Guilherme Aren Marchetti

Um método de transformação de modelos UML para a inclusão de componentes de frameworks com o uso de planejador lógico

São Paulo

2012

Guilherme Aren Marchetti

Um método de transformação de modelos UML para a inclusão de componentes de frameworks com o uso de planejador lógico

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Mestre em Ciências

Área de concentração:
Engenharia Elétrica - Sistemas Digitais

Orientador:
Prof. Dr. Edson Satoshi Gomi

São Paulo

2012

FICHA CATALOGRÁFICA

Marchetti, Guilherme Aren
Método de transformação de modelos UML para inclusão de componentes de frameworks com uso de planejador lógico / G.A. Marchetti. -- São Paulo, 2012.
80 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Engenharia de software 2. UML 3. Frameworks (Modelos) I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II. t.

Resumo

Frameworks são ferramentas importantes no contexto atual da tecnologia de desenvolvimento de software, fornecendo funcionalidades através de código previamente testado, o que leva a uma redução na quantidade de código necessário para se implementar soluções e no tempo de implementação. Porém, devido a complexidade dos frameworks atuais, seu uso se torna trabalhoso, requerendo um gasto de tempo elevado para se aprender um framework novo. Neste trabalho de pesquisa foi desenvolvido o Framer, um método de transformação de modelos capaz de selecionar componentes de um framework, a partir de seu modelo UML, que são necessários para implementar as funcionalidades providas pelo framework para uma aplicação em desenvolvimento. Este método de transformação utiliza uma versão modificada do algoritmo NONLIN de planejamento, para identificar não somente os componentes que serão utilizados, mas como e em qual ordem devem ser invocados pela aplicação. Uma vez identificados os componentes, são criados os Diagramas UML necessários para representar sua utilização.

Palavras-Chave: Transformação de Modelos. Reuso de Software. Frameworks. Engenharia de Software.

Abstract

Frameworks are important tools for current software development technology, providing functionalities through previously tested code, reducing the amount of code and time necessary to implement the solution. However, due to framework complexity, a developer needs a significant investment in time to learn it. This work presents the Framer, a model transformation method capable of selecting components, from a framework's UML model, that will be used to construct the funcionalities of the target application. Once the components are identified, the UML Diagrams necessary to discribe its use are created.

Keywords: Model Transformation. Software Reuse. Frameworks. Software Engineering.

Conteúdo

1 Introdução p. 6

1.1 Objetivos . . . . . p. 8

1.2 Estrutura da Dissertação . . . . . p. 9

2 Modelos em UML p. 10

2.1 Frameworks . . . . . p. 14

3 Transformação de Modelos p. 18

4 Planejamento Lógico p. 22

5 Método Framer p. 26

5.1 Modelos de Entrada . . . . . p. 27

5.1.1 Modelo da Aplicação . . . . . p. 27

5.1.2 Modelo do Framework . . . . . p. 29

5.2 Conversão dos Modelos de Entrada para a forma lógica . . . . . p. 31

5.2.1 Modelo da Aplicação . . . . . p. 32

5.2.2 Modelo do Framework . . . . . p. 33

5.2.2.1 Diagramas de Classe . . . . . p. 33

5.2.2.2 Diagrama de Sequência . . . . . p. 36

5.3 Algoritmo de Planejamento . . . . . p. 38

5.4 Alteração do Modelo Inicial . . . . . p. 41

5.5 Implementação do Método de Transformação . . . . . p. 44

6 Casos de Testes Realizados p. 50

6.1 Visualizador de Imagens . . . . . p. 50

6.1.1 Swing . . . . . p. 51

6.1.2 Blackberry . . . . . p. 55

6.1.3 Android . . . . . p. 60

6.2 Foto com dados do GPS . . . . . p. 63

6.2.1 Android . . . . . p. 64

6.2.2 Blackberry . . . . . p. 68

6.2.3 Swing . . . . . p. 70

7 Considerações Finais p. 71

Referências p. 73

Apêndice A – Código Fonte e Modelos Utilizados nos Testes p. 76

1 Introdução

Existem diversos paradigmas para a modelagem de sistemas computacionais, sendo que a orientação a objetos é uma forma muito utilizada atualmente. A programação orientada a objetos utiliza como base o conceito de objetos. Um objeto é a representação de um elemento relacionado com o sistema sendo desenvolvido. Por exemplo, em um sistema de venda de livros, cada um destes livros são representados por objetos, assim como um comprador também é representado por um objeto. Neste paradigma, cada objeto pertence a uma classe, que indica quais atributos serão comuns a todos os objetos daquela classe, bem como quais ações eles podem realizar, sendo que as ações levam o nome de métodos. No exemplo do sistema de venda de livros, cada livro sendo vendido pode ser um objeto do tipo Livro, onde todos possuiriam os atributos título, preço e autor.

Para auxiliar a modelagem de sistemas seguindo o paradigma de orientação a objetos, foi desenvolvida a Unified Modeling Language (UML) [1]. Esta é uma linguagem gráfica que permite a modelagem dos diversos aspectos que um sistema possui. A UML possui, por exemplo, Diagramas de Classes que permitem a modelagem de todas as classes que compõem o sistema, e Diagramas de Sequência que descrevem o comportamento dos métodos de cada classe.

Apesar de ser possível modelar um sistema completo com a UML, raramente este modelo existe no início de um novo projeto. Uma forma de se projetar aplicações orientadas a objetos, é iniciar com um modelo de alto nível da aplicação desejada. Este modelo conteria as classes básicas que compõem a aplicação, bem como uma descrição das ações que devem ser realizadas. Após esta modelagem inicial, o modelo passa por uma série de refinamentos, onde o nível de abstração diminuí gradativamente, adicionando-se mais informações para que seja possível implementar a aplicação desejada.

Em algum destes passos de refinamento do modelo, pode-se decidir pelo uso de algum framework, o que facilita a implementação de alguma funcionalidade da aplicação. Deste modo, o modelo da aplicação será alterado para incluir as classes e os métodos do framework que serão utilizados pela aplicação.

Desenvolvidos a partir dos conceitos de reuso de software, um framework é um conjunto de funções previamente implementadas e testadas, que auxiliam o desenvolvimento. É possível,

então, que um programador escolha um ou mais frameworks que possuam as funcionalidades que ele deseja, de modo a diminuir a quantidade de código a ser programado, e também reduzindo a quantidade de erros.

Devido às vantagens que oferecem como, por exemplo, redução do tempo de desenvolvimento e melhoria da qualidade do código, frameworks se tornaram ferramentas muito utilizadas, sendo que possível encontrar frameworks para diversos de tipos de aplicação, desde aplicações embarcadas, como o Android [2], até jogos eletrônicos, por exemplo o JMonkey Game Engine [3].

Quando decide-se pelo uso de frameworks, a aplicação conterá tanto classes próprias como classes provenientes do framework. A identificação das classes do framework que serão utilizadas é feita de modo manual, após um estudo do framework. Aplicações mais simples geralmente utilizam um número baixo de classes do framework, uma vez que utilizarão poucas funcionalidades, porém o número de classes utilizadas sobe conforme a complexidade da aplicação sendo desenvolvida.

Apesar de oferecer diversas vantagens, o uso de frameworks tem seus custos. Mesmo se for desconsiderado eventuais custos de licenciamento, deve-se sempre considerar o tempo que um desenvolvedor gastará para aprender um novo framework[4], muitas vezes só se justificando se o mesmo framework for utilizado em mais de um projeto. Geralmente, para auxiliar o desenvolvedor, os autores de frameworks fornecem tutoriais que demonstram como utilizar as funcionalidades básicas, deixando as ações mais complexas a cargo do usuário.

Uma outra dificuldade encontrada ao se utilizar frameworks, é que não há uma preocupação em padronizar quais funcionalidades são fornecidas, nem como utilizá-las. Isto faz com que o conhecimento que um desenvolvedor possua sobre um determinado framework não seja facilmente transferido para um novo framework.

Uma possibilidade para reduzir a dificuldade encontrada ao utilizar frameworks, é desenvolver métodos e ferramentas que identifiquem de modo automático quais classes devem ser utilizadas para realizar uma determinada ação, e como utilizá-las. Com a automatização do processo de identificação das classes, um desenvolvedor pode concentrar seu tempo no desenvolvimento de sua aplicação, reduzindo o tempo gasto aprendendo um novo framework.

Para desenvolver tal método, pode-se utilizar como base as técnicas desenvolvidas na transformação de modelo. A transformação de modelo é uma área de pesquisa que visa cumprir um dos objetivos propostos pela Model-Driven Architeture (MDA), que é, em linhas gerais, obter código executável a partir de modelos de aplicações através de uma série de transformações.

Outra área que pode auxiliar no desenvolvimento de métodos para identificar as classes de um framework, é a composição automática de serviços. Esta área busca métodos para compor serviços existentes, utilizando planejamento lógico, para se obter novos serviços.

Alguns trabalhos que buscam esta composição automática utilizam o planejamento lógico como base. Em [5], os autores utilizam uma descrição em BPEL [6] dos serviços já disponíveis, bem como uma descrição do serviço novo que se deseja, e aplicam o algoritmo de planejamento para determinar como compor os serviços antigos, de modo a gerar o serviço novo desejado. Além deste caso, diferentes algoritmos de planejamento vem sendo utilizados no desenvolvimento de softwares, concentrando-se em softwares baseados em componentes [7], *web services* [8, 9] e programas baseada em *streams* [10].

Unindo os conceitos utilizados pela transformação de modelo, com a utilização de algoritmos de planejamento, pode-se então desenvolver um método capaz identificar como compor as funcionalidades fornecidas por um framework alvo, de modo a alcançar os objetivos de uma aplicação.

1.1 Objetivos

Este trabalho tem como objetivo desenvolver um método de transformação de modelos UML que facilite o uso de frameworks. Este método deve ser capaz de, dado os modelos da aplicação sendo desenvolvida e o modelo do framework que se deseja utilizar, transformar o modelo original da aplicação de modo incluir os componentes do framework que devem ser utilizados.

O método de transformação deve ser capaz também de reaproveitar os modelos de entrada. Deste modo, um modelo de framework pode ser utilizado por múltiplas aplicações, bem como uma mesma aplicação pode ser alterada para utilizar outros frameworks.

O Diagrama de Classes do modelo alterado da aplicação deve conter as classes do framework que serão utilizadas, bem como classes novas que eventualmente serão criadas, seja para implementar interfaces ou especializar classes, de acordo com as necessidades específicas de cada framework. O modelo da aplicação deve incluir também os Diagramas de Sequência necessários para descrever como o framework deve ser utilizado, contemplando todos os métodos que deverão ser utilizados para a correta implementação da aplicação.

Para tanto, o método requer que os modelos de entrada estejam modelados corretamente. O modelo do framework deve conter todas as classes disponíveis, incluindo seus métodos e relacionamentos, bem como as informações sobre os serviços que o framework fornece. A aplicação também requer cuidados especiais na sua modelagem, onde seu comportamento deve ser modelado através de Diagramas de Atividades, onde cada atividade deve corresponder a um serviço de framework que deve ser utilizado. Ambos os modelos de entrada devem utilizar um dicionário de nomes para descrever os serviços fornecidos, no caso do modelo do framework, e dos serviços utilizados, no caso da aplicação.

1.2 Estrutura da Dissertação

Esta Dissertação está organizada da seguinte forma: no capítulo 2 serão discutidos conceitos de modelos UML e frameworks; no capítulo 3 serão apresentados os conceitos de Transformação de Modelos; no capítulo 4 serão apresentados conceitos de Planejamento Lógico; no capítulo 5 o método desenvolvido será apresentado, seguido dos casos de testes realizados, apresentados no capítulo 6; por fim, as considerações finais serão dadas no capítulo 7.

2 Modelos em UML

A Unified Modeling Language (UML) é uma linguagem gráfica utilizada para modelar sistemas orientados a objetos, mantida pelo Object Managament Group (OMG) [11]. Composta por diversos diagramas, a UML é capaz de representar diferentes aspectos do sistema sendo modelado, como a parte estrutural e a comportamental.

Um dos principais diagramas da UML, utilizados para modelar a parte estrutural, é o diagrama de classes, exemplificado na figura 1. Este diagrama é utilizado para representar as classes que compõem o sistema, com seus atributos e métodos, bem como o relacionamento entre as diversas classes.

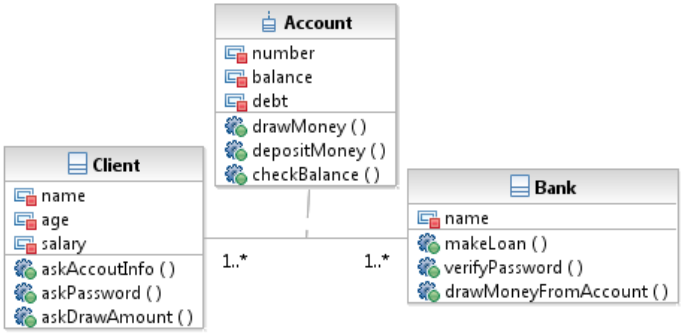


Figura 1: Exemplo de um Diagrama de Classes

Classes são abstrações usadas para representar os diferentes objetos que compõem o sistema. Cada classe representa um grupo de objetos, que possuem os mesmos atributos e responsabilidades. Durante a execução, cada objeto é uma instância de uma determinada classe, possuindo todos os atributos especificados pela classe, possuindo porém diferentes valores. Cada objeto possuirá, também, todos os métodos presentes na descrição da classe.

Outro aspecto possível de ser representado através do diagrama de classes é o relacionamento entre diferentes classes. Como foi desenvolvida para ser utilizada juntamente com linguagens orientadas a objeto, a UML é capaz de representar diversos tipos de relacionamento, onde cada um destes representam os conceitos utilizados na orientação a objeto como, por exemplo, herança, especialização, interfaces, etc.

Além da parte estática de um sistema, a UML também é utilizada para descrever sua parte dinâmica. Para tanto são utilizados outros diagramas, como por exemplo o diagrama de caso de uso, de sequência, de colaboração, de atividades, entre outros. É a partir destes diagramas que é possível modelar o comportamento esperado de um sistema. Cada um dos diagramas representam um aspecto diferente do comportamento do sistema sendo modelado.

Para facilitar a modelagem, utiliza-se o conceito de Caso de Uso. Um caso de uso é a descrição "do que" o sistema deve realizar. A descrição de um caso de uso deve conter os passos necessários para realizar uma determinada ação, sem os detalhes de implementação. Em uma descrição de caso de uso não há o conceito de classes ou objetos, e as ações são descritas utilizando linguagem natural.

Para exemplificar, podemos considerar um caso de uso simplificado de uma caixa eletrônica, onde um usuário deseje realizar um saque. A descrição deste caso de uso pode se dar da seguinte forma:

```
Caso de Uso 001: Consulta ao Saldo
Percurso Primário: - Usuário acessa a página de saque
 - Sistema requisita número da agência e conta corrente
 - Usuário informa o número da agência e conta corrente
 - Sistema requisita senha de acesso
 - Usuário fornece senha de acesso
 - Sistema confere a senha de acesso
 - Sistema requisita quantidade a ser sacada
 - Usuário fornece quantia
 - Sistema libera quantia desejada
Percurso Secundário: 1-) Caso senha não for confirmada com sucesso,
sistema emite mensagem de erro ao usuário.
 2-) Caso não possua saldo suficiente,
  oferece um empréstimo ao usuário.
```

Este seria apenas um dos Casos de Usos de um caixa eletrônico. Para se ter uma visão de todos os casos de uso do sistema, utiliza-se o Diagrama de Casos de Usos, que permite visualizar como os diversos casos de uso que compõem o sistema relacionam-se entre si, assim como os usuários do sistema, chamados de atores, utilizam os diferentes casos de uso.

A partir desta descrição em linguagem natural, é gerado um outro diagrama, chamado de Diagrama de Atividades. Este diagrama contém cada um dos passos que deverão ser realizados para implementar o caso de uso em questão. Um possível Diagrama de Atividades para o caso de uso apresentado anteriormente é mostrado na figura 2. Neste diagrama estão as ações que o

sistema irá implementar, e como é a sua sequência de execução.

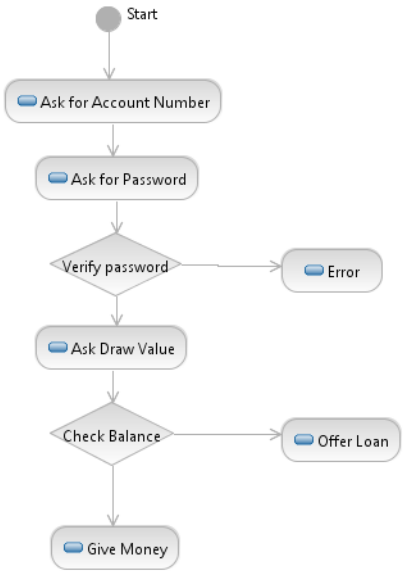


Figura 2: Exemplo de um Diagrama de Atividades

A seguir é gerado um outro diagrama que é utilizado para representar o "como" da ação, o diagrama de sequência, mostrado na figura 3. Neste diagrama estão representados os detalhes necessários para a implementação, como as classes e objetos envolvidos, bem como os métodos e sua sequência de execução.

O diagrama da figura 3 contém mais informações sobre como o sistema será implementado. Nele estão explicitados quais os métodos que serão utilizados, juntamente com os objetos e classes. Porém, antes de ser implementado, o modelo deste sistema provavelmente passaria por mais algumas fases de refinamento, onde para cada um dos métodos seria criado um diagrama de sequência, onde seria descrito em maiores detalhes o que cada um dos métodos deve fazer. Este processo de refinamento pode ser repetido múltiplas vezes, modelando cada vez em maiores detalhes o comportamento dos métodos, até que seja possível implementar o sistema em si.

Uma das desvantagens da UML, porém, é sua dependência em utilizar linguagem natural para descrever certos aspectos, como por exemplo os casos de uso. O uso de linguagem natural pode levar à ambiguidades e diferentes interpretações do que esta sendo descrito. Uma outra desvantagem em se utilizar linguagem natural é que, mesmo quando utilizada de forma a minimizar as ambiguidades, é muito difícil utilizar estas descrições como entrada para ferramentas de auxílio a geração de código.

Em uma tentativa de reduzir esta dependência da linguagem natural, a OMG criou uma extensão à UML, chamada de Object Constraint Language (OCL)[12]. A OCL é uma linguagem de especificação, para ser utilizada juntamente com a UML. Apesar de seus diversos diagramas, a UML não possui expressividade suficiente para descrever restrições às classes, sendo que estas

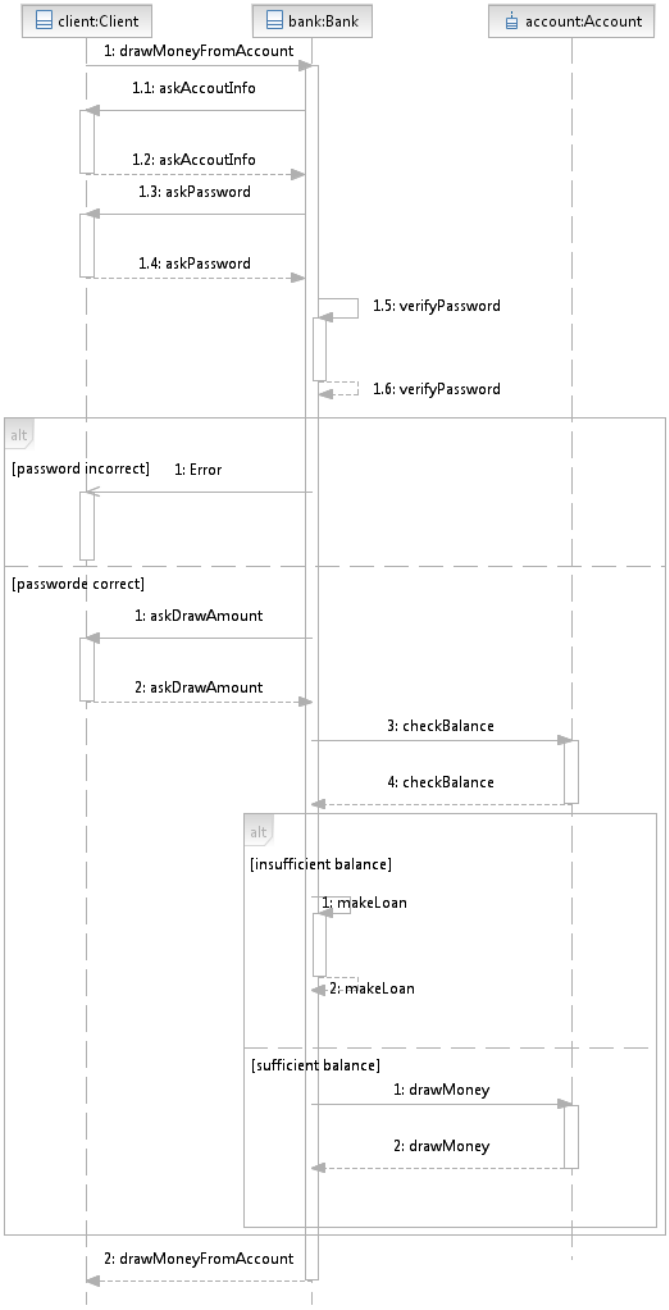


Figura 3: Exemplo de um Diagrama de Sequência

restrições são geralmente descritas em linguagem natural.

Desenvolvida de modo a ser simples de utilizar, a OCL é utilizada para descrever tais restrições a objetos. A OCL é capaz de descrever limitações aos estados possíveis de uma classe, as pré/pós condições de um método, entre outros aspectos do sistema.

Por ser uma linguagem de especificação, e não de programação, as expressões escritas em OCL não são executadas, ou seja não possuem um efeito sobre o estado de uma aplicação. Essas expressões são apenas avaliadas, sendo utilizadas para determinar se as ações não levarão o

sistema a um estado não permitido.

É possível criar expressões que utilizam os atributos das classes descritas no Diagrama de Classes. Estas expressões não alteram seu valor, apenas verificam se satisfazem as restrições impostas, retornando sempre um valor booleano.

Utilizando a OCL, é possível também determinar a tipagem dos atributos das classes. É especificado na OCL uma série de tipos básicos que podem ser utilizados, bem como as operações matemáticas possíveis de serem aplicadas a estes tipos.

Quando utilizada em conjunto com operações, a OCL pode ser utilizada nos parâmetros de entrada, bem como o que a operação deve retornar. Dentro de uma expressão OCL, pode-se acessar os parâmetros passados na chamada da função e, através da pós-condição, determinar o valor que será retornado.

Utilizando a modelagem descrita em UML, com as restrições especificadas através da OCL, é possível gerar um modelo mais detalhado do sistema sendo projetado. A utilização da OCL auxilia na modelagem de ações, facilitando a descrição dos requisitos associados àquela ação. Uma das desvantagens da OCL, assim como da UML, é a falta de expressividade para descrever ações. No caso da OCL, existem propostas para incluir descrição de ações, como por exemplo [13].

Uma grande vantagem de se utilizar a modelagem orientada a objetos, é a modularidade que se obtêm. Esta modularidade permite que suas partes possam ser reutilizadas mais facilmente. Pode-se, também, criar conjuntos de classes para serem utilizados em diversos projetos. Estes conjuntos de classes e métodos levam o nome de frameworks.

2.1 Frameworks

Framework é uma das técnicas utilizada para se fazer o reuso de código. Os frameworks são compostos basicamente de componentes de software e decisões de design [14], podendo ser vistos também como aplicações inacabadas, sendo possível estendê-las para produzir outras aplicações customizadas [4]. Frameworks, geralmente, são desenvolvidos para dar suporte à um tipo específico de aplicações, como por exemplo o Jmonkey Game Engine [3], que foi desenvolvido para auxiliar o desenvolvimento de jogos.

O principal propósito de um framework é auxiliar o desenvolvimento de aplicações, permitindo que programadores e designers utilizem código e decisões de arquiteturas já testadas previamente. A partir deste reuso, novas aplicações podem ser construídas com maior qualidade e em menor tempo, quando comparado com projetos que necessitem projetar e desenvolver todos os componentes.

Dentre os diversos tipos de frameworks, um dos mais comuns são os orientados a objetos. Frameworks orientados a objetos, por sua vez, também estão divididos em outros tipos [15], sendo *blackbox* e *whitebox* os tipos mais importantes. A classificação de um framework orientado a objeto depende da forma com que ele disponibiliza suas funcionalidades ao programador.

Um framework *whitebox* utiliza conceitos de orientação a objeto para disponibilizar suas funcionalidades. Deste modo, quando for necessário utilizar alguma destas funcionalidades, é necessário, por exemplo, estender uma das classes fornecidas ou sobre-escrever um dos seus métodos.

Já nos frameworks *blackbox*, classes são definidas de forma a serem compatíveis com uma determinada interface, e depois adicionada a métodos ou classes específicos. Deste modo, frameworks *blackbox* escondem parte da complexidade do framework do programador, porém abre mão da flexibilidade que os frameworks *whitebox* fornecem. Exitem outros tipos de frameworks além dos dois apresentados, que utilizam combinações diferentes das características destes dois tipos de frameworks.

Tomemos o seguinte exemplo, para mostrar uma possível utilização de frameworks:

Seja uma aplicação simples, para celulares onde se verifica o estado da conexão do celular com alguma rede de dados, e imprime o resultado na tela do celular. Utilizando a UML podemos modelar esta aplicação. Um diagrama de classe possível para o programa está representado na figura 4.

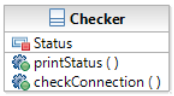


Figura 4: Diagrama de Classe da aplicação para celular

A aplicação, em uma primeira modelagem, é composta por apenas uma classe, que seria responsável tanto por verificar o estado quanto imprimir o resultado na tela.

Considerando apenas o caso de uso principal da aplicação, desconsiderando inicializações por exemplo, um diagrama de sequência que corresponde a aplicação está representado na figura 5.

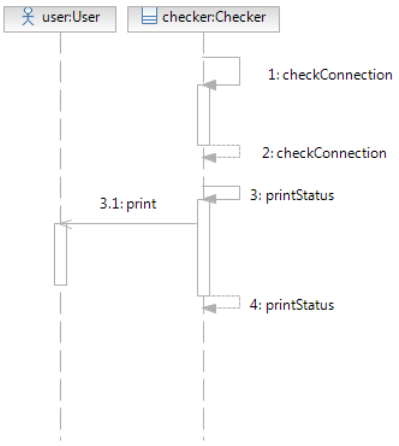


Figura 5: Diagrama de Sequência da impressão do status da conexão

Após a modelagem inicial, deve-se definir qual plataforma será utilizada. Neste exemplo foi utilizado a plataforma Android. O Android é um sistema operacional para dispositivos móveis, desenvolvido principalmente pelo Google. É disponibilizado para desenvolvedores um framework que permite acessar as funcionalidades da plataforma de modo simplificado, utilizando a linguagem JAVA.

Uma vez definida a plataforma e o framework que serão utilizados, deve-se identificar quais partes do framework são relevantes para a aplicação sendo desenvolvida. Após um estudo do funcionamento do framework, é possível identificar as classes que serão adicionadas ao modelo original, sendo que os diagramas de classe e de sequência alterados estão representados nas figuras 6 e 7 respectivamente.

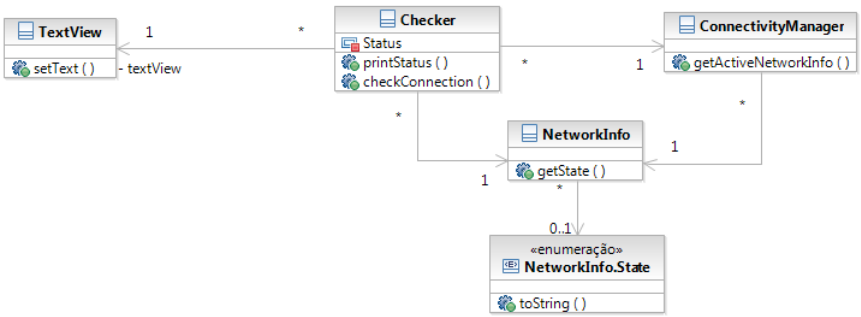


Figura 6: Diagrama de Classe da aplicação para celular modificado

A partir destes diagramas modificados é possível implementar o programa, de modo a utilizar corretamente o framework escolhido. A utilização do framework facilitou o desenvolvimento, dando acesso fácil a certos aspectos da plataforma escolhida, como por exemplo o estado da conexão, o que permite um desenvolvimento mais rápido.

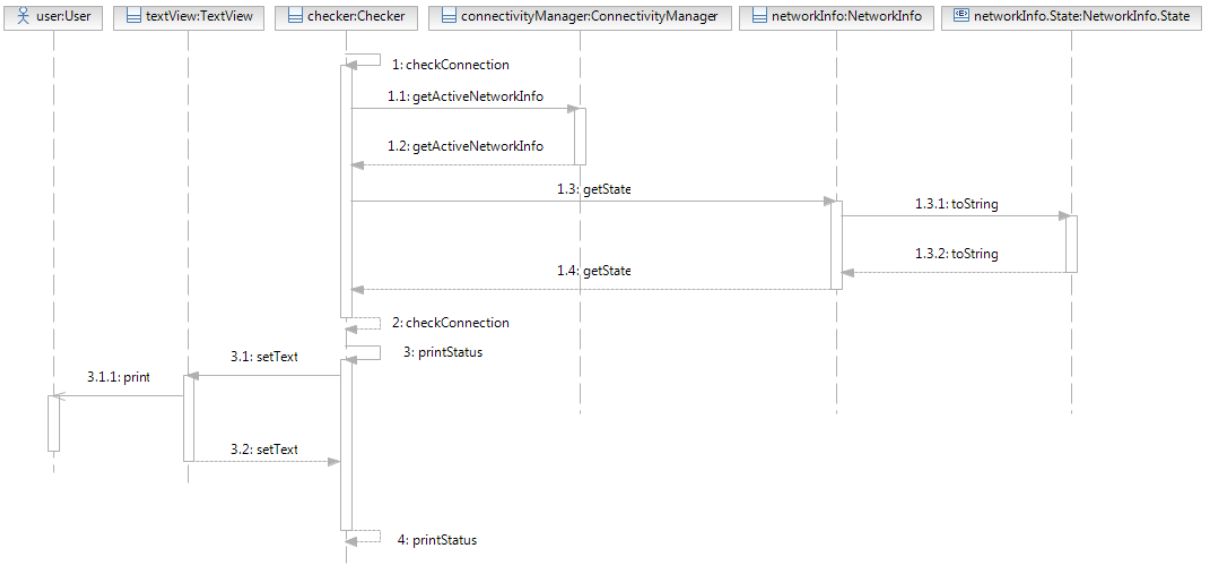


Figura 7: Diagrama de Sequência modificado da impressão do status da conexão

O exemplo apresentado é simples, porém, demonstra como a utilização de um framework pode auxiliar o desenvolvimento de software. Neste caso específico, o framework utilizado utiliza os conceito de "black box", uma vez que o caso de uso principal utiliza as funcionalidades do framework através das classes já fornecidas, sem ser necessário utilizar extensões.

Apesar das grandes vantagens oferecidas pelo uso de frameworks, existem também algumas desvantagens [4]. O uso de modo correto e eficiente de um framework requer um investimento considerável de tempo no seu aprendizado. Outro ponto importante a ser levado em conta, é que com o passar do tempo, frameworks evoluem, e como consequência, as aplicações desenvolvidas utilizando estes frameworks também podem necessitar de mudanças para continuar funcionando corretamente.

A grande vantagem dos frameworks, como mencionado anteriormente, é aumentar o reuso de código. Fornecendo um conjunto testado de classes e métodos, framework podem ser utilizados, junto com as técnicas de transformação de modelo, que serão apresentadas no capítulo 3, para cumprir os objetivos da iniciativa Model-Driven Architeture (MDA) [16].

3 Transformação de Modelos

A área de pesquisa denominada transformação de modelo, busca alcançar os objetivos propostos pela iniciativa Model-Driven Architeture (MDA)[16]. A ideia proposta pelo MDA é, a partir de modelos independentes de plataformas, obter código executável, através de uma série de transformações de modelo.

Desde o início, o problema de transformação de modelos vem atraindo grande interesse da comunidade, com diversas propostas de como seria possível resolver o problema. As técnicas de transformação de modelos podem ser divididas em dois grupos [17], modelo para código e modelo para modelo.

Mesmo pertencendo a um mesmo grupo, as técnicas do tipo modelo para código, ou modelo para texto como também é conhecido, apresentam grande variação na forma como o problema é encarado. Uma destas formas, utilizando [18] como exemplo, é a transformação de modelo baseada em regras. Nesta abordagem, seria criada uma série de padrões de transformação, a serem utilizados posteriormente.

Este padrões são compostos por três partes, o diagrama de classe, com o qual o diagrama será comparado, o padrão desejado e a regra de transformação. A ideia é, dado o diagrama de entrada, verificar a quais padrões de transformação ele se identifica, e aplicar as regras de transformação associadas.

As regras de transformação são descritas em uma forma similar à linguagem livre de contexto, possuindo símbolos terminais e não-terminais. Os símbolos terminais são compostos por código escrito na linguagem de destino, sendo organizados de modo a gerar código sintaticamente correto. Os símbolos não-terminais são utilizados como variáveis a serem utilizadas em transformações posteriores. A abordagem de utilizar regras para a transformação de modelos apresentou resultados bons, gerando algumas ferramentas, como por exemplo [19].

A transformação de modelos, quando aplicada aos métodos, ainda é uma questão em aberto. Uma das dificuldades encontradas, quando se aplica a transformação de modelos a métodos, é a falta de formalismo para descrever as ações. Algumas das propostas apresentadas, como a xUML [20], propõem primeiro uma forma para descrever os métodos, para depois aplicar as transformações.

Uma vez que se tenha um modelo de aplicação descrito em xUML, seria possível passar por um compilador de xUML, para gerar código executável. A ideia é traduzir as ações descritas em cada um dos estados do modelo para a linguagem de programação que se deseje. Seria possível, então, obter código executável a partir de um modelo.

Uma das desvantagens desta técnica, é a necessidade de se utilizar uma *action language* para se descrever as ações. Apesar de ser uma linguagem em um nível alto o suficiente para não conter detalhes de programação, a sintaxe utilizada é próxima o suficiente de uma linguagem de programação comum, que dependendo do caso pode ser mais eficiente programar diretamente na linguagem alvo. Outra desvantagem é a necessidade de se utilizar um compilador específico para cada plataforma que se deseje utilizar.

Como o grande problema da transformação de modelos, quando aplicada em métodos, é descrever as ações necessárias, principalmente quando são manipulações de dados, a principal proposta de xUML é fornecer uma linguagem que permite a descrição das ações utilizando uma linguagem com semântica bem definida.

Uma proposta similar interessante é apresentada em [21]. Neste artigo, os autores primeiro apresentam uma *action language*, que utiliza uma sintaxe similar à OCL, com um grupo de ações em conformidade com a UML/AS [22]. A seguir, eles utilizam a linguagem apresentada para descrever o próprio método de transformação. É interessante notar a possibilidade de realizar uma meta-programação, como os autores a chamam, utilizando esta linguagem proposta. Isto se torna possível porque se consegue descrever a sintaxe da linguagem utilizando a própria UML, tornando a linguagem reflexiva.

No grupo de transformações de modelo para modelo, também podemos encontrar diferentes abordagens ao tema. Estas técnicas variam deste mudanças no nível de abstração de um modelo até a transformação entre diferentes meta-modelos.

Em [23] os autores propõem um algoritmo de transformação de modelo que auxila o projeto de software, baseando-se no processo de modelagem por cenários. A ideia deste processo é gerar modelos abstratos baseado na descrição de uma série de cenários que o sistema deve suportar, que é chamada a fase de análise, para depois gerar um modelo único que será implementado posteriormente, chamada de fase de *design*. O que os autores propõem no artigo é um algoritmo de transformação que auxilia a geração do modelo de design a partir do modelo de análise.

Um outro tipo de transformação de modelo para modelo é o caso de transformação entre meta-modelos diferentes. Um meta-modelo é a descrição genérica de um modelo. Por exemplo, o Diagrama de Classes da UML possui um meta-modelo, que, de modo simplificado, é composto por classes, atributos, métodos e relacionamentos. Utilizando então este conceito de meta-modelos, temos então técnicas que transformam um dado modelo de um meta-modelo para outro, como por exemplo [24]. Neste artigo os autores mostram como é possível gerar um

transformador de modelos utilizando exemplos de transformações como guia. Eles mostram, para exemplificar o processo, como seria possível gerar diagramas de Entidade-Relacionamento para a descrição de banco de dados através de diagramas de classe UML. Utilizando alguns exemplos de transformações conhecidas o método é capaz de inferir regras mais genéricas.

Por fim, vale a pena citar o trabalho [25]. Neste trabalho os autores transformam um modelo UML para um modelo equivalente em lógica de predicados. Utilizando esta representação, o método é capaz de achar erros e inconsistências no modelo UML original.

Pesquisas mais recentes, começaram um processo de definir formalmente o que seria uma unidade de transformação, para validar os métodos de transformação de modelos [26], para garantir que eles convergem e terminam corretamente, bem como verificar como compor diferentes sistemas de transformação [27]. Porém, apesar dos diversos trabalhos pesquisados, poucos focam no problema de frameworks.

Dois trabalhos que lidam com o problema de reuso, uma generalização do problema dos frameworks, valem ser mencionados. O primeiro, [28], propõe um método para identificar e classificar componentes de um sistema para reuso. Com este método é possível, a partir de um sistema já pronto, identificar quais de suas partes poderiam ser utilizados em outros projetos. Porém, este método apresenta falhas ao ser aplicado a frameworks, uma vez que seu principal propósito é identificar componentes e não como incluí-los em um projeto novo, que é a grande dificuldade quando se trabalha com frameworks.

O segundo trabalho importante na área de reuso é [29]. Neste artigo, o autor propõe uma álgebra para se modelar e reutilizar componentes, o que poderia ser aplicada a softwares. Mas, devido a sua data de publicação, apresenta certas limitações, visto que foi publicado antes da UML ter sido criada, e se tornar o padrão *de facto* para modelagem de software. Isto faz com que a aplicação deste método se torne limitada, pois não considera muitos dos conceitos utilizados em orientação a objetos, como classes, relacionamentos, mensagens, etc.

Além da transformação de modelos, uma outra área de pesquisa que auxilia no problema de se trabalhar com frameworks, lida com a composição automática de *web-services*. Os artigos [30] e [9] são dois trabalhos representativos desta área. Ambos artigos utilizam um algoritmo de planejamento para criar um *web-service* novo, utilizando como entrada a descrição em BPEL dos *web-services* atualmente disponíveis. O novo serviço criado será uma composição dos serviços disponíveis. Os artigos diferem na estratégia empregada pelo planejador utilizado, onde o primeiro utiliza um planejador baseado em verificação de modelos (*model-checking based planning*) enquanto que o segundo utiliza uma abordagem baseada em conhecimento (*knowledge based planning*).

Um terceiro artigo, [8], apresenta um método de composição de *web-services* que, ao invés de utilizar uma descrição BPEL dos serviços disponíveis para composição, é utilizada uma descrição

comportamental dos serviços como entrada. Para descrever o comportamento dos serviços, os autores utilizam um dicionário de ações, onde cada símbolo deste dicionário corresponde um serviço específico, com uma semântica bem definida, sendo que todos os usuários deste dicionário tem o mesmo entendimento sobre o que cada símbolo significa. Utilizando estes símbolos, o método é capaz então de, a partir da descrição dos serviços disponíveis, criar uma máquina de estado que representa o novo serviço que se deseja compor.

Como pode-se perceber, não existem muitos trabalhos que são aplicados diretamente para o uso de frameworks. A composição de *web-services* é a área que possui um conjunto de técnicas que mais se aproximam seria necessário para auxiliar o uso de frameworks, mas estas técnicas são focadas em aplicações que utilizam um outro paradigma em sua arquitetura, focando em aplicações orientadas a serviços, enquanto que se deseja utilizar aplicações orientadas a objetos. Mas, apesar de não ser possível utilizá-las diretamente, muitos dos conceitos e soluções utilizadas nestes trabalhos serão utilizados como base para um método novo, o qual este trabalho apresenta. Porém, antes de apresentar o método desenvolvido, é necessário apresentar os conceitos de planejamento lógico, o que será feito no capítulo 4.

4 Planejamento Lógico

Planejamento é o nome dado ao conjunto de técnicas que permitem gerar uma sequência de ações para se alcançar um dado objetivo. Existem diversos métodos para resolver este tipo de problema, cada um mais indicado para uma determinada classe de problema.

Os algoritmos de planejamento são divididos em classes ou domínios, baseados nas características do problema. Por exemplo, uma das classes mais simples é a de planejamento clássico. Nesta classe, os problemas são totalmente observáveis, determinísticos, finitos, estáticos (toda mudança que ocorre é derivada de uma ação) e discretos [31].

Independente do domínio que se esteja trabalhando, é necessário primeiramente representar o problema em uma linguagem apropriada. Os elementos básicos que devem são representados são: os estados possíveis, os objetivos e as ações que o agente é capaz de realizar.

A descrição de ações é realizada de forma compatível com a utilizada para se descrever os estados. Independente da forma escolhida para a representação, a descrição de uma ação deve conter, principalmente, as condições necessárias para se realizar aquela ação (pré-condição), e o efeito que aquela ação gera no sistema (pós-condição). Se estivermos utilizando lógica de proposição, pode-se indicar tanto a pré quanto a pós-condição como uma conjunção de proposições, de modo que a pré-condição deve ser avaliada como verdadeira antes da execução da ação, e a pós avaliada como verdadeira depois.

Um dos primeiros algoritmos de planejamento lógico foi o STRIPS [32]. Este algoritmo realiza uma busca num espaço de estados, onde cada estado é descrito por um conjunto de afirmações em lógica de predicados de primeira ordem. A descrição de um problema em STRIPS é composto por três partes:

- A descrição do estado inicial do mundo;
- Um conjunto de operadores, descrevendo as ações possíveis de serem executadas;
- Um objetivo, descrita em lógica de predicados.

Este foi o artigo em que um dos exemplos clássicos de planejamento foi apresentado: o mundo dos blocos. Neste mundo a descrição de um estado é a posição de cada um dos blocos

que compõe o mundo, sendo que um estado possível seria "bloco A em cima do bloco B e bloco C no chão". Uma ação, ou operador na terminologia STRIPS, é todo evento que gera uma alteração no estado do mundo, no caso do mundo dos blocos seria colocar o bloco C em cima do bloco B, por exemplo.

Para se evitar o uso extenso de linguagem natural, as descrições de estados utilizam proposições lógicas. Uma proposição lógica é formada por um predicado e suas variáveis. No mundo dos blocos, por exemplo, poderíamos ter um predicado "emCima(x, y)" que significa que o bloco "x" está em cima do bloco "y" e "noChão(x)", indicando que o bloco x está no chão. A descrição do mundo utilizada anteriormente ficaria então:

```
noChão(A)
noChão(C)
emCima(B,A)
```

As ações, ou operadores, também possuem uma representação em lógica de predicados. Estes operadores contém são da forma:

```
Nome:
Pré-Condição:
Lista de Adição:
Lista de Remoção:
```

Sendo Precondições o conjunto de predicados que devem estar contido no estado atual para que o operador possa ser utilizado, Lista de remoção contém a lista de predicados que se tornam falsos após a execução, e portanto são removidos do estado atual, e Lista de adição os predicados a serem incluídos na descrição do estado atual. O operador pode eventualmente receber como entrada uma lista de parâmetros, onde os predicados de cada lista são avaliados com os respectivos valores passados como entrada. Um possível operador no mundo dos blocos seria:

```
 Nome :  ColocarNoChão(x)
  Pré Condição : emCima(x, y)
  Adição: noChão(x)
  Remoção: emCima(x, y)
```

que descreve a ação de colocar um bloco no chão. Este operador só pode ser utilizado se um bloco estiver em cima de outro (emCima(x, y)) e, após sua execução, o bloco passa a estar no chão (noChão(x)) e não mais em cima de outro bloco (emCima(x, y)).

Após a modelagem do mundo, o algoritmo de planejamento pode ser utilizado. O algoritmo realiza-se uma busca no espaço de estados possíveis, de modo a encontrar um caminho que permita alcançar o estado objetivo a partir do estado inicial. O STRIPS utiliza uma estratégia de busca chamada de encadeamento para frente, ou *forward planning*, onde o estado inicial é considerado a raiz de uma árvore. A esta raiz é adicionado os estados possíveis de serem alcançados através de uma única ação. Em seguida, usando um destes nós-filhos como referência, são adicionados os estados possíveis de serem alcançados a partir deste nó com uma única ação. O processo é repetido até que o estado objetivo seja alcançado. O plano de ações, então, corresponde ao caminho na árvore que leva da raiz até o nó representando o objetivo.

Ao final de sua execução, obtêm-se uma sequência de operadores, que fazem com que o mundo saia do estado inicial e chegue ao estado final. Este algoritmo gera um tipo de plano que é chamado de ordem total, gerando uma sequência fixa de operações que devem ser executadas. Esta sequência fixa é, na verdade, uma limitação do algoritmo, de modo que o plano não é capaz de considerar soluções com ações concorrentes. Além disso, devido a sua estratégia de busca, o algoritmo possui um problema de escalabilidade, onde seu espaço de busca pode crescer rapidamente com o acréscimo de operadores, já que a cada passo do planejamento deve-se considerar todos os operadores aplicáveis ao estado atual.

Devido a estas limitações, este algoritmo foi alterado mais tarde por Tate [33], gerando o algoritmo NONLIN. Este algoritmo utiliza o conceito de menor comprometimento. Segundo este conceito, evita-se fixar a ordem das ações. Deste modo, ao invés de afirmar que uma ação deve ocorrer em determinado instante, busca-se afirmações do tipo "a ação A deve ocorrer em algum momento antes da ação B". Assim, consegue-se expressar planos de modo mais flexíveis, obtendo-se o que se chama de planos de ordem parcial.

Além disso, o algoritmo realiza uma busca no espaço de planos, ao invés de espaço de estados. Para isso, ao conjunto de operadores é adicionado dois operadores especiais, o "INICIO" e o "FIM". O operador "INICIO" não possui pré-condições e possui como efeito as condições iniciais, enquanto que o operador "FIM" não possui efeito e possui como pré-condições o estado objetivo. Além do plano atual, o algoritmo também possui um conjunto que contém todas as pré-condições ainda não satisfeitas de todos os operadores que o plano contém.

O planejador é iniciado tanto com um plano vazio quanto com um conjunto vazio de pré-condições não satisfeitas. O primeiro passo é, então, adicionar o operador "FIM" ao plano e adicionar suas pré-condições ao conjunto. A cada iteração, o planejador procura operadores que satisfaçam pré-condições ainda em aberto no plano. O algoritmo continua a adicionar ações até que nenhuma ação possua condições em aberto e o operador "INICIO" for adicionado ao plano. Com esta estratégia de busca, consegue-se dividir o plano em sub-objetivos, construindo planos parciais para resolver cada condição em aberto de modo individual. Esta estratégia de busca

que é iniciada pelo objeto e termina nas condições iniciais, leva o nome de encadeamento para trás, ou *backward planning*.

A estratégia *backward planning* oferece vantagens em relação ao *forward planning*, principalmente em relação ao tamanho do espaço de busca. Como somente operadores cujos efeitos serão utilizados são considerados, precisa-se levar em consideração um número menor de operadores, o que reduz o tempo de busca. Porém, o algoritmo se torna mais complexo, uma vez que precisa ser levado em conta operadores concorrentes e controlar o aparecimento de loops. Apesar de suas diferenças, os dois algoritmos são completos, uma vez que ambos os algoritmos implementam uma estratégia de busca em largura, considerando eventualmente todo plano capaz de ser gerado pelos operadores, e corretos, já que nenhum estado gerado durante o planejamento é inválido.

O método de transformação desenvolvido utiliza um algoritmo de planejamento ordem parcial similar ao NONLIN. A principal diferença entre o algoritmo NONLIN e o implementado é que os operadores do algoritmo utilizado não possuem o conjunto de "Lista de Remoção". Esta simplificação foi feita pensando que cada operador iria representar um método de uma classe, que seria utilizado para criar ou executar elementos do sistema, sem remover ou desfazer nada que tinha sido executado previamente.

Existe uma segunda alteração no algoritmo implementado, onde um operador é adicionado ao plano atual se pelo menos uma de suas pós-condições estiverem contidas no conjunto de condições em aberto. Isto ocorre em oposição ao original, que somente adiciona operadores quando o conjunto de pós-condições está contido no conjunto de condições em abertos. Isto é feito devido a particularidades do domínio em que se está aplicando o planejador, permitindo que um método seja adicionado quando apenas um de seus efeitos sejam necessários, apesar de causar outros efeitos sobre o estado. Como nenhum dos efeitos extras do operador pode desfazer o que um outro operador irá realizar, não são introduzidos erros no plano utilizando esta alteração.

Apesar destas alterações o algoritmo implementado ainda se comporta como um planejador de ordem parcial padrão, realizando uma busca em largura no espaço de estados possíveis. Isto faz com que o planejador se mantenha completo, ou seja, se houver um plano que leve do estado inicial ao estado objeto será encontrado, e correto, onde todos os passos adicionados ao plano levam a um estado válido. Porém, esta estratégia trás alguns problemas, principalmente de escalabilidade, onde o tempo para percorrer o espaço de estados pode crescer rapidamente com a adição de operadores ao espaço de busca. O algoritmo de planejamento utilizado é descrito em mais detalhes na seção 5.3.

5 *Método Framer*

Neste capítulo será apresentado o método de transformação de modelos Framer, que foi desenvolvido neste trabalho. O método consiste em, de modo simplificado, utilizar um algoritmo de planejamento lógico para identificar os componentes do framework a serem utilizados pela aplicação em desenvolvimento. Uma vez identificados os componentes, o método de transformação altera, ou cria quando necessário, os Diagramas de Classe e Sequência da aplicação de forma a indicar como utilizar os componentes selecionados.

O Framer, então, requer dois modelos de entrada: o modelo do framework e o modelo da aplicação, ambos descritos em UML. O modelo do framework contém todos os componentes disponíveis, bem como informações sobre os serviços que o framework prove. Os serviços do framework devem utilizar o dicionário de nomes para descrever seus serviços. O modelo da aplicação é a descrição de alto nível da aplicação que será desenvolvida. Este modelo deve conter informações sobre as classes da aplicação, bem como uma descrição do comportamento que a aplicação deve possuir. O comportamento da aplicação deve ser descrito pelos Diagramas de atividades do modelo, sendo que estes diagramas devem utilizar, assim como o modelo do framework, o dicionário de nomes de serviços.

As informações sobre os componentes disponíveis, contidas no modelo do framework, permitem que o Framer identifique o que o framework é capaz de realizar. Unindo estas informações, com os objetivos da aplicação, contida em seu modelo, o método é capaz de identificar quais componentes do framework serão utilizados pela aplicação, bem como sua ordem de utilização.

Uma vez identificado os componentes, são criados os Diagrama de Sequência necessários para descrever o uso destes. Nestes Diagramas é possível identificar não somente os métodos utilizados do framework, bem como os objetos e como os objetos relacionam-se entre si.

Com base nos Diagramas de Sequência, é possível inferir um Diagrama de Classe válido, contendo os relacionamentos entre as classes que comporão a aplicação final. Este diagrama contém, também, eventuais classes novas que serão criadas para utilizar o framework corretamente.

Cada um dos passos do método de transformação será exposto em maiores detalhes nas seções seguintes. Na seção 5.1 é apresentado como são os modelos de entrada (aplicação e framework). Na seção 5.2 é apresentado como as informações contidas em cada modelo são

tratadas de forma a ser utilizável pelo planejador lógico. Na seção 5.3 o planejador em si é apresentado. Por fim, na seção 5.4 é apresentado como o modelo final da aplicação é criado a partir dos resultados obtidos pelo planejador.

5.1 Modelos de Entrada

Para a Transformação de Modelos para Frameworks funcionar, são necessários dois modelos de entrada, ambos descritos em UML. O primeiro é o modelo da aplicação que se deseja alterar. O segundo é o modelo do framework que será utilizado.

Como a UML utiliza linguagem natural para descrever alguns aspectos dos seus modelos, como por exemplo relacionamento e comportamento de objetos, o método assume três premissas. A primeira premissa assume que todos os modelos estão corretos e, que se forem respeitadas as restrições descritas nele, o código resultante irá funcionar. Por exemplo, se a descrição de determinado método não conter nenhuma restrição específica, assume-se que sua execução sempre é possível se todos os argumentos de entrada forem fornecidos.

A segunda premissa diz respeito à descrição das atividades da aplicação, onde assume-se que ela será descrita como uma sequência linear de serviços a serem utilizados.

A terceira premissa, e talvez mais importante, diz respeito aos nomes utilizados para se descrever comportamentos, tanto do framework quanto da aplicação. Devido a dificuldade de se descrever de forma precisa o comportamento de objetos, assume-se que existe um dicionário de nomes para os serviços do framework e para as atividades da aplicação. Cada um destes nomes é único e possui uma semântica bem definida, de comum acordo tanto entre os desenvolvedores do framework quanto da aplicação. Por exemplo, caso um serviço de frameworks para interface gráfica tenha o nome "Criar Janela", qualquer framework que fornecer tal serviço deve gerar o mesmo efeito no sistema, mesmo que este serviço seja acessado de forma diferente entre os frameworks. Da mesma forma, se duas aplicações diferentes utilizarem o serviço "Criar Janela", assume-se que o efeito que ambas desejam seja o mesmo.

A seguir será mostrado, então, como são os modelos de entrada utilizadas pelo método.

5.1.1 Modelo da Aplicação

O modelo da aplicação é composto por um Diagrama de Classe, Diagramas de Atividades e Diagramas de Sequência. O Diagrama de Classes contém as classes específicas que serão utilizadas pela aplicação. Posteriormente, este diagrama será atualizado pelo método para incluir as classes do framework.

Além da descrição da estrutura da aplicação, é necessário descrever seu comportamento,

utilizando para isso os Diagramas de Atividades. Cada um dos Diagramas de Atividades descrevem um caso de uso específico da aplicação, utilizando os nomes do dicionário de serviços para se descrever o comportamento desejado. No seu estado atual, o método suporta apenas Atividades lineares, ou seja, não é possível utilizar *loops* ou *branches* no Diagrama de Atividades.

Por fim, o modelo deve conter um Diagrama de Sequência para cada Diagrama de Atividades que o modelo possui. Estes Diagramas de Sequências servem para juntar as informações dos Diagramas de Classes com os Diagramas de Atividades, sendo que os Diagramas de Sequência são utilizados para identificar qual método, e portanto qual objeto, é responsável por iniciar cada caso de uso descrito pelos Diagramas de Atividades.

Para ilustrar como seria um modelo de aplicação, consideremos como exemplo uma aplicação que pede a um usuário que escolha uma imagem e, em seguida, a imprime em uma janela previamente criada.

Um modelo simplificado desta aplicação é formado por uma única classe, com um único método, como indicado na figura 9. O comportamento esperado da aplicação é descrito pelo Diagrama de Atividades da figura 8. Observa-se neste diagrama que para se cumprir os requisitos desta aplicação é necessário primeiro "Criar uma Janela", pedir que usuário "Escolha um Arquivo" e em seguida "Desenhar a Imagem" na janela criada.

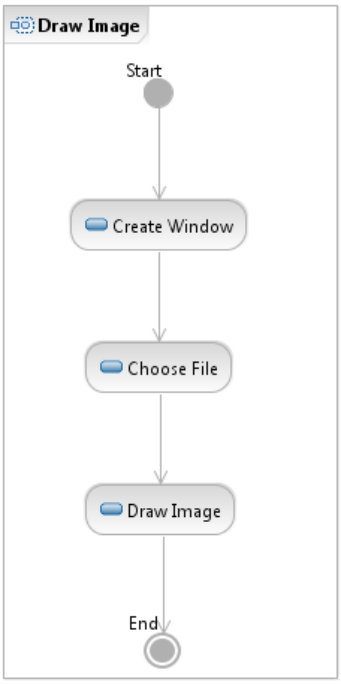


Figura 8: Diagrama de Atividades da Aplicação de Exemplo

Por se tratar de uma aplicação simples, ela será modelada através de uma única classe, que contém também um único método. O Diagrama de Classes resultante é apresentado na figura

9.

Figura 9: Diagrama de Classe da Aplicação de Exemplo

Para completar o modelo, é incluído o Diagrama de Sequência da aplicação. Este diagrama mostra que o método "printImage" é responsável por iniciar o caso de uso da aplicação.

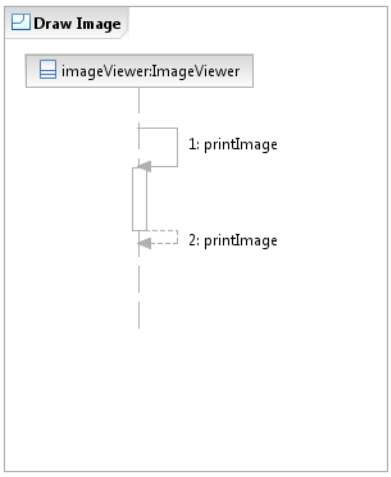


Figura 10: Diagrama de Sequência da Aplicação de Exemplo

Neste caso a aplicação possui apenas um Caso de Uso, mas é possível que a aplicação possua múltiplos casos de uso, cada um sendo iniciado por um método distinto. A seguir será mostrado como o framework é modelado para ser utilizado pelo método de transformação.

5.1.2 Modelo do Framework

O modelo do framework é a segunda entrada que o método de transformação necessita. Ele contém as informações sobre a estrutura do framework, assim como sobre seu funcionamento. O modelo será composto pelos Diagramas de Classe e de Sequência da UML.

O Diagrama de Classe é utilizado para informar quais classes o framework disponibiliza. Além das informações usuais sobre classes, como atributos, métodos e relacionamentos, o diagrama também contém informações sobre os serviços que o framework fornece. Cada serviço fornecido pelo framework é representado por uma interface, associada a uma classe através do relacionamento indicado na figura 11. A interface é nomeada com base no dicionário de nome de serviços. A interface indica também qual o método que é invocado para se utilizar o serviço que ela implementa, sendo que, se nenhum método estiver explicitado, assume-se que o serviço é executado sempre que se cria um objeto do tipo da classe associada à interface.

Figura 11: Relacionamento que indica um Serviço fornecido pela Classe

Além do Diagrama de Classe, o modelo do framework pode conter um ou mais Diagramas de Sequência. Cada um destes diagramas irá expor o funcionamento interno de algum mecanismo do framework. Este diagrama é utilizado quando alguma funcionalidade que o framework fornece necessita utilizar algum método de *callback*. Desse modo, é possível identificar qual é o método que inicia a sequência de mensagens, assim como qual é o método e objeto que serão utilizados como *callback*. Para este método, assume-se que um método que não inicia uma mensagem dentro de sua descrição de comportamento pode ser sobre-escrito sem introduzir inconsistências no funcionamento do framework, e portanto, será utilizado como o método de *callback* do mecanismo sendo descrito.

Para exemplificar o modelo de um framework, utilizaremos um modelo simplificado do framework Swing. Este é framework que auxilia o desenvolvimento de interfaces gráficas para JAVA.

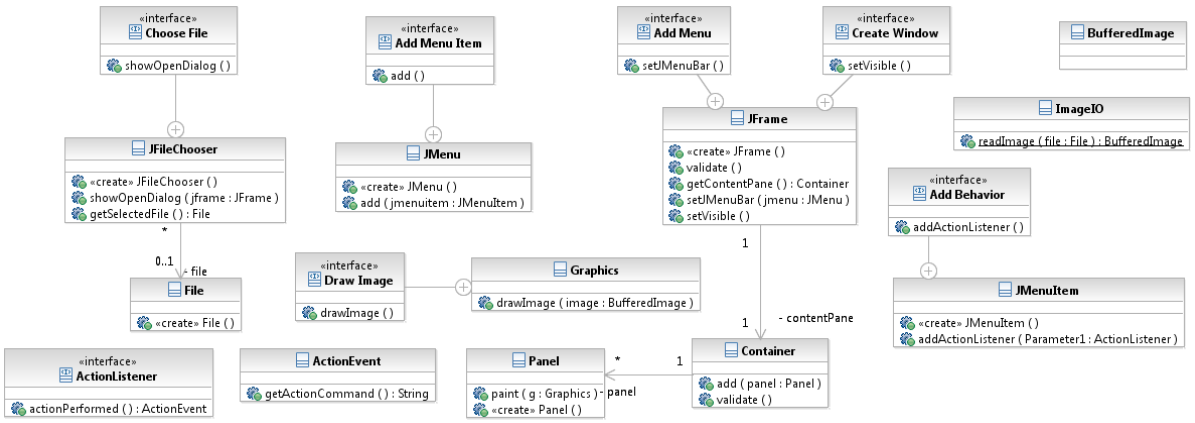


Figura 12: Diagrama de Classes do Framework Swing

Na figura 12 é apresentado um Diagrama de Classe simplificado do framework. É possível identificar no diagrama as diversas classes que o framework fornece. É possível, também, identificar os serviços que o framework fornece, através das interfaces, como por exemplo o serviço “Create Window” da classe “jFrame” e “Draw Image” da classe “Graphics”.

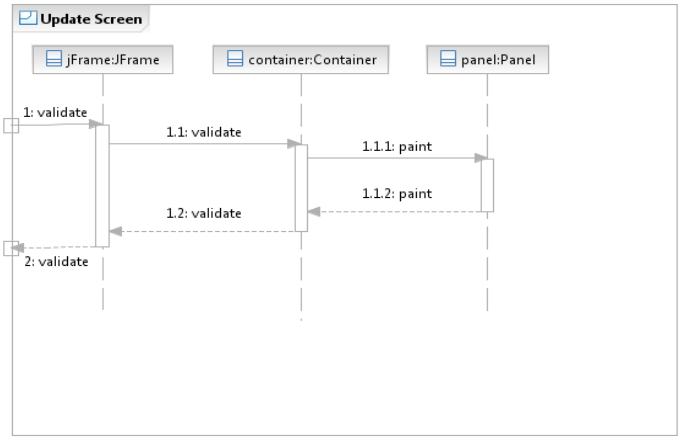


Figura 13: Diagrama de Sequência do Framework Swing

Além do Diagrama de Classes, o modelo possui um Diagrama de Sequências, apresentado na figura 13. Este diagrama representa a sequência de mensagens envolvida no mecanismo de se atualizar uma tela do Swing. O mecanismo é iniciado quando o método "validate" de um objeto do tipo "jFrame" é utilizado. Este método invoca, então, o método "validate" do objeto do tipo "Container" associado a ele. Por fim, o objeto "Container" invoca o método "paint" do objeto "Panel". Nota-se que este último método, "paint", não inicia nenhuma mensagem nova, portanto este método é utilizado como *callback* pelo mecanismo de atualização da tela, o que indica que toda a lógica de como desenhar um objeto específico dentro do framework Swing será feita dentro deste método.

Com isto, os modelos de entradas estão completos. Porém, na sua forma atual, descritos em Diagramas da UML,não é possível aplicar o algoritmo de planejamento. Portanto, ambos os modelos serão representados em uma forma lógica, forma esta que é mais fácil de ser utilizada pelo planejador. Na seção 5.2 será mostrado como tal conversão ocorre.

5.2 Conversão dos Modelos de Entrada para a forma lógica

Nesta seção, será mostrado como os modelos de entrada do método de transformação, tanto o modelo do framework quanto o modelo da aplicação apresentados na seção 5.1 , são representados em uma forma lógica. Cada um dos modelos possui um propósito específico dentro do método, sendo que, de modo geral, o modelo da aplicação irá fornecer os objetivos para o planejador e o framework os operadores possíveis, sendo então, que cada modelo é transformado de forma distinta. Iniciaremos a transformação pelo modelo da aplicação, na seção 5.2.1, seguido pela transformação do modelo do framework, na seção 5.2.2.

5.2.1 Modelo da Aplicação

O modelo da Aplicação, como mencionado anteriormente, precisa de uma representação lógica para ser utilizado pelo planejador. Como o modelo da aplicação contém as informações sobre o que se deseja produzir, sendo nele que estão as informações sobre o comportamento desejado, será extraído dele os objetivos que serão utilizados pelo planejador lógico.

A ideia é, então, gerar um conjunto de objetivos a partir de cada um dos Diagramas de Atividades que compõem o modelo da aplicação. A intenção de cada um destes conjuntos é representar os diversos estados pelos quais a aplicação percorrerá durante a execução de uma de suas atividades.

Cada um destes conjunto possui um nome único, de modo que é possível identificar qual Diagrama de Atividade deu origem a um determinado conjunto. Em adição aos nomes, cada conjunto é composto por uma sequência de predicados, onde cada predicado representa uma interface que será executada.

Os predicados deste conjunto são da forma "interface(Nome)", onde Nome representa a interface que será executada, indicada em cada uma das atividades do diagrama. A ordem dos predicados deve preservar a ordem das atividades descritas no diagrama.

Para ilustrar tal conjunto, utilizemos o modelo da aplicação apresentado anteriormente. O modelo desta aplicação contém apenas um Diagrama de Atividades, indicado na figura 8. Utilizando este diagrama como base, iremos construir um conjunto de objetivos. Este conjunto receberá o nome de Draw Image, mesmo nome do diagrama. Este diagrama é composto por três atividades, a "Create Window", "Choose File" e "Draw Image". Cada uma destas atividades serão representadas por um predicado diferente. Deste modo, o conjunto terá a forma:

```
Draw Image = {interface(Create Window, interface(Choose File),
              interface(Draw Image))}
```

Uma maneira de se interpretar este conjunto é que, para a aplicação cumprir o que se deseja, é necessário primeiro executar a interface "Create Window", em seguida a interface "Choose File" e por fim "Draw Image". Somente quando estas três interfaces forem executadas, e nesta ordem, a aplicação será bem sucedida.

Vendo a aplicação como um espaço de estados, ela será bem sucedida se ela passar por um estado que contém o predicado "interface(Create Window)", em seguida por um estado que contenha o predicado "interface(Choose File)" e por fim alcançar um estado que contenha "interface(Draw Image)". Nesta representação, a aplicação muda de estado toda vez que algum dos operadores do framework é utilizado. A forma como se contrói estes operadores é mostrado na seção seguinte.

5.2.2 Modelo do Framework

O propósito do modelo do framework é descrever as funcionalidades que são utilizáveis por aplicações. Desse modo, sua representação em lógica de predicados terá como foco descrever as ações que o framework executa. Portanto, o modelo do framework será representado por um conjunto de operadores, onde cada um destes operadores são da forma

```
Nome :
 Pré-Condição:
 Efeito:
```

onde a pré-condição é formada por um conjunto de proposições que precisam ser satisfeitas para que o operador seja utilizado. Efeito é o conjunto de proposições que se tornam verdadeiras ao se executar o operador.

Uma proposição é uma afirmação sobre o estado do mundo atual. Cada proposição é composta por um predicado e as variáveis deste predicado. Os predicados utilizados são os seguintes:

- objeto(Tipo) : indica que um objeto com o Tipo indicado já foi instanciado;
- interface(Interface) : indica que a Interface foi executada;
- inicia(Mecanismo) : representa o inicio de um dos mecanismos do framework;
- link(Classe1, Classe2) : indica que existe alguma relação entre um objeto do tipo Classe1 e Classe2;
- especialização(Classe) : indica uma especialização da Classe.

Utilizando estes predicados, é construído cada um dos operadores que serão a representação do framework e que serão utilizados pelo planejador mais adiante. O modelo do framework é composto por dois tipos de diagramas, o diagrama de Classe e os diagramas de Sequência. Como estes diagramas descrevem funcionalidades diferentes do framework, a forma de se construir os operadores irá depender de qual diagrama ele é baseado. A forma como cada diagrama é tratado é apresentado a seguir.

5.2.2.1 Diagramas de Classe

O Diagrama de Classe do framework contém as diversas classes que compõem o framework. Cada uma destas classes é responsável por um conjunto de operações, ou métodos. Estas operações estão disponíveis para serem utilizadas assim que suas pré-condições sejam satisfeitas,

a menos que esteja indicado o contrário, como no caso de operações abstratas por exemplo. Cada uma destas operações será utilizada como base para um operador diferente.

Como mencionado anteriormente, cada operador é composto por um nome, um conjunto de pré-condições e um conjunto de efeitos. Para se construir o operador com base em uma operação do framework, iniciamos pelo seu nome. O nome do operador será NomeClasseDona.NomeMétodo, onde NomeClasseDona é o nome da classe que possui o método e NomeMétodo é o nome do método transformado. O nome da classe é adicionado ao nome do operador para evitar conflitos entre métodos de mesmo nome, mas de classes diferentes.

A seguir, temos o conjunto de pré-condições. Ele é construído de forma a representar os requisitos que são necessários para se utilizar um determinado método. Portanto, este conjunto é composto por:

- objeto(ClasseDona), se a operação não for de construção ou estática;
- uma proposição objeto(Tipo) para cada um dos parâmetros de entrada da operação, com o tipo correspondente à assiantura da operação.

Se a assinatura da operação não possuir algum destes itens, basta ignorar a regra associada. Isto pode levar a casos de operadores com um conjunto de pré-condições vazio, como no caso de construtores por exemplo. Nestes casos, o operador, e portanto a operação que ele representa, podem ser executados a qualquer momento que se desejar.

Este conjunto é construído de forma a representar o modo como cada operação é utilizada por um programador. Antes de se utilizar as operações é necessário possuir um objeto do tipo correto (primeira regra), possuir todos os argumentos de entrada (segunda regra) e garantir que as restrições sobre o estado da classe sejam satisfeitas (terceira regra).

Por fim, é construído o conjunto dos efeitos da operação. Este conjunto representa as consequências que a utilização de um método gera durante a execução de um programa. Este conjunto é composto por :

- uma proposição objeto(TipoDoRetorno), onde TipoDoRetorno é a tipo do parâmetro de retorno do método;
- uma proposição interface(Nome), que representa a interface por qual aquele método é responsável;
- uma proposição inicia(Mecanismo), onde Mecanismo é o nome do mecanismo iniciado pela invocação do método;
- uma proposição link(ClasseDona, TipoDoObjeto), se o método criar uma associação entre a classe e um outro objeto, ou retorna um objeto que possui uma associação com a classe.

Este conjunto tenta representar os efeitos que uma operação pode ter no sistema. Apesar de possuir vários itens, raramente uma operação possuirá todos. Em grande parte as operações irão conter um ou dois destes itens. A ideia destes predicados é indicar os motivos que levariam uma operação a ser utilizada. Por exemplo, uma operação é invocada porque é uma operação de construção e portanto irá criar um objeto novo, ou a operação em questão é responsável pela execução de uma interface que o usuário deseje.

Além dos operadores baseados em métodos das classes, é adicionado ao conjunto um operador com o nome "especializar:Classe" para cada uma das classes contidas no diagrama, e um operador "implementar:Interface" para cada interface necessária. Ambos estes operadores tem um conjunto vazio de pré-condições e possuem como efeito o predicado "especializar(Classe)" para as classe e "implementar(Interface)" para as interfaces. Estes operadores servem para identificar que é possível estender as classes originais do framework, bem como implementar interfaces que eventualmente forem necessárias.

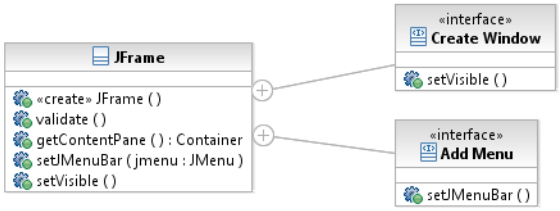


Figura 14: Classe JFrame do Framework Swing

Com isto, consegue-se construir os operadores com origem no diagrama de classe. Para ilustrar estas construções, consideremos a classe JFrame do modelo do framework apresentado anteriormente, na figura 14. O primeiro método desta classe é o construtor JFrame(). Seguindo os passos descritos, é obtido o seguinte operador:

```
Nome : JFrame.JFrame
 Pré-Condição (PC)  : {}
 Efeito (E) : objeto(JFrame)
```

Este operador não possui pré-condição, ou seja, pode ser executado a qualquer momento e seu efeito é adicionar a proposição objeto(JFrame) ao estado atual do mundo. Com este operador, representa-se a instanciação de um novo objeto do tipo JFrame durante a execução de algum programa que utilize o Swing. O método de construção, da mesma forma que operador, pode ser invocado a qualquer momento, e seu efeito é adicionar um objeto do tipo JFrame ao programa. As regras de construção dos operadores foram feitas para tentar representar como os métodos são usados pelos programadores, bem como seus efeitos no sistema.

Continuando com o exemplo, pegando um outro método seguinte da mesma classe, o getContentPane. Neste caso, o operador torna-se

```
Nome : JFrame.getContentPane
 PC : objeto(JFrame)
 E: objeto(Container), link(JFrame, Container)
```

Neste caso, a pré-condição não é vazia, uma vez que é necessária a existência de um objeto do tipo JFrame para ser possível invocar o método getContentPane. O efeito, por sua vez, é que um objeto do tipo Container passa a ser disponível, sendo que este mesmo objeto está associado ao objeto JFrame.

Como último exemplo, consideremos o método setJMenuBar. Seu operador fica da forma

```
Nome : JFrame.setJMenuBar
 PC : objeto(JFrame), objeto(JMenuBar)
 E : interface(add Menu)
```

Como nos caso anterior, uma das condições para se utilizar este método é existir um objeto do tipo JFrame. Porém, além deste objeto, é necessário um objeto do tipo JMenuBar, visto que este método possui um parâmetro de entrada. Como este método está associado a uma interface fornecida pelo framework, no caso a interface "add Menu", esta é adicionada ao seu conjunto de efeitos.

Este processo é repetido para todos os métodos, de todas as classes descritas no diagrama. Com isto, obtém-se uma parte do conjunto dos operadores que servirão como entrada para o algoritmo de planejamento. Este conjunto será completado com os operadores criados a partir dos Diagramas de Sequência, que serão apresentados a seguir.

5.2.2.2 Diagrama de Sequência

Os operadores do Diagrama de Classe, apresentados anteriormente, representam as ações que o framework já fornecem prontas, ou seja, são utilizáveis sem a necessidade de se realizar alterações no framework. Porém, em alguns casos é necessário alterar, ou até mesmo criar, certas operações para o framework realizar o que se deseja, por exemplo criando a realização de uma interface necessária ou especializando uma classe. Para tanto, são utilizados os Diagramas de Sequência do framework.

Os Diagramas de Sequência do framework contêm informações sobre seu funcionamento interno, indicando as mensagens que são trocadas entre objetos do framework durante a execução

de alguma ação. Com estas informações é possível identificar quais operações serão alteradas para que o framework realize a ação desejada.

O primeiro passo para se construir estes operadores é identificar as operações que podem ser sobre-escritas sem alterar o comportamento do framework. Para tanto, são selecionadas as operações de cada um dos diagramas que, dentro da descrição de seu comportamento, não invoquem nenhuma outra operação. Com isto, cria-se um conjunto de operações que serão utilizadas como base dos operadores.

A seguir, para cada uma das operações selecionadas, será criado um operador. Este operador receberá o nome de "sobreescrever:NomeDaClasse.NomeDoMétodo", onde NomeDaClasse é a classe que possui o método sendo sobre-escrito e NomeDoMétodo o método em si, se for uma operação de uma classe normal, ou "implementar:NomeDaClasse.NomeDoMétodo" se for uma operação de uma interface necessária.

O conjunto de pré-condições do operador será construído com base nas informações do diagrama de sequência que possui a operação sendo utilizada como base. Este conjunto será composto por:

- inicia(Mecanismo), sendo Mecanismo o nome do mecanismo que contém o método sendo sobre-escrito;
- link(ClasseInicio, ClasseDonaDoMétodo), o predicado que indica a existência de uma associação entre a classe que possui o método que inicia o mecanismo e a classe dona do método sendo sobre-escrito;
- especializar(ClasseDona), que indica a necessidade de criar uma especialização da classe ClasseDona, que possui o método sendo sobre-escrito, ou implementar(Interface) caso seja uma interface.

Estes operadores representam o que é necessário fazer para se utilizar corretamente um método sobre-escrito ou implementado. Para ser utilizado pelo framework, o método deverá ser chamado por alguma função framework (primeira regra). Para ser chamado pelo framework, é necessário criar um relacionamento entre o objeto que contém o método novo e um objeto do framework (segunda regra). Por fim, para se alterar o comportamento de uma operação de alguma classe é necessário criar uma especialização desta classe, ou no caso de interfaces, criar uma implementação desta.

A seguir, é construído o conjunto de operadores que representam os efeitos, que será composto pelos seguintes predicados:

- um objeto(TipoDoParametroEntrada), objeto que representa cada tipo dos parâmetros de

entrada do método sendo sobreescrito;

- interface(NomeDaInterface), onde NomeDaInterface é o nome interface que o método implementa.

Este conjunto, ao contrário do conjunto de efeitos dos operadores com base no Diagrama de Classe, não buscam representar as consequências que sua invocação causam no estado de um programa. Ao invés, este conjunto representa objetos e funcionalidades que um programador tem acesso ao sobre-escrever tal método, que não teria apenas através de invocações dos métodos do framework.

Consideremos como exemplo, o Diagrama de Sequência do framework Swing, apresentado anteriormente na figura 13. Neste diagrama, a única operação que satisfaz os requisitos necessários é a operação "paint" da classe Panel. Utilizando esta operação como base, obtemos o seguinte operador

```
Nome : sobreescrever:Panel.paint
 PC : inicia(Update Screen), link(JFrame, Panel), especializar(Panel)
 E : objeto(Graphics)
```

Como o nome do mecanismo neste caso é Update Screen, é necessário executar o método que o inicia nos passos anteriores, fazendo parte do conjunto de pré-condições. A este conjunto é adicionado uma associação entre os objetos JFrame, dono do método que inicia o mecanismo, e o objeto Panel, dono do método sendo sobre-escrito. Por fim, é criada uma classe de especialização da classe Panel, para ser possível sobre-escrever o método paint.

Um dos efeitos de se sobre-escrever este método é o acesso ao seu parâmetro de entrada, neste caso um objeto do tipo Graphics. Neste método em particular, não existe mais nenhum efeito associado a ele, mas nem sempre este será o caso. Com isto, o conjunto de operadores está completo. A seguir, será mostrado como o algoritmo de planejamento é aplicado.

5.3 Algoritmo de Planejamento

Nas seções anteriores, foi mostrado como converter os modelos, tanto da aplicação quanto do framework, escritos em UML, para uma representação em lógica de predicados. O modelo do framework foi transformado em um conjunto de operadores, enquanto que o modelo da aplicação em um conjunto de objetivos. Ambos estes conjuntos serão utilizados agora pelo algoritmo de planejamento.

Para realizar o planejamento, foi implementado um planejador de ordem parcial. A implementação utilizada é baseada no algoritmo original NONLIN [33], sendo que algumas alterações

foram realizadas para acomodar certas características do domínio com que estamos trabalhando.

A primeira alteração ocorre durante a seleção de possíveis operadores. No algoritmo original, um operador só é adicionado ao plano se seu conjunto de efeitos está contido no conjunto de condições em aberto. Na variação utilizada, desde que o operador possua pelo menos um dos predicados de seu conjunto de efeitos contidos no conjunto de condições em aberto, o operador pode ser utilizado.

Existe também uma diferença no conjunto de operadores. No algoritmo original, como ele era construído manualmente e com o proposito específico de ser utilizado por um algoritmo de planejamento, não era considerado o caso de mais de um operador satisfazer a mesma condição em aberto. Devido a forma como se modela sistemas orientado a objetos, é comum possuir mais de um método que gerem efeitos similares. Juntando isto com a primeira a alteração, é comum cair no caso em que existe mais de um operador pode ser adicionado ao plano, para resolver o mesmo predicado.

Para resolver esta situação, foi adicionado um conjunto de regras para auxiliar a resolução destes conflitos. Estas regras foram escritas de modo a aproximar o resultado a uma modelagem feita por pessoas. Uma regra tenta reaproveitar elementos já presentes, dando preferência a operadores que possuem o maior número de pré-condições resolvidas pelo conjunto de condições iniciais. Se aplicando esta regra não for possível escolher um único operador, dá-se preferencia aos operadores do tipo "implementar:X" ou "especializar:X", ou seja, operadores com origem nos diagramas de sequência do framework. Se ainda não for possível escolher um operador, será escolhido aquele que irá gerar um plano com o menor número de passos.

Por fim, existe uma última alteração, com respeito aos operadores. Nos operadores do algoritmo original, o efeito de cada operador possui uma lista de adição, indicando os predicados que se tornam verdadeiros, e uma lista de remoção, com os predicados que se tornam falsos e portanto são removidos da descrição do estado atual do mundo. No entanto, os operadores utilizados, como mostrado anteriormente, possuem apenas uma lista de adição.

O fato dos operadores não possuírem a lista de remoção não reduz sua capacidade de representar as operações do framework, pois durante a execução de uma atividade não é necessário remover objetos, destruir relacionamentos ou desfazer alguma ação tomada anteriormente. Porém, sem a lista de remoção, o algoritmo de planejamento é simplificado, já que a adição de um operador ao plano nunca irá gerar "risco" de desfazer alguma parte. Vale notar que, apesar de que o algoritmo de planejamento gere planos corretos e completos, não significa que o método de transformação também será, já que este é limitado também pelos modelos de entrada e saída, que podem não ser completos, apesar de corretos.

Este algoritmo necessita, da mesma forma que os planejadores tradicionais, três parâmetros de entrada: um conjunto de operadores, o estado inicial e o estado objetivo. A partir dos modelos

de entrada, framework e aplicação, obtemos o conjunto de operadores e o estado objetivo. O estado inicial para cada execução do planejador é gerado com base nos resultados dos planos anteriores.

Temos um conjunto de objetivos para cada Diagrama de Atividades da aplicação. Este conjunto ordenado de estados objetivos, que serão apresentados em sequência ao planejador. Para o primeiro objetivo do conjunto, é utilizado um estado inicial vazio, representando que ainda não foi utilizado nada do framework. Uma vez gerado o primeiro plano, é construído o estado inicial para o objetivo seguinte. Este estado é composto pela união dos efeitos dos operadores contidos no plano gerado. Com isto, representa-se todos os elementos que já foram utilizados e possíveis serem reutilizados. Este processo é repetido para cada um dos objetivos de um mesmo conjunto.

Deste modo, é gerado um plano distinto para cada objetivo de um mesmo conjunto. Uma vez que todos os planos forem gerados, estes são combinados em um único plano, gerando assim uma única árvore de ações para cada Diagrama de Atividades da aplicação. Esta árvore de ações será utilizada para guiar o processo de alteração do modelo inicial da aplicação. Com base nela, será determinado como as operações do framework serão utilizadas, bem como quais classes deverão ser adicionadas ao modelo da aplicação.

Para exemplificar o uso do planejador, utilizaremos como entrada os conjuntos gerados nas seções anteriores, onde foi construído o conjunto de operadores e o conjunto de objetivos. O conjunto obtido a partir da aplicação foi

$$G = \{interface(CreateWindow), interface(ChooseFile), interface(DrawImage)\}$$

Tomando o primeiro objetivo do conjunto, e utilizando um estado inicial vazio, obtemos do algoritmo o plano representado pela figura 15.

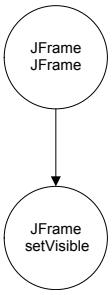


Figura 15: Resultado do primeiro objetivo

A seguir, é necessário construir o estado inicial para o próximo objetivo. Como o plano contém as ações “JFrame.JFrame” e “JFrame.setVisible”, temos que o estado inicial a ser utili-

zado, resultante da união dos seus efeitos, é

$$I_1 = \{interface(CreateWindow), objeto(JFrame)\}$$

Utilizando estas condições iniciais e o segundo objetivo do conjunto G, interface(Choose File), obtemos o plano representado pela figura 16.

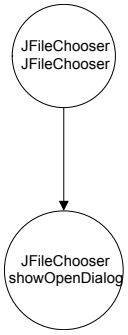


Figura 16: Resultado do secundo objetivo

Adicionando-se os efeitos destes operadores às condições iniciais, obtemos o conjunto

$$I_2 = \{interface(CreateWindow), objeto(JFrame), interface(ChooseFile),$$

$$objeto(JFileChooser)\}$$

que serão as condições iniciais para o último objetivo. Utilizando estas condições e o objetivo, obtemos plano da figura 17.

Com este plano, todos os objetivos da aplicação foram cumpridos. Porém, ainda é necessário juntar todos os planos gerados para formar uma única árvore de ações, que será utilizada como a base das transformações do modelo. O plano final, resultante da união destes planos é indicado na figura 18.

Com a árvore completa, podemos realizar o processo de transformação do modelo da aplicação. A forma como as ações descritas na árvore são utilizadas pelo processo de transformação é descrito na seção seguinte.

5.4 Alteração do Modelo Inicial

Até o momento, foi apresentado como criar a representação lógica dos modelos originais, tanto da aplicação quanto do framework, e como essa representação foi utilizada como entrada para o planejador lógico. Nesta seção, será mostrado como utilizar o resultado obtido do planejador para realizar de fato a transformação do modelo da aplicação original.

Como mostrado no final da seção anterior (fig. 18), o resultado que obtemos do planejador

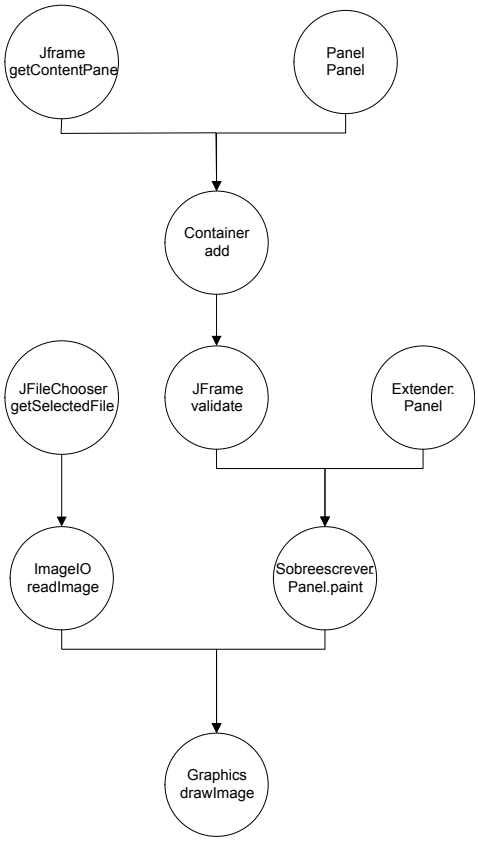


Figura 17: Resultado do terceiro objetivo

é uma árvore, onde cada nó representa uma operação do framework, que será utilizada para cumprir os objetivos da aplicação. Porém, por se tratar de uma árvore, é necessário realizar um pré-processamento, de modo a deixar as invocações de forma linear. Pode ser necessário também dividir a árvore em pedaços menores, dependendo das ações contidas nos nós da árvore.

Esta divisão tem o propósito de separar as ações em grupos menores, que darão origem a diferentes diagramas de sequência. Se for dividida, uma parte dela dará origem a um diagrama que descreve as operações invocadas pelas operações sobre-escritas ou implementadas, outra parte aos construtores das novas classes e o restante pela operação da aplicação original. Vale notar que esta divisão é realizada de modo automático.

Para se determinar como dividir a árvore, inicia-se pelo último nó da árvore. Sobe-se os ramos da árvore até que se encontre um nó que contenha uma ação de sobre-escrita ou de implementação de alguma operação do framework. Ao ser identificado este nó, a árvore será dividida em três grupos :

1. Os nós filhos;
2. Os nós “irmãos” e seus pais;
3. Os nós pais.

Os nós filhos irão representar as operações invocadas pela operação alterada. Os nós irmãos e seus pais darão origem ao construtor da nova classe e as operações dos nós pais são invocadas pela operação da aplicação original. Este processo é repetido até que a árvore não possa mais ser dividida. Utilizando a árvore da figure 18 como exemplo, temos as sub-árvores indicadas pela figura 19.

Cada uma destas árvores são identificadas, de modo que seja possível determinar em qual diagrama cada uma será utilizada.

Uma vez com as árvores divididas, é necessário realizar a linearização de cada uma. Isto é feito para tornar o plano uma sequência linear de ações, uma forma mais próxima do diagrama de sequência e, também, mais simples de se trabalhar. Para linearizar a árvore, cada ação de um mesmo nível são executadas antes de se executar as ações do nível seguinte, mas qualquer outra forma de linearização é aplicável, desde que seja respeitado as restrições impostas pelo plano.

A seguir, as ações contidas na árvore correspondente serão adicionadas ao diagrama de sequência da aplicação original. Para cada operação contida, é adicionada uma mensagem, com origem no objeto da aplicação e com destino no objeto correspondente. Utilizando a árvore da figura 19c, é gerado o diagrama de sequência da figura 20.

A seguir, para cada uma das outras árvores, é criado um novo diagrama de sequência. Estes diagramas representarão as novas operações que serão criadas, seja elas de construção ou as operações sobre-escritas ou implementação de interfaces. Utilizando a divisão anterior, a árvore 19b será utilizada para criar o novo construtor, indicado na figura 21a e a árvore 19a pela operação que sobre-escreve o método "Panel.paint", na figura 21b.

Uma vez que os diagramas de sequência foram construídos, o diagrama de classe original será alterado. A este diagrama serão adicionadas as classes do framework que são utilizadas, bem como as novas classes que eventualmente foram criadas, sejam elas uma especialização ou implementação de uma interface. Com todas as classes adicionadas, cria-se um relacionamento entre as classes que trocam alguma mensagem nos diagramas de sequência. Utilizando os diagramas de sequência criados nesta seção, pode-se alterar o diagrama de classe da aplicação original (fig. 9), obtendo-se o diagrama da figura 22.

Com isto, o processo de transformação está concluído. Os diagramas do modelo alterado possuem informações suficientes para que seja possível utilizar o framework de forma correta, para que os objetivos da aplicação sejam cumpridos. No capítulo seguinte serão apresentados alguns casos de usos onde o processo de transformação foi utilizado, bem como o resultado obtido.

5.5 Implementação do Método de Transformação

Para a realização dos testes, foi desenvolvida uma ferramenta de transformação que implementa o método proposto. A ferramenta utiliza a arquitetura descrita pela figura 23.

Todos os módulos, com exceção do Planner, foram implementados utilizando um interpretador da linguagem de transformação ATLAS [34, 35] para JAVA. O módulo Planner foi implementado em JAVA para facilitar a comunicação com os outros módulos.

O módulo "Application2Goals" é responsável por implementar a criação dos objetivos a partir do modelo da aplicação, enquanto que o módulo "Framework2Search" gera o conjunto de operadores a partir do modelo do framework escolhido. As informações geradas por estes dois módulos é utilizado pelo Planner, que implementa o algoritmo de planejamento descrito na seção 5.3. Utilizando o plano gerado, o módulo "Plan2Changes" identifica quais as classes e métodos do framework que serão utilizados e, por fim, o módulo "AddChanges" realiza as alterações necessárias no modelo da aplicação.

Vale notar que esta ferramenta implementa apenas o processo de transformação. Para utilizá-la, é necessário gerar os modelos em ferramentas de modelagem UML com capacidade de exportação em XMI. Da mesma forma, é necessário utilizar uma ferramenta de modelagem capaz de importar modelos UML descritos em XMI para visualizar o modelo final gerado. Devido a diferentes níveis de suporte às funcionalidades necessárias foi utilizado o Rational Software Modeler para exportar os modelos iniciais e o Visual Studio 2010 para visualizar o modelo alterado.

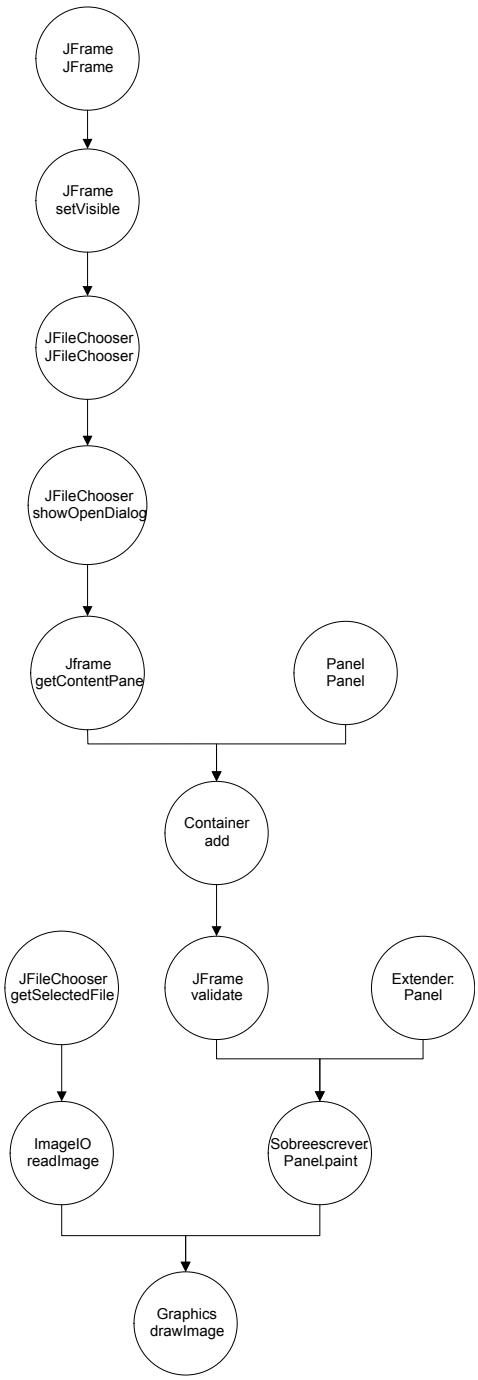
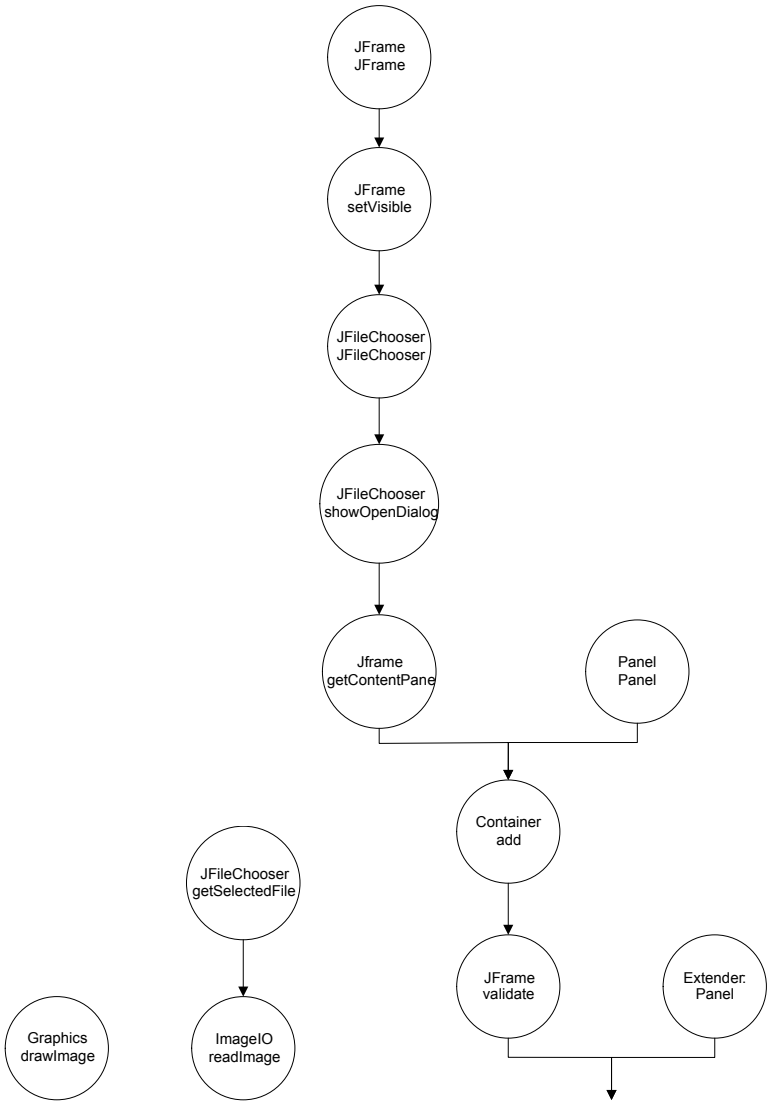

JFrame
JFrame
JFrame
setVisible
JFileChooser
JFileChooser
JFileChooser
showOpenDialog
Jframe
getContentPane
Panel
Panel
Container
add
JFileChooser
getSelectedFile
JFrame
validate
Extender:
Panel
ImageIO
readImage
Sobreescrever
Panel.paint
Graphics
drawImage

Figura 18



(a) Primeira Árvore

(b) Segunda Árvore

(c) Terceira Árvore

Figura 19: Divisão da Árvore Original

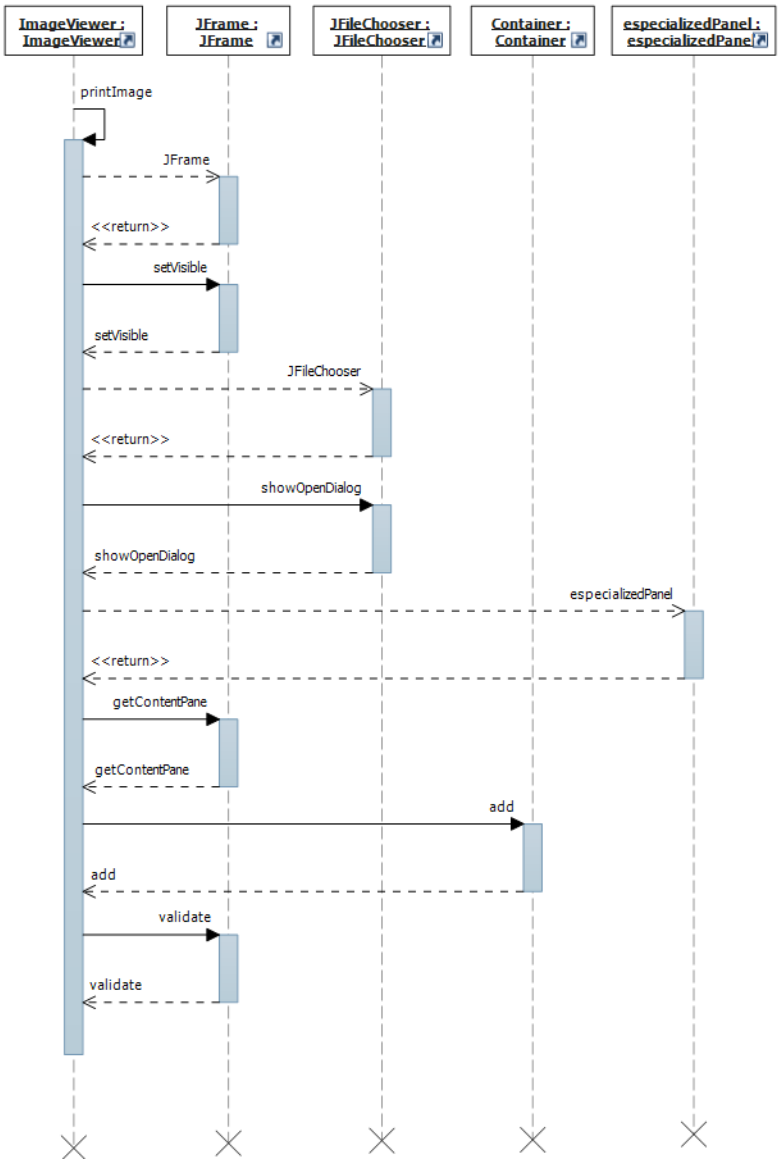
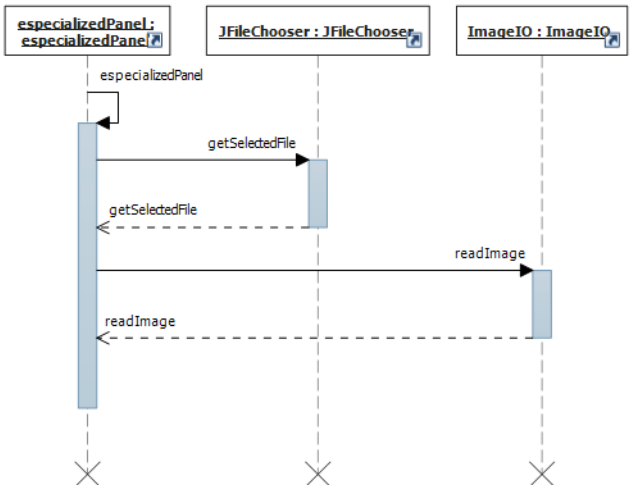
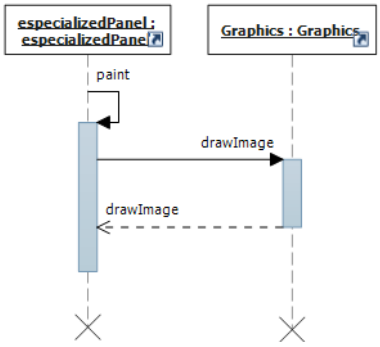


Figura 20: Diagrama de Sequência Original Modificado



(a) Novo Construtor



(b) Nova Operação

Figura 21: Novos Diagramas de Sequência

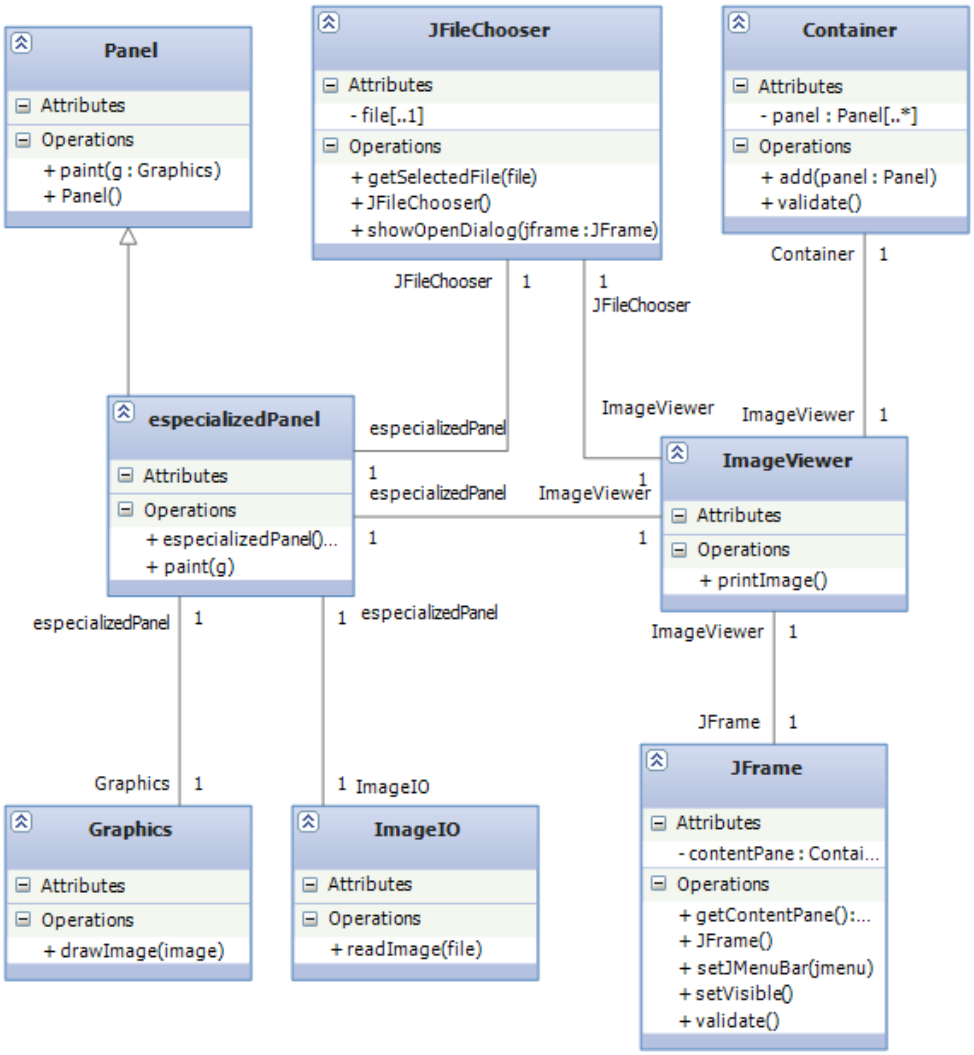


Figura 22: Diagrama de Classe Alterado

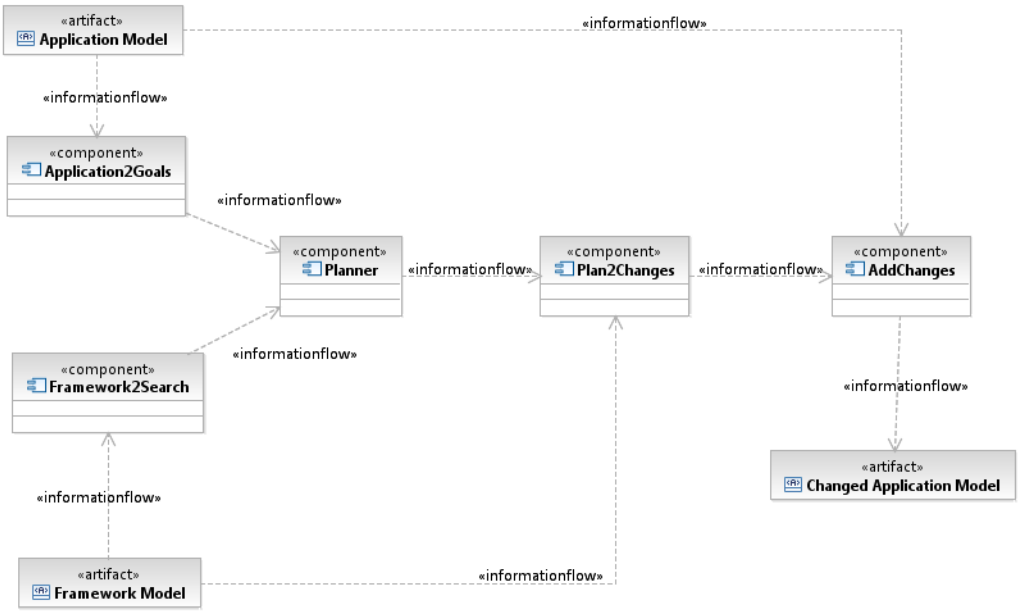


Figura 23: Arquitetura da Ferramenta de Transformação

6 Casos de Testes Realizados

Os resultados a seguir foram obtidos utilizando a implementação do método descrito na seção 5.5. Foram criadas duas aplicações de exemplo, uma aplicação para a visualização de imagens e uma aplicação para tirar fotos de um aparelho celular com informações de localização de GPS.

Como frameworks alvos, foram utilizados três frameworks: o framework para interfaces gráfica em JAVA chamado Swing [36], e os frameworks para desenvolvimento de aplicações para celulares Blackberry [37] e Android [2]. Com isto, obtêm-se um total de seis casos de transformação. Destes, quatro casos foram bem-sucedidos, de modo que o método gerou os diagramas necessários para descrever o modo como cada framework deve ser utilizado, e em dois casos a transformação não foi bem sucedida, por motivos distintos, que serão expostos adiante.

Com estes casos é possível mostrar que o método é capaz de identificar a forma como os serviços dos frameworks serão utilizados, independentemente da forma como estes são implementados pelos respectivos frameworks. Através destes casos é possível identificar também a possibilidade de se reutilizar tanto os modelos da aplicação quanto do framework e, analisando os casos em que o método falha, quais são as atuais limitações do método.

A seguir serão apresentados os casos de testes: na seção 6.1 será apresentada a aplicação de visualização de imagens, com suas transformações para os frameworks Swing, Blackberry e Android; na seção 6.2 será apresentada a aplicação para tirar fotos, e suas transformações para os frameworks mesmos frameworks.

6.1 Visualizador de Imagens

Suponhamos uma aplicação simples, para visualização de imagens. Desejamos que esta aplicação crie uma janela, mostre uma tela onde selecionaremos a imagem que se deseja visualizar e apresentar esta imagem na janela criada. A partir desta descrição, podemos criar um modelo para esta aplicação.

Iniciando com o Diagrama de Atividades, podemos modelar a aplicação como possuindo apenas um diagrama, que conterá os três passos desejados para a aplicação. O Diagrama resultante

é apresentado na figura 24.

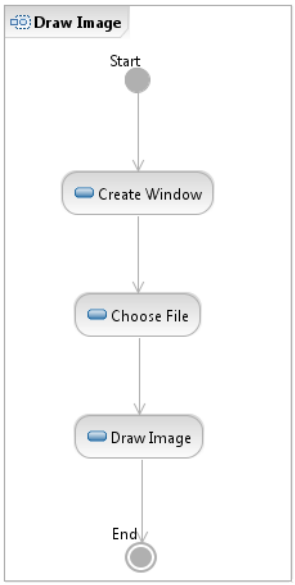


Figura 24: Diagrama de Atividades da Aplicação de Visualização de Imagens

Como a aplicação não é muito complexa, podemos modelar a aplicação como sendo composta por apenas uma classe, possuindo apenas um método, que será responsável por implementar as ações necessárias. O diagrama de classe e sequencia estão representados nas figuras 25 e 26 respectivamente.

Após modelar a aplicação, é necessário escolher o framework que será utilizado. Para este exemplo, foram gerados dois modelos diferentes, um utilizando o framework Swing e um segundo utilizando o framework para Blackberry. Estes casos são apresentados a seguir.

6.1.1 Swing

Para transformar o modelo da aplicação de forma correta, é necessário como entrada o modelo do framework a ser utilizado, neste caso o Swing. Para este exemplo foi utilizado um modelo simplificado do framework, apresentado a seguir.

Figura 25: Diagrama de Classe da Aplicação de Visualização de Imagens

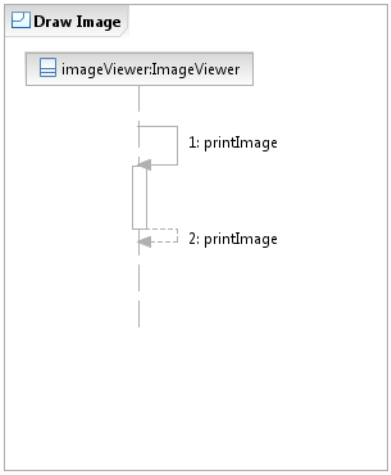


Figura 26: Diagrama de Sequência da Aplicação de Visualização de Imagens

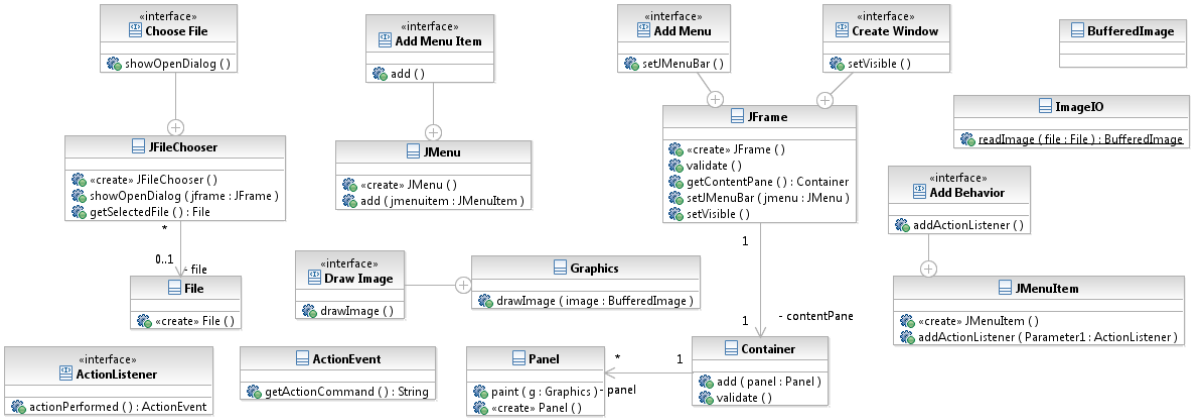


Figura 27: Diagrama de Classes Simplificado do Framework Swing

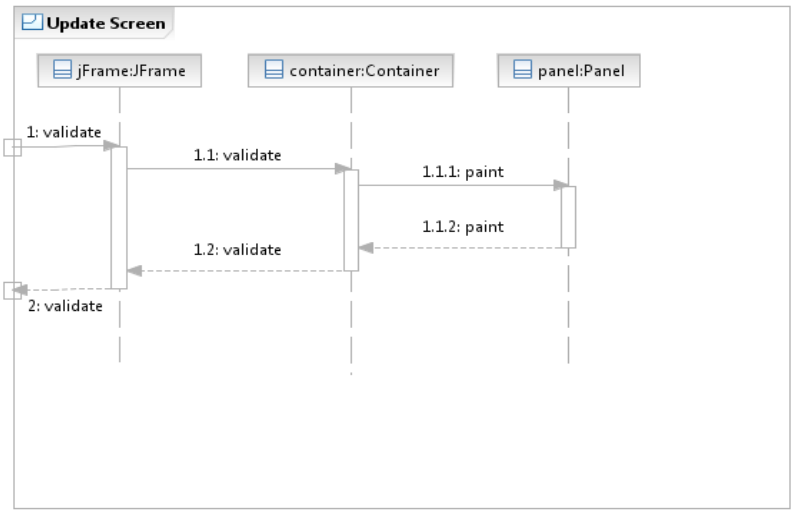


Figura 28: Diagrama de Sequência do Framework Swing

O modelo do Swing é composto por um Diagrama de classe (figura 27) e um Diagrama de

Sequência (figura 28). No diagrama de classe estão representadas as classes que a aplicação pode utilizar, bem como as interfaces fornecidas pelo framework. No Diagrama de Sequência estão representadas as trocas de mensagens interna do framework que ocorrem quando uma imagem é desenhada pelo Swing.

Utilizando este modelo como entrada, juntamente com o modelo da aplicação, obtemos o seguintes diagramas após a execução do método de transformação.

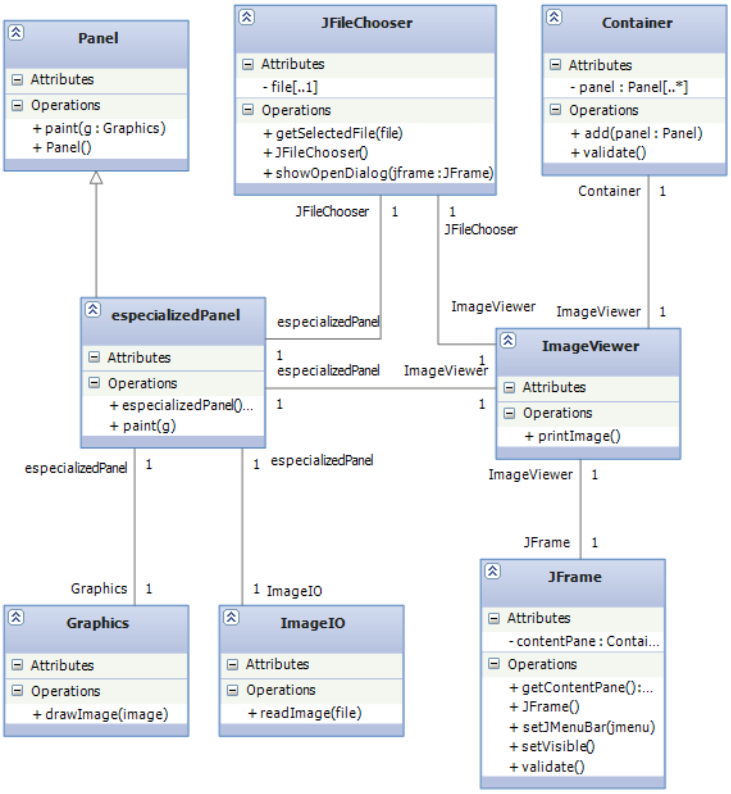


Figura 29: Diagrama de Classes da Aplicação de Visualização de Imagens utilizando o Framework Swing

A figura 29 mostra o Diagrama de Classes obtido após a transformação. Nele estão indicados as classes do framework que serão utilizadas, bem como as novas classes que serão criadas, especializando alguma classe do framework por exemplo.

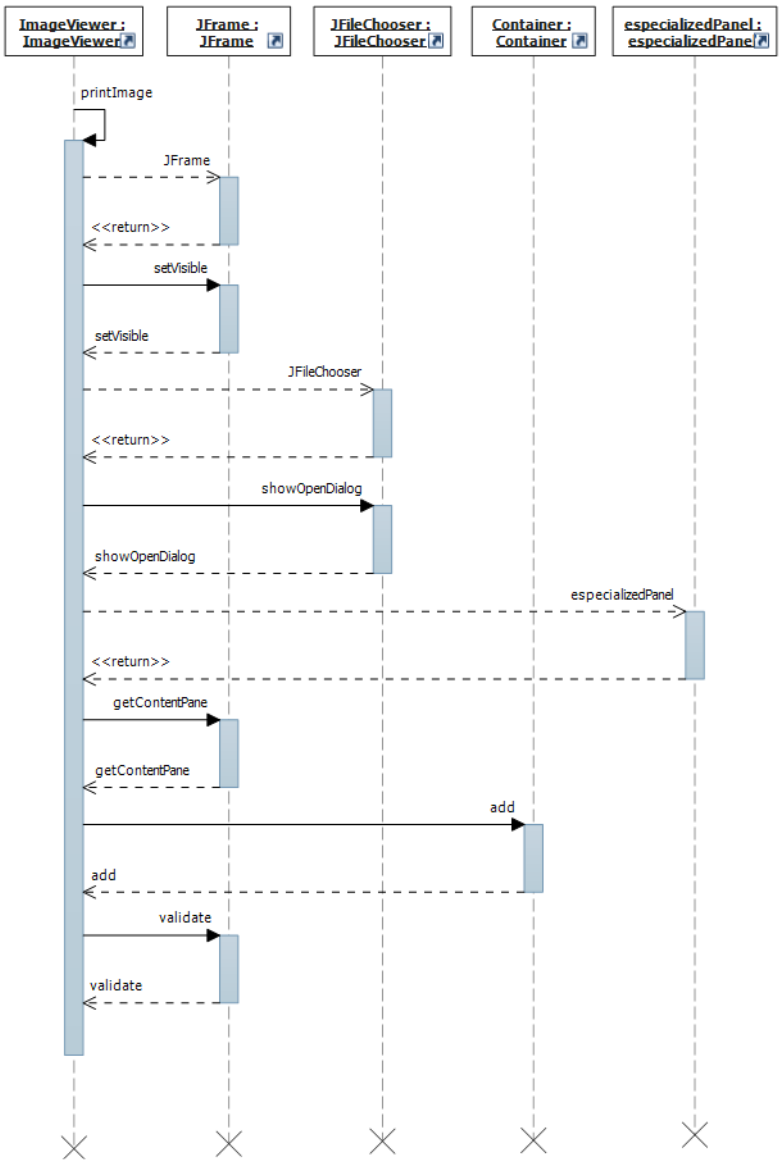


Figura 30: Diagrama de Sequência da Aplicação de Visualização de Imagens utilizando o Framework Swing

A seguir, na figura 30, está representada o Diagrama original da aplicação, alterado de forma a incluir as mensagens para objetos do framework. Além deste Diagrama de Sequência, foram criados dois diagramas adicionais, que descrevem as operações que são invocadas pelo construtor da classe que especializa a classe do Framework "Panel", bem como as operações invocadas pela operação que irá sobre-escrever a operação "paint" desta nova classe. Os diagramas destas novas operações estão apresentados nas figuras 31 e 32 respectivamente.

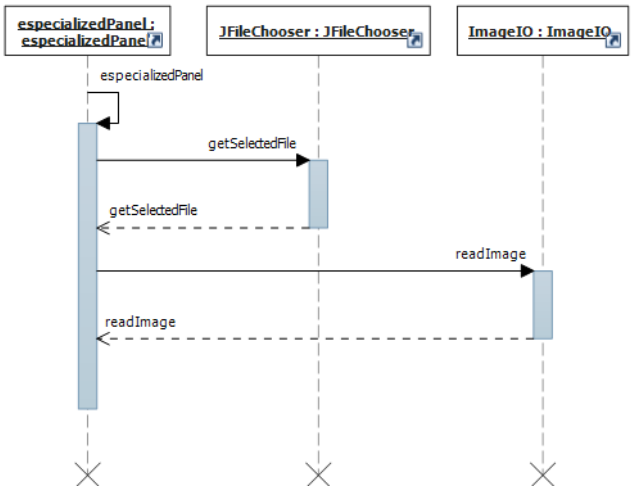


Figura 31: Diagrama de Sequência do Construtor da Classe que especializa a classe Panel do Framework Swing

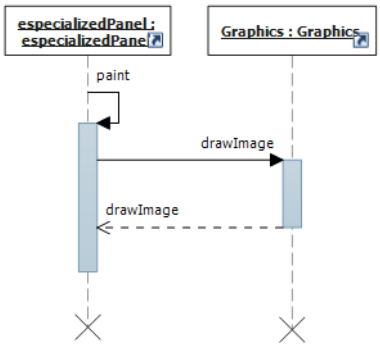


Figura 32: Diagrama de Sequência do Método Paint sobre-escrito

Utilizando este modelo gerado, é possível implementar uma programa em Swing que corretamente apresenta o comportamento esperado pela aplicação.

6.1.2 Blackberry

Para exemplificar uma das vantagens deste método de transformação de modelos, iremos utilizar o framework do Blackberry para gerar um outro modelo, utilizando como base o mesmo modelo de aplicação. O modelo do framework utilizado é apresentado a seguir.

A figura 33, da mesma forma que no caso do Swing, contém as classes disponíveis para a aplicação, bem como as interfaces fornecidas pelo framework. Além do Diagrama de Classe, o modelo contém alguns diagramas de sequência, que descrevem comportamentos internos do framework, que podem ser utilizados pela aplicação.

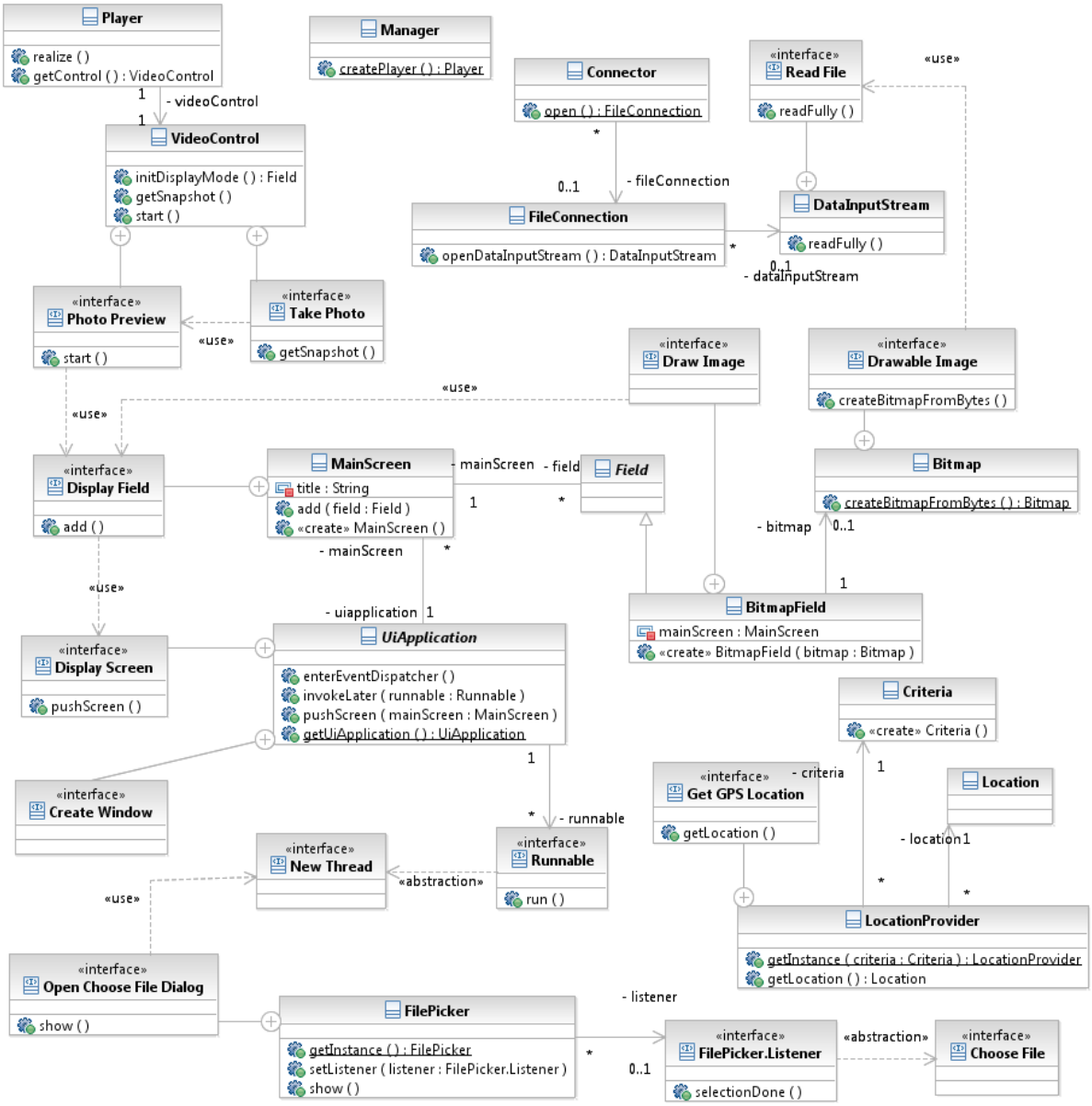
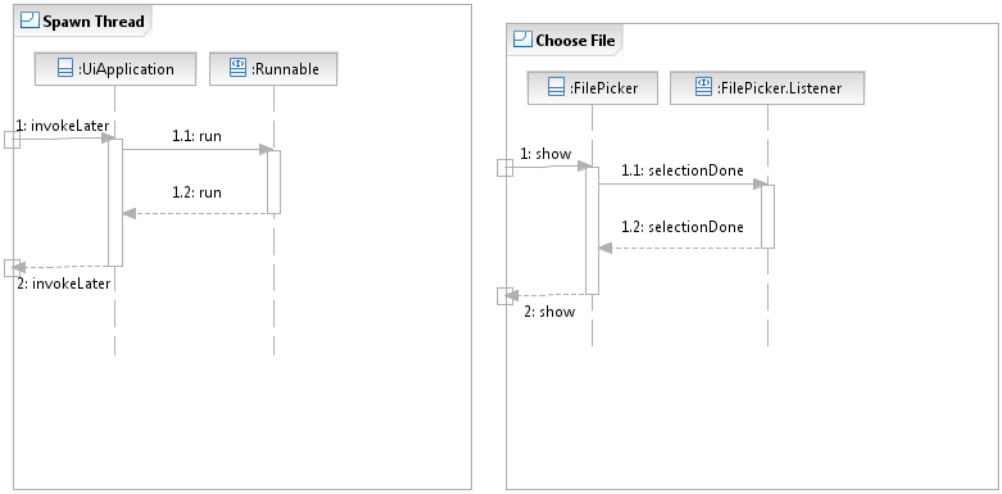


Figura 33: Diagrama de Classes do Framework Blackberry

Além deste diagrama, o modelo contém dois Diagramas de Sequência. Um Diagrama de Sequência (figura 34a) mostra as operações envolvidas na criação de novas *threads*, e o diagrama da figura 34b as operações invocadas durante a escolha de um arquivo.

A partir deste modelo de framework e utilizando como base a mesma aplicação do caso anterior, obtemos o modelo descrito a seguir.



(a) Diagrama de Sequência para Criação de Threads

(b) Diagrama de Sequência para a escolha de Arquivos

Figura 34: Diagrama de Sequência dos Mecanismos Internos do Framework Blackberry

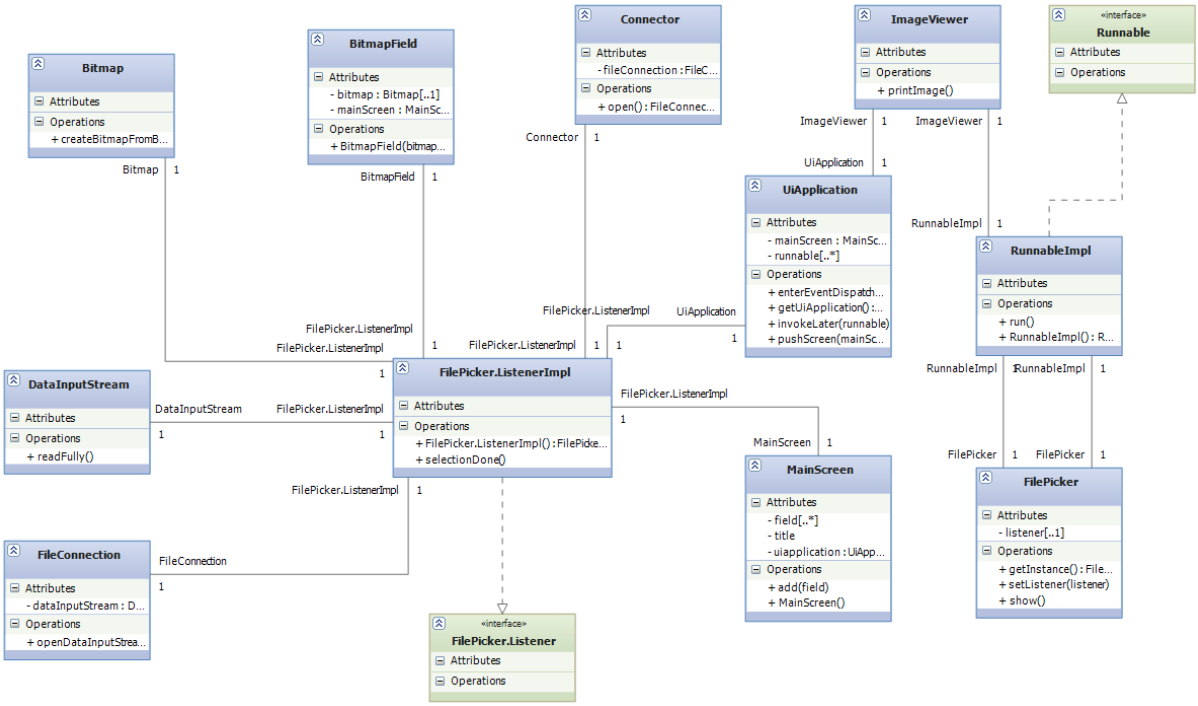


Figura 35: Diagrama de Classes da Aplicação de Visualização de Imagens utilizando o Framework Blackberry

Na figura 35, está apresentado o Diagrama de Classe resultante do método de transformação. Além da classe original da aplicação, ela contém as classes do framework que serão utilizadas, bem como classes novas que devem ser criadas, neste caso, criando realizações de algumas interfaces necessárias pelo framework.

A seguir, na figura 36, é apresentado o Diagrama de Sequência da aplicação alterado, para

incluir as mensagens que serão enviadas à objetos do framework.

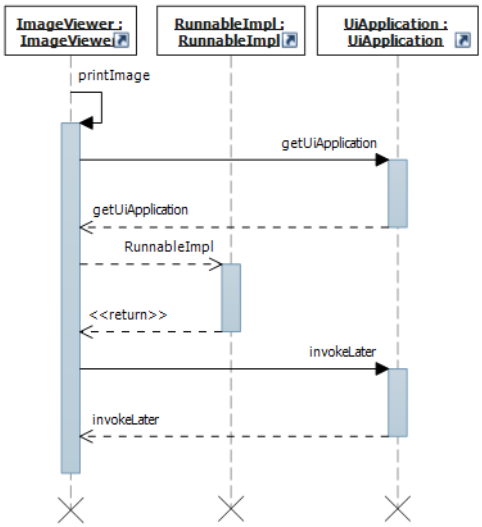


Figura 36: Diagrama de Sequência Principal da Aplicação de Visualização de Imagens utilizando o Framework Blackberry

Além deste diagrama, foi criado mais 3 diagramas de sequência. Estes diagramas representam operações que são invocadas pelos métodos das novas classes criadas. O primeiro destes diagramas, apresentado na figura 37, mostra as operações que são invocadas pelo método "SelectionDone" da classe que implementa a interface "FilePicker.Listener"

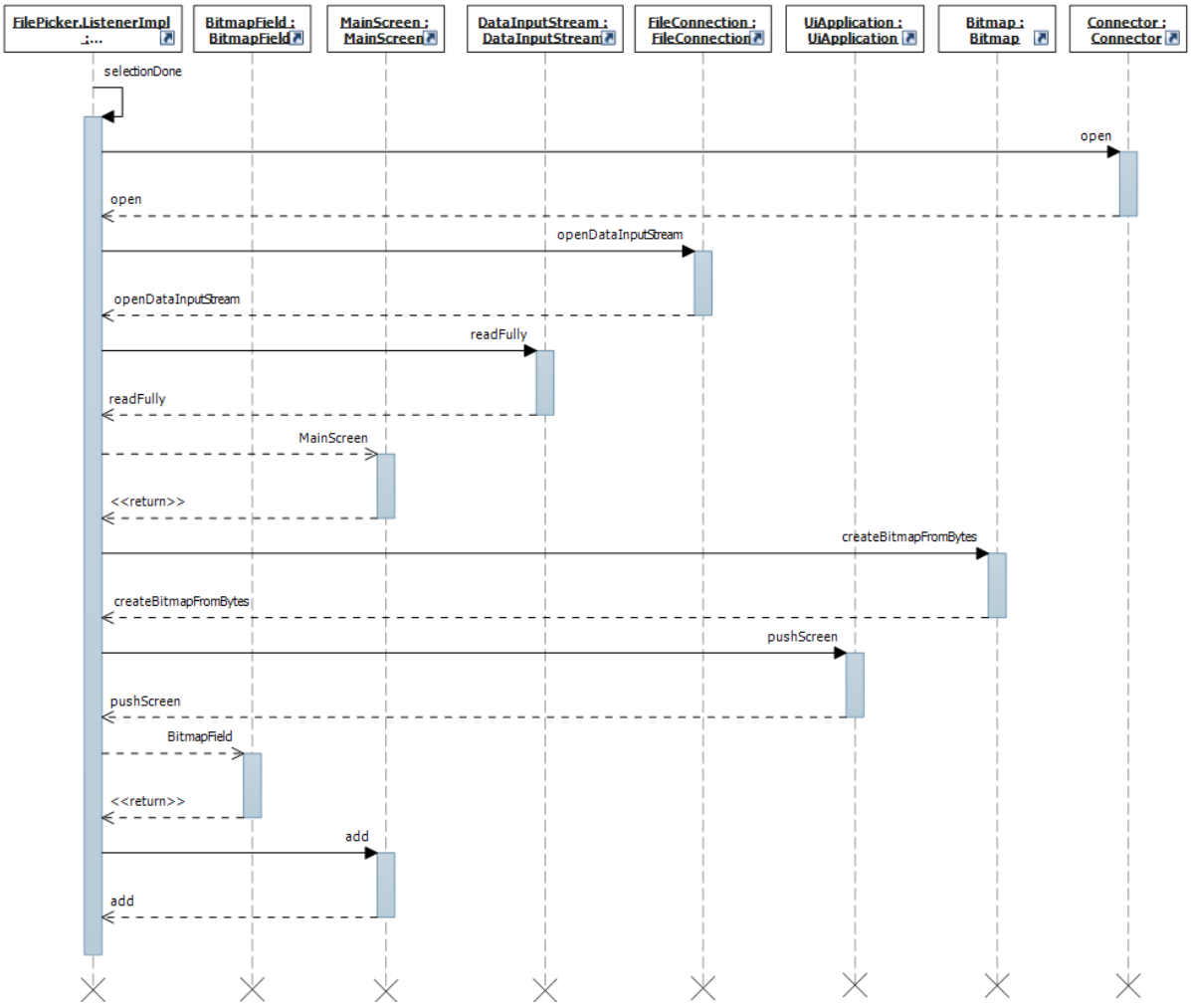


Figura 37: Diagrama de Sequência do método de callback utilizado pelo framework Blackberry

No próximo diagrama, figura 38, as operações invocadas pelo construtor da implementação da interface "Runnable".

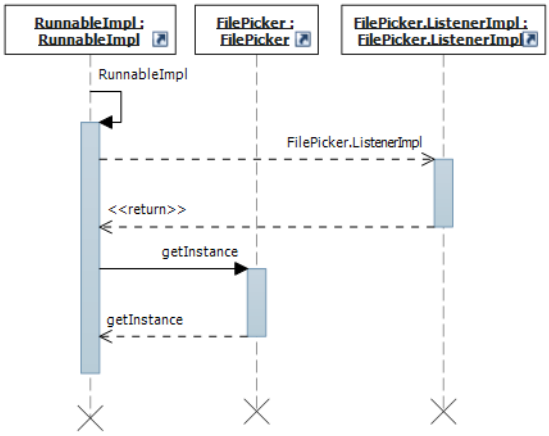


Figura 38: Diagrama de Sequência do contrutor da nova classe que implementa a interface Runnable do framework Blackberry

Por fim, o diagrama que mostra as operações utilizadas pelo método “run” da classe que implementa a interface “Runnable”, indicado na figura 39.

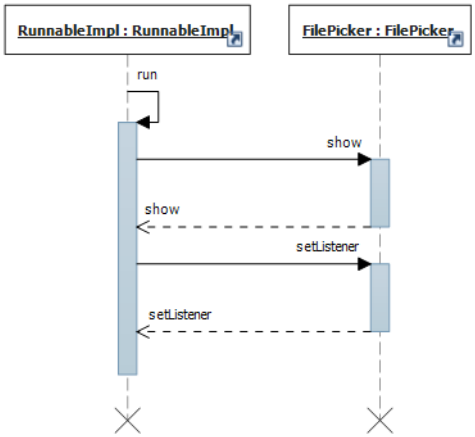


Figura 39: Diagrama de Sequência do método que implementa o método Run da interface Runnable do framework Blackberry

Antes de se utilizar estes diagramas, é necessário alterar o diagrama de classes, de modo que a aplicação especialize a classe ’UiApplication’ do framework, bem como invocar o método ’enterEventDispatcher’. Com exceção destas alterações, o diagrama contém as informações necessárias para se implementar a aplicação. Após estas alterações, é possível utilizar o framework Blackberry corretamente para implementar a aplicação desejada. Além disto, mostra a possibilidade de se o mesmo modelo de aplicação para gerar aplicações em frameworks diferentes.

6.1.3 Android

Para realizar a transformação de modelo, é utilizado um modelo simplificado do framework Android. Este modelo é composto por um Diagrama de Classe, representado na figura 40, e três Diagramas de Sequência. O primeiro indica a sequência de métodos invocados pelo framework quando se deseja tirar um foto (fig. 41), o segundo representa os métodos associados à aquisição da posição através do serviço de gps (fig. 42) e o último diagrama representa a criação de uma ’superfície’, local onde objetos são desenhados pelo framework (fig. 43).

O método de transformação falha quando se utiliza o modelo da aplicação de visualização de imagens e o framework Android. Esta falha ocorre, entretanto, devido a limitações em como os serviços do framework são modelados. Neste caso, o framework fornece, teoricamente, todos os recursos necessários para se implementar a aplicação neste framework. A falha ocorre, porém, porque o framework não fornece um serviço nativo para a escolha de arquivos.

Em ambos os casos em que a transformação foi bem sucedida, com os frameworks Swing e Blackberry, os frameworks possuíam um serviço específico para se realizar a escolha de arquivos, embora fornecido de forma diferente. O mesmo não ocorre no framework Android. Apesar

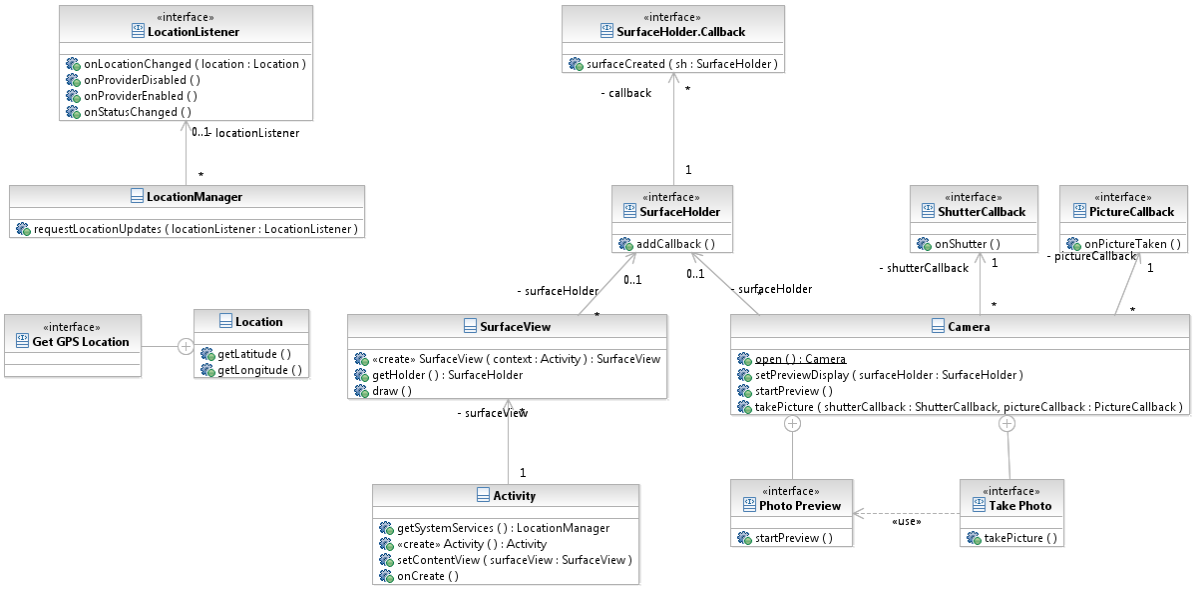


Figura 40: Diagrama de Classes do Framework Android

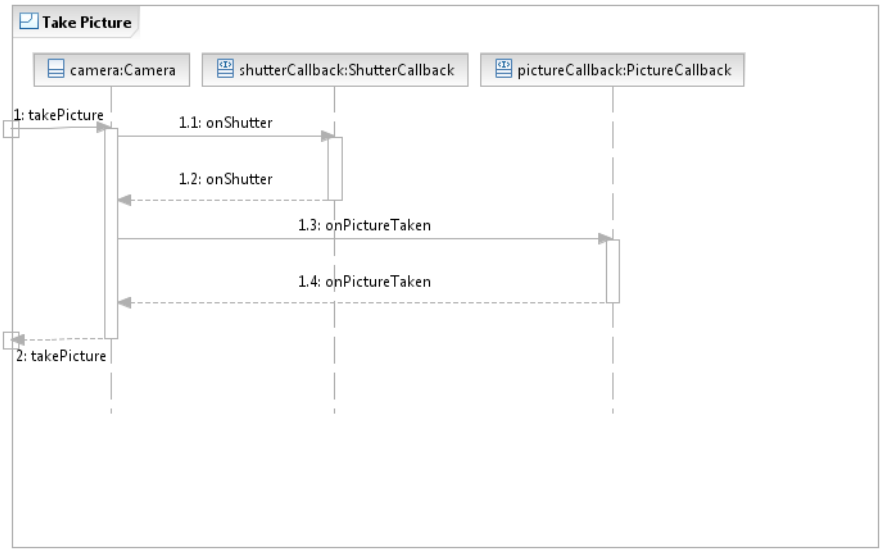


Figura 41: Diagrama de Sequência para tirar Foto com o framework Android

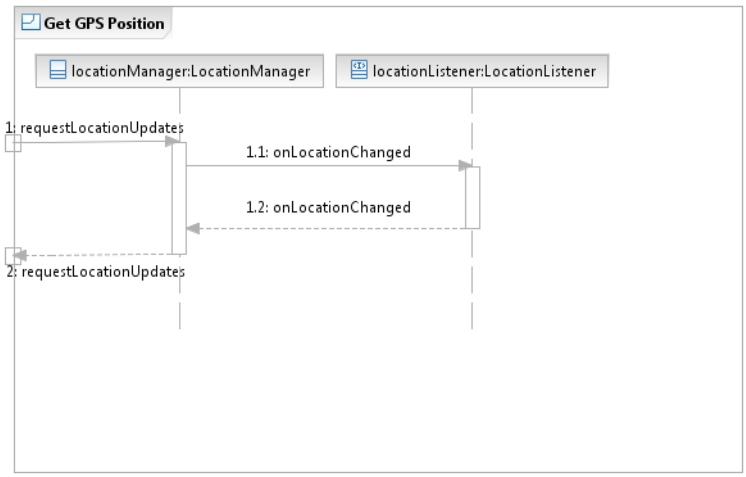


Figura 42: Diagrama de Sequência para adiquirir posição GPS com o framework Android

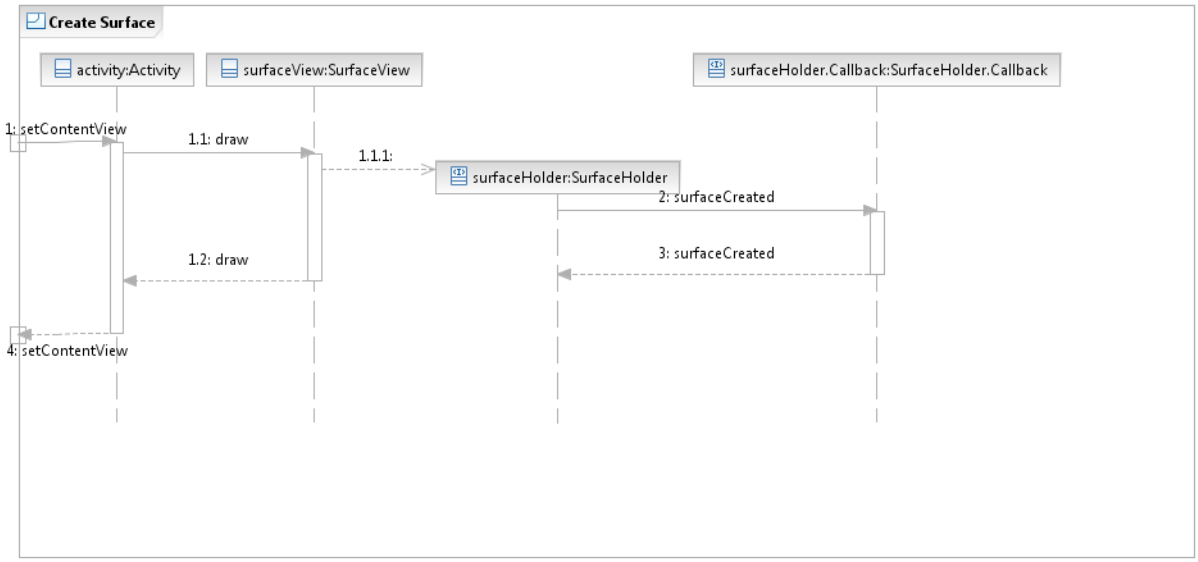


Figura 43: Diagrama de Sequência para criar uma Superfície do framework Android

deste framework fornecer todos os elementos que poderiam ser utilizados para reproduzir o comportamento desejado, não existe um componente nativo que o faça.

Esta ausência de um componente nativo revela uma das limitações do método de transformação. Como mencionado no capítulo 5, os modelos de entrada, tanto da aplicação quanto do framework, utilizam um dicionário de nomes para descrever tanto o comportamento desejado, no caso da aplicação, quanto os serviços fornecidos. Com uso deste dicionário pode-se assumir um comum entendimento sobre o que cada símbolo significa, não sendo necessário então modelar formalmente o comportamento de cada um destes símbolos. Porém, isto limita a capacidade do algoritmo de planejamento de determinar uma sequência de ações, caso a aplicação utilize algum serviço que o framework não forneça nativamente.

Uma forma de se reduzir esta limitação seria incluir no dicionário uma descrição do comportamento de cada um dos símbolos, de forma que o algoritmo de planejamento seja capaz de, caso o framework não fornecer o serviço desejado diretamente, tentar compô-lo utilizando outros serviços disponíveis. Um dos problemas dessa abordagem, porém, é que a própria descrição dos serviços provavelmente usaria um outro conjunto de símbolos para descrever seu comportamento e assim sucessivamente, o que rapidamente levaria a um conjunto de símbolos muito grande para se gerenciar.

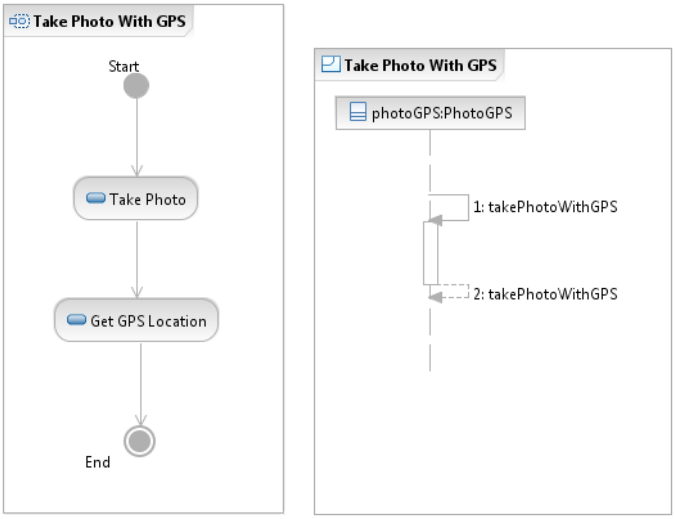
6.2 Foto com dados do GPS

Como segundo caso de uso do método de transformação, consideremos a seguinte aplicação: uma aplicação para *smartphones* que tira uma foto com o aparelho e obtêm a posição atual através do GPS. Da mesma forma que no caso anterior a aplicação será modelada com uma única classe, indicada na figura 44.

Figura 44: Diagrama de Classes da Aplicação de Foto com Informações de GPS

Além do diagrama de classe, o modelo possui o diagrama de atividades, indicando o que se deseja que a aplicação realize, representado na figura 45a bem como o diagrama de sequência, figura 45b indicando qual o método é responsável por iniciar a atividade da aplicação.

Como esta aplicação tem como objetivo ser executado em *smartphones*, foram escolhidos dois frameworks que permitem desenvolver aplicações para estes dispositivos, o framework Android e o framework do Blackberry, sendo este último o mesmo framework utilizado no caso anterior. A seguir serão apresentados os resultados obtidos utilizando este frameworks como o alvo da



(a) Diagrama de Atividade (b) Diagrama de Sequência

Figura 45: Diagramas que descreve o comportamento da aplicação de Foto com Informação de GPS

transformação de modelo.

6.2.1 Android

Utilizando, o modelo deste framework como uma das entradas para o método de transformação, junto com o modelo da aplicação, obteve-se os seguintes resultados. Na figura 46 está representado o diagrama de classe alterado da aplicação. Nele pode-se identificar quais as classes do framework serão utilizadas, bem como o modo como se relacionam.

Além deste diagrama, o modelo contém mais dois quatro Diagramas de Sequência, que indicam quais operações do framework serão utilizadas, bem como a ordem das invocações. O primeiro diagrama, fig. 47 representa as operações que são invocadas pela operação da classe da aplicação original, para iniciar a atividade principal.

A seguir, na figura 48 indica as operações invocadas pelo construtor da classe que irá implementar a interface 'SurfaceHolder.Callback'.

Na figura 49 indica as operações que serão utilizadas pela implementação da operação 'surfaceCreated' da classe que implementa a interface correspondente.

Por fim, a figura 50 indicaria os métodos invocados pela operação que implementa a operação 'locationChanged' da interface correspondente. Porém, como a aplicação não indica quais ações serão tomadas após a aquisição da posição do GPS, esta operação esta vazia. Caso fosse desejada realizar alguma ação após obter a posição, estas ações seriam realizadas dentro deste método.

Utilizando estes diagramas, é possível implementar uma aplicação que utiliza o framework

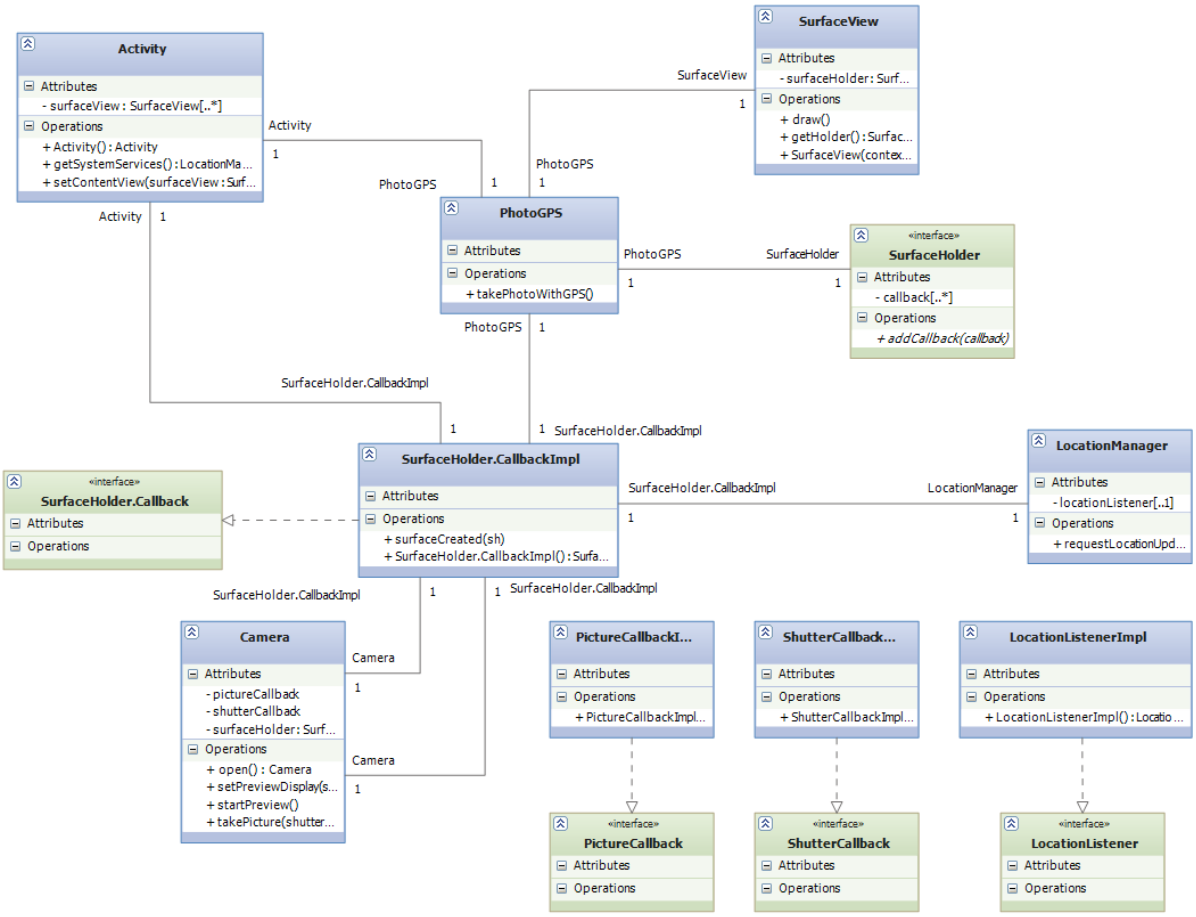


Figura 46: Diagrama de Classes alterado com o framework Android

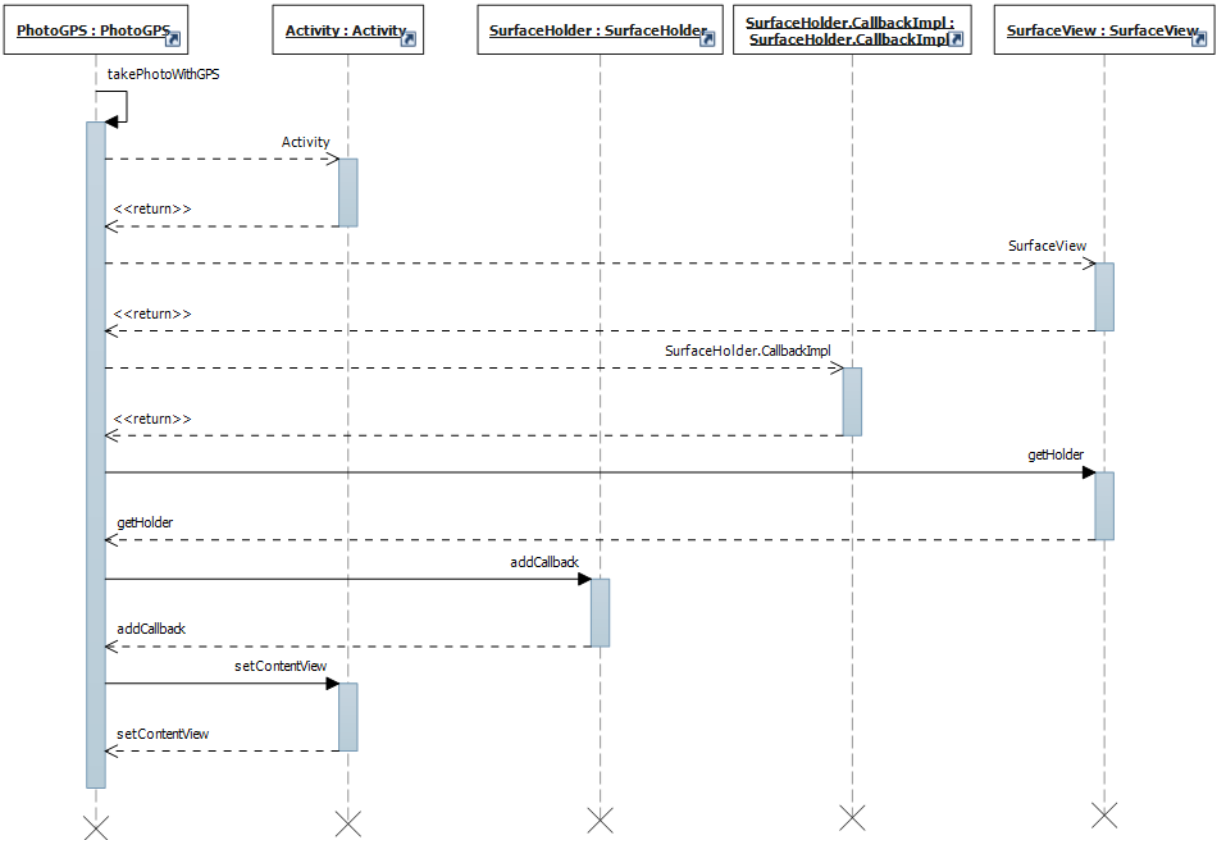


Figura 47: Diagrama de Sequência Principal utilizando o Framework Android

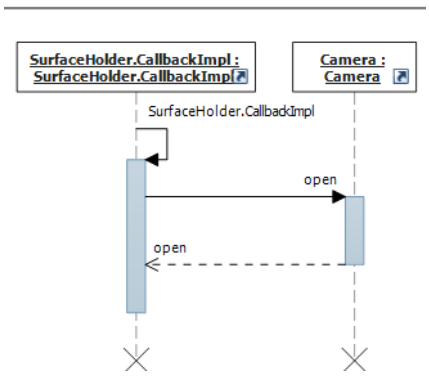


Figura 48: Diagrama de Sequênica do construtor da nova classe

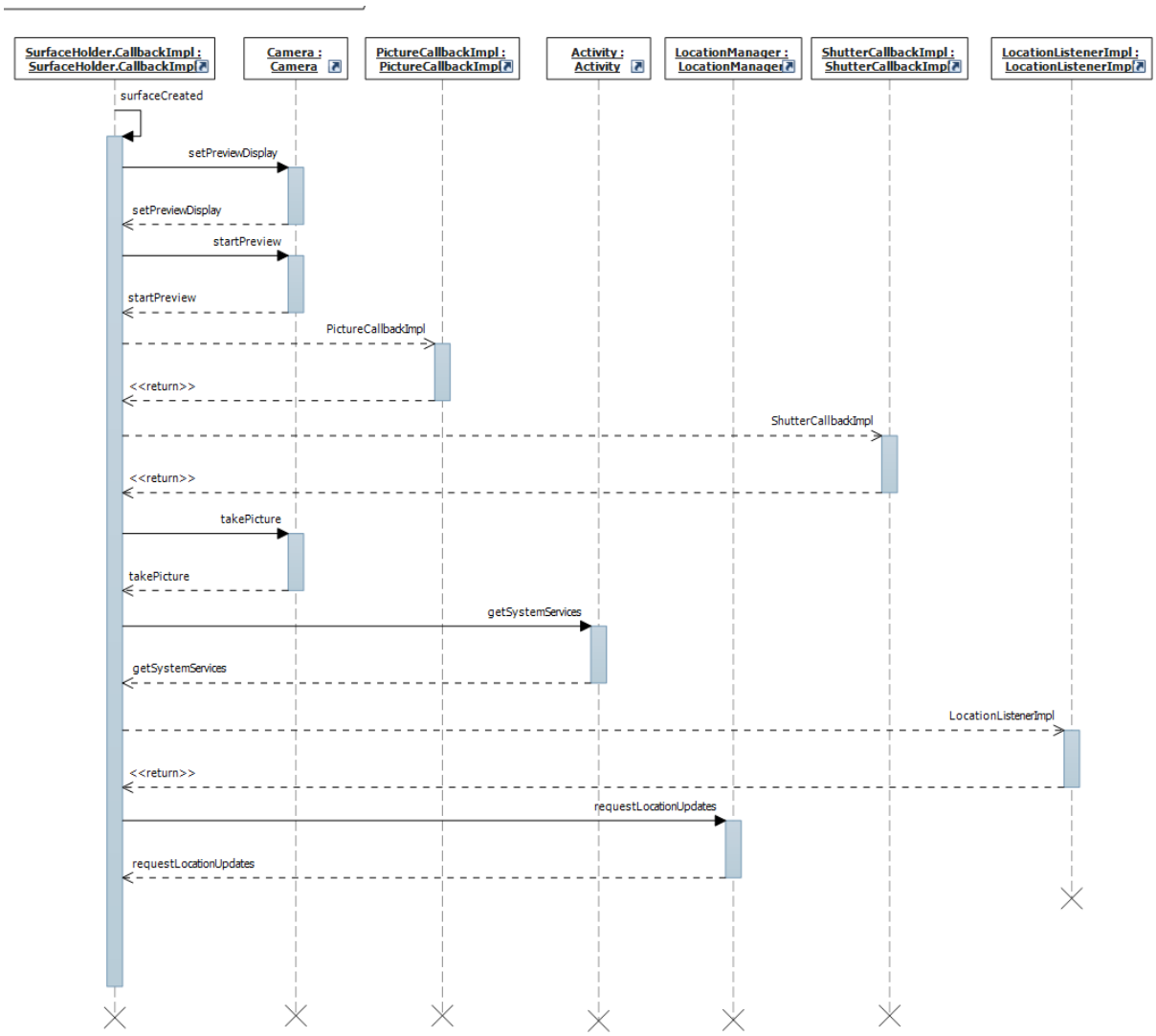


Figura 49: Diagrama de Sequência da Operação de Callback implementada

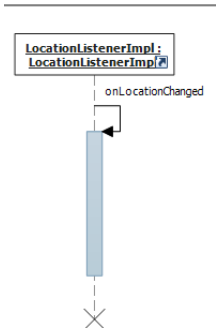


Figura 50: Diagrama de Sequência Vazio

de modo correto, e cumpre os requisitos especificados pelo modelo inicial. Existe uma ressalva, porém, neste caso. Para a aplicação funcionar corretamente, a classe principal deve especializar a classe 'Activity', e é necessário sobre-escrever a operação 'onCreate' deste nova classe de modo que seja chamada a operação principal da classe da aplicação.

6.2.2 Blackberry

Para finalizar este caso, será utilizado o framework Blackberry, por se tratar de um outro framework de desenvolvimento para *smartphones*. O modelo do framework utilizado como entrada para este caso é o mesmo utilizado na seção 6.1.2.

Após a transformação, obtêm-se o Diagrama de Classes indicado na figura 51. Neste caso, como não foi necessário implementar nenhuma interface, nem especializar nenhuma classe, o diagrama contém apenas a classe original e as classes do framework utilizadas.

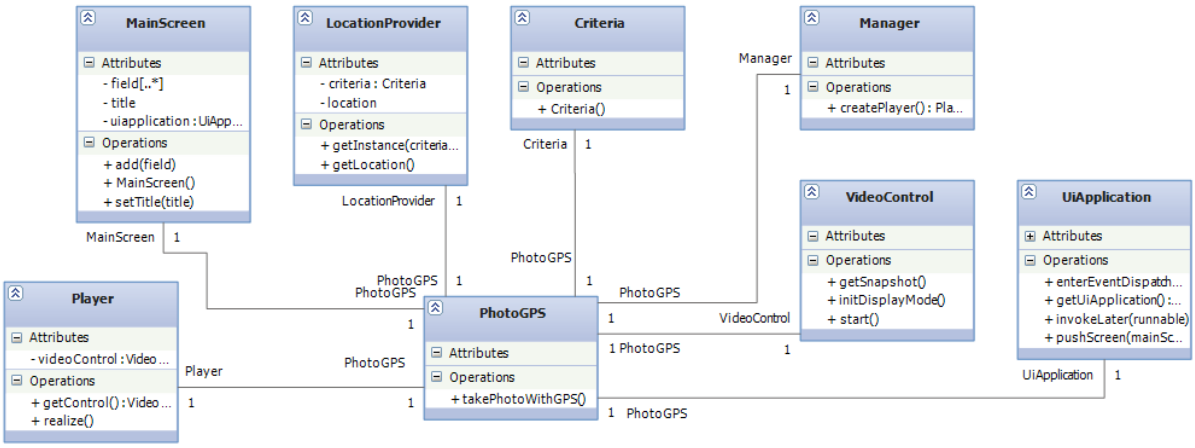


Figura 51: Diagrama de Classes Alterado para utilizar o framework Blackberry

Como neste caso não foi necessário criar nenhuma classe nova, o modelo transformado só possui um Diagrama de Sequência, representando a sequência principal de operações. Este diagrama é apresentado na figura 52.

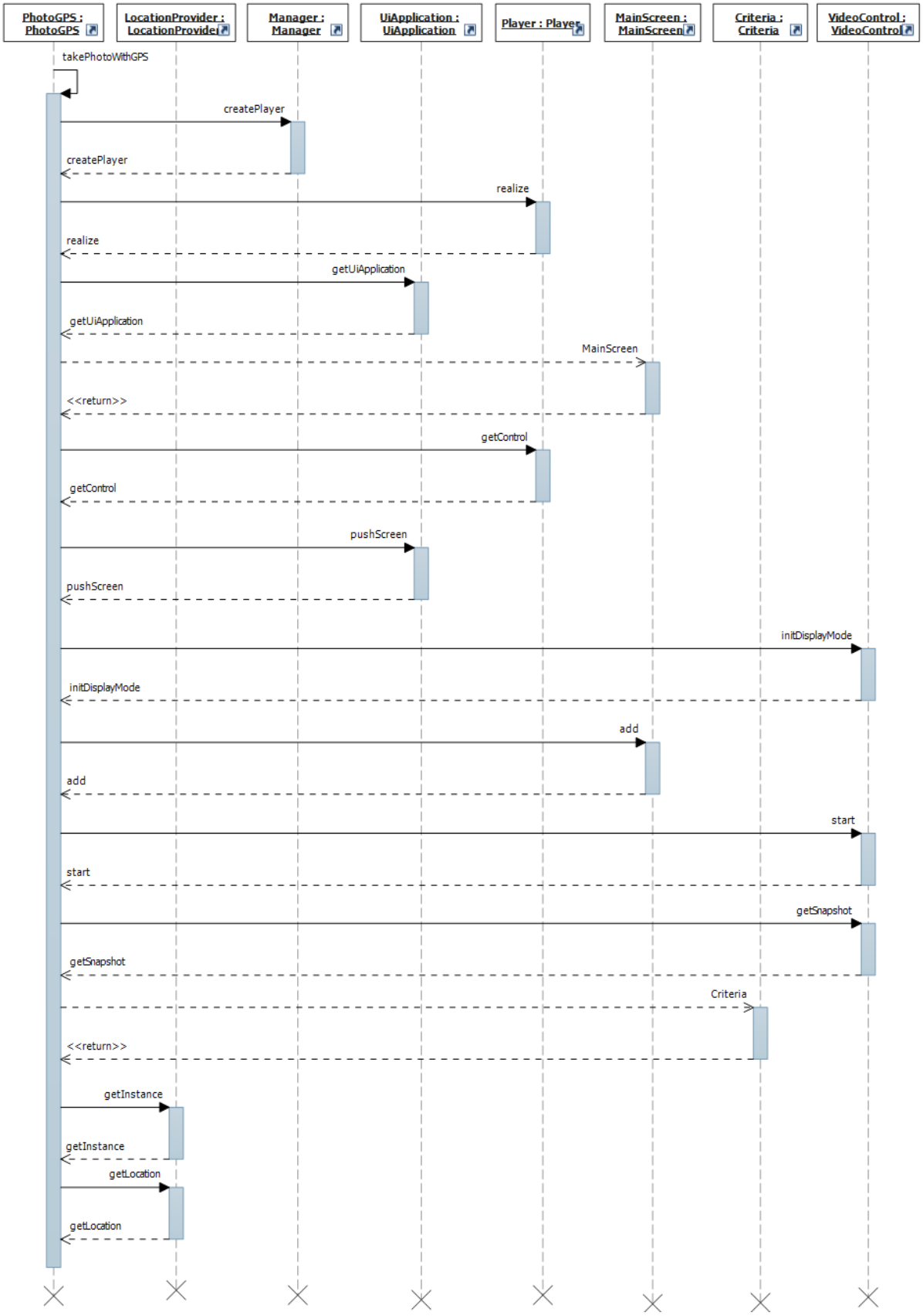


Figura 52: Diagrama de Sequência Principal alterado para se utilizar o Framework Blackberry

De modo similar ao que ocorreu com o resultado do caso 6.1.2, é necessário que a aplicação especialize a classe 'UiApplication' do framework, para que seja executada de forma correta, bem como invocar o método 'enterEventDispatcher. Com exceção destas alterações, o diagrama contém as informações necessárias para se implementar a aplicação.

6.2.3 Swing

Quando se tenta utilizar como entrada o modelo de aplicação para fotos de celular e o framework Swingo método falha. Ao contrário da falha que ocorre quando se utiliza a aplicação ImgViewer no framework Android, esta falha é esperada, uma vez que o framework Swing não fornece nenhuma das funcionalidades necessárias para que a aplicação funcione.

Uma forma de se contornar este problema seri permitir o uso de múltiplos frameworks por uma mesma aplicação. Porém, esta possibilidade gera outros problema, uma vez que seria necessário incluir muito mais informações sobre os frameworks em seus modelos para garantir a compatibilidade entre eles.

7 Considerações Finais

O resultado do projeto de pesquisa que foi realizado é o método de transformação de modelos Framer, que é capaz de, dado um modelo de uma aplicação, identificar quais os componentes de um framework serão utilizados para implementar a aplicação desejada. Para tanto, este método utiliza como guia o planejador lógico de ordem parcial NONLIN.

Nos testes realizados, o Framer foi capaz de identificar num framework os componentes necessários para implementar uma aplicação,ndesde de que os modelos de entrada estejam corretos e completos, o Diagrama de Atividades da aplicação seja descrito como uma sequência linear de serviços, e que os serviços sejam descritos utilizando o dicionário de nomes . O uso deste método possibilita, também, o reuso dos modelos de entrada, tanto da aplicação quanto do framework. A possibilidade de se reutilizar o modelo do framework reduz o custo de aprendizado por outras pessoas, enquanto que o reuso do modelo da aplicação reduz o custo de desenvolvimento, além de auxiliar o desenvolvimento de uma mesma aplicação para diferentes plataformas. A utilização deste método pode trazer outras vantagens. Uma delas é que, devido a necessidade de se modelar corretamente tanto o framework quanto a aplicação, isto leva a um uso correto do framework, o que pode reduzir a quantidade de erros na aplicação.

Porém, o modelo gerado apesar de correto, em alguns casos, não é completo. Devido a certas imposições de uso do framework, em especial ao modo como as classes principais devem ser tratadas, o modelo gerado pelo método de transformação não possuía todas as informações necessárias para a utilização do framework. As informações sobre a inicialização do framework, bem como eventuais configurações, não são contempladas atualmente pelo método de transformação.

Existe um outro ponto que vale ressaltar. Enquanto que o modelo gerado utilizava de forma correta o framework, não existe uma preocupação com elementos de boas práticas de desenvolvimento de software, como por exemplo clareza do código resultante ou desacoplamento entre as classe. O objetivo é apenas gerar código funcional.

Como mencionado no capítulo 6, a utilização de um dicionário de nomes gera limitações nos casos em que se pode utilizar o método. Para eliminar o uso deste dicionário, ou pelo menos reduzir a dependência do método dele, é necessário investigar outras formas para se

modelar a parte comportamental, tanto da aplicação quanto do framework, para que não seja tão dependente do uso de linguagem natural.

Apesar destas fatos, podemos considerar que os objetivos propostos foram alcançados. Porém, trabalhos futuros são necessários para investigar se os problemas encontrados são devido ao método de transformação ou à limitações da UML. Independente de onde estiver a causa destes problemas em particular, é necessário investigar que modelos alterados seriam obtidos se fosse utilizado um algoritmo de planejamento mais complexo. Precisa-se, também, investigar outros meios de descrever as interfaces dos modelos, para eliminar a necessidade de utilizar nomes fixos. Além disto, é necessário também realizar testes para verificar a viabilidade do uso do método em larga escala, para determinar como o método se comporta com aplicações e frameworks mais complexos.

Referências

[1] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *Unified Modeling Language User Guide, The (2nd Edition) (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005. Hardcover. ISBN 0321267974. Disponível em: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0321267974>.

[2] ANDROID developers. `http://developer.android.com/index.html`. Acesso em Junho de 2012.

[3] JMONKEY Game Engine. `http://www.jmonkeyengine.com/`. Acesso em Junho de 2012.

[4] FAYAD, M. E.; SCHMIDT, D. C. Object-Oriented Application Frameworks. *Communications of the ACM*, v. 40, n. 10, p. 32–38, 1997.

[5] BERTOLI, P.; PISTORE, M.; TRAVERSO, P. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence*, 2009. ISSN 00043702. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0004370209001489>.

[6] BUSINESS Process Execution Language for Web Services version 1.1. `http://www.ibm.com/developerworks/library/specification/ws-bpel/`. Acesso em Junho de 2012.

[7] KICHKAYLO, T.; IVAN, A.; KARAMCHETI, V. Constrained component deployment in wide-area networks using AI planning techniques. In: *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. [S.l.]: IEEE Comput. Soc, 2003. p. 10. ISBN 0-7695-1926-1.

[8] BERARDI, D. et al. Automatic service composition based on behavioral descriptions. *International Journal of Cooperative Information Systems*, Citeseer, v. 14, n. 4, p. 333–376, 2005.

[9] MARTÍNEZ, E.; LESPéRANCE, Y. Web service composition as a planning task: Experiments using knowledge-based planning. In: *Proceedings of the ICAPS-2004 Workshop on Planning and Scheduling for Web and Grid Services*. [S.l.: s.n.], 2004. p. 62–69.

[10] RIABOV, A.; LIU, Z. Planning for stream processing systems. In: *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*. [S.l.]: Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005. v. 20, n. 3, p. 1205.

[11] OBJECT Management Group. `http://www.omg.org/`. Acesso em Junho de 2012.

[12] GROUP, O. M. *Object Constraint Language*. `http://www.omg.org/technology/documents/formal/ocl.htm`. Acesso em Junho de 2012.

[13] KLEPPE, A.; WARMER, J. Extending OCL to include actions. *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, v. 1939/2000, p. 440–450, 2000. Disponível em: <http://www.springerlink.com/index/ce1khnf4gm1q0w6h.pdf>.

[14] JOHNSON, R. E. Frameworks = Patterns + Components. *Communications of the ACM*, v. 40, n. 10, p. 39–42, 1997.

[15] KRAJNC, A.; HERICKO, M. Classification of object-oriented frameworks. *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, Ieee, p. 57–61, 2003. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1248147>.

[16] GROUP, O. M. *Model Driven Architecture*. 2001. `http://www.omg.org/mda`. Acesso em Junho de 2012.

[17] CZARNECKI, K.; HELSEN, S. Classification of model transformation approaches. In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. [S.l.]: Citeseer, 2003. p. 1–17.

[18] ENGELS, G. et al. Uml collaboration diagrams and their transformation to java. In: *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*. Berlin, Heidelberg: Springer-Verlag, 1999. (UML'99), p. 473–488. ISBN 3-540-66712-1. Disponível em: <http://dl.acm.org/citation.cfm?id=1767297.1767347>.

[19] CSERTáN, G. et al. VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. *Automated Software Engineering*, p. 15–18, 2002.

[20] MELLOR, S. J.; BALCER, M. *Executable UML: A Foundation for Model-Driven Architectures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. Foreword By-Jacoboson, Ivar. ISBN 0201748045.

[21] GUENNEC, A. L.; SUNYé, G.; JEZEQUEL, J.-M. Using UML Action Semantics for model execution and transformation. *Information Systems*, v. 27, n. 6, p. 445–457, set. 2002. ISSN 03064379. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0306437902000145>.

[22] UML 1.4 with Action Semantics, Final Adopted Specification. `http://www.omg.org/cgi-bin/doc?ptc/02-01-09`. Acesso em Junho de 2012.

[23] SCHöNBERGER, S.; KELLER, R. K.; KHRISS, I. Algorithmic support for model transformation in object-oriented software development. *Concurrency and Computation: Practice and Experience*, John Wiley & Sons, Ltd., v. 13, n. 5, p. 351–383, 2001. ISSN 1532-0634. Disponível em: <http://dx.doi.org/10.1002/cpe.555>.

[24] VARRó, D. Model Transformation by Example. In: NIERSTRASZ, O. et al. (Ed.). *Model Driven Engineering Languages and Systems*. Springer Berlin / Heidelberg, 2006, (Lecture Notes in Computer Science, v. 4199). p. 410–424. Disponível em: <http://dx.doi.org/10.1007/11880240_29>.

[25] BERARDI, D.; CALVANESE, D.; GIACOMO, G. D. Reasoning on UML class diagrams. *Artificial Intelligence*, v. 168, p. 70–118, 2005.

[26] KUSTER, J.; HECKEL, R.; ENGELS, G. Defining and validating transformations of UML models. *IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003*, Ieee, p. 145–152. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1260218>.

[27] BISZTRAY, D.; HECKEL, R.; EHRIG, H. Compositionality of Model Transformations. *Electronic Notes in Theoretical Computer Science*, v. 236, p. 5–19, abr. 2009. ISSN 15710661. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S1571066109000905>.

[28] CALDIERA, G.; BASILI, V. R. Identifying and qualifying reusable software components. *Computer*, v. 24, n. 2, p. 61–70, 1991. ISSN 0018-9162.

[29] PARISI-PRESICCE, F. A rule-based approach to modular system design. In: *Software Engineering, 1990. Proceedings., 12th International Conference on*. [S.l.: s.n.], 1990. p. 202–211.

[30] BERTOLI, P.; PISTORE, M.; TRAVERSO, P. Automated composition of Web services via planning in asynchronous domains. *Artificial Intelligence*, Elsevier B.V., v. 174, n. 3-4, p. 316–361, mar. 2010. ISSN 00043702. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/S0004370209001489>.

[31] RUSSELL, S.; NORVIG, P. In: _____. *Artificial Intelligence - A modern approach 2nd Edition*. Upper Saddle River, New Jersey: Pearson Education., Inc., 2003. cap. 11. ISBN 0137903952.

[32] FIKES, R. E.; NILSSON, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, v. 2, n. 3-4, p. 189–208, 1971. ISSN 00043702. Disponível em: <http://linkinghub.elsevier.com/retrieve/pii/0004370271900105>.

[33] TATE, A. Generating project networks. In: *Proceedings of the 5th international joint conference on Artificial intelligence - Volume 2*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1977. p. 888–893.

[34] JOUAULT, F.; KURTEV, I. Transforming models with ATL. *Satellite Events at the MoDELS 2005 Conference*, v. 3844/2006, p. 128 – 138, 2006. Disponível em: <http://www.springerlink.com/index/7143g735r4j59463.pdf>.

[35] ATL Integrated Environnement. `http://www.eclipse.org/atl/`. Acesso em Junho de 2012.

[36] SWING (Java Foundation Classes). `http://docs.oracle.com/javase/6/docs/technotes/guides/swing/index.html`. Acesso em Junho de 2012.

[37] BLACKBERRY Developer Zone. `http://na.blackberry.com/eng/developers/`. Acesso em Junho de 2012.

APÊNDICE A – Código Fonte e Modelos Utilizados nos Testes

Este trabalho também produziu uma ferramenta Open Source, que implementa o método Framer descrito no capítulo 5. O código da ferramenta está disponível no site: "http://code.google.com/p/mt-framer/".

O Framer necessita que uma máquina virtual JAVA esteja instalada e alguma IDE Java. Por exemplo, para executar o Framer no Eclipse, basta criar um projeto Java vazio, importar os arquivos para o projeto e selecionar "Run". Se for pedido para indicar classe principal, selecionar a classe Framer.

Ao ser iniciado, será apresentada a tela da figura 53. Para utilizá-lo, deve-se primeiro selecionar o modelo da aplicação que será utilizado (figura 54). A seguir, selecionar o modelo do framework (figura 55). Uma vez com os dois modelos selecionados, clicar em "Run" (figura 56) e o método será executado. Após a execução, aparecerá um botão com a opção de salvar o modelo gerado (figura 57).

Para visualizar o modelo gerado, é necessário utilizar uma ferramenta de Modelagem UML, que suporte o formato XMI.

Além do código fonte e bibliotecas necessárias para se utilizar a ferramenta, o projeto contém também os modelos de testes utilizados neste trabalho. Os modelos dos frameworks Swing, Android e Blackberry estão contidos nos arquivos SwingFrame.uml, AndroidFramework.uml e BlackFramework.uml respectivamente. Além dos modelos do framework, estão inclusos os modelos das aplicações ImgViewer e PhotoGPS, contidas nos arquivos ImgViewerApp.uml e

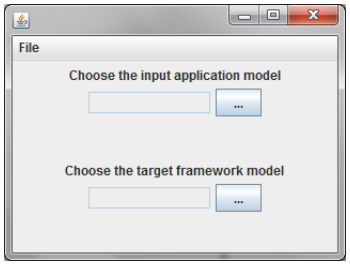


Figura 53: Tela Inicial do Framer

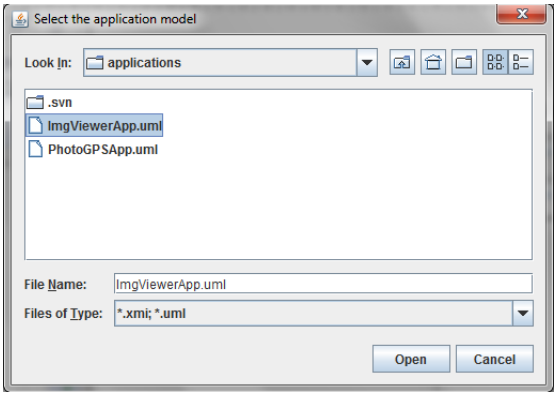


Figura 54: Tela de Seleção do Modelo da Aplicação

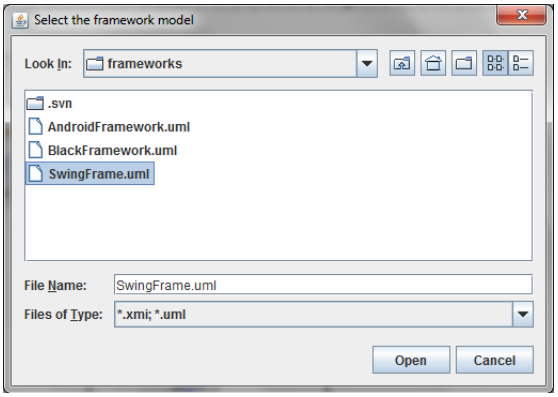


Figura 55: Tela de Seleção do Modelo do Framework

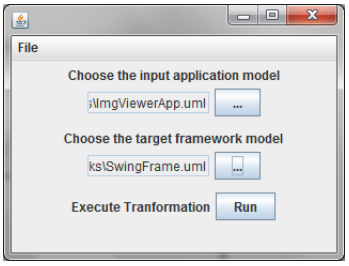


Figura 56: Framer configurado para ser utilizado

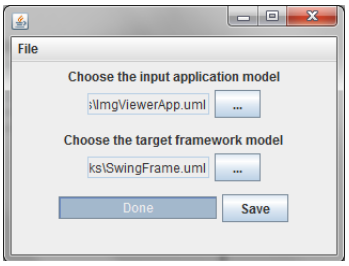


Figura 57: Tela após a execução do método

PhotoGpsApp.uml respectivamente. Todos os modelos disponibilizados estão no formato XMI, sendo que se for necessário visualizá-los, será necessário utilizar uma ferramenta de modelagem UML que suporte a importação de arquivos XMI.