes realizados pelo arquiteto, no nível arquitetural. Esta atividade, encarada como uma sequência de transformação de modelos, é representada na Figura 3.7.

Considerando que esta atividade tem como entrada o modelo arquitetural recuperado ajustado pelo arquiteto, as seguintes operação precisam ser realizadas durante a automação desta atividade:

1. Utilizando-se do mesmo processo descrito na atividade de Comparação, obtém-se o modelo de diferenças arquiteturais entre o modelo arquitetural recuperado ajustado e o modelo arquitetural originalmente recuperado. Com isso, o resultado obtido são as diferenças aplicadas pelo arquiteto, ou seja, as alterações arquiteturais que precisam ser realizadas no sistema.

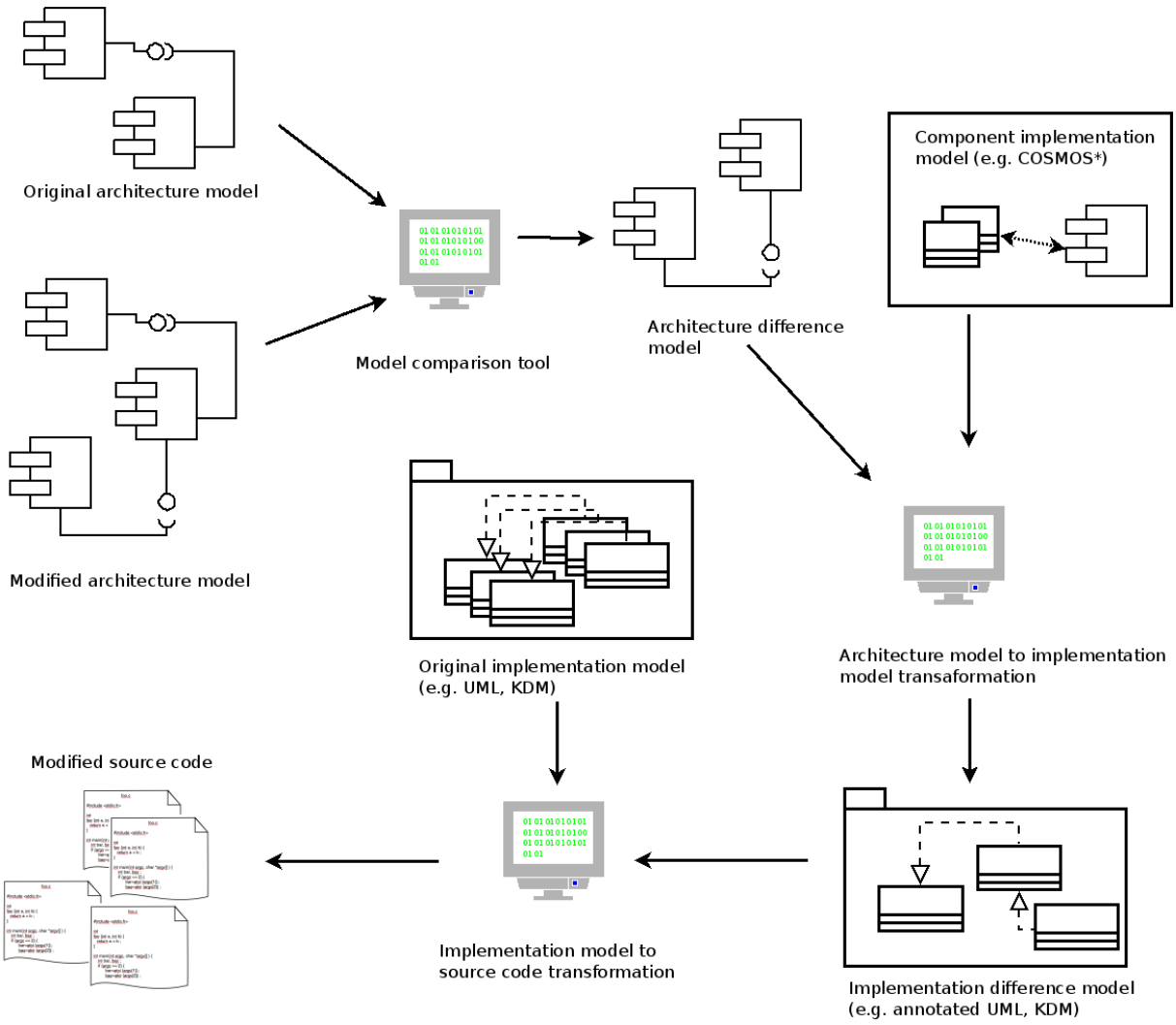


Figura 3.7: Sequência de transformações de modelos para o mapeamento de mudanças arquiteturais em mudanças no código-fonte.

2. Possuindo-se o modelo de implementação de componentes, similarmente ao realizado na atividade de Recuperação, utiliza-se seu metamodelo para guiar a transformação entre elementos arquiteturais em elementos de um modelo mais próximo ao de implementação, como, por exemplo, o modelo de classes.

3. Comparando-se o modelo de classes obtido no item anterior, com o modelo de classes do sistema recuperado durante a atividade de Recuperação, obtém-se as diferenças que necessitam ser realizadas no nível de implementação.

No item 3, o processo de comparação pode ser realizado de maneira similar ao utilizado na atividade de comparação. Entretanto, quanto mais próximo da implementação é o mo-

delo, mais ele necessitará representar maior quantidade de detalhes. Desta forma, certas adaptações serão necessárias ao processo de comparação. Por exemplo, será necessário, durante a transformação do modelo em grafo, anotar as arestas, de forma a representar as diferentes possíveis relações entre dois elementos. Ainda, dependendo do metamodelo imposto pelo modelo de implementação de componentes, outras características dos elementos precisarão ser comparadas, como o nível de visibilidade de uma classe ou a assinatura dos métodos de uma interface.

3.2.5 Ativ. 5: Implementação das mudanças arquiteturais

Uma vez completada a atividade de Mapeamento, tem-se em mãos um modelo que representa as diferenças entre dois modelos de classes. A partir dele, é possível a geração automática de código que materializa as diferenças encontradas no modelo de classes em mudanças no código-fonte do sistema.

3.2.6 Ativ. 6: Versionamento dos modelos arquiteturais

De maneira similar a atividade de Adequação, o apoio computacional possível nesta atividade restringe-se ao armazenamento dos modelos construídos e recuperados durante as prévias atividades. Idealmente, deseja-se que o versionamento dos modelos ocorra juntamente com o código-fonte. Desta forma, é possível acompanhar a evolução dos sistema, tanto considerando os artefatos de implementação, quanto os modelos arquiteturais. Uma maneira de manter o versionamento dos modelos e do código-fonte em sincronia é usar a mesma ferramentas de versionamento de código para ambos.

3.3 Resumo

Este capítulo descreve o método de apoio à evolução arquitetural Meissa. O método é composto de seis atividades: (i) ele inicia-se com a **recuperação** do modelo arquitetural do sistema com base no código-fonte; (ii) a seguir, a atividade de **comparação** examina o modelo arquitetural recuperado e o modelo originalmente especificado pelo arquiteto a fim de identificar problemas de desvio e erosão arquitetural; (ii) a terceira atividade é a de **adequação** do modelo arquitetural implementado com o objetivo de corrigir os problemas identificados na atividade anterior; (iv) na próxima atividade, é realizado o **mapeamento** das mudanças arquiteturais em mudanças aplicáveis ao código-fonte; (v) com isso é possível realizar a **implementação** das mudanças no código-fonte do sistema de forma a aplicar as correções necessárias à arquitetura do sistema; (vi) por fim, realiza-se o

versionamento dos modelos arquiteturais envolvidos durante a execução das atividades anterior.

Também foram descritos os pontos principais relacionados a automação do método. Dentre as atividades anterior, as atividades de **adequação** e **implementação** são as única que requerem intervenção humana. O arquiteto ao executar a atividade de adequação precisa utilizar seu conhecimento sobre o domínio e a arquitetura para decidir quais diferenças entre o modelo arquitetural recuperado e o modelo arquitetural originalmente especificado devem ser classificadas como problemas de envelhecimento de software que afetam a arquitetura. Já na atividade de implementação, o arquiteto ou engenheiro de software necessita examinar as mudanças aplicadas ao código-fonte e prover implementações adequadas. Por exemplo, a adição de uma operação em uma interface arquitetural, dependendo do modelo de implementação de componentes, poderá gerar uma nova classe no código-fonte; é responsabilidade do engenheiro de software, prover a implementação das operações dessa classe, a fim de obter o corretamento funcionamento do sistema.

No próximo capítulo, a discussão sobre os detalhes de automação de cada uma das atividades do método será estendida a fim de especificar e implementar uma ferramenta que proverá o necessário apoio computacional para a execução do método Meissa.

Capítulo 4

Meissa: Ferramenta de apoio à evolução arquitetural

Meissa é, além do nome dado ao método, a ferramenta desenvolvida para torná-lo eficientemente aplicável. A ferramenta atua em conjunção com outras ferramentas desenvolvidas pelo grupo de Engenharia de Software e Tolerância a Falhas do Instituto de Computação da Universidade Estadual de Campinas, destacando-se, entre elas, Bellatrix [58].

Bellatrix é um ambiente de desenvolvimento integrado que apoia o desenvolvimento baseado em componentes [6], com ênfase na arquitetura de software, estendendo o popular ambiente de desenvolvimento Eclipse [23]. Como Meissa, Bellatrix também o nome de uma das estrelas da constelação de Orion.

Neste capítulo, a ferramenta Meissa é descrita, envolvendo desde sua concepção, especificação até os detalhes de sua implementação. Contudo, antes de apresentada a ferramenta, são descritos os elementos de infraestrutura e ambientes em que a ferramenta se alicerça.

4.1 Infraestrutura e ambientes

4.1.1 Eclipse

Eclipse é um ambiente de desenvolvimento integrado, escrito em Java. Uma de suas principais características arquiteturais é sua composição através de *plugins*. Um *plugins*, como um componente arquitetural, define suas interfaces providas, ou, na nomenclatura do ambiente, *extensões*, e interfaces requeridas, ou *pontos de extensão*. O framework do ambiente Eclipse provê um conjunto base de APIs[1] sobre as quais todo o restante das funcionalidades é construída, usando-se *plugins*.

[1]Do inglês, *Application Programing Interface*, ou interface de programação de aplicação.

Plugins são dinamicamente carregados pelo ambiente e suas extensões conectadas aos respectivos pontos de extensão. Além disso, *plugins* são capazes de avaliar seus pontos de extensão e identificar se existem extensões presentes, sendo capazes de decidir quais extensões providas por outros *plugins* devem ser utilizadas, até mesmo, não utilizar dado ponto de extensão, caso não existam extensões providas a ele.

Eclipse modeling framework

Eclipse modeling framework[2], ou EMF, é um conjunto de *plugins* que estende o ambiente Eclipse com capacidades de modelagem [1]. A base do EMF é a definição de um metamodelo, sobre o qual é possível modelar sistemas de software orientado a objetos e através do *framework*, realizar a geração automática de código para os modelos desenhados.

O código gerado pelo EMF não depende do ambiente Eclipse durante sua execução e pode ser utilizado em qualquer aplicação *Java*.

Como será discutido na próxima seção, o projeto Bellatrix aproveita-se da infraestrutura provida por este framework de modelagem e a utiliza como base para construir seus próprios modelos, reduzindo o custo de desenvolvimento da ferramenta. De maneira similar, Meissa, ao estender a ferramenta Bellatrix, integra-se com o EMF, definindo também seus próprios modelos utilizando a mesma infraestrutura.

4.1.2 Ambiente Bellatrix

Bellatrix é uma extensão do ambiente Eclipse, cujo objetivo é prover ferramentas que guiam o desenvolvimento baseado em componentes, com foco na arquitetura de software [58]. Com essa extensão, o arquiteto é provido de editores gráficos, nos quais ele pode desenhar componentes e suas interfaces, a partir destes, compor arquiteturas e, por fim, ativar a geração de código-fonte.

Sendo uma extensão do ambiente Eclipse, Bellatrix é composto por um conjunto de *plugins*. Cada *plugins* do ambiente Bellatrix é responsável por certa funcionalidade como, por exemplo, prover ao arquiteto uma interface gráfica para desenhar diagrama de arquiteturas ou persistir um dado modelo em disco. Assim, alguns *plugins* podem ser removidos que o ambiente ainda continuará funcional, perdendo apenas a funcionalidade provida por aquele *plugin*, dado que não haja outros *plugins* que dependam dele. Essa é uma das características fundamentais da plataforma Eclipse que permite enorme flexibilidade e modularidade às ferramentas que sobre ela são construídas.

[2] *Framework* de modelagem Eclipse

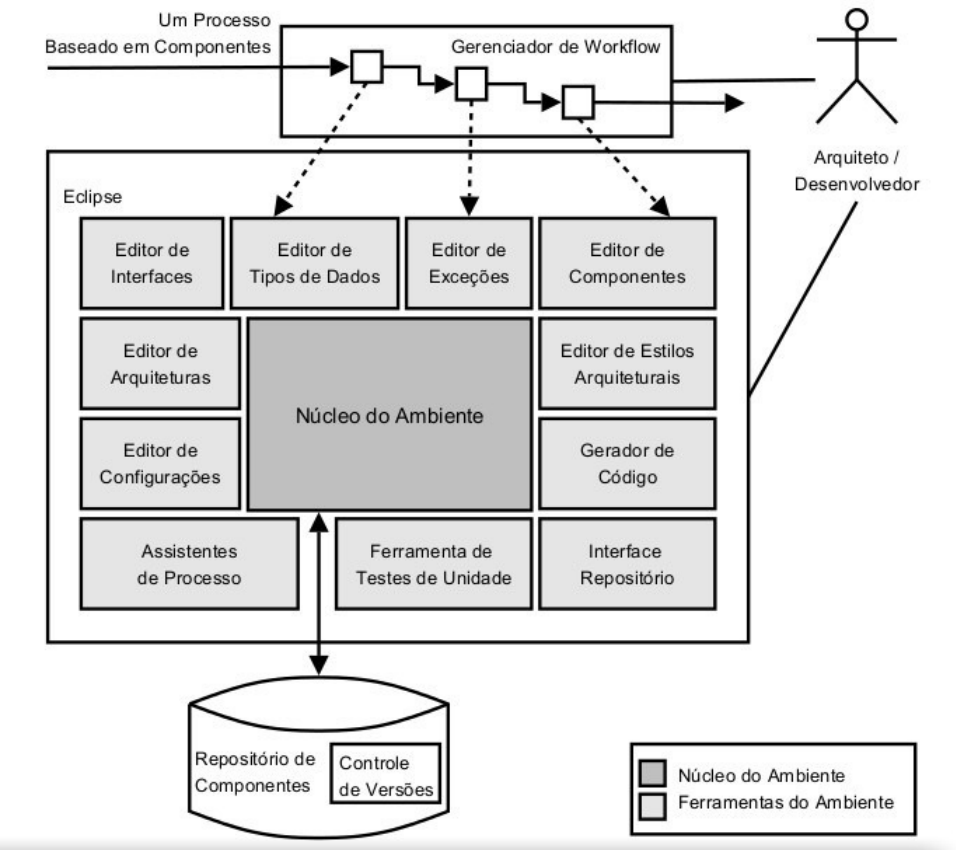


Figura 4.1: Visão geral do ambiente Bellatrix [58].

4.2 A ferramenta Meissa

O objetivo da ferramenta Meissa é a automação das atividades do método Meissa, descritas na Seção 3.1 e cujos detalhes de automação foram discutidos na Seção 3.2. Com o intuito de aproveitar-se de funcionalidades já existentes, Meissa foi construída sobre o ambiente Bellatrix, utilizando-se de sua infraestrutura e provendo capacidades adicionais de forma a tornar a aplicação do método Meissa extremamente simples e pragmática.

A utilização do ambiente Bellatrix segue na linha de outros trabalhos do grupo de Engenharia de Software e Tolerância a Falhas do Instituto de Computação da Universidade Estadual de Campinas, nas áreas de arquitetura de software, ferramentas e evolução de sistemas baseados em componentes [56, 58, 55, 42]. Devido à natureza modular do ambiente Eclipse (e por consequência, do Bellatrix), todas essas solução podem ser utilizadas em conjunto, provendo ao arquiteto de software um arcabouço para o desenvolvimento de software baseado em componentes, com foco na arquitetura de software.

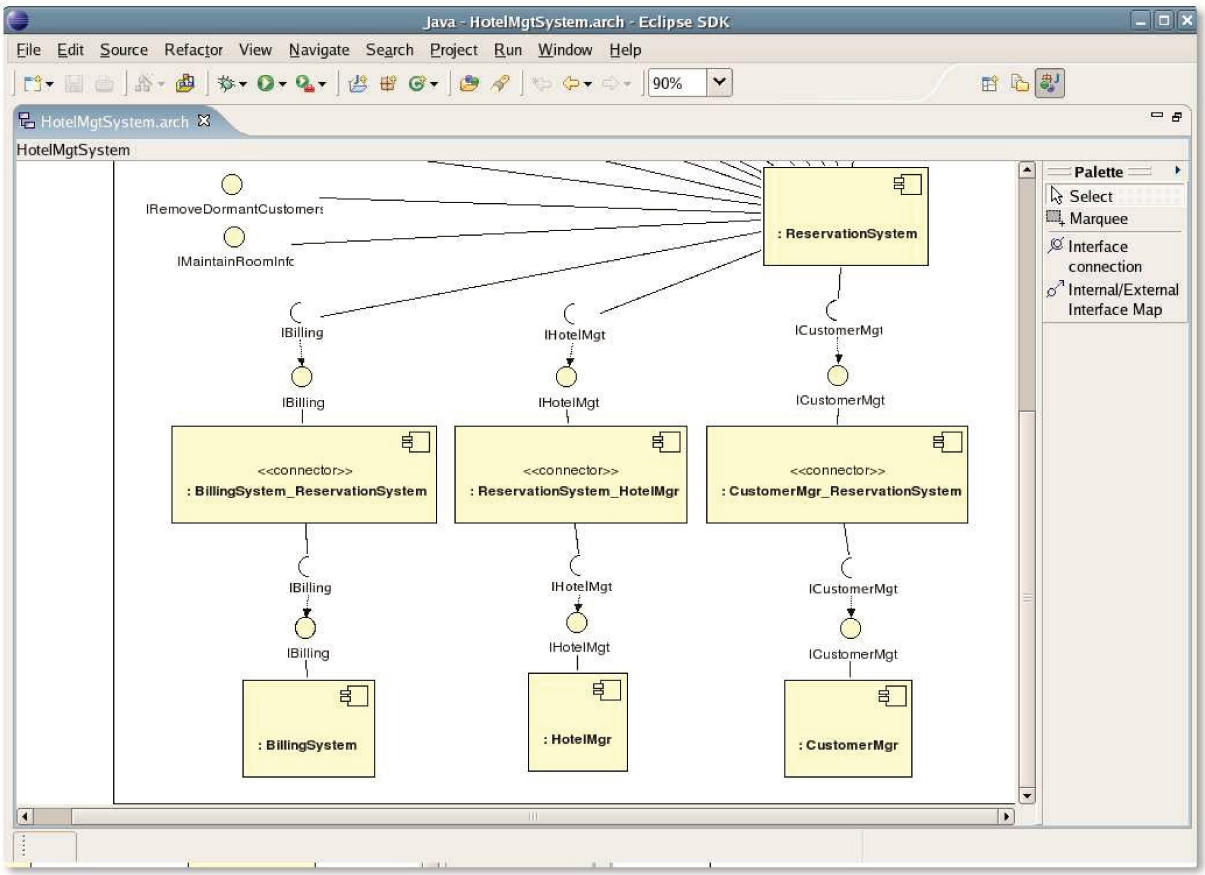


Figura 4.2: Editor gráfico de diagrama de arquitetura provido pelo ambiente Bellatrix [58].

O desenvolvimento da ferramenta Meissa segue um processo de desenvolvimento baseado em componentes, mais especificamente o UML Components, como discutido na Seção 2.4. Desta forma, o restante deste capítulo foca-se em descrever a especificação, projeto e implementação da ferramenta Meissa, seguindo tal processo de desenvolvimento de componentes. Na Seção 4.3 será especificada a ferramenta, enumerando-se seus requisitos e restrições. Na Seção 4.3.4, serão descritos os casos de uso levantados para a ferramenta. Na Seção 4.4 será detalhado o projeto arquitetural da ferramenta e, por fim, na Seção 4.6 serão discutidos os detalhes de implementação da ferramenta.

4.3 Especificação da ferramenta Meissa

A ferramenta Meissa foi concebida para tornar a aplicação do método prática, removendo atividades repetitivas, propensas a error e fatídicas da responsabilidade dos arquitetos

e engenheiros de software que conduzem o projeto. O método baseia-se fortemente em modelos de implementação de componentes. Como mencionado na Seção 2.5, existe uma grande variedade desses modelos e portanto não faria sentido restringir a ferramenta a apenas um modelo de implementação de componentes específico. Desta forma, apesar de o modelo de implementação de componentes COSMOS* ter sido o elegido para a implementação inicial da ferramenta, ela foi especificada e arquitetada para que seja extensível a qualquer modelo de implementação de componentes.

4.3.1 Especificação dos requisitos funcionais

De forma simplista, espera-se da ferramenta apenas uma coisa: permitir a execução do método Meissa de forma prática e eficiente. Contudo, para facilitar as atividades de especificação e projeto da ferramenta, os seguintes requisitos foram levantados, com base no que foi discutido na Seção 3.1, na qual o método Meissa foi apresentado e cada uma de suas atividades extensamente descritas.

RF-1 Caso o código-fonte não se adeque às regras de implementação do modelo de componentes, a ferramenta deverá informar as razões da não conformidade.

RF-2 A ferramenta deverá ser capaz de reconhecer cada componente da arquitetura, bem como suas interfaces, baseando-se no modelo de implementação de componentes utilizado.

RF-3 A ferramenta deverá ser capaz de reconhecer as relações entre os componentes, ou seja, ser capaz de conhecer conectores e identificar as interfaces que participam das relações arquiteturais.

RF-4 A ferramenta deverá ser capaz de recuperar a arquitetura de um sistema baseado em componentes.

RF-5 A ferramenta deverá ser capaz de gerar um modelo arquitetural da arquitetura recuperada.

RF-6 A ferramenta deverá ser capaz de importar o modelo arquitetural da etapa de projeto do sistema.

RF-7 A ferramenta deverá ser capaz de comparar o modelo arquitetural de projeto com o modelo arquitetural de implementação e reportar as inconsistências.

RF-8 A ferramenta deverá permitir ao arquiteto alterar o modelo arquitetural.

RF-9 A ferramenta, com base nas alterações feitas ao modelo arquitetural, deverá gerar o código-fonte que materializa as alterações arquiteturais.

RF-10 A ferramenta deverá ser capaz de exibir visualmente o modelo arquitetural, bem como permitir que as edições sejam feitas de forma visual.

RF-11 A ferramenta deverá armazenar os modelos arquiteturais para consultas históricas.

RF-12 A ferramenta deverá integrar-se ao ambiente de desenvolvimento Eclipse [23] e ao ambiente Bellatrix [58].

A decisão de tornar a ferramenta Meissa integrada ao ambiente Bellatrix, e por consequência, ao Eclipse deve-se ao fato do Bellatrix prover uma série de facilidades e infraestrutura que permitirá que o desenvolvimento da ferramenta Meissa foque-se nos aspectos relevantes a este trabalho, como, por exemplo, a recuperação e comparação de modelos arquiteturais.

4.3.2 Restrições da ferramenta

1. O modelo arquitetural suportado pela ferramenta deverá ser um diagrama UML de componentes.

2. Durante a geração automática de código, a ferramenta não poderá alterar porções de código não relacionadas ao objetivo da refatoração, ou seja, a ferramenta só deverá alterar os trechos de código alvos da refatoração.

Deve-se notar a importância das restrições acima. Contrariamente a segunda restrição, outras abordagens de refatoração [62] optam por reconstruir todo o código-fonte, o que levará a perda de código-fonte manualmente adicionado pelos engenheiros de software.

4.3.3 Requisitos de qualidade da ferramenta

RNF-1 Manutenibilidade - Deve ser fácil de evoluir a ferramenta, tanto para a correção de falhas ou para a adição de novas funcionalidades

RNF-2 A ferramenta deverá ser capaz de verificar se o código-fonte segue às regras do modelo de implementação de componentes utilizado..

RNF-3 Desempenho - O tempo de execução das operações de recuperação da arquitetura e geração automática de código deve ser linearmente proporcional ao número de elementos arquiteturais. O tempo de execução da operação de comparação de arquiteturas deve ser proporcional não mais que ao quadrado do número de elementos arquiteturais.

RNF-4 Portabilidade - A ferramenta deverá ser portável para todos sistemas para os quais existam uma versão do ambiente Eclipse disponível.

RNF-5 Usabilidade - A ferramenta deve ser fácil de ser utilizada, provendo guias passo a passo para a execução das atividades.

4.3.4 Casos de uso

O levantamento dos casos de uso para a ferramenta foram feitos com base em cenários de execução do método pelo arquiteto, considerando o uso de uma ferramenta de suporte. Uma vez que os casos de uso foram levantados, um diagrama inicial de casos de uso foi desenhado e continuamente refinado, até que se atingisse a versão final, a qual é apresentada na Figura 4.3.

No apêndice B, são detalhados os casos de uso representados na Figura 4.3.

4.3.5 Exemplo de cenário de uso da ferramenta

Abaixo é apresentado uma estória de usuário [3] considerada para o levantamento dos casos de uso para a ferramenta.

Lancy é a arquiteta responsável pelo novo cliente de e-mail desenvolvido pela empresa SpammerJoe. Lancy decidiu que o sistema seguiria um desenvolvimento baseado em componentes, utilizando COSMOS como o modelo de implementação de componentes. Após alguns meses de desenvolvimento, Lancy decide aplicar o método Meissa para verificar se a implementação do sistema está conforme a arquitetura que ela especificou no início do projeto.*

Lancy inicia a ferramenta e cria um novo projeto, nomeando-o "Spamless e-mail client". A ferramenta pergunta a Lancy qual é a linguagem de programação e o modelo de componentes utilizados no projeto. Lancy escolhe Java para a primeira pergunta e COSMOS para a segunda. Lancy então decide importar o modelo arquitetural especificado por ela no início do projeto e o código-fonte atual do projeto.*

Quando a ferramenta termina de importar tais artefatos, Lancy então inicia a primeira atividade do método, Recuperação. Para isso, Lancy requisita a ferramenta recuperar a arquitetura implementada do sistema. A ferramenta informa a Lancy que o código-fonte apresenta alguns problemas de conformidade com o modelo de componentes utilizado e pede que Lancy resolva tais problemas antes de prosseguir. Lancy salva a lista de erros reportada pela ferramenta e envia um e-mail a Joe, o engenheiro líder, pedindo que as

[3] Do inglês *user story*.

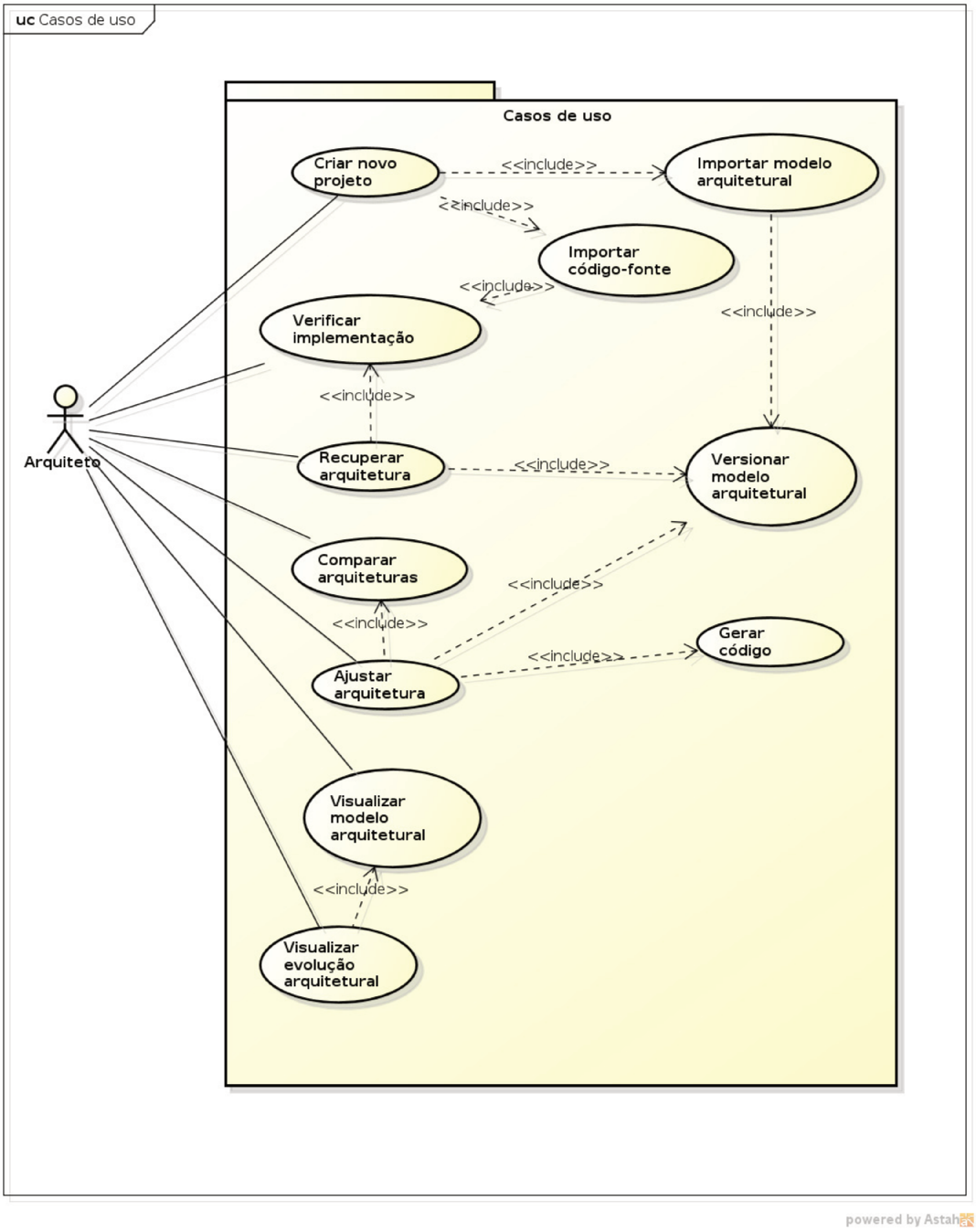


Figura 4.3: Diagrama de casos de uso para a ferramenta Meissa.

inconformidades do código-fonte em relação ao modelo de implementação de componentes sejam corrigidas.

No dia seguinte, Lancy atualiza sua copia local do código-fonte para obter as correções

feitas no dia anterior. Lancy novamente executa a ferramenta para recuperar a arquitetura implementada. Desta vez, a ferramenta não reporta nenhum erro de conformidade relacionado ao modelo de implementação de componentes e inicia o processo de recuperação da arquitetura.

Após recuperada a arquitetura, a ferramenta abre um visualizador gráfico mostrando o diagrama da arquitetura recuperada pela ferramenta. Lancy navega pelo modelo, expandindo o modelo arquitetural do componente composto responsável por implementar os requisitos relacionados a verificação de spams, o principal diferencial da aplicação em que Lancy é responsável. Intrigada com o que vê, Lancy decide executar a comparação de modelos através da ferramenta. A ferramenta pergunta a Lancy quais modelos devem ser comparados. Lancy provê o modelo recuperado e o modelo por ela especificado no início do projeto.

Ao final da comparação a ferramenta mostra a Lancy uma lista com todas as diferenças encontradas entre os dois modelos. Lancy, interessada no componente de prevenção de spam, seleciona a visão arquitetural, o que faz a ferramenta mostrar um diagrama arquitetural de diferenças, colorindo as diferenças entre os modelos. Lancy logo identifica o componente composto de prevenção de spam em azul, indicando que existem mudanças em sua arquitetura. Lancy clica duas vezes sobre tal componente, fazendo com que a ferramenta expanda sua arquitetura e colorindo as diferenças encontradas.

Fica claro para Lancy que um dos subcomponentes responsáveis pela atualização das regras de spam está faltando na arquitetura recuperada. Lancy também percebe que a interface provida por tal componente faltante está sendo implementada por um outro subcomponente na arquitetura recuperada. Lancy logo desconfia que dois dos subcomponentes foram implementados em apenas um subcomponente e prevê graves problemas que essa nova arquitetura terá em relação a como o servidor de atualizações das regras de spam está sendo implementado por outro time da empresa.

Lancy, usando a ferramenta, altera o diagrama arquitetural recuperado para que tais subcomponentes sejam implementados separadamente, adequando também suas interfaces providas. Lancy salva suas alterações e requisita a ferramenta que gere o código-fonte necessário para que suas alterações arquiteturais sejam aplicadas ao código-fonte. Quando a ferramenta informa a Lancy sobre o término da operação, Lancy cria um pacote "diff" com as mudanças do código-fonte e as envia por e-mail para o Joe, engenheiro líder do projeto, explicando o problema.

No dia seguinte, Lancy recebe uma resposta de Joe, reconhecendo o problema e informando a Lancy que a partir de então a ferramenta será executada pelo sistema de submissão de código-fonte, para que erros de conformidade em relação ao modelo de implementação de componentes e desvios arquiteturais possam ser identificados mais cedo durante o desenvolvimento do sistema.

4.4 Arquitetura da ferramenta

Uma vez levantados os requisitos, visitados os cenários de uso da aplicação e definidos os casos de uso podemos seguir em frente na aplicação do processo UML Components, como discutido na Seção 2.4.

4.4.1 Modelo de conceito do negócio

O primeiro passo é a definição do *Modelo de conceito do negócio*[4]. Esse modelo, representado em um diagrama de classes, representa os elementos principais envolvido nos processos do negócio, neste caso, a automação das atividades do método Meissa. O objetivo deste modelo é mapear os termos relevantes para o negócio, servindo como base para as próximas etapas de especificação do sistema. Ele não necessariamente representa classes que serão codificadas durante a implementação do sistema, contudo, é provável que alguns desses termos acabem por se tornar classes do sistema [13].

Através de análise textual [51] dos cenários e casos de uso levantados para a ferramenta, construiu-se o modelo de conceito do negócio para a ferramenta Meissa apresentado na Figura 4.4.

4.4.2 Interfaces de sistema

A etapa seguinte é a definição das interfaces de sistema. O processo UML Components, como mencionado na Seção 2.4, define duas camadas principais: a camada de sistema e a camada de negócio. Resumidamente, a camada de sistema é a camada que interage diretamente com o usuário, enquanto a camada de negócio implementa fundamentalmente a lógica de negócio da aplicação. Desta forma, a camada de sistema é responsável por alavancar as funcionalidades providas pela camada de negócio a fim de realizar os fluxos de utilização do sistema definidos nos casos de uso.

Portanto, a definição das interfaces da camada de sistema inicia-se por analisar os casos de uso e como se dá a interação dos atores com o sistema, identificando as responsabilidades do sistema. A partir desta análise, abstrai-se as operações que o sistema provê aos atores durante a execução dos casos de uso. Tais operações são agrupadas em interfaces, considerando sua coesão e afinidade.

As interfaces de sistema definidas para a ferramenta Meissa, de acordo com o discutido previamente, são representadas na Figura 4.5.

[4]Do inglês, *Business concept model*.

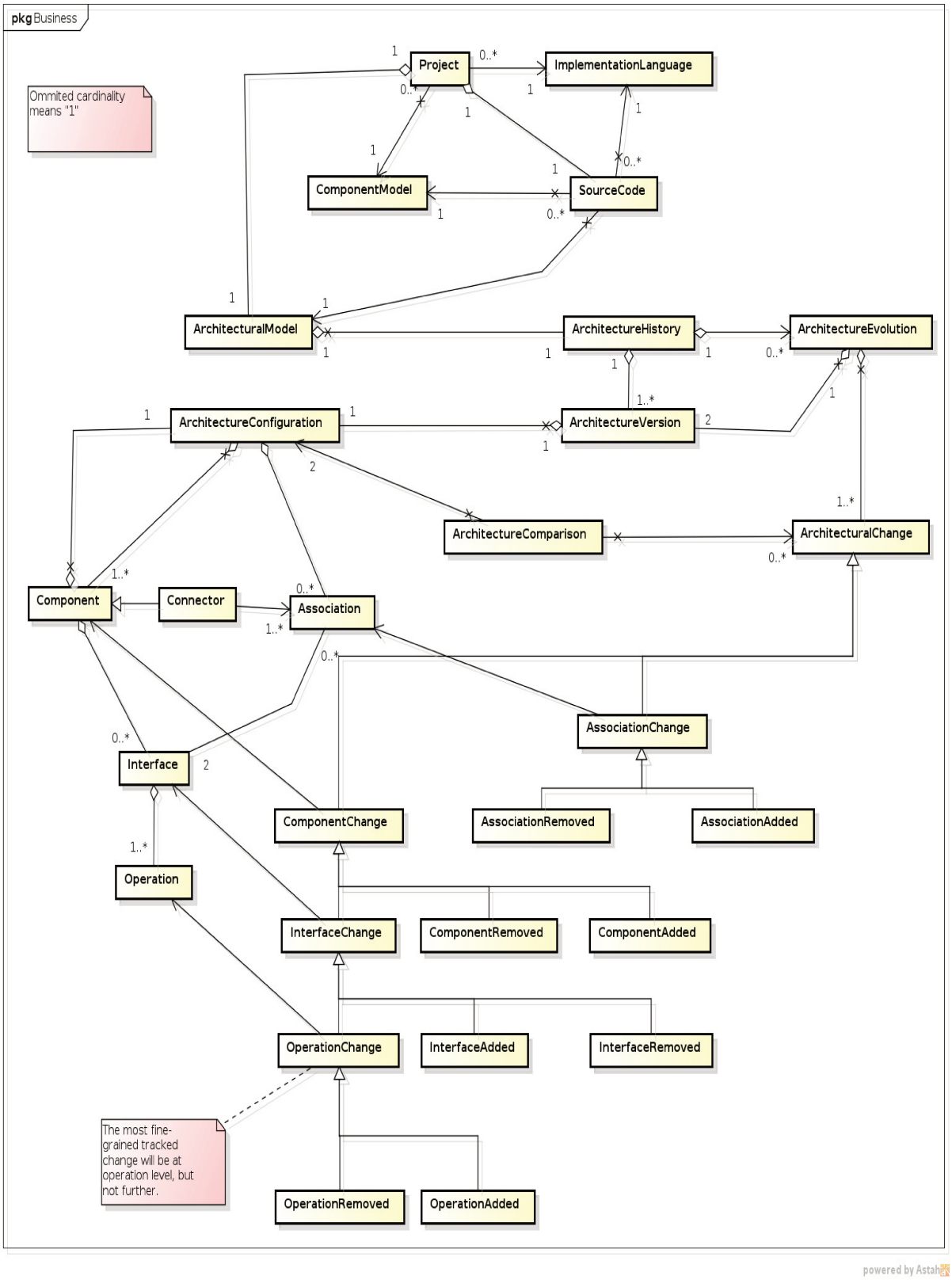


Figura 4.4: Modelo de conceito do negócio para a ferramenta Meissa.

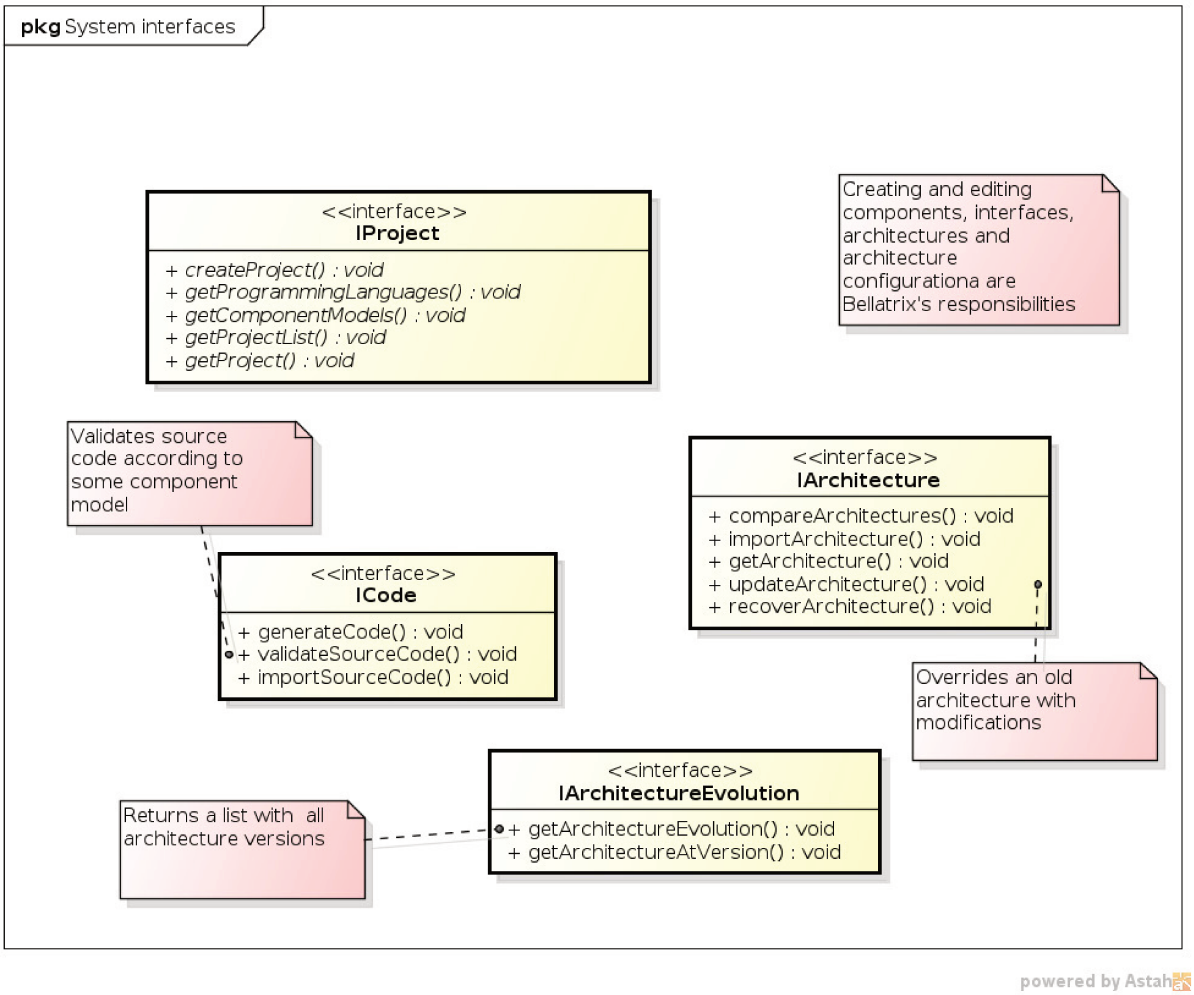


Figura 4.5: Interfaces de sistema para a ferramenta Meissa.

4.4.3 Modelo de tipos principais

Neste ponto, a interface entre o sistema e os atores está definida. É necessário, agora, detalhar como a camada de sistema e a camada de negócios interagem. Posto de outra maneira, é necessário definir as responsabilidades da camada de negócios e como elas são expostas através das interfaces de negócio. Revisitando a Seção 2.4, a definição das interfaces da camada de negócio é feita através do refinamento do modelo de conceito de negócio.

Tomando-se o modelo de conceito de negócio, obtemos a partir dele o modelo de tipos de negócio. Esse modelo representa a visão do sistema quanto aos elementos de negócio, detalhando as informações que o sistema precisa conhecer sobre cada um dos elementos de negócio. Ele contém um subconjunto dos elementos encontrados no modelo de conceito de

negócio, obtido removendo e adicionado-se elementos até que o escopo do modelo esteja correto.

Após definido o modelo de tipos de negócio, um novo modelo é definido, o modelo de de tipos principais. Tal modelo determina quais elementos do modelo de tipos de negócio são principais e quais não são. O objetivo é identificar as dependências de informação, clarificando que elementos dependem de outros e quais elementos existem independentemente de outros.

O modelo de tipos de negócio para a ferramenta Meissa é suficientemente similar ao modelo de negócio, devido à natureza do domínio envolvido, e por brevidade foi omitido. O modelo de tipos principais, contudo, foi representado na Figura 4.6.

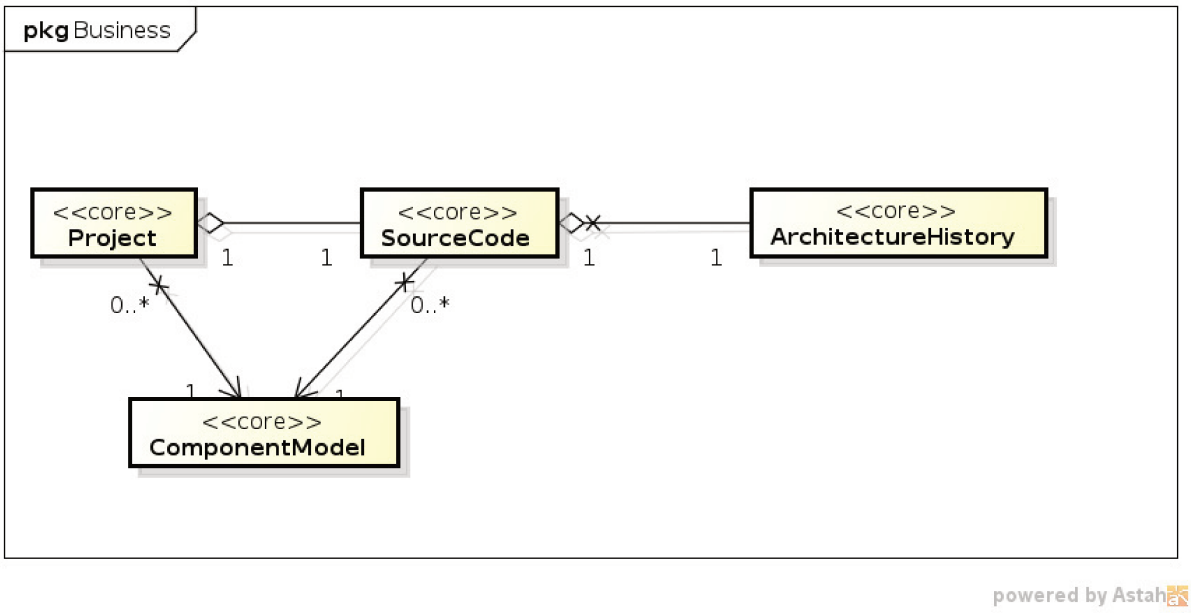


Figura 4.6: Modelo de tipos principais para a ferramenta Meissa.

4.4.4 Interfaces de negócio

Uma vez que os tipos principais de negócio foram definidos é possível iniciar a definição das interfaces da camada de negócio. Como discutido na Seção 2.4, a regra geral é a definição de uma interface por tipo principal de negócio, a não ser que existam razões para que as operação sejam divididas em mais de uma interface. Esta foi a abordagem seguida para a definição das interfaces de negócio para a ferramenta Meissa.

Cada interface de negócio é responsável por expor as operações relacionadas ao tipo principal de negócio associado e aos tipos a ele relacionado. O conjunto das interfaces

definidas para cada tipo principal de negócio formam as interfaces de negócio, isto é, as interfaces da camada de negócio.

Uma vez que as interfaces da camada de negócio sejam definidas, o arquiteto pode utilizar-se de cenários ou dos fluxos descritos nos casos de uso para validar a cobertura das interfaces em relação as operações que o sistema deve executar. Isso é feito através da execução de cenários ou fluxos de casos de uso levando em consideração quais operações, tanto na camada de sistema, quanto na camada de negócio devem ser executadas para que a operação possa ser completamente servida.

Essa abordagem pode ser tomada em partes. Inicialmente, o arquiteto pode revisitar os fluxos principais e excepcionais dos casos de uso e verificar se as interfaces por ele definidas na camada de sistema são de fato o ponto de entrada de todas as possíveis iterações entre os atores e o sistema. Em seguida, o arquiteto pode validar as interfaces da camada de negócio. Considerando cada operação definida nas interfaces de sistema, ele pode percorrer quais seriam as operações nas interfaces de negócio que precisam ser executadas para servir a operação iniciada na camada de sistema.

4.4.5 Especificação dos componentes

Como é discutido em [13], existem diversas abordagens para a determinação dos componentes nas camadas de sistema e de negócio, como, por exemplo, considerando componentes já existentes para reúso. A seguir, são apresentadas algumas ideias de como especificar os componentes e qual abordagem foi tomada para a ferramenta Meissa.

A definição de componentes na camada de sistema é, em geral, simples e direta: para cada interface de sistema especificada é criado um componente que a proveja. Isso ocorre devido à natureza da especificação das interfaces de sistema. Cada interface possui operações coesas, mas duas interfaces tenderão a possuírem operações suficientemente distintas para que não faça sentido que um mesmo componente implemente ambas interfaces. Certamente, é necessário o julgamento do arquiteto, uma vez que devido à natureza dos casos de uso e cenários utilizados para o levantamento das interfaces de sistema é possível que existam situações em que duas interfaces sejam, em algum nível, relacionadas e portanto, é sensato que sejam implementadas por um mesmo componente.

Para a ferramenta Meissa, a abordagem anterior não foi totalmente aplicável. Percebe-se que, na verdade, existe certo relacionamento entre as operações providas entre algumas interfaces, o suficiente para que seja sensato sua implementação por um mesmo componente.

No caso das interfaces *ICode* e *IProject* (Figura 4.5), decidiu-se que um mesmo componente, *ProjectSystem*, implementasse ambas. Isso se deve ao fato das operações providas por *ICode* estarem relacionadas também ao projeto sob análise da ferramenta, pois não faz

sentido tratar um código-fonte que não faça parte de um projeto, na qual uma linguagem de programação foi definida e um modelo de componentes foi escolhido.

Para as interfaces *IArchitecture* e *IArchitectureEvolution* (Figura 4.5) um mesmo componente *ArchitectureSystem* provê as duas interfaces. É fácil perceber a relação de ambas, uma vez que as duas operam sobre o mesmo conceito, apenas em escopos ligeiramente distintos; desta forma, é sensato que sejam implementadas por um mesmo componente.

Na Figura 4.7, as interfaces e componentes de sistema são apresentados.

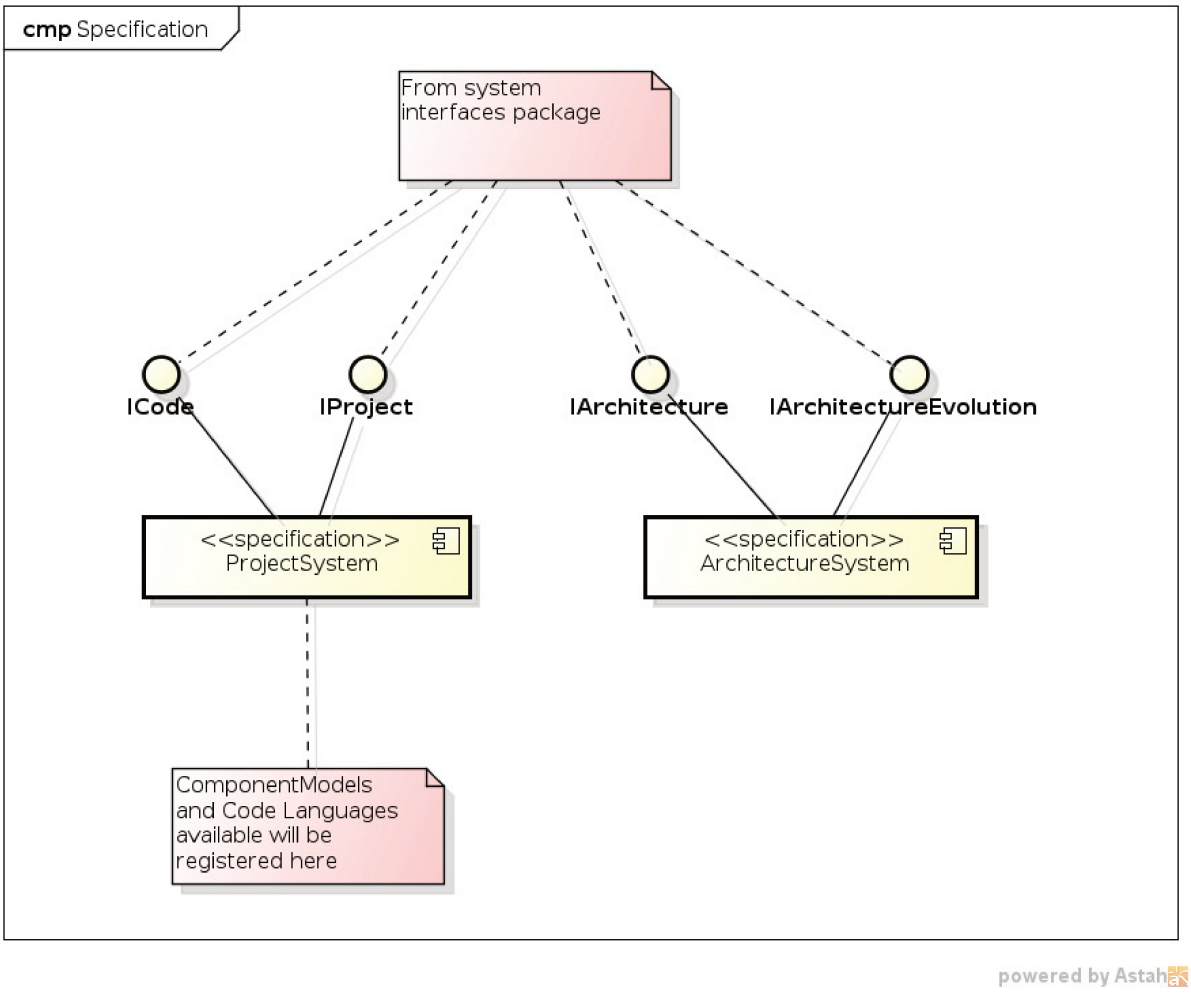


Figura 4.7: Interfaces de sistema e seus componentes para a ferramenta Meissa.

Na camada de negócios, a especificação de componentes é guiada pelo processo de levantamento de tipos principais de negócio. A partir destes, são criados componentes cuja responsabilidade é prover as operações que manipulam tais tipos (e por consequência, prover as interfaces de negócios relacionadas a tais tipos principais). [13] descreve outras

possíveis abordagens para a especificação dos componentes na camada de negócios, por exemplo, selecionar componentes já existentes, em prol do reúso, implicando em possível adaptação entre as interfaces de negócio especificadas anteriormente e as interfaces providas pelos componentes reutilizados.

A abordagem adotada na especificação de componentes da camada de negócios para a ferramenta Meissa foi hibrida. Para cada tipo principal de negócio foi definido um componente que proveja a interface a ele relacionado. A partir daí, quando apropriado, cada componente especificado foi analisado em relação a sua própria arquitetura, considerando, portanto os subcomponentes que os compões. Nesta etapa, os subcomponentes foram selecionados tendo em mente a reutilização de componentes já existentes, como, por exemplo, analisadores sintáticos de código-fonte. A seguir, discutiremos em detalhes cada um dos componentes de negócio cujas arquiteturas foram detalhadas.

Na Figura 4.8, os componentes e interfaces de negócio são apresentados.

4.5 Levantamento de componentes para reúso

Como parte da especificação, é necessário levantar componentes existentes que possam ser reutilizados na arquitetura do sistema de forma a reduzir os custos de implementação. Essa atividade é fundamental para a especificação do sistema, uma vez que um potencial componente a ser reutilizado precisa ser considerado antes da definição da arquitetura, ou corre-se o risco de ser necessário adequar a arquitetura a fim de incorporar o componente, uma vez que é, em geral, necessário algum tipo de adaptação das interfaces implementadas pelo componente reutilizado e as demais interfaces do sistema.

Nesta seção, são descritos componentes existentes que podem ser incorporados ao projeto da ferramenta.

4.5.1 Ambiente Bellatrix

Bellatrix, como mencionado anteriormente, é um ambiente de suporte para o desenvolvimento baseado em componentes, apoiado pela plataforma Eclipse. Bellatrix foi desenvolvido seguindo a abordagem *UML Components* [13]. Como discutido na Seção 2.4, esse processo define duas camadas principais, a de sistema e a de negócio, sendo esta última a que implementa a lógica de negócios da aplicação. Desta forma, os componentes da camada de negócios do ambiente Bellatrix devem ser considerados para reúso e integração com a ferramenta Meissa.

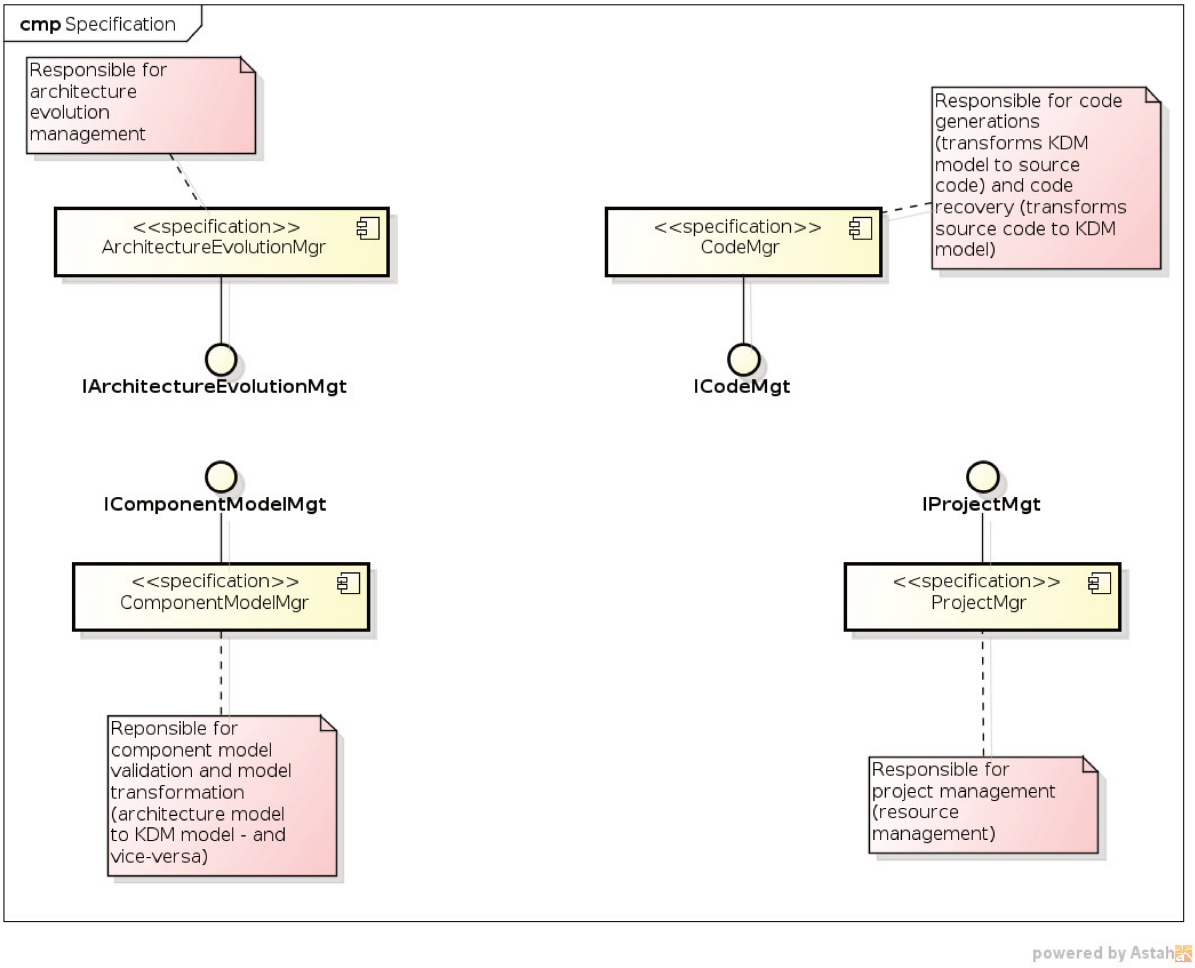


Figura 4.8: Interfaces de negócio e seus componentes para a ferramenta Meissa.

4.5.2 MoDisco

MoDisco é um projeto que estende a plataforma Eclipse provendo um *framework* extensível para desenvolvimento de ferramentas dirigidas por modelo a fim de suportar a modernização de softwares existentes [41, 12]. Este projeto permite a transformação de código-fonte em um modelo KDM, *knowledge discovery meta-model*[5] [28]. KDM é um modelo intermediário de representação de um sistema de software existente e de seu ambiente de operação, definindo meta dados comumente usados para integração semântica por ferramentas de gerenciamento do ciclo de vida de aplicações [29].

O modelo KDM pode ser visto, de maneira simplificada, como um superconjunto de um modelo de classes UML. Ele permite a representação de artefatos de linguagens de

[5]Em tradução livre, metamodelo de descobrimento de conhecimento.

implementação orientadas a objeto como classes, métodos, pacotes, interfaces, atributos de visibilidade, entre outros elementos, e ao mesmo tempo é genérico o suficiente para representar artefatos de linguagens estruturadas, como funções, estruturas e cabeçalhos. Além disso, um modelo KDM é capaz de representar estruturas do sistema de arquivos relacionadas ao código-fonte implementado, por exemplo, arquivos e diretórios e quais classes, por exemplo, são implementadas em quais arquivos. Essa flexibilidade e extensiva capacidade de representação, não apenas do código-fonte em si, mas do ambiente de operação que o engloba, tornam o modelo KDM bastante interessante para a ferramenta Meissa, uma vez que são exatamente esses tipos de informações as necessários para o trabalho de automação da recuperação arquitetural.

A fim de exemplificar a utilidade do projeto MoDisco, pode-se tomar um projeto escrito em Java. O projeto MoDisco provê uma implementação de um conversor Java-KDM, ou seja, dado o código-fonte escrito em Java o conversor irá gerar um modelo KDM representando os arquivos que contém o código-fonte, as classes implementadas em cada arquivo, as operações de cada classe e assim por diante. Esse tipo de informação é, essencialmente, o que a ferramenta Meissa precisa para iniciar a recuperação da arquitetura do sistema. Desta forma, o projeto MoDisco toma grande importância na automação da recuperação da arquitetura.

Ainda, é interessante ressaltar que o projeto MoDisco utiliza-se de um linguagem de transformação automática de modelos, o ATL[6] [11, 5, 19]. Com isso, é possível estender de maneira simplificada o projeto MoDisco a fim de adicionar novas transformações, que permitam a recuperação de modelos KDM para outras linguagens de programação.

4.5.3 Acceleo

O projeto Acceleo [2] é uma implementação do padrão MOFM2T [30], definido pela OMG[7]. MOFM2T é a especificação de uma linguagem de transformação de modelos [35] cujo objetivo é a transformação de um modelo em texto (M2T[8]).

Através da integração do projeto Acceleo e o Eclipse é possível escrever scripts que convertem modelos EMF em texto (ou código-fonte) [34, 33]. Assim, este projeto é interessante ao permitir a geração de código-fonte a partir de um modelo EMF. No caso da ferramenta Meissa, serão utilizados modelos KDM, que seguem o metamodelo EMF, quando recuperados pelo projeto MoDisco. Desta forma, o projeto Acceleo fecha o ciclo ao permitir que os modelos KDM possam ser novamente convertidos em código-fonte.

[6] ATL *transformation language*, do inglês, linguagem de transformação ATL.
[7] Do inglês, *Object Management Group*.
[8] Do inglês, *model to text*.

4.5.4 Especificação da arquitetura

Uma vez que os componentes e interfaces de ambas as camadas de negócio e sistema foram definidas é possível passar para a especificação da arquitetura do sistema. Em regra, os componentes da camada de sistema são responsáveis por implementar os fluxos definidos nos casos de uso, utilizando-se das funcionalidades providas pelos componentes da camada de negócios.

A Figura 4.9 representa a arquitetura da ferramenta Meissa, com os componentes das camadas de negócio e de sistema.

A ferramenta Meissa baseia-se no ambiente Bellatrix, que provê a infraestrutura para a criação de diagramas arquiteturais. Para tanto, foi adicionado um novo componente na camada de sistema: *ArchitectureBuilder*. Esse componente é responsável por interfacear com o ambiente Bellatrix e prover à camada de negócios uma abstração para a criação e edição de diagramas de componentes, interfaces e arquiteturas.

4.6 Implementação da ferramenta

Até este momento, todos os componentes foram apresentados em forma de sua especificação, ou seja, sem considerações de detalhes implementação. Ao revisitar o diagrama arquitetural apresentado na Figura 4.9, existem dois componentes na camada de negócio para os quais há grande interesse em se realizar uma análise aprofundada de sua implementação. Tais componentes são: *CodeMgr* e *ComponentModelMgr*.

4.6.1 CodeMgr

O componente *CodeMgr* é o responsável pela transformação de código-fonte em um modelo independente de implementação e vice-versa. Como mencionado anteriormente, para a ferramenta Meissa tal modelo escolhido foi o KDM.

Uma implementação do componente *CodeMgr* deve claramente restringir-se a apenas uma linguagem de programação. Desta forma, cada implementação do componente pode manter-se coesa e modular. Utilizando-se da infraestrutura de *plugins* do ambiente Eclipse, é possível encapsular cada implementação do componente em um *plugin* que pode ser distribuído juntamente com o restante da ferramenta, ou separadamente utilizando-se o sistema de distribuição de *plugins* do Eclipse.

É possível aproveitar-se do sistema de notificação do ambiente de forma a tornar o suporte a uma linguagem de programação dinâmico, dependendo da existência da implementação do componente (*plugin*) no ambiente. Essa tarefa pode ser realizada através do suporte do sistema de *plugins* do Eclipse. Quando uma implementação do componente *CodeMgr* for carregada no ambiente, ela registrará uma extensão (interface provida) para

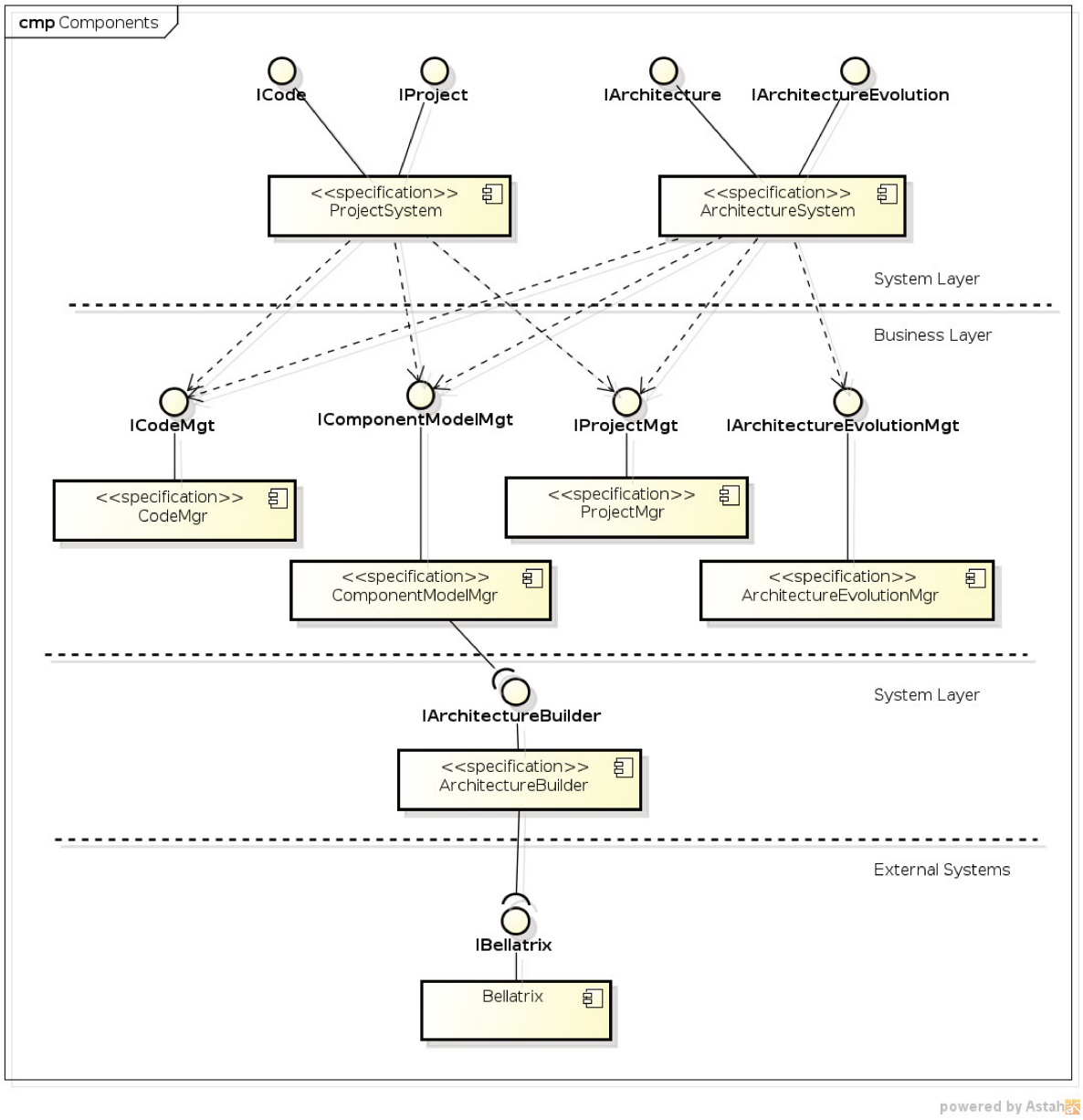


Figura 4.9: Especificação da arquitetura para a ferramenta Meissa.

um ponto de extensão (interface requirida) definido pelo componente da camada de sistema *ProjectSystem*. Este componente, como mencionado anteriormente, é o responsável pela criação do projeto e da seleção tanto da linguagem de programação quanto do modelo de implementação de componentes utilizados no novo projeto.

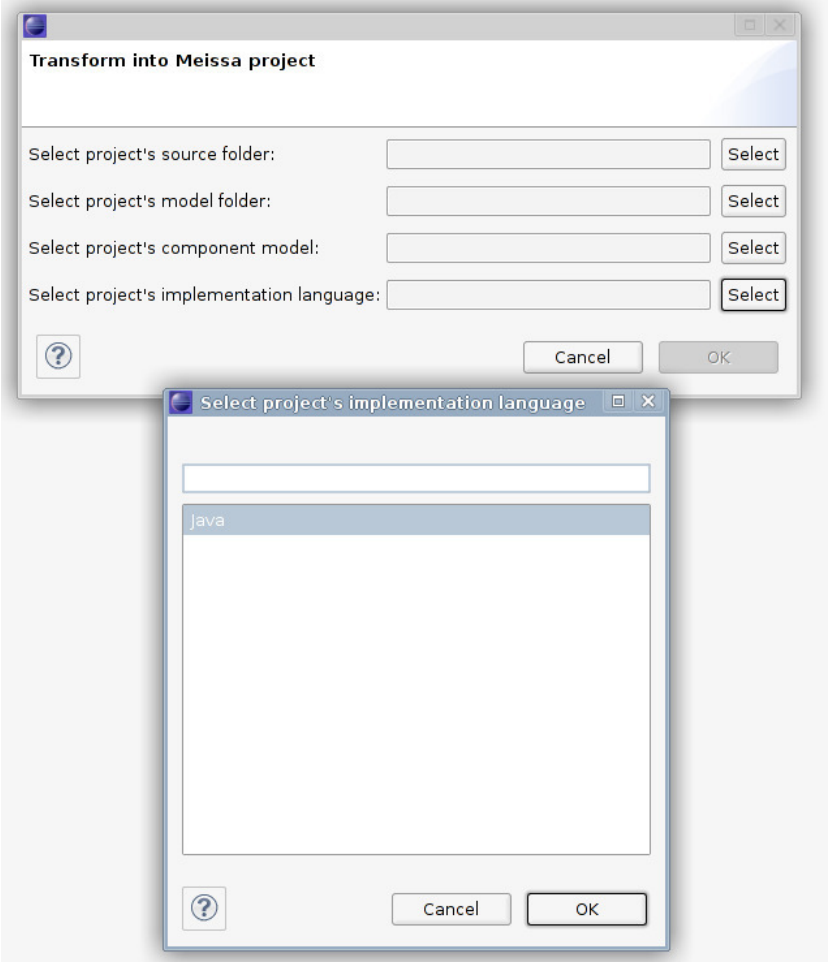


Figura 4.10: Interface gráfica da ferramenta Meissa que permite ao usuário selecionar qual é a linguagem de programação utilizada no projeto.

A Figura 4.10 mostra a interface gráfica da ferramenta responsável pela seleção da linguagem de programação utilizada em um novo projeto. O código que é executado para mostrar ao usuário o conjunto de linguagens de programação disponíveis naquele momento irá acionar o componente *ProjectSystem* através de sua interface provida *IProject* para obter o conjunto de linguagens suportadas, ou seja, o conjunto de implementações do componente *CodeMgr* que está disponível no ambiente.

Ainda que a implementação componente *CodeMgr* dedique-se a apenas uma linguagem de programação, ela é responsável por duas grandes operação: a conversão do código-fonte

em um modelo KDM e a conversão de um modelo KDM em código-fonte. Devido a essa dicotomia nas responsabilidades do componente, sua arquitetura foi especificada a fim de facilitar sua implementação.

A Figura 4.11 representa a arquitetura do componente *CodeMgr* separando sua especificação na parte superior e sua implementação na parte inferior. Devido à dupla responsabilidade requerida ao componente, a especificação da arquitetura do componente subdivide-o em dois subcomponentes: *CodeRecoveryMgr*, responsável pela transformação do código-fonte em um modelo KDM e *CodeGenerationMgr*, responsável pela transformação do modelo KDM em código-fonte.

Do ponto de vista da implementação, um novo subcomponente é necessário para agir como adaptador entre a interface externa provida pelo componente *CodeMgr* e as interfaces dos subcomponentes de recuperação e geração de código.

Para a ferramenta Meissa, a implementação feita do componente *CodeMgr* toma a linguagem Java como alvo. De forma a realizar a transformação do código-fonte em modelo KDM foi implementado o subcomponente *ModiscoMgr* (implementação da especificação do subcomponente *CodeRecoveryMgr*). Tal componente utiliza-se do projeto MoDisco, discutido na Seção 4.5.2, para realizar a transformação do código-fonte em um modelo KDM. Com o objetivo de geração de código, o subcomponente *AcceleoMgr* (implementação da especificação do subcomponente *CodeGenerationMgr*) foi implementado utilizando-se o projeto Acceleo, apresentado na Seção 4.5.3, que auxilia na geração automática de código-fonte para a linguagem Java.

4.6.2 ComponentModelMgr

O componente *ComponentModelMgr* é o responsável por várias operações relacionadas a um modelo de implementação de componentes: (i) validar a conformidade do código-fonte implementado com as regras impostas pelo modelo de componentes (através da validação do modelo KDM obtido a partir do código-fonte pelo componente *CodeMgr*); (ii) transformar o modelo KDM que representa o código-fonte em um modelo arquitetural (recuperando a arquitetura implementada); e (iii) converter um modelo arquitetural em um modelo KDM representando o código-fonte necessário para implementar a arquitetura proposta.

De maneira similar ao discutido para o componente *CodeMgr*, cada implementação deste componente está restrita a um modelo de implementação de componentes. Da mesma forma, cada implementação registrará sua extensão ao ponto de extensão de modelos de componentes provido pelo componente *ProjectSystem*, permitindo desta forma que o suporte a um modelo de implementação de componentes seja introduzido no ambiente com a adição de um novo *plugin*. Na Figura 4.12 é mostrada a interface gráfica da

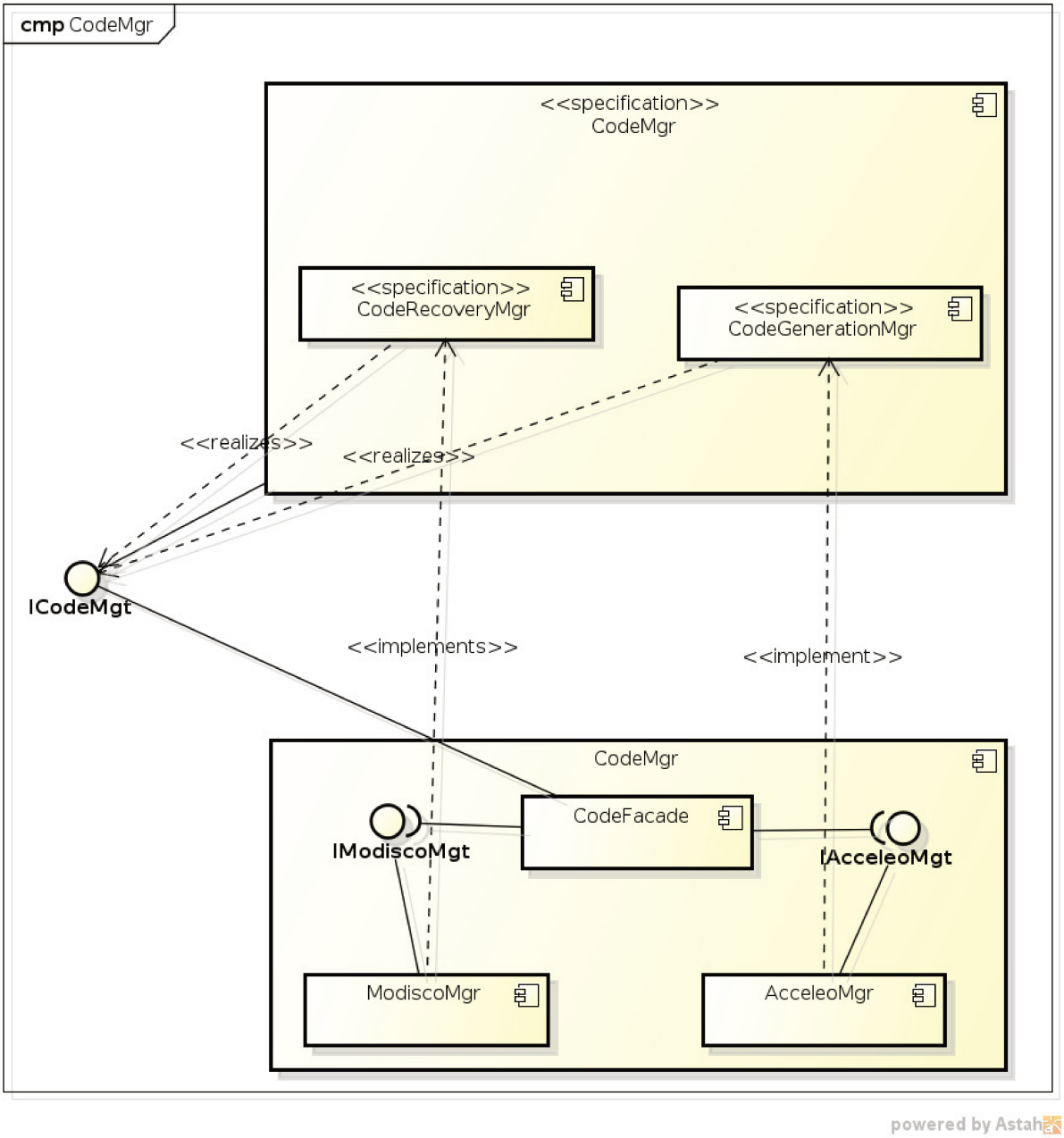


Figura 4.11: Detalhamento da arquitetura do componente *CodeMgr*.

ferramenta que permite a seleção de um modelo de implementação de componentes a ser utilizado no projeto.

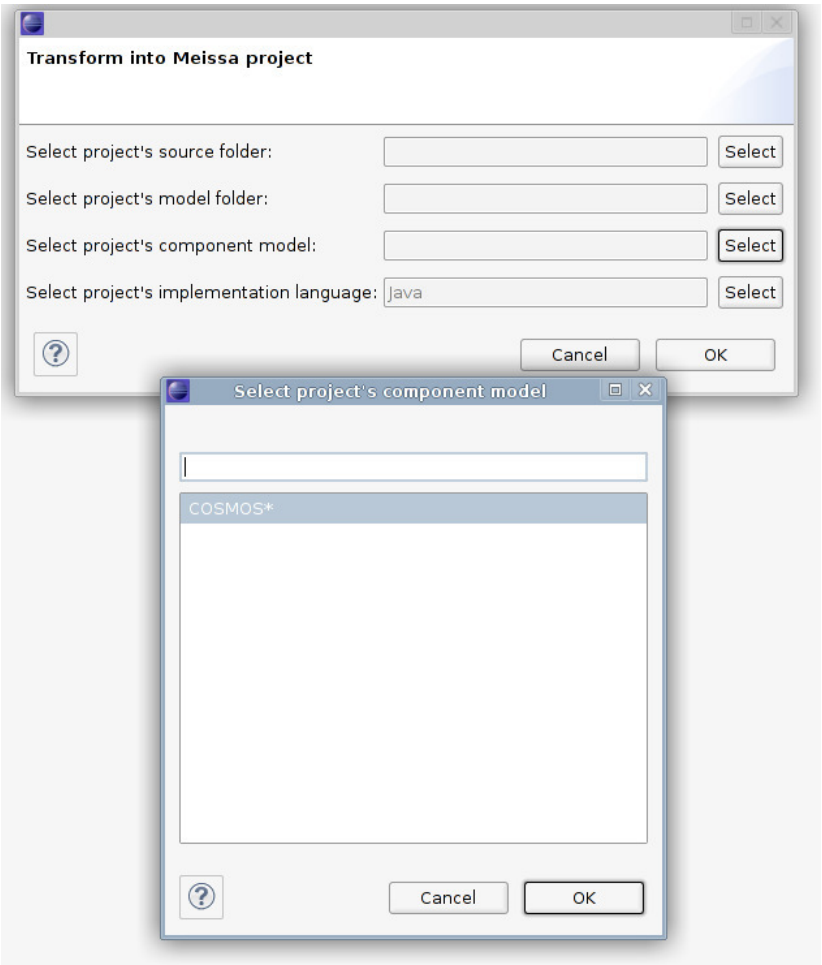


Figura 4.12: Interface gráfica da ferramenta Meissa que permite ao usuário selecionar qual é o modelo de implementação de componentes utilizado no projeto.

Quanto à arquitetura do componente *ComponentModelMgr* as mesmas ideias apresentadas para o componente *CodeMgr* aplicam-se. Na Figura 4.13, pode-se visualizar a arquitetura especificada para o componente e dois subcomponentes: *ModelValidationMgr*, responsável por validar um modelo KDM representando o código-fonte quanto a conformidade das regras do modelo de implementação de componentes (mais detalhes no apêndice A) e *ModelTransformationMgr*, responsável pelas transformação de um modelo KDM em um modelo arquitetural e vice-versa.

Implementado juntamente com a ferramenta Meissa, foi o componente *CosmosStarMgr*. Como discutido na Seção 2.5, COSMOS* foi o modelo de implementação de componentes adotado para o desenvolvimento da ferramenta, bem como nos estudos de

caso considerados neste trabalho. A arquitetura do componente *CosmosStarMgr* segue a especificação, com um subcomponente para a validação de modelos e outros para sua transformação. Também foi adicionado um subcomponente para atuar como adaptador entre as interfaces dos subcomponentes e as interfaces externas.

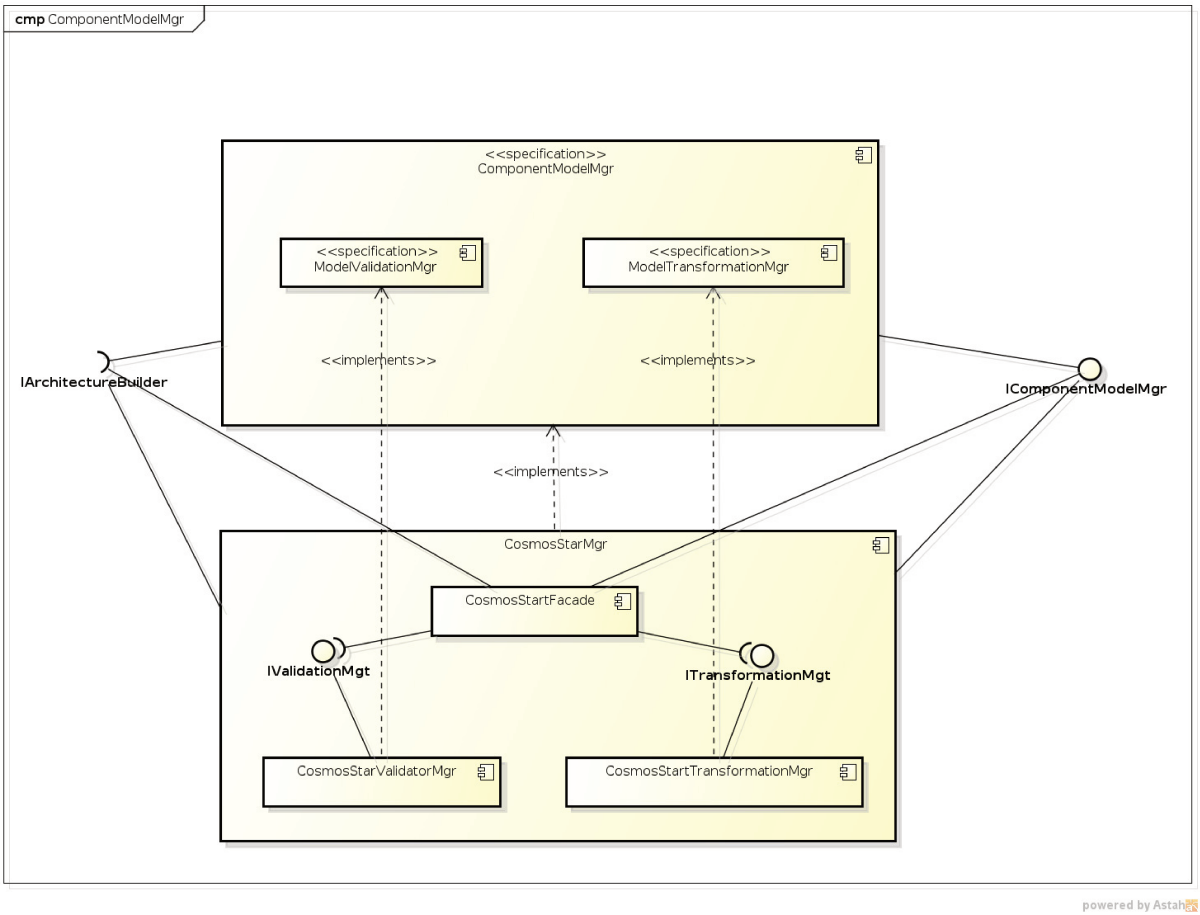


Figura 4.13: Detalhamento da arquitetura do componente *ComponentModelMgr*.

4.6.3 Outras considerações sobre a implementação

Verificador COSMOS*

Foi implementado, como parte do componente *CosmosStarMgr*, uma etapa de validação da adequação do código-fonte quanto a corretude de implementação do modelo de componentes COSMOS*. No apêndice A foram levantados os requisitos que o código sob análise deve satisfazer para ser considerado uma implementação correta do modelo de implementação de componentes COSMOS*.

Essa etapa de validação é necessária durante a atividade de recuperação da arquitetura, já que a não conformidade entre o código-fonte e o modelo de implementação de componentes leva a inconsistências na arquitetura recuperada. Com a garantia que o código-fonte a ser recuperado está conforme a especificação do modelo de implementação de componentes, a implementação do algoritmo de recuperação arquitetural pode seguramente assumir certas condições, o que simplifica a sua implementação.

Devido à natureza modular dos componentes implementados, é possível utilizar o componente *CosmosStarMgr* isoladamente para a validação da conformidade entre o código-fonte e o modelo de implementação de componente. Assim é possível construir uma pequena ferramenta, que se utilize de tal componente, para verificar a conformidade do código-fonte a cada compilação do sistema. Isso permitiria aos desenvolvedores um maior grau de confiança no código produzido, uma vez que não importando que mudanças eles realizem durante o processo de desenvolvimento ou manutenção, estará garantido, por uma etapa de pós-compilação, que o código-fonte sempre segue as regras especificas pelo modelo de implementação de componentes adotado.

4.7 Resumo

Neste capítulo foi apresentada a ferramenta Meissa, que compõe a solução proposta neste trabalho juntamente com o método Meissa. Seu objetivo é a automação das atividades do método, capacitando o arquiteto de software a aplicar o método de forma prática e ágil. A ferramenta integra-se com o ambiente Bellatrix, que provê os editores gráficos para a criação e edição de modelos arquiteturais. Meissa adiciona funcionalidades ao ambiente, ao permitir ao arquiteto recuperar o modelo arquitetural baseado no código-fonte do um sistema, ao destacar o resultado da comparação entre modelos arquiteturais em um editor de diagramas arquitetural ou ao gerar código-fonte para persistir alterações arquiteturais no código-fonte do projeto.

Através do processo UML Components, descrito na Seção 2.4, a ferramenta Meissa foi especificada e implementada, utilizando outros projetos da comunidade Eclipse [23] como componentes da arquitetura, como, por exemplo, MoDisco [41] e Acceleo [2]. A arquitetura da ferramenta segue o modelo em camadas proposto pelo processo UML Components, no qual existem a camada de sistema, responsável pela iteração do usuário com o sistema, e a camada de negócio, a qual é responsável pela implementação da lógica de negócios do sistema.

Na camada de negócios, encontram-se os dois principais componentes da ferramenta: o *ComponentModelMgr* e o *CodeMgr*. O componente *CodeMgr* é o responsável pela transformação de código-fonte em um modelo independente de implementação e vice-versa. Já o componente *ComponentModelMgr* é o responsável pelas operações relacionadas a um

modelo de implementação de componentes, como a validação a conformidade do código-fonte implementado e a transformação entre modelos de implementação em modelos arquiteturais.

Com este capítulo, é encerrada a descrição principal da solução Meissa. No próximo capítulo, a solução é posta em prática em dois estudos de caso, analisados em detalhes, a fim de ponderar a aplicabilidade da solução proposta.

Capítulo 5

Estudos de caso

Neste capítulo são apresentados dois estudos de casos que procuram avaliar os benefícios e desvantagens da aplicação da solução Meissa, ou seja, da aplicação do método e ferramenta detalhados neste trabalho. O objetivo dos estudos de caso aqui apresentados é analisar qualitativamente e quantitativamente os resultados obtidos após a aplicação da solução em diferentes situações e sistemas de software.

5.1 Escolha dos estudos de caso

Um dos principais pontos é ponderar a eficiência e eficácia da ferramenta construída para auxiliar a aplicação do método. Ela será o elemento que habilitará ao método ser pragmaticamente utilizado em grandes e complexos projetos computacionais. Será sua capacidade de tratar, em tempo hábil, grandes repositórios de código e de manipular modelos arquiteturais com várias dezenas de componentes que ditará o sucesso ou fracasso da adoção do método proposto.

Os estudos de caso não se focam somente na ferramenta, serão ponderadas as contribuições do método quanto a sua capacidade de combater o envelhecimento de software associado a fatores como erosão e desvio arquiteturais. Será verificado qual é a real habilidade da solução de guiar o arquiteto de software em uma sequência clara de passos que resultem em uma redução da inconsistência entre a arquitetura implementada e a especificada.

Foram selecionados dois grandes sistemas computacionais para os estudos de caso, sendo o primeiro deles o *MobileMedia* [61] e o segundo o próprio projeto da ferramenta Meissa. A seguir cada um dos estudos de casos são apresentados com a seguinte configuração de tópicos: (i) descrição do estudo de caso; (ii) planejamento do estudo de caso; (iii) execução do estudo de caso; e (iv) discussão dos resultado obtidos.

5.2 Estudo de caso 1: MobileMedia

5.2.1 Descrição do estudo de caso 1

MobileMedia é uma linha de produtos originalmente criada pela *University of British Columbia* [61].

Uma linha de produtos de software é um conjunto de produtos de software com grande similaridade entre si, que atendem às necessidades específicas de um segmento de mercado ou missão [15]. No contexto do desenvolvimento baseado em componentes, os produtos de uma linha de produtos de software tendem a compartilhar um grande conjunto de componentes centrais e cada produto é especializado com novos componentes, provendo novas funcionalidades àquele específico produto da linha.

Um claro exemplo de linhas de produtos de software são sistemas de software para telefones móveis. Um certo modelo de telefone de menor custo, poderá restringir-se a receber e realizar chamadas. Um modelo de custo médio, além das chamadas, pode enviar e receber mensagens. Já o modelo topo de linha, poderá fazer todas as tarefas anteriores, além da capacidade de fotografar e gravar vídeos.

No exemplo anterior, temos uma linha de produtos de software extremamente simples, no qual cada modelo da linha possui apenas uma funcionalidade a mais em relação ao modelo anterior. Contudo, este não é o caso do MobileMedia. Os produtos derivados da linha de produtos MobileMedia manipulam fotos, músicas e vídeos em dispositivos com recursos limitados, como celulares, através da utilização de várias tecnologias baseadas na plataforma Java ME, por exemplo, SMS, WMA e MMAPI.

MobileMedia foi submetida a 7 evoluções, gerando um total de 8 diferentes versões. Considerando a última versão e todos as possíveis combinações de funcionalidades que podem ser constituídas a partir dela, é possível derivar 168 diferentes produtos.

A decisão de utilizar o MobileMedia como estudo de caso deve-se ao fato de ele ser um sistema real, com diversas evoluções e consequentes mudanças arquiteturais. Ainda o projeto já foi base de diversos outros estudos relacionados a sistemas baseados em componentes [21, 22, 16, 57, 17].

5.2.2 Planejamento do estudo de caso 1

MobileMedia, com suas 7 evoluções, foi originalmente implementado no trabalho de Figueiredo et al. [21]. Entretanto, em Tizzei [57], o código-fonte original foi refatorado para seguir o modelo de implementação COSMOS*. O código-fonte, bem como diagramas da arquitetura, estão disponível em [54] para cada evolução.

Com o objetivo de validar a aplicação do método e da ferramenta, o estudo de caso foi dividido em duas partes: (i) validação da recuperação da arquitetura e (ii) verificação

da geração de código.

Para a validação da recuperação da arquitetura, selecionou-se a terceira evolução do MobileMedia, isto é, a evolução que levou o projeto de sua terceira versão para a quarta. Tal evolução foi especificamente escolhida por ser a que apresenta a mais complexa evolução arquitetural entre duas versões sequenciais. Por trazer grande diversidade de cenários de evolução da arquitetura, essa escolha propicia um rico ambiente para a validação da implementação da ferramenta proposta.

Associada a cada versão, existe um diagrama arquitetural detalhando os componentes e suas relações. Para os fins deste estudo de caso, tais diagramas serão considerados como a especificação arquitetural de cada uma das versões. Desta forma, a evolução arquitetural que ocorreu entre a terceira e a quarta versões pode ser deduzida ao se comparar os diagramas arquiteturais de cada uma das versões.

A primeira validação desejada é verificar a corretude da comparação de dois diagramas arquiteturais. Isto é, verificar se a ferramenta é capaz de encontrar as diferenças arquiteturais dado dois diagramas distintos. Para isso, os diagramas arquiteturais da terceira e quarta versões serão desenhados dentro do ambiente Bellatrix e serão submetidos a ferramente Meissa para comparação. De mesma maneira, os diagramas serão manualmente comparados a fim de se obter um gabarito para validar o resultado obtido pela ferramenta.

A seguir, para verificar a capacidade de recuperação arquitetural da ferramenta, cada uma das verões será submetida a recuperação da arquitetura pela ferramenta. A fim de identificar possíveis falhas no resultado obtido pela ferramenta, também será realizada a recuperação manual das arquiteturas de cada uma das versões. Assim o modelo arquitetural manualmente recuperado e o obtido pela ferramenta serão comparados. Diferenças entre os modelos indicarão erros durante a recuperação, seja no processo manual ou na ferramenta, portanto as diferenças deverão ser escrutinadas a fim de se identificar de onde as falhas advém.

Resumidamente, abaixo são apresentados os passos a serem tomados para a verificação da capacidade da ferramenta quanto a comparação de modelos:

1. Desenhar o modelo arquitetural de especificação no ambiente Bellatrix
2. Recuperar manualmente o modelo arquitetural implementado no projeto
3. Comparar manualmente o modelo de especificação com o modelo manualmente recuperado
4. Analisar os resultados obtidos pela ferramenta e verificar sua corretude

E a seguir, os passos para verificação da capacidade de recuperação arquitetural:

1. Recuperar manualmente o modelo arquitetural implementado no projeto

2. Recuperar, através da ferramenta, o modelo arquitetural implementado no projeto

3. Comparar o modelo recuperado pela ferramenta com o modelo recuperado manualmente

4. Analisar os resultados obtidos e revisar casos em que hajam diferenças entre os modelos

Após as validações anteriores, existirá suficiente grau de confiança na implementação da ferramenta para analisar a evolução do projeto MobileMedia. Dado que as arquiteturas da terceira e quarta versões foram recuperadas pela ferramenta e que os modelos arquiteturais de especificação das mesmas versões também foram desenhados dentro do ambiente da ferramenta, poderemos utilizar a capacidade de comparação de modelos da ferramenta para identificar se ocorreu algum tipo de erosão arquitetural durante a evolução da arquitetura entre as versões sob estudo.

Caso identificado algum tipo de erosão arquitetural, aplicaremos a atividade de *Adequação* do método Meissa, descrita na Seção 3.1.4. Isso significa que após análise dos elementos erodidos, a arquitetura recuperada deverá ser ajustada. Isto é, a fim de se mitigar a erosão da arquitetura, ou a arquitetura recuperada poderá ser aceita pelo arquiteto, reconhecendo-se uma falha na especificação da arquitetura planejada, ou a arquitetura recuperada é alterada a fim de torná-la concordante com a especificação, ou então uma mistura de ambas abordagens, na qual alguns elementos erodidos serão alterados e outros mantidos.

Após a atividade de adequação ser realizada, as mudanças arquiteturais serão utilizadas para verificar a capacidade de geração de código da ferramenta. A arquitetura adequada será submetida a ferramenta para a geração de código-fonte. Então a ferramenta será novamente ativada para recuperar a arquitetura do código-fonte gerado. Espera-se que o modelo arquitetural recuperado, após a geração de código, e o modelo adequado não apresentem diferenças, indicando que a geração de código realmente alterou o código-fonte do projeto de forma a corrigir as mudanças arquiteturais realizadas pelo arquiteto.

Assim, abaixo resume-se os passos para a verificação da capacidade de geração de código da ferramenta:

1. Comparar o modelo arquitetural recuperado com o modelo arquitetural especificado

2. Aplicar a atividade de *Adequação* do método Meissa para correção de problemas de erosão arquitetural

3. Submeter o modelo arquitetural após adequação para geração de código-fonte

4. Recuperar o modelo arquitetural do projeto após a geração de código-fonte

5. Comparar o modelo recuperado após a geração de código-fonte contra o modelo arquitetural adequado

6. Analisar os resultados obtidos e revisar casos em que hajam diferenças entre os modelos

5.2.3 Execução do estudo de caso 1

Comparação de modelos arquiteturais

Para a comparação dos modelos arquiteturais, inicialmente desenhou-se, dentro da ferramenta, o diagrama arquitetural de especificação da terceira versão do projeto MobileMedia. O modelo original foi construído através da ferramenta de modelagem Jude [31] e foi disponibilizado juntamente com o código-fonte em [54].

A seguir, é detalhada cada atividade do estudo de caso utilizando a terceira versão do projeto MobileMedia.

A Figura 5.1 mostra o diagrama arquitetural de especificação da terceira versão do projeto MobileMedia, como disponível em [54]. E a Figura 5.2 representa o mesmo modelo após sua criação dentro do ambiente Bellatrix.

A seguir, foi realizada a recuperação manual da arquitetura. Como mencionado na Seção 5.2.2, o código-fonte do projeto MobileMedia segue o modelo de implementação de componentes COSMOS*. Para reduzir o esforço necessário durante a recuperação manual da arquitetura, bem como para preparar o projeto para a recuperação automatizada, o código-fonte original do projeto MobileMedia foi submetido ao verificador COSMOS* implementado como parte da ferramenta Meissa. Como discutido na seção 4.6.3, o verificador irá indicar inconformidades e inconsistências encontradas no código-fonte, tendo por base a especificação COSMOS*.

Para a terceira versão do projeto MobileMedia, a tabela 5.1 mostra os erros encontrados durante a execução do verificador. Para um resumo dos erros por tipo, a tabela 5.2 é apresentada.

Os erros encontrados pela ferramenta foram divididos em três categorias: (i) erros de **implementação** estão relacionados ao não cumprimento das especificações do modelo de implementação de componentes COSMOS*, ou seja, inconformidades quanto a especificação; (ii) erros devido à implementação da ferramenta **Meissa**, estes não são erros associados ao projeto sob análise, mas limitações da ferramenta ou da forma como erros são reportados; e (iii) erros devido à imprecisão ou omissão da especificação **COSMOS*** para as quais a implementação do verificador teve que definir suas expectativas.

Uma breve análise dos erros encontrados ao expor o código-fonte ao verificador revela alguns pontos interessantes. Como, por exemplo, cerca de 20% dos erros encontrados

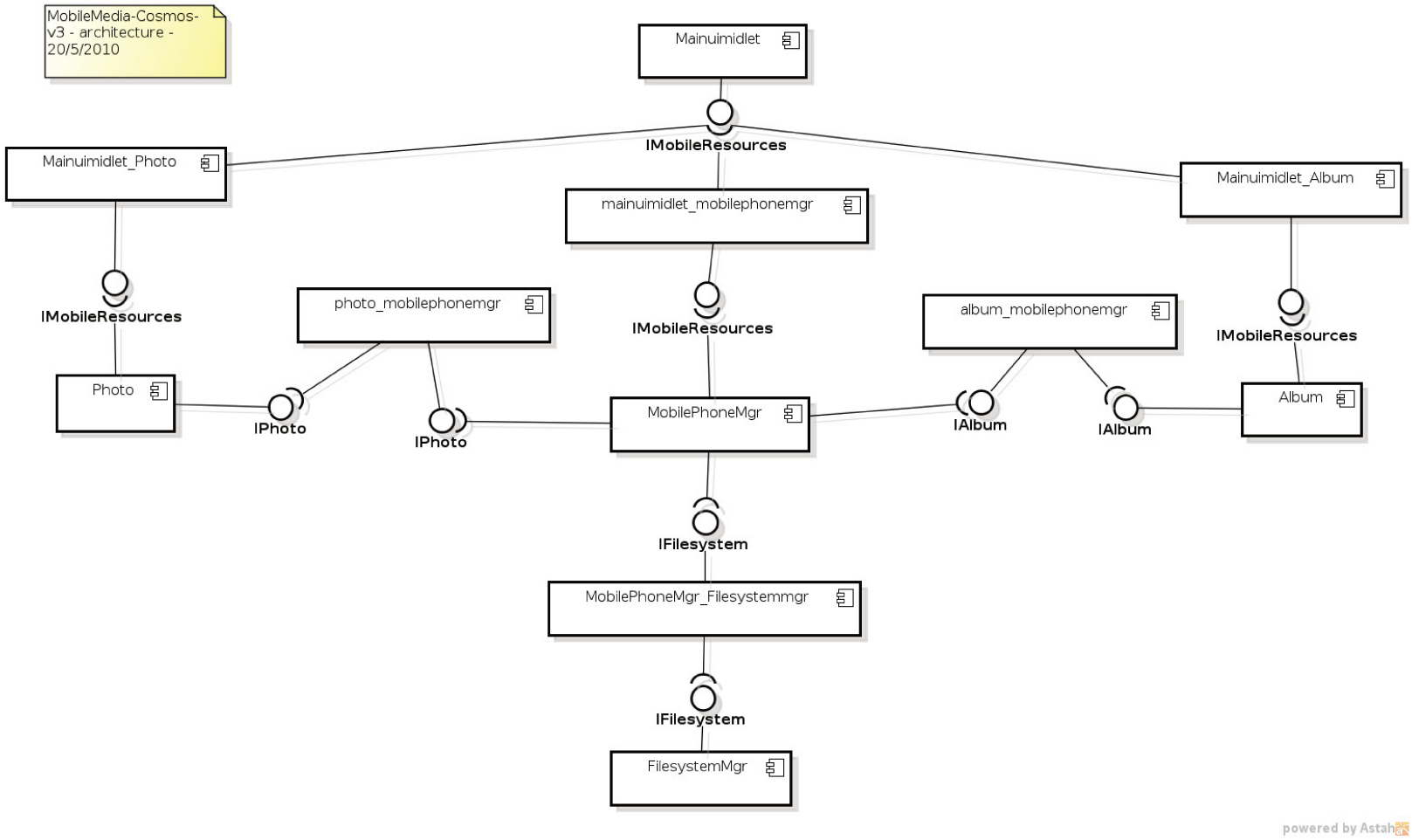


Figura 5.1: Modelo da arquitetura da terceira versão do projeto MobileMedia no formato original disponível em [54].

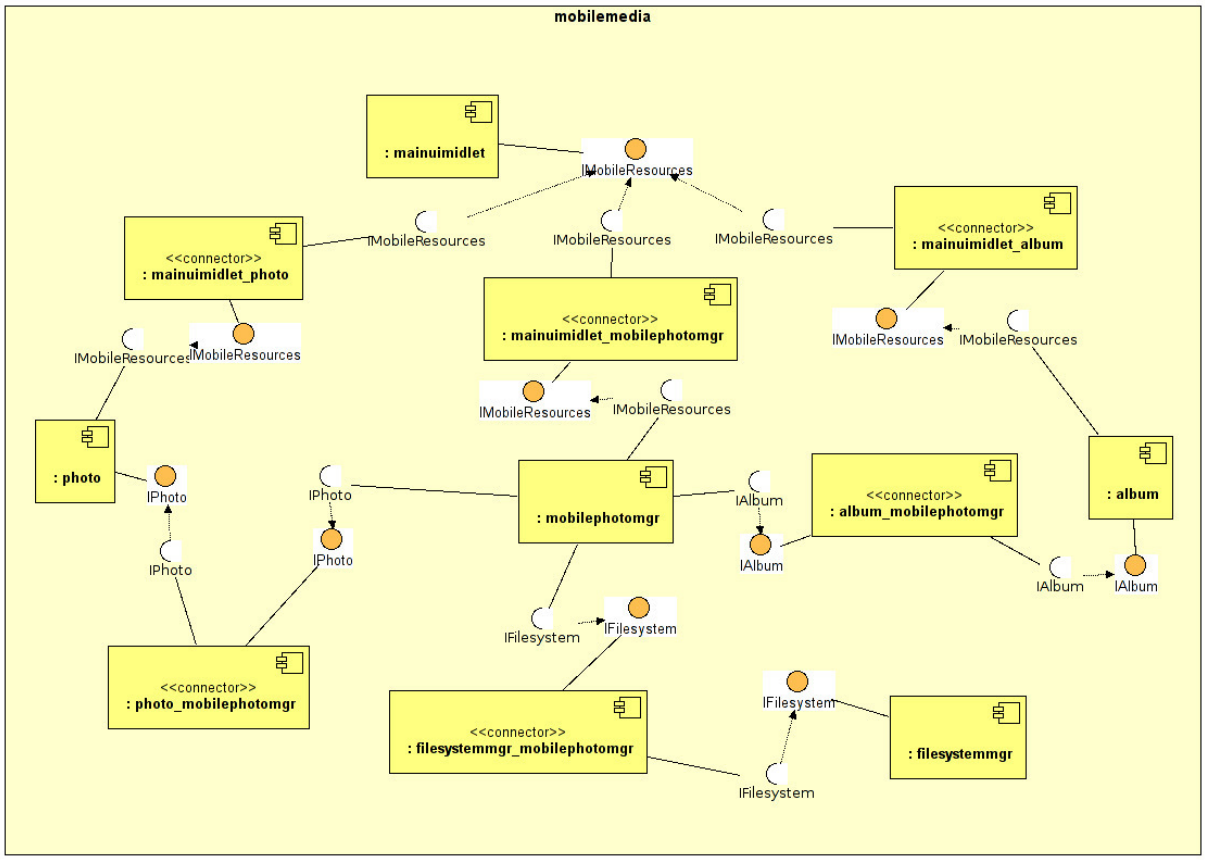


Figura 5.2: Modelo da arquitetura da terceira versão do projeto MobileMedia reproduzido dentro do ambiente Bellatrix.

Tipo de erro	Problema	Ocorrência	Frequência
Implementação	Elemento não presente	2	2,08%
Implementação	Nomenclatura	10	10,42%
Implementação	Visibilidade	19	19,79%
Implementação	Interface não declarada	29	30,21%
Meissa	Erro duplicado	8	8,33%
Meissa	Limitação	16	16,67%
COSMOS*	Imposição	12	12,50%

Tabela 5.1: Relatório de erros encontrados pelo verificador COSMOS* na terceira versão do projeto MobileMedia.

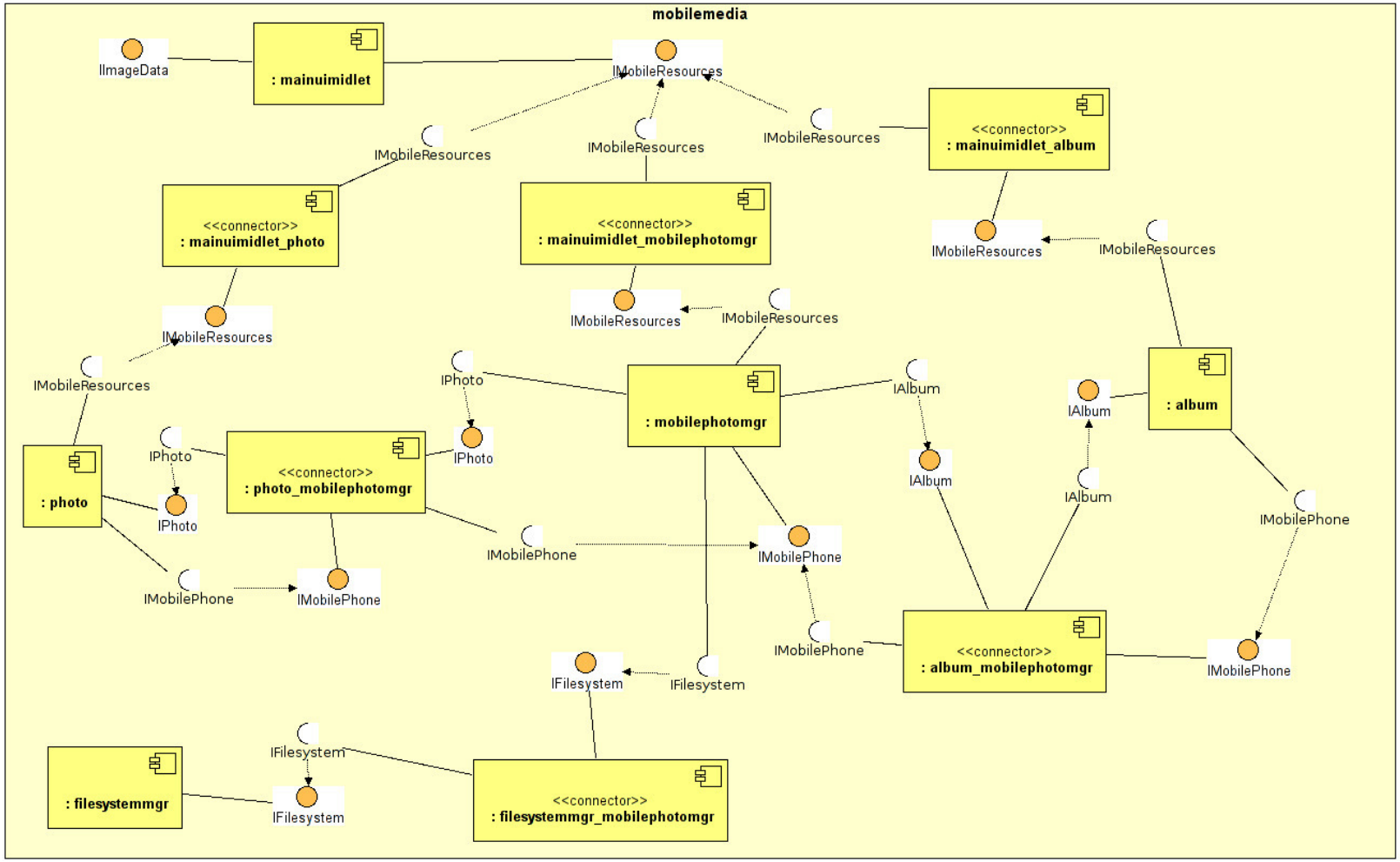


Figura 5.3: Modelo da arquitetura da terceira versão do projeto MobileMedia manualmente recuperado a partir do código-fonte.

Tipo de erro	Ocorrência	Frequência
COSMOS*	12	12,50%
Meissa	24	25,00%
Implementação	60	62,50%
Total	96	100%

Tabela 5.2: Resumo das proporções dos tipos de erros descritos na tabela 5.1.

estavam relacionados a **visibilidade** de elementos do código-fonte. Isso significa que, apesar de um modelo de implementação de componentes ter sido utilizado para a implementação do código, seria possível que classes pertencentes a diferentes componentes trocassem mensagens, minando os benefícios do uso de um modelo de implementação de componentes.

Outro exemplo são os relativos 30% de erros devido a **não declaração de interfaces**. No modelo COSMOS* um componente deve declarar quais são as interfaces providas e requeridas em sua implementação da classe *Manager*. Contudo, todos os erros de não declaração de interface são exclusivos de interfaces requeridas. Uma possível explicação para essa situação é o fato de que a declaração de interfaces providas é *essencial* para o funcionamento da aplicação (devido à estrutura imposta pelo modelo COSMOS*), enquanto que a declaração de interfaces requeridas serve apenas para situações mais incomuns, como a ligação dinâmica de componentes ou metadado para a recuperação arquitetural, e portanto, não recebem a devida atenção dos desenvolvedores.

Os erros anteriormente discutidos estão relacionados à implementação do código sob análise, contudo 37,50% dos erros são devidos a limitações do verificador ou da especificação do modelo COSMOS*. Desses 37,50%, 12,50 pontos percentuais são devidos a limitações na especificação do modelo COSMOS*. A especificação do modelo define que a ligação entre interfaces providas e requeridas por componentes de uma arquitetura deve ocorrer na classe *Manager* do super-componente relacionado a tal arquitetura. Entretanto, a especificação é omissa ao definir em que elemento da classe (por exemplo, no construtor ou em um método específico) a ligação deve ser implementada. Neste caso, a ferramenta Meissa adotou que essa implementação ocorra no construtor da classe. Os trechos de código que foram implementados diferentemente emitiram os 12,50% de erros inicialmente mencionados.

O restante dos erros relacionados a limitações do verificador COSMOS* divide-se entre: (i) *erros duplicados*, que, em geral, ocorrem devido a um efeito dominó, como por exemplo a não declaração de uma interface requerida causará a emissão de um erro reportando a não declaração da interface e outro erro quando houver a tentativa de prover tal interface requerida ao componente que a necessita; e (ii) *limitações* da ferramenta, que precisa restringir situações devido às dificuldades em tratá-las. Neste estudo de caso, os 16

erros de limitação da ferramenta foram causados por uma falha existente no componente *MoDisco* utilizado para a obtenção do modelo KDM a partir do código-fonte Java. Tal falha não permite a análise de variáveis declaradas e instanciadas em diferentes escopos. Devido a isso, a ferramenta Meissa restringe que a declaração de variáveis *IManager* (para detalhes sore a interface vide apêndice A) ocorra junto com sua instanciação.

Uma vez corrigidos os erros reportados pelo verificador, a arquitetura foi recuperada manualmente. No apêndice A existem mais informações para a recuperação da arquitetura utilizando o modelo COSMOS*. Resumidamente, a recuperação manual foi realizada através dos seguintes passos:

1. A partir da classe *Manager* do componente *MobileMedia*, que define a arquitetura do sistema, identifica-se cada componente e conector que compõe a arquitetura do sistema. Para isso, são consideradas as chamadas ao método *createInstance()* de classes *ComponentFactory*. Cada componente definirá sua classe *ComponentFactory* que permitirá a criação de uma nova instância daquele componente.

2. Para cada componente identificado anteriormente, é analisada a estrutura do pacote que o define. Como descrito no apêndice A, as interfaces providas e requeridas são definidas por cada componente nos pacotes *spec.prov* e *spec.req*, respectivamente. Assim, são recuperadas as interfaces relacionadas a cada componente.

3. Retorna-se a implementação da classe *Manager* do componente *MobileMedia* para identificar as relações entre os componentes e conectores do sistema. O componente *MobileMedia* funciona como um componente composto, representando o sistema. Cada componente do sistema é, de fato, um subcomponente que pertence à arquitetura do componente *MobileMedia*. Assim, é ele o responsável por declarar cada subcomponente, os conectores e realizar o mapeamento das interfaces providas e requeridas. A recuperação desses relacionamentos é feita como descrito a seguir:

   (a) Cada chamada da forma [nomeDoComponente].getProvidedInterface ([nomeDaInterface]) indica que a interface "*nomeDaInterface*" é provida pelo componente "*nomeDoComponente*". Tal chamada retornará um objeto que implementa a interface provida.

   (b) Cada chamada da forma [nomeDoComponente].setRequiredInterface ([nomeDaInterface], [instanciaDaInterface]) indica que a interface "*nomeDaInterface*", requerida pelo componente "*nomeDoComponente*", está sendo implementada e provida pelo objeto "*instanciaDaInterface*". Este objeto deve ser obtido através da chamada *getProvidedInterface*.

O modelo arquitetural manualmente recuperado é apresentado na Figura 5.3.

A seguir, o modelo manualmente recuperado (Figura 5.3) e o modelo de especificação (Figura 5.2) foram comparados, tanto manualmente, como através da ferramenta Meissa. A expectativa é que tais modelos não apresentem diferença alguma, indicando não haver inconformidades entre a arquitetura e sua especificação. Entretanto, o resultado encontrado mostra que uma nova interface (*IMobilePhone*), adicionada ao modelo manualmente recuperado (e portanto à arquitetura implementada pelo código-fonte). Essa interface foi exposta pelo componente *MobilePhotoMgr* e requerida pelos componentes *Album* e *Photo*. Isso fez com que alterações também ocorressem nos conectores que ligam tais componentes entre si.

Outra diferença encontrada foi a adição, no modelo manualmente recuperado, da interface provida *IImageData* pelo componente *MainUiMidlet*. Contudo, essa interface não é consumida por nenhum outro componente da arquitetura, tornando sua existência, pouco relevante para o sistema em questão.

A Figura 5.4 mostra as diferenças entre o modelo de especificação original da arquitetura e o modelo manualmente recuperado. É possível ver, coloridos em verde, as novas interfaces adicionadas à arquitetura e suas respectivas ligações. Essa figura foi obtida através da ferramenta Meissa, utilizando sua funcionalidade de comparação de modelos arquiteturais. O resultado obtido corrobora com o resultado encontrado pelo processo manual.

Recuperação da arquitetura implementada

Na seção anterior foi brevemente discutido como a recuperação manual da arquitetura implementada foi realizada. Nesta seção, é executada a recuperação da arquitetura utilizando o processo automatizado pela ferramenta Meissa. O código-fonte da terceira versão do projeto MobileMedia será submetido a ferramenta para que seja recuperado o modelo arquitetural implementado no projeto. A expectativa é que tal modelo seja idêntico ao modelo manualmente recuperado (Figura 5.3), indicando que a ferramenta é capaz de corretamente recuperar a arquitetura implementado no projeto.

A Figura 5.5 apresenta o resultado da comparação entre o modelo manualmente recuperado e o obtido através da recuperação automática. A visão em árvore do resultado da comparação provê uma maneira sucinta e prática para que o arquiteto possa analisar sistemas com muitos componentes e potencialmente muitas alterações. Selecionando o nó de interesse e selecionando a opção “Visualizar diferenças no diagrama” leva o arquiteto para o diagrama indicando as mudanças encontradas pelo algoritmo de comparação. Nesta visão, a raiz da árvore representa o sistema em si. Filhos imediatos da raiz representam

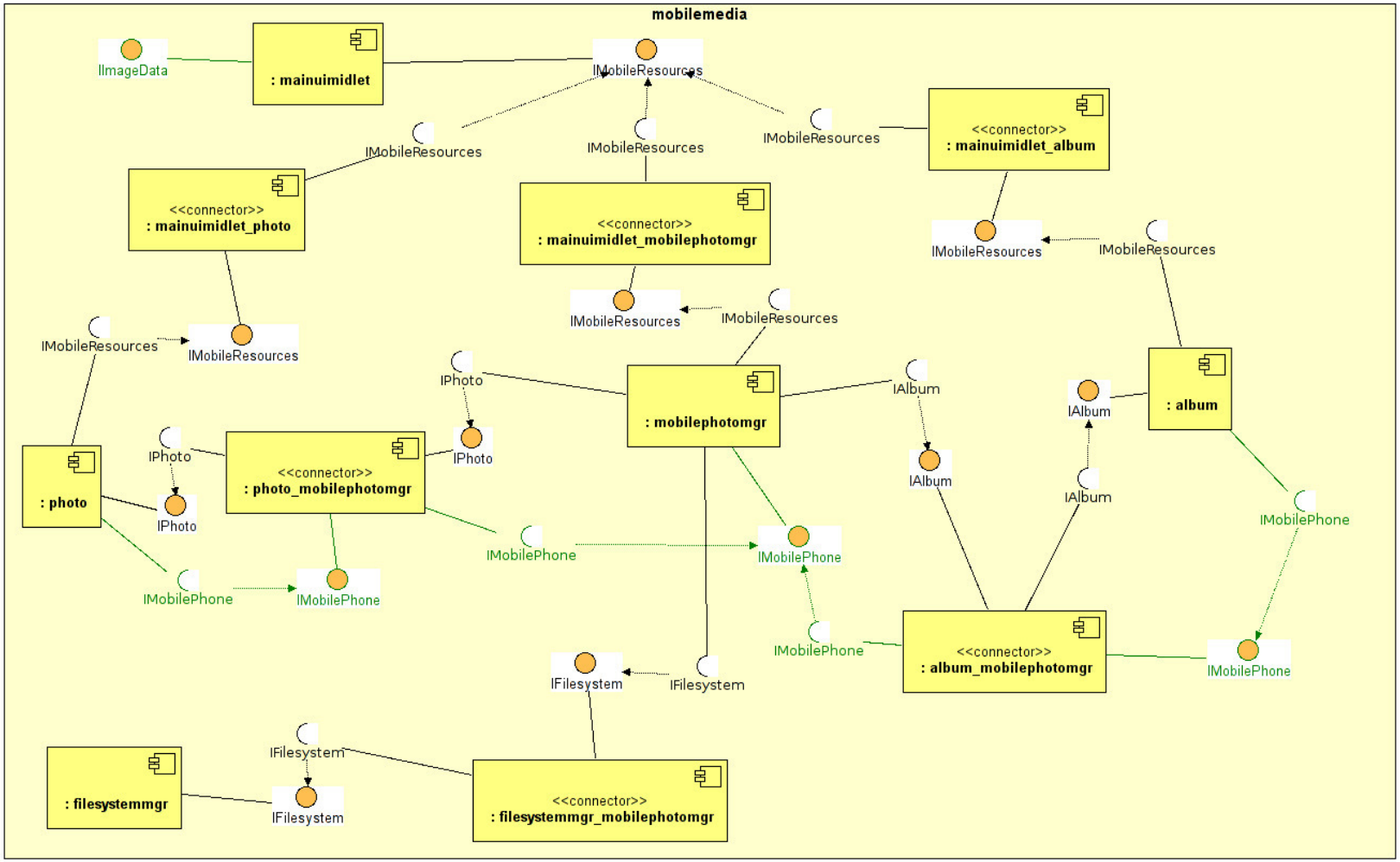


Figura 5.4: Diagrama representando as diferenças entre o modelo original de especificação da arquitetura da terceira versão do projeto MobileMedia e o modelo arquitetural manualmente recuperado a partir do código-fonte.

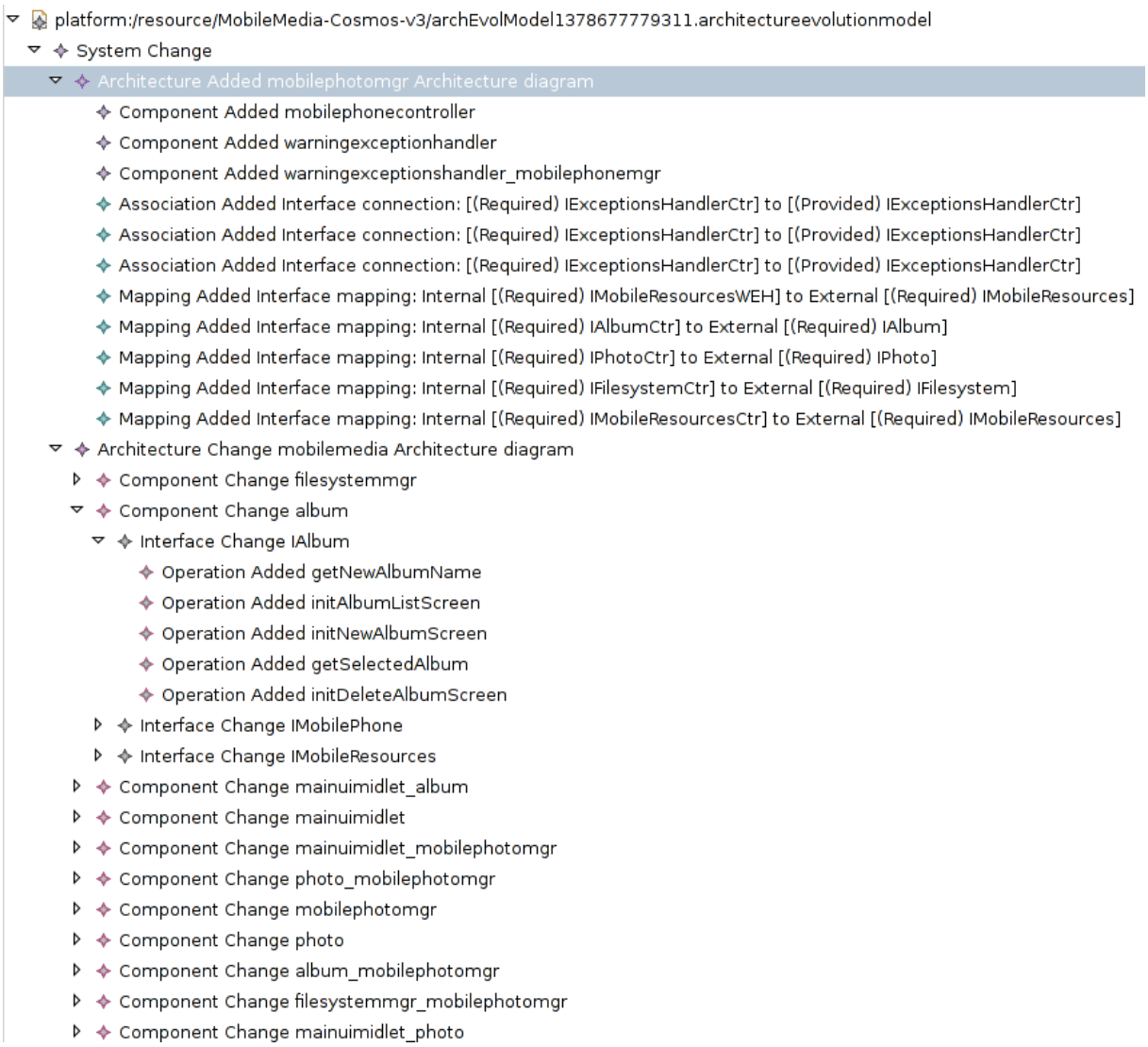


Figura 5.5: Visão em árvore do resultado da comparação entre os modelos arquiteturais recuperados manual e automaticamente pela ferramente Meissa.

mudanças arquiteturais de componentes compostos[1]. Por sua vez, nós no terceiro nível representam detalhes das alterações da arquitetura, como componentes adicionados ou novos mapeamentos entre interfaces externas e internas. Essa representação hierárquica segue a diante até o nível das interfaces, no qual operações são adicionadas ou removidas. Para mais detalhes, vide a Seção 4.4.3.

Na visão em árvore do resultado verifica-se que existem dois nós que representam mudanças em diagramas arquiteturais, sendo elas: *"Architecture added mobilephotomgr"* e *"Architecture change mobilemedia"*. A seguir, cada um desses resultados é discutido separadamente.

A primeira mudança, **"Architecture added mobilephotomgr"**, significa que uma nova arquitetura foi especificada para o componente *MobilePhotoMgr*. Isto é, na arquitetura recuperada pela ferramenta, o componente *MobilePhotoMgr* é, de fato, um componente composto, possuindo uma arquitetura própria que define como seus subcomponentes se relacionam. Entretanto, a recuperação manual da arquitetura não identificou que tal componente possuía uma arquitetura interna. A razão dessa discrepância entre o resultado manual e o obtido pela ferramenta se deve a como o processo de recuperação manual ocorreu. Revisitando, na seção anterior, os passos utilizados para recuperar manualmente a arquitetura, fica claro que a recuperação manual, influenciada pelo modelo arquitetural de especificação, não considerou que nenhum outro componente além de *MobileMedia*, componente composto que define o sistema, poderia ter uma arquitetura interna. Por conta disso, os passos de recuperação manual que analisam a arquitetura interna de cada componente foram restringidos apenas ao *MobileMedia*. A Figura 5.6 representa o modelo arquitetural do componente composto *MobilePhotoMgr* da terceira versão do projeto MobileMedia.

Revisitando as classes *Manager* de cada componente do sistema em busca de outros componentes compostos que poderiam ter passado despercebidos tanto pela recuperação manual quanto pela ferramenta, verifica-se que, de fato, *MobileMedia* e *MobilePhotoMgr* são os únicos componentes compostos do sistema.

A segunda mudança, **"Architecture change mobilemedia"**, indica que a arquitetura do componente composto *MobileMedia* existe tanto no modelo manualmente recuperado, como no recuperado pela ferramenta, mas que ambos não são exatamente iguais. Cada nó abaixo deste indica o tipo de mudança, por exemplo, o nó *"Componente change album"* indica que mudanças ocorreram e afetaram o componente *Album*. Navegando em cada um dos nós pertencentes a ele, percebe-se que as mudanças ocorreram nas interfaces dos componentes; e todas elas, são, na verdade, adições de operações às interfaces. Em resumo, todas as mudanças relacionadas ao componente *MobileMedia* são apenas adições de operações às interfaces de seus subcomponentes. O motivo dessa situação é que, ao

[1]No ambiente Bellatrix, a arquitetura de um sistema é modelada como um componente composto.

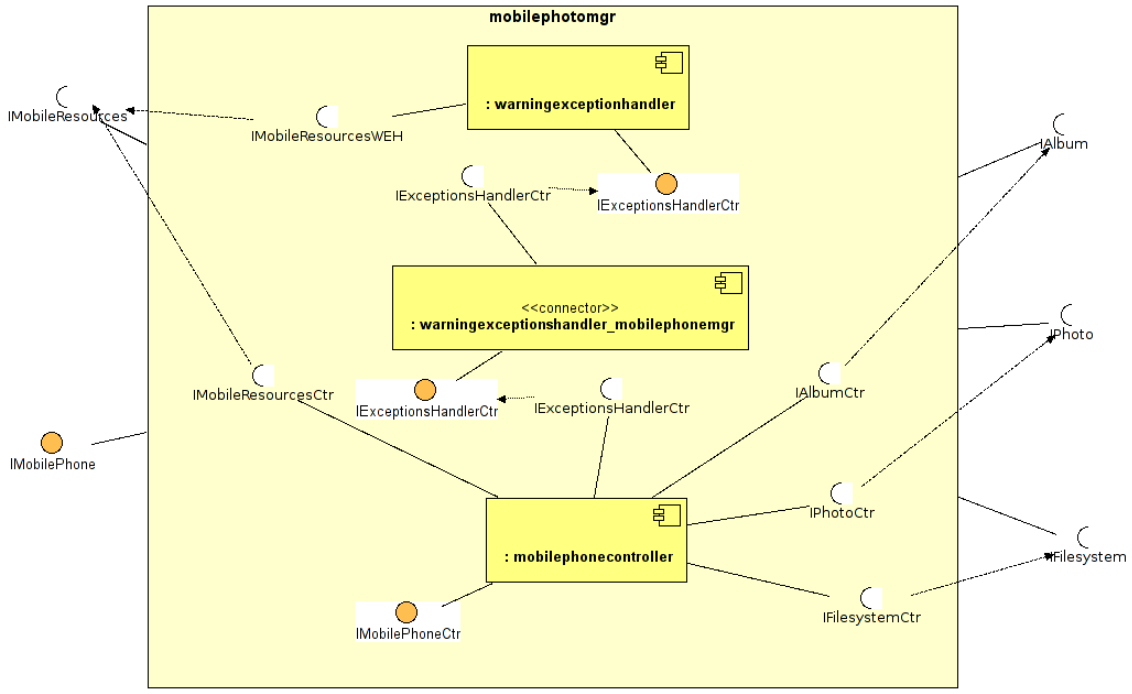


Figura 5.6: Modelo arquitetural do componente composto *MobilePhotoMgr* da terceira versão do projeto MobileMedia.

recuperar manualmente a arquitetura do sistema, não houve o cuidado de se recuperar cada operação de cada interface existente. Ao invés disso, cada interface, provida ou requerida, foi apenas adicionada ao modelo, sem nenhuma operação nela especificada. Desta forma, quando a ferramenta comparou ambas arquiteturas, a recuperada manualmente não possuía operações nas interfaces, enquanto a recuperada pela ferramenta havia identificado-as e corretamente adicionado-as ao modelo.

Em resumo, a recuperação automatizada da arquitetura corretamente obteve a arquitetura implementada do sistema, trazendo ainda mais detalhes do que o modelo manualmente recuperado.

Correção de problemas de erosão arquitetural

Nas seções anteriores, a terceira versão do projeto MobileMedia foi utilizada para a validação inicial das capacidades de comparação e recuperação arquitetural da ferramenta. Para esta seção, entretanto, será utilizada também a quarta versão do projeto MobileMedia. Todos os passos anteriores foram aplicados também a essa nova versão, resultando no diagrama da Figura 5.7, que mostra as diferenças entre a arquitetura especificada para a quarta versão e a arquitetura recuperada pela ferramenta com base no código-fonte.

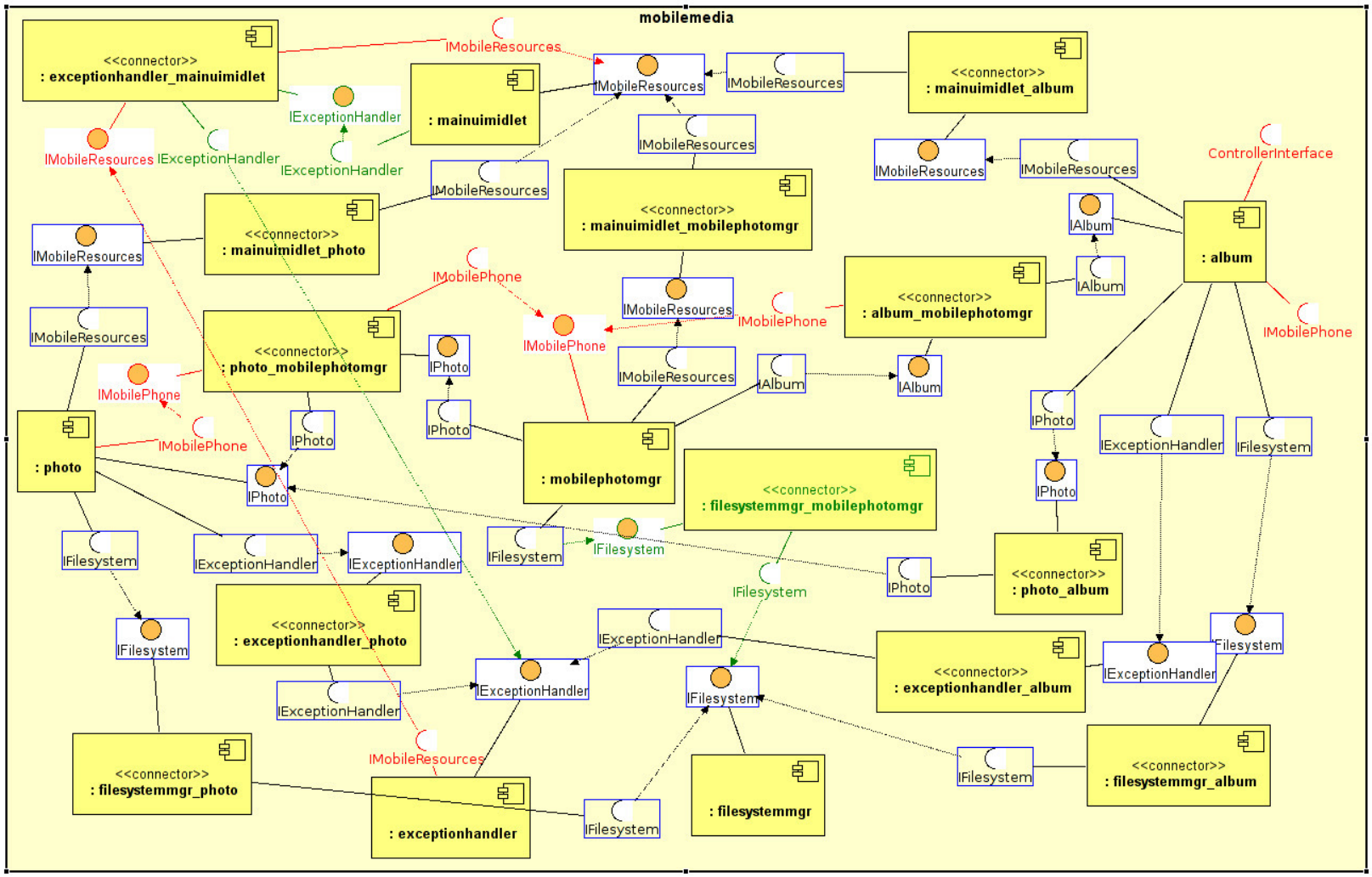


Figura 5.7: Diagrama representando as diferenças entre o modelo original de especificação da arquitetura da quarta versão do projeto MobileMedia e o modelo arquitetural recuperado a partir do código-fonte pela ferramenta Meissa.

Cor	Presente na implementação	Presente na especificação
Preta	Sim	Sim
Azul	Sim	Sim
Vermelha	Sim	Não
Verde	Não	Sim

Tabela 5.3: Legenda das cores utilizadas na comparação de modelos arquiteturais da Figura 5.7.

A ordem com que os modelos são providos à ferramenta influenciam nas cores aplicadas ao diagrama para indicar as mudanças ocorridas. O primeiro modelo fornecido a ferramenta é considerado o modelo original, usado como base para as comparações. O segundo modelo é considerado o resultado da evolução arquitetural. Desta forma, novos elementos, presentes no segundo modelo, mas não no primeiro, são coloridos em verde, ou seja, o elemento foi adicionado durante a evolução arquitetural[2] Já se colorido em vermelho, isso significa o oposto: o elemento foi removido durante a evolução[3]. Se marcado em azul, isso significa que o elemento teve alguma propriedade alterada, por exemplo, uma interface colorida em azul indica que operações foram adicionadas ou removidas.

Para a Figura 5.7, a ordem dos modelos foi invertida, ou seja, o modelo inicial é o recuperado a partir da implementação da quarta versão do projeto e o final é o modelo de especificação de tal versão. A inversão foi feita como forma de validar a corretude da comparação dos modelos, independentemente da ordem com que os modelos são providos a ferramenta, ela deve ser capaz de compará-los e marcá-los com as cores apropriadas a fim de destacar as mudanças arquiteturais.

Assim o significo das cores discutido anteriormente aplica-se à figura em ordem oposta. A fim de facilitar a análise, a tabela 5.3 relaciona as cores aplicadas a Figura 5.7 com seu respectivo significado.

Analisando a Figura 5.7 com apoio da tabela 5.3, destaca-se o número de interfaces envoltas em uma caixa colorida em azul. A cor indica que houve alterações internas à interface, por exemplo, adição e remoção de operações da interface. Isso ocorreu devido ao fato do modelo de especificação desenhado dentro do ambiente Bellatrix não definir as operações de cada interface. Assim, ao recuperar cada interface arquitetural, a ferramenta também recuperou as operações definidas em cada interface. Com isso, a comparação das interfaces indicou a diferença entre as operações especificadas e as recuperadas. Para o objetivo desta análise, pode-se ignorar as alterações no nível de operações das interfaces. O interesse está focado em compreender as diferenças entre a implementação e a

[2]Ou, considerando a análise entre modelo de especificação e implementação, um elemento colorido em verde significa que o elemento foi implementado, mas não especificado.

[3]Ou, em um cenário de erosão arquitetural, o elemento foi especificado, mas não foi implementado.

especificação, no nível arquitetural, e decidir como tratá-las, justamente como descrito na atividade de Adequação, do método Meissa (Seção 3.1.4).

Continuando a análise, verifica-se que uma série de interfaces *IMobilePhone* foram adicionadas ao diagrama, ou seja, o código-fonte implementou tais componentes expondo e requerendo essa nova interface. Ela é provida pelo componente *MobilePhotoMgr* e requerida pelos componentes *Photo* e *Album*, ainda que apenas o primeiro desses dois componentes teve a ligação entre a interface provida e a requerida completamente realizada. Para o componente *Album*, tal ligação limita-se ao conector entre ele e *MobilePhotoMgr*, não chegando a, de fato, prover a interface requerida pelo componente.

Seguindo com a análise, percebe-se que um conector está faltando na implementação: o *FilesystemMgr-MobilePhotoMgr*, que seria responsável por conectar a interface requerida *IFilesystem* pelo componente *MobilePhotoMgr* com a mesma interface provida pelo componente *MobilePhotoMgr*. Isso significa que a implementação do componente *MobilePhotoMgr* não tem acesso nenhum ao sistema de arquivos, tendo sua funcionalidade restringida ou totalmente incapacitava. Uma outra possibilidade é que os desenvolvedores tenham codificado toda a comunicação com o sistema de arquivos dentro do próprio componente *MobilePhotoMgr*, o que, do ponto de vista arquitetural, seria uma infração muito mais grave das premissas arquiteturais do que simplesmente a falta da conexão entre interfaces providas e requeridas.

Próximo na lista de desvios arquiteturais está o conector *ExceptionHandler- MainUiMidlet*. Ele foi especificado com o fim de ligar a interface requerida *IExceptionHandler* por *MainUiMidlet* com a mesma interface provida pelo componente *ExceptionHandler*. Entretanto, tal conector não só deixou de executar tal ligação, como tornou-se responsável por permitir que a interface *IMobileResources* provida pelo componente *MainUiMidlet* fosse consumida pelo componente *ExceptionHandler*; um relacionamento não definido no modelo arquitetural de especificação do projeto. Vários podem ser os motivos que levaram a essa situação. Um deles, por exemplo, pode ser que tenha havido uma falha na especificação do componente *ExceptionHandler* não considerando a necessidade de dependências externas, fazendo com que os desenvolvedores expusessem a interface requerida *IMobileResources* como solução para suprir a recém descoberta dependência externa do componente *ExceptionHandler*.

Apenas para fins de completude, deve-se mencionar que a implementação do componente *Album* requer a interface *ControllerInterface*, entretanto não parece existir conector ou componente capaz de suprir tal interface. Analisando a implementação do componente, essa interface, de fato, não é necessária; possivelmente ela foi esquecida pelos desenvolvedores e nunca removida.

De acordo com o método Meissa, o arquiteto deve decidir quais diferenças entre a implementação e especificação são de fatos desvios arquiteturais e devem ser corrigidas e

quais são possíveis falhas de especificação e devem ser mantidas, ou talvez ajustadas de alguma maneira a melhor se adequarem às premissas da arquitetura do projeto. A tabela 5.4 resume as diferenças arquiteturais, um provável motivo que causou a diferença e uma possível decisão do arquiteto responsável pela atividade de Adequação.

Baseando-se na tabela 5.4, o modelo arquitetural recuperado foi alterado a fim de corresponder as mudanças arquiteturais desejadas. Após atualizado, o modelo foi provido à ferramenta, para que ela gerasse as alterações no código-fonte a fim de materializar as mudanças requisitadas pelo arquiteto. Ao término da geração de código, a análise do código-fonte do projeto indica que não houve alterações inesperadas, removendo ou adicionando código a arquivos não relacionados as mudanças descritas na tabela que sumariza as diferenças.

A fim de validar a geração automática de código, a ferramenta Meissa foi utilizada para recuperar a arquitetura implementada pelo código-fonte após a geração automática de código. Após recuperado o modelo, a ferramenta foi utilizada para comprar tal modelo com o modelo atualizado com as mudanças desejadas pelo arquiteto. A Figura 5.8 mostra a representação em árvore do resultado da comparação. Como esperado, não foram apresentadas diferenças entre os dois modelos, indicando que, de fato, as alterações arquiteturais desejadas pelo arquiteto foram materializadas no código-fonte.

Figura 5.8: Resultado da comparação entre o modelo arquitetural atualizado com as mudanças desejadas pelo arquiteto e o modelo arquitetural recuperado após a execução da geração de código.

Diferença #1 Adição não especificada das interfaces *IMobilePhone* **Razão da diferença** Necessidade do uso da operação *postCommand* para executar a navegação entre páginas da aplicação móvel. **Decisão do arquiteto** A navegação entre páginas deve ser uma responsabilidade do componente *MobilePhotoMgr* e não deve ser delegada. Interfaces devem ser removidas e o código deve ser ajustado.
Diferença #2 Componente *MobilePhotoMgr* não possui sua interface requerida *IFilesystem* suprida por nenhum conector. **Razão da diferença** Falha de implementação. **Decisão do arquiteto** Adicionar o conector *FilesystemMgr-MobilePhotoMgr* e relacionar as interfaces. Rever com a equipe de testes os casos de teste para o componente *MobilePhotoMgr* em relação ao acesso ao sistema de arquivos.
Diferença #3 Conector *ExceptionHandler-MainUiMidlet* não supre a interface requerida *IExceptionHandler* pelo componente *MainUiMidlet*. **Razão da diferença** Falha de implementação. **Decisão do arquiteto** Alterar o conector para que a interface mencionada seja corretamente provida ao componente.
Diferença #4 Conector *ExceptionHandler-MainUiMidlet* provê a interface requerida *IMobileResources* pelo componente *ExceptionHandler*. **Razão da diferença** O componente que lida com as exceções necessita de tal interface para mostrar mensagens de erro na interface gráfica da aplicação móvel. **Decisão do arquiteto** Falha de especificação. Essa alteração deve ser mantida.
Diferença #5 Adição da interface não especificada *ControllerInterface*. **Razão da diferença** Falha de implementação. **Decisão do arquiteto** Remoção da interface.

Tabela 5.4: Resumo das diferenças arquiteturais encontradas ao analisar a quarta versão do projeto MobileMedia e a respectiva decisão do arquiteto quanto a correção de tais diferenças.

5.2.4 Análise dos resultados do estudo de caso 1

Na Seção 5.2.3, foram discutidos os erros reportados pela ferramenta ao submeter o código-fonte do projeto MobileMedia ao validador COSMOS*. É natural que existam inconformidades entre o código-fonte e o modelo de implementação de componentes quando não existe constante apoio computacional a fim de se validar e apontar tais inconsistências. Esse é um problema inerente aos modelos de implementação de componentes que não requerem alterações na linguagem de programação e o uso de um compiladores próprios, como ArchJava [3]. Eventualmente programadores cometerão erros, se esquecerão de adicionar porções de código requeridas pelo modelo de componentes ou simplesmente basearam-se na implementação de um componente já problemático para desenvolver um novo. Não havendo apoio computacional que permita de maneira fácil e direta a exposição de tais problemas, eles apenas tendem a se acumular no sistema, minando a utilidade que a adoção de um modelo de implementação de componentes trás ao projeto.

Tendo em vista que a conformidade do código-fonte com a especificação do modelo de implementação de componentes é essencial para que atividades, como a recuperação automática da arquitetura possam ocorrer, é fundamental a presença de apoio computacional para garantir a conformidade entre a implementação e o modelo de componentes adotado. Neste ponto, a ferramenta Meissa ao ser incorporada em um ambiente de desenvolvimento integrado, como o Eclipse, propicia meios aos desenvolvedores para validarem que o código sendo produzido adequa-se ao especificado pelo modelo de componentes adotado.

Na mesma seção, os experimentos em relação a comparação de modelos arquiteturais corroboram com a expectativa de que o apoio computacional para tal tarefa é extremamente bem-vindo, livre de erros [4], muito mais ágil e prático que a mesma atividade manual. A comparação de modelos arquiteturais é fundamental para prover as informações necessárias ao arquiteto para quando a atividade de Adequação (Seção 3.1.4) for executada. Nessa atividade, o arquiteto deve considerar cada diferença entre o modelo arquitetural especificado e o recuperado. Retirar do arquiteto a responsabilidade de ter que manualmente realizar a comparação trás não só enormes ganhos de tempo, mas a garantia de que problemas de erosão e desvio arquiteturais não passarão desapercebidos pelo arquiteto por haver falhas humanas durante a etapa de comparação dos modelos arquiteturais.

Na Seção 5.2.3, foi recuperada a arquitetura da terceira versão do projeto MobileMedia. Ao compará-la com a arquitetura manualmente recuperada observou-se que o componente *MobilePhotoMgr*, apesar de especificado como um componente simples, foi, de fato, implementado como um componente composto. É possível questionar se esta situação se enquadra em um caso de erosão arquitetural, uma vez que a implementação

[4] Quando corretamente implementado.

diverge da especificação da arquitetura. De um outro ponto de vista, é possível imaginar que o desenvolvedor não infringiu nenhuma premissa arquitetural, já que não havia especificação da arquitetura de tal componente.

Não havendo algum marcador no modelo de especificação, indicando que tal componente deva ser um componente simples, ao invés de um componente composto, ambos pontos de vista torna-se válidos e, como descrito na atividade de Adequação do método Meissa (Seção 3.1.4), caberá ao arquiteto decidir quais diferenças são classificadas como desvios em relação à especificação e quais mudanças são relevantes e consistentes o suficiente para serem incorporadas à arquitetura do projeto.

Outra consideração importante a ser tomada é o fato da recuperação manual da arquitetura não ter obtido a arquitetura interna do componente *MobilePhotoMgr*, mas a recuperação através da ferramenta tê-lo conseguido. Esse resultado corrobora com a motivação principal da proposta da ferramenta Meissa: a necessidade de intervenção manual para a recuperação da arquitetura é certamente propensas a falhas humanas, além de ser uma atividade extremamente fatídica e, por vezes, complexa, dependendo do tamanho do sistema em questão e do modelo de implementação de componentes utilizado.

Uma das dificuldades encontradas durante o estudo de caso realizado foi a visualização gráfica das diferenças entre dois modelos arquiteturais, como é possível ver na figura 5.7. Conforme a arquitetura do sistema vai se tornando mais complexa, com um maior número de componentes e interligações entre eles, mais difícil torna-se entendê-la utilizando-se um diagrama gráfico. Mais extenuante é ainda quando dois complexos modelos arquiteturais são comparados e o número de diferenças entre eles é considerável. O diagrama a ser visualizado estará repleto de elementos de ambos os modelos, propriamente coloridos, representando remoções, adições ou alterações dos elementos. Conforme o número de diferenças aumenta, mais informações o diagrama deverá convir e mais difícil será expor todos esses elementos de forma inteligível em uma mesma imagem.

Como forma de contornar essa dificuldade, o diagrama em forma de árvore, como o da Figura 5.5, pode ser usado. Ele descreve sucintamente as diferenças encontradas de maneira hierárquica, contudo isso limita a visão global das mudanças e como elas estão relacionadas e afetam os demais elementos do modelo. Encontrar alguma outra forma de permitir que o arquiteto acesse as diferenças arquiteturais e atue sobre elas, certamente será um útil aperfeiçoamento da ferramenta.

Na Seção 5.2.3, foram apresentados os resultados da análise das diferenças arquiteturais encontradas entre a arquitetura especificada para a quarta versão do projeto MobileMedia e a arquitetura recuperada a partir da implementação. Foram discutidas as diferenças encontradas pela ferramenta, as causas que as originaram e uma possível decisão do arquiteto em relação as alterações a serem feitas de forma a mitigar a erosão arquitetural. A ferramenta foi, então, aplicada para a geração do código-fonte que ma-

terializa as alterações arquiteturais ditadas pelo arquiteto. Como validação das etapas anteriores, a capacidade comprovada de recuperação arquitetural foi novamente utilizada para recuperar a arquitetura implementada após a atualização do código-fonte. Em posse da arquitetura atualizada, o modelo recuperado foi comparado com o modelo utilizado durante a geração do código-fonte. O resultado da comparação indicou que ambos modelos eram idênticos, o que significa que a geração de código, de fato, materializou as alterações arquiteturais desejadas.

Contudo, nessa mesma seção, pode-se perceber que não é tão fácil analisar as razões que levaram à ocorrência das disparidades vistas entre o modelo arquitetural de especificação e o de implementação. Certamente a capacidade da ferramenta de recuperar as operações definidas nas interfaces arquiteturais ajudaram a guiar a busca pelo motivo das diferenças encontradas. Entretanto, observou-se que, em vários dos casos, houve a necessidade de analisar o código-fonte por mais detalhes de implementação que levaram às decisões arquiteturais divergentes às especificadas pelo arquiteto. É sensato afirmar que são necessárias melhorias tanto de processo e ferramental para auxiliar o arquiteto a compreender os motivos que levaram aos desvios arquiteturais encontrados. Conhecendo tais razões, o arquiteto estará muito mais equipado para deliberar sobre as diferenças arquiteturais encontradas e decidir se deve aceitá-las ou corrigi-las de alguma forma, seja removendo-as ou reorganizando-as.

Este estudo de caso mostrou a capacidade do método e da ferramenta ao lidar com duas diferentes versões de um projeto de aplicação móvel. Tendo sido construído baseado em componentes e seguindo um modelo de implementação de componentes, esse sistema pode ser utilizado para diversos testes de validação da ferramenta implementada, desde o verificador COSMOS*, passando pela comparação de modelos arquiteturais, pela recuperação arquitetural a partir do código-fonte, até a geração e modificação de código-fonte do sistema para adequação de problemas arquiteturais.

Foram discutidos os resultados, ponderando as vantagens que o apoio computacional, através da ferramenta implementada, agrega a solução proposta neste trabalho. Também foram levantas as desvantagens ou limitações encontradas durante o estudo de caso. Elas foram discutidas aqui e são novamente mencionadas, com um pouco mais de detalhes, na Seção 6.3, na qual são discutidos os trabalhos futuros.

5.3 Estudo de caso 2: ferramenta Meissa

5.3.1 Descrição do estudo de caso 2

Neste estudo de caso será analisada a aplicação do método e ferramenta Meissa no próprio código-fonte da ferramenta concebida durante este projeto. Tendo seu desenvolvimento

durado pouco mais de um ano e meio e com cerca de 36 mil linhas de código[5], a ferramenta Meissa representa um bom alvo inicial na escala de grande projetos de software.

É esperado que durante este estudo de caso, sejam descobertos problemas que advém da natureza de grandes projetos, como, por exemplo, degradação de performance ou aumento excessivo no consumo de recursos. Tais problemas trariam dificuldades a aplicação da solução proposta em sistemas de grande porte, o que impactaria no uso prático da solução proposta.

5.3.2 Planejamento do estudo de caso 2

A ferramenta Meissa foi implementada utilizando o modelo de implementação de componentes COSMOS* e é distribuída através de um conjunto de *plugins* no ambiente Eclipse, como discutido na seção 4.1.1. Cada componente é encapsulado em um *plugin*. Isso permite que implementações de novos componentes que lidem com diferentes modelos de implementação de componentes e linguagens de programação sejam dinamicamente carregados no ambiente, estendendo as capacidades da ferramenta.

Cada *plugin* Eclipse é definido em um projeto Eclipse distinto. A primeira tarefa é unificar os projetos, já que a ferramenta Meissa foi concebida para analisar o código-fonte de um projeto por vez.

A seguir, as mesmas etapas tomadas durante o estudo de caso do projeto MobileMedia serão executadas. Isto é, a recuperação manual da arquitetura seguida da recuperação automatizada através da ferramenta e subsequente comparação arquitetural. Com isso, serão identificadas qualquer falhas na recuperação arquitetural da arquitetura.

A próxima tarefa será a conversão do modelo arquitetural de especificação da ferramenta, discutido na Seção 4, em um modelo arquitetural no ambiente Bellatrix. Com isso, será possível a comparação entre o modelo recuperado e o modelo especificado. A partir da análise dos resultados da comparação, serão ponderadas as diferenças arquiteturais encontradas e então serão aplicadas as adequações necessárias a fim de corrigi-las.

Durante a execução do estudo de caso, serão referenciados diagramas apresentados na Seção 4, na qual a implementação da ferramenta é descrita.

5.3.3 Execução do estudo de caso 2

Após a unificação dos projetos, foi executada a recuperação da arquitetura no código-fonte da ferramenta. Como discutido no estudo de caso do MobileMedia, o verificador do modelo de componente COSMOS* é executado antes da recuperação em si, exibindo os resultados disponibilizados na tabela 5.5.

[5] A fins de comparação, o projeto MobileMedia possui cerca de 7 mil linhas de código.

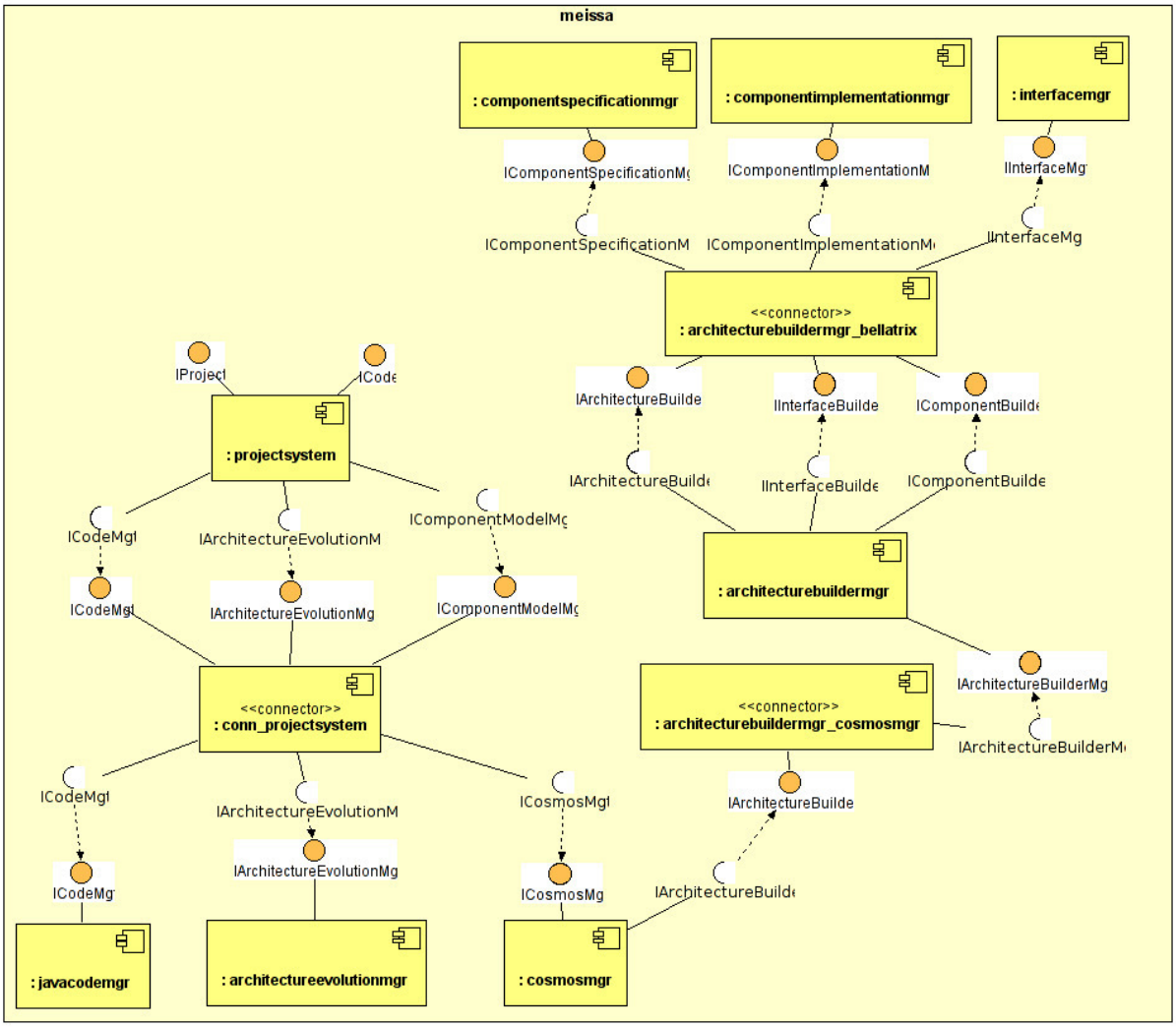


Figura 5.9: Modelo arquitetural do projeto Meissa recuperado através da ferramenta.

Tipo	Problema	Ocorrência	Frequência
Implementação	Elemento não declarado	3	1,42%
Implementação	Nomenclatura	74	35,07%
Implementação	Visibilidade	22	10,43%
Meissa	Erro duplicado	109	51,66%
Meissa	Limitação	3	1,42%

Tabela 5.5: Relatório de erros COSMOS* para o código-fonte da ferramenta Meissa.

Tipo de erro	Ocorrência	Frequência
Meissa	112	53,08%
Implementação	99	46,92%
Total	211	100%

Tabela 5.6: Resumo das proporções dos tipos de erros descritos na tabela 5.5.

A recuperação manual da arquitetura foi feita seguindo a mesma abordagem realizada no estudo de caso 1 (MobileMedia). Após obtido o modelo recuperado, a comparação entre os modelos manual e automaticamente recuperados foi realizada. O resultado obtido indica que a recuperação automatizada foi corretamente efetuada, apontando diferenças apenas nas interfaces, uma vez que tais não tiveram suas operações representadas no modelo manualmente recuperado. Uma vez que o resultado da comparação não apresentou diferenças relevantes, o diagrama foi omitido.

O modelo arquitetural recuperado para o projeto Meissa é apresentado na Figura 5.9. Em comparação, o modelo arquitetural de especificação do projeto foi apresentado na Figura 4.9, no Capítulo 4. É possível ver na arquitetura recuperada cada uma cadas camadas: *ProjectSystem* e *Conn_ProjectSystem* formam a camada de sistema, *JavaCodeMgr*, *ArchitectureEvolutionMgr* e *CosmosMgr* formam a camada de negócios. O componente *ArchitectureBuilderMgr* também pertence a camada de sistema e é o responsável por interfacear os componentes de negócio com a plataforma subjacente que constrói os diagramas, que é o Bellatrix. Os componentes *InterfaceMgr*, *ComponentImplementationMgr* e *ComponentSpecificationMgr* são de fato os componentes da camada de negócio do ambiente Bellatrix, que são utilizados por Meissa para a construção dos diagramas arquiteturais.

Mesmo sem o auxílio da ferramenta, é fácil observar diferenças entre a arquitetura especifica e a implementada. Algumas diferenças são obvias, como o nome dado aos componentes. Por exemplo, *CodeMgr* no modelo de especificação é materializado pelo componente *JavaCodeMgr* no modelo recuperado. Outra diferença visível é a falta de conectores na arquitetura de especificação. A fim de não poluir o resultado da comparação entre tais dois modelos com tais discrepâncias, após importar o modelo de especificação no ambiente Bellatrix, os componentes foram renomeados de acordo com o modelo recupe-

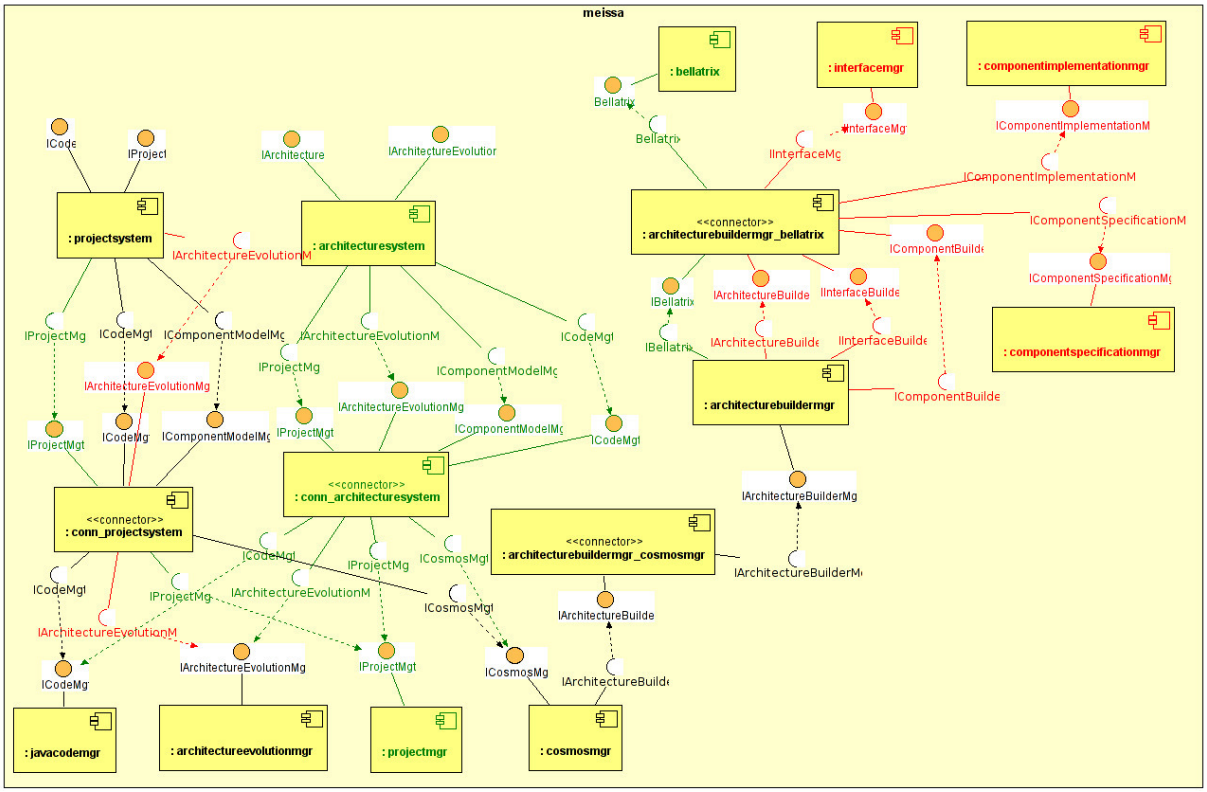


Figura 5.10: Resultado da comparação entre o modelo arquitetural recuperado e o modelo arquitetura originalmente especificado para a ferramenta Meissa.

rado e os conectores utilizados na implementação foram adotados. A Figura 5.10 mostra o resultado da comparação entre a arquitetura recuperada e a arquitetura especificada.

Observando o resultado da comparação, os elementos destacados em vermelho são aqueles presentes na arquitetura implementada (recuperada), mas que não foram especificados. Os elementos em verde são os especificados, mas não implementados. Os elementos em preto existem em ambos os modelos de especificação e implementação.

Os elementos em vermelho no canto superior direito do diagrama estão relacionados aos componentes do ambiente Bellatrix utilizados pela ferramenta. Durante a especificação, o uso de tais componentes foi modelado como apenas o uso de um componente genérico *Bellatrix* e sua interface *IBellatrix*. Entretanto, durante a implementação de fato necessitou-se estabelecer quais eram os componentes do ambiente que seriam reutilizados e quais são suas interfaces. Assim, em relação aos componentes do ambiente Bellatrix, a arquitetura implementada deve ser levada em consideração e a arquitetura especificada deve ser substituída.

Ainda analisando as diferenças arquiteturais, percebe-se que, na camada de negócio, o componente *ProjectMgr* foi especificado, mas não implementado. Após breve análise do

código, fica evidente que as responsabilidades do componente de negócio *ProjectMgr*, foram de fato implementadas na camada de sistema no componente *ProjectSystem*. Devido a isso, a arquitetura implementada não necessitou expor no componente *ProjectSystem* a interface requerida *IProjectMgt*, uma vez que todas as operações relacionadas a projeto foram implementadas em tal componente. Essa diferença arquitetural deve ser corrigida, por se tratar de um real desvio entre a arquitetura especificada e a arquitetura implementada. O código no componente *ProjectSystem* deve ser refatorado a fim de criar a implementação do componente *ProjectMgr*, na camada de negócios, e com ela, a interface requerida *IProjectMgt* no componente *ProjectSystem*.

A terceira diferença arquitetural a ser observada é a existência do componente *ArchitectureSystem* e, por consequência, do conector *Conn_ArchitectureSystem*. A razão desta diferença é a mesma de antes: as responsabilidades que seriam delegadas ao componente *ArchitectureSystem* foram implementadas no *ProjectSystem* e por isso tal componente foi suprimido da arquitetura implementada. A fim de separar as responsabilidades relacionadas ao gerenciamento do projeto e da arquitetura, decidiu-se que o componente *ArchitectureSystem* deve ser implementado de acordo com a arquitetura especificada.

Por fim, deve-se notar que, no modelo arquitetural recuperado, o componente da camada de sistema *ProjectSystem* possui duas interfaces providas que não possuem conexão com nenhum outro componente. Isso se deve ao fato de tais interfaces serem consumidas por *plugins* Eclipse que controlam a interface gráfica e constituem os editores da ferramenta, e por limitação do ambiente, não foram implementados utilizando o modelo COSMOS*. Idealmente, tais interfaces deveriam ser providas pelo super-componente *Meissa* e a elas devem ser relacionadas as interfaces da camada de sistema. Tal mudança será realizada no diagrama arquitetural da arquitetura implementada e submetida a geração de código, juntamente com as demais adequações arquiteturais anteriormente mencionadas.

O modelo arquitetural da implementação foi alterado a fim de adequar a arquitetura com base nas decisões tomadas após ponderadas as diferenças arquiteturais encontradas. O modelo alterado foi submetido a geração de código e então uma nova recuperação arquitetural foi efetuada sobre o código-fonte gerado. A Figura 5.11 contém o resultado da recuperação arquitetural.

Durante a execução deste estudo de caso, percebeu-se uma excessiva diferença no tempo necessário para realizar a recuperação da arquitetura para o código-fonte do projeto Meissa. A fim de analisar como se distribui o tempo de execução da atividade de recuperação da arquitetura, foi introduzido um cronômetro na operação da camada de sistema que inicia o processo de recuperação a fim de medir o tempo total de execução. A recuperação é implementada através de duas sub-atividades principais: (i) a extração do modelo KDM a partir do código-fonte e (ii) a conversão do modelo KDM em um modelo arquitetural através do modelo de implementação de componentes utilizado. Para cada

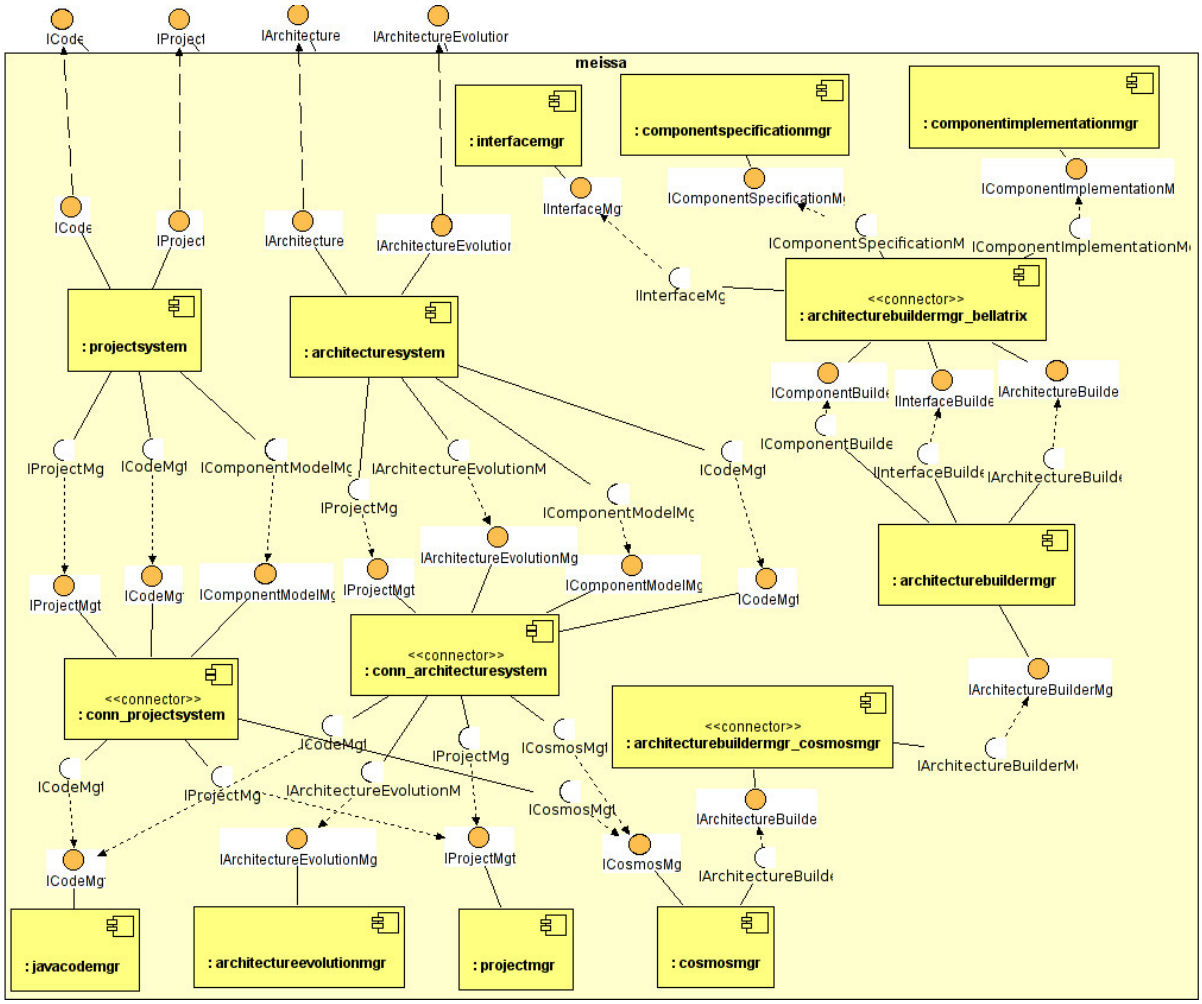


Figura 5.11: Arquitetura recuperada através da ferramenta após a geração de código.

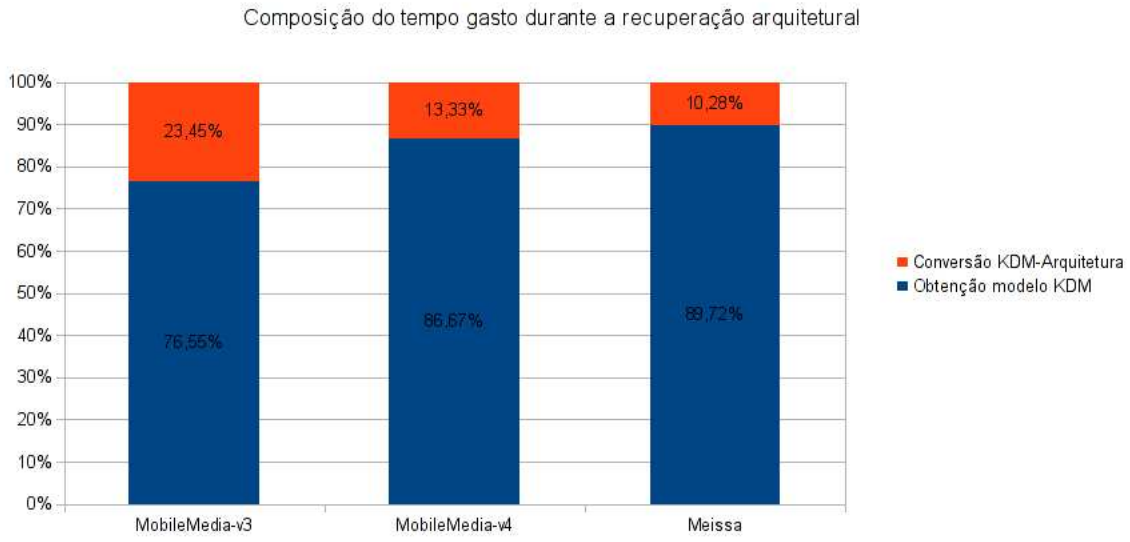


Figura 5.12: Medidas de tempo para a recuperação arquitetural do código-fonte do projeto Meissa e MobileMedia.

uma dessas sub-atividades, também foi adicionado um cronômetro a fim de mensurá-las.

Foram obtidas métricas para a recuperação do código-fonte da ferramenta Meissa e para a terceira e quarta versões do projeto MobileMedia. Os resultados são apresentados no gráfico da imagem 5.12, construído a partir dos resultados apresentados na tabela 5.7.

Análise dos resultados do estudo de caso 2

Este estudo de caso é extremamente relevante para a solidificação da confiança na solução proposta. Não apenas pela curiosa situação de uma ferramenta de recuperação arquitetural poder recuperar a sua própria arquitetura e então manipulá-la, mas por sua escala de tamanho. Meissa pode ser considerado um projeto de software inicial na escala de grande porte, não só pelo número de linhas de código, classes ou componentes, mas por sua relação com componentes de terceiros e a extensão da sua interface gráfica. A aplicação da solução proposta em um projeto dessas dimensões e a obtenção de resultados positivos, como os encontrados durante a execução do estudo de caso, mostram a viabilidade da solução em projetos de larga escala.

Entretanto existem pontos a serem melhorados na solução proposta. A tabela 5.7 mostra os tempos de execução para a recuperação arquitetural de três projetos de software. É clara a correlação entre a complexidade do projeto e o tempo necessário para a recuperação do modelo KDM. Tal atividade ocorre em várias etapas e é executada pela biblioteca *MoDisco*. Inicialmente o código-fonte necessita ser analisado e convertido em

Métrica	MobileMedia-v3	MobileMedia-v4	Meissa
Número de componentes	15	18	16
Linhas de código	6400	7155	30933
Geração modelo KDM (s)	14,00 (76,55%)	17,33 (86,67%)	151,00 (89,72%)
Conversão KDM-Arquitetura (s)	4,33 (23,45%)	2,66 (13,33%)	17,33 (10,28%)
Recuperação arquitetura (s)	18,33 (100%)	20,00 (100%)	168,33 (100%)

Tabela 5.7: Tempos de execução para a recuperação da arquitetura.

um modelo que representa sua estrutura sintática, ou seja, convertido em cada elemento da linguagem que está sendo utilizado (classes, interfaces, variáveis, chamada de métodos, parâmetros, etc.) e quais são seus relacionamentos. Após esse modelo ser obtido, ele é convertido em um modelo KDM, que como discutido na Seção 4.5.2, é independente de linguagem e plataforma. Tanto o modelo representando a análise sintática Java e o modelo KDM têm seu tamanho proporcional ao número de artefatos de implementação, ou seja, quanto maior e mais complexo o projeto, maior será o modelo KDM que o representa e que precisa ser obtido. Isso explica porque o tempo de recuperação de tais modelos aumenta com o tamanho do projeto. Por sua vez, a recuperação arquitetural é baseada no modelo KDM. Quanto maior for tal modelo, mais elementos precisam ser considerados ao tentar identificar-se componentes, conectores, interfaces e sua interligações. Desta forma, o tempo de recuperação arquitetural também é proporcional ao tamanho do projeto de software sob análise.

Como facilmente visto na Figura 5.12, o tempo gasto na recuperação do modelo KDM compõe a maior parte do tempo total gasto na recuperação arquitetural. A fim de reduzir o tempo gasto nesta atividade, a melhoria no tempo de obtenção do modelo KDM pode ser atingida através da recuperação incremental do modelo, ou seja, recuperar apenas partes do modelo para os artefatos de implementação (ou arquivos) que foram alterados desde a última recuperação. Assim, em vez de ser necessário processar todo o código-fonte, pode-se focar apenas nas partes que foram alteradas desde a última recuperação. Isso não melhorará o tempo gasto na primeira recuperação, mas certamente reduzirá o tempo total gasto nas iterações seguintes.

Por fim, é interessante notar como em um projeto especificado e desenvolvido pela mesma pessoa, ao longo de pouco mais de um ano e meio, teve tanto falhas de confor-

midade com o modelo de implementação de componentes COSMOS* como problemas de erosão arquitetural, ainda que extremo cuidado tenha sido tomado para que tais eventos não acontecessem. Isso leva a crer que a chance de se encontrar problemas de erosão arquitetural em grandes projetos de computação são, de fato, significantes; especialmente na indústria, onde o foco de desenvolvimento volta-se muito mais a entregar os projetos dentro das restrições de tempo e orçamento, com requisitos não-funcionais, como desempenho e segurança, muitas vezes a frente de outros como manutenibilidade e reusabilidade.

5.4 Resumo

Este capítulo dedicou-se ao estudo de casos, nos quais a solução Meissa foi aplicada, considerando dois sistemas de software distintos. O primeiro deles é o MobileMedia, uma linha de produtos com 7 diferentes versões originalmente criada pela Universidade da Colúmbia Britânica [61]. O segundo é a própria ferramenta Meissa, com suas mais de 30 mil linhas de código.

O estudo de caso 1 (MobileMedia), exercitou o cenário principal esperado para o uso da solução Meissa, que é a análise da evolução da arquitetura de um sistema de software. Foi analisada a evolução da arquitetura relacionada a transição da terceira para a quarta versão da linha de produtos MobileMedia. Após a análise, foram encontrados problemas de evolução arquitetural que permitiram exercitar as capacidades de adequação e geração de código da ferramenta.

O estudo de caso 2 (ferramenta Meissa), avaliou outro cenário comum para a aplicação da solução Meissa: o desenvolvimento inicial de um projeto. Nessa situação, existirá um modelo arquitetural do sistema especificado pelo arquiteto de software. Tal modelo guiará os engenheiros de software durante o projeto detalhado do software e espera-se que o código-fonte implementado realize a materialização da arquitetura inicialmente elaborada pelo arquiteto. Com a solução Meissa, foi possível recuperar a arquitetura implementada pelo código-fonte da ferramenta Meissa e compará-la com a arquitetura especificada no início do projeto (Capítulo 4). Nesse estudo de caso, também foram encontrados problemas de erosão arquitetural, que foram sanados utilizando-se o suporte de adequação e geração de código-fonte da ferramenta Meissa.

Com a analise dos resultados dos estudos de caso, foi possível identificar que a solução Meissa desempenhou como esperado, encontrando, de fato, problemas de envelhecimento de software nas arquiteturas dos sistemas estudados. Entretanto, também foram encontrados problemas na solução, que precisam ser corrigidos. Entre eles, foi identificado que o tempo necessário para a recuperação da arquitetura pode torna-se inconveniente (alguns minutos) para grandes projetos de software, ainda que tolerável, e pode ser combatido com a recuperação incremental da arquitetura. Os problemas encontrados podem ser

classificados como melhorias desejáveis, mas que não impactam diretamente no uso ou corretude da solução Meissa.

O capítulo seguinte propõe-se a descrever as possibilidades de trabalhos futuros baseando-se no que foi produzido neste projeto de pesquisa. Também é realizada a análise final deste trabalho, ponderando as contribuições deste trabalho.

Capítulo 6

Conclusões e trabalhos futuros

Neste capítulo, este documento é encerrado apresentando as conclusões, as contribuições e os trabalhos futuros.

6.1 Conclusões

O fenômeno da redução da vida útil do software é conhecido por diversos termos: (i) erosão arquitetural [46], (ii) desvio arquitetural[1] [46] e (iii) envelhecimento de software [45]. Argumenta-se que seria possível evitar a erosão arquitetural caso fosse utilizada uma abordagem como a engenharia dirigida por modelos [63]. Entretanto não é possível, na prática, impedir alterações diretas ao código-fonte [59, 60]. Entre as principais razões destacam-se restrições de orçamento e tempo, além da falta de suporte de ferramentas que auxiliem no processo de transformação dos modelos para código-fonte [14].

Este trabalho propõe uma solução composta por método e ferramenta, fundados nos conceitos de engenharia dirigida por modelos. A solução permite ao arquiteto de software identificar problemas de envelhecimento de software, como a erosão e desvios arquiteturais, e agir de maneira pró-ativa sobre eles, prevenindo o excessivo acúmulo de problemas arquiteturais que tendem a reduzir a vida útil do software. Através das capacidades de recuperação da arquitetura implementada, comparação de modelos arquiteturais e geração automática de código, totalmente independente do domínio, a solução proposta é capaz de atuar sobre grandes sistemas de software baseado em componentes.

Descrito no Capítulo 3 desta dissertação, o método Meissa é composto por 6 diferentes atividades: recuperação, comparação, adequação, mapeamento, implementação e versionamento. Dado um sistema baseado em componentes e implementado utilizando-se um modelo de implementação de componentes, como COSMOS*, através da execução

[1] Do inglês, *architectural drift*.

do método, o arquiteto é guiado através de cada uma das atividades, recuperando a arquitetura implementada do sistema, analisando-a em busca de problemas arquiteturais e aplicando as mudanças necessárias para adequar a arquitetura do sistema.

Auxiliando a execução do método, a ferramenta completa a solução proposta, como descrito no Capítulo 4. Ela foi desenvolvida para ser utilizada juntamente com outras ferramentas para sistemas baseados em componentes, como a Bellatrix [58], no popular ambiente de desenvolvimento integrado Eclipse [23]. Sua principal capacidade está na automação das atividades do método, retirando do arquiteto o trabalho repetitivo, extenuante e propenso a erros, devido a sua complexa natureza. Estando presente durante toda a execução do método, a ferramenta permite a recuperação da arquitetura implementada baseando-se no código-fonte do projeto sob análise e permite também a comparação de modelos arquiteturais, facilitando a análise de problemas arquiteturais pelo arquiteto. Ainda, ela pode ser utilizada pelo arquiteto para materializar correções na arquitetura implementada, gerando código-fonte e alterando o projeto sob análise de forma a implementar as mudanças arquiteturais desejadas. Por fim, a ferramenta mantém o histórico das arquiteturas recuperadas e adequadas, permitindo ao arquiteto comparar as evoluções arquiteturais do sistema, como recomendado pelo método.

Ainda que não totalmente automatizável, a solução proposta reduz extremamente o esforço humano nas atividades mais trabalhosas e complexas: a recuperação arquitetural, a comparação de modelos e a geração de código. Certamente, como dita o método, é necessária a intervenção humana na fase de implementação. Isso é requerido pelo fato da solução ater-se a estrutura arquitetural do projeto e não a implementação do domínio. Com isso é permitido que a solução seja agnóstica em relação ao domínio, mas, por outro lado, necessita a intervenção humana para a implementação da lógica relacionada ao domínio do projeto.

Os estudos de casos do Capítulo 5 demonstram como a solução é aplicada em dois distintos projetos: (i) o MobileMedia, um aplicativo construído através de linhas de produto de software para telefones móveis [15, 61], e (ii) a ferramenta implementada neste projeto. O primeiro estudo de caso mostra em detalhes a aplicação do método e uso da ferramenta. Ele também trás a análise da arquitetura do MobileMedia, ponderando, como faria um arquiteto de tal produto, sobre os problemas de desvio e erosão arquiteturais e falhas de especificação encontrados. Seguindo o método, em tal estudo de caso, a arquitetura do projeto MobileMedia é alterada a fim de se corrigir os problemas de envelhecimento arquitetural detectados e verificar o comportamento correto da ferramenta implementada. Similar ao caso do MobileMedia, no segundo estudo é testada como a implementação da ferramenta comporta-se em um grande projeto de software. Foram identificados e discutidos problemas de desempenho e como seria possível tratá-los. Os pontos de melhorias identificados nos estudos de caso foram compilados em uma lista, juntamente com outras

propostas de trabalhos futuros, na Seção 6.3.

Ao fim da análise dos estudos de caso, percebe-se que os resultados obtidos são extremamente positivos, construindo a confiança necessária para a utilização da solução proposta em reais projetos, seja na indústria ou academia. A adição de novas funcionalidades, melhorias nas funcionalidades atuais ou investigação mais aprofundada em certos pontos, como os mencionados na seção de trabalhos futuros, aumentariam a atratividade da solução. Entretanto ela é capaz de ser utilizada como descrita neste trabalho e tem o potencial de aumentar a vida útil de sistemas de softwares de grande porte, baseados em componentes.

6.2 Contribuições

A principal contribuição deste projeto de pesquisa é, em si, a solução proposta discutida ao longo desta dissertação. A discussão do método que a compõe, ocorrida no Capítulo 3, levanta em detalhes problemas relacionados a recuperação da arquitetura, comparação de modelos e adequação da arquitetura, elicitando possíveis soluções e ponderando suas fraquezas e qualidades. É dedicada extensa porção do mesmo capítulo para descrever as possibilidades de automação das abordagens ali discutidas. A solução proposta por este trabalho, também é composta pela ferramenta descrita no Capítulo 4. A ferramenta implementa os conceitos e soluções apresentados pelo método e é fundamental para o pragmatismo e aplicabilidade da solução Meissa em grandes projetos de software.

A ferramenta foi concebida como a infraestrutura para que terceiros possam adicionar novos componentes que implementem a análise de novas linguagens de programação e modelos de implementação de componentes. Com isso, as contribuições deste trabalho estendem-se a mais do que uma solução de gerenciamento arquitetural para projetos escritos em Java e que utilizem o modelo COSMOS*, mas sim a qualquer projeto, que utilize qualquer linguagem de programação e qualquer modelo de implementação de componentes, bastando que estejam disponíveis os componentes para tal linguagem e modelo de componentes escolhidos.

Em menor nível, este trabalho contribuiu na melhoria do ambiente Bellatrix, com correção de falhas e aprimoramento nos editores de modelos arquiteturais. Também foi implementado um verificador de conformidade da implementação com o modelo de componentes COSMOS*.

6.3 Trabalhos futuros

Essa seção discute pontos de melhorias ou novas ideias relacionadas a este trabalho.

- Existe uma limitação imposta pelo editor gráfico de diagramas arquiteturais do ambiente Bellatrix. Ele não permite que, em um componente composto, mais de uma interface interna [2] ao componente seja mapeada a uma mesma interface externa [3]. Quando a ferramenta Meissa recupera arquiteturas em que existe tal situação, isso resulta na visualização incorreta dos mapeamentos para a interface em questão. Ainda que isso não cause nenhum problema além da visualização gráfica, a resolução dessa falha é desejável.

- A ferramenta Meissa permite a ligação dinâmica de componentes que implementem o suporte a diferentes linguagens de programação e a diferentes modelos de implementação de componentes. Neste trabalho foram examinadas a linguagem Java e o modelo de implementação COSMOS*. A implementação de componentes para suportar outras linguagens de programação e modelos de componentes põe-se como um relevante trabalho futuro, sobre a qual interessantes estudos de casos podem ser feitos, como, por exemplo, um único modelo arquitetural pode gerar diversas implementações da arquitetura em diferentes linguagens de programação, ou, até mesmo, a conversão de um projeto de um dado modelo de implementação de componentes para um segundo.

- Uma possível implementação de componentes para a ferramenta Meissa que suporte novos modelos de implementação de componentes pode ser feita através de abordagens similares ao uso da linguagem ATL, como discutido na Seção 4.5.2. Isso permitiria expor uma linguagem de transformação de modelos que o arquiteto poderia usar para definir seu próprio modelo de implementação de componentes e assim, utilizar toda a infraestrutura já provida pela ferramenta.

- De maneira similar ao item anterior, uma interessante ideia é expor um editor gráfico de metamodelos ao ambiente Bellatrix. Com tal editor seria possível permitir ao arquiteto construir o metamodelo que represente o modelo de implementação de componentes. Com isso, o arquiteto teria extrema facilidade para definir o modelo de implementação de componentes que melhor se adeque ao projeto.

- Como discutido na Seção 5.2.4, conforme o número de componentes no diagrama arquitetural cresce, mais complexa tende a ser a visualização das diferenças entre dois modelos arquiteturais. Por um lado, a visualização gráfica ajuda na compreensão global das mudanças, por outro, uma visão em árvore, como a implementada pela ferramenta Meissa, restringe o fácil acesso ao resultado, já que possui um foco muito localizado nas mudanças de um elemento específico. O estudo de possíveis outras

[2] Neste contexto, o termo interface interna refere-se as interfaces dos subcomponentes.
[3] Interface exposta pelo componente composto.

formas de apresentação do resultado para o arquiteto irá contribuir para facilitar ainda mais a aplicação do método e reduzir a complexidade das tarefas que necessitam ser manualmente executadas sobre o modelo, como, por exemplo, a atividade de Adequação, do método Meissa.

- A fim de reduzir o tempo gasto na atividade de recuperação arquitetural, melhorias no tempo de obtenção do modelo KDM irão reduzir drasticamente o tempo total requerido para a recuperação da arquitetura. Afim de melhorar o tempo total necessário, uma possibilidade é a recuperação incremental do modelo KDM, ou seja, recuperar apenas partes do modelo para os artefatos de implementação (ou arquivos) que foram alterados desde a última recuperação. Assim, em vez de ser necessário processar todo o código-fonte, pode-se focar apenas nas partes que foram alteradas desde a última recuperação. Isso não melhorará o tempo gasto na primeira recuperação, mas certamente reduzirá o tempo total gasto nas iterações seguintes.

- Como mencionado na Seção 5.2.4, existe certa dificuldade intrínseca à atividade de Adequação, no método Meissa. Uma vez que o arquiteto, muito provavelmente, não participou intensivamente da atividade de desenvolvimento do sistema, ele talvez não possua as informações necessárias para inferir a razão por trás de cada uma das diferenças arquiteturais encontradas entre o modelo de especificação e o recuperado a partir do código-fonte. Sem conhecer a motivação por trás da mudança, torna-se difícil decidir como ela deve ser lidada. Certamente questionar os desenvolvedores é uma possível abordagem, outra é a leitura do código-fonte. Entretanto seriam essas as melhores abordagens? Existiriam outras possibilidades ou meios que o arquiteto poderia utilizar-se para obter mais informações sobre as diferenças arquiteturais?

- Uma ideia interessante seria mesclar a abordagem utilizada neste trabalho para combater a erosão arquitetural e auxiliar o arquiteto de software durante a evolução da arquitetura com as propostas em Lytra et al. [37], em relação à manutenção das decisões arquiteturais tomadas durante a evolução da arquitetura. Suponha que se tenha em mãos diversas versões de um sistema legado que utilize um modelo de implementação de componentes. Seria possível recuperar a arquitetura de cada uma dessas versões e compará-las contra um banco de dados de decisões arquiteturais conhecidas. Desta forma, existiria a possibilidade de inferir quais foram as decisões arquiteturais tomadas em cada uma das evoluções do sistema legado, aumentando ainda mais o entendimento do mesmo.

Referências Bibliográficas

[1] Eclipse modeling framework - emf web site. http://www.eclipse.org/modeling/emf.

[2] Acceleo. Acceleo web site. http://www.acceleo.org.

[3] Jonathan Aldrich, Criag Chambers, and David Notkin. Archjava web site. http://www.archjava.org.

[4] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd edition, 2003.

[5] ATL. Atl transformation language web site. http://www.eclipse.org/atl/.

[6] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume ii: Technical concepts of component-based software engineering. In *Technical Report CMU/SEI-2000-TR-008*, Software Engineering Institute at Carnegie-Mellon University, April 2000.

[7] Rami Bahsoon and Wolfgang Emmerich. Architectural stability. In *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*, pages 304–315. Springer, 2009.

[8] Jean Bézivin. In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, 5(2):21–24, 2004.

[9] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[10] Alan W. Brown. *Large-Scale, Component Based Development*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[11] Hugo Bruneliere. How to deal with your it legacy? reverse engineering using models: Modisco in a nutshell. In *JavaTech Journal*, 10, pages 21–24, 2011.

[12] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings*

of the IEEE/ACM international conference on Automated software engineering, ASE '10, pages 173–174, New York, NY, USA, 2010. ACM.

[13] John Cheesman and John Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[14] Feng Chen, Qianxiang Wang, Hong Mei, and Fuqing Yang. An architecture-based approach for component-oriented development. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, COMPSAC '02, pages 450–458, Washington, DC, USA, 2002. IEEE Computer Society.

[15] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley Boston, 2002.

[16] Marcelo de Oliveria Dias. Projeto e implementação de variabilidades em arquiteturas baseadas no modelo de componentes cosmos*. In *Universidade Estadual de Campinas, Instituto de Computação*, 2010.

[17] Marcelo Dias, Leonardo Tizzei, Cecília M. F. Rubira, Alessandro Garcia, and Jaejoon Lee. Leveraging aspect-connectors to improve stability of product line variabilities. In *Fourth International Workshop on Variability Modelling of Software-intensive Systems*, pages 21–28, Linz, Austria, January 2010.

[18] Claire Dimech and Dharini Balasubramaniam. Maintaining architectural conformance during software development: A practical approach. In Khalil Drira, editor, *Software Architecture*, volume 7957 of *Lecture Notes in Computer Science*, pages 208–223. Springer Berlin Heidelberg, 2013.

[19] Cristina Gómez Elena Planas, Jordi Cabot. Two basic correctness properties for atl transformations: Executability and coverage. In *Proceedings of the 3rd International Workshop on Model Transformation with ATL*, MtATL 2011, pages 1–9, 2011.

[20] Jean-Marie Favre. Towards a basic theory to model model driven engineering. In *3rd Workshop in Software Model Engineering, WiSME*. Citeseer, 2004.

[21] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 261–270, New York, NY, USA, 2008. ACM.

[22] Fernando Castor Filho, Nelio Cacho, Eduardo Figueiredo, Raquel Maranhão, Alessandro Garcia, and Cecília Mary F. Rubira. Exceptions and aspects: the devil is in the details. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 152–162, New York, NY, USA, 2006. ACM.

[23] The Eclipse Foundation. Eclipse web site. http://www.eclipse.org.

[24] Cristina Gacek, Ahmed Abd-allah, Bradford Clark, and Barry Boehm. On the definition of software system architecture. Technical report, Center for Software Engineering, University of Southern California, Los Angeles, CA, USA, April 1995.

[25] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[26] Leonel Aguilar Gayard, Cecília Mary Fischer Rubira, and Paulo Asterio de Castro Guerra. Cosmos*: a component system model for software architectures. Technical report, Institute of Computing, University of Campinas, Campinas, SP, Brazil, February 2008.

[27] OMG Object Management Group. Mda: Model driven architecture web site. http://www.omg.org/mda.

[28] The Object Management Group. Knowledge discovery model. http://www.omg.org/technology/kdm/index.htm.

[29] The Object Management Group. Architecture-driven modernization (adm): Knowledge discovery meta-model (kdm), v1.0. *OMG Document Number: formal/08-01-01*, 2008.

[30] The Object Management Group. Mof model to text transformation language, v1.0. *OMG Document Number: formal/2008-01-16*, 2008.

[31] Change Vision Inc. Web site da ferramenta de modelagem jude. http://http://jude.change-vision.com/jude-web/index.html.

[32] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 109–120, 2005.

[33] Stéphane Lacrampe Jonathan Musset, Étienne Juliot. Acceleo architect guide 2.6. http://www.acceleo.org/doc/obeo/en/acceleo-2.6-architect-tutorial.pdf.

[34] Stéphane Lacrampe Jonathan Musset, Étienne Juliot. Acceleo user guide 2.6. http://www.acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf.

[35] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language mola. In *in: Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004*, pages 14–28, 2004.

[36] Anthony J Lattanze. The architecture centric development method. *Thecnical report CMU-ISRI-05-103. School of Computer Science. Carnegie Mellon*, 2005.

[37] Ioanna Lytra, Huy Tran, and Uwe Zdun. Supporting consistency between architectural design decisions and component models through reusable architectural knowledge transformations. In *European Conference on Software Architecture (ECSA)*, LNCS 7957, pages 224–239, Heidelberg, July 2013. Springer.

[38] Stephen J Mellor. *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional, 2004.

[39] Microsoft. Microsoft .net web site. http://www.microsoft.com/net.

[40] Russ Miles and Kim Hamilton. *Learning UML 2.0*. O'Reilly Media, Inc., 2006.

[41] MoDisco. Modisco web site. http://www.eclipse.org/MoDisco.

[42] Thiago César Moronte. Uma infra-estrutura de software para apoiar a construção de arquiteturas de software baseadas em componentes. Dissertação de mestrado. In *Universidade Estadual de Campinas, Instituto de Computação*, 2007.

[43] Robert L Nord and James E Tomayko. Software architecture-centric methods and agile development. *Software, IEEE*, 23(2):47–53, 2006.

[44] Oracle. J2ee web site. http://www.oracle.com/technetwork/java/javaee/overview.

[45] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Sorrento, Italy, 1994. IEEE Computer Society Press.

[46] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17:40–52, October 1992.

[47] Ghulam Rasool and Nadim Asif. Software architecture recovery. *International Journal of Science, Engineering and Technology*, 23:434–439, 2007.

[48] Claudio Riva, Petri Selonen, Tarja Systä, and Jianli Xu. A profile-based approach for maintaining software architecture: an industrial experience report. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(1):3–20, 2011.

[49] Ansgar Schleicher and Bernhard Westfechtel. Beyond stereotyping: Metamodeling approaches for the uml. In *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2001.

[50] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39:25–31, 2006.

[51] Ian Sommerville. *Software Engineering: (8th Edition)*. Addison Wesley, 8 edition, June 2006.

[52] Davor Svetinovic and Michael Godfrey. A lightweight architecture recovery process. In *SWARM - SoftWare Architecture Recovery and Modelling*, WCRE '01, Stuttgart, Baden-Württemberg, Germany, October 2001.

[53] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[54] Leonardo P. Tizzei. Projeto mobilemedia-cosmos em github. https://github.com/leotizzei.

[55] Leonardo P. Tizzei. Uma infra-estrutura de suporte à evolução para repositórios de componentes. Dissertação de mestrado. In *Universidade Estadual de Campinas, Instituto de Computação*, 2007.

[56] Leonardo P. Tizzei, Paulo A. Guerra, and Cecília M. F. Rubira. Uma abordagem sistemática para reutilização e versionamento de componentes de software. In *Workshop de Manutenção de Software Moderna - VI SBQS*, WMSWM '07, Porto de Galinhas, PE, Brazil, June 2007.

[57] Leonardo Pondian Tizzei. *Evolução de Arquiteturas de Linhas de Produtos baseadas em Componentes e Aspectos*. PhD thesis, Institute of Computing, University of Campinas, Campinas, Brazil, July 2012. in Portuguese.

[58] Rodrigo Teruo Tomita. Bellatrix : um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes. Dissertação de mestrado. In *Universidade Estadual de Campinas, Instituto de Computação*, 2006.

[59] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61:105–119, March 2002.

[60] Jilles van Gurp, Jan Bosch, and Sjaak Brinkkemper. Design erosion in evolving software products. In *International workshop on the Evolution of Large-scale Industrial Software Applications*, ELISA '03, pages 134–139, Amsterdam, The Netherlands, September 2003.

[61] Trevor J. Young. Using aspectj to build a software product line for mobile devices. MSc dissertation. In *University of British Columbia, Department of Computer Science*, 2005.

[62] Yongjie Zheng. 1.x-way architecture-implementation mapping. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1118–1121, Waikiki, Honolulu, HI, USA, 2011. ACM.

[63] Yongjie Zheng and Richard N. Taylor. Taming changes with 1.x-way architecture-implementation mapping. In *26th IEEE/ACM International Conference On Automated Software Engineering*, ASE '11, pages 396–399, Lawrence, Kansas, USA, 2011.

Apêndice A

Requisitos de conformidade para COSMOS*

A.1 Modelo COSMOS*

A.1.1 Especificação

As informações contidas nessa seção foram obtidas em [26]. O modelo COSMOS* define as seguintes regras para um componente:

COMSPEC-1 O componente deve seguir uma estruturação em pacotes (UML), com nomes em minúsculo, como descrito a seguir:

- Pacote raiz: representa o componente
- Pacote *impl*: contém a implementação do componente
- Pacote *spec*: contém pacotes de interfaces providas e requeridas
- Pacote *spec.req*: contém interfaces UML requeridas pelo componente
- Pacote *spec.prov*: contém interfaces UML providas pelo componente

COMSPEC-2 A visibilidade dos elementos dentro de cada pacote deve ser:

- Pacote raiz: não aplicável [1]
- Pacote *impl*: pacote (com exceção da classe pública *ComponentFactory*)
- Pacote *spec*: não aplicável
- Pacote *spec.req*: pública
- Pacote *spec.prov*: pública

COMSPEC-3 O pacote *spec.prov* deverá conter:

- Uma interface *IManager* implementada de acordo com a interface especificada em *br.unicamp.ic.sed.cosmos.IManager*;

COMSPEC-4 O pacote *impl* deverá conter:

- Uma classe *Manager* que implemente a interface provida *spec.prov .IManager*;
- Se o componente for composto, a classe *Manager* deverá estender a classe abstrata *br.unicamp.ic.sed.cosmos.AManagerComposite*;
- Uma classe pública *ComponentFactory* que esteja de acordo com a implementação provida na seção A.1.1;

[1] Não há elementos de interesse, fora pacotes, para o modelo COSMOS* contidos neste pacote.

- Uma classe **Façade** para cada interface provida distinta de *spec.prov .IManager*;

COMSPEC-5 Em um **componente composto**, não deve haver exposição dos componentes internos para o cliente do componente, portanto:

- Cada interface requerida de cada componente interno que possua dependência externa ao componente composto deverá ser adaptada por uma classe **Adaptadora** no pacote *impl*, a uma interface requerida do componente externo.

O modelo COSMOS* define as seguintes regras para um conector:

CTORSPEC-1 O conector contém apenas um pacote (raiz), contendo:

- Uma classe pública ComponentFactory idêntica a definida para um componente.
- Uma classe com visibilidade de pacote Manager idêntica a definida para um componente.
- Uma classe com visibilidade de pacote, atuando como **Adaptador**, para cada par de interfaces a serem conectadas.

O modelo COSMOS* define as seguintes recomendações para componentes:

RECSPEC-1 O conector poderá conter um pacote de definição de tipos de dados (*datatypes*). Esse pacote deve estar contido no pacote *spec* e ser nomeado *datatypes*. Apenas interfaces são permitidas nesse pacote.

RECSPEC-2 O conector poderá conter um pacote de definição de exceções. Esse pacote deve estar contido no pacote *spec* e ser nomeado *exceptions*. Esse pacote deverá conter apenas definição de exceções.

Classe ComponentFactory

A classe ComponentFactory é responsável por criar uma instância do componente.

A classe deverá prover a implementação de um método estático, sem argumentos, nomeado `creatInstance` que retorne um objeto do tipo *br.ic.unicamp.sed.IManager*.

Apêndice B

Casos de uso para a ferramenta Meissa

B.1 Gerar código

B.1.1 Breve descrição

A ferramenta gera código para realizar as alterações feitas no modelo arquitetural.

B.1.2 Atores

- Arquiteto de software

B.1.3 Pré-condições

1. Um projeto foi previamente adicionado à ferramenta, contendo código-fonte associado
2. A arquitetura implementada foi previamente recuperada e adicionada ao modelo
3. As alterações da arquitetura foram feitas sobre a arquitetura implementada

B.1.4 Pós-condições

1. É gerado código para sincronizar a arquitetura implementada com o modelo arquitetural especificado.

B.1.5 Requisitos funcionais

- RF 9

B.1.6 Requisitos não funcionais

B.1.7 Fluxo básico

1. O arquiteto requisita à ferramenta a geração de código, informando o projeto e o conjunto de alterações.
2. A ferramenta verifica se as alterações da arquitetura foram feitas sobre um modelo arquitetural de mesma versão que o código-fonte.
3. A ferramenta gera código-fonte e o mescla com o código-fonte associado ao projeto.
4. Caso seja necessária a revisão manual do código gerado, a ferramenta informa a necessidade de revisão do código gerado.

B.1.8 Fluxo(s) alternativo(s)

- No passo 4, se já existir um modelo arquitetural recuperado no projeto, a ferramenta substitui o modelo anterior.

B.2 Versionar modelo arquitetural

B.2.1 Breve descrição

A ferramenta mantém informações históricas sobre o modelo em questão.

B.2.2 Atores

- Tempo

B.2.3 Pré-condições

1. O modelo a ser versionado deve estar adequadamente importado em algum projeto da ferramenta.

B.2.4 Pós-condições

1. A ferramenta armazena informações de versão sobre o modelo.

B.2.5 Requisitos funcionais

- RF 11

B.2.6 Requisitos não funcionais

B.2.7 Fluxo básico

1. A ferramenta verifica se o modelo a ser versionado possui alguma informação de versão anterior.
2. A ferramenta carrega as informações da versão anterior e cria nova versão com as mudanças ocorridas.
3. A ferramenta persiste a nova versão.

B.2.8 Fluxo(s) alternativo(s)

- No passo 1, se o modelo não possuir versões anteriores, é criada a primeira versão contendo todas as informações do modelo. Retorna-se ao passo 3.

B.3 Comparar arquiteturas

B.3.1 Breve descrição

A ferramenta compara dois modelos arquiteturais e indica as diferenças encontradas entre os modelos.

B.3.2 Atores

- Arquiteto de software

B.3.3 Pré-condições

1. Os modelos a serem comparados devem estar de acordo com a especificação de um modelo arquitetural suportada pelo ambiente Bellatrix.

B.3.4 Pós-condições

1. A ferramenta retorna as diferenças encontradas entre os modelos.

B.3.5 Requisitos funcionais

- RF 7
- RF 10

B.3.6 Requisitos não funcionais

- RNF 3
- RNF 5

B.3.7 Fluxo básico

1. O arquiteto requisita à ferramenta a comparação dos modelos arquiteturais.
2. A ferramenta requisita os arquivos contendo os respectivos modelos.
3. O arquiteto informa à ferramenta os arquivos dos modelos.
4. A ferramenta compara os modelos providos.

5. A ferramenta cria um visualizador gráfico, mostrando ao arquiteto as diferenças encontradas entre as arquiteturas.

B.3.8 Fluxo(s) alternativo(s)

B.4 Importar modelo arquitetural

B.4.1 Breve descrição

O arquiteto importa, para um determinado projeto, um modelo arquitetural.

B.4.2 Atores

- Arquiteto de software

B.4.3 Pré-condições

1. O projeto no qual o modelo será importado deve existir.

B.4.4 Pós-condições

1. O projeto reconhece o modelo importado.

B.4.5 Requisitos funcionais

- RF 6

B.4.6 Requisitos não funcionais

B.4.7 Fluxo básico

1. O arquiteto requisita à ferramenta a importação de um modelo arquitetural.
2. A ferramenta requisita o nome do projeto no qual o modelo será importado.
3. O arquiteto seleciona o projeto de uma lista de projetos existentes.
4. A ferramenta solicita o caminho para o arquivo de modelo.
5. O arquiteto informa o arquivo que contém o modelo.
6. A ferramenta verifica se o arquivo contém o modelo no formato especificado pelo ambiente Bellatrix.
7. A ferramenta importa o modelo no projeto.
8. `<<include>>` “Versionar modelo arquitetural”.

B.4.8 Fluxo(s) alternativo(s)

- No passo 6, se o modelo não estiver de acordo com o especificado pelo ambiente Bellatrix, a ferramenta emite uma mensagem e retorna para o passo 5.
- No passo 7, se o projeto já possuir um modelo importado, a ferramenta pergunta ao arquiteto se ele deseja sobrescrever o modelo importado.
  - Em caso negativo, termina-se a execução do caso de uso.
  - Em caso positivo, a ferramenta remove todas informações de versão sobre o modelo e retorna-se ao passo 8.

B.5 Visualizar modelo arquitetural

B.5.1 Breve descrição

A ferramenta permite a visualização gráfico de um modelo arquitetural.

B.5.2 Atores

- Arquiteto de software

B.5.3 Pré-condições

1. O modelo deve estar no formato especificado pelo ambiente Bellatrix.

B.5.4 Pós-condições

B.5.5 Requisitos funcionais

- RF 10

B.5.6 Requisitos não funcionais

B.5.7 Fluxo básico

1. O arquiteto requisita à ferramenta a visualização de um modelo arquitetural, informando o caminho do modelo à ferramenta.

2. A ferramenta cria um visualizador gráfico contendo os elementos do modelo.

B.6 Visualizar evolução arquitetural

B.6.1 Breve descrição

A ferramenta permite a visualização da evolução do modelo arquitetural com o tempo.

B.6.2 Atores

- Arquiteto de software

B.6.3 Pré-condições

1. O modelo deve estar no formato especificado pelo ambiente Bellatrix.
2. O modelo deve estar associado a um projeto da ferramenta.

B.6.4 Pós-condições

1. Não é realizada modificação no modelo.

B.6.5 Requisitos funcionais

- RF 10

B.6.6 Requisitos não funcionais

B.6.7 Fluxo básico

1. O arquiteto requisita à ferramenta a visualização da evolução de um modelo arquitetural.
2. A ferramenta exibe uma lista contendo as datas de modificação relacionadas àquele modelo e o número de alterações.
3. O arquiteto seleciona uma entrada da lista.
4. `<<include>>` “Visualizar modelo arquitetural” relacionado à versão do modelo selecionado na lista.

B.7 Alterar arquitetura

B.7.1 Breve descrição

A ferramenta permite a edição gráfica do modelo arquitetural.

B.7.2 Atores

- Arquiteto de software

B.7.3 Pré-condições

1. O modelo arquitetural deve estar associado a um projeto da ferramenta.

B.7.4 Pós-condições

1. O modelo arquitetural é alterado e salvo no projeto.

B.7.5 Requisitos funcionais

- RF 10
- RF 8

B.7.6 Requisitos não funcionais

B.7.7 Fluxo básico

1. O arquiteto abre um editor com o modelo arquitetural que deseja modificar.
2. O arquiteto pode:
   (a) Adicionar ou remover um componente
   (b) Adicionar ou remover interfaces de um componente
   (c) Atualizar interface de um componente (adicionar, remover ou alterar operações da interface)
   (d) Adicionar ou remover relacionamentos entre interfaces de componentes
3. O arquiteto informa o fim das alterações à ferramenta.
4. `<<include>>` “Comparar arquiteturas” entre o modelo arquitetural após a alteração e o modelo arquitetural anterior à alteração.

5. A ferramenta contabiliza as mudanças realizadas.
6. `<<include>>` “Gerar código” para implementar as mudanças realizadas.

B.8 Verificar implementação

B.8.1 Breve descrição

A ferramenta verifica se o código-fonte associado a um projeto está segue as regras do modelo de componentes definido no projeto.

B.8.2 Atores

- Arquiteto de software

B.8.3 Pré-condições

1. O código-fonte a ser validado deve estar previamente associado a um projeto da ferramenta.

B.8.4 Pós-condições

1. A ferramenta retorna informações sobre a conformidade do código-fonte com as regras do modelo de componentes definido no projeto.

B.8.5 Requisitos funcionais

- RF 2
- RF 1
- RF 2
- RF 3

B.8.6 Requisitos não funcionais

B.8.7 Fluxo básico

1. O arquiteto requisita à ferramenta verificação do código-fonte.
2. A ferramenta requisita o nome do projeto no qual o código-fonte será verificado.
3. O arquiteto seleciona o projeto de uma lista de projetos existentes.
4. A ferramenta obtém o caminho para o código-fonte.

5. A ferramenta verifica a aderência do código-fonte às regras do modelo de componentes definido no projeto.

6. A ferramenta cria uma lista com os problemas encontrados, mapeando-os às linhas de código problemáticas.

7. A ferramenta exibe a lista de erros, caso haja algum, para o arquiteto.

B.8.8 Fluxo(s) alternativo(s)

B.9 Importar código-fonte

B.9.1 Breve descrição

A ferramenta importa o código-fonte referente aos componentes da arquitetura a ser gerenciada.

B.9.2 Atores

- Arquiteto de software

B.9.3 Pré-condições

1. O projeto no qual o código-fonte será importado deverá existir.

B.9.4 Pós-condições

1. A ferramenta importa o código-fonte informado.

B.9.5 Requisitos funcionais

- RF 6

B.9.6 Requisitos não funcionais

B.9.7 Fluxo básico

1. O arquiteto requisita à ferramenta a importação do código-fonte.
2. A ferramenta requisita o nome do projeto no qual o código-fonte será importado.
3. O arquiteto seleciona o projeto de uma lista de projetos existentes.
4. A ferramenta solicita o caminho para o código-fonte.
5. O arquiteto informa o diretório do código-fonte.
6. `<<include>>` “Verificar implementação”.
7. A ferramenta importa o código-fonte no projeto.

B.9.8 Fluxo(s) alternativo(s)

- No passo 6, se o código-fonte não estiver implementado de acordo com o modelo de implementação de componentes, a ferramenta informa ao arquiteto que não foi possível importar o código-fonte. Termina-se a execução do caso de uso.

B.10 Criar novo projeto

B.10.1 Breve descrição

O arquiteto cria um novo projeto que inclui todas as informações necessárias para o uso da ferramenta.

B.10.2 Atores

- Arquiteto de software

B.10.3 Pré-condições

1. Não há

B.10.4 Pós-condições

1. Um novo projeto é criado.

B.10.5 Requisitos funcionais

- RF 6

B.10.6 Requisitos não funcionais

- RNF 5

B.10.7 Fluxo básico

1. O arquiteto requisita à ferramenta a criação de um novo projeto.
2. O sistema mostra ao arquiteto uma lista de linguagens de programação e modelos de componentes suportados pela ferramenta.
3. O arquiteto seleciona uma linguagem de programação e um modelo de componente para utilizar no projeto.
4. A ferramenta requisita os dados: nome do projeto, diretório de modelos do projeto, diretório do código-fonte do projeto
5. O arquiteto informa os dados.

6. `<<include>>` “Importar modelo arquitetural”.
7. `<<include>>` “Importar código-fonte”.
8. A ferramenta informa ao arquiteto que o projeto foi criado com sucesso.

B.10.8 Fluxo(s) alternativo(s)

- No passo 5, se o arquiteto não informar algum dos dados, então a ferramenta mostra uma mensagem de erro e retorna para o passo 4.

B.11 Recuperar arquitetura

B.11.1 Breve descrição

A ferramenta recupera um modelo arquitetural a partir do código-fonte fornecido.

B.11.2 Atores

- Arquiteto de software

B.11.3 Pré-condições

1. O código-fonte deve estar associado a um projeto da ferramenta.

B.11.4 Pós-condições

1. A ferramenta obtém um modelo arquitetural a partir do código-fonte fornecido.

B.11.5 Requisitos funcionais

- RF 4
- RF 5

B.11.6 Requisitos não funcionais

- RNF 3

B.11.7 Fluxo básico

1. O arquiteto requisita a recuperação da arquitetura informando: o projeto cujo código-fonte associado deverá ser recuperado.
2. A ferramenta obtém o código-fonte associado ao projeto.
3. `<<include>>` “Verificar implementação”
4. A ferramenta gera um modelo arquitetural baseado no código-fonte e o adiciona ao diretório de modelos do projeto.
5. `<<include>>` “Versionar modelo arquitetural”

B.11.8 Fluxo(s) alternativo(s)

- No passo 4, se já existir um modelo arquitetural recuperado no projeto, a ferramenta substitui o modelo anterior.