Uma Abordagem de Reestruturação de Sistemas Baseada em Requisitos de Qualidade Pré-Estabelecidos

RELATÓRIO CIENTÍFICO PARCIAL - 01/05/2013 a 28/02/2015
Apresentado à Fundação de Amparo à Pesquisa do Estado de São Paulo - FAPESP

Número do Processo:	FAPESP 2012/05168-4
Período:	Maio/2013 a Fevereiro/2015

Bolsista:	Rafael Serapilha Durelli (rdurelli@icmc.usp.br)
Orientador:	Prof. Dr. Márcio Eduardo Delamaro (delamaro@icmc.usp.br)

USP - São Carlos
Fevereiro de 2015

Resumo

Relatório Científico Parcial apresentado à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) com o objetivo de elucidar as atividades realizadas pelo bolsista Rafael Serapilha Durelli durante o segundo período de vigência da bolsa concebida sob o Processo Número 2012/05168-4. O referido período teve inicio em Maio de 2013 e foi finalizado em Fevereiro de 2015. Além disso, este relatório também descreve as atividades que foram finalizadas, as atividades que estão em andamentos, bem como as atividades a serem realizadas no próximo período. Durante o período de vigência anterior, o bolsista conduziu atividades voltadas principalmente à obtenção de uma ampla visão da literatura científica sobre os principais temas de pesquisa relacionado ao projeto em questão. Para tal, uma revisão sistemática foi conduzida. Os resultados da revisão forneceram evidências que suportam e motivam a realização do projeto proposto. A condução da revisão sistemática bem como os principais resultados foram apresentados no relatório anterior. Vale ressaltar que no segundo período de vigência da bolsa, correspondente a este relatório, cinco principais atividades foram conduzidas. Tais atividades são: (*i*) redação de artigos, (*ii*) adaptação de um catalogo de refatoração para o metamodelo Knowledge Discovery Metamodel (KDM), (*iii*) implementação de uma ferramenta semi-automática que fornecer suporte ao catalogo de refatoração adaptado para o KDM, (*iv*) criação de um metamodelo de refatorações, padronizado que contenha características similares ao KDM, ou seja, independente de plataforma e linguagem, com o principal objetivo de auxiliar o engenheiro de modernização durante a elaboração de refatorações e (*v*) definição de uma Linguagem Especifica de Domínio (do inglês Domain-Specific Language - DSL) para auxiliar a instanciação do metamodelo de refatoração. Esse relatório apresenta as atividades desenvolvidas no período de Maio/2013 a Fevereiro/2015.

1 Introdução

Este relatório tem por objetivo apresentar as atividades realizadas pelo bolsista Rafael Serapilha Durelli durante o período de Maio/2013 a Fevereiro/2015, referente à bolsa de doutorado concebida pela Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) sob o Processo Número 2012/05168-4.

É importante salientar que o trabalho em questão tem sido desenvolvido no Departamento de Ciências da Computação e Estatística do Instituto de Ciência Matemáticas e de Computação (ICMC) da Universidade de São Paulo (campos São Carlos/SP). Este trabalho se insere no contexto do grupo de pesquisas em Engenheira de Software, sob a orientação do Prof. Dr. Márcio Eduardo Delamaro. Além disso, é importante salientar que este trabalho esta sendo executado em colaboração com o grupo de engenharia de software da Universidade Federal de São Carlos (UFSCAR)[1]. Mais especificamente em colaboração com o Prof. Dr. Valter Vieira de Camargo[2], o qual tem grande experiência na área de engenharia de software com ênfase no desenvolvimento de frameworks no contexto da programação orientada a aspectos e reuso de software. Ressalta-se que o bolsista criou um vínculo científico com o *Institut National de Recherche en Informatique et en Automatique* (INRIA), onde realizou um ano de doutorado sanduíche sobre orientação do Prof. Dr. Nicolas Anquetil[3] o qual tem grande experiência na área de manutenção e reengenharia de software. O doutorado sanduíche em questão foi realizado em Maio de 2013 até Março de 2014.

Durante o período concernente a este relatório, o bolsista dedicou-se às atividades técnicas requeridas para concretização do seu projeto de doutorado bem com às atividades exigidas pelo Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional do ICMC[4]. Considerando as atividades técnicas previstas no cronograma para o período vigente, todas foram devidamente realizadas ou estão progredindo de acordo com o estipulado, a saber: (*i*) redação da monografia e aprovação no exame de qualificação, (*ii*) implementação da abordagem proposta no exame de qualificação, (*iii*) redação de artigos. O exame de proficiência em língua inglesa, exigido pelo Programa de Pós-Graduação do ICMC, foi devidamente realizado e uma pontuação satisfatória foi obtida.

[1] `http://dc.ufscar.br`
[2] `http://buscatextual.cnpq.br/buscatextual/visualizacv.do?id=S819089`
[3] `http://rmod.lille.inria.fr/web/pier/team/Nicolas-Anquetil`
[4] Destaca-se que algumas atividades já foram reportadas no relatório anterior

1.1 Convenções adotadas neste relatório

Ao longo deste relatório, *Itálico* é utilizado para dar ênfases, introduzir novos termos e palavras em inglês. `Typewriter` é utilizado para operador Java, operador da DSL, palavras chaves, nome de métodos, variáveis e URL que aparecem no texto. Símbolos ❶, ❷, ❸, ❹ ou ⓐ, ⓑ, ⓒ, ⓓ, são utilizados para chamar a atenção do leitor para informações importantes em figuras e códigos.

1.2 Estrutura do relário

Neste relatório enfatiza-se adaptação de um catalogo de refatoração para o metamodelo *Knowledge Discovery Metamodel* (KDM). Além disso, neste relatório também é enfatizado a implementação de uma ferramenta semi-automática que fornecer suporte ao catalogo de refatoração adaptado para o KDM. Em seguida é apresentado a criação de um metamodelo de refatorações, padronizado que contenha características similares ao KDM, ou seja, independente de plataforma e linguagem. E a definição de uma Linguagem Especifica de Domínio (do inglês *Domain-Specific Language - DSL*) para auxiliar a instanciação do metamodelo de refatoração.

Organização do relatório

2 Plano de Trabalho

Hoje em dia diversas companhias estão enfrentando problemas de gerenciamento, manutenção e/ou troca (de parte) de um sistema existente. Esses sistemas geralmente são caracterizados como sistemas legados (SL). SL geralmente são aplicações de grande porte que possuem um papel crucial e importante no contexto de gerenciamento de informação de companhias. Melhorar a compreensão desses SLs (e.g., arquitetura, características, acoplamento, modularização, etc) é o fator principal durante a evolução/modernização desses sistemas. O processo de obtenção de uma representações de alto nível de um determinado SL é chamado de Engenharia Reversa (ER).

Diferentemente da Engenharia Avante (EA), a ER é comumente definida como o processo de examinar um SL para representa-lo formalmente em um modelo de alto nível de abstração (Chikofsky e Cross, 1990). O principal objetivo para realizar esse processo é para facilitar o entendimento do estado atual do SL. Por exemplo, utilizando esse modelo de alto nível de abstração é possível corrigir possíveis erros, adicionar novas características, reutilizar partes do SL em outros sistemas, ou até mesmo moderniza-lo completamente (Griffith et al, 2011). De acordo com Canfora et al (2011), isto está acontecendo com mais frequência nos dias atuais, em função da necessidade de não só satisfazer novas exigências e expectativas dos usuários, mas também para a adaptação dos SLs para modelos de negócios emergentes, aderindo à mudança da legislação, lidar com a inovação tecnológica (frameworks, *Application Programming Interface* (API), ambientes de desenvolvimento, etc) e ainda preservar a estrutura do sistema para que o mesmo não se deteriore.

Claramente, uma vez que a ER é um processo demorado e sujeito a erros, qualquer solução que auxilie (semi)automaticamente o processo de ER traria ajuda para os engenheiros de software/modernização e, assim, facilitaria sua utilização (Perez-Castillo et al, 2011b; Ulrich e Newcomb, 2010a). No entanto, essa solução teria de enfrentar vários problemas, a saber: (*i*) heterogeneidade técnica dos sistemas legados; (*ii*) complexidade estrutural destes sistemas legados; (*iii*) escalabilidade da solução desenvolvida; e (*iv*) edaptabilidade.

Na década de 90 algumas pesquisas tinham como intuito desenvolver soluções (semi)automática para auxiliar a ER. No entanto, tais soluções eram focadas em tecnologias orientadas a objectos (OO) (Smith e Nair, 1992). Também surgiu-se o interesse em processos e ferramentas para auxiliar a compreensão de programas desenvolvidos em OO. Entre muitas propostas, algumas focaram na extração e análise de informações relevantes a partir do código fonte ou componentes de software (Canfora et al, 1994), enquanto outras focaram em banco de dados relacionais (Premerlani e Blaha, 1993), código compilado ou arquivos binários (Eilam, 2005), etc. No entanto, estas pesquisas eram bastante específicas para uma tecnologia em particular ou um determinado cenário de ER (por exemplo, migração técnica, análise de software).

Com o surgimento de *Model-Driven Engineering* (MDE) (Kent, 2002), suas diretrizes e técnicas fundamentais tem sido utilizadas para auxiliar a construção de soluções eficazes de ER, ou seja, *Model-Driven Reverse Engineering* (MDRE). MDRE formaliza as representações (modelos) derivadas de SL para garantir um entendimento comum sobre o seu conteúdo. Estes modelos são então utilizados como ponto de partida para a ER. Dessa forma, MDRE beneficia diretamente da extensibilidade, da cobertura, reutilização, integração e automação das tecnologias MDE para fornecer um bom suporte para a ER. No entanto, ainda há uma falta de soluções completas destinadas a cobrir todo o processo de MDRE.

Neste contexto, em 2003 a *Object Management Group* (OMG) criou uma força tarefa para analisar e evoluir os tradicionais processos de ER, formalizando-os e fazendo com que eles fossem totalmente apoiados pelas diretrizes e princípios de MDE. Logo, o termo Modernização Dirigida à Arquitetura (*Architecture-Driven Modernization* - ADM) surgiu como uma solução para os problemas de padronização. A ADM é um processo de modernização de SL que utiliza um conjunto de metamodelos para representar complemente um sistema por meio de diferentes representações arquiteturais. Esses modelos são então submetidos à refatorações e otimizações e o código-fonte é então gerado novamente. Durante a modernização de um sistema são gerados vários modelos de acordo com os metamodelos da ADM, que representam diferentes partes do sistema, como: fluxos de dados, banco de dados, elementos de programação (métodos, classes, tipos de dados, etc.) e arquitetura (Ulrich e Newcomb, 2010b; Pérez-Castillo et al, 2013).

O *Knowledge Discovery Metamodel* (KDM) é o principal metamodelo da ADM com uma ampla quantidade de metaclasses, cobrindo desde os níveis mais baixos de abstração de um sistema, como o código-fonte, até níveis mais altos, permitindo a representação de conceitos de qualquer domínio. A idéia principal da ADM é que a comunidade comece a desenvolver ferramentas que atuem somente sobre instâncias do KDM, ao invés de serem dependentes de plataformas e linguagens específicas. Por exemplo, um catálogo de refatorações para o

KDM (Durelli et al, 2014c)[5] tem o poder de reestruturar um sistema independentemente da linguagem de programação que foi usada em seu desenvolvimento, uma vez que as refatorações ocorrem em nível do KDM, ou seja, um modelo independente de plataforma e/ou linguagem.

Sistemas Legados precisam ser refatorados durante toda a sua vida útil para se que adequem a novos requisitos. No entanto, geralmente a má aplicação de refatorações/-modernizações em SLs pode causar desvios arquiteturais. Embora refatoração (tanto de baixa granularidade, quanto alta granularidade) seja uma técnica poderosa, e uma atividade recorrente durante a modernização em SLs, foi constatado durante a condução de um mapeamento sistemático (Durelli et al, 2014b)[6] que a versão original da ADM, e consequentemente do KDM, não fornecem apoio (por exemplo, a catálogos de refatoração, a metamodelos para definir refatorações, etc) para tal atividade. Além disso, dificilmente a arquitetura de um sistema legado permanece intacta depois de anos de manutenção, isso é, sua arquitetura atual possivelmente é diferente da arquitetura que foi previamente planejada. ADM também não fornece apoio à checagem de conformidade entre a arquitetura planejada e a atual.

Nesse sentido, este relatório enfatiza-se quatro principais atividades, a saber: (*i*) adaptação de um catalogo de refatoração para o metamodelo KDM, (*ii*) implementação de uma ferramenta semi-automática que fornecer suporte ao catalogo de refatoração adaptado para o KDM, (*iii*) criação de um metamodelo de refatorações, padronizado que contenha características similares ao KDM, ou seja, independente de plataforma e linguagem, com o principal objetivo de auxiliar o engenheiro de modernização durante a elaboração de refactorações e (*iv*) definição de uma Linguagem Especifica de Domínio (do inglês Domain-Specific Language - DSL) para auxiliar a instanciação do metamodelo de refatoração. A seção seguinte expõe os principais conceitos que dão embasamento para o entendimento das atividades realizadas pelo outorgado. Na Seção 3.1 é apresentado brevemente o con-

[5]No periodo de vigência pertinente a esse relatório, um artigo descrevendo um Catalogo de Refatoração Adaptado para o KDM foi publicado em um evento qualis B2 voltado para Engenharia de Software

[6]No periodo de vigência pertinente a esse relatório, um artigo descrevendo um Mapeamento Sistemático foi publicado em um evento qualis B2 voltado para Engenharia de Software

ceito sobre Refatoração, na Seção 3.2 é descrito os conceitos sobre *Model-Driven Development* (MDD). A Seção 3.3 brevemente discorre sobre *Model Driven Reverse Engineering* (MDRE), bem como, *Architecture-Driven Modernization* (ADM) e *Knowledge Discovery Metamodel* (KDM) que são os temas guarda-chuva para este projeto. Finalmente, na Seção 3.4 alguns dos apoios ferramentais utilizados nesse projeto são destacados.

Na Seção 4 são descritas detalhadamente as atividades realizadas pelo outorgado durante o período de vigência da bolsa. Por sua vez, o plano de trabalho para as etapas seguintes são mencionados na Seção Z.

3 Fundamentação Teórica

3.1 Refatoração

A refatoração (*refactoring*), de acordo com Fowler et al (1999), surgiu na comunidade de programadores Smalltalk[7]. Refatoração, consiste no processo de alterar um software, melhorando a sua estrutura interna, de forma que o comportamento externo do código não seja alterado. Além disso, refatoração permiti a distribuição de classes, variáveis e métodos na hierarquia de classes, com o objetivo de facilitar futuras atividades de desenvolvimento ou de manutenção Opdyke (1992); Fowler et al (1999); Demeyer et al (2004); Mens e Tourwe (2004).

No contexto da reengenharia, a refatoração é empregada para converter o código legado em um código mais modular e estruturado ou até com o objetivo de migrá-lo para uma nova linguagem de programação Mens e Tourwe (2004). No entanto, a medida que o código é alterado o mesmo torna-se gradualmente difícil de entender.

De acordo com Fowler et al (1999) a maioria das refatorações introduz indireção, ou seja, tendem a dividir objetos de maior granularidade em objetos menores e métodos longos são transformado em vários métodos menores. A seguir é apresentado algumas vantagens relacionadas à indireção:

[7]Linguagem de programação orientada a objeto fracamente tipada

- **Explicar intenção e implementação separadamente**: o nome de cada método, variável ou classe fornece a oportunidade de explicar sua intenção. A implementação de classes ou métodos explicam como a intenção é realizada;
- **Isolar a mudança**: facilita a introdução de funcionalidade.
- **Codificar a lógica condicional**: alterando a lógica condicional por mensagens polimórficas evita-se duplicações de código e aumenta-se a flexibilidade.

Hoje em dia várias *Integrated Development Environments* (IDE) conseguem automatizar algumas refatorações. Porém, nenhuma IDE consegue determinar qual trecho de código deve ser refatorado e nem quais tipos de refatoração devem ser aplicadas. Portanto, o desenvolvedor ainda é responsável por determinar qual trecho de código deve ser refatorado. De acordo com Fowler et al (1999) essa etapa pode ser feita por meio de análise de "*bad smells*". Vale ressaltar que nenhum critério exato pode ser utilizado para determinar quando o código deve ser refatorado ou não.

3.2 *Model-Driven Development* (MDD)

Pesquisas apontam que com a utilização de MDD muitos benefícios podem ser obtidos ao mover de abordagens que são totalmente centradas a código-fonte habituais para outras baseadas em modelos. Este paradigma (MDD) é amplamente baseada na suposição de que "Tudo é um modelo" (Bézivin, 2005). Dessa forma, MDD basicamente se baseia em quatro principais conceitos: **meta-metamodel**, **metamodelo**, **modelo** e **transformações de modelos**. Um **meta-metamodelo** define linguagens de modelagem, como a UML, por exemplo. Um exemplo de meta-metamodel é o padrão *Meta-Object Facility* (MOF)[8]. Um **metamodelo** define os possíveis elementos e estrutura dos **modelos**, de forma semelhante à relação entre a gramática e programas correspondentes no campo de programação. **Transformações de modelos** são na verdade definido ao nível do **meta-modelo**, e depois aplicado no nível do **modelo**, a partir dos **modelos** que conformam aos

[8] `http://www.omg.org/mof/`

metamodelos. Por exemplo, as **transformações de modelos** são executadas entre um modelo fonte e um modelo alvo. Além disso, **transformações de modelos** podem ser ou do tipo *Model-To-Model* (e.g., Eclipse ATL[9]) ou do tipo *Model-To-Text* (e.g., Eclipse Acceleo[10]).

Note-se que o MDD, também conhecido como *Modelware* (OMG, 2012b), não é tão diferente do grammarware (ou seja, onde as linguagens de programações são definidas em termos de gramáticas), em termos de definição de base e infra-estrutura. Tal afirmação pode ser visualizada na Figura 1.

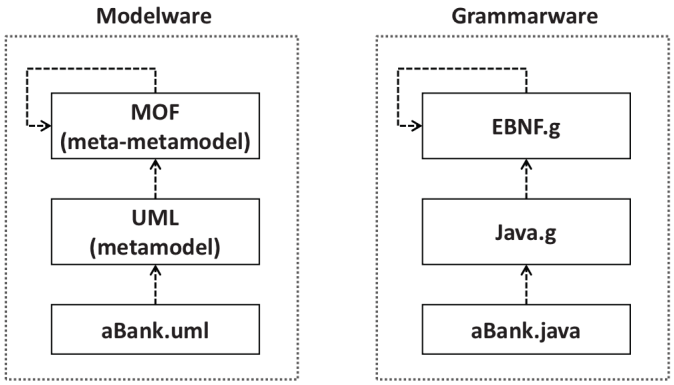


Figura 1: Modelwarevs.Grammarware.

MDD é ainda considerado um paradigma atual no contexto de Engenharia de Software. MDD foi popularizado pela *Object Management Group* (OMG) como *Model Driven Architecture* (MDA) (Kleppe et al, 2003). Hoje em dia o Eclipse, o qual é um *Integrated Development Environment*, é o ambiente padrão para MDD uma vez que o mesmo contêm uma implementação de referência do MOF, nomeado *Eclipse Modeling Framework* (EMF).

3.3 *Model-Driven Reverse Engineering* (MDRE)

A utilização de MDD no contexto da ER (ou seja, MDRE) é um campo relativamente recente (Rugaber e Stirewalt, 2004). No início, os modelos eram apenas utilizados, principalmente, para especificar os sistemas antes da sua implementação (durante a Engenharia

[9] `https://www.eclipse.org/atl/`
[10] `https://www.eclipse.org/acceleo/`

Avante). MDRE propõe que modelos não sejam apenas artefatos que "guiam" o engenheiro durante tarefas de desenvolvimento e manutenção de software, mas como parte integrante do software (Thomas, 2004).

Devido ao grande interesse em MDRE, a OMG em 2003 inicio uma força tarefa (ADM *taskforce*) que tinha como intuito padronizar o processo de engenharia reversa. Da mesma forma que MDA, a OMG criou a *Architecture-Driven Mordernization* (ADM) que tem como objetivo criar especificações/padronizações para auxiliar processo de modernização de sistemas legados por meio de um conjunto de metamodelos padronizados. O fluxo de um processo de MDRE apoiada pela ADM possui três fases e é semelhante ao contorno de uma ferradura, são elas: Engenharia Reversa (ER) (do inglês, *Reverse Engineering*), Reestruturação (do inglês, *Restructuring*) e Engenharia Avante (EA) (do inglês, *Forward Engineering*), como pode ser visto na Figura 2. Partindo do lado inferior esquerdo, na parte da ER, o conhecimento é extraído do sistema legado e um modelo *Plataform Specific Model* (PSM) é gerado. O modelo PSM serve como base para a geração de um modelo *Platform Independent Model* (PIM), ou seja, durante a fase de ER, transformações são feitas como o intuito de se obter uma representação de alto nível do software, independentemente da plataforma utilizada anteriormente.

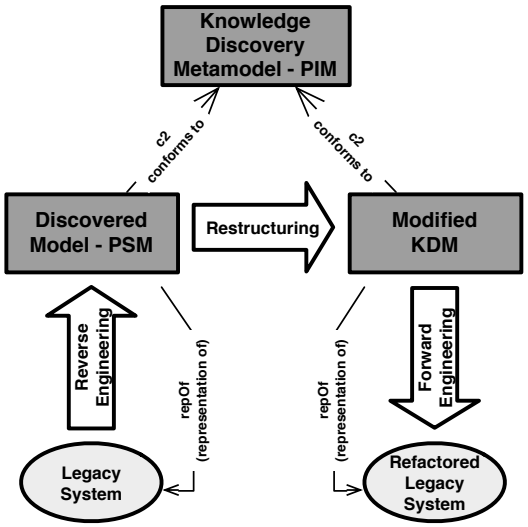


Figura 2: Fluxo do processo de modernização apoiada pela ADM (Adaptada (OMG, 2012a))

O modelo PSM é um modelo especifico de uma plataforma, ou seja, nele existem metadados relacionados a uma plataforma ou linguagem de programação especifica. O modelo PIM é um modelo de abstração mais alto, pois não contém informações especificas de uma determinada plataforma ou linguagem.

Na fase de reestruturação, refatorações, melhorias e novas regras de negócios podem ser introduzidas no sistema, e, com uma representação independente de plataforma do software modernizado, segue-se para a fase de Engenharia Avante. Nessa ultima fase, os modelos são novamente submetidos a uma série de transformações para chegar ao nível de artefatos executáveis, ou seja, código-fonte.

No contexto da ADM, para dar suporte ao processo de modernização independentemente de plataforma e linguagem (PIM), foi criado um metamodelo que possibilita a comunicação entre diferentes plataformas e linguagens, e foi denominado pela ADM *task-force* de *Knowledge Discovery Metamodel* (KDM) (ISO/IEC DIS 19506, 2011). O KDM provê representações para os sistemas de software existentes em diferentes camadas de abstrações. Cada modelo é representado por um conjunto de visões arquiteturais, ou seja, modelos KDM representando diferentes perspectivas de conhecimento sobre os artefatos dos sistemas de software existentes. Esses modelos são criados automaticamente, semi-automaticamente ou manualmente por meio da aplicação de várias técnicas de extrações de conhecimento, de análises e de transformações. (Perez-Castillo et al, 2011a).

O KDM representa artefatos físicos e lógicos de software dos sistemas legados em diferentes níveis de abstração e constitui dozes pacotes organizados em quatro camadas, são elas: infraestrutura (*infrastructure*), elementos de programa (*program elements*), recursos de tempo de execução (*runtime resources*) e abstração (*abstractions*) (Perez-Castillo et al, 2011a). Na Figura 3 está representada a arquitetura do KDM ilustrando a forma como as camadas se relacionam. Cada camada baseia-se na camada anterior, dessa forma, elas estão organizadas em pacotes e definem um conjunto de metaclasses, cujo propósito é representar um interesse específico e independente do conhecimento relacionado a sistemas legados.

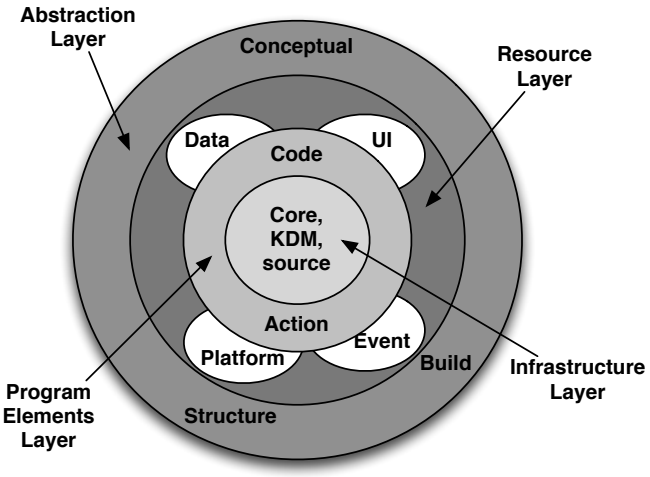


Figura 3: Arquitetura do KDM (adaptada (OMG, 2012a))

Três pacotes importantes do KDM no contexto deste trabalho são: (*i*) *Code*, (*ii*) *Action* e (*iii*) *Structure*. Os dois primeiros pacotes contêm metaclasses para representar elementos de programação, tais como, classes, métodos, atributos, etc. Dessa forma, utilizando esses dois pacotes foi possível criar/adaptar um catalogo de refatoração para o contexto do KDM. Assim, todas as refatorações que foram adaptadas para o KDM agora são independentes de linguagem e plataforma. O terceiro pacote por sua vez (*Structure*), define elementos de metamodelo que representam componentes arquitetural dos sistemas de software existentes. Por exemplo, esse pacote define as seguintes metaclasses: *Subsystem*, *Component*, *Layer*, *SoftwareSystem*, *ArchitectureView*. Ressalta-se ainda que esse ultimo pacote, é de suma importância para as próximas atividades do outorgado, uma vez que verificação de conformidade em nível arquitetural serão implementadas para o KDM. Além disso, também serão definidas refatorações em nível arquitetural.

3.4 Apoio ferramental

Diversas ferramentas de modelagem estão disponíveis para o desenvolvimento empresarial baseado em modelo, uma plataforma de ferramentas que se tornou proeminente no mundo da MDRE é o IDE Eclipse. Um conjunto de ferramentas interessantes para MDRE foram disponibilizados para essa IDE, permitindo, assim diversas iniciativas sobre esta

plataforma. No contexto, do projeto em questão algumas iniciativas foram utilizadas: (*i*) *Eclipse Modeling Framework* (EMF), (*ii*) Xtext, (*iii*) Acceleo e (*iv*) *ATL Transformation Language* (ATL).

Eclipse Modeling Framework (EMF) é a principal iniciativa do IDE Eclipse no contexto de MDRE por várias razões. Primeiro, EMF permite a definição de metamodelos tendo como base uma linguagem de metamodelagem denominada Ecore. Em segundo lugar, EMF fornece um gerador de metamodelos, ou seja, uma API baseada em Java para manipulação de modelos. Em terceiro lugar, EMF contêm uma poderoso API que abrangem diferentes aspectos, tais como a serialização e deserialização de modelos de/para *XML Metadata Interchange* (XMI). Xtext é um framework para desenvolvimento de Linguagens Específicas de Domínio (do inglês, *Domain-Specific Language*). Acceleo é uma implementação pragmática do OMG para fornecer suporte a transformações *Model2Text*. Por fim, ATL é uma linguagem de transformação de modelos, ou seja provê suporte a transformações *Model2Model*.

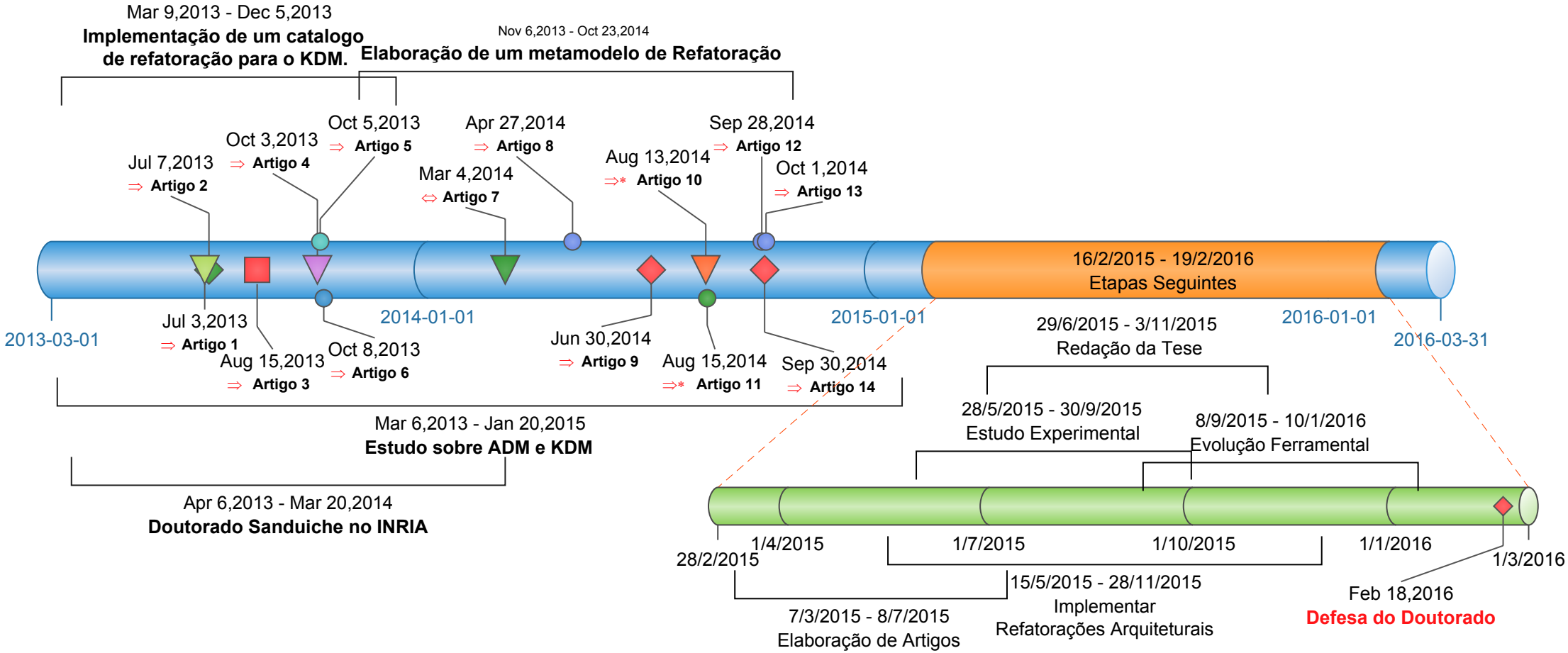
4 Cronograma e Resumo das Atividades Realizadas no Período

A seguir é apresentado uma *timeline* que descreve todas as atividades realizadas durante o período de vigência e que serão realizadas pelo bolsista para o desenvolvimento deste projeto. Em seguida, é apresentada uma descrição de cada evento presente na *timeline*. Ressalta-se que todos os artigos que foram publicados durante o período de vigência desse relatório estão destacados tanto na *timeline* quanto na descrição pelo símbolo $\Rightarrow$ ou $\Leftrightarrow$. Note que, $\Leftrightarrow$ ilustra capítulo de livro, enquanto o símbolo $\Rightarrow$ refere-se a artigos publicados em conferências. Além disso, vale destacar que maiores informações sobre cada evento serão descritas nas seções seguintes.

- **Mar 6, 2013 - Jan 20, 2015** - Estudo detalhado sobre ADM e KDM;

- **Mar 9, 2013 - Dec 5, 2013** - Implementação de um catalogo de refatoração para o KDM;
- **Apr 6, 2013 - Mar 20, 2014** - Realização do Doutorado Sanduíche no INRIA, Lille, França;
- **Nov 6, 2013 - Oct 23, 2014** - Elaboração de um metamodelo de refatoração;
- ⇒ Artigo 1 - Concern-Based Refactorings Supported by Class Models to Reengineer Object-Oriented Software into Aspect-Oriented Ones;
- ⇒ Artigo 2 - F3: From features to frameworks;
- ⇒ Artigo 3 - An Approach to Develop Frameworks from Feature Models;
- ⇒ Artigo 4 - F3T: From Features to Frameworks Tool;
- ⇒ Artigo 5 - CCKDM - A Concern Mining Tool for Assisting in the Architecture-Driven Modernization Process;
- ⇒ Artigo 6 - A Combined Approach for Concern Identification in KDM models;
- ⇔ Artigo 7 - Developing Frameworks from Extended Feature Models;
- ⇒ Artigo 8 - Evaluating the Effort for Modularizing Multiple-Domain Frameworks towards Framework Product Lines with Aspect-Oriented Programming and Model-Driven Development;
- ⇒ Artigo 9 - Data Network in Development of 3D Collaborative Virtual Environments: A Systematic Review;
- ⇒ Artigo 10 - Towards a Refactoring Catalogue for Knowledge Discovery Metamodel;
- ⇒ Artigo 11 - A Mapping Study on Architecture-Driven Modernization;
- ⇒ Artigo 12 - KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel;

- ⇒ Artigo 13 - Investigating Lightweight and Heavyweight KDM Extensions for Aspect-Oriented Modernization;
- ⇒ Artigo 14 - KDM-RE: A Model-Driven Refactoring Tool for KDM;
- **16/2/2015 - 19/2/2016** - Esse período representa as Etapas Seguintes;
  - **7/3/2015 - 8/7/2015** - Elaboração de Artigos;
  - **15/5/2015 - 28/11/2015** - Implementar Refatorações Arquiteturais;
  - **28/5/2015 - 30/09/2015** - Estudo Experimental;
  - **29/06/2015 - 3/11/2015** - Redação da Tese;
  - **08/09/2015 - 10/01/2016** - Evolução Ferramental;
  - **Fevereiro** - Defesa do Doutorado;


Mar 9,2013 - Dec 5,2013
Implementação de um catalogo de refatoração para o KDM.
Nov 6,2013 - Oct 23,2014
Elaboração de um metamodelo de Refatoração
Jul 7,2013 ⇒ Artigo 2
Oct 3,2013 ⇒ Artigo 4
Oct 5,2013 ⇒ Artigo 5
Mar 4,2014 ⇔ Artigo 7
Apr 27,2014 ⇒ Artigo 8
Aug 13,2014 ⇒* Artigo 10
Sep 28,2014 ⇒ Artigo 12
Oct 1,2014 ⇒ Artigo 13
2013-03-01
Jul 3,2013 ⇒ Artigo 1
Aug 15,2013 ⇒ Artigo 3
Oct 8,2013 ⇒ Artigo 6
2014-01-01
Jun 30,2014 ⇒ Artigo 9
Aug 15,2014 ⇒* Artigo 11
Sep 30,2014 ⇒ Artigo 14
2015-01-01
16/2/2015 - 19/2/2016
Etapas Seguintes
2016-01-01
2016-03-31
29/6/2015 - 3/11/2015
Redação da Tese
28/5/2015 - 30/9/2015
Estudo Experimental
8/9/2015 - 10/1/2016
Evolução Ferramental
Mar 6,2013 - Jan 20,2015
Estudo sobre ADM e KDM
Apr 6,2013 - Mar 20,2014
Doutorado Sanduiche no INRIA
28/2/2015
1/4/2015
1/7/2015
1/10/2015
1/1/2016
1/3/2016
7/3/2015 - 8/7/2015
Elaboração de Artigos
15/5/2015 - 28/11/2015
Implementar
Refatorações Arquiteturais
Feb 18,2016
Defesa do Doutorado

4.1 Resumo das Atividades Realizadas no Período

Nesta seção são apresentadas as atividades desenvolvidas no período de Maio/2013 a Fevereiro/2015, previstas no plano inicial que foi enviado à FAPESP. Vale ressaltar que, em parte do período deste relatório, o bolsista esteve no exterior realização um estágio sanduíche no *Institut National de Recherche en Informatique et en Automatique* (INRIA) . As atividades realizadas são brevemente descritas nas subseções a seguir.

4.2 Estágio no Exterior

O estágio contou com o apoio financeiro da CNPQ (Processo: 241028/2012-4) e foi realizado no *Institut national de recherche en informatique et en automatique* (INRIA), França. O aluno ficou na França por um ano (Abril de 2013 - Abril de 2014) sob a supervisão do professor Doutor Nicolas Anquetil. O grupo de pesquisa é especializado em remodularização e modernização de sistemas orientados a objetos com enfoque em *Model-Driven Development*.

O vínculo com o CNPQ foi encerrado no mês de Abril e a bolsa da FAPESP foi reatividade. Uma cópia do parecer emitido pelo orientador no exterior pode ser encontrado no Anexo 6.

4.3 Mapeamento Sistemático

Quando se conduz uma revisão de literatura sem o pré-estabelecimento de um protocolo de revisão há um direcionamento por interesses pessoais, o que leva a resultados pouco confiáveis. Neste contexto, pesquisadores vem utilizando uma técnica denominada de Mapeamento Sistemático (MS) para auxiliar o pesquisador a conduzir um revisão bibliográfica de forma totalmente sistemática com o intuito de evitar que trabalhos importantes fiquem fora de suas pesquisas. Um MS é caracterizada por ser um meio de avaliar e interpretar todas as pesquisas disponíveis, referentes a um questão de pesquisa, tema, área ou fenômeno de interesse. O MS tem como objetivo apresentar uma avaliação justa de um tema

de pesquisa, utilizando uma metodologia confiável, rigorosa e auditável Kitchenham et al (2009).

De acordo com Kitchenham et al (2009) o MS implica na forma mais adequada para se identificar, avaliar e interpretar toda pesquisa importante para um tema em particular. Resume-se que um MS configura um alicerce para novas atividades de pesquisa acerca de determinado tema. Dessa forma, foi realizado um MS sobre *Architecture-Driven Modernization* (ADM) e *Knowledge-Discovery Metamodel* (KDM). A nossa motivação para realizar esse MS é identificar os temas que têm sido mais investigados, bem como os temas que ainda não foram investigados. Embora, a ADM é uma abordagem relativamente nova, OMG afirma que ela é uma importante abordagem pois combina dois dos principais campos da Engenharia de Software, ou seja, *Model-Driven Development* e Engenharia Reversa. Desde a definição da ADM muitos esforços têm enfatizado a modernização de sistemas legados por meio desta abordagem. Neste contexto, é importante realizar uma investigação mais sistemática dos temas englobados por esta área de pesquisa.

Para atingir este objetivo, foi realizado um MS. Os resultados, bem como o protocolo de pesquisa elaborado, foram publicados no *IEEE Information Reuse and Integration* (IRI 2014) , o qual tem qualis B2. O artigo foi apresentado pelo bolsista em Agosto de 2014 em São Francisco, California. Maiores detalhes sobre esse artigo pode ser obtido no Apêndice A deste relatório. O mesmo anexo contem uma cópia da obra submetida.

4.4 Estabelecimento do Catalogo de Refatoração para o metamodelo KDM

Foi elaborado um catalogo dedicado para ser aplicado no metamodelo do KDM. Mais especificamente esse catalogo foi adaptado com base em catálogos já disponíveis na literatura. Para a definição desse catalogo de refatoração, foi utilizado um formato inspirado pelo Fowler et al (1999). Em outras palavras, o catalogo foi descrito da seguinte forma: (*i*) o nome do refatoração, (*ii*) descrição da típica situação onde a refatoração deve ser aplicada, (*iii*) descrição da solução para solucionar uma determinada situação problemática, (*iv*)

pre-condições que devem ser satisfeitas para aplicar a refatoração, (*v*) os parâmetros necessários para executar a refatoração e (*vi*) descrição dos passos para realizar a refatoração.

O catalogo de refatoração é estruturado em quatro grupos como pode ser observado na Tabela **??**, a qual contêm 17 refatorações. O primeiro grupo chamada *Rename Feature* consiste de refatorações para renomear *ClassUnit*, *StorableUnits* e *MethodUnits*. O segundo grupo, *Moving Features Between Objects* consiste de refatorações simples, tais como mover ou criar características, ou seja, criar ou mover atributos, métodos ou classes. O terceiro grupo, *Organizing Data*, é responsável por definir um conjunto de refatorações para ser organizar a estrutura do código-fonte. Por fim, o quatro grupo, *Dealing With Generalization*, representa refatorações para mover métodos e/ou atributos sobre uma especifica hierarquia de classes.

Tabela 1: Refactorings Adapted to KDM

Rename Feature	Moving Features Between Objects	Organing Data	Dealing with Generalization
Rename ClassUnit	Move MethodUnit	Replace data value with Object	Push Down MethodUnit
Rename StorableUnit	Move StorableUnit	Encapsulate StorableUnit	Push Down StorableUnit
Rename MethodUnit	Extract ClassUnit	Replace Type Code with ClassUnit	Pull Up StorableUnit
	Inline ClassUnit	Replace Type Code with SubClass	Pull Up MethodUnit
		Replace Type Code with State/Strategy	Extract SubClass
			Extract SuperClass
			Collapse Hierarchy

Vale ressaltar que um artigo descrevendo o catalogo de refatoração adaptado para o KDM também foi no *IEEE Information Reuse and Integration* (IRI 2014) , o qual tem qualis B2. O artigo foi apresentado pelo bolsista em Agosto de 2014 em São Francisco, California. Além disso, o bolsista participou das sessões técnicas e palestras. Maiores detalhes sobre esse artigo pode ser obtido no Apêndice B deste relatório. O mesmo contem uma cópia da obra submetida.

4.5 Um ambiente integrado para desenvolvimento e apoio para o catalogo de refatorações do metamodelo KDM

Durante a revisão sistemática pode-se averiguar que tanto o processo ADM e o seu metamodelo KDM têm sido vastamente utilizado na literatura para auxiliar a modernização de sistemas legados. No entanto, também pode-se verificar qua até o momento não existe

nenhum ambiente integrado de desenvolvimento para guiar o engenheiro para automaticamente aplicar as refatorações e modernizações como existe em outros paradigmas, tais como o paradigma orientado a objetos. Para mitigar tal limitação, durante o período de vigência da bolsa foi desenvolvido um *plug-in* utilizando a plataforma do Eclipse.

Esse *plug-in* fornece um ambiente para realizar as refatorações apresentadas na Seção 4.4 de forma totalmente automatizada. Na Figura 4 é ilustrado todo o processo no qual o *plug-in* é baseado. Como pode ser observado nessa figura a utilização do *plug-in* pode ser ilustrada em três passos: (*i*) engenharia reversa (*reverse engineering*), (*ii*) refatorações (*refactorings*) e (*iii*) engenharia avante (*forward engineering*). Maiores detalhes sobre tais passos são descritos nas próximas seções.

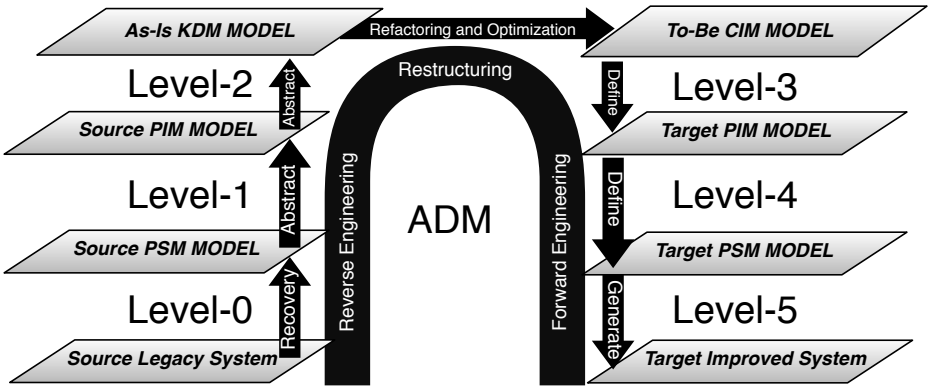


Figura 4: Passos para utilização do *plug-in*

4.5.1 Engenharia Reversa

Para iniciar esse passo o engenheiro de software deve entrar com um arquivo que representa uma instância do metamodelo KDM ou um determinado código-fonte para realizar a refatoração. No caso do código-fonte ser a entrada escolhida, o engenheiro de software começa o processo no **Level-0** escolhendo um projeto no Eclipse[11] que contenha o código-fonte para realizar as refatorações. Posteriormente, no **Level-1** o código-fonte precisa ser transformado para um modelo especifico de plataforma (no inglês - Platform-Specific Model (PSM)). Esse PSM representa uma instância do código-fonte em um nível mais abstrato do

[11] https://www.eclipse.org/

código-fonte. Para realizar essa transformação (código-fonte para PSM) foi implementado um extrator de modelo em Java.

Após criar o PSM o próximo level (**Level-2**) consiste em transformar o PSM para um Modelo Independente de plataforma (no inglês - Platform-Indented Model (PIM)) o qual é baseado no metamodelo do KDM. Nesse level o *plug-in* utiliza o *framework* MoDisco[12] para realizar a transformação de PSM para PIM.

Na Figura 5 é apresentado uma visão geral do *plug-in* desenvolvido pelo bolsista durante a período de vigência da bolsa. Apenas para o propósito de explicação, foi identificado quatro principais regiões do *plug-in*, veja Figura5 ⓐ, ⓑ, ⓒ and ⓓ.

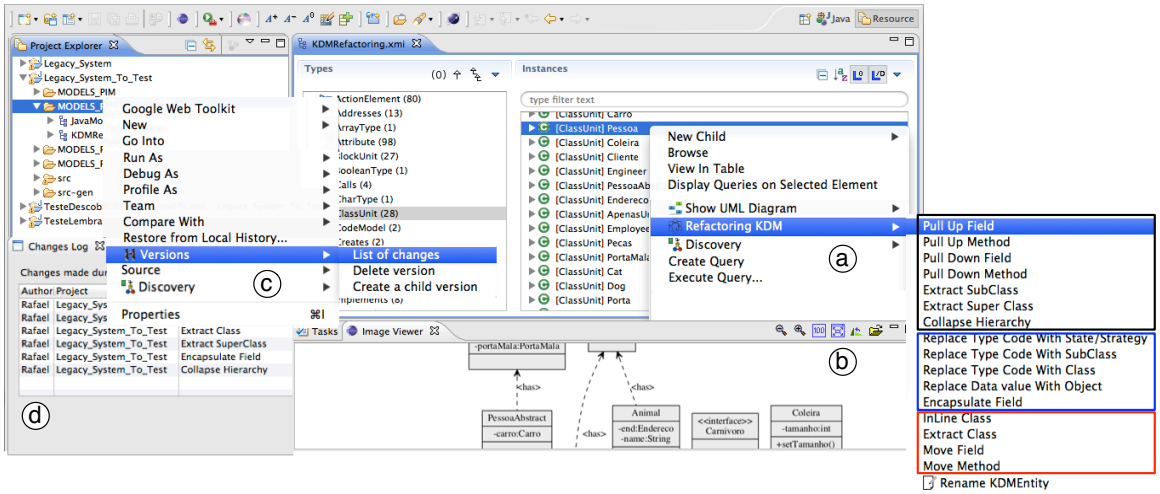


Figura 5: Visão geral do *plug-in* desenvolvido

Como já salientado anteriormente todas as refatorações fornecidas pelo *plug-in* são feitas com base no metamodelo KDM. Dessa forma, para auxiliar o engenheiro de software a realizar as refatorações um menu chamado *Refactoring KDM* foi adicionado, veja Figura 5ⓐ. Utilizando esse menu o engenheiro de software pode interagir com o metamodelo do KDM e escolher qual refatoração deve ser executada. Note que na região ⓐ da Figura 5 é possível ver todas as 17 refatorações implementadas no *plug-in*.

Na região ⓑ da Figura 5 é apresentado um diagrama de classe. Esse diagrama pode ser utilizado pelo engenheiro de software antes ou depois de aplicar as refatorações no modelo

[12]http://www.eclipse.org/MoDisco/

KDM. Usualmente o engenheiro pode utilizar esse diagrama antes de aplicar refatorações para decidir onde deve-se realmente aplicar as refatorações. Além disso, esse diagrama é útil por que geralmente sistemas legados não contêm nenhum tipo de documentação, sendo o código-fonte o único artefato disponível do mesmo. Portanto, criar um diagrama de classe durante a execução de refatorações no sistema legado pode ser uma boa alternativa para melhorar a documentação de um determinado sistema.

O *plug-in* também fornece múltiplas versões de um sistema em nível de modelos, ou seja, em nível de modelos KDM. O objetivo é permitir que o engenheiro de software trabalhe interativamente em vários modelos, permitindo assim que o engenheiro de software escolha e explore diferentes caminhos de refatoração. Como pode ser observado na região ⓒ da Figura 5, o engenheiro deve selecionar o arquivo KDM e escolher a opção "*Versions*". Três opções são disponíveis nesse menu (*i*) *List of Changes*, (*ii*) *Delete version* and (*iii*) *Create a child version*. A primeira opção mostra todas refatorações que já foram feitas pelo engenheiro (ver região ⓓ) - a segunda opção é responsável por deletar uma versão - e a ultima opção criar uma cópia do arquivo KDM, permitindo assim que o engenheiro aplique outras refatorações e explore diferentes caminhos de refatoração sem afetar o modelo KDM principal do projeto.

4.5.2 Executando Refatorações no *Plug-in*

Após o engenheiro clicar no menu da região ⓐ (ver Figura 5) e escolher qual refatoração aplicar um *Wizard* será mostrado. Para explicação, considere que o engenheiro de software escolheu aplicar a refatoração denominada *Extract Class*. Então o engenheiro deve selecionar qual metaclass o mesmo deseja extrair, esse passo é ilustrado na Figura 6 (a). Posteriormente o *plug-in* executa o *Wizard* como ilustrado na Figura 6 (b). Como pode ser observado, aqui o engenheiro de software pode atribuir um nome para a nova metaclasse. Além disso, uma prévia de todos os detectados *StorableUnits* e *MethodUnis* que podem ser extraídos e adicionados na outra classe também são mostrados. O engenheiro pode tam-

bém selecionar se a nova classe será uma classe interna ou uma classe normal. Também é possível especificar se métodos assessores (*getters* e *setters*) devem ser criados ou não.

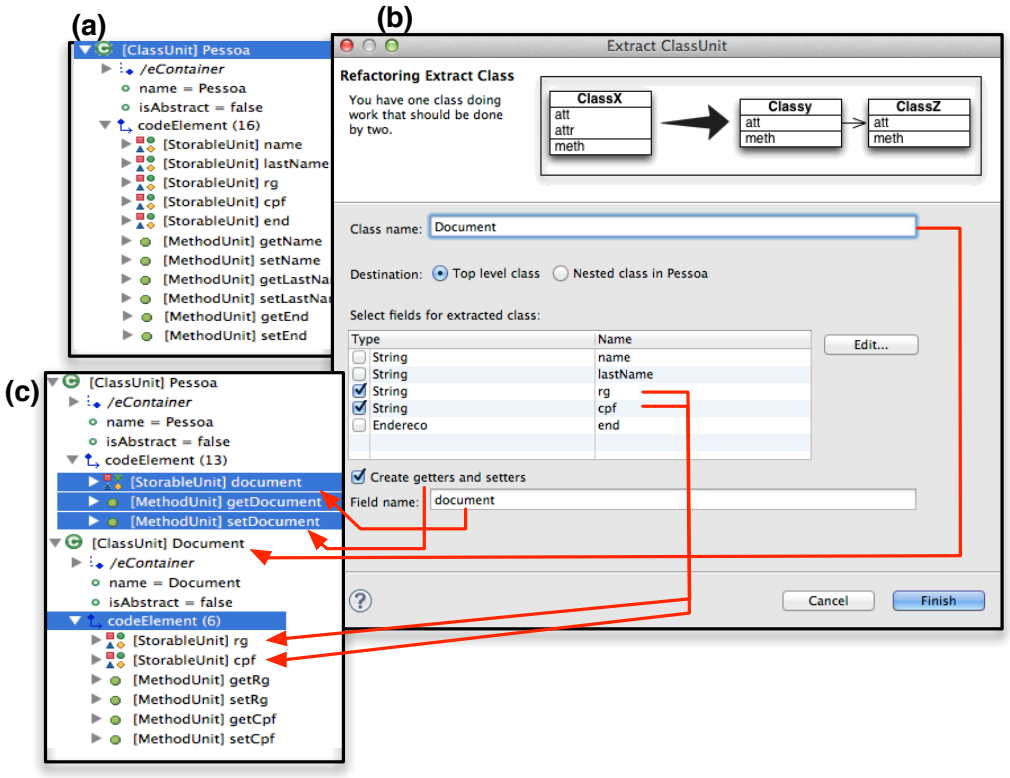


Figura 6: Extract Class Wizard

Após o engenheiro preencher todos os campos necessários, ele pode clicar no botão *Finish* e então a refatoração *Extract Class* é executada. Como pode ser observado na Figura 6 (c) uma nova instância de *ClassUnit* denominada *Document* foi criada - dois *StorableUnits* da metaclasse *Pessoa*, "rg" e "CPF" foram movidas para *Document*.

4.5.3 Engenharia Avante

Após o engenheiro realizar todas as refatorações no KDM os próximos passos são: (*i*) transformar o KDM para um PSM e (*ii*) transformar o PSM para artefatos físicos (código-fonte). O primeiro passo é executado baseado em um conjunto de transformações utilizando a linguagem ATL Transformation Languag (ATL)[13]. O último passo consiste em utilizar

[13]https://www.eclipse.org/atl/

templates para gerar o código-fonte refatorado. Na Figura 7 é apresentado como é feita a geração de código-fonte.

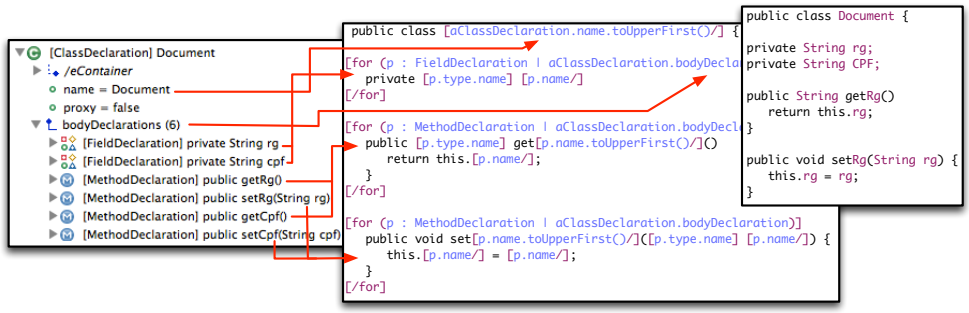


Figura 7: Passos da Engenharia Avante

4.5.4 Arquitetura do *plugin*

Na Figura 8 é ilustrado a arquitetura do *plug-in* desenvolvido durante o período de vigência da bolsa. Como pode ser observado nessa figura, a primeira camada (*layer*) é o *Core Framework*. Essa camada representa que o *plug-in* foi desenvolvida utilizando como base a plataforma de desenvolvimento Eclipse. Além disso, nessa camada pode-se observar que também foi utilizado Java e Groovy como linguagem de programação. Também é possível identificar que nessa camada alguns *plug-ins* da plataforma de desenvolvimento Eclipse foram utilizados, tais como MoDisco[14] e EMF[15]. Modisco e EMF ambos foram utilizados pois fornecem uma *Application Programming Interface* (API) para facilitar o acesso ao metamodelo KDM.

A segunda camada, *Tool Core*, é onde todas as refatorações fornecidas pelo *plug-in* foram implementadas. A ultima camada é onde a interface gráfica do *plug-in* foi desenvolvida. Vale ressaltar que um artigo sobre esse *plugin* foi publicado no 2nd *Workshop on Software Visualization, Evolution and Maintenance* (VEM). O bolsista esteve presente nesse evento, apresentado o artigo e participando das sessões técnicas e palestras. Mais detalhes sobre este *plugin* podem ser encontrados na cópia desse artigo no Apêndice C.

[14]http://www.eclipse.org/MoDisco/
[15]https://www.eclipse.org/modeling/emf/

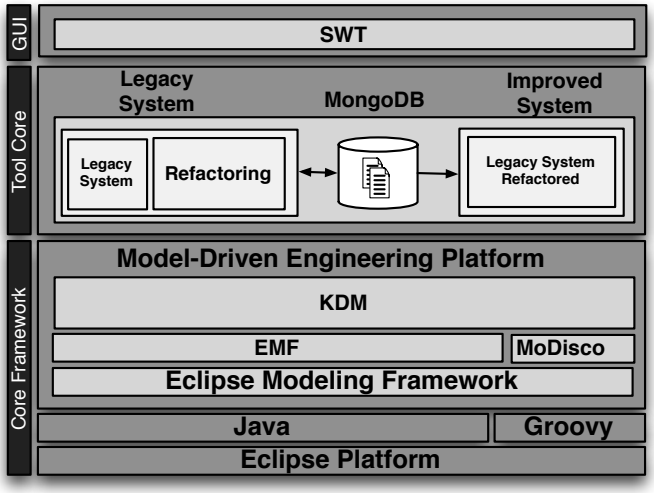


Figura 8: Arquitetura do *plug-in*

4.6 Um Metamodelo para Especificar Refatorações para o KDM - (*Refac-KDM*)

Durante o desenvolvimento e evolução de software novas funcionalidades geralmente são adicionadas ou são ajustadas a novas exigências. Devido a tais alterações, a flexibilidade da arquitetura desse sistema pode ser um fator desafiador. Em outros casos, após aplicar um conjunto de alterações a arquitetura continua perfeitamente adequada. Porém, usualmente é necessário aplicar mudanças na arquitetura desse sistema para confortar tais alterações.

Para melhorar o projeto de um determinado software, geralmente os engenheiros de software gastam uma quantidade significativa de tempo reestruturando o software manualmente. No entanto, reestruturar código manualmente além de ser uma tarefa que demanda tempo é totalmente propicia a erros. Não importa a atenção dispendida pelo engenheiro de software durante a atividade de reestruturação, se o sistema é relativamente grande, há uma boa chance de que o mesmo irá ter o seu comportamento alterado após tal atividade. Como ressaltado anteriormente é de suma importância que o comportamento do sistema seja preservado após a reestruturação, assim, o conceito de refatoração deve ser aplicado (ver Seção 3.1).

Hoje em dia refatoração é utilizada tanto no âmbito acadêmico quanto na industria. Refatoração é uma área muito madura e muito difundida. Além disso, várias IDEs executam facilmente e de forma segura um conjunto refatorações para um vasto número de linguagens de programação. Com o surgimento de MDD é importante que os conceitos de refatorações sejam adaptados tanto para MDRE (ver Seção 3.3) e ADM. Uma iniciativa têm sido conduzida pelo outorgado. Mais especificadamente o bolsista definiu um ambiente integrado para desenvolvimento e apoio para o catalogo de refatorações do metamodelo KDM (Durelli et al, 2014a). Maiores informações podem ser obtidas no artigo publicado, o mesmo pode ser visualizado na integra no Apêndice X.

Um olhar mais atento à reutilização de refatorações levanta a questão sobre o que pode ser reutilizado e o que não pode. Trabalhos anteriores nesta área tem mostrado que há potencial para reutilização, mas é limitado, de uma forma ou de outra. No entanto, pesquisas também apontam que algumas características não podem ser capturadas na parte reutilizável de uma refatoração. Por exemplo, uma refatoração genérica não pode fazer suposições sobre a semântica de uma linguagem de programação.

Embora a ADM e, principalmente o KDM, tenham sido propostos para apoiar a modernização de sistemas legados, até o momento não existem propostas de um metamodelo para especificar refatorações para o KDM. Portanto, os engenheiros de modernização precisam desenvolver suas próprias soluções para transformar instâncias KDM origem em alvo. Durante o mapeamento sistemático conduzido e publicado pelo outorgado (Durelli et al, 2014b) (ver Apêndice X) pôde-se observar na literatura a carência de estudos que definem um metamodelo para especificar refatorações para KDM. Sem a adequada representação de refatorações no KDM, a realização de uma refatoração pode se tornar propensa a erros. Assim, é de suma importância definir uma extensão para o KDM em que os engenheiros de modernização possam especificar refatorações independentes de plataforma.

Como já ressaltado existe uma forte necessidade de reutilizar refatorações no contexto da ADM e, principalmente para o metamodelo KDM. Para permitir tal reutilização, as partes de refatorações que podem ser generalizadas devem ser separadas das que são es-

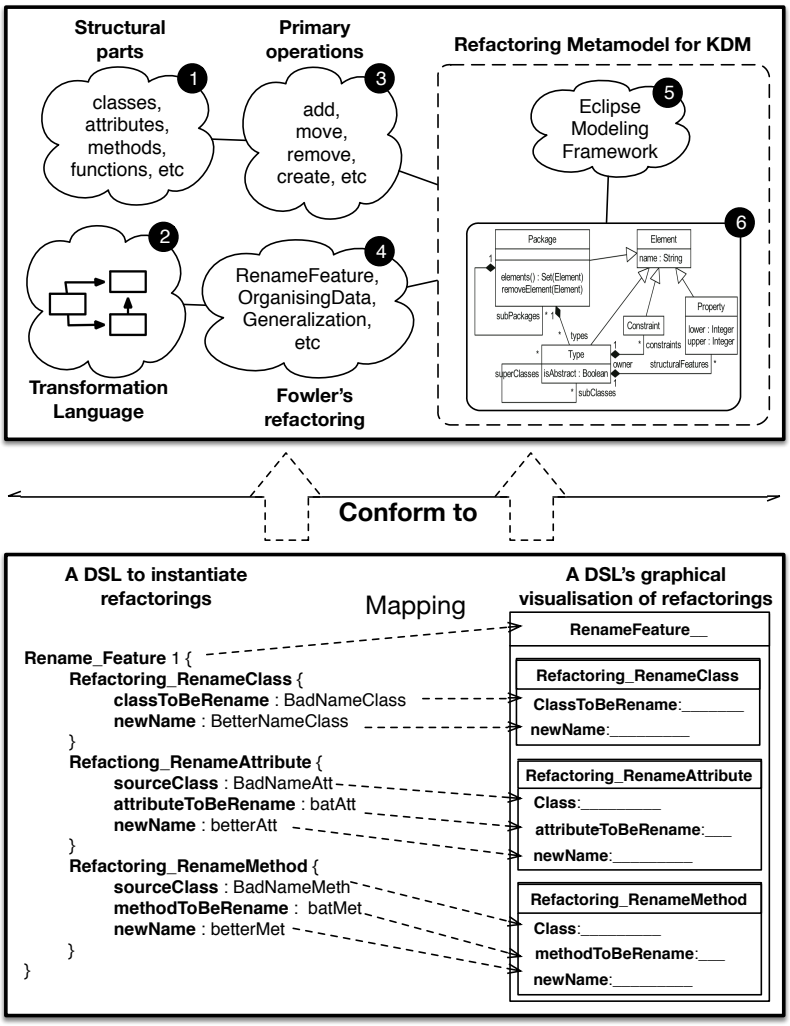


Figura 9: Princípios e critérios utilizados para criar o metamodelo de refatoração para o KDM

pecíficas. Por exemplo, considere a refatoração *RenameElement*. As etapas necessárias para executar tal refatoração são iguais, não importa que tipo de elemento precisa ser renomeado. Por exemplo, depois de alterar o valor de um atributo do tipo *String*, todas as referências daquele elemento precisa também ser atualizado. O atributo concreto pode variar dependendo da linguagem de programação, mas o procedimento é o mesmo.

Com base no exemplo anterior é possível identificar algumas idéias iniciais. Na parte superior da Figura 9 são ilustrados os principais conceitos que foram utilizados para criar o metamodelo de refatoração para o KDM. Em primeiro lugar, as partes estruturais de uma

refatoração (classes, métodos, atributos, etc), ou seja, os elementos que são transformados, foram considerados bons candidatos para o reuso (ver Figura 9 ❶) e assim utilizados como base para a criação do metamodelo de refatoração. Em segundo lugar, foi possível identificar na literatura pesquisas que executam uma refatoração utilizando um conjunto de composições de transformações por meio de linguagens de transformações, tais como OCL e ATL (ver Figura 9 ❷). Além disso, também foi constatado que a grande maioria das refatorações são realizadas utilizando operações primárias, tais como: *add*, *remove*, *move*, *create*, entre outras. Tais operações primárias também foram incluídas no metamodelo de refatoração para o KDM (ver Figura 9 ❸). Assim, os engenheiros de modernização podem realizar um *chain of primary operations* permitindo a especificar/criar novas refatorações. O catalogo de refatoração proposto por Fowler (Fowler et al, 1999) também foi considerado durante a criação do metamodelo de refatoração para o KDM como pode ser observado na Figura 9 ❹.

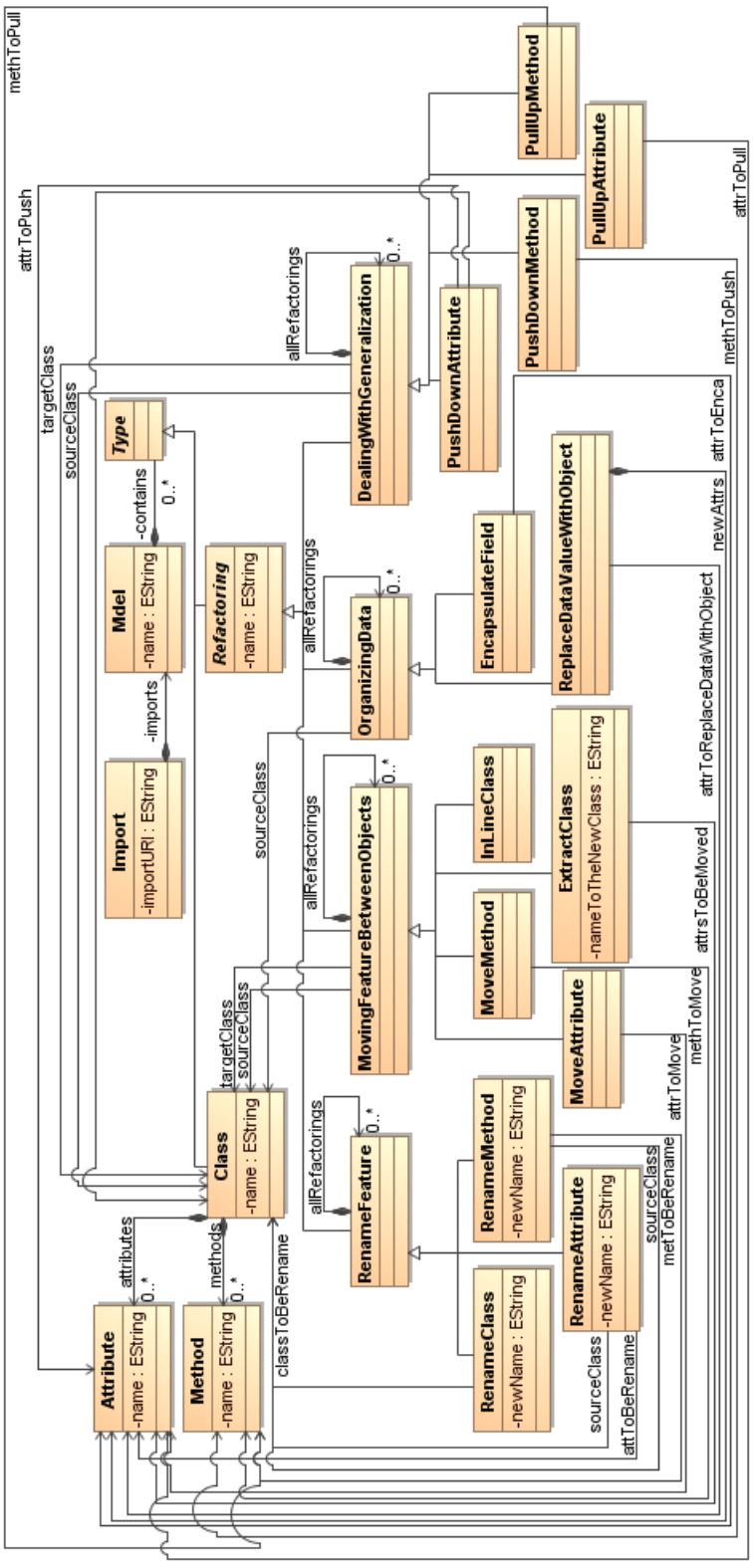


Figura 10: Metamodel de Refatoração para o KDM

Para reutilizar a parte estrutural de uma refatoração, um modelo que represente essa estrutura é necessário. Para o exemplo *RenameElement*, a estrutura basicamente consiste de uma *metaclass* em um *metamodel*. Outras refatorações, tais como *ExtractClass* possuem restrições estruturais mais complexas, como a exigência de que a classe que será extraída esteja contida em um *container object*. O metamodelo de refatorações para o KDM foi definido utilizado EMF (ver Figura 9 ❺). O metamodelo propriamente dito, Refac-KDM, pode ser visualizado na Figura 10. As *metaclasses* que compõem o metamodelo de refatoração, ou seja, Refac-KDM são definidas da seguinte forma:

- *Refactoring* é o elemento raiz utilizado para armazenar todas as informações sobre as refatorações.
- *RenameFeature* corresponde às categorias de refatorações relacionadas a ação de renomear.*RenameFeature* também armazena todas as refatorações relacionadas com a ação de renomear utilizando o *metaattribute allRefactorings*.
- *RenameClass* representa uma instância da refatoração *rename Class*. A semântica para a utilização da *RenameClass* consiste em: (i) especificar uma *ClassUnit* (*classToBeRenamed*), (ii) bem como uma *string* que representa o novo nome (*newName*) da *ClassUnit* após a aplicação da refatoração *renameClassUnit*.
- *RenameAttribute* representa uma instância da refatoração *rename Attribute*. A semântica para a utilização da *metaclass RenameAttribute* consiste em: (i) especificar a *ClassUnit* (*sourceClass*) que armazena todos os atributos e todos os métodos, (ii) o atributo que será renomeado (*attToBeRename*) e (iii) uma *string* que representa o novo nome (*newName*) do atributo após a aplicação da refatoração *renameAttribute*.
- *RenameMethod* representa uma instância da refatoração *rename Method*. A semântica para a utilização da *metaclass RenameMethod* consiste em: (i) especificar a *ClassUnit* (*sourceClass*) que armazena todos os atributos e todos os métodos, (ii) o método que será renomeado (*methToBeRename*) e (iii) uma *string* que representa o novo nome (*newName*) do método após a aplicação da refatoração *renameMethod*.

- *MovingFeatureBetweenObjects* corresponde às categorias de refatorações relacionadas a ação de movimentar características entre objetos.

- *MoveAttribute* representa uma instância da refatoração *move field*. A semântica para a utilização da *metaclass MoveAttribute* consiste em: (*i*) especificar a *ClassUnit* (*sourceClass*) que armazena todos os atributos e todos os métodos, (*ii*) a *ClassUnit* (*targetClass*) que irá receber o atributo a ser movido, (*iii*) e o atributo que será movimentado (*attrToMove*).

- *MoveMethod* representa uma instância da refatoração *move method*. A semântica para a utilização da *metaclass MoveMethod* consiste em: (*i*) especificar a *ClassUnit* (*sourceClass*) que armazena todos os atributos e todos os métodos, (*ii*) a *ClassUnit* (*targetClass*) que irá receber o método a ser movido, (*iii*) e o método que será movimentado (*methToMove*).

- *ExtractClass* representa uma instância da refatoração *extract class*. A semântica para a utilização da *metaclass ExtractClass* consiste em especificar em: (*i*) a *ClassUnit* (*sourceClass*) que armazena todos os atributos e todos os métodos, (*ii*) uma lista de atributos (*attrsToBeExtracted*) e uma lista de métodos (*methsToBeExtracted*) para serem extraídos, (*iii*) uma *string* que representa o nome (*newName*) da classe que foi extraída, (*iv*) bem como um *Package* (*targetPackage*) que irá receber a classe criada após a execução da refatoração.

- **TERMINAR AQUI TODAS AS METACLASSES**

4.6.1 Uma DSL para auxiliar a instânciação de refatorações com base no Refac-KDM

A fim de utilizar plenamente as vantagens das refatorações, os desenvolvedores precisam ter um bom conhecimento de linguagem de programação avançada. Na verdade os desenvolvedores devem estar familiarizados como as semânticas das refatorações (por exemplo, qual(is) é (são) o(s) pré-requisito(s) para a execução de uma refatoração) e como/onde

utilizar programar tais refatorações. A instanciação de uma refatoração utilizando o Refac-KDM é bastante verbosa, complexa e propensa a erros, uma vez que exige conhecimento avançadas de refatoração e habilidades avançadas de programação em relação a API Ecore. Com o objetivo de diminuir a quantidade de código-fonte, esforço e competência necessários para instanciar refatorações utilizando o Refac-KDM, foi desenvolvido uma linguagem especifica de domínio (do inglês, *Domain-Specific Language* - DSL) que auxilia a instanciação de refatorações sistematicamente. Na parte inferior a esquerda da Figura 9 é possível visualizar um exemplo da sintaxe da DSL criada para auxiliar a instanciação do metamodelo Refac-KDM.

A DSL para auxiliar a instanciação do Refac-KDM foi desenvolvida utilizando Xtext 3.4[16]. Xtext é um *framework* do Eclipse[17] que facilita a definição de gramática[18] com a utilização de um metamodelo que foi definido utilizando EMF. Xtext tem como principal objetivo automatizar e agilizar o processo de desenvolvimento de DSLs.

Em Xtext a gramática para especificar DSLs segue uma notação similar ao *Backus–Naur Form* (BNF) chamada de regras do *parser*. Tais regras representam a sintaxe concreta da DSL. Note que para facilitar o entendimento da DSL, trechos da mesma são mostradas em listagens de códigos separados, bem como símbolos para explanar o propósito de uma terminada linha da gramática. Na Listagem de código 1 é ilustrado o primeiro trecho da gramática da DSL.

Listing 1: Gramática da DSL - parte 1

```
❶ grammar com.br.refactoring.xtext.RefacKdm with org.eclipse.xtext.common.Terminals
❷ import ''platform:/resource/com.br.refactoring.RefacKdm/model/RefacKdm.ecore'
❸ import ''http://www.eclipse.org/emf/2002/Ecore'' as ecore
Model:
   ❹ 'model' name=ID
   ❺ (imports+=Import)*
```

[16] https://www.eclipse.org/Xtext/

[17] https://www.eclipse.org

[18] Gramáticas representam a definição formal de um sintaxe textual concreta. Consistem em um conjunto de regras de produção para definir como o *textual input* (, i.e., sentenças) são representadas. Basicamente, as regras de produção podem ser representadas utilizando *Backus–Naur Form* (BNF), por exemplo, *S ::= P1 ... Pn*, essa gramática define um símbolo *S* por um conjunto de expressões *P1 ... Pn*.

```
❻ (contains+=Type)*;
```

A gramática começa com a definição do nome da DSL (RefacKdm) (ver Listagem de código 1 ❶). Em sequência é definido os metamodelos que devem ser importados para serem utilizados durante a criação da DSL, ou seja, o metamodelo RefacKdm❷ e o Ecore❸.

Em seguida é criado a primeira regra. Essa regra começa com a definição da *metaclass* `Model`, o corpo da regra começa após os `:`. Primeiramente para o entendimento da regra, é importante destacar que literais de *string* (que em Xtext podem ser expressas com aspas simples ou duplas) definem palavras-chave da DSL. Como pode ser observado na Listagem de código 1 é esperado a palavra-chave `model`❹ seguido por um `ID`. A gramática que rege o objeto `ID` é definida como uma sequência ilimitada de maiúsculas e minúsculas, números e o carácter de sublinhado, embora possa não começa por um dígito. A gramática que representa o nó `ID`❶ pode ser visualizada na Listagem de código 2.

Listing 2: Gramática da DSL - parte 2

```
❶ terminal ID: ('a'..'z' | 'A'..'Z'|'_')('a'..'z' | 'A'..'Z'|'_'|'0'..'9')*;
```

Ainda na Listagem de código 1 a expressão `(imports+=Import)*`❺ especifica que pode-se instanciar várias instancias da *metaclass* `Import`. O operador estrela, `*`, ilustra que o número de elementos (nesse caso `Import`) é arbitrário; em particular, ele pode ser qualquer número `>=` 0. Operador `+=` por sua vez representa que a propriedade `imports` será uma lista do tipo `Import`. A expressão `(contains+=Type)*`❻ descrita na Listagem de código 1 especifica que pode-se instanciar várias instâncias da *metaclass* `Type`.

Listing 3: Gramática da DSL - parte 3

```
Type:
  ❶ Refactoring | FormalizedDefinition;
Refactoring:
  ❷ (RenameFeature | MovingFeaturesBetweenObjects
    ➥ | OrganizingData | DealingWithGeneralization | PrimaryOperation);
```

A definição da *metaclass* `Type` é interessante, uma vez que `Type` representa uma *metaclass* abstrata com dois subtipos, como pode ser observado na Listagem de código 3.

Ainda nessa listagem, é possível observar que o operador, `|`, é utilizado para expressar alternativas ❶, o que é traduzido para o conceito de herança no metamodelo do Refac-KDM. A definição da *metaclass* `Refactoring` é ilustrada por um conjunto de *submetaclass* ❷ que são expressas na gramática da DSL pelo operador, `|`, a saber: `RenameFeature`, `MovingFeaturesBetweenObjects`, `OrganizingData`, `DealingWithGeneralization` and `Create`.

Listing 4: Gramática da DSL - parte 4

```
RenameFeature:
  ❶ 'Rename_Feature' name = ID '{'
    ➥ ❷ (allRefactorings+=RenameClass)*
    ➥ ❸ (allRefactorings+=RenameAttribute)*
    ➥ ❹ (allRefactorings+=RenameMethod)*
   '}';
```

A definição da *metaclass* `RenameFeature` é ilustrada na Listagem de código 4. Como pode ser observado essa regra começa com a definição da palavra-chave `Rename_Feature` seguida por um `ID` ❶. As expressões descritas em ❷, ❸ e ❹ representam que pode haver qualquer número de instancias das *metaclasses* `RenameClass`, `RenameAttribute` e `RenameMethod`, respectivamente.

Listing 5: Gramática da DSL - parte 5

```
RenameClass:
  ❶ 'Refactoring_RenameClass' name = ID '{'
    ➥ ❷ 'classToBeRenamed' ':' classToBeRenamed = [ClassUnit]
    ➥ ❸ 'newName' ':' newName = ID
  '}';
RenameAttribute:
  'Refactiong_RenameAttribute' name = ID '{'
    ➥ 'sourceClass' ':' sourceClass = [ClassUnit]
    ➥ ❹ 'attributeToBeRenamed' ':' attributeToBeRenamed = [StorableUnit]
    ➥ 'newName' ':' newName = ID
  '}';
RenameMethod:
  'Refactoring_RenameMethod' name = ID '{'
    ➥ 'sourceClass' ':' sourceClass = [ClassUnit]
    ➥ ❺ 'methodToBeRenamed' ':' methodToBeRenamed = [MethodUnit]
    ➥ 'newName' ':' newName = ID
  '}';
```

A definição da *metaclass* `RenameClass` é ilustrada na Listagem de código 5. A regra começa com a definição da palavra-chave `Refactoring_RenameClass` seguida por um `ID` e a chave, `{` ❶. Essa chave representa o inicio do escopo relacionado a refatoração *Rename Class*, ou seja, é um contexto delimitante aos quais valores e expressões estão associados a refatoração. Em seguida a palavra-chave `classToBeRenamed` é esperada, seguido por `:` ❷. Posteriormente uma instância da `ClassUnit` que será renomeada deve ser atribuída. Em Xtext, a notação `classToBeRenamed = [ClassUnit]` indica que é espero uma instância da *metaclass* `ClassUnit`, e não apenas um nome. Em seguida a palavra-chave `newName` é esperado, seguido por `:`, finalmente o novo nome da classe deve ser especificado ❸.

A gramática que define a semântica da refatoração *rename attribute* é similar a anterior. Como visualizado na Listagem de código 5 ❹, a única diferença é que deve-se atribuir uma instância da *metaclass* `StorableUnit`, ou seja, o atributo que almeja-se renomear. A semântica da refatoração *rename method* é praticamente a mesma que a refatoração *rename attribute*. A única diferença é que deve-se atribuir uma instância da *metaclass* `MethodUnit` ❺.

Listing 6: Gramática da DSL - parte 6

```
MovingFeaturesBetweenObjects:
  ❶ 'MovingFeaturesBetweenObjects' name = ID '{'
    ➥ ❷ (allRefactorings+=MoveAttribute)*
    ➥ ❸ (allRefactorings+=MoveMethod)*
    ➥ ❹ (allRefactorings+=ExtractClass)*
    ➥ ❺ (allRefactorings+=InlineClass)*
  '}';
```

A definição da *metaclass* `MovingFeaturesBetweenObjects` é ilustrada na Listagem de código 6. Perceba que tal definição é similar a da *metaclass* `RenameFeature` (ver Listagem de código 4). Como pode ser observado a regra começa com a definição da palavra-chave `MovingFeaturesBetweenObjects` seguida por um `ID` ❶. As expressões descritas em ❷,

❸, ❹ e ❺ representam que pode haver qualquer número de instancias das *metaclasses* `MoveAttribute`, `MoveMethod`, `ExtractClass` e `InlineClass`, respectivamente.

Listing 7: Gramática da DSL - parte 7

```
MoveAttribute:
  ❶ 'Refactoring_MoveAttribute' name = ID '{'
    ➥ 'sourceClass' ':' sourceClass = [ClassUnit]
    ➥ 'targetClass' ':' targetClass = [ClassUnit]
    ➥ 'attributeToBeMoved' ':' attributeToBeMoved = [StorableUnit]
  '}';
MoveMethod:
  'Refactoring_MoveMethod' name = ID '{'
    ➥ 'sourceClass' ':' sourceClass = [ClassUnit]
    ➥ 'targetClass' ':' targetClass = [ClassUnit]
    ➥ ❷ 'methodToBeMoved' ':' methodToBeMoved = [MethodUnit]
  '}';
ExtractClass:
  'Refactoring_ExtractClass' name = ID '{'
    ➥'sourceClass' ':' sourceClass = [ClassUnit]
    ➥ ❸ 'attribute(s)ToBeMoved' ':'
      ➥ '{'attributesToBeMoved += [StorableUnit]
        ➥ (','(attributesToBeMoved += [StorableUnit]))*'}'
    ➥ ❹ 'nameToTheNewClass' ':' newName = ID
  '}';
InlineClass:
  'Refactoring_InlineClass' name = ID '{'
    ➥ ❺ 'classToGetAllFeatures' ':' classToGetAllFeatures = [ClassUnit]
    ➥ ❻ 'classToRemove' ':' classToRemove = [ClassUnit]
  '}';
```

A semântica da refatoração *move attribute* é definido na gramática da Listagem de código 7 ❶. Como pode ser observado é esperado a palavra-chave `Refactoring_MoveAttribute` seguida por um `ID` e `{`. Posteriormente deve-se especificar a palavra-chave `sourceClass`, seguido por `:`. Uma instância da `ClassUnit` que contém o atributo que será movimentado deve ser atribuída. Em seguida, deve-se indicar a `targetClass`. Novamente, uma instância de *ClassUnit* deve ser atribuída, ou seja, a classe que receberá o atributo. Por fim, deve-se especificar qual atributo realmente será movimentado. Assim, a palavra-chave

`attributeToBeMoved` deve ser especificada, seguido por `:`. Uma instância de `StorableUnit` que representa o atributo que será movimentado deve ser atribuída.

A regra que descreve a refatoração `move method` é similar a refatoração *move attribute* como pode ser observado na Listagem de código 7, duas principais diferenças podem ser notadas. A primeira diferença é a primeira palavra-chave da regra `Refactoring_MoveMethod`. A segunda diferença pode ser ser observado na expressão `methodToBeMoved = [MethodUnit]` ❷. Essa expressão mostra que deve-se especificar o método que será movimentado.

A refatoração *extract class* é definida na sequência. A regra sempre começa com a palavra-chave `Refactoring_ExtractClass` seguida por um `ID` e `{`. Similarmente, também deve-se especificar a palavra-chave `sourceClass` seguido por `:`. Uma instância da `ClassUnit` que contém os atributos que serão extraídos deve ser especificada. A diferença pode ser observado na expressão ilustrada pelo símbolo ❸. Essa expressão inicia-se com a palavra-chave `attribute(s)ToBeMoved` seguido por `:` e `{`, nesse contexto as chaves representam o(s) atributo(s) que será(ão) extraído(s). Em seguida a expressão descreve que ao menos uma instância de `StorableUnit` deve ser atribuída, porém, pode-se especificar mais do que uma instância de `StorableUnit` para ser extraído, simplesmente separando cada `StorableUnit` por virgula (`,`). Em seguida, a palavra-chave `nameToTheNewClass` deve ser especifica seguido por `:` e `ID`.

Para satisfazer a semântica da gramática relacionada a refatoração *inline class* deve-se primeiramente especificar a palavra-chave `Refactoring_InlineClass` seguida por um `ID` e `{`. Em seguida, deve-se especificar a palavra-chave `classToGetAllFeatures` e `:` (ver ❺). Uma instância de uma `ClassUnit` deve ser atribuída. Finalmente, deve-se especificar a palavra-chave `classToRemove`, `:` e uma instância de `ClassUnit` que representa a classe que será removida (ver ❺).

Listing 8: Gramática da DSL - parte 8

```
OrganizingData:
  ❶ 'OrganizingData' name = ID '{'
    ➥ ❷ (allRefactorings+=ReplaceDataValueWithObject)*
    ➥ ❸ (allRefactorings+=EncapsulateField)*
```

```
  '}';
```

A definição da *metaclass* `OrganizingData` é ilustrada na Listagem de código 8. Note que a regra começa com a definição da palavra-chave `OrganizingData` seguida por um `ID` ❶. As expressões descritas em ❷ e ❸ representam que pode haver qualquer número de instancias das *metaclasses* `ReplaceDataValueWithObject` e `EncapsulateField`, respectivamente.

Listing 9: Gramática da DSL - parte 9

```
ReplaceDataValueWithObject:
  'Refactoring_ReplaceDataValueWithObject' name = ID '{'
    ➥ ❶ 'sourceClass' ':' sourceClass = [ClassUnit]
    ➥ ❷ 'attributeToReplaceDataWithObject' ':'
      ➥ attributeToReplaceDataWithObject = [StorableUnit]
    ➥ ❸ 'newAttributes' ':' '{'
      ➥ (newAttributes+=StorableUnit)*
    '}'
  '}';
EncapsulateField:
  'Refactoring_EncapsulateField' name = ID '{'
    ➥ ❹' sourceClass' ':' sourceClass = [ClassUnit]
    ➥ ❺ 'attributeToEncapsulate' ':' attributeToEncapsulate = [StorableUnit]
  '}';
```

A semântica da refatoração *replace data value with object* é definido na gramática da Listagem de código 9. Note que é esperado a palavra-chave `Refactoring_ReplaceDataValueWithObject` seguida por um `ID` e `{`. Posteriormente deve-se especificar a palavra-chave `sourceClass`, `:` e uma instância da `ClassUnit` deve ser atribuída (ver ❶). Em seguida, deve-se especificar a palavra-chave `attributeToReplaceDataWithObject`, `:` e uma instância de *StorableUnit* também deve ser atribuído (ver ❷). Por fim, deve-se especificar a palavra-chave `newAttributes`, `:` seguida por um conjunto `StorableUnit` (ver ❸).

A refatoração *encapsulate field* é definida na sequência. A regra sempre começa com a palavra-chave `Refactoring_EncapsulateField` seguida por um `ID` e `{`. Similarmente, também deve-se especificar a palavra-chave `sourceClass` seguido por `:` e uma instância de

`ClassUnit` (ver ❹). Em seguida, deve-se especificar a palavra-chave `attributeToEncapsulate`, `:` e uma instância de *StorableUnit*, o qual será encapsulado (ver ❺).

Listing 10: Gramática da DSL - parte 10

```
DealingWithGeneralization:
  ❶ 'DealingWithGeneralization' name = ID '{'
    ➥ ❷ (allRefactorings+=PushDownAttribute)*
    ➥ ❸ (allRefactorings+=PushDownMethod)*
    ➥ ❹ (allRefactorings+=PullUpAttribute)*
    ➥ ❺ (allRefactorings+=PullUpMethod)*
  '}';
```

A definição da *metaclass* `DealingWithGeneralization` é ilustrada na Listagem de código 10. Note que a regra começa com a definição da palavra-chave `DealingWithGeneralization` seguida por um `ID` e a chave, `{` ❶. As expressões descritas em ❷, ❸, ❹ e ❺ representam que pode haver qualquer número de instancias das *metaclasses* `PushDownAttribute`, `PushDownMethod`, `PullUpAttribute` e `PullUpMethod`, respectivamente.

Listing 11: Gramática da DSL - parte 11

```
PushDownAttribute:
  'Refactoring_PushDownAttribute' name = ID '{'
    ➥ 'sourceClass' ':' sourceClass = [ClassUnit]
    ➥ 'attributeToPushDown' ':' attributeToBePushed = [StorableUnit]
    ➥ 'targetClass' ':' targetClass = [ClassUnit] '}';
PushDownMethod:
  'Refactoring_PushDownMethod' name = ID '{'
    ➥ 'sourceClass' ':' sourceClass = [ClassUnit]
    ➥ 'methodToPushDown' ':' methodToBePushed = [MethodUnit]
    ➥ 'targetClass' ':' targetClass = [ClassUnit] '}';
PullUpAttribute:
  'Refactoring_PullUpAttribute' name = ID '{'
    ➥ 'sourceClass' ':' sourceClass = [ClassUnit]
    ➥ 'attributeToPullUp' ':' attributeToBePulled = [StorableUnit]
    ➥ 'targetClass' ':' targetClass = [ClassUnit] '}';
PullUpMethod:
  'Refactoring_PullUpMethod' name = ID '{'
    ➥ 'sourceClass' ':' sourceClass = [ClassUnit]
    ➥ 'methodToPullUp' ':' methodToBePulled = [MethodUnit]
    ➥ 'targetClass' ':' targetClass = [ClassUnit] '}';
```

As regras que definem a semântica das *metaclasses* `PushDownAttribute`, `PushDownMethod`, `PullUpAttribute` e `PullUpMethod` estão definidas na Listagem de código 11.

Listing 12: Gramática da DSL - parte 12

```
❶ PrimaryOperation:
  (CreateClass | CreateAttribute | CreateMethod |
    ➥ RemoveClass | RemoveAttribute | RemoveMethod |
      ➥ MoveClass | CreateInheritance)
;
CreateClass:
  'Refactoring_CreateClass' '{'
    ➥ ❷ 'package' ':' package = [Package]
    ➥ ❸ 'class' ':' name = ID
  '}';
CreateAttribute:
  'Refactoring_CreateAttribute' '{'
    ➥ 'class' ':'className = ID
    ➥ '@' attribute = ID
  '}';
```

A definição da *metaclass* `PrimaryOperation` é ilustrada na Listagem de código 12 ❶. Essa *metaclass* é responsável por representar algumas das operações básicas que podem constituir uma refatoração (*add*, *remove*, *move*, *create*). Na Listagem de código 12 apenas algumas dessas operações são destacadas. A operação *create class* é destacado na Listagem de código 12 ❷, perceba que a regra começa com a definição da palavra-chave `Refactoring_CreateClass` seguida pela outra palavra-chave `package` e `:`. Em sequência a palavra-chave `class` e `:` devem ser especificadas (ver ❸). A operação *create attribute* é similar a anterior.

Listing 13: Gramática da DSL - parte 13

```
❶ FormalizedDefinition:
  'Refactoring_FormalizedDefinition' '{'
    ➥ (allDefinition+=BooleanExpression)*
  '}';
BooleanExpression:
  ❷ AndExpression | OrExpression | NotExpression;
AndExpression:
    ➥ 'AndExpr' ':' ref1 = Refactoring
```

```
    ➥ '&&' ref2 = Refactoring;
OrExpression:
    ➥ 'OrExpr' ':' ref1 = Refactoring
    ➥ '| |' ref2 = Refactoring ;
NotExpression:
    ➥ 'NotExpr' ':' ref = Refactoring;
```

A definição da *metaclass* `FormalizedDefinition` é ilustrada na Listagem de código 13 ❶. Utilizando as regras ilustradas na Listagem de código 13 ❷ os engenheiros de modernização podem realizar um *chain of primary operations* permitindo a especificar/criar novas refatorações. Por exemplo, é possível criar expressões da seguinte forma, `createClass && (moveAttribute && createMethod)`.

4.6.2 Uma notação gráfica (concreta) da DSL para auxiliar a instânciação de refatorações com base no Refac-KDM

Usualmente os modelos gráficos são mais intuitivos de se utilizar. Assim, com o intuito de prover maior facilidade na instanciação do metamodelo Refac-KDM criou-se uma notação gráfica da DSL. Essa notação gráfica também auxilia o engenheiro de modernização, uma vez que o mesmo não precisa escrever as refatorações utilizando uma linguagem textual, ou seja, apenas elementos gráficos são utilizados para criar e definir as refatorações. Posteriormente, a IDE automaticamente fornece uma sincronização entre a DSL concreta e a DSL textual, provendo assim uma multi-visualização da DSL, gráfica e textual.

A notação gráfica é composta por elementos que permitem a visualização e a edição das informações definidas na DSL de forma gráfica. A construção da sintaxe concreta de uma DSL é dependente da ferramenta adotada para esse fim. Por exemplo, o *Graphical Modeling Framework* (GMF)[19] utiliza três tipos de modelos para definir a notação gráfica da DSL: (*i*) *Gmfgraph*, (*ii*) *Gmftool* e (*iii*) Gmfmap. Na parte inferior a direita da Figura 9 é possível visualizar um exemplo da notação gráfica que foi criada para dar suporte a DSL apresentada na Seção 4.6.1.

[19] `https://www.eclipse.org/modeling/gmp/`

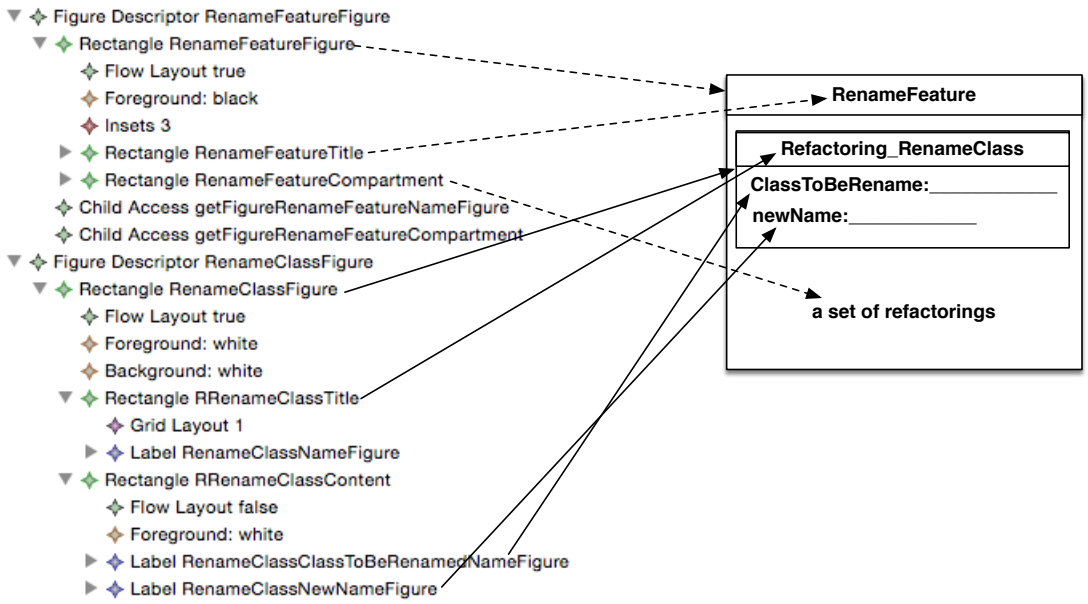


Figura 11: Definição da notação gráfica da DSL

No modelo *Gmfgraph* são definidos os componentes da notação gráfica da DSL que correspondem às refatorações e seus relacionamentos. Cada componente gráfico pode ser formado pela junção de vários componentes de forma aninhada. Por exemplo, na Figura 11 é mostrado que o conjunto de refatoração relacionado a *rename feature* é representada graficamente por um retângulo (*Rectangle RenameFeatureFigure*) composto de outros retângulos que contêm um rótulo (*RenameFeatureTitle*) e um compartimento (*RenameFeatureCompartment*). Similarmente, a refatoração *rename class* também é representada graficamente por um retângulo (*Rectangle RenameClassFigure*) composto de outros retângulos que contêm um rótulo (*RRenameClassTitle*) e um compartimento (*RRenameClassContent*) que também contêm dois outros rótulos (*RenameClassClassToBeRenamedNameFigure* e (*RenameClassNewNameFigure*)).

No modelo *Gmftool* são definidos os itens de menu dos elementos e dos relacionamentos que podem ser utilizados. Na Figure 12 é mostrado o modelo *Gmftool* da DSL, no qual são definidos as refatorações. Note que foi criado um item para cada uma das refatorações: *extract class*, *move attribute*, *move method*, etc.

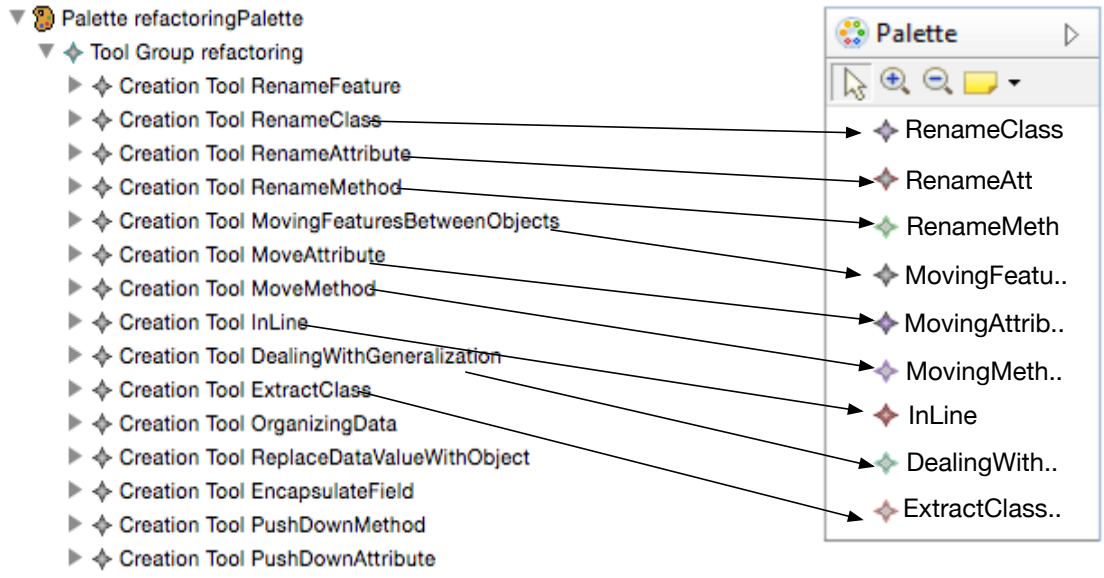


Figura 12: Definição dos itens do menu da DSL

No modelo *Gmfmap* os elementos do metamodelo são combinados com os seus respectivos componentes da notação gráfica e itens de menu. Na Figura 13 é ilustrado um exemplo de modelo *Gmfmap* que combina as *metaclasses* e os relacionamentos do metamodelo (Figura 10) com a notação gráfica definida no modelo *Gmfgraph* (Figura 11) e com os itens de menu definidos no modelo *Gmftool* (Figura 12). Nesse modelo cada refatoração é definida com a inclusão de um *Top Node Reference* contendo um *Node Mapping*. *Feature Labels* são acrescentados para permitir a visualização e a edição dos nomes e dos tipos das refatorações.

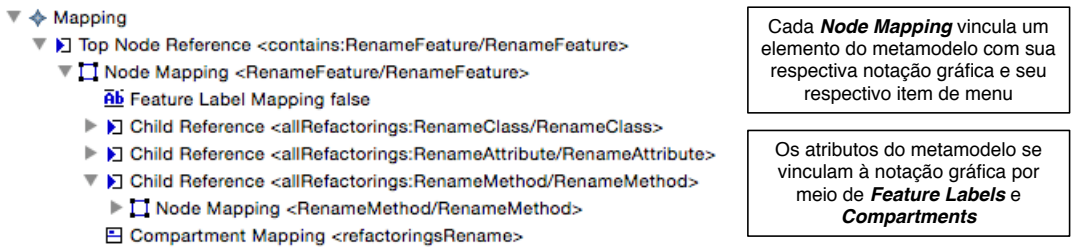


Figura 13: Modelo *Gmfmap da DSL*

Na Figuras 14 e 15 são mostrados o uso da DSL textual e gráfica para especificar algumas refatorações, respetivamente. É importante destacar que todas as alterações realizadas

na DSL textual implica na sincronização automática da DSL gráfica. Da mesma forma, se o engenheiro de software preferir utilizar a notação gráfica para instanciar refatorações a DSL textual também é sincronizada automaticamente.

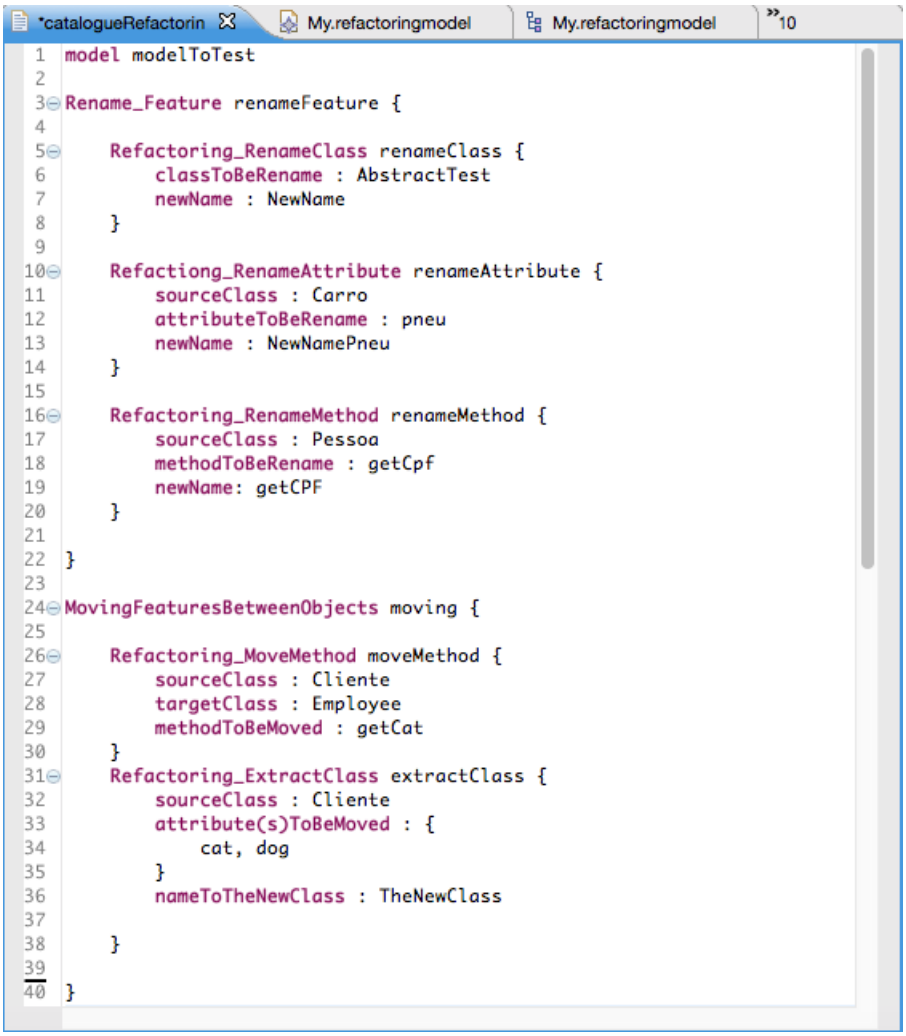


Figura 14: Exemplo de utilização da DSL textual

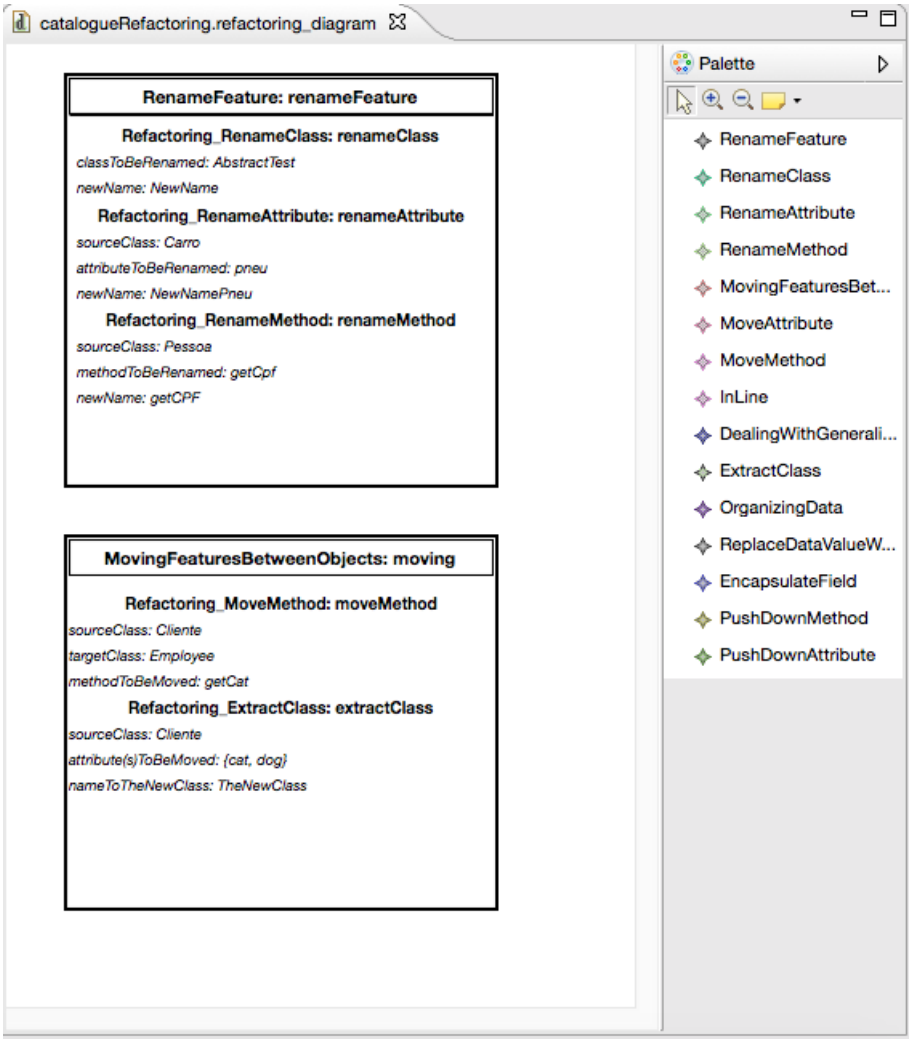


Figura 15: Exemplo de utilização da DSL gráfica

4.7 Publicações, Participação em Eventos Científicos e Apresentações de Trabalhos

Nesta seção são apresentadas as publicações, participações em eventos científicos e apresentações de trabalho realizadas pelo bolsista durante o segundo período de vigência da bolsa de doutorado.

A seguir são descritos as publicações que foram conduzidas pelo bolsista. Vale resssaltar que alguns artigos foram desenvolvidos em parceria com outros membros do grupo de pesquisa. Todos os artigos descrito abaixo foram aceitos e apresentados por um dos autores.

É importante salientar que os artigos que foram conduzidos com outros membros do grupo de pesquisa e que não estão totalmente ligado a projeto do outorgado nenhum recurso do bolsista foi utilizado (Reserva técnica), tais como: inscrição, viagem ou hospedagem. Porém, destaca-se que em todos os artigos é mencionando o número do processo FAPESP do outorgado.

- **Trabalhos completos publicados em capítulos de livros**

  1. Viana, Matheus ; Penteado, Rosângela ; Prado, Antônio do ; **Durelli, Rafael** . Developing Frameworks from Extended Feature Models. Advances in Intelligent Systems and Computing. 1ed.: Springer International Publishing, 2014, v. 263, p. 263-284.[20].

- **Trabalhos completos publicados em Journal**

  1. GOTTARDI, THIAGO ; **DURELLI, RAFAEL** ; LÓPEZ, ÓSCAR ; DE CAMARGO, VALTER . Model-based reuse for crosscutting frameworks: assessing reuse and maintenance effort. Journal of Software Engineering Research and Development, v. 1, p. 4, 2013.

- **Trabalhos completos publicados em anais de congressos**

  1. PARREIRA JUNIOR, P. A. ; VIANA, M. C. ; **DURELLI, R. S.** ; CAMARGO, V. V. ; COSTA, H. A. X. ; PENTEADO, R. A. D. . Concern-Based Refactorings Supported by Class Models to Reengineer Object-Oriented Software into Aspect-Oriented Ones. In: International Conference on Enterprise Information Systems (ICEIS), 2013, ANGERS/FR. XV International Conference on Enterprise Information Systems, 2013. [20].
  2. VIANA, M. C. ; **DURELLI, R. S.** ; PENTEADO, R. A. D. ; PRADO, A. F. . F3: From features to frameworks.. In: International Conference on En-

[20]Vale ressaltar que esse artigo é uma atividade complementar e não relacionada ao projeto de pesquisa. Esse artigo foi desenvolvido em parceria com outros membros do grupo de pesquisa. Tal artigo foi aceito e apresentado por um dos autores. Além disso é importante salientar que não foi usado nenhum recurso disponível em Reserva Técnica para inscrição, viagem ou hospedagem do bolsista.

terprise Information Systems (ICEIS), 2013, ANGERS/FR. XV International Conference on Enterprise Information Systems, 2013. [20].

3. VIANA, M. C. ; PENTEADO, R. A. D. ; PRADO, A. F. ; **DURELLI, R. S**. . An Approach to Develop Frameworks from Feature Models. In: International Conference on Information Reuse and Integration, 2013, San Francisco. An Approach to Develop Frameworks from Feature Models, 2013 [20].

4. VIANA, M. C. ; PENTEADO, R. A. D. ; PRADO, A. F. ; **DURELLI, RAFAEL S**. . F3T: From Features to Frameworks Tool. In: XXVII Simpósio Brasileiro de Engenharia de Software (SBES 2013), 2013, Brasília. F3T: From Features to Frameworks Tool, 2013 [20].

5. SANTIBÁÑEZ, DANIEL S. M. ; **DURELLI, RAFAEL S.** ; CAMARGO, V. V. . CCKDM - A Concern Mining Tool for Assisting in the Architecture-Driven Modernization Process. In: XXVII Simpósio Brasileiro de Engenharia de Software - XXVII Sessão de Ferramenta, 2013, Brasilia. Simpósio Brasileiro de Engenharia de Software, 2013. [20].

6. SANTIBANEZ, D. S. M. ; **DURELLI, RAFAEL S.** ; CAMARGO, V. V. . A Combined Approach for Concern Identification in KDM models. In: Latin American Workshop on Aspect-Oriented Software Development (LA-WASP), 2013, Brasília. Congresso Brasileiro de Software: Teoria e Prática (CBSoft), 2013. [20] [21].

7. PINTO, Victor Hugo S. C. ; **DURELLI, R. S.** ; OLIVEIRA, A. L. ; CAMARGO, V. V. . Evaluating the Effort for Modularizing Multiple-Domain Frameworks towards Framework Product Lines with Aspect-Oriented Programming and Model-Driven Development. In: International Conference on Enterprise Information Systems (ICEIS), 2014, Lisboa. International Conference on Enterprise Information Systems (ICEIS), 2014. [20]

[21] Os autores foram convidados para publicar uma versão estendida desse artigo na revista *Journal of the Brazilian Computer Society* (Qualis B1)

8. DIAS, D. R. C. ; **DURELLI, R. S.** ; BREGA, J. R. F. ; GNECCO, B. B ; TREVELIN, L. C. ; GUIMARAES, M. P. . Data Network in Development of 3D Collaborative Virtual Environments: A Systematic Review. In: The 14th International Conference on Computational Science and Applications (ICCSA 2014), 2014, Guimarães. The 14th International Conference on Computational Science and Applications (ICCSA 2014), 2014. [20]

9. **DURELLI, R. S.** ; SANTIBANEZ, D. S. M. ; DELAMARO, MÁRCIO E. ; CAMARGO, V. V. . Towards a Refactoring Catalogue for Knowledge Discovery Metamodel. In: IEEE International Conference on Information Reuse and Integration, 2014, San Francisco. IEEE International Conference on Information Reuse and Integration, 2014. p. 1-8. [22].

10. **DURELLI, R. S.** ; SANTIBANEZ, D. S. M. ; MARINHO, B. S. ; HONDA, R. R. ; DELAMARO, M. E. ; ANQUETIL, N. ; CAMARGO, V. V. . A Mapping Study on Architecture-Driven Modernization. In: IEEE International Conference on Information Reuse and Integration, 2014, San Francisco. IEEE International Conference on Information Reuse and Integration, 2014. p. 1-8. [22]

11. MARINHO, B. S. ; CAMARGO, V. V. ; HONDA, R. R. ; **DURELLI, R. S.** . KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel. In: 28th Brazilian Symposium on Software Engineering (SBES), 2014, Maceió. 28th Brazilian Symposium on Software Engineering (SBES), 2014. p. 1-10. [20]

12. MARINHO, B. S. ; **DURELLI, RAFAEL S.** ; HONDA, R. R. ; CAMARGO, V. V. . Investigating Lightweight and Heavyweight KDM Extensions for Aspect-Oriented Modernization. In: 11th Workshop on Software Modularity (WMod) – Brazilian Conference on Software: theory and practice, 2014, maceio. 11th Workshop on Software Modularity (WMod) – Brazilian Conference on Software: theory and practice, 2014. [20]

[22]Foi usado recurso disponível em Reserva Técnica para pagar a inscrição desse evento, passagem áerea e diários.

13. **DURELLI, RAFAEL S.** ; MARINHO, B. S. ; HONDA, R. R. ; DELAMARO, MÁRCIO E. ; CAMARGO, V. V. . KDM-RE: A Model-Driven Refactoring Tool for KDM.. In: II Workshop on Software Visualization, Evolution and Maintenance – Brazilian Conference on Software: theory and practice, 2014, Maceio. II Workshop on Software Visualization, Evolution and Maintenance – Brazilian Conference on Software: theory and practice, 2014. p. 1-8. [22]

5 Atividades Previstas para o Próximo Período

De acordo com a *timeLine* apresentado na Seção 4 é possível salientar algumas atividades previstas para as etapas seguintes. Dessa forma, a seguir são descritas as atividades que o bolsista pretende conduzir durante as próximas etapas.

1. **Redação de artigos:** pretende-se continuar produzindo artigos, com o intuito de documentar o progresso do projeto em questão.

2. **Implementar Refatorações Arquiteturais:** o objetivo é gerar um KDM em que o pacote *Structure* represente a arquitetura atual, contendo todos os relacionamentos existentes entre seus elementos. Para isso, um algoritmo deverá analisar o KDM modificado para identificar todos os relacionamentos existentes entre elementos de mais baixo nível (classes, interfaces, métodos) e criar relacionamentos de mais alto nível entre os elementos arquiteturais. As atividades de refatoração possuem então a responsabilidade de transformar esse KDM com o objetivo de solucionar os problemas arquiteturais encontrados. No caso das refatorações específicas de arquitetura, uma atividade que deve ser feita é a migração de elementos do código-fonte entre elementos arquiteturais, como camadas, componentes e subsistemas. Por exemplo, movendo uma classe de uma pacote para outro.

3. **Estudo Experimental:** após a implementação da abordagem proposta pelo bolsista, pretende-se avaliar o ambiente de modernização resultante. Pretende-se realizar dois tipos de experimentos. O primeiro consiste em realizar um conjunto de

estudos de casos para investigar a viabilidade da abordagem proposta, bem como avaliar o uso das funcionalidades do apoio computacional para fornecer suporte à modernização de sistemas legados. O segundo experimento consistem em realizar avaliações controladas utilizando a metodologia experimental Wohlin et al (2000), a fim de avaliar o impacto da abordagem proposta, bem como do apoio computacional relacionado a eficiência e impacto das equipes e também a qualidade em termos de modularidade, reuso e manutenibilidade dos sistema resultantes durante a atividade de modernização.

4. **Evolução ferramental:** realizar melhorias nas ferramentas para aumentar a automatização da abordagem em questão. Resultados do estudo experimental podem auxiliar a elicitar as limitações e sugerir tais melhorias.

5. **Redação da Tese.**

6. **Defesa do Doutorado.**

Referências

Bézivin, J. On the unification power of models. *Software and Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.

Canfora, G.; Cimitile, A.; Munro, M. Re2: Reverse-engineering and reuse re-engineering. *Journal of Software Maintenance: Research and Practice*, vol. 6, pp. 53–72, 1994.

Canfora, G.; Di Penta, M.; Cerulo, L. Achievements and challenges in software reverse engineering. *Commun. ACM*, vol. 54, no. 4, pp. 142–151, 2011.

Chikofsky, E.; Cross, J.H., I. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, vol. 7, no. 1, pp. 13–17, 1990.

Demeyer, S.; Du Bois, B.; Verelst, J. Refactoring - Improving Coupling and Cohesion of Existing Code. *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, 2004.

Durelli, R.; Santos M, B.; Honda, R.; Delamaro, M.; Camargo, V. Kdm-re: A model-driven refactoring tool for kdm. *Workshop on Software Visualization, Evolution and Maintenance*, pp. 1–8, 2014a.

Durelli, R. S.; Santibáñez, D. S. M.; Delamaro, M. E.; Camargo, V. V. A mapping study on architecture-driven modernization. In: *IEEE 15th International Conference on Information Reuse and Integration (IRI)*, 2014b, pp. 168–178.

Durelli, R. S.; Santibáñez, D. S. M.; Delamaro, M. E.; Camargo, V. V. Towards a refactoring catalogue for knowledge discovery metamodel. In: *IEEE 15th International Conference on Information Reuse and Integration (IRI)*, 2014c, pp. 158–168.

Eilam, E. *Reversing: Secrets of Reverse Engineering*. Wiley, 624 pp., 2005.

Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

Griffith, I.; Wahl, S.; Izurieta, C. Evolution of legacy system comprehensibility through automated refactoring. In: *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, New York, NY, USA: ACM, 2011, pp. 35–42.

ISO/IEC DIS 19506 Knowledge Discovery Meta-model (KDM), v1.1. Online, `http://www.iso.org/iso/catalogue_detail.htm?csnumber=32625` - Acessado em 03/07/2012, 2011.

Kent, S. Model driven engineering. In: Butler, M.; Petre, L.; Sere, K., eds. *Integrated Formal Methods*, Springer Berlin Heidelberg, pp. 286–298, 2002.

Kitchenham, B.; Pearl Brereton, O.; Budgen, D.; Turner, M.; Bailey, J.; Linkman, S. Systematic literature reviews in software engineering - a systematic literature review. *Information and Software Technology*, vol. 51, pp. 7–15, 2009.

Kleppe, A. G.; Warmer, J.; Bast, W. *Mda explained: The model driven architecture: Practice and promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

Mens, T.; Tourwe, T. A Survey of Software Refactoring. *Transactions on Software Engineering, IEEE*, vol. 30, 2004.

OMG Object Management Group (OMG) Architecture-Driven Modernisation. Disponível em: *http://www.omgwiki.org/admtf/doku.php?id=start*, (Acessado 2 de Agosto de 2012), 2012a.

OMG Object Management Group (OMG) Model-Driven Development — OMG Available Specification without Change Bars. Disponível em: *http://www.omg.org*, (Acessado 2 de Agosto de 2012), 2012b.

Opdyke, W. F. *Refactoring Object-Oriented Frameworks*. Ph.D. Thesis, University of Illinois, 1992.

Perez-Castillo, R.; Garcia-Rodriguez de Guzman, I.; Piattini, M. Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, vol. 33, no. 6, pp. 519–532, 2011a.

Pérez-Castillo, R.; de Guzmán, I. G. R.; Caballero, I.; Piattini, M. Software modernization by recovering web services from legacy databases. *Journal of Software: Evolution and Process*, vol. 25, no. 5, pp. 507–533, 2013.

Perez-Castillo, R.; de Guzmán, I. G.-R.; Piattini, M. *Architecture driven modernization.* San Francisco, CA, USA: Information Science Reference, 2011b.

Premerlani, W.; Blaha, M. An approach for reverse engineering of relational databases. In: *Proceedings of Working Conference on Reverse Engineering*, 1993, pp. 151–160.

Rugaber, S.; Stirewalt, K. Model-driven reverse engineering. *Software, IEEE*, vol. 21, no. 4, pp. 45–53, 2004.

Smith, J.; Nair, R. *Software Reuse and Reverse Engineering in Practice.* Chapman and Hall Ltd, 608 pp., 1992.

Thomas, D. Mda: revenge of the modelers or uml utopia? *Software, IEEE*, pp. 15 – 17, 2004.

Ulrich, W. M.; Newcomb, P. *Information systems transformation: Architecture-driven modernization case studies.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010a.

Ulrich, W. M.; Newcomb, P. *Information systems transformation: Architecture-driven modernization case studies.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010b.

Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M. C.; Regnell, B.; Wesslén, A. *Experimentation in software engineering: an introduction.* Norwell, MA, USA: Kluwer Academic Publishers, 2000.

6 Anexos

Anexo 1: Comprovante da publicação no *Journal of Software Engineering Research and Development* (JSERD)

- Uma cópia do artigo é apresentado a seguir (a partir da próxima página).

Gottardi *et al. Journal of Software Engineering Research and Development* 2013, **1**:4
www.jserd.com/content/1/1/4

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH Open Access

Model-based reuse for crosscutting frameworks: assessing reuse and maintenance effort

Thiago Gottardi[1*], Rafael Serapilha Durelli[2], Óscar Pastor López[3] and Valter Vieira de Camargo[1]

*Correspondence:
thiago_gottardi@dc.ufscar.br
[1]Departmento de Computação, Universidade Federal de São Carlos, Caixa Postal 676, 13.565-905, São Carlos, São Paulo, Brazil
Full list of author information is available at the end of the article

Abstract

Background: Over the last years, a number of researchers have investigated how to improve the reuse of crosscutting concerns. New possibilities have emerged with the advent of aspect-oriented programming, and many frameworks were designed considering the abstractions provided by this new paradigm. We call this type of framework Crosscutting Frameworks (CF), as it usually encapsulates a generic and abstract design of one crosscutting concern. However, most of the proposed CFs employ white-box strategies in their reuse process, requiring two mainly technical skills: (i) knowing syntax details of the programming language employed to build the framework and (ii) being aware of the architectural details of the CF and its internal nomenclature. Also, another problem is that the reuse process can only be initiated as soon as the development process reaches the implementation phase, preventing it from starting earlier.

Method: In order to solve these problems, we present in this paper a model-based approach for reusing CFs which shields application engineers from technical details, letting him/her concentrate on what the framework really needs from the application under development. To support our approach, two models are proposed: the Reuse Requirements Model (RRM) and the Reuse Model (RM). The former must be used to describe the framework structure and the later is in charge of supporting the reuse process. As soon as the application engineer has filled in the RM, the reuse code can be automatically generated.

Results: We also present here the result of two comparative experiments using two versions of a Persistence CF: the original one, whose reuse process is based on writing code, and the new one, which is model-based. The first experiment evaluated the productivity during the reuse process, and the second one evaluated the effort of maintaining applications developed with both CF versions. The results show the improvement of 97% in the productivity; however little difference was perceived regarding the effort for maintaining the required application.

Conclusion: By using the approach herein presented, it was possible to conclude the following: (i) it is possible to automate the instantiation of CFs, and (ii) the productivity of developers are improved as long as they use a model-based instantiation approach.

1 Content

This article is organized as follows: In Section 2 is presented the introduction of this article. Section 3 presents the necessary background to understand this article. More specifically, it is split into three sections, they are: Section 3.1 presents the concepts of Model-Driven Development, Section 3.2 showns the general notion of Aspect oriented programming and in Section 3.3 is presented the concepts of Crosscutting frameworks. In Section 4 is presented the proposed approach. In Section 5 is presented the evaluation of our approach. In Section 7 is presented some related works. Finally, in Section 8 we present the conclusion of this article.

2 Introduction

Aspect-Oriented Programming (AOP) is a programming paradigm that overcomes the limitations of Object- Orientation (Programming) providing more suitable abstractions for modularizing crosscutting concerns (CC) such as persistence, security, and distribution. AspectJ is one of the programming languages that implements these abstractions (AspectJ Team 2003). Since the advent of AOP in 1997, a substantial effort has been invested in discovering how such abstractions can enhance reuse methodologies such as frameworks (Fayad and Schmidt 1997) and product lines (Clements and Northrop 2002). One example is the research that aims to design a CC in a generic way so that it can be reused in other applications (Bynens et al. 2010; Camargo and Masiero 2005; Cunha et al. 2006; Huang et al. 2004; Kulesza et al. 2006; Mortensen and Ghosh 2006; Sakenou et al. 2006; Shah and Hill 2004; Soares et al. 2006; Soudarajan and Khatchadourian 2009; Zanon et al. 2010). Because of the absence of a representative taxonomy for this kind of design, in our previous work we have proposed the term "Crosscutting Framework" (CF) to represent a generic and abstract design and implementation of a single crosscutting concern (Camargo and Masiero 2005).

Most of the CFs which are found in the literature adopt white-box reuse strategies in their reuse process, relying on writing source code to reuse the framework (Bynens et al. 2010; Camargo and Masiero 2005; Cunha et al. 2006; Huang et al. 2004; Kulesza et al. 2006; Mortensen and Ghosh 2006; Sakenou et al. 2006; Shah and Hill 2004; Soares et al. 2006; Soudarajan and Khatchadourian 2009; Zanon et al. 2010). This strategy is flexible in terms of framework evolution; however, application engineers need to cope with details not directly related to the requirements of the application under development. Therefore, the following problems exist when using such strategies: (i) the learning curve is steep because application engineers need to learn the programming paradigm employed in the framework design; (ii) a number of errors can be inserted because of the manual creation of the source code.; (iii) the development productivity is negatively affected as several lines of code must be written to define a small number of hooks, and (iv) the reuse processes can only be initiated during the implementation phase as there is no source code available in earlier phases.

To overcome these problems, we present a new approach for supporting the reuse of CFs using a Model-Driven Development (MDD) strategy. MDD consists of a combination of generative programming, domain-specific languages and model transformations. MDD aims at reducing the semantic gap between the program domain and its implementation, using high-level models that screen software developers from complexities of the underlying implementation platform (France and Rumpe 2007). Our approach is based

on two models: the Reuse Requirements Model (RRM) and the Reuse Model (RM). Built by a framework engineer, RRM documents all the features and variabilities of a CF. Application engineers can then select just the desired features from the RRM and generate a more specific model, referred to as the RM. Later, the application engineer can conduct the reuse process by completing the RM fields with information from the application and automatically generate the reuse code.

Furthermore, we present the results of two comparative experiments which used the same Persistence CF (Camargo and Masiero 2005). The first experiment aimed to compare the productivity of conducting a reuse process when using our model-based approach versus the ad-hoc approach, i.e., writing the source code manually. The purpose of the second experiment was to compare the effort of maintaining applications developed with both our model-based approach versus the ad-hoc way. Our approach presented clear benefits for the instantiation time (productivity); however, no differences were identified regarding the maintenance effort. Therefore, the main contribution of this paper is twofold: (*i*) introduction of a model-based approach for supporting application engineers during the reuse process of CFs and (*ii*) presentation of the results of two experiments.

3 Background

This section describes the background necessary to understand our proposed models. It is split into three subsections: the first one contains the concepts of Model-Driven Development, the second subsection has a basic description of aspect-oriented programming and the third one exposes the general notion of Crosscutting Frameworks.

3.1 Model-driven development

Software systems are becoming increasingly complex as customers demand richer functionality be delivered in shorter timescales (Clark et al. 2004). In this context, Model-Driven Development (MDD) can be used to speed up the software development and to manage its complexity in a better way by shifting the focus from the programming level to the solution-space.

MDD is an approach for software development that puts a particular emphasis upon making models the primary development artifacts and upon subjecting such models to a refinement process by using automatic transformations until a running system is obtained. Therefore, MDD aims to provide a higher abstraction level in the system development which further results in the improved understanding of complex systems (Pastor and Molina 2007).

Furthermore, MDD can be employed to handle software development problems that originate from the existence of heterogeneous platforms. This can be achieved by keeping different levels of model abstractions and by transforming models from Platform Independent Models (PIMs) to Platform Specific Models (PSMs) (Pastor and Molina 2007). Therefore, the automatic generation of application specific code offers many advantages such as: a rapid development of high quality code; a reduced number of accidental programming errors and the enhanced consistency between the design and the code (Schmidt 2006).

It is worth highlighting that models in MDD are usually represented by a domain-specific language (Fowler 2010), i.e., a language that adequately represents the information of a given domain. Instead of representing elements using a general purpose language

(GPL), the knowledge is described in the language which domain experts understand. Besides, as the experts use a suitable language to describe the system at hand, the accidental complexity that one would insert into the system to describe a given domain is reduced, leaving just the essential complexity of the problem.

3.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) aims at improving the modularization of a system by providing language abstractions that are dedicated to modularize crosscutting concerns (CCs). CCs are concerns which cannot be accurately modularized by using conventional paradigms (Kiczales et al. 1997). Without proper language abstractions, crosscutting concerns become scattered and tangled with other concerns of the software, affecting maintainability and reusability. In AOP, there is usually a distinction between base concerns and crosscutting concerns. The base concerns (or Core-concerns) are those which the system was originally designed to deal with. The crosscutting concerns are the concerns which affect on other concerns. Examples of crosscutting concerns include global restrictions, data persistence, authentication, access control, concurrency and cryptography (Kiczales et al. 1997).

Aspect-Oriented Programming languages allow programmers to design and implement crosscutting concern decoupled from the base concerns. The AOP compiler has the ability to weave the decoupled concerns together in order to attain a correct software system. Therefore, on the source-code level, there is a complete separation of concerns and the final release delivers the functionality expected by the users.

In this work we have employed the AspectJ language (Kiczales et al. 2001), which is an aspect-oriented extension for Java, allowing the Java code to be compiled seamlessly by the AspectJ compiler. The main constructs in this language are: aspect - a structure to represent a crosscutting concern; pointcut - a rule used to capture join points of other concerns; advices - types of behavior to be executed when a join point is captured; and intertype declarations - the ability to add static declarations from the outside of the affected code. In our work, intertype declarations are used to insert more interface realizations into classes of the base concern.

3.3 Crosscutting frameworks

Crosscutting Frameworks (CF) are aspect-oriented frameworks which encapsulate the generic behavior of a single crosscutting concern (Camargo and Masiero 2005; Cunha et al. 2006; Sakenou et al. 2006; Soudarajan and Khatchadourian 2009). It is possible to find CFs to support the implementation of persistence (Camargo and Masiero 2005; Soares et al. 2006), security (Shah and Hill 2004), cryptography (Huang et al. 2004), distribution (Soares et al. 2006) and other concerns (Mortensen and Ghosh 2006). The main objective of CFs is to make the reuse of crosscutting concerns a reality and a more productive task during the development of an application.

As well as other types of frameworks, CFs also need specific pieces of information regarding the base application to be reused correctly and to work properly. We name this kind of information “Reuse Requirements” (RR). For instance, the RR for an Access Control CF includes: 1) the application methods that need to have their access controlled; 2) the roles played by users; 3) the number of times a user is allowed get an incorrect password. This information is commonly documented in manuals known as “Cookbooks”.

Unlike application frameworks, which are used to generate a whole new application, a CF needs to be coupled to a base application to become operational. The conventional process to reuse a CF is composed by two activities: instantiation and composition. During the instantiation, an application engineer chooses variabilities and implements hooks, while during the composition, he/she provides composition rules to couple the chosen variabilities to a base code.

CF-based applications, i.e, applications which were developed with the support of CFs, are composed by three types of modules: a base code module, a reuse code module and framework itself. The "base code" represents the source code of the base application and the "framework code" is the CF source code, which is untouched during the reuse process. The "reuse module" is the connection between the base application and the framework and it is developed/written by the application engineer. Applications can be composed by several CFs, each one coupled by one reuse module. The source code created specifically to reuse a CF, is referred here as "reuse code".

In our previous work we have developed a Persistence CF (Camargo et al. 2004) which is used here as a case study. This CF was designed like a product-line, so it has certain mandatory features, for instance, "Persistence" and "Connection". The first one aims to introduce a set of persistence operations (e.g., store, remove, update, etc) into application persistence classes. The second feature is related to the database connection and identifies points in the application code where a connection needs to be established or closed. This feature has variabilities as the Database Management System (e.g., MySQL, SyBase, Native and Interbase). This CF also has a set of optional features such as "Caching", which is used to improve the performance by keeping copies of data in the local memory, and "Pooling", which represents a number of active database connections.

4 Model-based reuse approach

In this section we present our approach and the models that support during the instantiation and composition of CFs: Reuse Requirements Model (RRM) and Reuse Model (RM). These models have been formulated on top of Eclipse Modeling Framework and Graphical Modeling Framework (Eclipse Consortium 2011). The formal definition of both models is specified by the metamodel shown in Figure 1. It is comprised of a set of enumerations, abstract and concrete metaclasses.

The metamodel was built based on the vocabulary commonly used in the context of CFs, for example: pointcuts, classifier extensions, method overriding, and variability selection. These concepts were mapped into concrete metaclasses, which are visible under the dashed line of Figure 1.

Above the dashed line, there are also the following enumerations: "Visibility", "SuperType" and "CompositionType", which are sets of literals used as metaclass properties. The other elements above the line are abstract metaclasses, which were created after generalizing the properties of the concrete metaclasses. These abstract metaclasses can be applied in similar approaches and are also important to improve modularity and to avoid code replication of the reuse code generator.

Both of our proposed models are identical, however they are employed in different moments of the process. The first proposed model, the RRM, is a graphical documentation for Reuse Requirements, i.e., it graphically documents all the information needed to couple a CF to a base application. Conventionally, this is known as "cookbooks". This

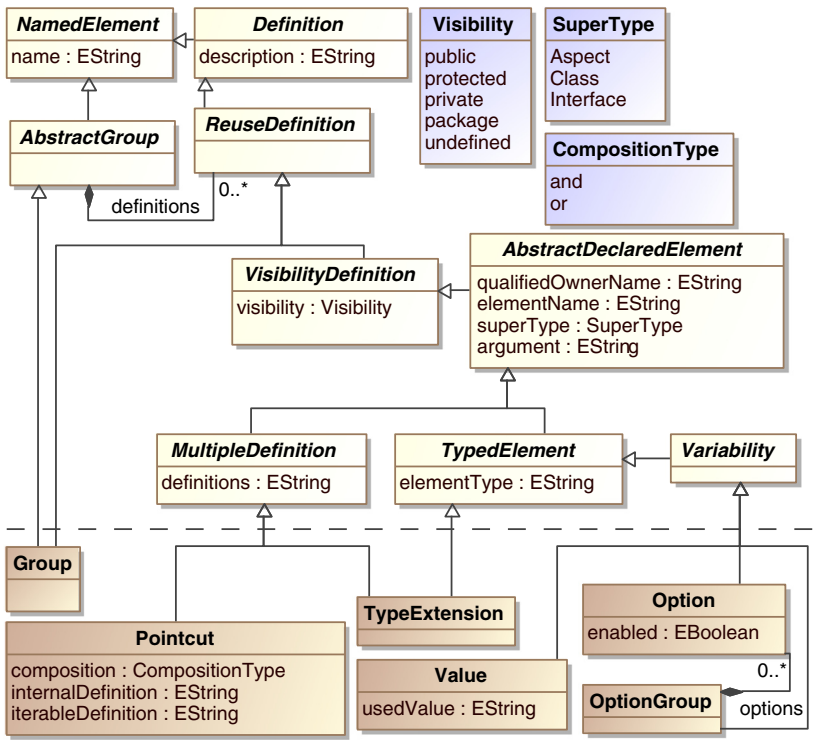


Figure 1 Metamodel of the proposed models. A metaclass diagram of 'CFReuse', which contains all metaclasses and enumerations.

model involves information regarding all CF features and must be developed/provided by a framework engineer. The second model, the RM, is a subset of the RRM and contains only the selected features for conducting a reuse process. Since both models share the same metamodel, it is possible to employ a direct model transformation to instantiate a RM from a RRM by selecting a valid set of features. Both of our models are represented as forms containing boxes, as seen in Figure 2. Each box is an instance of a concrete metaclass element and represents a reuse requirement. Each box contains three lines. The first one contains both an icon representing the type of the element, (which is the same type visible in the "Palette") and the name of the reuse requirement. The second line shows a description and the last line must be filled by the application engineer to provide the necessary information regarding the base application. Notice that the last line is used only in RMs.

By analyzing a RRM, the application engineer can identify all the information required by the framework to conduct the reuse process. For example, this model represents the variabilities that must be chosen by the application engineer and also indicates join-points of the base code where crosscutting behavior must be applied to, as well as classes, interfaces, or aspect names that must be affected.

Framework variabilities that must be chosen during reuse process are also visible. For example, to instantiate a persistence CF, several activities must be done, among them: i) informing points of the base application in which the connection must be open and closed; ii) informing methods that represent data base transactions and iii) choosing variabilities, e.g., the driver that should be used to connect to the database.

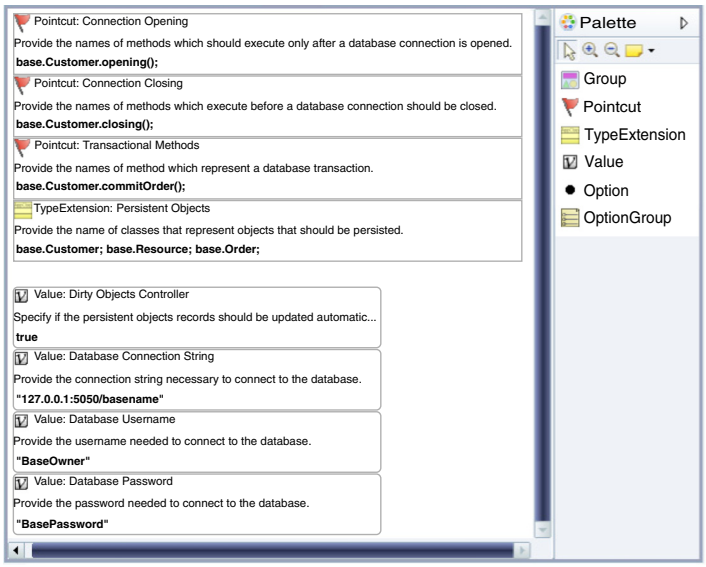


Figure 2 Reuse requirements models and a reuse models editor. A graphical user interface of the model editor. The editor is capable of editing Reuse Requirement Models and Reuse Models. The pallete on the right of the figure lists every concrete metaclass, which can be modelled on model level by dragging and dropping.

The another model, the RM, is shown in Figure 2. It supports the reuse process of a crosscutting framework by filling in the third line of the boxes. Therefore, RM must be used by the application engineer to reuse a framework. For instance, the value "*base.Customer.opening()*" is a method of the base application that was inserted by the application engineer into the third line of the "Connection Opening" box to inform that the DB connection must be established before this method runs.

The code generator transforms the Reuse Model into the Reuse Code, which consists of pieces of AspectJ code used to couple the base application to the crosscutting framework. This transformation is not a one to one conversion, i.e., every element in the model not always generates the same number of code elements. This was a special underlying challenge we have experienced when implementing this approach. The code generator needs to read the RM completely and to aggregate all data to identify how many files need to be generated.

The reuse model elements contain attributes to define the super classes to be extended; several elements may identify the same superclass. Therefore, the code generator must identify every superclass in order to create a single subclass per superclass when generating "Pointcuts", "Options" and "Value Definitions".

The generation of "Type Extensions" is slightly different. Whenever there is a single type extension, the code generator creates a single aspect that aggregates every type extension using "declare parents"; a specific type of intertype declaration.

The architecture of the generator is represented in Figure 3. Initially, the XTend (Efftinge 2006) library is used as a front end of the compiler, loading the data of the model into a hierarchical structure in memory, similar to a Domain Object Model. After the structure is loaded, it is processed in order to identify the units that must be generated. This process creates another structure that represents the resulting code, which is similar to an abstract syntax tree. The "AJGenerator" is a back end of the generator that we have also created; it is capable of transforming this tree into actual files of valid AspectJ code.

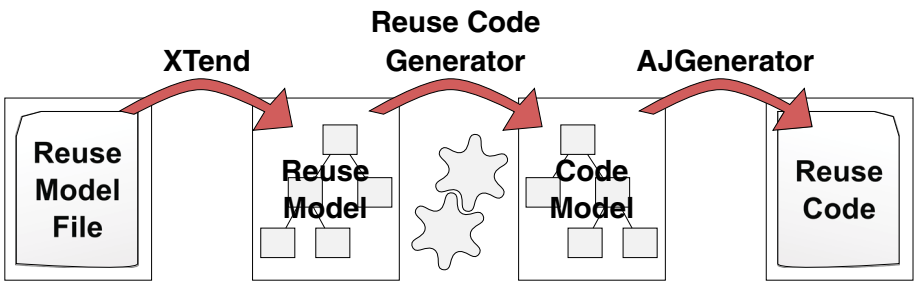


Figure 3 Reuse code generator architecture. The reuse code generator creates a reuse code by processing a reuse model. This process is divided into three sub processes, which are shown in this figure.

4.1 Reuse process

This subsection explains the reuse process that is defined when using the new proposed models (RRM and RM). From this point it is important to clarify the distinction between the terms *model* and *diagram*. Model is a more generic term and it is physically represented by XML files, while a diagram is a visual representation of a model. So, in our case, the Reuse Requirements Diagram (RRD) is a diagram that represents the Reuse Requirements Model and the Reuse Diagram (RD) is a diagram that represents the Reuse Model. It is also worth mentioning that these diagrams are similar to forms, in which they must be filled in. In order to explain the new process, there is an activity diagram in Figure 4 illustrating the perspective of both developers: framework engineers and application engineers.

Since the CF must be completely defined before its reuse process is started, this explanation begins from the framework engineer's point of view. At the right side of the Figure 4, the framework engineer starts developing a new CF for a specific crosscutting concern. The first activity is to develop the framework itself (marked with 'A'). Then, the engineer should make the CF code available for reuse ('B') and should create the RRD ('C'), graphically indicating the information required to couple his CF to a base application. This diagram ('D') will be available for the application engineer. Upon finishing this process, the framework engineer has two artifacts that will be used by the application engineer: the Reuse Requirements Diagram ('D') and the Framework code ('B').

The reuse process starts on the left side of the figure, where the perspective of application engineers is considered. This engineer is responsible for developing the application, which is composed by both the "Base" and "Reuse" modules. By analyzing the application being developed ('a'), the application engineer must identify the concerns that would affect the software, possibly by using an analysis diagram ('b'). By having these concerns identified, the application engineer is able to select the necessary frameworks and to start the reuse process since the earlier development phases. After selecting and analyzing the RRD of the selected frameworks ('c'), it is necessary to select a subset of the optional variabilities ('d') because some elements may not be necessary (since the framework may be supplied with default values), or to select mutually exclusive features. The selected elements will be carried to a new "Reuse" diagram ('e'). If there are more than one CF being reused, then there should be a "Reuse" diagram for each one of them. The application engineer should then design the base application ('g') documenting the name of the units,

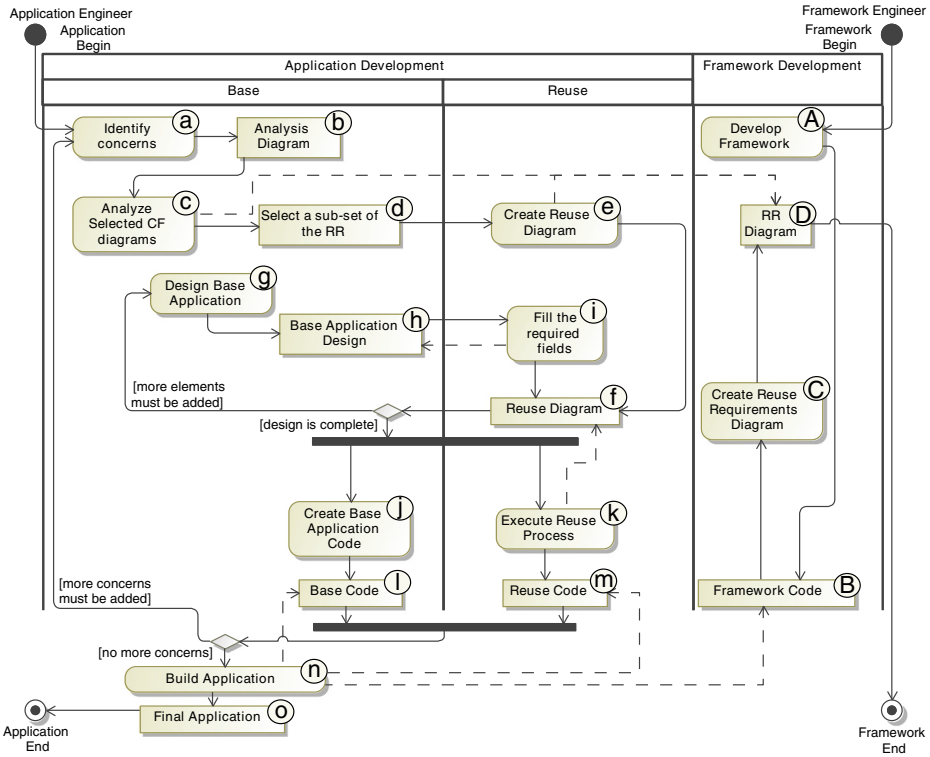


Figure 4 Reuse process activity diagram. A UML activity diagram that represents the reuse process to be employed when using the proposed models and the tools.

methods and attributes found on the base application ('h'). By designing the names of elements needed by the framework, they will become available, meaning that it is already possible to enter these names in the RD. This should be done before all required elements of the iteration are designed. After defining these names, which are the values needed by the reuse portion, they must be filled ('i') in the reuse diagram ('f') to enable the coupling among the modules.

The base application can be developed ('j') in parallel with the reuse process execution ('k'), which is a model transformation to generate the "Reuse Code" ('m') from the "Reuse Diagram" ('f'). After completing the "Base Code" ('l') and the "Reuse Code" ('m'), the application engineer may choose between adding a new concern (and extending the base application) or finishing the process. At that moment, the following pieces of code are available: the "Base Code" ('l'), the "Reuse Codes" ('m') and the selected "Framework Codes" ('B'). All of these codes are processed to build ('n') the "Final Application" ('o') and to conclude the process.

The transformation employed to create the RD avoids manual creation of this model. This is possible by identifying the selected framework and by processing its RRD. Besides accelerating the creation of this model, this also allows the RD to take all the needed elements from the earlier diagram to the code generation. However, the values regarding the base application are still needed and must be informed by the application engineer. The RRD contains information needed by the framework being reused. By identifying that information during earlier development phases it is easier to define it correctly. Consequently, the base application is not oblivious of the framework and its behaviors, however,

the modules are completely isolated and have no code dependency among them. It is important to point out that the Reuse Code itself depends on the Base Code during the creation process, however, its definition can be made as soon as the base application design is complete.

4.2 Approach usage example

An usage example of our approach is described in this section. Firstly, we briefly describe the domain engineering which contains the creation of the framework reuse model. Finally, the application engineering is described, which consists of reuse model completion and reuse code generation, thus completing the process.

4.3 Domain engineering

The domain engineer must create a reuse model which contains the information necessary to reuse a crosscutting framework. In the example provided herein, every information needed to create a reuse model for a persistence framework. After the model creation, its completion is shown during application engineering to reuse the framework and couple it to an example application.

The reuse model template for the crosscutting framework in Figure 5, which was derived from a reuse requirements model by describing the framework hotspots. In Figure 6, the reuse model is shown after its completion.

The model elements are defined as follows: there are four value objects, two pointcut objects, and one type extension object. The value objects are used to define strings needed

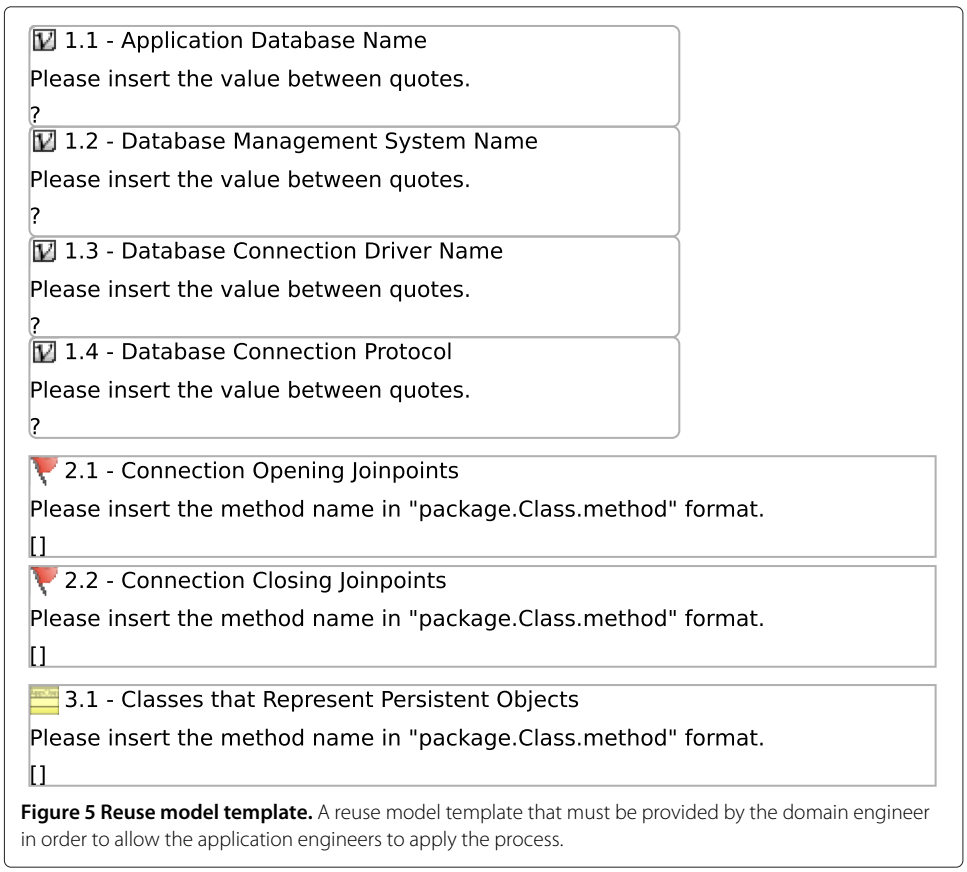


Figure 5 Reuse model template. A reuse model template that must be provided by the domain engineer in order to allow the application engineers to apply the process.

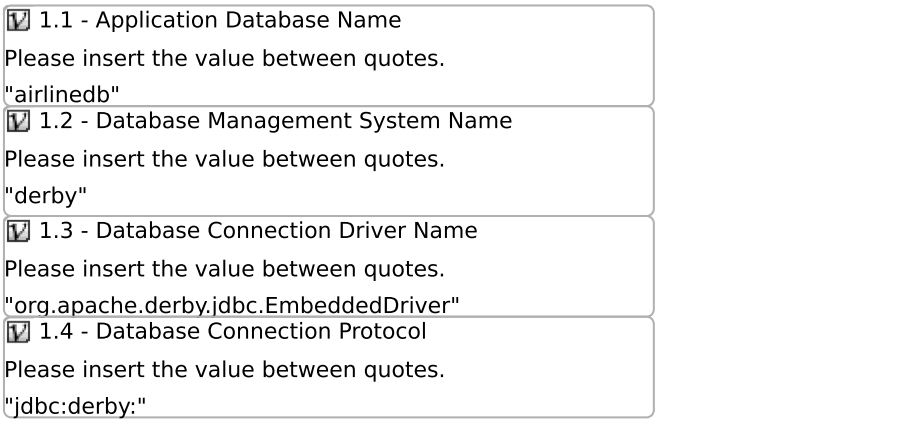


Figure 6 Complete reuse model. A complete reuse model after the application engineer provides the information regarding the base application, which will allow framework reuse by code generation.

by the framework in order to connect it to the database. They are used to define the database name, the name of the database management system, the database connection driver, and the database connection protocol. Every property of these items are then represented on Tables 1, 2, 3 and 4.

The pointcut objects are used to define joinpoints of the base application. The first pointcut object is represented on Table 5 and it must be used to inform where DB connections must be established. To do that, the application engineer needs to inform which methods execute right after a DB connection is established, i.e., methods that operate properly only if there is a connection open. The second point cut object is represented on Table 6 and it is used to inform methods that execute right before the connection is closed, therefore, the last method that needs an open connection.

The last object is represented on Table 7, which is used to define the classes found in the base application. These base application define object types that must be persisted on the database.

This reuse model is provided along with the crosscutting framework to be used by the application engineer in order to instantiate the framework, which is described in Section 4.3.

4.4 Application engineering

An example of an application development is given in this subsection. This application is referred to as Airline Ticket Management and must be coupled to the persistence CF

Table 1 Application database name

Value object: application database name	
Name	***Name***
"1.1 - Application database name"	
Description	***Description***
"Please insert the value between quotes."	
Qualified name of the owner	***QualifiedOwnerName***
"persistence.instantiation.helper.ExtendedConnectionVariabilities"	
Method to be overridden	***ElementName***
"ExtendedConnectionVariabilities.setDatabaseName"	
Value data type	***ElementType***
"String"	
Select supertype (aspect, class or interface)	***SuperType***
"Class"	

A string needed by the database connection API in order to connect to a specific database managed by the database system.

previously mentioned. This application uses the Apache Derby Database Management System (Apache Software Foundation 2012). The design of this application is shown on Table 8.

Upon the reuse model completion, the resulting reuse model is similar to that shown in Figure 7. Despite not being shown in the application details, every base application class was created inside the package "baseapp". After validating the model, the reuse code is generated; it is divided into three units.

The first generated unit is an aspect that extends a framework class. The overridden methods are used to return constant values that are necessary for the framework to successfully get connected to the database. It is important to emphasize that the four values have been defined in the same unit because they are owned by the same superclass. This would not happen if their superclasses were different.

The second unit is shown in Figure 8, which is an aspect that overrides pointcuts *openConnection* and *closeConnection*. These pointcuts are used to capture base

Table 2 Database management system name

Value object: database management system name	
Name	***Name***
"1.2 - Database management system name"	
Description	***Description***
"Please insert the value between quotes."	
Qualified name of the owner	***QualifiedOwnerName***
"persistence.instantiation.helper.ExtendedConnectionVariabilities"	
Method to be overridden	***ElementName***
"ExtendedConnectionVariabilities.setSpecificDatabase"	
Value data type	***ElementType***
"String"	
Select supertype (aspect, class or interface)	***SuperType***
"Class"	

A string needed by the database connection API in order to select the database connection driver.

Table 3 Connection driver protocol

Value object: database connection driver name	
Name	***Name***
"1.3 - Database connection driver name"	
Description	***Description***
"Please insert the value between quotes."	
Qualified name of the owner	***QualifiedOwnerName***
"persistence.instantiation.helper.ExtendedConnectionVariabilities"	
Method to be overridden	***ElementName***
"ExtendedConnectionVariabilities.getDriver"	
Value data type	***ElementType***
"String"	
Select supertype (aspect, class or interface)	***SuperType***
"Class"	

A string needed by the database connection API in order to select the database connection driver.

application joinpoints that trigger the database connections and disconnections. They are defined in a single aspect because they also share the same superclass.

Figure 9 shows another aspect, which uses static crosscutting features to define classes that extend the interface specified by domain engineer by using the "Declare Parents" syntax.

Our model generator is also capable of generating a validation code, which checks if the base element names inserted into the reuse model are valid.

5 Methods

Two experiments have been conducted to compare our model-based reuse approach with the conventional way of reusing CFs, i.e., manually creating the reuse code. The first experiment is called Reuse Study and was planned to identify the gains in productivity when reusing a framework. The second experiment is denominated "Maintenance Study" and was designed to identify whether the our models help or not in the maintenance of

Table 4 Connection driver protocol

Value object: database connection protocol	
Name	***Name***
"1.4 - Database connection protocol name"	
Description	***Description***
"Please insert the value between quotes."	
Qualified name of the owner	***QualifiedOwnerName***
"persistence.instantiation.helper.ExtendedConnectionVariabilities"	
Method to be overridden	***ElementName***
"ExtendedConnectionVariabilities.getJDBC"	
Value data type	***ElementType***
"String"	
Select supertype (aspect, class or interface)	***SuperType***
"Class"	

A string needed by the database connection API in order to connect to the database system.

Table 5 Connection opening joinpoint definition

Pointcut Object: connection opening joinpoints	
Name	***Name***
"2.1 - Connection opening joinpoints"	
Description	***Description***
"Please insert the method name in "package.Class.method" format."	
Qualified name of the owner	***QualifiedOwnerName***
"persistence.instantiation.helper.ExtendedConnectionCompositionRules"	
Pointcut to be overridden	***ElementName***
"openConnection"	
Composition operator	***Composition***
"or"	
Internal pointcut definition	***InternalDefinition***
""	
Iterable pointcut definition	***IterableDefinition***
"execution (* %s(..))"	
Select supertype (aspect, class or interface)	***SuperType***
"Class"	

The second pointcut object is used to define methods that run right after the database connection must be open.

a CF-based application. This second study is important because maintenance activities are usually performed more often than the reuse process. Each experiment has been performed twice. In this paper, the first execution is referred to as "First" and the second execution is referred to as "Replication". Since there have been only two executions for each experiment, we present four study executions in this section. The structure of the studies has been defined according to the recommendations of Wohlin et al. (2000).

Table 6 Connection closing joinpoint definition

Pointcut object: connection closing joinpoints	
Name	***Name***
"2.2 - Connection closing joinpoints"	
Description	***Description***
"Please insert the method name in "package.Class.method" format."	
Qualified name of the owner	***QualifiedOwnerName***
"persistence.instantiation.helper.ExtendedConnectionCompositionRules"	
Pointcut to be overridden	***ElementName***
"closeConnection"	
Composition operator	***Composition***
"or"	
Internal pointcut definition	***InternalDefinition***
""	
Iterable pointcut definition	***IterableDefinition***
"execution (* %s(..))"	
Select supertype (aspect, class or interface)	***SuperType***
"Class"	

The second pointcut object is used to define methods that run right before the database connection must be closed.

Table 7 Persistent objects definition

Type extension object: persistent objects	
Name	***Name***
"3.1 - Classes that represent persistent objects"	
Description	***Description***
"Please insert the class name in "package.Class" format."	
Qualified name of the owner	***QualifiedOwnerName***
"persistence.PersistentRoot"	
Select supertype (aspect, class or interface)	***SuperType***
"Interface"	

The last object from the example that is used to define the persistent objects of the application.

5.1 Reuse study definition

The objective was to compare the effort of reusing frameworks by using a conventional technique with the effort of using a model-based technique. The Persistence CF, briefly presented in Subsection 3.3, has played the role of "study subject" and it was used in both reuse techniques (conventional and model-based). The quantitative focus was determined

Table 8 Base application details

Constant definition		
Application database name	"airlinedb"	
Database management system name	"derby"	
Database connection driver name	"org.apache.derby.jdbc.EmbeddedDriver"	
Database connection protocol	"jdbc:derby:"	
Joinpoint definition (method execution)		
Joinpoints	**Method execution**	
Connection opening joinpoints	**Class**	**Method**
	Airplane	Landing
	Airport	Opening
	Luggage	Dispatch
	Checkin	Confirm
	Passenger	Onboard
	Flight	Depart
Connection closing joinpoints	**Class**	**Method**
	Airplane	Takeoff
	Airport	Closing
	Luggage	Retrieve
	Checkin	Cancel
	Passenger	Unboard
	Flight	Arrive
Type extensions		
Classes that represent persistent objects	**Class**	
	Airport	
	Luggage	
	Checkin	
	Passenger	
	Flight	

Details of a base application that needs to be coupled to the crosscuting framework.

ConcreteExtendedConnectionVariabilities0.aj

```
package persistence.reuse;
import persistence.instantiation.helper.ExtendedConnectionVariabilities;
public aspect ConcreteExtendedConnectionVariabilities0 extends
 ExtendedConnectionVariabilities {
    public String ExtendedConnectionVariabilities.setDatabaseName (){
        return "airlinedb";
    }
    public String ExtendedConnectionVariabilities.setSpecificDatabase (){
        return "derby";
    }
    public String ExtendedConnectionVariabilities.getDriver (){
        return "org.apache.derby.jdbc.EmbeddedDriver";
    }
    public String ExtendedConnectionVariabilities.getJDBC (){
        return "jdbc:derby:";
    }
}
```

Figure 7 Reuse code - first unit. The first generated reuse code fragment contains an aspect that is used to define the "Connection Variabilities" of the framework: these "Connection Variabilities" are provided by overriding methods.

considering the time spent in conducting the reuse process. The qualitative focus was to determine which technique takes less effort during the reuse process. This experiment was conducted from the perspective of application engineers reusing CFs: the study object was the 'effort' to perform a CF reuse.

5.2 Reuse study planning

The first experiment was planned considering the following question: "Which reuse technique takes less effort to reuse a CF?";

ConcreteExtendedConnectionCompositionRules1.aj

```
package persistence.reuse;
import persistence.instantiation.helper.ExtendedConnectionCompositionRules;
public aspect ConcreteExtendedConnectionCompositionRules1 extends
    ExtendedConnectionCompositionRules {
    public pointcut openConnection ():
        (
                execution (* baseapp.Airplane.landing(..)) ||
                execution (* baseapp.Airport.opening(..)) ||
                execution (* baseapp.Luggage.dispatch(..)) ||
                execution (* baseapp.Checkin.confirm(..)) ||
                execution (* baseapp.Passenger.board(..)) ||
                execution (* baseapp.Flight.depart(..))
        );
    public pointcut closeConnection ():
        (
                execution (* baseapp.Airplane.takeoff(..)) ||
                execution (* baseapp.Airport.closing(..)) ||
                execution (* baseapp.Luggage.retrieve(..)) ||
                execution (* baseapp.Checkin.cancel(..)) ||
                execution (* baseapp.Passenger.unboard(..)) ||
                execution (* baseapp.Flight.arrive(..))
        );
}
```

Figure 8 Reuse code - second unit. The second generated reuse code fragment contains an aspect that is used to define the "Composition Rules" of the framework. The "CompositionRules" are provided by overriding pointcuts.

ParentsDeclaration.aj

```
package persistence.reuse;
public aspect ParentsDeclaration extends Object {
    declare parents : baseapp.Airplane implements
        persistence.PersistentRoot;
    declare parents : baseapp.Airport implements
        persistence.PersistentRoot;
    declare parents : baseapp.Luggage implements
        persistence.PersistentRoot;
    declare parents : baseapp.Checkin implements
        persistence.PersistentRoot;
    declare parents : baseapp.Passenger implements
        persistence.PersistentRoot;
    declare parents : baseapp.Flight implements
        persistence.PersistentRoot;
}
```

Figure 9 Reuse code - third unit. The third generated reuse code fragment contains an aspect that is used to define static crosscutting features needed during the framework reuse. By default, all static crosscutting declarations are merged into a single aspect.

5.2.1 Context selection

Both studies have been conducted by students of Computer Science. In this section, they are referred to as "participants". Sixteen participants took part in the experiments, eight of those were undergraduate students and the other eight were post-graduate students. Every participant had a prior AspectJ experience.

5.2.2 Formulation of hypotheses

Table 9 contains our formulated hypotheses for the reuse study, which are used to compare the productivity of our tool with the conventional process.

There are two variables shown on the table: "Tc_r" and "Tm_r". "Tc_r" represents the overall time necessary to reuse the framework using the conventional technique while "Tm_r" represents the overall time necessary to reuse the framework using the model-based technique. There are three hypotheses shown on the table: "$H0_r$", "Hp_r" and "Hn_r". "$H0_r$" represents the null hypothesis, which is true when both techniques are equivalent; then, the time spent using the conventional technique minus the time spent

Table 9 Hypotheses for the reuse study

H0$_r$	There is no difference between using our tool and using an ad-hoc reuse process in terms of productivity (time) to successfully couple a CF with an application. Then, the techniques are equivalent. $Tc_r - Tm_r \approx 0$
Hp$_r$	There is a positive difference between using our tool and using an ad-hoc reuse process in terms of productivity (time) to successfully couple a CF with an application. Then, the conventional technique takes more time than the model-based tool. $Tc_r - Tm_r > 0$
Hn$_r$	There is a negative difference between using our tool and using an ad-hoc reuse process in terms of productivity (time) to successfully couple a CF with an application. Then, the conventional technique takes less time than the model-based tool. $Tc_r - Tm_r < 0$

Considering the Reuse Study, there are three hypotheses for the outcome. In the first one, both are equivalent, while in two of them, a technique is faster.

using the model-based tool is approximately zero. "Hp_r" represents the first alternate hypothesis, which is true when the conventional technique takes longer than the model-based tool; then, the time spent to use the conventional technique minus the time of the model-based tool is positive. "Hn_r" represents the second alternate hypothesis, which is true when the conventional technique takes longer than the model-based tool; then, the time taken to use the conventional technique minus the time taken to use the model-based tool is negative. As these hypotheses consider different ranges of a single resulting real value, then, they are mutually exclusive and only one of them is true.

5.2.3 Variable selection

The dependent variable in this work is the "time spent to complete the process". The independent variables are Base Application, Technique and Execution Types, which, are controlled and manipulated.

5.2.4 Participant selection criteria

The participants were selected through a non-probabilistic approach by convenience, i. e., the probability of all population elements belong to the same sample is unknown. We have invited every student from the computing department of Federal University of São Carlos that attended the AOP course, a total of 17 students. Every student had to be able to reuse the framework by editing code during the training. Because of that, one undergraduate student was rejected before the execution.

5.2.5 Design of the study

The participants were divided into two groups. Each group was composed by four graduate students and four undergraduate students. Each group was also balanced considering a characterization form and their results from the pilot study. Table 10 shows the planned phases.

Table 10 Study design

Phase	Group 1	Group 2
General training	Reuse and maintenance training	
	Repair shop	
1^{st} Reuse	Conventional	Models
Pilot phase	Hotel application	
2^{nd} Reuse	Models	Conventional
Pilot phase	Library application	
1^{st} First	Conventional	Models
Reuse phase	Deliveries application	
2^{nd} First	Models	Conventional
Reuse phase	Flights application	
1^{st} Replication	Conventional	Models
Reuse phase	Medical clinic application	
2^{nd} Replication	Models	Conventional
Reuse phase	Restaurant application	
1^{st} First	Conventional	Models

The Study Design contains every phase from both studies. It contains the sequence of operations, technique and the conidered applications.

5.2.6 Instrumentation for the reuse study

Base applications were provided together with two documents. The first document was a manual regarding the current reuse technique, and the second document was a list of details, which described the classes, methods and values regarding the application to be coupled.

The provided applications had the same reuse complexity. The participants had to specify four values, twelve methods and six classes in order to reuse the framework and to couple it to each application. These applications were designed with exactly the same structure of six classes. Each class contained six methods plus a class with a main method which is used to run the test case.

Each phase row of the Table 10 is divided into sub-rows that contain the name of the application and the technique employed to reuse the framework. For instance, during the First Reuse Phase, the participants of the first group coupled the framework to the "Deliveries Application" by using the conventional technique. The participants of the second used the model-based tool to perform the same exercise.

5.3 Operation for reuse study

5.3.1 Preparation

During the maintenance study, the students had to fix a reuse artifact to complete the process. Every participant had to fix every application by using only one of the techniques in equal numbers.

5.3.2 Execution

The participants had to work with two applications; each group started with a different technique. The secondary executions were replications of the primary executions with two other applications They were created to avoid the risk of getting unbalanced results during the primary execution, since some data that we gathered during the pilot study were rendered invalid.

5.3.3 Data validation

The forms filled by the participants were confirmed with the preliminary data gathered during the pilot study. In order to provide a better controllability, the researchers also watched the notifications from the data collector to check if the participants had concluded the maintenance process and had gathered the necessary data.

5.3.4 Data collection

The recorded timings during the reuse processes with both techniques are listed on the Table 11. Each table has five columns. Each column is defined by a letter or a word: "G" stands for the group of the participant during the activity; "A" stands for the application being reused; "T" stands for the reuse technique which is either "C" for conventional or "M" for model-based tool; "P" column lists an identifying code of the participants (students), whereas, the least, eight values are allocated to graduate students and the rest are undergraduate students; "Time" column lists the time the participant spent to complete each phase. The raw data we have gathered during the reuse study is also available as Additional file 1.

Table 11 Reuse process timings

Primary execution					Secondary execution				
G	**A**	**T**	**P**	**Time**	**G**	**A**	**T**	**P**	**Time**
1	Flights	M	15	04:19.952015	2	Clinic	M	10	02:59.467569
1	Flights	M	13	04:58.604963	1	Restaurant	M	13	03:56.785359
1	Flights	M	8	05:18.346829	1	Restaurant	M	15	04:23.629206
2	Delivery	M	11	05:24.249952	2	Clinic	M	11	04:25.196135
2	Delivery	M	5	05:31.653952	1	Restaurant	M	8	04:33.954349
2	Delivery	M	9	05:45.484577	2	Clinic	M	9	04:41.254920
2	Delivery	M	3	06:16.392424	1	Restaurant	M	12	05:05.524264
2	Delivery	M	10	06:45.968790	2	Clinic	M	3	05:45.333167
2	Delivery	M	14	07:05.858718	2	Clinic	M	14	05:57.009310
2	Delivery	M	6	07:39.300214	2	Clinic	M	5	06:31.365498
2	Delivery	M	2	08:02.570996	2	Clinic	M	2	06:59.967490
1	Flights	M	1	08:38.698360	2	Restaurant	C	2	07:18.927029
2	Flights	C	2	08:42.389884	2	Clinic	M	6	07:45.403075
1	Flights	M	16	10:18.809487	2	Restaurant	C	10	08:56.765163
1	Delivery	C	13	10:25.359836	1	Clinic	C	16	09:20.284593
2	Flights	C	9	10:51.761493	1	Restaurant	M	7	09:23.574403
1	Flights	M	7	10:52.183247	1	Restaurant	M	4	09:25.089084
2	Flights	C	10	10:52.495216	2	Restaurant	C	14	09:27.112225
1	Delivery	C	8	11:39.151434	2	Restaurant	C	3	09:55.736324
1	Delivery	C	15	12:03.519008	1	Clinic	C	15	10:25.475603
1	Flights	M	4	12:17.693128	2	Restaurant	C	5	10:37.460834
2	Flights	C	3	12:26.993837	2	Restaurant	C	9	10:49.014842
2	Flights	C	14	12:49.585392	1	Restaurant	M	16	10:56.743477
2	Flights	C	11	13:04.272941	1	Clinic	C	13	11:04.485390
1	Delivery	C	4	13:16.470523	1	Clinic	C	4	12:06.690347
1	Delivery	C	1	15:47.376327	1	Clinic	C	8	13:38.014602
1	Delivery	C	16	18:02.259692	1	Clinic	C	12	14:37.197260
1	Flights	M	12	20:03.920754	1	Restaurant	M	1	17:09.073104
2	Flights	C	5	21:32.272442	2	Restaurant	C	11	17:11.980052
2	Flights	C	6	23:10.727760	1	Clinic	C	7	19:35.816561
1	Delivery	C	7	23:20.991158	2	Restaurant	C	6	28:02.391335
1	Delivery	C	12	41:29.414342	1	Clinic	C	1	28:18.301114

The timings table contains data captured for each study, in this case, the maintenance study. By analysing this table, it is possible to identify the time of each participant, their group, the technique and the considered application.

We have developed a data collector to gather the experiment data. This system has stored the timings with milliseconds precision considering both the server and clients' system clocks. However, the values presented in this paper only consider the server time. The delays of transmission by the computers are not taken into consideration; preliminary calculations considering the clients' clocks have indicated that these delays are insignificant, i.e., have not changed the hypothesis testing results. The server's clock was considered because we could verify that its clock had not been changed throughtout the execution.

That system was able to gather the timings and the supplied information transparently. The participants only had to execute the start time, which was supervised, and to

work on the processes independently. After the test case had provided successful results, which meant that the framework was correctly coupled, the finish time was automatically submitted to the server before notifying the success to the participant.

5.4 Data analysis and interpretation for reuse study

The data of the first study is found on Table 11, which is arranged by the time taken to complete the process. The first noteworthy information found on this table is that the model-based reuse tool, which is identified by the letter 'M', is found in the first twelve results. The conventional process, which is identified by the letter 'C', got the last four results.

The timings data of Table 11 is also represented graphically in a bar graph, which is plotted on Figure 10. The same identifying code for each participant and the elapsed time in seconds are visible on the graph. The bars for the used conventional technique and the used model tool are paired for each participant, allowing easier visualization of the amount of time taken by each of them. In other words, the taller the bar, the more time it took to complete the process with the specified technique.

The second significant information found during the first study was that not a single participant could reuse the framework faster by using the conventional process than by using the reuse tool in the same activity.

Table 12 shows the average timings and their proportions. If we analyze the average time that the participants from both groups have taken to complete the processes, we could conclude that the conventional technique took approximately 97.64% longer than the model-based tool.

5.5 Maintenance study definition

It is necessary to remind here that our objective was to compare the effort in modifying a CF-based application by editing the reuse code (conventional technique) with the effort in modifying the same application by editing the RM. The Persistence CF, shown in the Section 3.3 was again used in the two maintenance exercises. The quantitative focus was measured by means of the time spent in the maintenance tasks and the qualitative focus was to determine which artifact (source code or RM) takes less effort during maintenance. This experiment was conducted from the perspective of application engineers

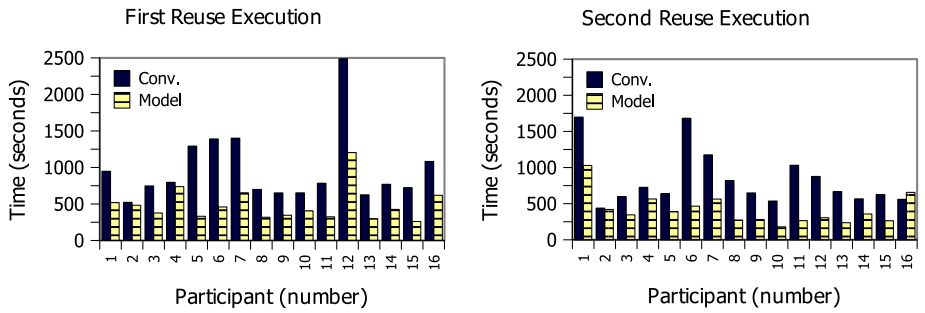


Figure 10 Reuse process timings bars graph. A graph in which the reuse experiment timings are plotted. Each bar represents the time taken by each participant in order to complete the reuse activities, which is identifiable by the participant number.

Table 12 Reuse study average timings

A.	Tech.	Avg.	Sum of Avg.	Percents
First	Conv.	16:13.44008	30:03.79341	66.7766%
Replication		13:50.35333		
First	Model	08:04.980525	14:57.441176	33.2234%
Replication		06:52.460651		
Total		45:01.234586		100.0000%

The Study Average Timings table contains averages for the Reuse Study. It is possible to compare the general time effort needed to complete the activities of both techniques.

who intended to maintain CF-based applications. Therefore, the study object is the 'effort' of maintaining a CF-based application.

5.6 Maintenance study planning

The core question we wanted to answer here was: "Which artifact takes less editing effort during maintenance: the reuse model or the reuse code?" During this experiment we have gathered and analyzed the timings taken to complete the process for each activity.

5.6.1 Context selection

Both studies were conducted by students of the Computer Science Department. In this section, they are referred to as "participants". Sixteen participants took part in the experiments: eight of them were undergraduate students and the other eight were graduate students. Every participant had a prior AspectJ experience.

5.6.2 Formulation of hypotheses

Table 13 contains three variables. "Tc_m" represents the overall time to edit the reuse code during maintenance. "Tm_m" represents the overall time to edit the reuse model during maintenance. "$H0_m$" represents the null hypothesis, which is true when the edition of both artifacts is equivalent. "Hp_m" represents the first alternate hypothesis, which is true when the edition of the reuse code takes longer than editing the RM. "Hn_m" represents the second alternate hypothesis, which is true when the edition of the reuse code takes less time than editing the RM. These hypotheses are also mutually exclusive: only one of them is true.

Table 13 Hypotheses for the maintenance study

H0$_m$	There is no difference between using editing a reuse model and editing the reuse code in terms of productivity (time) when maintaining an application that reuses a CF. Then, it is equivalent to edit any of the artifacts. $Tc_m - Tm_m \approx 0$
Hp$_m$	There is a positive difference between using editing a reuse model and editing the reuse code in terms of productivity (time) when maintaining an application that reuses a CF. Then, editing the reuse code takes more time than editing a reuse model during maintenance. $Tc_m - Tm_m > 0$
Hn$_m$	There is a negative difference between using editing a reuse model and editing the reuse code in terms of productivity (time) when maintaining an application that reuses a CF. Then, editing the reuse code takes less time than editing a reuse model during maintenance. $Tc_m - Tm_m < 0$

Considering the Reuse Study, there are three hypotheses for the outcome. In the first one, both are equivalent, while in two of them, a technique is faster.

5.6.3 Variable Selection

The dependent variable analyzed here was the "time spent to complete the process". The independent variables, which were controlled and manipulated, are: "Base Application", "Technique" and "Execution Types".

5.6.4 Participant selection criteria

The participants were selected through a non probabilistic approach by convenience, i. e., the probability of all population elements belong to the same sample is unknown. Both studies share the same participants.

5.6.5 Design of the Maintenance Study

The participants were divided into two groups. Each group was composed of four graduate students and four undergraduate students. Each group was also balanced considering the characterization form of each participant and their results from the first study. The phases for this study are shown in Table 14.

5.6.6 Instrumentation for the maintenance study

The base applications provided for the second study were modified versions of the same applications that had been supplied during the first study. These applicationswere provided with incorrect reuse codes (conventional) and incorrect reuse models (model-based): these incorrect artifacts had to be fixed by the participants. The participants received a document describing possible generic errors that could happen when a reuse code or a model are defined incorrectly. It is important to point out that that document did not have details regarding the base applications; the participants had to find the errors by browsing the source code.

The provided applications had the same reuse complexity: the reuse codes and models had the same amount of errors. In order to fix each CF coupling, the participants had to fix three outdated class names, three outdated method names, and three mistyped characters. It is also important to emphasize that errors specific for the manual edition of reuse codes were not inserted in this study.

Each phase row of the Table 14 is divided into sub-rows that contain the name of the application and the technique employed during the maintenance. For instance, the participants of the first group had to fix the reuse code of the "Deliveries Application" during

Table 14 Maintenance study design

Phase	Group 1	Group 2
General training	Maintenance training	
	Deliveries application	
2^{nd} First	Models	Conventional
Maintenance phase	Flights application	
1^{st} Replication	Conventional	Models
Maintenance phase	Medical clinic application	
2^{nd} Replication	Models	Conventional
Maintenance phase	Restaurant application	

The Study Design contains every phase from both studies. It contains the sequence of operations, technique and the conidered applications.

the First Maintenance Phase, while the participants of the second group had to fix the reuse model to perform the same exercise.

5.7 Operation for maintenance study

5.7.1 Preparation

During the maintenance study, the students had to fix a reuse artifact to complete the process. Every participant had to fix every application. They have fixed each application only once, by using only one of the techniques in equal numbers.

5.7.2 Operation Execution

The participants had to work with two applications; each group started with a different technique. The secondary executions were replications of the primary executions with two other applications They were created to avoid the risk of getting unbalanced results during the primary execution, since some data that we gathered during the pilot study were rendered invalid.

5.7.3 Data validation

The forms filled by the participants were confirmed with the preliminary data gathered during the pilot study. In order to provide a better controllability, the researchers also watched the notifications from the data collector to check if the participants had concluded the maintenance process and had gathered the necessary data.

5.7.4 Data collection

The timings for the maintenance study are presented in Table 15. The column "G" stands for the group of the participant; "A" stands for the application being reused; "T" stands for the reuse technique which is either "C" for conventional or "M" for model-based tool; "Time" column lists the time the participant spent to complete each phase, and finally; and "P" column lists an identifying code of the participants. At least eight values are allocated to graduate students and the rest are undergraduate students; The raw data we have gathered during the maintenance study is also available as Additional file 2.

The data collector that was employed to gather the experiment data stored the timings with milliseconds precision: both the server and clients' system clocks were taken into consideration. However, the values presented in this paper consider only the server time. The delay of data transmission over the network was not taken into consideration. We believe that they are insignificant in this case because preliminary calculations considering the clients' clocks did not change the order of results.

That system was able to gather the timings and the supplied information transparently. The unique task of the participants was to click in a button to initialize the starting time. Once the provided test case had succeed (meaning that the framework was correctly coupled) the finishing time was automatically submitted to the server before notifying the success to the participant.

5.8 Data analysis and interpretation for maintenance study

The data of the second study is found on Table 15. This study has provided results similar to the first study. The first eleven values are related to the model-based tool, while the last

Table 15 Maintenance process timings

Primary execution					Secondary execution				
G	**A**	**T**	**P**	**Time**	**G**	**A**	**T**	**P**	**Time**
2	Flights	C	10	02:30.944685	2	Clinic	M	5	01:43.801965
2	Flights	C	9	02:54.232578	2	Clinic	M	3	02:17.158954
1	Delivery	C	8	03:02.751342	1	Clinic	C	8	02:34.248260
2	Flights	C	2	03:11.695431	1	Clinic	C	14	02:57.405545
1	Delivery	C	15	03:31.801582	2	Restaurant	C	2	03:01.547524
2	Flights	C	12	03:45.692316	2	Restaurant	C	10	03:09.169865
2	Flights	C	3	05:09.817914	2	Clinic	M	2	03:25.640129
2	Flights	C	5	05:44.462030	2	Restaurant	C	3	03:39.443080
1	Flights	M	8	05:53.407296	1	Clinic	C	7	04:28.998071
2	Flights	C	11	07:08.687074	2	Restaurant	C	6	04:35.517498
2	Flights	C	6	07:38.576312	2	Restaurant	C	12	04:41.052812
1	Flights	M	4	07:53.595699	2	Restaurant	C	11	04:46.028085
1	Flights	M	14	08:14.148937	1	Restaurant	M	8	04:51.290971
2	Delivery	M	3	08:27.092566	2	Clinic	M	6	04:53.800449
1	Delivery	C	1	08:37.138931	1	Restaurant	M	15	04:58.094389
1	Flights	M	13	08:50.185469	1	Clinic	C	15	05:21.846560
1	Flights	M	1	09:15.253791	2	Restaurant	C	5	05:42.389865
2	Delivery	M	5	09:15.934211	2	Clinic	M	10	07:18.533351
1	Delivery	C	14	09:32.031612	1	Restaurant	M	14	07:24.342788
1	Delivery	C	7	10:04.694800	1	Clinic	C	16	07:37.332151
1	Flights	M	15	11:07.617639	1	Restaurant	M	1	07:44.516376
2	Delivery	M	6	11:32.482992	2	Clinic	M	11	08:08.144168
2	Delivery	M	2	11:49.247460	2	Restaurant	C	9	08:13.115942
1	Delivery	C	16	12:12.576158	1	Restaurant	M	13	08:32.056119
1	Flights	M	7	12:27.297563	1	Restaurant	M	16	11:28.592180
1	Delivery	C	13	12:49.443610	1	Restaurant	M	7	11:45.459699
2	Delivery	M	11	13:00.604583	2	Clinic	M	9	12:42.958789
1	Delivery	C	4	13:25.433748	1	Restaurant	M	4	13:57.879299
2	Delivery	M	9	15:51.117061	1	Clinic	C	1	14:46.465482
2	Delivery	M	12	15:56.048486	1	Clinic	C	4	17:55.176353
2	Delivery	M	10	21:23.533192	1	Clinic	C	13	18:02.486509
1	Flights	M	16	32:32.875079	2	Clinic	M	12	25:54.176697

The timings table contains data captured for each study, in this case, the maintenance study. By analysing this table, it is possible to identify the time of each participant, their group, the technique and the considered application.

four are related to the conventional technique. Only the Participant 16 was able to reuse the framework faster by applying the conventional process, which contradicts the results of the same participant in the previous study. This participant said he got confused when he had to correct the reuse model. That was the reason why he had to restart the process from the very beginning, causing this longer time.

The plots for the maintenance study are found on Figure 11. These plots follow the same guidelines that were used when plotting the graphs for the previous study. Considering the timings of the maintenance study, the reuse model edition does not provide any advantage in terms of productivity, since most of participants took longer to edit the model than the reuse code.

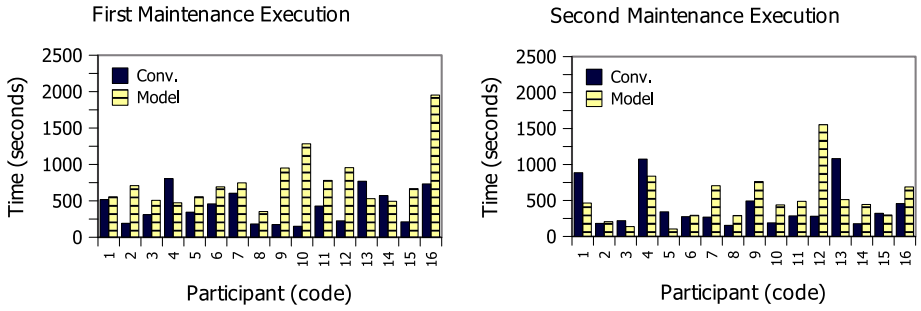


Figure 11 Maintenance process timings bars graph. A graph in which the maintenance experiment timings are plotted. Each bar represents the time taken by each participant in order to complete the maintenance activities, which is identifiable by the participant number.

Table 16 illustrates the average timings and their proportions. Considering only the average time, the participants who applied the conventional technique took less time than their counterparts who used our model-based approach.

6 Results and discussion

6.1 Hypotheses testing for reuse study

In this section, we present statistical calculations to evaluate the data of the reuse study. We applied Paired T-Tests for each execution and another T-Test after removing eight outliers. The time consumed in each execution was processed using the statistic computation environment "R" (Free Software Foundation, Inc 2012). The results of the T-Tests are shown on Table 17, which is actually a pair of tables. The time unit is "seconds".

The first columns of these tables contain the type of T-Test and the second ones indicate the source of the data. The "Means" columns indicate the resultant mean for each T-Test. For a paired T-Test, there is one mean, which is the average of subtracting each set member by its counterpart in the other set. For the non-paired T-Tests, there are two means, which are the averages for each set. In this case, the first set represents the conventional technique; the second set represents the use of the model-based tool. The "d.t." columns stand for the degrees of freedom, which is related to how many different values are found in the sets; "t"and "p" are variables considered in the hypothesis testing.

The Paired T-Test is used to compare the differences between two samples related to each participant. In this case, the time difference of every participant is considered individually; then, the means of the differences are calculated. In the "Two-Sided" T-Tests, which are unpaired, the means are calculated for the entire group, because a participant

Table 16 Maintenance study average timings

A.	Tech.	Avg.	Sum of Avg.	Percents
First	Conv.	06:57.498758	13:55.762733	39.5521%
Replication		06:58.263975		
First	Model	12:43.152626	21:17.305521	60.4479%
Replication		08:34.152895		
Total			35:13.068254	100.0000%

The Study Average Timings table contains averages for the Maintenance Study. It is possible to compare the general time effort needed to complete the activities of both techniques.

Table 17 Reuse study t-test results

T-Test	Data	Means	d.f.	t	p
Paired	First	488.4596	15	5.841634	$3.243855 \cdot 10^{-05}$
Paired	Replication	417.8927	15	5.285366	$9.156136 \cdot 10^{-05}$
Two-sided	Both	771.4236 409.4295	43.70626	6.977408	$1.276575 \cdot 10^{-08}$

T-Test is a statistical test used to determine the correct hypothesis for both studies. This table lists the results for the reuse study.

may be an outlier in a specific technique, which breaks the pairs. It is referred to as two-sided because the two sets have the same number of elements, since the same number of outliers was removed from each group.

The "Chi-squared test" was applied to both studies in order to detect the outliers, which were then removed when calculating the unpaired T-Test. On the table, the unpaired T-Tests are refered as "Two-sided". The results of the "Chi-squared test" for the reuse study are found on Table 18. The 'M' in the techniques column indicates the use of our tool, while 'C' indicates the conventional technique. The group column indicates the number of the group. The X^2 indicates the result of subtracting each value by the variance of the complete set. The position column indicates their position on the set, i.e., highest or lowest. The outlier column shows the timings in seconds that were considered abnormal.

In order to achieve a better visualization of the outliers, we also provide two plots of the data sets. In Figure 12 there are line graphs which may be used to visualize the dispersion of the timing records. In these plots, the timings for each technique are ordered by their performance; therefore, the participant numbers in these plots are not related to their identification codes.

Considering the reuse study and according to the analysis from Table 17 we can state the following. Since all p-values are less than the margin of error (0.01%), which corresponds to the established significance level of 99.99%, then, statistically, we can reject the "$H0_r$" hypothesis that states the techniques are equivalent. Since every t-value is positive, we can accept the "Hp_r" hypothesis, which implies that the conventional technique takes more time than ours.

6.2 Hypotheses testing for maintenance study

In this section, we present statistical calculations to evaluate the data of the maintenance study. Similarly to the reuse study, we applied Paired T-Tests for each execution

Table 18 Chi-squared test for outlier detection applied on reuse study

Study	T.	G.	X^2	p	Position	Outlier
First	C	1	5.104305	0.02386654	highest	2489.414342
		2	2.930583	0.08691612	highest	1390.72776
	M	1	4.091151	0.04310829	highest	1203.920754
		2	2.228028	0.1355267	highest	482.570996
Replication	C	1	4.552248	0.03287556	highest	1698.301114
		2	5.013908	0.02514448	highest	1682.391335
	M	1	3.917559	0.04778423	highest	1029.073104
		2	2.943313	0.08623369	lowest	179.467569

Chi-squared test is a statistical test used to detect outliers. It was employed to detect eight outliers in the reuse study. These outliers are removed in the last t-test.

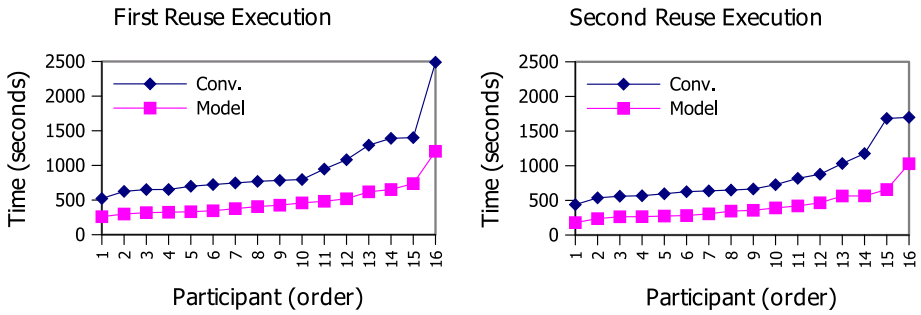


Figure 12 Reuse process timings line graph. A graph in which the reuse experiment timings are plotted. The timings are plotted as points in ascendant order, which are then linked with lines. This graph is useful to visualize the data dispersion.

and another T-Test after removing eight outliers. The seconds that were spent during the process were processed using the statistic computation environment "R" (Free Software Foundation, Inc 2012). The results of the T-Tests are shown on Table 19.

The first column of this table contain the type of T-Test. The second columns indicate the source of the data, which refers to the datasets created for each technique. The "Means" columns indicate the resultant means. The "d.t." columns stand for the degree of freedom; "t" and "p" are variables considered in the hypothesis testing.

The "Chi-squared test" was applied in order to detect the outlier. The results of the "Chi-squared test" for the maintenance study are found on Table 20. These outliers were removed when calculating the unpaired T-Test. On the table, the unpaired T-Test is refered as "Two-sided". The 'M' in the *techniques* column indicates the use of our tool, while 'C' indicates the conventional technique. The *group* column indicates the number of the group. The X^2 indicates the results of an comparison to the variance of the complete set. The position column indicates their position on the set, i.e., highest or lowest. And finally, the outlier column shows the timings in seconds that were considered abnormal.

In order to achieve better visualization of the outliers, we also provide two plots of the data sets. In Figure 13, there are line graphs which may be used to visualize the dispersion of the timing records. In these plots, the timings for each technique are ordered independently. Therefore, the participant numbers in these plots are not related to their identification codes.

If we take into consideration the maintenance study and its analysis illustrated on Table 19, we cannot reject the "$H0_m$" hypothesis that states the techniques are equivalent because all p-values are bigger than the margin of error (0.01%), which corresponds to the established significance level of 99.99%. Therefore, statistically , we can assume that the effort needed to edit a reuse code and a reuse model is approximately equal.

Table 19 Maintenance study t-test results

T-test	Data	Means	d.f.	t	p
Paired	First	-345.6539	15	-3.971923	0.001227479
Paired	Replication	-95.88892	15	-1.191781	0.2518624
Two-sided	Both	431.3323 641.0024	24.22097	-2.662684	0.0135614

T-Test is a statistical test used to determine the correct hypothesis for both studies. This table lists the results for the maintenance study.

Table 20 Chi-squared test for outlier detection applied on maintenance study

Study	T.	G.	X^2	p	Position	Outlier
First	C	1	2.350449	0.1252469	lowest	182.751342
		2	2.152789	0.1423112	highest	458.576312
	M	1	5.788559	0.0161308	highest	1952.875079
		2	3.598538	0.05783041	highest	1283.533192
Replication	C	1	1.771974	0.183138	highest	1082.486509
		2	4.338041	0.03726978	highest	493.115942
	M	1	2.422232	0.1196244	highest	837.879299
		2	4.87366	0.02726961	lowest	1554.176697

Chi-squared test is a statistical test used to detect outliers. It was employed to detect eight outliers in the maintenance study. These outliers are removed in the last t-test.

6.3 Threats to validity

6.3.1 Internal validity

- Experience Level of Participants. The different levels of knowledge of the participants could have compromised the data. To mitigate this threat, we divided the participants in two balanced groups considering their experience level and later we rebalanced the groups considering the preliminary results. Although all participants already had a prior experience in how to reuse the CF in the conventional way, during the training phase, they were taught how to make the reuse with the model-based tool and also how to reuse it in the normal way. So, this could have provided the participants even more experience with the conventional technique.
- Productivity under evaluation. The students could have thought that their results in the experiment will influence their grades in the course. In order to mitigate this, we explained to the students that no one was being evaluated and their participation was considered anonymous.
- Facilities used during the study. Different computers and configurations could have affected the recorded timings. However, participants used the same configuration, make, model, and operating system in equal numbers. The participants were not allowed to change their computers during the same activity. This means thatevery participant had to execute every exercise using the same computer.

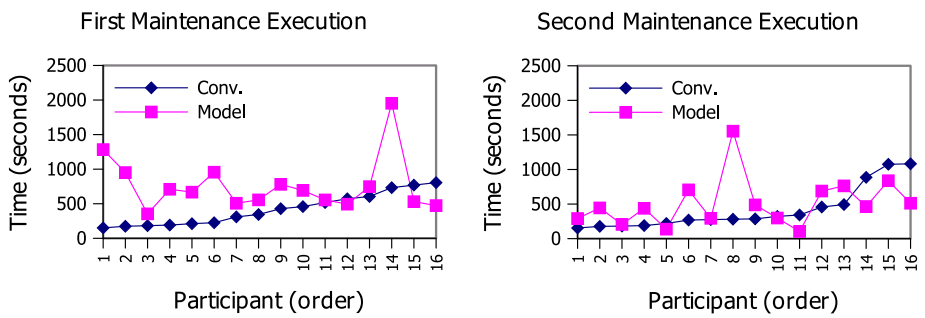


Figure 13 Maintenance process timings line graph. A graph in which the maintenance experiment timings are plotted. The timings are plotted as points in ascendant order, which are then linked with lines. This graph is useful to visualize the data dispersion.

6.3.2 Validity by construction

- Hypothesis expectations: the participants already knew the researchers and knew that the model-based tool was supposed to ease the reuse process, which reflects one of our hypothesis. Both of these issues could affect the collected data and cause the experiment to be less impartial. In order to avoid impartiality, we enforced that the participants had to keep a steady pace during the whole study.

6.3.3 External validity

- Interaction between configuration and treatment. It is possible that the reuse exercises were not accurate for every reuse of a crosscutting framework for real world applications. Only a single crosscutting framework was considered in our study and the base applications were of the same complexity. To mitigate this threat, we designed the exercises with applications that were based on the ones existing in reality.

6.3.4 Conclusion validity

- Measure reliability. It refers to metrics used to measuring the reuse effort. To mitigate this threat, we have used only the time, necessary to complete the process, which was captured by a data collector in order to allow better precision;
- Low statistic power. The ability of a statistic test to reveal the reliable data. We applied three T-Tests to analyze the experiment data statistically to avoid the low statistic power.

7 Related work

The approach proposed by Cechticky et al. (2003) allows the reuse of object-oriented application framework by applying the tool called OBS Instantiation Environment. That tool supports graphical models to define the settings to generate the expected application. The model-to-code transformation generates a new application that reuses the framework.

In another related work, Braga and Masiero (2003) proposed a process to create framework instantiation tools. The process is specific for application frameworks defined by a pattern language. The process application assures that the tool is capable of generating every possible framework variability.

The approach defined by Czarnecki et al. (2006) defines a round-trip process to create domain specific languages for framework documentation. These languages can be employed to represent the information that the framework programming interfaces need during the instantiation and the description of these interfaces. This is a bilateral process, i.e., two transformers are fashioned: a transformer from model to code and a transformer from code to model. Generated codes are transformed back to models. This allow the comparison of the source model with the generated model, which should be perfectly equal. If differences are found, the language or the transformers should be improved in the activity called “conciliation”.

Santos et al. proposed a process and a tool to suport framework reuse (Santos et al. 2008). In this approach, the domain engineer must supply a reuse example which must be tagged. These tags mark points of the reuse example that is to be replaced in order to create different applications.

The tags are mapped into a domain specific language that lists information that the application engineer should supply in order to reuse the framework. This domain specific language instances are, then, interpreted by a tool that is capable of listing the points to complete framework instantiation. This is the only related work that uses AspectJ and the aspect-oriented programming.

Our proposal differs from their approach on the following topics: 1) their approach is restricted to frameworks known during the development of the tool; 2) the reuse process is applied to application frameworks, which are used to create new applications.

Another approach was proposed by Oliveira et al. (2011). Their approach can be applied to a greater number of object oriented frameworks. After the framework development, its developer may use the approach to ease the reuse by writing the cookbook in a formal language known as Reuse Definition Language (RDL) which can also be used to generate the source code. This process allows us to select variabilities and resources during the reuse procedure, as long as the framework engineer specifies the RDL code correctly. These approaches were created to support the reuse procedure during the final development stages. Therefore, the approach that is proposed in this paper differs from others by supporting earlier development phases. This allows the application engineer to initiate the reuse process since the analysis phase while developing an application compatible to the reused frameworks.

Although the approach proposed by Cechticky et al. (2003) is specific for only one framework, it can be employed since the design phase. The other related approach can be employed in a greater number of frameworks: however, it is used on a lower abstraction level, and does not support the design phase. Another difference is the generation of aspect-oriented code, which improves code modularization. Finally, the last difference that must be pointed out is the use of experiments to evaluate the approach, while the presented related works only show case studies employing their tools.

8 Conclusions

In this article we presented a model-based approach that raises the abstraction level of reusing Crosscutting Frameworks (CFs) - a type of aspect-oriented framework. The approach is supported by two models, called Reuse Requirements Model (RRM) and Reuse Model (RM). The RRM serves as a graphical view for enhancing cookbooks and the RM supports application engineers in performing the reuse process by filling in this model and generating the source-code. Considering our approach, a new reuse process is delineated allowing engineers to start the reuse in early development phases. Using our approach, application developers do not need to worry about either architectural or source-code details, shortening the time necessary to conduct the process.

We have evaluated our approach by means of two experiments. The first one was focused on comparing the productivity of our model-based approach to the ad-hoc approach. The results showed the improvement of approximately 97% in favor of our approach. We claim that this improvement can be influenced by the framework characteristics but not by the application characteristics. If a CF requires a lot of heterogeneous joinpoints we think this percentage will go down because the application engineer will need to write the joinpoints (method names, for instance) either using both our approach or the ad-hoc one. However, if the CF is heavily based on inter-type declarations and the

returning of values, then we claim that the productivity can be even higher, as it is very straightforward to do so while using our approach.

The second experiment was focused on observing the effort in maintaining applications that were developed with our approach (CF-based applications) and with the ad-hoc one. It was not possible to conclude which process takes less effort in this case; however, we believe that they are approximately equivalent. The participants argued that the tool could be improved to avoid opening new forms while entering the model attributes, which, as they claim, had disrupted their work and prevented them from reaching a better performance in this case. It is important that in this experiment we did not provide errors that developers could create while using the conventional approach, since our model approach shields the developers from doing that. We have also provided the raw data gathered during the studies as Additional files 1 and 2.

As the possible limitations of our work, we can mention the following. Once the models have been created on top of the Eclipse Modeling Project, they cannot be used in another development environment. Besides, the code generator only produce codes for AspectJ, therefore, only crosscutting frameworks developed in this language can be currently supported. A simple extension is possible to allow this approach to support the reuse of non-crosscutting frameworks written in Java and AspectJ. Also, we have not yet evaluated how to deal with coupling of multiple CFs to a single base application. Although this functionality is already supported in our approach, some frameworks may select the same joinpoints, which may cause conflicts and lead to unpredictable results.

Long term future works are: (i) providing a support for framework engineers so that they do not have to build the RRM manually. The idea is to develop a tool which can assist them in creating this model in a more automatic way; (ii) performing an experiment to verify whether the abstractions of the model elements are on a suitable level (iii) analyzing the reusability of the metamodel abstract classes.

Additional files

Additional file 1: Contains the raw data gathered during the Reuse study.
Additional file 2: Contains the raw data gathered during the Maintenance study.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
Every author was important for the completion of this article, however, their previous activities are also important to reach the research results, then, these activities are listed in this section. TG developed the models, model editors and model transformers. He also designed and conducted the studies and the considered applications. This work was also presented as a master's thesis in Federal University of São Carlos, Brazil. RSD was a contributor who developed the related tools in which this work is related to. He developed a repository for crosscutting frameworks that allow their sharing and integrates to the tool described herein in order to provide a full featured crosscutting framework reuse environment. These tools are available as Eclipse plugins. VVdC is a professor at Federal University of São Carlos which is responsible for the crosscutting framework reuse project. He also developed the crosscutting framework used as example and worked on the study executions. OPL is a professor at Polytechnic University of Valencia that provided useful background information regarding model-driven development and code generator tools. He is also part of the crosscutting framework reuse project. All authors read and approved the final manuscript.

Acknowledgements
The authors would like to thank CNPq for funding (Processes 132996/2010-3 and 560241/2010-0) and for the Universal Project (Process Number 483106/2009-7) in which this article was created. Thiago Gottardi would also like to thank FAPESP (Process 2011/04064-8).

Author details
[1]Departmento de Computação, Universidade Federal de São Carlos, Caixa Postal 676, 13.565-905, São Carlos, São Paulo, Brazil. [2]Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo, Av. Trabalhador São Carlense, 400, São Carlos, São Paulo, Brazil. [3]Universidad Politecnica de Valencia, Camino de Vera s/n, Valencia, Spain.

Received: 7 February 2013 Accepted: 6 September 2013
Published: 29 October 2013

References

Antkiewicz M, Czarnecki K (2006) Framework-specific modeling languages with round-trip engineering. In: ACM/IEEE 9th international conference on model driven engineering languages and systems (MoDELS). Springer-Verlag, Genova, pp 692–706. http://www.springerlink.com/content/y08152212701l160/fulltext.pdf

Apache Software Foundation (2012) Apache Derby. http://db.apache.org/derby/

AspectJ Team (2003) The AspectJ(TM) Programming Guide. http://www.eclipse.org/aspectj/doc/released/progguide/

Braga R, Masiero P (2003) Building a wizard for framework instantiation based on a pattern language. In: Konstantas D, Léonard M, Pigneur Y, Patel S (eds) Object-oriented information systems, Volume 2817 of Lecture notes in computer science. Springer, Berlin / Heidelberg, pp 95–106. http://dx.doi.org/10.1007/978-3-540-45242-3_10

Bynens M, Landuyt D, Truyen E, Joosen W (2010) Towards reusable aspects: the mismatch problem. In: Workshop on Aspect, Components and Patterns for Infrastructure Software (ACP4IS'10). Rennes and Saint Malo, France. ACM, New York, NY, USA, pp 17–20

Camargo VV, Masiero PC (2005) Frameworks Orientados A Aspectos. In: Anais Do 19° Simpósio Brasileiro De Engenharia De Software (SBES'2005). Uberlândia-MG, Brasil, Outubro

Camargo VV, Masiero PC (2004) An approach to design crosscutting framework families. In: Proc. of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software, ACP4IS '08. Brussels, Belgium. ACM, New York, NY, USA. http://dl.acm.org/citation.cfm?id=1404891.1404894

Cechticky V, Chevalley P, Pasetti A, Schaufelberger W (2003) A generative approach to framework instantiation. In: Proceedings of the 2nd international conference on Generative programming and component engineering, GPCE '03. Springer-Verlag, New York, Inc., New York, pp 267–286. http://portal.acm.org/citation.cfm?id=954186.954203

Clark T, Evans A, Sammut P, Willans J (2004) Transformation Language Design: A Metamodelling Foundation, ICGT, Volume 3256 of Graph Transformations, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, pp 13–21

Clements P, Northrop L (2002) Software product lines: practices and patterns, 3rd edn. The SEI series in software engineering, 563 pages, first edition. Addison-Wesley Professional, Boston, United States of America. http://www.pearsonhighered.com/educator/product/Software-Product-Lines-Practices-and-Patterns/9780201703320.page

Cunha C, Sobral J, Monteiro M (2006) Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In: Aspect-Oriented Software Development Conference (AOSD'06). Bonn, Germany. ACM, New York, NY, USA

Eclipse Consortium (2011) Graphical Modeling Framework, version 1.5.0. Graphical Modeling Project. http://www.eclipse.org/modeling/gmp/

Efftinge S (2006) openArchitectureWare 4.1 Xtend language reference. http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html

Fayad M, Schmidt DC (1997) Object-oriented application frameworks. Commun ACM 40: 32–38

Fowler M (2010) Domain specific languages, 1st edition. 640 pages, first edition. Addison-Wesley Professional, Boston, United States of America. http://www.pearsonhighered.com/educator/product/DomainSpecific-Languages/9780321712943.page

France R, Rumpe B (2007) Model-driven development of complex software: a research roadmap. In: 2007 Future of Software Engineering, FOSE 07. IEEE Computer Society, Washington, pp 37–54

Free Software Foundation, Inc (2012) R. http://www.r-project.org/

Huang M, Wang C, Zhang L (2004) Towards a reusable and generic aspect library. In: Workshop of the Aspect Oriented Software Development Conference at AOSDSEC'04, AOSD'04. Lancaster, United Kingdom. ACM, New York, NY, USA

Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, marc Loingtier J, Irwin J (1997) Aspect-oriented programming. In: ECOOP. Springer-Verlag, Heidelberger, Berlin, Germany

Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG (2001) An overview of aspectJ. Springer-Verlag, Heidelberger, Berlin, Germany, pp 327–353

Kulesza U, Alves E, Garcia R, Lucena CJPD, Borba P (2006) Improving Extensibility of object-oriented frameworks with aspect-oriented programming. In: Proc. of the 9th Intl Conf. on software reuse (ICSR'06). Torino, Italy, June 12-15, 2006, Lecture Notes in Computer Science, Programming and Software Engineering, vol 4039. Springer-Verlag, Heidelberger, Berlin, Germany, pp 231–245

Mortensen M, Ghosh S (2006) Creating pluggable and reusable non-functional aspects in AspectC++. In: Proceedings of the fifth AOSD workshop on aspects, components, and patterns for infrastructure Software. Bonn, Germany. ACM, New York, NY, USA

Oliveira TC, Alencar P, Cowan D (2011) ReuseTool-An extensible tool support for object-oriented framework reuse. J Syst Softw 84(12): 2234–2252. http://dx.doi.org/10.1016/j.jss.2011.06.030

Pastor O, Molina JC (2007) Model-driven architecture in practice: a software production environment based on conceptual modeling. Springer-Verlag, New York, Secaucus

Sakenou D, Mehner K, Herrmann S, Sudhof H (2006) Patterns for re-usable aspects in object teams. In: Net Object Days. Erfurt, Germany. Object Teams, Technische Universität Berlin, Berlin, Germany

Santos AL, Koskimies K, Lopes A (2008) Automated domain-specific modeling languages for generating framework-based applications. Softw Product Line Conf Int 0: 149–158

Schmidt DC (2006) Model-driven engineering. IEEE Computer 39(2). http://www.truststc.org/pubs/30.html

Shah V, Hill V (2004) An aspect-oriented security framework: lessons learned. In: Workshop of the Aspect Oriented Software Development Conference at AOSDSEC'04, AOSD'04. Lancaster, United Kingdom. ACM, New York, NY, USA

Soares S, Laureano E, Borba P (2006) Distribution and persistence as aspects. Software: Practice and Experience 36(7): 711–759. John Wiley & Sons, Ltd. Hoboken, NJ, USA. http://onlinelibrary.wiley.com/doi/10.1002/spe.715/abstract

Soudarajan N, Khatchadourian R (2009) Specifying reusable aspects. In: Asian Workshop on Aspect-Oriented and Modular Software Development (AOAsia'09). Auckland, New Zealand. AOAsia, Chinese University of Hong Kong, Hong Kong, People's Republic of China

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in software engineering: an introduction. First edition. 204 pages. Kluwer Academic Publishers, Norwell, MA, USA

Zanon I, Camargo VV, Penteado RAD (2010) Reestructuring an application framework with a persistence crosscutting framework. INFOCOMP 1: 9–16

doi:10.1186/2195-1721-1-4
Cite this article as: Gottardi *et al.*: Model-based reuse for crosscutting frameworks: assessing reuse and maintenance effort. *Journal of Software Engineering Research and Development* 2013 **1**:4.

Anexo 2: Comprovante da publicação no *International Conference on Information Reuse and Integration* (IRI'13)

- Uma cópia do artigo é apresentado a seguir (a partir da próxima página).

An Approach to Develop Frameworks from Feature Models

Matheus Viana, Rosângela Penteado, Antônio do Prado
Department of Computing
Federal University of São Carlos
13565-905, São Carlos, SP, Brazil
Email: {matheus_viana, rosangela, prado}@dc.ufscar.br

Rafael Durelli
Institute of Mathematical and Computer Sciences
University of São Paulo
13566-590, São Carlos, SP, Brazil
rdurelli@icmc.usp.br

***Abstract*—Frameworks are reusable software composed of concrete and abstract classes that implement the functionality of a domain. Applications can reuse framework design and code in order to improve their quality and be developed more efficiently. However, to develop software for reuse, such as a framework, is harder than to develop an application. Hence, in this paper we present an approach, named From Features to Framework (F3), to facilitate the development of white box frameworks. This approach is divided in two steps: Domain Modeling, in which the features of the framework are defined; and Framework Construction, in which the framework is designed and implemented according to its features and their relationships. We also present an experiment that evaluated the F3 approach showing that it makes framework development easier and more efficient.**

I. Introduction

Reuse is a practice that aims to reduce time spent in development process and to increase software quality. These advantages can be achieved because the software is not developed from scratch and the reusable artifacts were previously tested [1]. There are different levels of reuse. Copy/paste is the simplest one. Programming languages contain mechanisms, such as class inheritance, that provide reuse of code. Yet there are other more sophisticated forms of reuse, such as frameworks, that provide not only reuse of code, but also design and experience [2].

Frameworks are reusable software composed of concrete and abstract classes that implement the functionality of a domain [3]. Applications reuse the design and the implementation of a framework, adding their specific characteristics to its functionality [4], [5].

Despite the advantages frameworks offer, they are more complex to develop than applications [6]. Frameworks demand an adaptable design. Their classes will be reused by applications that are unknown during framework development, thereby frameworks need mechanisms to identify and to access application-specific classes. Thus, design patterns and advanced resources of programing languages, such as abstract classes, interfaces, polymorphism, generics and reflection, are often used in framework development. In addition to design and implementation complexities, it is also necessary to determine the domain of applications of the framework, the features that compose this domain and the rules that constraint these features [7].

In a previous paper we presented an approach for building Domain-Specific Modeling Languages (DSML) to facilitate framework reuse [8]. In that approach a DSML could be built by identifying the features of the framework domain and the information required to instantiate them. Then application models created with the DSML could be used as input for an application generator to transform them into application code. The experiment presented in this previous paper showed that, besides the gain of efficiency obtained from code generation, the use of DSML protects developers from framework complexities.

In order to promote reuse in application development, in this paper we propose an approach, named From Features to Framework (F3), that aims to facilitate the development of white box frameworks. In this approach framework development starts from defining its domain in a F3 model, which is an extended version of the feature model. Then a set of patterns, called F3 patterns, assist the design and the implementation of the framework according to the features defined in its F3 model.

We also have carried out an experiment in order to verify whether the F3 approach leads the developer to devise frameworks better than the adhoc one. The experiment showed that the F3 approach reduced the problems of incoherence, structure, bad smells and interface found in the outcome frameworks and, consequently, reduced the time spent to develop these frameworks.

The remainder of this paper is organized as follows: the background concepts applied in this research are discussed in Section II; the F3 approach is presented in Section III; an experiment to evaluate the F3 approach is show in Section IV; some related works are discussed in Section V; and conclusions and further works are presented in Section VI.

II. Background

The basic concepts applied in the F3 approach are about patterns, frameworks and domain engineering.

Patterns are successful solutions that can be reapplied to different contexts. They provide reuse of experience to help developers to solve common problems [3]. The documentation of a pattern usually contains its name, the context it can be applied, the problem it is intended to solve, the solution it proposes, illustrative class models and examples of use [9].

IEEE IRI 2013, August 14-16, 2013, San Francisco, California, USA
978-1-4799-1050-2/13/$31.00 ©2013 IEEE

Frameworks act like skeletons that can be instantiated to implement applications [3]. Their classes embody an abstract design to provide solutions for domains of applications [5]. Applications are connected to a framework by reusing its classes. Unlike library classes, whose execution flux is controlled by applications, frameworks control the execution flux accessing the application-specific code [4].

The fixed parts of the frameworks, known as frozen spots, implement common functionality of the domain that is reused by all applications. The variable parts, known as hot spots, can change according to the specifications of the desired application [5].

According to the way they are reused, frameworks can be classified as: white box, which are reused by class specialization; black box, which work like a set of components; and gray box, which are reused by the two previous ways [3].

Domain engineering represents software development related not to a specific application, but to a domain of applications that share common features [10], [11]. A feature is a distinguishing characteristic that aggregates value to applications. Thus, feature models are often used to model domains, illustrating the features that mandatory or optional, variations and require or exclude other features [12].

Different domain engineering approaches can be found in the literature [11], [13], [14]. Although there are differences between them, the basic idea of these approaches is to identify the features of a domain and to develop the artifacts that implement these features and are reused in application engineering.

Domains can also be modeled with metamodel languages, which are used to create Domain-Specific Languages (DSL) [15]. Metamodels are similar to class models, which makes them more appropriate to developers accustomed to the UML. While in feature models, only features and their constraints are defined, metaclasses in the metamodels can contain attributes and operations. On the other hand, feature models can define dependencies between features, while metamodels depend on declarative languages to do it [15].

III. Proposed Approach

The F3 approach provide mechanisms to define a domain at a high level of abstraction and to systematically construct a framework which implements this domain. Thereby, framework development is divided into two steps: A) Domain Modeling, in which a domain is defined and modeled; and B) Framework Construction, in which the framework is designed and implemented according to its domain. These two steps are illustrated in Figure 1.

Fig. 1: Steps of the F3 approach.

A. Domain Modeling

The domain of applications that can be developed with the framework is defined and modeled in this step. Usually, a domain is defined by analyzing applications that belong to the desired domain or consulting an specialist in this domain [12]–[14]. The first features to be identified are the mandatory ones, because they represent the code asset of the domain and the minimum necessary to develop an application. Then optional features and variants can be added and the dependencies between them can be specified. It is possible to develop a framework with a small domain at first and keep increasing it as soon as more features become required.

In the F3 approach, domains are modeled in F3 models, which are feature models that includes some elements of metamodels, such as attributes, operations and multiplicity, so that frameworks can be developed from these models. As in conventional feature models, features in F3 models must be arranged in a tree-view, in which the main feature is decomposed in others. However, F3 models do not necessarily form a tree, since a feature can have a relationship targeting a sibling or even itself. Moreover, the graphical notation of F3 models is similar to the one of UML class models. This notation has been adopted because it allows that F3 models can be created by using any UML tool. The elements and relationships in F3 models are:

- **Feature**: graphically represented by a rounded square, it must have a name and it can contain any number of attributes and operations;
- **Decomposition**: relationship that indicates that a feature is composed of another feature. Its minimum multiplicity indicates whether the target feature is optional (`0`) or mandatory (`1`). Its maximum multiplicity indicates how many instances of the target feature can be associated to each instance of the source feature. In white box frameworks an instance of a feature is an application class that extends the framework class of this feature. The maximum multiplicity can assume the following values: `1` (simple), for a single feature instance; `*` (multiple), for a list of a single feature instance; and `**` (variant), for a list of different feature instances.
- **Generalization**: relationship that indicates that a feature is a variation and it can be generalized by another feature.
- **Dependency**: relationship that define constraints for feature instantiation. There are two types of dependency: `requires`, when the A feature requires the B feature, an application that contains the A feature has to include the B feature as well; and `excludes`, when the A feature excludes the B feature, no application can include both features at the same time.

A simplified F3 model for the domain of automated vehicles is shown in Figure 2. This domain is based on Lego Mindstorms NXT 2.0[1], whose hardware can be controlled by Lejos Java API[2]. The requirements of this domain are:

[1] http://mindstorms.lego.com/en-us/products/default.aspx#t
[2] http://lejos.sourceforge.net/#t

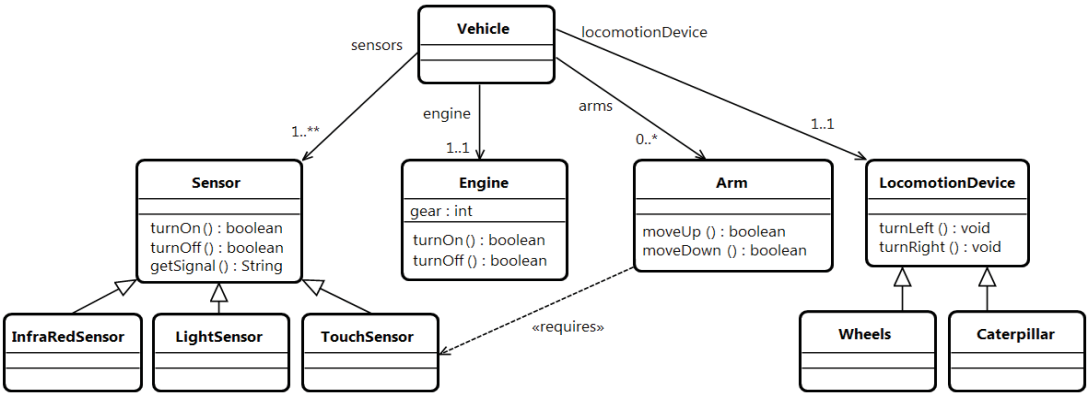


Fig. 2: F3 model for the domain of automated vehicles.

1) An automated vehicle is composed of one engine, one locomotion device, zero or more arms and one or more sensors.
2) Engine provides power for the vehicle. It can be turned on and off and movement sense (forward/reverse) is controlled by its gear.
3) Locomotion device uses the power generated by the engine to move the vehicle and to change its direction. There are two types: wheels or caterpillar.
4) Sensors collect information from environment and return a signal. How the vehicle interprets this signal depends on the purpose of the vehicle. There are three types of sensor: infra-red, light and touch.
5) Arms can be used to grab objects. They can only move up and down and require a touch sensor.

As there are different types of sensor, the relationship between `Vehicle` and `Sensor` is a variant decomposition. A vehicle must have at least one sensor of any type. However, when it has arms, touch sensor becomes necessary, hence the `requires` dependency between `Arm` and `TouchSensor`.

B. Framework Construction

The F3 approach define a set of patterns to assist developers to design and implement a framework from the domain model. These patterns solves problems that go from the creation of classes for the features to the definition of the framework interface. Some of the F3 patterns are presented in Table I.

The documentation of the F3 patterns is organized into topics to help developers to identify when a certain pattern should be used. This documentation is described as follows:

- **Name**: identifies each pattern and summarizes its purpose;
- **Context**: describes a desired behavior for the framework/domain;
- **Scenario/Problem**: describes the arrangement of features and relationships in F3 models that can imply the use of the pattern;
- **Solution**: indicates the code units that should be created to implement the scenario identified by the pattern;
- **Model**: shows a generic graphical representation of the scenario/problem and the solution;
- **Implementation**: displays a fragment of code, in a programming language, that illustrates how the solution can be implemented.

For example, the third pattern listed in Table I, Optional Decomposition, suggests the creation of an operation that must be overridden in the instances of the source feature to specify which class is an instance of the target feature. The documentation of this pattern is:

Name: Optional Decomposition.

Context: when a target feature is optional to a source feature, every instance of the source feature may be associated with a instance of the target feature.

Scenario/Problem: a feature has a decomposition relationship with minimum multiplicity equals `0`.

TABLE I: The F3 patterns that are most commonly applied.

Pattern	Purpose
Domain Feature	Indicates the structures that should be created for a feature.
Mandatory Decomposition	Indicates the code units that should be created when there is a mandatory decomposition linking two features.
Optional Decomposition	Indicates the code units that should be created when there is an optional decomposition linking two features.
Simple Decomposition	Indicates the code units that should be created when there is a simple decomposition linking two features.
Multiple Decomposition	Indicates the code units that should be created when there is a multiple decomposition linking two features.
Variant Decomposition	Indicates the code units that should be created when there is a variant decomposition linking two features.
Variant Feature	Defines a hierarchy of classes for features with variants.
Modular Hierarchy	Defines a hierarchy of classes for features with common attributes and operations.
Requiring Dependency	Indicates the code units that should be created when a feature requires another one.
Excluding Dependency	Indicates the code units that should be created when a feature excludes another one.

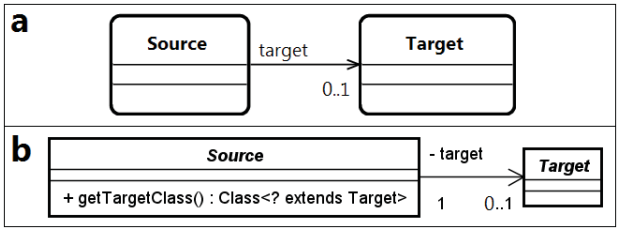


Fig. 3: The (a) pattern scenario and (b) its design solution.

Solution: the class that implements the source feature must have an operation that indicates what class implements the target feature in the applications. By default, this operations returns null indicating that the target feature is not being used.

Model: the (a) scenario and the (b) design solution of this pattern are shown in Figure 3.

Implementation:

```
public abstract class Source {
  public Class<? extends Target> getTargetClass() {
     return null;
  }
}
```

Considering the F3 model in Figure 2, the Optional Decomposition pattern should be applied for the decomposition relationship between `Vehicle` (source) and `Arm` (target). The solution created based on the pattern is shown in Figure 4.

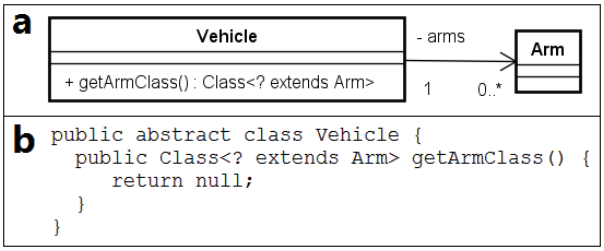


Fig. 4: Solution applied to the relationship between `Vehicle` and `Arm`.

IV. Evaluation

In this section we present an experiment in which the F3 approach has been compared with an adhoc approach. This experiment followed all steps described by Wohlin et al. (2000) and it can be defined as: (*i*) **Analyse** the F3 approach, described in Section III, (*ii*) **for the purpose of** evaluation, (*iii*) **with respect to** efficiency (time) and easiness (problems), (*iv*) **from the point of view of** the developer, and (*v*) **in the context of** MSc and PhD students of Computer Science.

The context of the experiment corresponds to multi-test within object study [16], since the experiment consisted of experimental tests executed by a group of subjects to study a single approach, which is the F3 approach.

A. Planning

The planning phase was divided into the six steps described in the following subsections:

1. Context Selection

The experiment has been performed in laboratory of Computer Science at an university environment. It involved the participation of MSc and PhD students of Computer Science with prior experience in software development using Java language, design patterns and frameworks.

2. Formulation of Hypotheses

The first question the experiment had to answer was: **RQ_1:** "**Which approach takes to a more efficient framework development in terms of time?**". In order to answer this question, the subjects had to measure the time spent (τ) to develop each framework. According to this, the following hypotheses were elaborated:

RQ_1, Null hypothesis, H_0: The F3 approach is not more efficient than the adhoc one in terms of time spent to develop a framework. It can be formalized as:

RQ_1H_0: $\tau_{F3} \geq \tau_{adhoc}$

RQ_1, Alternative hypothesis, H_1: The F3 approach is more efficient than the adhoc one in terms of time spent to develop a framework. It can be formalized as:

RQ_1H_1: $\tau_{F3} < \tau_{adhoc}$

The second question the experiment had to answer was: **RQ_2:** "**Which approach facilitates framework development reducing the number of problems in the frameworks during their development?**". In order to answer this question, we analyzed the reports of the subjects, in which they documented the problems found (ρ) in their frameworks during development, as well as the source-code of the outcome frameworks. By problems we mean defects and bad smells in the source-code of the frameworks. According to this, the following hypotheses were elaborated:

RQ_2, Null hypothesis, H_0: The F3 approach does not facilitate framework development, as the number of problems in the frameworks during their development is not reduced. It can be formalized as:

RQ_2H_0: $\rho_{F3} \geq \rho_{adhoc}$

RQ_2, Alternative hypothesis, H_1: The F3 approach facilitates framework development, reducing the number of problems in the frameworks during their development. It can be formalized as:

RQ_2H_1: $\rho_{F3} < \rho_{adhoc}$

3. Variables Selection

The dependent variables of this experiment were "**time spent to develop a framework**" and "**number of problems found in the frameworks**". The independent variables were:

- **Application**: Each subject had to develop two frameworks: one (Fw1) for the domain of trade and rental transactions and the other (Fw2) for the domain of automated vehicles. Both Fw1 and Fw2 were composed of 10 features.
- **Development Environment**: Eclipse 4.2.1, Astah Community 6.4.
- **Technologies**: Java version 6.

4. Selection of Subjects

Subjects were selected according to convenience sampling [16]. In this non-probabilistic technique, the selected participants were the closest and most convenient to conduct the experiment. Altogether, 26 Msc and PhD students voluntarily participated in the experiment.

5. Experiment Design

The experiment followed the design of grouping the subjects in homogeneous blocks [16], avoiding that their experience level could directly impact in the results. We used a Participant Characterization Form to determine the experience level of each subject. In this form the subjects had to answer multiple-choice questions about their knowledge regarding Java programming, design patterns and frameworks.

The design type of the experiment was **one factor with two treatments paired** [16]. The **factor** is the approach used to develop a framework and the **treatments** are the adhoc and the F3 approaches. Each subject had to develop two frameworks, one applying the adhoc approach and the other applying the F3 approach. The order in which the subjects applied the treatments had no effect in the result. Therefore, the subjects were divided into two blocks of 13 participants with two tasks, as follows:

- **Block 1**: Task 1, development of Fw1 applying the adhoc approach; and Task 2, development of Fw2 applying the F3 approach;
- **Block 2**: Task 1, development of Fw2 applying the adhoc approach; and Task 2, development of Fw1 applying the F3 approach;

6. Instrumentation

The subjects received all necessary materials to assist them during the execution of the experiment. These documents consist of: textual description and models of the framework domains; manual for creating F3 models; documentation of the F3 patterns; Data Collection Form, in which the subjects had to report the time spent to develop the frameworks and the problems found during their development; one Test Application for each framework, which should be used by the subjects to verify the correctness and the completeness of the outcome frameworks; and Feedback Form, in which the subjects should describe their difficulties and write their opinion after the experiment.

B. Operation

After defining and planning the experiment, its operation was carried out in two steps: (*1*) Preparation and (*2*) Execution.

1. Preparation

At first, the subjects signed a Consent Form, stating the objectives and confidentiality of the experiment, and filled the Participant Characterization Form in, reporting their experience in the concepts and technologies utilized in the experiment. After this, the subjects had a training in: adhoc framework development, in which they learned design patterns and code structures commonly used in frameworks to identify application-specific elements; and the F3 approach. After training, the subjects were able to carry out the experiment tasks.

2. Execution

Before starting the execution of the experiment, the subjects were positioned in the blocks and received the materials referent to their respective Task 1. Each subjects had access to an individual computer equipped with the tools required for framework development and the Test Applications.

When all subjects were commanded to execute Task 1 (applying the adhoc approach), they started to measure the time. They used the Astah Community to create a class model of the framework and then use the Eclipse IDE to implement its source-code. When they finished framework implementation, they executed its Test Application to verify whether or not it was developed as expected. If the Test Application showed a message of a problem, the subjects had to report it in the Data Collection Form and fix the problem(s) found. Only when the Test Application returned a successful message, the subject could stop measuring the time. Task 2 (applying the F3 approach) was performed in a similar way to Task 1. In the end, the subjects received the Feedback Form to comment the difficulties and advantages in applying each approach.

C. Analysis of Data

The experiment data is presented in Table II. In general, the groups developed the tasks satisfactorily and the collected data was within the expected limits. This means that the treatments were executed correctly and in accordance with the planning. The analysis of data is divided into two subsections: (*1*) Descriptive Statistics and (*2*) Hypotheses Testing.

1. Descriptive Statistics

In Table II, it can be seen that the F3 approach spent less time to develop a framework than the adhoc approach, i.e., approximately 38.7% against 61.3%. According to the feedback provided by the subjects in a form, this result is due to the fact that, in the F3 approach, although the subjects have spent part of the time trying to identify the F3 patterns that should be used, they saved some time because these patterns assisted them indicating the classes, attributes and operations that should be created. On the other side, when the subjects were developing the frameworks applying the adhoc approach, they spent part of the time trying to find out the code units they should implement. Moreover, most of the subjects reported that they spent a long time maintaining their frameworks, because the Test application returned lots of problem messages. The dispersion of time spent by the subjects are also represented graphically in a boxplot on left side of Figure 5.

In Table II it is also possible to visualize four types of problems that we analyzed in the outcome frameworks: incoherence, structure, bad smells and interface.

The problem of incoherence indicates that the subjects did not develop the frameworks with the correct features and constraints (mandatory, optional and alternative features) of the domain. In other words, they did not designed and implemented the classes, attributes and operations that could make the framework to behave as expected by its domain. In Table II it can be seen that the F3 approach helped the subjects to develop frameworks with less incoherence problems, approximately, 26% in opposition to 74% for the adhoc approach.

TABLE II: Results of the frameworks developed by the subjects.

Subject	Time Spent (minutes)		Number of Problems									
			Incoherence		Structure		Bad Smells		Interface		Total	
	Adhoc	F3	Adhoc	F3	Adhoc	F3	Adhoc	F3	Adhoc	F3	Adhoc	F3
S1	108	72	5	1	2	0	2	2	7	0	16	3
S2	113	74	7	1	4	1	2	0	4	1	17	3
S3	139	83	9	3	11	1	3	1	12	2	35	7
S4	124	78	7	1	5	2	2	1	7	2	21	6
S5	101	67	4	0	3	0	1	0	3	0	11	0
S6	133	81	8	4	7	3	3	3	9	3	27	13
S7	131	79	5	3	3	1	2	1	6	2	16	7
S8	116	73	6	1	5	0	3	0	5	1	19	2
S9	109	79	7	1	4	2	2	1	7	2	20	6
S10	106	69	4	2	3	0	1	0	3	1	11	3
S11	119	71	4	1	4	1	2	0	7	0	17	2
S12	148	83	8	3	6	1	3	3	11	4	28	11
S13	110	74	4	1	2	1	3	1	5	0	14	3
S14	107	72	2	1	3	0	3	0	6	1	14	2
S15	117	76	5	3	5	2	2	1	4	2	16	8
S16	97	68	3	1	1	0	2	0	3	0	9	1
S17	137	80	8	5	9	4	3	3	10	3	30	15
S18	121	75	4	1	6	2	2	2	2	2	14	7
S19	115	73	3	0	4	1	2	0	4	0	13	1
S20	134	81	7	2	6	3	3	1	9	3	25	9
S21	144	86	9	3	7	3	3	3	12	6	31	15
S22	111	76	3	2	4	1	1	1	5	1	13	5
S23	129	83	7	4	8	3	3	2	11	3	29	12
S24	123	79	5	2	5	1	3	1	7	2	20	6
S25	127	77	5	3	3	1	1	0	4	1	13	5
S26	131	78	6	2	4	1	2	2	6	2	18	7
AVG	**121,154**	**76.4231**	**5.57692**	**1.961538**	**4.76923**	**1.346154**	**2.26923**	**1.115385**	**6.5**	**1.692308**		
%	**61.3198**	**38.6802**	**73.9796**	**26.02041**	**77.9874**	**22,01258**	**67.0455**	**32,95455**	**79,3427**	**20.65728**		

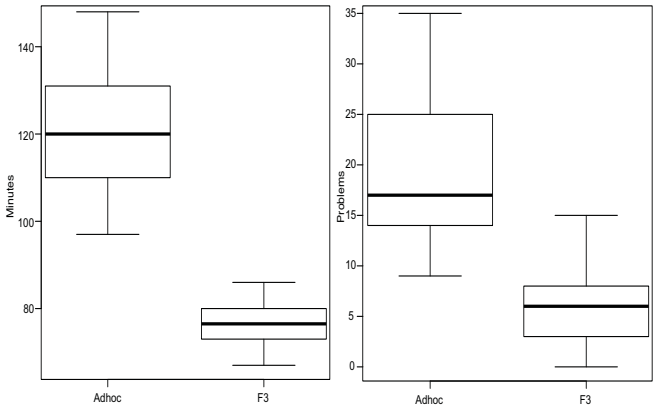


Fig. 5: Dispersion of the total time and number of problems.

The problem of structure indicates that the subjects did not implement the frameworks properly, for example, implementing classes with no constructor, non-abstract when they should be or incorrect relationships. In Table II it can be seen that the F3 approach helped the subjects to develop frameworks with less structure problems, i.e., 22% in opposition to 78%.

The problem of bad smells indicates design weaknesses that do not affect functionality, but make the frameworks harder to maintain. This problem is not a defect, so the Test Applications could not detect it and the subjects did not fixed it. We identified it by analyzing the source-code of the frameworks. In Table II we can remark that the use of the F3 approach resulted in a design with higher quality than the use of the adhoc approach, respectively, 33% against 67%.

The problem of interface indicates absence of getter/setter operations and the lack of operations that allows the applications to reuse the framework and so on. Usually, this kind of problem is a consequence of problems of structure, hence the number of problems of these two types are quite similar. As it can be observed in Table II that the F3 approach helped the subjects to design a better framework interface than when they developed the framework through the adhoc approach, respectively, 21% against 79%.

In the last two columns of Table II it can be seen that the F3T reduced the total number of problems found in the frameworks developed by the subjects. It is also graphically represented in the boxplot on right side of Figure 5.

2. Hypotheses Testing

The objective of this section is to verify with any degree of significance, whether it is possible to reject the nulls hypotheses (see Section IV-A) in favor of the alternative hypothesis based on the data set obtained. As we defined two nulls hypotheses this section is divided into two: (*1*) Hypotheses Testing - Time and (*2*) Hypotheses Testing - Problems.

1) **Hypotheses Testing - Time**: Since some statistical tests are applicable only if the population follows a normal distribution, we applied the Shapiro-Wilk test and created a Q-Q chart to verify whether or not the experiment data departs from linearity before choosing a proper statistical test. As it can be seen in the upper Q-Q charts in Figure 6, the experiment data related to the time spent in framework development is normally distributed. Thus, we decided to apply the Paired T-Test to the experiment data. According to *StatSoft*[3], we carried out this test by calculating: the difference of time between both approaches, $d = \{36, 39, 56, 46, 34, 52, 52, 43, 30, 37, 48, 65, 36, 35, 41, 29, 57, 46, 42, 53, 58, 35, 46, 44,$

[3]http://www.statsoft.com/textbook/distribution-tables/#t

50, 53}; the standard deviation of this difference, S_d = 9.357597; the number of degrees of freedom, $F = N - 1 = 26 - 1 = 25$, where N is the number of subjects; t_0 = 24.3741; and $t_{0.05,25} = 1.708141$. Since $t_0 > t_{0.05,25}$ it is possible to reject the null hypothesis with a two sided test at the 0.05 level. Therefore, statistically, we can assume that the F3 approach reduces the time spent in framework development when compared with the adhoc approach.

2) **Hypotheses Testing - Problems**: Similarly, we used the Shapiro-Wilk test and Q-Q chart on the data shown in the last two columns of Table II, which represent the total number of problems found in the outcome frameworks by using the F3 approach and the adhoc one, respectively. As it can be seen in the lower Q-Q charts in Figure 6, the data depart from linearity, indicating a normal distribution of data. Thus, we also used a Paired T-Test in this case. Again according to *StatSoft*[4], we carried out this test by calculating: the difference of number of problems between both approaches, d = { 13, 14, 28, 15, 11, 14, 9, 17, 14, 8, 15, 17, 11, 12, 8, 8, 15, 7, 12, 16, 16, 8, 17, 14, 8, 11 }; the standard deviation of this difference, S_d = 4.463183; the number of degrees of freedom, $F = N - 1 = 26 - 1 = 25$, where N is the number of subjects; t_0 = 4.463183; and $t_{0.05,25} = 1.708141$. Since $t_0 > t_{0.05,25}$, it is possible to reject the null hypothesis with a two sided test at the 0.05 level. Therefore, statistically, we can conclude that the F3 approach reduces the number of problems found in the outcome frameworks when compared with the adhoc approach.

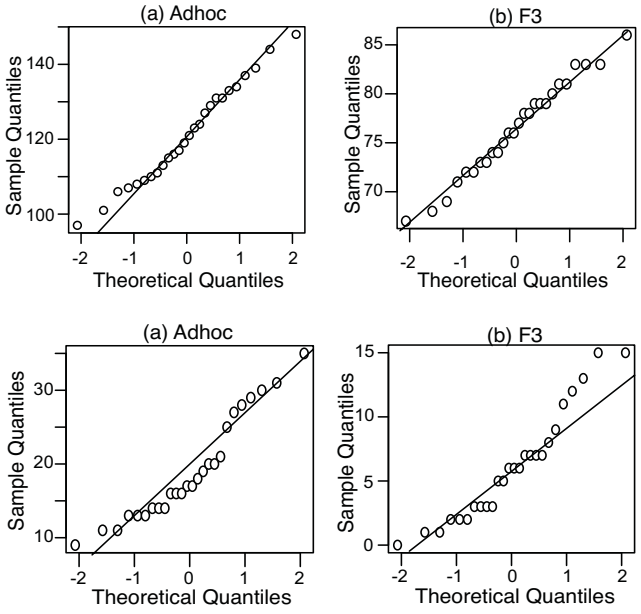


Fig. 6: Q-Q charts of time (upper) and problems found (lower).

[4]http://www.statsoft.com/textbook/distribution-tables/#t

D. Threats to Validity

Internal Validity:

- Experience level of participants: different levels of knowledge of the subjects could affect the collected data. To mitigate this threat, we divided the subjects into two balanced blocks based on their answers in the Participant Characterization Form. All subjects had prior experience in application development reusing frameworks and they were trained in the F3 approach.
- Productivity under evaluation: it might influence the experiment results because subjects often tend to think they are being evaluated. To mitigate this, we explained to the subjects that no one was being evaluated and their participation was considered anonymous.
- Facilities used during the study: different computers and installations could affect the recorded timings. However, the subjects used the same hardware configuration and operating system.

Validity by Construction:

- Hypothesis expectations: the subjects knew the researchers and knew that the F3 approach was supposed to ease framework development before the experiment. These issues could affect the collected data and cause the experiment to be less impartial. In order to keep impartiality, we enforced that the subjects had to keep a steady pace during the whole study.

External Validity:

- Interaction between configuration and treatment: it is possible that the exercises performed in the experiment are not accurate for every framework development in real world applications. Only two frameworks were developed and both had similar complexity. To mitigate this threat, the tasks were designed considering framework domains based on the real world.

Conclusion Validity:

- Measure reliability: it refers to metrics used for measuring development effort. To mitigate this threat we have used only the time spent which was captured in forms fulfilled by the subjects;
- Low statistic power: the ability of a statistic test in reveal reliable data. To mitigate we applied two tests: T-Tests to statistically analyze the time spent to develop the frameworks and Wilcoxon signed-rank test to statistically analyze the number of problems found in the outcome frameworks.

V. Related Works

Xu and Butler [17] proposed an cascaded refactoring method to develop frameworks. In this method, a framework is specified by different models, sorted by abstraction level (from feature model to source-code). Refactorings are performed on these models following their sequence until the framework is completely developed. In the F3 approach the domain is also defined in feature models and framework design and

implementation are assisted by patterns, which provide more information to help developers than refactorings.

Zhang et al. [18] proposed a Feature-Oriented Framework Model Language (FOFML) to develop frameworks following the MDA approach. When this language is used, domain features are defined in a graphical metamodel and domain constraints are specified in a textual role model. Then, framework classes are implemented according to the features and roles defined in these models. In comparison, our approach define domain features and constraints in a single model.

Antkiewicz et al. [19] presented a framework-specific modeling language approach to modeling and instantiation framework. They used feature model to describe functional requirements of frameworks. However, they focused on framework instantiation level concepts and ignored framework design level concepts.

Amatriain and Arumi [20] also proposed a method to develop frameworks through iterative and incremental activities. In their method, the domain of the framework could be defined from existing applications and the framework could be implemented through a series of refactorings over these applications. The advantage of this method is a small initial investment and the reuse of the applications. Although it is not mandatory, the F3 approach can also be applied in iterative and incremental activities, starting from a small domain and then adding features. Applications can also be used to facilitate the identification of the features of the domain. However, the advantage of the F3 approach is the fact that the design and the implementation of the frameworks are performed with the support of patterns specific for framework development.

VI. Concluding Remarks and Future Work

In this paper we proposed the F3 approach to facilitate the development of white box frameworks. In this approach the framework domain is defined in F3 models, which include elements from feature models and metamodels. Then, the framework is designed and implemented with the support of F3 patterns, structuring code units according to the elements and relationships defined in F3 models.

Our approach promotes reuse in different levels. F3 models represent domains that can be reused and improved in different frameworks. The F3 patterns represent reuse of experience on framework development. Moreover, the outcome frameworks can be reused in the development of several applications, acting as a core asset for a software product line.

The experiment presented in this paper indicated that F3 approach facilitates framework development, because it shows developers how to proceed, making them less prone to insert defects and bad smells in the outcome frameworks. Our approach allowed that even subjects with no experience in framework development could execute this task correctly and spending less time.

In future works we intend to create more F3 patterns to deal with other scenarios in F3 models. Moreover, a tool with a F3 model editor and a code generator based in the F3 patterns is being developed to automatize the use of the F3 approach.

Acknowledgment

The authors would like to thank CAPES and FAPESP for sponsoring our research.

References

[1] S. G. Shiva and L. A. Shala. Software reuse: Research and practice. In *Information Technology, 2007. ITNG '07. Fourth International Conference on*, pages 603–609, april 2007.

[2] W. Frakes and K. Kang. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, 31(7):529–536, july 2005.

[3] R. E. Johnson. Frameworks = (Components + Patterns). *Communications of ACM*, 40(10):39–42, Oct 1997.

[4] M. Abi-Antoun. Making Frameworks Work: a Project Retrospective. In *Companion to the 22nd ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 1004–1018, New York, NY, USA, 2007. ACM.

[5] S. Srinivasan. Design Patterns in Object-Oriented Frameworks. *Computer*, 32(2):24 –32, feb 1999.

[6] D. Kirk, M. Roper, and M. Wood. Identifying and Addressing Problems in Object-Oriented Framework Reuse. *Empirical Software Engineering*, 12(3):243–274, Jun 2007.

[7] V. Stanojevic, S. Vlajic, M. Milic, and M. Ognjanovic. Guidelines for Framework Development Process. In *Software Engineering Conference in Russia (CEE-SECR), 7th Central and Eastern European*, pages 1–9, Nov 2011.

[8] M. Viana, R. Penteado, and A. do Prado. Generating Applications: Framework Reuse Supported by Domain-Specific Modeling Languages. In *14th International Conference on Enterprise Information Systems (ICEIS'14)*, Jun 2012.

[9] M. Fowler. Patterns. *IEEE Software*, 20(2):56–57, 2003.

[10] K. Lee, K. C. Kang, and J. Lee. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In *7th International Conference on Software Reuse: Methods, Techniques and Tools*, pages 62–77, London, UK, 2002. Springer-Verlag.

[11] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA): Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[12] J. M. Jezequel. Model-Driven Engineering for Software Product Lines. *ISRN Software Engineering*, 2012, 2012.

[13] H. Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2004.

[14] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J. DeBaud. PuLSE: a Methodology to Develop Ssoftware Product Lines. In *Proceedings of the Symposium on Software Reusability*, pages 122–131. ACM, 1999.

[15] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009.

[16] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[17] L. Xu and G. Butler. Cascaded refactoring for framework development and evolution. *Software Engineering Conference, Australian*, pages 319–330, 2006.

[18] T. Zhang, X. Xiao, H. Wang, and L. Qian. A Feature-Oriented Framework Model for Object-Oriented Framework: An MDA Approach. In *9th IEEE International Conference on Computer and Information Technology*, volume 2, pages 199–204, 2009.

[19] M. Antkiewicz, K. Czarnecki, and M. Stephan. Engineering of Framework-Specific Modeling Languages. *Software Engineering, IEEE Transactions on*, 35(6):795–824, Nov-Dec 2009.

[20] X. Amatriain and P. Arumi. Frameworks Generate Domain-Specific Languages: a Case Study in the Multimedia Domain. *IEEE Transactions on Software Engineering*, 37(4):544–558, Jul-Aug 2011.

Anexo 3: Comprovante da publicação no *Simpósio Brasileiro de Engenharia de Software* (SBES'13)

- Uma cópia do artigo é apresentado a seguir (a partir da próxima página).

F3T: From Features to Frameworks Tool

Matheus Viana, Rosângela Penteado, Antônio do Prado
Department of Computing
Federal University of São Carlos
São Carlos, SP, Brazil
Email: {matheus_viana, rosangela, prado}@dc.ufscar.br

Rafael Durelli
Institute of Mathematical and Computer Sciences
University of São Paulo
São Carlos, SP, Brazil
rdurelli@icmc.usp.br

***Abstract*—Frameworks are used to enhance the quality of applications and the productivity of development process, since applications can be designed and implemented by reusing framework classes. However, frameworks are hard to develop, learn and reuse, due to their adaptive nature. In this paper we present the From Features to Framework Tool (F3T), which supports framework development in two steps: Domain Modeling, in which the features of the framework domain are modeled; and Framework Construction, in which the source-code and the Domain-Specific Modeling Language (DSML) of the framework are generated from the features. In addition, the F3T also supports the use of the framework DSML to model applications and generate their source-code. The F3T has been evaluated in a experiment that is also presented in this paper.**

I. Introduction

Frameworks are reusable software composed of abstract classes implementing the basic functionality of a domain. When an application is developed through framework reuse, the functionality provided by framework classes is complemented with the application requirements. As this application is not developed from scratch, the time spent in its development is reduced and its quality is improved [1]–[3].

Frameworks are often used in the implementation of common application requirements, such as persistence [4] and user interfaces [5]. Moreover, a framework is used as a core asset when many closely related applications are developed in a Software Product Line (SPL) [6], [7]. Common features of the SPL domain are implemented in the framework and applications implement these features reusing framework classes.

However, frameworks are hard to develop, learn and reuse. Their classes must be abstract enough to be reused by applications that are unknown beforehand. Framework developers must define the domain of applications for which the framework is able to be instantiated, how the framework is reused by these applications and how it accesses application-specific classes, among other things [7], [8]. Frameworks have a steep learning curve, since application developers must understand their complex design. Some framework rules may not be apparent in its interface [9]. A framework may contain so many classes and operations that even developers who are conversant with it may make mistakes while they are reusing this framework to develop an application.

In a previous paper we presented an approach for building Domain-Specific Modeling Languages (DSML) to support framework reuse [10]. A DSML can be built by identifying framework features and the information required to instantiate them. Thus, application models created with a DSML can be used to generate application source-code. Experiments have shown that DSMLs protect developers from framework complexities, reduce the occurrence of mistakes made by developers when they are instantiating frameworks to develop applications and reduce the time spent in this instantiation.

In another paper we presented the From Features to Framework (F3) approach, which aims to reduce framework development complexities [11]. In this approach the domain of a framework is defined in a F3 model, which is a extended version of the feature model. Then a set of patterns, called F3 patterns, guides the developer to design and implement a white box framework according to its domain. One of the advantages of this approach is that, besides showing how developers can proceed, the F3 patterns systematizes the process of framework development. This systematization allowed that this process could be automatized by a tool.

Therefore, in this paper we present the From Features to Framework Tool (F3T), which is a plug-in for the Eclipse IDE that supports the use of the F3 approach to develop and reuse frameworks. This tool provides an editor for developers to create a F3 model of a domain. Then, framework source-code and DSML can be generated from the domain defined in this model. Framework source-code is generated as a Java project, while the DSML is generated as a set of Eclipse IDE plug-ins. After being installed, a DSML can be used to model applications. Then, the F3T can be used again to generate the application source-code from models created with the framework DSML. This application reuses the framework previously generated.

We also have carried out an experiment in order to evaluate whether the F3T facilitates framework development or not. The experiment analyzed the time spent in framework development and the number of problems found the source-code of the outcome frameworks.

The remainder of this paper is organized as follows: background concepts are discussed in Section II; the F3 approach is commented in Section III; the F3T is presented in Section IV; an experiment that has evaluated the F3T is presented in Section V; related works are discussed in Section VI; and conclusions and future works are presented in Section VII.

II. Background

The basic concepts applied in the F3T and its approach are presented in this section. All these concepts have reuse as their basic principle. Reuse is a practice that aims: to reduce time spent in a development process, because the software

/13 $31.00 © 2013 IEEE
DOI 10.1109/SBES.2013.15

is not developed from scratch; and to increase the quality of the software, since the reusable practices, models or code were previously tested and granted as successful [12]. Reuse can occur in different levels: executing simple copy/paste commands; referencing operations, classes, modules and other blocks in programming languages; or applying more sophisticated concepts, such as patterns, frameworks, generators and domain engineering [13].

Patterns are successful solutions that can be reapplied to different contexts [3]. They provide reuse of experience helping developers to solve common problems [14]. The documentation of a pattern mainly contains its name, the context it can be applied, the problem it is intended to solve, the solution it proposes, illustrative class models and examples of use. There are patterns for several purposes, such as design, analysis, architectural, implementation, process and organizational patterns [15].

Frameworks act like skeletons that can be instantiated to implement applications [3]. Their classes embody an abstract design to provide solutions for domains of applications [9]. Applications are connected to a framework by reusing its classes. Unlike library classes, whose execution flux is controlled by applications, frameworks control the execution flux accessing the application-specific code [15]. The fixed parts of the frameworks, known as frozen spots, implement common functionality of the domain that is reused by all applications. The variable parts, known as hot spots, can change according to the specifications of the desired application [9]. According to the way they are reused, frameworks can be classified as: white box, which are reused by class specialization; black box, which work like a set of components; and gray box, which are reused by the two previous ways [2].

Generators are tools that transform an artifact into another [16], [17]. There are many types of generators. The most common are Model-to-Model (M2M), Model-to-Text (M2T) and programming language translators [18]. Such as frameworks, generators are related to domains. However, some generators are configurable, being able to change their domain [19]. In this case, templates are used to define the artifacts that can be generated.

A domain of software consists of a set of applications that share common features. A feature is a distinguishing characteristic that aggregates value to applications [20]–[22]. For example, Rental Transaction, Destination Party and Resource could be features of the domain of rental applications. Different domain engineering approaches can be found in the literature [20], [22]–[24]. Although there are differences between them, their basic idea is to model the features of a domain and develop the components that implement these features and are reused in application engineering.

The features of a domain are defined in a feature model, in which they are arranged in a tree-view notation. They can be mandatory or optional, have variations and require or exclude other features. The feature that most represents the purpose of the domain is put in the root and a top-down approach is applied to add the other features. For example, the main purpose of the domain of rental applications is to perform rentals, so Rental is supposed to be the root feature. The other features are arranged following it.

Domains can also be modeled with metamodel languages, which are used to create Domain-Specific Modeling Languages (DSML). Metamodels, such as defined in the MetaObject Facility (MOF) [25], are similar to class models, which makes them more appropriate to developers accustomed to the UML. While in feature models, only features and their constraints are defined, metaclasses in the metamodels can contain attributes and operations. On the other hand, feature models can define dependencies between features, while metamodels depend on declarative languages to do it [18]. A generator can be used along with a DSML to transform models created with this DSML into code. When these models represent applications, the generators are called application generators.

III. The F3 Approach

The F3 is a Domain Engineering approach that aims to develop frameworks for domains of applications. It has two steps: 1) Domain Modeling, in which framework domain is determined; and 2) Framework Construction, in which the framework is designed and implemented according to the features of its domain.

In Domain Modeling step the domain is defined in a feature model. However, an extended version of feature model is used in the F3 approach, because feature models are too abstract to contain information enough for framework development and metamodels depend on other languages to define dependencies and constraints. This extended version, called F3 model, incorporates characteristics of both feature models and metamodels. As in conventional feature models, features in the F3 models can also be arranged in a tree-view, in which the root feature is decomposed in other features. However, features in the F3 models do not necessarily form a tree, since a feature can have a relationship targeting a sibling or even itself, as in metamodels. The elements and relationships in F3 models are:

- **Feature**: graphically represented by a rounded square, it must have a name and it can contain any number of attributes and operations;
- **Decomposition**: relationship that indicates that a feature is composed of another feature. This relationship specifies a minimum and a maximum multiplicity. The minimum multiplicity indicates whether the target feature is optional (`0`) or mandatory (`1`). The maximum multiplicity indicates how many instances of the target feature can be associated to each instance of the source feature. The valid values to the maximum multiplicity are: `1` (simple), for a single feature instance; `*` (multiple), for a list of a single feature instance; and `**` (variant), for any number of feature instances.
- **Generalization**: relationship that indicates that a feature is a variation generalized by another feature.
- **Dependency**: relationship that defines a condition for a feature to be instantiated. There are two types of dependency: `requires`, when the A feature requires the B feature, an application that contains the A feature also has to include the B feature; and `excludes`, when the A feature excludes the B feature, no application can include both features.

Framework Construction step has as output a white box framework for the domain defined in the previous step. The F3 approach defines a set of patterns to assist developers to design and implement frameworks from F3 models. The patterns treat problems that go from the creation of classes for the features to the definition of the framework interface. Some of the F3 patterns are presented in Table I.

TABLE I: Some of the F3 patterns.

Pattern	Purpose
Domain Feature	Indicates structures that should be created for a feature.
Mandatory Decomposition	Indicates code units that should be created when there is a mandatory decomposition linking two features.
Optional Decomposition	Indicates code units that should be created when there is an optional decomposition linking two features.
Simple Decomposition	Indicates code units that should be created when there is a simple decomposition linking two features.
Multiple Decomposition	Indicates code units that should be created when there is a multiple decomposition linking two features.
Variant Decomposition	Indicates code units that should be created when there is a variant decomposition linking two features.
Variant Feature	Defines a class hierarchy for features with variants.
Modular Hierarchy	Defines a class hierarchy for features with common attributes and operations.
Requiring Dependency	Indicates code units that should be created when a feature requires another one.
Excluding Dependency	Indicates code units that should be created when a feature excludes another one.

In addition to indicate the code units that should be created to implement the framework functionality, the F3 patterns also determine how the framework can be reused by the applications. For example, some patterns suggest to include abstract operations in the classes of the framework that allows it to access application-specific information. In addition, the F3 patterns make the development of frameworks systematic, allowing it to be automatized. Thus, the F3T tool was created to automatize the use of the F3 approach, enhancing the processes of framework development.

IV. The F3T

The F3T assists developers to apply the F3 approach in the development of white box frameworks and to reuse these frameworks through their DSMLs. The F3T is a plug-in for the Eclipse IDE. So developers can make use of the F3T resources, such as domain modeling, framework construction, application modeling through framework DSML and application construction, as well the other resources provided by the IDE. The F3T is composed of three modules, as seen in Figure 1: 1) Domain Module (DM); 2) Framework Module (FM); and 3) Application Module (AM).

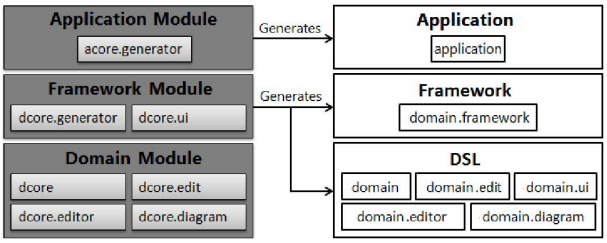


Fig. 1: Modules of the F3T.

A. Domain Module

The DM provides a F3 model editor for developers to define domain features. This module has been developed with the support of the Eclipse Modeling Framework (EMF) and the Graphical Modeling Framework (GMF) [18]. The EMF was used to create a metamodel, in which the elements, relationships and rules of the F3 models were defined as described in the Section III. The metamodel of F3 models is shown in Figure 2. From this metamodel, the EMF generated the source-code of the Model and the Controller layers of the F3 model editor.

GMF has been used to define the graphical notation of the F3 models. This graphical notation also can be seen as the View layer of the F3 model editor. With the GMF, the graphical figures and the menu bar of the editor were defined and linked to the elements and relationships defined in the metamodel of the F3 models. Then, the GMF generates the source-code of the graphical notation. The F3 model editor is shown in Figure 3 with an example of F3 model for the domain of trade and rental transactions.

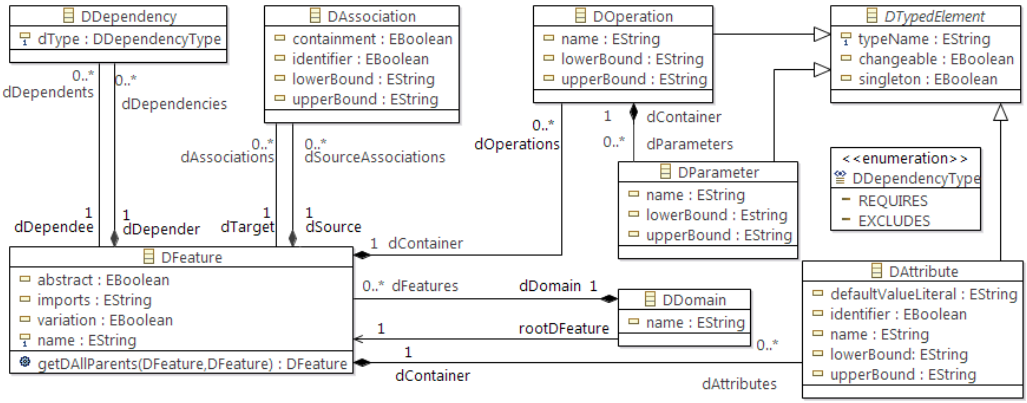


Fig. 2: Metamodel containing elements, relationships and rules of F3 models.

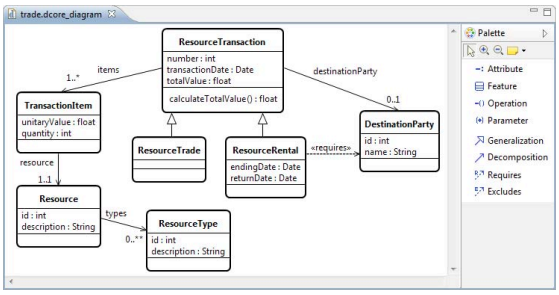


Fig. 3: F3 model for the domain of trade and rental transactions.

B. Framework Module

The FM is a M2T generator that transforms F3 models into framework source-code and DSML. Despite their graphical notation, F3 models actually are XML files. It makes them more accessible to other tools, such as a generator. The FM was developed with the support of Java Emitter Templates (JET) in the Eclipse IDE [26].

JET contains a framework that is a generic generator and a compiler that translate templates into Java files. These templates are XML files, in which tags are instructions to generate an output based on information in the input and text is a fixed content inserted in the output independently of input. The Java files originated from the JET templates reuse the JET framework to compose a domain-specific generator. Thus, the FM depend on the JET plug-in to work.

The templates of the FM are organized in two groups: one related to framework source-code; and another related to framework DSML. Both groups are invoked from the main template of the DM generator. Part of the JET template which generates Java classes in the framework source-code from the features found in the F3 models can be seen as follows:

```
public
 <c:if test="($feature/@abstract)">abstract </c:if>
class <c:get select="$feature/@name"/> extends
 <c:choose select="$feature/@variation">
  <c:when test="'true'">DVariation</c:when>
  <c:otherwise> <c:choose>
    <c:when test="$feature/dSuperFeature">
      <c:get select="$feature/dSuperFeature/@name"/>
    </c:when>
    <c:otherwise>DObject</c:otherwise> </c:choose>
  </c:otherwise>
 </c:choose> { ... }
```

The framework source-code that is generated by the FM is put in a Java project identified by the domain name and the suffix ".framework". Its classes follow the patterns defined by the F3 approach. For example, the FM generates a class for each feature found in a F3 model. These classes contain the attributes and operations defined in its original feature. All generated classes also, directly or indirectly, extend the `DObject` class, which implements non-functional requirements, such as persistence and logging. Generalization relationships result in inheritances and decomposition relationships result in associations between the involving classes. Additional operations are included in framework classes to treat feature variations and constraints of the domains defined in the F3 models. For example, according to the *Variant Decomposition* F3 pattern, the `getResourceTypeClasses` operation was included in the code of the `Resource` class so that the framework can recognize which classes implement the `ResourceType` feature in the applications. Part of the code of the `Resource` class is presented as folows:

```
/** @generated */
public abstract class Resource extends DObject {

  /** @generated */
  private int id;

  /** @generated */
  private Sting name;

  /** @generated */
  private List<ResourceType> types;

  /** @generated */
  public abstract Class<?>[] getResourceTypeClasses();
```

Framework DSML is generated as a EMF/GMF project identified only by the domain name. The FM generates the EMF/GMF models of the DSML, as seen in Figure 4.a, which was generated from the F3 model shown in Figure 3. Then, source-code of the DSML must be generated by using the generator provided by the EMF/GMF in three steps: 1) using the EMF generator from the *genmodel* file (Figure 4.a); 2) using the GMF generator from the *gmfmap* file (Figure 4.b); and 3) using the GMF generator from the *gmfgen* file (Figure 4.c). After this, the DSML will be composed of 5 plug-in projects in the Eclipse IDE. The projects that contain the source-code and the DSML plug-ins of the framework for the trade and rental transactions domain are shown in Figure 4.d.

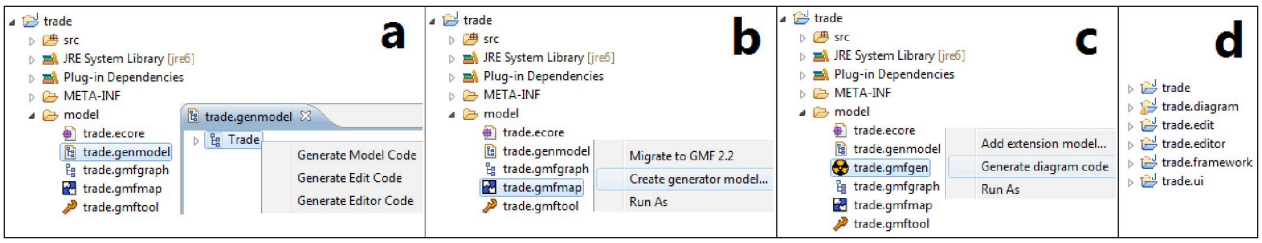


Fig. 4: Generation of the DSML plugins.

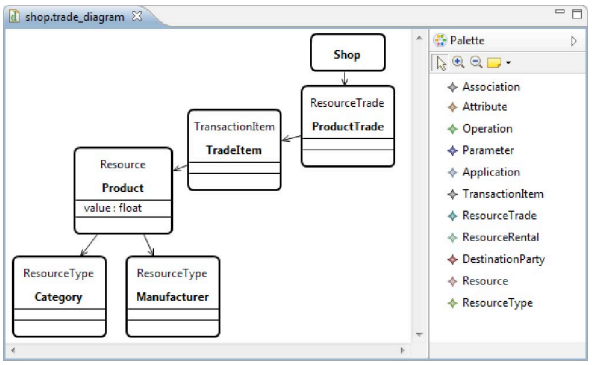


Fig. 5: Application model created with the framework DSML.

C. Application Module

The AM has been also developed with the support of JET. It generates application source-code from an application model based on a framework DSML. The templates of the AM generate classes that extend framework classes and override operations that configure framework hot spots. After the DSML plug-ins are installed in the Eclipse IDE, the AM recognizes the model files created from the DSML. An application model created with the DSML of the framework for the domain of trade and rental transactions is shown in Figure 5.

Application source-code is generated in the source folder of the project where the application model is. The AM generates a class for each feature instantiated in the application model. Since the framework is white box, the application classes extend the framework classes indicated by the stereotypes in the model. It is expected that many class attributes requested by the application requirements have been defined in the domain. Thus, these attributes are in the framework source-code and they must not be defined in the application classes again. Part of the code of the `Product` class is presented as follows:

```
public class Product extends Resource {

  /** @generated */
  private float value;

  /** @generated */
  public Class<?>[] getResourceTypeClasses() {
    return new Class<?>[] {
      Category.class, Manufacturer.class };
  }
```

V. Evaluation

In this section we present an experiment, in which we evaluated the use of the F3T to develop frameworks, since the use of DSMLs to support framework reuse has been evaluated in a previous paper [10]. The experiment was conducted following all steps described by Wohlin et al. (2000) and it can be summarized as: (*i*) **analyse** the F3T, described in Section IV; (*ii*) **for the purpose of** evaluation; (*iii*) **with respect to** time spent and number of problems; (*iv*) **from the point of view of** the developer; and (*v*) **in the context of** MSc and PhD Computer Science students.

A. Planning

The experiment has been planned to answer two research questions:

- **RQ_1: Does the F3T reduce the effort to develop a framework?**
- **RQ_2: Does the F3T result in a outcome framework with a fewer number of problems?**

All subjects had to develop two frameworks, both applying the F3 approach, but one manually and the other with the support of the F3T. The context of our study corresponds to multi-test within object study [27], hence the experiment consisted of experimental tests executed by a group of subjects to study a single tool. In order to answer the first question, we measured the time spent to develop each framework. Then, to answer the second question, we analyzed the frameworks developed by the subjects, then we identified and classified the problems found in the source-code. The planning phase was divided into seven parts, which are described in the next subsections:

1. Context Selection

26 MSc and PhD students of Computer Science have participated in the experiment, which has been made in an off-line situation. All participants had prior experience in software development, Java programming, patterns and framework reuse.

2. Formulation of Hypotheses

The experiment questions have been formalized as follows:

RQ_1, Null hypothesis, H_0: Considering the F3 approach, there is no significant difference, in terms of time, between developing frameworks with the support of F3T and doing it manually. Thus, the F3T does not reduce the time spent to develop frameworks. This hypothesis can be formalized as:

$$\mathbf{H_0}: \mu_{F3T} = \mu_{manual}$$

RQ_1, Alternative hypothesis, H_1: Considering the F3 approach, there is a significant difference, in terms of time, between developing frameworks with the support of F3T and doing it manually. Thus, the F3T reduces the time spent to develop frameworks. This hypothesis can be formalized as:

$$\mathbf{H_1}: \mu_{F3T} \neq \mu_{manual}$$

RQ_2, Null hypothesis, H_0: Considering the F3 approach, there is no significant difference, in terms of problems found in the outcome frameworks, between developing frameworks using the F3T and doing it manually. Thus, the F3T does not reduce the mistakes made by subjects while they are developing frameworks. This hypothesis can be formalized as:

$$\mathbf{H_0}: \mu_{F3T} = \mu_{manual}$$

RQ_2, Alternative hypothesis, H_1: Considering the F3 approach, there is a significant difference, in terms of problems found in the outcome frameworks, between developing frameworks using the F3T and doing it manually. Thus, the F3T reduces the mistakes made by subjects while they are developing frameworks. This hypothesis can be formalized as:

$$\mathbf{H_1}: \mu_{F3T} \neq \mu_{manual}$$

3. Variables Selection

The dependent variables of this experiment were:

- **time spent to develop a framework**;
- **number of problems found in the frameworks**.

The independent variables were as follows:

- **Application**: Each subject had to develop two frameworks: one (Fw1) for the domain of trade and rental transactions and the other (Fw2) for the domain of automatic vehicles. Both Fw1 and Fw2 had 10 features.
- **Development Environment**: Eclipse 4.2.1, Astah Community 6.4, F3T.
- **Technologies**: Java version 6.

4. Selection of Subjects

The subjects has been selected through a non probabilist approach by convenience, so that the probability of all population elements belong to the same sample is unknown.

5. Experiment Design

The subjects were carved up in two blocks of 13 subjects:

- **Block 1**, development of Fw1 manually and development of Fw2 with the support of the F3T;
- **Block 2**, development of Fw2 manually and development of Fw1 with the support of the F3T.

We have chosen use block to reduce the effect of the experience of the students, that was measured through a form in which the students answered about their level of experience in software development. This form was given to the subjects one week before the pilot experiment herein described. The goal of this pilot experiment was to ensure that the experiment environment and materials were adequate and the tasks could be properly executed.

6. Design Types

The design type of this experiment was **one factor with two treatments paired** [27]. The **factor** in this experiment is the manner how the F3 approach was used to develop a framework and the **treatments** are the support of the F3T against the manual development.

7. Instrumentation

All necessary materials to assist the subjects during the execution of this experiment were previously devised. These materials consisted of forms for collecting experiment data, for instance, time spent to develop the frameworks and a list of the problems were found in the outcome frameworks developed by each subject. In the end of the experiment, all subjects received a questionnaire, in which they should report about the F3 approach and the F3T.

B. Operation

The operation phase of the experiment was divided into two parts, Preparation and Execution, as described in the next subsections:

1. Preparation

Firstly, the subjects received a characterization form, containing questions regarding their knowledge about Java programming, Eclipse IDE, patterns and frameworks. Then, the subjects were introduced to the F3 approach and the F3T.

2. Execution

Initially, the subjects signed a consent form and then answered a characterization form. After this, they watched a presentation about frameworks, which included the description of some known examples and their hot spots. The subjects were also trained on how to develop frameworks using the F3 approach with or without the support of the F3T.

Following the training, the pilot experiment was executed. The subjects were split into two groups considering the results of the characterization forms. Subjects were not told about the nature of the experiment, but were verbally instructed on the F3 approach and its tool. The pilot experiment was intended to simulate the real experiments, except that the applications were different, but equivalent. Beforehand, all subjects were given ample time to read about approach and to ask questions on the experimental process. This could affect the experiment validity, then, the data from this activity was only used to balance the groups.

When the subjects understood what their had to do, they received the description of the domains and started timing the development of the frameworks. Each subject had to develop the frameworks applying the F3 approach, i.e., creating its F3 model from a document which describes its domain features and then applying the F3 patterns to implement it.

C. Analysis of Data

This section presents the experimental findings. The analysis is divided into two subsections: (*1*) Descriptive Statistics and (*2*) Hypotheses Testing.

1. Descriptive Statistics

The time spent by each subject to develop a framework and the number of problems found in the outcome frameworks are shown in Table II. From this table, it can be seen that the subjects spent more time to develop the frameworks when they were doing it manually, approximately 72.5% against 27.5%. This result was expected, since the F3T generates framework source-code from F3 models. However, it is worth highlighting that most of the time spent in the manual framework development was due to framework implementation and the effort to fix the problems found in the frameworks, while most of the time spent in the framework development supported by the F3T was due to domain modeling. The dispersion of time spent by the subjects are also represented graphically in a boxplot on left side of Figure 6.

In Table II it is also possible to visualize four types of problems that we analyzed in the outcome frameworks: (*i*) incoherence, (*ii*) structure, (*iii*) bad smells, (*iv*) interface.

The problem of incoherence indicates that, during the experiment, the subjects did not model the domain of the framework as expected. Consequently, the subjects did not develop the frameworks with the correct domain features and

TABLE II: Development timings and number of problems.

constraints (mandatory, optional, and alternative features). As the capacity to model the framework domains depend more on the subject skills than on tool support, incoherence problems could be found in equivalent proportions, approximately 50%, when the framework was developed either manually or with the support of the F3T.

The problem of structure indicates that the subjects did not implement the frameworks properly during the experiment. For example, they implemented classes with no constructor and incorrect relationships or when they forgot to declare the classes as abstract. This kind of problem occurred when the subjects did not properly follow the instructions provided by the F3 patterns. In Table II it can be seen that the F3T helped the subjects to develop frameworks with less structure problems, i.e., 10% in opposition to 90%.

The problem of bad smells indicates design weaknesses that do not affect functionality, but make the frameworks harder to maintain. In the experiment, this kind of problem occurred when the subjects forgot to apply some F3 patterns related to the organization of the framework classes, such as the *Modular Hierarchy* F3 pattern. By observing Table II we can remark that the F3T made a design with higher quality than the manual approach, i.e, 0% against 100%, because the F3T automatically identified which patterns should be applied from the F3 models.

The problem of interface indicates absence of getter/setter operations and the lack of operations that allows the applications to reuse the framework and so on. Usually, this kind of problem is a consequence of problems of structure, hence the number of problems of these two types are quite similar. As it can be observed in Table II that the F3T helped the subjects to design a better framework interface than when they developed the framework manually, i.e., 8.6% against 91.4%.

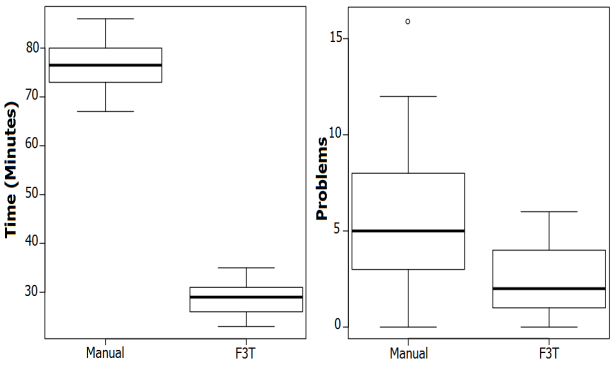


Fig. 6: Dispersion of the total time and number of problems.

In the last two columns of Table II it can be seen that the F3T reduced the total number of problems found in the frameworks developed by the subjects. It is also graphically represented in the boxplot on right side of Figure 6.

2. Testing the Hypotheses

The objective of this section is to verify, based on the data obtained in the experiment, whether it is possible to reject the null hypotheses in favor of the alternative hypotheses. Since some statistical tests are applicable only if the population follows a normal distribution, we applied the Shapiro-Wilk test and created a Q-Q chart to verify whether or not the experiment data departs from linearity before choosing a proper statistical test. The tests has been carried out as follows:

1) **Time**: We have applied the Shapiro-Wilk test on the experiment data that represents the time spent by

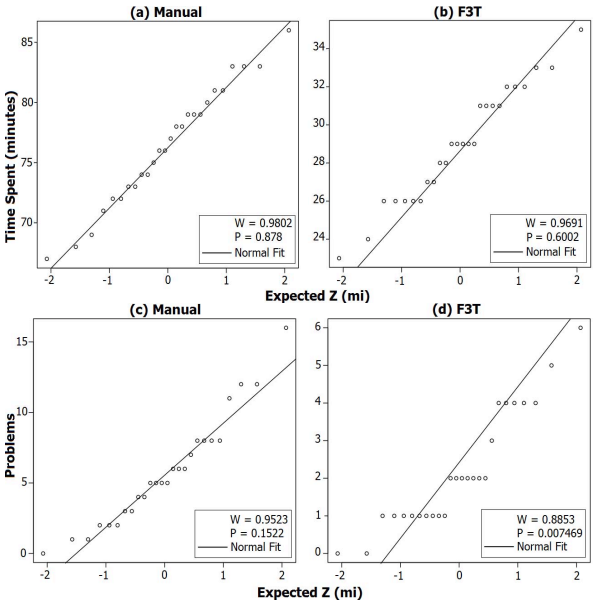


Fig. 7: Normality tests.

each subject to develop a framework manually or using the F3T, as shown in Table II. Considering an α = 0.05, the p-values are 0.878 and 0.6002 and Ws are 0.9802 and 0.9691, respectively, for each approach. The test results confirmed that the experiment data related to the time spent in framework development is normally distributed, as it can be seen in the Q-Q charts (a) and (b) in Figure 7. Thus, we decided to apply the Paired T-Test to these data. Assuming a Paired T-Test, we can reject H_0 if $\mid t_0 \mid > t_{\alpha/2,n-1}$. In this case, $t_{\alpha,f}$ is the upper α percentage point of the t-distribution with f degrees of freedom. Therefore, based on the samples, n = 26 and $d = \{46,42,52,49,41,49,55,50,53,42,42,52,48,43,45, 42,47,48,44,49,51,48,52,51,48,45\}$, S_d = 9.95 and t_0 = 1.6993. The average values of each data set are $\mu_{manual} = 76.42$ and $\mu_{F3T} = 28.96$. So, $d = 76.42 - 28.96 = 47.46$, which implies that $S_d = 3.982$ and $t_0 = 60.7760$. The number of degrees of freedom is $f = n - 1 = 26 - 1 = 25$. We take $\alpha = 0.025$. Thus, according to *StatSoft*[1], it can be seen that $t_{0.025,25} = 2.05954$. Since $\mid t_0 \mid > t_{0.025,25}$ **it is possible to reject the null hypothesis with a two sided test at the 0.025 level**. Therefore, statistically, we can assume that, when the F3 approach is applied, the time needed to develop a framework using F3T is less than doing it manually.

2) **Problems**: Similarly, we have applied the Shapiro-Wilk test on the experiment data shown in the last two columns of Table II, which represent the total number of problems found in the outcome frameworks that were developed whether manually or using the F3T. Considering an α = 0.05, the p-values are 0.1522 and 0.007469, and Ws are 0.9423 and 0.8853, respectively, for each approach. As it can be seen in the Q-Q charts (c) and (d) in Figure 7, the test results confirmed that date related to manual development is normally distributed, but the data related to the F3T can not be considered as normally distributed. Therefore we applied a non-parametric test, the Wilcoxon signed-rank test in these data. The signed rank of these data are S/R of $\mid t_{problems_{manual}} - t_{problems_{F3T}} \mid$ = { +3.5, +7.5, +7.5, +16.5, -3.5, +23, +3.5, +3.5, +10.5, +10.5, +3.5, +18.5, +10.5, +14, +24, +18.5, +3.5, +21, +21, +14, +21, +10.5, +14, +16.5}, S/R stand for "signed rank". As result we got a p-value = 0.001078 with a significance level of 1%. Based on these data, we conclude there is considerable difference between the means of the two treatments. We were able to reject H0 at 1% significance level. The p-value is very close to zero, which further emphasizes that the F3T reduces the number of problems found in the outcome frameworks.

[1]http://www.statsoft.com/textbook/distribution-tables/#t

D. Opinion of the Subjects

We analyzed the opinion of the subjects in order to evaluate the impact of using the F3T. After the experiment operation, all subjects received a questionnaire, in which they could report their perception about applying the F3 approach manually or with the support of the F3T.

The answers in the questionnaire has been analyzed in order to identify the difficulties in the use of the F3 approach and its tool. As it can be seen in Figure 8, when asked if they encountered difficulties in the development of the frameworks by applying the F3 approach manually, approximately 52% of the subjects reported having significant difficulty, 29% mentioned partial difficulty and 19% had no difficulty. In contrast, when asked the same question with respect to the use of the F3T, 73% subjects reported having no difficulty, 16% subjects mentioned partial difficulty and only 11% of all subjects had significant difficulty.

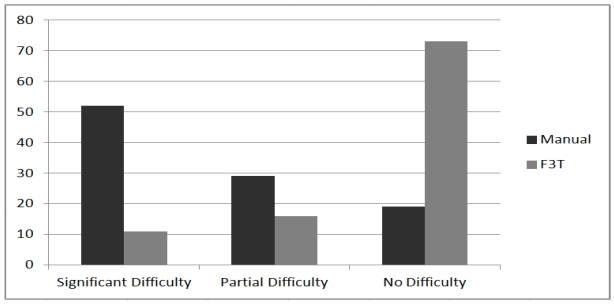


Fig. 8: Level of difficulty of the subjects.

The reduction of the difficulty to develop the frameworks, shown in Figure 8, reveals that the F3T assisted the subjects in this task. The subjects also answered in the questionnaire about the difficulties they found during framework development. The most common difficulties pointed out by the subjects when

they developed the frameworks manually were: 1) too much effort spent on coding; 2) mistakes they made due to lack of attention; 3) lack of experience for developing frameworks; and 4) time spent identifying the F3 patterns in the F3 models. In contrast, the most common difficulties faced by the subjects when they used the F3T were: 1) lack of practice with the tool; and 2) some actions in the tool interface, for instance, opening the F3 model editor, take many steps to be executed. The subjects said that the F3 patterns helped them to identify which structures were necessary to implement the frameworks in the manual development. They also said the F3T automatized the tasks of identifying which F3 patterns should be used and of implementing the framework source-code. Then, they could keep their focus on domain modeling.

E. Threats to Validity

Internal Validity:

- Experience level of the subjects: the subjects had different levels of knowledge and it could affect the collected data. To mitigate this threat, we divided the subjects in two balanced blocks considering their level knowledge and rebalanced the groups considering the preliminary results. Moreover, all subjects had prior experience in application development reusing frameworks, but not for developing frameworks. Thus, the subjects were trained in common framework implementation techniques and how to use the F3 approach and the F3T.
- Productivity under evaluation: there was a possibility that this might influence the experiment results because subjects often tend to think they are being evaluated by experiment results. In order to mitigate this, we explained to the subjects that no one was being evaluated and their participation was considered anonymous.
- Facilities used during the study: different computers and installations could affect the recorded timings. Thus, the subjects used the same hardware configuration and operating system.

Validity by Construction:

- Hypothesis expectations: the subjects already knew the researchers and knew that the F3T was supposed to ease framework development, which reflects one of our hypothesis. These issues could affect the collected data and cause the experiment to be less impartial. In order to keep impartiality, we enforced that the participants had to keep a steady pace during the whole study.

External Validity:

- Interaction between configuration and treatment: it is possible that the exercises performed in the experiment are not accurate for every framework development for real world applications. Only two frameworks were developed and they had the same complexity. To mitigate this threat, the exercises were designed considering framework domains based on the real world.

Conclusion Validity:

- Measure reliability: it refers to metrics used to measuring the development effort. To mitigate this threat, we used only the time spent which was captured in forms fulfilled by the subjects;
- Low statistic power: the ability of a statistic test in reveal reliable data. To mitigate this threat, we applied two tests: T-Tests to statistically analyze the time spent to develop the frameworks and Wilcoxon signed-rank test to statistically analyze the number of problems found in the outcome frameworks.

VI. Related Works

In this section some works related to the F3T and the F3 approach are presented.

Amatriain and Arumi [28] proposed a method for the development of a framework and its DSL through iterative and incremental activities. In this method, the framework has its domain defined from a set of applications and it is implemented by applying a series of refactorings in the source-code of these applications. The advantage of this method is a small initial investment and the reuse of the applications. Although it is not mandatory, the F3 approach can also be applied in iterative and incremental activities, starting from a small domain and then adding features. Applications can also be used to facilitate the identification of the features of the framework domain. However, the advantage of the F3 approach is the fact that the design and the implementation of the frameworks are supported by the F3 patterns and it is automatized by the F3T.

Oliveira et al. [29] presented the ReuseTool, which assists framework reuse by manipulating UML diagrams. The ReuseTool is based in the Reuse Description Language (RDL), a language created by these authors to facilitate the description of framework instantiation processes. Framework hot spots can be registered in the ReuseTool with the use of the RDL. In order to instantiate the framework, application models can be created based on the framework description. Application source-code is generated from these models. Thus, the RDL works as a meta language that registers framework hot spots and the ReuseTool provides a more friendly interface for developers to develop applications reusing the frameworks. In comparison, the F3T supports framework development through domain modeling and application development through framework DSML.

Pure::variants [30] is a tool that supports the development of applications by modeling domain features (Feature Diagram) and the components that implement these features (Family Diagram). Then the applications are developed by selecting a set of features of the domain. Pure::variants generates only application source-code, maintaining all domain artifacts in model-level. Besides, this tool has private license and its free version (Community) has limitations in its functionality. In comparison, the F3T is free, uses only one type of domain model (F3 model) and generates frameworks as domain artifacts. Moreover, the frameworks developed with he support of the F3T can be reused in the development of applications with or without the support of the F3T.

VII. Conclusions

The F3T support framework development and reuse through code generating from models. This tool provides an F3 model editor for developers to define the features of the framework domain. Then, framework source-code and DSML can be generated from the F3 models. Framework DSML can be installed in the F3T to allow developers to model and to generate the source-code of applications that reuses the framework. The F3T is a free software available at: http://www.dc.ufscar.br/∼matheus_viana.

The F3T was created to semi-automatize the applying of the F3 approach. In this approach, domain features are defined in F3 models in order to separate the elements of the framework from the complexities to develop them. F3 models incorporate elements and relationships from feature models and properties and operations from metamodels.

Framework source-code is generated based on patterns that are solutions to design and implement domain features defined in F3 models. A DSML is generated along with the source-code and includes all features of the framework domain and in the models created with it developers can insert application specifications to configure framework hot spots. Thus, the F3T supports both Domain Engineering and Application Engineering, improving their productivity and the quality of the outcome frameworks and applications. The F3T can be used to help the construction of software product lines, providing an environment to model domains and create frameworks to be used as core assets for application development.

The experiment presented in this paper has shown that, besides the gain of efficiency, the F3T reduces the complexities surrounding framework development, because, by using this tool, developers are more concerned about defining framework features in a graphical model. All code units that compose these features, provide flexibility to the framework and allows it to be instantiated in several applications are properly generated by the F3T.

The current version of the F3T generates only the model layer of the frameworks and applications. In future works we intend to include the generation of a complete multi-portable Model-View-Controller architecture.

Acknowledgment

The authors would like to thank CAPES and FAPESP for sponsoring our research.

References

[1] V. Stanojevic, S. Vlajic, M. Milic, and M. Ognjanovic. Guidelines for Framework Development Process. In *7th Central and Eastern European Software Engineering Conference*, pages 1–9, Nov 2011.

[2] M. Abi-Antoun. Making Frameworks Work: a Project Retrospective. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, 2007.

[3] R. E. Johnson. Frameworks = (Components + Patterns). *Communications of ACM*, 40(10):39–42, Oct 1997.

[4] JBoss Community. Hibernate. http://www.hibernate.org, Jan 2013.

[5] Spring Source Community. Spring Framework. http://www.springsource.org/spring-framework, Jan 2013.

[6] S. D. Kim, S. H. Chang, and C. W. Chang. A Systematic Method to Instantiate Core Assets in Product Line Engineering. In *11th Asia-Pacific Conference on Software Engineering*, pages 92–98, Nov 2004.

[7] David M. Weiss and Chi Tau Robert Lai. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[8] D. Parsons, A. Rashid, A. Speck, and A. Telea. A Framework for Object Oriented Frameworks Design. In *Technology of Object-Oriented Languages and Systems*, pages 141–151, Jul 1999.

[9] S. Srinivasan. Design Patterns in Object-Oriented Frameworks. *ACM Computer*, 32(2):24–32, Feb 1999.

[10] M. Viana, R. Penteado, and A. do Prado. Generating Applications: Framework Reuse Supported by Domain-Specific Modeling Languages. In *14th International Conference on Enterprise Information Systems*, Jun 2012.

[11] M. Viana, R. Durelli, R. Penteado, and A. do Prado. F3: From Features to Frameworks. In *15th International Conference on Enterprise Information Systems*, Jul 2013.

[12] Sajjan G. Shiva and Lubna Abou Shala. Software Reuse: Research and Practice. In *Fourth International Conference on Information Technology*, pages 603–609, Apr 2007.

[13] W. Frakes and K. Kang. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, 31(7):529–536, Jul 2005.

[14] M. Fowler. Patterns. *IEEE Software*, 20(2):56–57, 2003.

[15] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science, 7th edition, 2009.

[16] A. Sarasa-Cabezuelo, B. Temprado-Battad, D. Rodrguez-Cerezo, and J. L. Sierra. Building XML-Driven Application Generators with Compiler Construction. *Computer Science and Information Systems*, 9(2):485–504, 2012.

[17] S. Lolong and A.I. Kistijantoro. Domain Specific Language (DSL) Development for Desktop-Based Database Application Generator. In *International Conference on Electrical Engineering and Informatics (ICEEI)*, pages 1–6, Jul 2011.

[18] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009.

[19] I. Liem and Y. Nugroho. An Application Generator Framelet. In *9th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'08)*, pages 794–799, Aug 2008.

[20] J. M. Jezequel. Model-Driven Engineering for Software Product Lines. *ISRN Software Engineering*, 2012, 2012.

[21] K. Lee, K. C. Kang, and J. Lee. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In *7th International Conference on Software Reuse: Methods, Techniques and Tools*, pages 62–77, London, UK, 2002. Springer-Verlag.

[22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA): Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, Nov 1990.

[23] H. Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2004.

[24] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J. DeBaud. PuLSE: a Methodology to Develop Software Product Lines. In *Symposium on Software Reusability*, pages 122–131. ACM, 1999.

[25] OMG. OMG's MetaObject Facility. http://www.omg.org/mof, Jan 2013.

[26] The Eclipse Foundation. Eclipse Modeling Project, Jan 2013.

[27] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[28] X. Amatriain and P. Arumi. Frameworks Generate Domain-Specific Languages: A Case Study in the Multimedia Domain. *IEEE Transactions on Software Engineering*, 37(4):544–558, Jul-Aug 2011.

[29] T. C. Oliveira, P. Alencar, and D. Cowan. Design Patterns in Object-Oriented Frameworks. *ReuseTool: An Extensible Tool Support for Object-Oriented Framework Reuse*, 84(12):2234–2252, Dec 2011.

[30] Pure Systems. Pure::Variants. http://www.pure-systems.com/pure_variants.49.0.html, Fev 2013.

Anexo 4: Comprovante da publicação no *Latin American Workshop on Aspect-Oriented Software Development* (LA-WASP)

- Uma cópia do artigo é apresentado a seguir (a partir da próxima página).

A Combined Approach for Concern Identification in KDM models

Daniel S. M. Santibáñez*, Rafael Serapilha Durelli†, Bruno Marinho* and Valter Vieira de Camargo*
*Departamento de Computação, Universidade Federal de São Carlos,
Caixa Postal 676 – 13.565-905, São Carlos – SP – Brazil
Email: {daniel.santibanez,bruno.santos,valter}@dc.ufscar.br
†Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo,
Av. Trabalhador São Carlense, 400, São Carlos – SP – Brazil
Email: rsdurelli@icmc.usp.br

***Abstract*—The several maintenance tasks a system is submitted during its life usually cause its architecture deviates from the original conceived design, therefore software engineers need techniques for recovering the knowledge embedded in legacy systems in order to get a better software comprehension. With the advent of ADM (Architecture-driven modernization), an OMG standard for modernizing legacy software systems, the reverse engineering process follows a model-driven approach using the KDM metamodel (Knowledge Discovery Metamodel) as the cornerstone of the standard. Nevertheless, although ADM provides the process to modernize legacy systems, it does not support the identification and modularization of crosscutting concerns which is a sort of modernization. In order to overcome this disadvantage it is necessary to extend ADM with new techniques and tools. This paper proposes an approach called CCKDM for identifying crosscutting concerns by means a combination of a concern library and a K-means clustering algorithm. The input of the approach is a KDM model instance and the result is the same KDM model with annotated concerns. To provide some evidence of the precision and recall of our approach we conducted an empirical evaluation that involve two well-known systems. In this evaluation we compared our approach with other ones (XScan and Timna) by using three *levenshtein*. The results that we achieved seen to be similar or equal as compared to those approaches.**

***Keywords*—ADM, KDM, crosscutting concerns, concern mining.**

I. Introduction

Software systems are considered legacy when their maintenance costs are raised to undesirable levels but they are still valuable for organizations. However, they can not be discarded because they incorporate a lot of embodied knowledge due to years of maintenance and this constitutes a significant corporate asset. As these systems still provide significant business value, they must then be modernized/re-engineered so that their maintenance costs can be manageable and they can keep on assisting in the regular daily activities.

Several approaches have been developed to support software engineers in the comprehension of systems where reverse engineering (RE) is one of them [1]. RE supports program comprehension by using techniques that explore the source code in order to find relevant information related to functional and non-functional features [2]. In a parallel research line, researchers have been shifted from the typical RE approach to the so-called Architecture-Driven Modernization (ADM) [3]. ADM has been proposed by OMG (Object Management Group) and advocates conducting RE following the principles of MDA (Model-Driven Architecture) [4], i.e., it treats all the software artifacts involved in the legacy system as models and can establish transformations among them. For instance, firstly a reverse engineering is performed starting from the source code and a model instance (**P**lataform **S**pecific **M**odel - PSM) is created. Next successive refinements are applied to this model up to reach a good abstraction level (PSM) in model called KDM (**K**nowledge **D**iscovery **M**etamodel). Upon this model, several refactorings, optimizations and modifications can be performed to solve problems found in the legacy system. Secondly, a forward engineering is carried out and the source code of the modernized target system is generated again.

According to the OMG the most important artifact provided by ADM is the KDM metamodel, which is a multipurpose standard metamodel that represents all aspects of the existing information technology architectures. The idea behind the standard KDM is that the community starts to create parsers from different languages to KDM, thus, everything that takes KDM as input can be considered platform and language-independent. For example, a refactoring catalogue for KDM can be used for refactoring systems implemented in different languages. The KDM is divided into four layers representing both physical and logical software assets of information systems at several abstraction levels. Each layer is further organized into packages. Each package defines a set of meta-model elements whose purpose is to represent a certain independent facet of knowledge related to existing legacy systems. However, in this paper we are only interested in the Program Elements layer, which is used to represent a language-independent intermediate representation for programming languages.

On the other hand mining of crosscutting concerns or Aspect Mining as is also known is another important research field that has be exploited in the last years and it is totally related to reverse engineering [5] [6]. The main purpose is to be able to automatically locate existing crosscutting concerns in the source code of a system [7]. They are indispensable for modernization processes because most of the legacy systems suffer from the existence of a lot of crosscutting concerns spread over their architecture.

Although ADM/KDM had been created to support

modernization of legacy systems, to the best of our knowledge there is not research that address the mining of crosscutting concerns using the KDM metamodel. We claim that this is indispensable because once the legacy systems are instantiated by the KDM they tend to: (*i*) have complex architectures with several clones spread out throughout the KDM model, (*ii*) involve several kinds of crosscutting concerns, e.g., patterns, architectural styles, business rules and non-functional properties and (*iii*) be very large, making the manual mining impractical. Therefore, identifying those concerns which are scattered and tangled with others concerns, it could add an additional value to the KDM model, and also for helping engineers in the software comprehension to take better decisions in software maintenance activities. Besides, we also argue that creating a mining approach which takes as input the KDM makes this language-independent, considering the existence of parsers that generate KDM instances from several languages [8] [9] [3].

In order to overcome this limitation, in this paper we present a mining approach for crosscutting concerns based on a concern library and string clustering algorithm which uses the standard metamodel KDM (CCKDM). In other words, the input of our technique is a KDM model and the output is the same model but with annotated concerns. Thus, the software engineers may perform refactorings over the KDM model, modularizing the concerns without touching the source code. Furthermore, to evaluate our technique, we applied the approach CCKDM in three well known systems - HealthWatcher v10, PetStore v1.3.2 and PrograpWeb.The aim of the evaluation is to identify the precision and the recall during the mining of crosscutting concern of "Persistence".

II. Background

Knowledge Discovery Metamodel (KDM) is the key within set of standards [10]. KDM allows standardized representation of knowledge extracted from legacy systems by means of reverse engineering. KDM provides a common repository structure that makes possible the exchange of information about existing software assets in legacy systems. This information is currently represented and stored independently by heterogeneous tools focused on different software assets [4, p. 32]. Figure 1 shows each of the varying views of the existing IT architecture represented by the KDM. For example, the build view, depicts system artifacts from a source, executable, and library viewpoint. Other perspectives include design, conceptual, data, and scenario views.

The Level 0 (L0) encompasses the Infrastructure and Program Elements Layer. Infrastructure Layer consists of the Core, kdm, and Source packages which provide a small common core for all other packages. Program Elements Layer consists of the Code and Action packages providing programming elements such as data types, data items, classes, procedures, macros, prototypes, templates and captures the low level behavior elements of applications, including detailed control and data flow between statements. The Level 1 (L1) cover the Resource Layer which represents the operational environment of the existing software system. For example, the knowledge related to events and state-transition, the knowledge related to the user interfaces of the existing software system and the knowledge related to persistent data, such as indexed files, relational databases, and other kinds of data storage. The

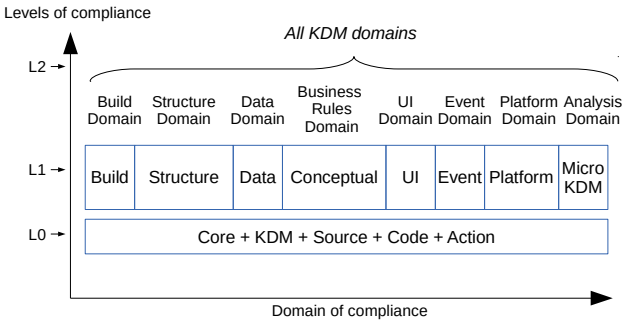


Figure 1: KDM domains of artifact representation (Adapted from Ulrich and Newcomb [4])

Level 2 (L2) cover the Abstraction Layer which represents domain and application abstractions.

As we stated earlier, herein we are only interested in the Program Element Layer - more specifically in the Code Package, which represents the code elements of a program and their associations. Therefore, it is important to dig a little deeper in this metamodel because it is mainly used by our approach in order to identify concerns.

In a given KDM instance, each instance of the code meta-model element represents some programming language construct, determined by the programming language of the existing software system. Each instance of a code meta-model element corresponds to a certain region of the source code in one of the artifacts of the existing software system. In addition, the Code package consists of 24 classes and contains all the abstract elements for modeling the static structure of the source code. However, we are particularly interested in *MethodUnit* and *StorableUnit* packages because they implement crosscutting concerns. In Figure 2 is depicted a chunk of the Code package. It worth to notice that the more important metaclasses used herein are highlighted.

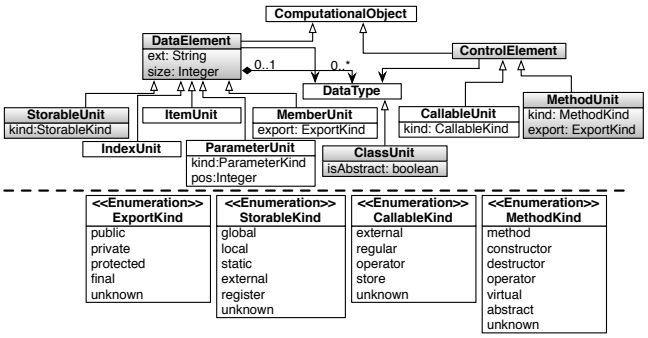


Figure 2: Chunk of the Code Package (OMG Group [11])

As can be seen in Figure 2 the root metaclass is *ComputationalObject* which has two sub-metaclasses, i.e., *DataElement* and *ControlElement*. The former sub-metaclass, *DataElement*, is a generic modeling element that defines the common properties of several concrete classes that represent the named data items of existing software systems, for example, global and lo-

cal variables, record files, and formal parameters. *DataElement* has five sub-metaclasses - *StorableUnit*, *IndexUnit*, *ItemUnit*, *ParameterUnit* and *MemberUnit*. *StorableUnit* is a concrete sub-metaclass of the *StorableElement* meta-class that represents variables of the existing software system. *IndexUnit* class is a concrete subclass of the *DataElement* class that represents an index of an array datatype. Instances of *ItemUnit* class are endpoints of KDM data relations which describes access to complex datatypes. *ParameterUnit* class is a concrete subclass of the *DataElement* class that represents a formal parameter; for example, a formal parameter of a procedure. *MemberUnit* class is a concrete subclass of the *DataElement* class that represents a member of a class type. Finally, the latter, *ControlElement* is a sub-metaclass that contains two sub-metaclasses - *MethodUnit* and *CallableUnit*. *MethodUnit* element represents member functions owned by a *ClassUnit*, including user-defined operators, constructors and destructors. The *CallableUnit* represents a basic stand-alone element that can be called, such as a procedure or a function. As can be seen below the dashed line in Figure 2 there are also the following enumerations: "*ExportKind*", "*StorableKind*", "*CallableKind*", "*MethodKind*", which are sets os literals used as properties of the metaclasses.

Our approach uses those meta-classes to identify the crosscutting concerns rather than using source code. More information about the approach can be seen in Section III.

III. Concern Identification

The developed technique called CCKDM aims to identify code structures into KDM models which may implement crosscutting concerns. The output is an annotated KDM with concerns. Our technique could be classified as a token-based approach, that means analysis of sequences of characters [7].

The Figure 3 depicts the overall process, which is divided into four sub-processes denoted by its corresponding letters at the left side.

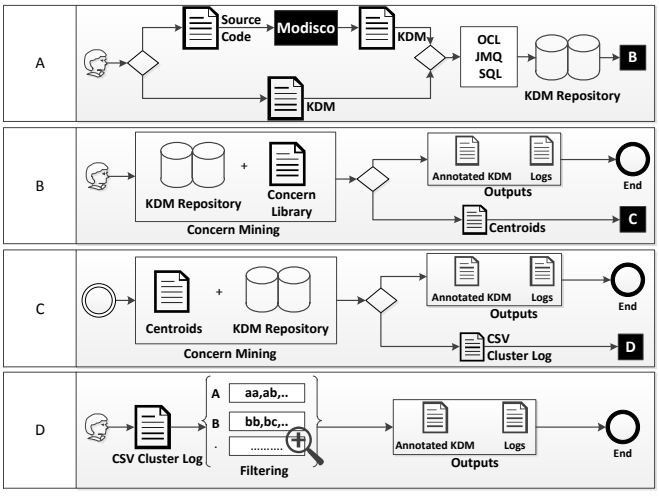


Figure 3: The CCKDM mining process

A. Recovery of Code Structures

The subprocess [A] of the technique starts with user intervention which provide the source code or a KDM file as input to the technique. If the source code is the input, then it is converted to a KDM model by means *Modisco*^{TM} [1] which makes available some APIs called discoverers to convert the source code of a system into a KDM model. After that, the KDM model is queried to recover code structures of the application under study such as methods and properties because they are the most suitable items to find concern seeds [12]. To do that, a combination of three technologies are used to achieve this purpose; a set of simple OCL (Object Constraint Language) queries to recuperate all packages, all classes, all methods and all properties of the model. Then, using the Java Model Query (JMQ) of *Modisco*^{TM} programmatically are retrieved method calls, methods containers, container classes, method signatures, method types and property types. The last step is to persist these elements (strings) into a KDM repository which is a relational database. Figure 4 shows the EER (Enhanced Entity Relationship) model which is composed by eight tables. Table *Class* persists class names. Table *Method* persists method names. Table *Import* persists application imports. Table *Package* persists application packages. Table *GlobalProperty* persists class properties. Table *MethodProperty* persists variables belonging to a certain method. Table *Class_has_Import* persists the relation import/class and table *Method_has_Method* persists method calls.

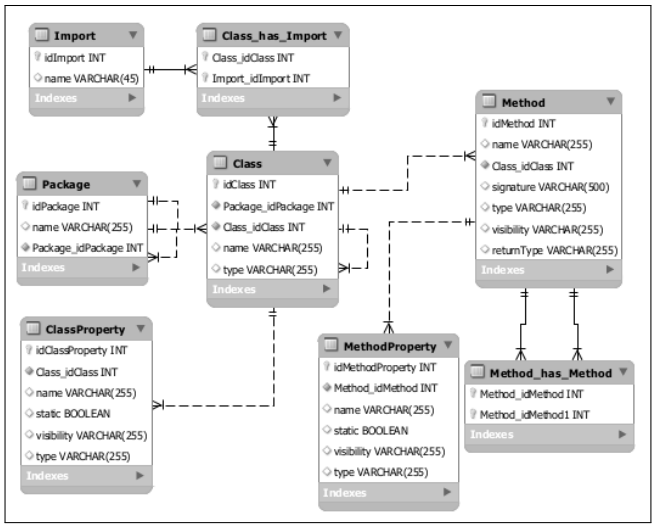


Figure 4: EER model

B. Concern Identification by Concern Library

The subprocess [B] is triggered by user. SQL queries are performed over the KDM repository with information of our concern library which is an XML file containing concern definitions based on [13]. A concern definition is an XML entry composed by a concern name and one or more well-known names of classes implementing the concern, as shown

[1]Modisco provides an extensible framework to develop model-driven tools to support use-cases of existing software modernization, http://www.eclipse.org/MoDisco/.

in Figure 5. Specifically, we are interested in identify possible APIs used by the system because they could be used to implement several concerns. The result is a set of elements (strings) related with some of the classes of the concern library. These strings will be the initial seeds of our approach and the initial information (centroids) for our string clustering approach detailed in the next section. If user does not want to perform the subprocess [C] then the mining process stop and the outputs are log files and the annotated KDM with the identified concern names.

```
▼<ConcernLibrary>
  ▼<Concern name="Persistence">
    ▶<Package name="java.sql">...</Package>
    </Concern>
  ▶<Concern name="Logging">...</Concern>
  ▼<Concern name="Authentication">
    ▼<Package name="javax.security.auth">
       <Element>Destroyable</Element>
       <Element>Refreshable</Element>
       <Element>AuthPermission</Element>
       <Element>Policy</Element>
       <Element>PrivateCredentialPermission</Element>
       <Element>Subject</Element>
       <Element>SubjectDomainCombiner</Element>
       <Element>DestroyFailedException</Element>
       <Element>RefreshFailedException</Element>
     </Package>
    ▼<Package name="java.security">
       <Element>Principal</Element>
     </Package>
    ▼<Package name="javax.ejb">
       <Element>SessionBean</Element>
       <Element>SessionContext</Element>
     </Package>
   </Concern>
 </ConcernLibrary>
```

Figure 5: Concern Library

C. Concern Identification by Clustering

The subprocess [C] performs a string clustering by means a K-means algorithm [14]. The main idea behind this is that "programmers tend to use similar variable names to implement logic programming that meets the same objectives where the contexts are different". In that sense, this subprocess complements the previous one because its objective is to identify names code structures which were not previously identified but have similar names with the strings identified in subprocess B. Thus, we could suppose they implement same concerns. For example, when we applied our approach to HealthWatcher, it identified variables declared by *PersistenceMechanismException* and *PersistenceMechanism* which are not part of Java API.

The strings identified in subprocess [B], which already belong to a certain concern, are the K centroids for our cluster algorithm (*i.e.* the strings where others strings will be clustered). Then, these centroids are compared with method names and property names of the KDM repository by using the *Levenshtein* distance which is a string metric for measuring the difference between two sequences [15]. If the *Levenshtein* resulting value is closer to 1.0 then the compared strings are more similar, on the other hand if it is more closer to 0.0 then the compared strings are more dissimilar. Users can determine the *Levenshtein* distance threshold if they want a more flexible or more restrictive string clustering. Finally, the new identified strings are annotated with their respective concern into the KDM model. The outputs are log files and the annotated KDM.

D. Manual Filtering

The subprocess [D] is optional and gives the possibility to users performing manual filtering of elements identified by the subprocess [C] by means a CSV file generated in the previous subprocess containing all the strings identified by our cluster algorithm with their respective concern name. Users must tag with a "X" at the end of the line in the CSV file if it still be annotated into the KDM file. For all the strings without the "X" tag in the CSV file, the annotation concern into the KDM file will be remove. The outputs are the updated KDM file and log files.

E. Annotating the concerns into the KDM model

The technique performs the annotation activity by introducing a new attribute to the KDM meta-model called *concern*. Thus, methods and properties are annotated with their respective concerns in the following form: *concern="CONCERN"*. Figure 6 shows an annotated example of part of a KDM file for Persistence and Logging. Of course, the new attribute added to the model does not belong to the KDM meta-model so we extended it in order to work with this new feature.

```
    </codeElement>
    <codeElement concern="Persistence" xsi:type="code:
      <attribute tag="export" value="private"/>
      <source language="java">
        <region file="/0/@model.2/@inventoryElement.9"
      </source>
      <comment text="/** &#x0A; */&#x0A;"/>
    </codeElement>
    <codeElement xsi:type="code:StorableUnit" name="me
      <attribute tag="export" value="private"/>
      <source language="java">
        <region file="/0/@model.2/@inventoryElement.9"
      </source>
    </codeElement>
    <codeElement concern="Persistence" xsi:type="code:
model.0/@codeElement.0/@codeElement.0/@codeElement.2/@
      <attribute tag="export" value="public"/>
      <source language="java">
        <region file="/0/@model.2/@inventoryElement.9"
      </source>
    </codeElement>
</codeElement>
<codeElement xsi:type="code:Package" name="model">
  <codeElement xsi:type="code:ClassUnit" name="User" i
    <attribute tag="export" value="public"/>
    <source language="java">
      <region file="/0/@model.2/@inventoryElement.10"
    </source>
    </codeElement>
    <codeElement xsi:type="code:StorableUnit" name="id
      <attribute tag="export" value="private"/>
      <source language="java">
        <region file="/0/@model.2/@inventoryElement.10
      </source>
    </codeElement>
    <codeElement concern="Logging" xsi:type="code:Stor
```

Figure 6: Annotated concerns into KDM

However, we argue this is neither a problem nor a limitation of our approach once according to the KDM's specification it is quite easy to extend it by using the light-weight extension mechanism. The KDM light-weight extension mechanism is a

standard way of adding new "virtual" meta-model elements to KDM [11].

IV. Experimental Study

The goal of our experimental study was to evaluate the effectiveness of our combined technique. Unlike other empirical case studies of concern mining techniques, which describe in detail the kinds of candidates discovered by their tools, we focus here on the precision and recall. We compared empirically CCKDM using 3 *levenshtein* values with *XScan* [16] and *Timna* [17]. The *levenshtein* values were chosen after an statistical analysis which was not included int this work due space limitation reasons.

XScan identifies code units named concern peers and they are detected based on their similar interactions, i.e., similar calling relations in similar contexts, either internally or externally. *Timna* is a framework in which new and existing mining analyses can be easily added and included in the decision as to whether a code segment is a seed of a scattered concern. The main difference is that our technique use a KDM model as input instead of the source code.

A. Subject Programs

In Table II we summarize the applications software under study. All the listed software were used to perform a comparative analysis in terms of precision and recall with others concern mining tools. As we can see, they have a reasonable size (KLOC $\leq$ 9K) which make them suitable to perform a manual analysis of concerns to calculate the recall metric.

Table II: Applications software under study

	System	LOC	Classes	Methods	Properties
1	HealthWatcher v10	8K	118	894	1290
2	PetStore v1.3.2	9K	228	1917	3002

The HealthWatcher is a real health complaint system developed to improve the quality of the services provided by health care institutions. The system has a web-based user interface for registering complaints and performing several other associated operations. PetStore is an application that allows customers to purchase goods via a browser.

B. Empirical Evaluation

In Table I we show the comparison of our technique with *XScan* and *Timna* in terms of precision and recall for Persistence. *XScan* just presents the recall value for HealthWatcher and *Timna* just presents the precision value for PetStore.

Our technique with a *levenshtein* value of 0.3 obtained 100% of precision and recall when we analyzed HealthWatcher. Which means, there are no false negatives and false positives. For *levenshtein* values of 0.3 and 0.4 precision value still remains but the recall decrease. That is normal because higher values of *levenshtein* implies that strings must match more precisely and as a consequence generating false negatives. The scenario changes for PetStore where while the *levenshtein* value increase, the precision increase and the recall decrease. That is because on one hand false negatives increase and on the other hand false positives decrease.

The recall is sensitive to false negatives and on the other hand, precision is sensitive to false positives. If the levenshtein value tends to be high then false positives could increase and if the *levenshtein* value tends to be low then false negatives could increase.

If we compare precision and recall for CCKDM using the three *levenshtein* values with *XScan* and *Timna*, we can see it reaches similar or equal values than the other techniques, so it is possible to say CCKDM has the same effectiveness.

C. Threats to Validity

The lack of representativeness of the subject programs may pose a threat to external validity. We argue that this is a problem that all software engineering research, since we have theory to tell us how to form a representative sample of software. Apart from not being of industrial significance, another potential threat to the external validity is that the investigated programs do not differ considerably in size and complexity. To partially ameliorate that potential threat, the subjects were chosen to cover a broad class of applications. Also, this experiment is intended to give some evidence of the efficiency and applicability of our implementation solely in academic settings. A threat to construct validity stems from possible faults in the implementations of the techniques. With regard to our mining techniques, we mitigated this threat by running a carefully designed test set against several small example programs. Similarly, XSCan and Timna have been extensively used within academic circles, so we conjecture that this threat can be ruled out.

V. Related Work

Concern mining or aspect recommendation has been a popular research topic in recent years. Static mining and history-based mining are two major techniques based on source code analysis. The static technique analyzes source code of a version of software to extract seeds of concerns. A Fan-in value, which is the number of unique callers of each method/function, was first introduced by Marin and others [18] and further generalized by Zhang and others [19] to propose Clustering-Based Fan-in Analysis (CBFA). The history-based mining technique was first adopted by Breu and others [20], who proposed History-based Aspect Mining (HAM). HAM clusters methods/functions that add or remove a call to the same method/function, and groups together methods/functions that are called by the same cluster as concern seeds.

Lengyel et al. [21] proposes a semi-automatic approach to identify crosscutting constraints. The approach uses several algorithms to support the detection of the crosscutting constraints in metamodel-based model transformations. The input of the approach is a transformation (transformation rules and a control flow model), and the expected output is the list of the crosscutting constraints separated as aspects. Van Gorp et al. [22] proposed a UML profile to express pre and post-conditions of source-code refactorings using Object Constraint Language (OCL) constraints. The proposed profile allows that a CASE tool: (*i*) verify pre and post-conditions for the composition of sequences of refactorings; and (*ii*) use the OCL consulting mechanism to detect bad smells such as crosscutting concerns.

Table I: Comparison values of precision and recall for persistence

	CCKDM-0.3		CCKDM-0.4		CCKDM-0.5		XScan		Timna	
Systems	P	C	P	C	P	C	P	C	P	C
HealthWatcher v10	100%	100%	100%	80%	100%	76,11%	-	100%	-	-
PetStore v1.3.2	95%	100%	95,73%	84,15%	98,79%	75,44%	-	-	93.80%	-

The differential of our approach described herein in relation to the others is that our approach mines crosscutting concerns by using KDM instead of another models or source code. It is important to note that to the best of our knowledge there is no previous research that addresses mining crosscutting concerns by using KDM model as input.

VI. Conclusions

We presented a new mining approach for crosscutting concerns called CCKDM which aims to identify and tag crosscutting concerns into KDM models. It is important to note this is the first work in concern mining area that use a standardized model in the context of ADM to perform search of concerns and we believe that ADM standards will be widely used in a near future because is an OMG initiative.

We also argue, although there are a number of other tools based on more sophisticated techniques than string analysis, our combined technique obtained reasonable values in terms of precision and recall according to our empirical evaluation. In the future we plan to improve our mining technique to identify other type of concerns such as architectural and business patterns in KDM models. Thus, it could be interesting determine possible relations between concerns belonging to high layer levels with concerns belonging to low layer levels for better comprehension of software systems.

Acknowledgments

Daniel Santibáñez would like to thank the financial support provide by CAPES. Rafael Durelli would like to thank the financial support provided by FAPESP, 2012/05168-4. Valter Camargo would like to thank FAPESP, 2012/00494-0.

References

[1] G. Canfora, M. Di Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Commun. ACM*, vol. 54, pp. 142–151, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1924421.1924451

[2] E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, 1990.

[3] L. Mainetti, R. Paiano, and A. Pandurino, "Migros: A model-driven transformation approach of the user experience of legacy applications," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7387 LNCS, pp. 490–493, 2012, cited By (since 1996) 0.

[4] W. M. Ulrich and P. Newcomb, *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[5] M. von Detten, M. Meyer, and D. Travkin, "Reverse engineering with the reclipse tool suite," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 299–300. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810360

[6] M. Marin, A. V. Deursen, and L. Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, pp. 3:1–3:37, December 2007. [Online]. Available: http://doi.acm.org/10.1145/1314493.1314496

[7] R. Durelli, D. M. Santibáñez, N. Anquetil, M. E. Delamaro, and V. V. Camargo, "A Systematic Review on Mining Techniques for Crosscutting Concerns." Coimbra, Portugal: ACM SAC, 2013.

[8] G. Deltombe, O. L. Goaer, and F. Barbier, "Bridging kdm and astm for model-driven software modernization." in *SEKE*. Knowledge Systems Institute Graduate School, 2012, pp. 517–524.

[9] R. PÃ©rez-Castillo, I. De GuzmÃ¡n, D. Caivano, and M. Piattini, "Database schema elicitation to modernize relational databases," vol. 1 DISI, no. AIDSS/-, 2012, pp. 126–132, cited By (since 1996) 0.

[10] R. Perez-Castillo, I. G.-R. de Guzman, O. Avila-Garcia, and M. Piattini, "On the use of adm to contextualize data on legacy source code for software modernization," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 128–132. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2009.20

[11] OMG, "Object Management Group (OMG) Architecture-Driven Modernisation," Disponível em: *http://www.omgwiki.org/admtf/doku.php?id=start*, 2012, (Acessado 2 de Agosto de 2012).

[12] K. Mens, A. Kellens, and J. Krinke, "Pitfalls in aspect mining," in *Proceedings of the 2008 15th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 113–122.

[13] P. Parreira Junior, W. Mendes, V. de Camargo, R. Penteado, and H. Costa, "Mining crosscutting concerns with comscid: A rule-based customizable mining tool," in *Informatica (CLEI), 2012 XXXVIII Conferencia Latinoamericana En*, 2012, pp. 1–9.

[14] J. Han, *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[15] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[16] T. T. Nguyen, H. V. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Aspect recommendation for evolving software," in *Proceeding of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011, pp. 361–370.

[17] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll, "Timna: a framework for automatically combining aspect mining analyses," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005, pp. 184–193.

[18] M. Marin, A. V. Deursen, and L. Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, pp. 1–37, 2007.

[19] Z. Danfeng, G. Yao, and C. Xiangqun, "Automated aspect recommendation through clustering-based fan-in analysis," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 278–287.

[20] S. Breu and T. Zimmermann, "Mining aspects from version history," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006.

[21] L. Lengyel, T. Levendovszky, and L. Angyal, "Identification of crosscutting constraints in metamodel-based model transformations," in *EUROCON 2009, EUROCON '09. IEEE*, 2009, pp. 359–364.

[22] P. Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards automating source-consistent uml refactorings," 2003.

Anexo 5: Comprovante da publicação no *The 14th International Conference on Computational Science and Applications* (ICCSA 2014)

- Uma cópia do artigo é apresentado a seguir (a partir da próxima página).

Data Network in Development of 3D Collaborative Virtual Environments: A Systematic Review

Diego Roberto Colombo Dias[1], Rafael Serapilha Durelli[1], José Remo Ferreira Brega[2], Bruno Barberi Gnecco[3], Luis Carlos Trevelin[1], and Marcelo de Paiva Guimarães[4]

[1] Computer Science Department, Federal University of São Carlos, São Carlos, SP, Brazil
[2] Computer Science Department, UNESP, Bauru, SP, Brazil
[3] Corollarium Technologies, São Paulo, SP, Brazil
[4] Open University of Brazil, Federal University of São Paulo/Faccamp's Master Program, São Paulo, SP, Brazil
{diegocolombo.dias,rafael_durelli,trevelin}@dc.ufscar.br, remo@fc.unesp.br, brunobg@corollarium.com, marcelodepaiva@gmail.com

Abstract. Background: 3D Collaborative Virtual Environments (3DCVE) have been used to allow multiple geographically distant users to share virtual reality environment. This remote user collaboration requires to deal with network problems. Objective: The objective of this systematic review is twofold: (1) to identify possible solutions to network issues, especially those related to the Internet, such as jitter, latency and packet loss; and (2) to identify the protocols and network topologies commonly used in 3DCVE. Results: We selected 132 papers from the most commonly used search portals, i.e., IEEE, ACM, Scopus, Springer, Science Direct and Web of Science. We also describe the studies conducted over the past 10 years, highlighting the network protocols and topologies, which are commonly used. Finally, we suggest a framework architecture for 3DCVE based on graphic cluster.

1 Introduction

The main feature of 3DCVE is the simulation of immersive and interactive 3D virtual environments, such as serious and multiplayer games. 3DCVE allow multiple users to interact with each other in real time even when they are located in different places. Other features of 3DCVE are the sharing of space presence and time. According to Singhal and Zyda [1], 3DCVE consist of four basic components: graphics engines and display; communication and control devices; processing system; and data network. This review covers the primary studies related to traffic in data networks.

The development of 3DCVE requires knowledge in several areas, such as the design and implementation of network protocols, parallel and distributed systems, computer graphics, multithreaded systems and user interface. Several problems must be resolved regarding the design of 3DCVE, however, most of them related to the network, i.e. consistent management of distributed information, real-time interaction and adaptation of applications to limited network bandwidth. 3DCVE have been

B. Murgante et al. (Eds.): ICCSA 2014, Part I, LNCS 8579, pp. 769–785, 2014.

researched since the 1980s and attracted particular attention in the 1990s thanks to the evolution of data networks. In this study, we present publications covering the last 10 years of research and applications, a period in which data networks have experienced a remarkable evolution.

In recent years, research has been undertaken to minimize the adverse issues of different types of network, particularly the protocols supported by the Internet. This study presents a systematic review of 3DCVE and describes the implementation challenges. The primary studies reviewed summarize some of the problems and possible solutions for treating jitter, packet delivery delay, packet loss, etc. We also reviewed studies that addressed solutions based on network topologies, protocols and implementation solutions, such as buffering and data prediction. This study is not a survey, but a general overview of what has been achieved in the 3DCVE research field in recent years.

With the results obtained from the systematic review, it was possible to characterize values for common problems encountered in the development of 3DCVE. Thus, it was also possible to devise and implement a framework to support the development of 3DCVE in relation to the quality of service connection between distributed graphics clusters.

Section 2 presents the study, planning, execution, reporting and validation of the systematic review. Section 3 presents the architecture of a management connection framework based on graphics clusters, focusing in the results listed in Section 2. Section 4 concludes.

2 Systematic Review

This study has been undertaken as a systematic review based on the guidelines proposed by Kitchenham and Brereton [2]. According to them, three main phases are involved: (1) planning the review, (2) conducting the review and (3) reporting the review. We also used a visual text mining (VTM) technique to support study selection. VTM uses text mining algorithms and methods combined with interactive visualizations. The following sections present details of how each phase was carried out.

2.1 Planning the Systematic Review

In this phase, we defined the review protocol. This protocol contains: (i) the research questions, (ii) the search strategy, (iii) the inclusion and exclusion criteria and (iv) the data extraction and synthesis method. Research questions must embody the review study purpose. Moreover, these questions reflect the general scope of the review study. The scope is comprised of population (i.e., population group observed by the intervention), intervention (i.e., what is going to be observed in the context of the planned review study), and outcomes of relevance (i.e., the results of the intervention). Furthermore, during conduction of this step, it was also necessary to establish the scope of the review study. According to the systematic review process [2], the scope has to be established using the PICO criteria, it is a method of

combining a search strategy that allows a more evidence based approach to literature searching (Population, Intervention, Comparison and Outcome).

As described before, the objective of this review is twofold: (i) to find out what are the main issues that someone has to deal during the devising and implementation of a 3DCVE; and (ii) as for the problems ascertaining what are the solutions that were applied to solve them. In order to achieve such objectives we worked out three Research Questions (RQ). The questions are:

RQ1: Which communication protocols are utilized to accomplish the interconnection among 3DCVE?

RQ2: Which network topologies are employed to establish interconnection among 3DCVE?

RQ3: What are the challenges of implementing such systems?

A search string was constructed using boolean operators, i.e., AND and OR. The search encompassed electronic databases, which are deemed as the most relevant scientific sources and therefore likely to contain important primary studies [3]. We applied the search string to the following electronic databases: IEEE, ACM, Scopus, Springer, Science Direct and Web of Science.

Then, in order to determine which primary studies were relevant to our research questions, we applied a set of inclusion and exclusion criteria. Inclusion criteria were as follows:

1) Primary studies presenting at least one solution for 3DCVE: such solutions assisted the implementation of 3DCVE in various research fields, i.e., network protocols and topologies.

2) Primary studies presenting at least one type of evaluation for verifying the performance and efficiency of the 3DCVE: without the results of the evaluation it would not possible to make comparisons between different solutions.

Studies had to meet at least criterion (1). If all criteria were mandatory, the number of eligible primary studies would have dropped significantly.

Exclusion criteria were as follows:

1) Primary studies not evincing any innovation in terms of 3DCVE: primary studies that comprised only a case study and did not provide important details about the architecture and development process.

2) Primary studies comprising short papers: papers with two pages or fewer were not considered herein, since we assumed that this kind of study would not offer sufficient information.

During the extraction process, the data of each primary study were independently gathered by five reviewers. The review was started in December 2012 by two Ph.D. students and three post-doctoral researchers in software engineering and distributed systems; the achieved results were crossed and then validated.

2.2 Conducting the Systematic Review

In this phase, we first identified primary studies in the digital libraries. IEEE returned more primary studies than the others (449), i.e., ACM, Scopus, Springer and Science Direct returned 181, 212, 394 and 267 studies, respectively. The digital library that

returned the fewest primary studies was Web of Science, i.e., 15 papers. In sum, we obtained 1518 primary studies. Subsequently we selected primary studies by reading the titles and abstracts and applied the inclusion and exclusion criteria. As a result, we obtained 451 primary studies that were read in their entirety, and the final total was 132 studies.

As regards the network infrastructure used, we found studies using the Internet and gigabit networks (local and regional). Generally, for applications running on gigabit networks, we did not identify the same problems found in applications running on the Internet, because they are high-speed and reliable networks. In this kind of network, the protocol normally used is Transmission Control Protocol (TCP) [4], [5], [6], [7], [8], [9], [10], [11].

2.3 Validation

In the validation phase an approach that uses the visual text mining (VTM) technique and an associated tool - Projection Explorer (PEx) - were applied to support the inclusion and exclusion decisions [12]. It was found a great amount of papers and some of them have no relevant information to the research, so the PEx was used to assist the primary classification process.

Figure 1 presents a document map generated from PEx. This map is composed of all primary studies analyzed in the review, highlighting them in different shades of gray to differentiate in which of the stages a study was removed from the review. White points are the papers excluded in the first stage, gray points are the papers excluded in second stage and the black points are the included papers.

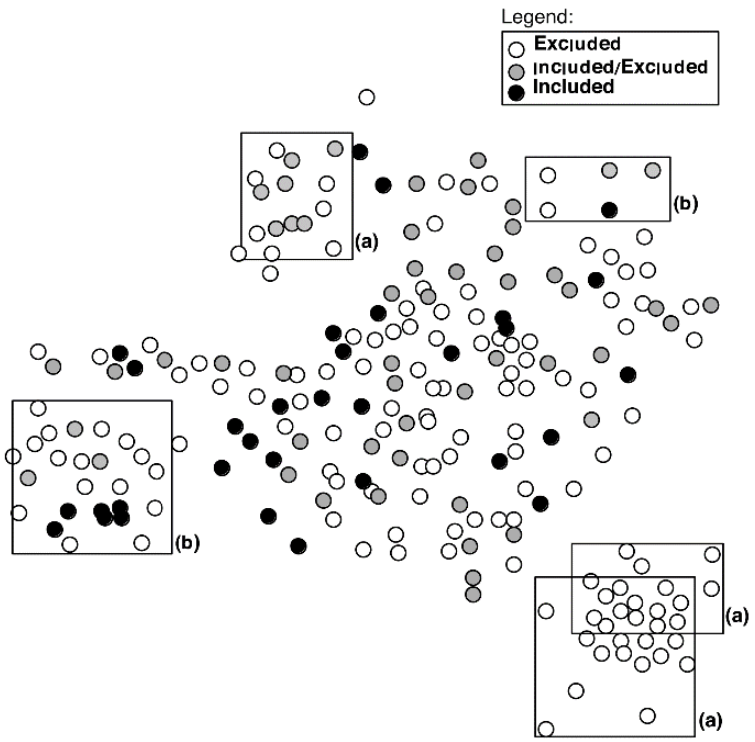


Fig. 1. Document map colored to show the history of the study inclusions and exclusions.

The exploration of a document map is conducted in two steps: (1) a clustering algorithm is applied to the document map, creating groups of strongly related documents; (2) the resulting clusters are analyzed in terms of pure clusters - all documents belonging to a cluster have the same classification (all included or excluded, regardless of exclusion stage) - and mixed clusters - which represent documents with different classifications in the same cluster. These cases are hints to the reviewer, and the studies grouped should be reviewed following the traditional method. In order to facilitate visualization, in Figure 1 just five clusters generated by PEx are depicted. Examples of pure clusters (all excluded) are identified in Figure 1 by label "(a)" and therefore did not need to be reviewed. Mixed clusters (clusters containing black (included) and white or gray (excluded) studies) are identified by label "(b)" and they were reviewed by the authors of this paper. In the end, we kept the initial classifications, which had been conducted manually.

2.4 Reporting the Systematic Review

This section presents an overview of the researches conducted in the last 10 years concerning 3DCVE, especially covering use of network protocols and topologies. Furthermore, the information collected from primary studies was used to answer the research questions.

Most primary studies selected were published on the IEEE website (48 studies). The other studies were selected from ACM, Scopus, Science Direct and Springer, numbering 34, 23, 19 and 8, respectively. We selected primary studies published in conferences, workshops and journals. Most studies were on conferences (75) followed by journals (30), books (15) and workshops (12).

Results presented in primary studies were classified as experiments, case studies or unspecified. In order to answer the research questions we analyzed 132 papers individually. Some primary studies presented experiments that gave information about traffic rate, latency, package loss and comparisons between different protocols [10], [13]. Some contained case studies that presented examples of the use of 3DCVE: medicine [14], [15], [16], [17], engineering [18], [19] and education [20], [21]. Not all papers are referenced in this paper because of space limitations, but the complete research can be visualized in: omitted due to blind review.

The frequency of publications that uses different protocols for developing 3DCVE is presented in a bubble plot in Figure 2. The bubble plot consists of two axes (X and Y), in which bubbles represent categories. The bubble size determines the number of primary studies that were classified as belonging to a category (protocols). This summary provides a visual overview that helps to identify which categories have been emphasized in recent research, with gaps and opportunities for future research. Figure 2 has two coordinates, one representing the year of publication and other the protocols, showing the most used protocols in the implementation of 3DCVE, and answering RQ1.

The protocols found in the studies were: TCP, User Datagram Protocol (UDP), AppTraNet, Stream Control Transmission Protocol (SCTP), Real-time Transport

Protocol (RTP/RTCP), Real-Time Events Protocol (RTEP), Pragmatic General Multicast (PGM), Interactive Stream Transport Protocol (ISTP) and Quanta Framework.

The protocol most frequently used was UDP, found in eight primary studies. The high frequency is related to two factors: in networks where the access is reliable UDP can be used, because these environments are exempt, or have no significant package loss; it can also be used in networks based on the Internet, because it is based on datagrams and does not perform verification of delivery, increasing the flow of information, leaving to the application the verification of packages that were not delivered and the order of arrival.

The second most commonly used protocol was TCP. 3DCVE that are implemented on high-speed networks are not affected by steps that a message has to take when it is a TCP connection. The use of the TCP protocol in data networks based on the Internet is not indicated, because the size of the header and ACK are responsible for reducing the package flow capacity [10]. For this kind of environment, as mentioned, the UDP protocol is more suitable. Some studies showed comparisons between UDP and TCP [10], [5]. The UDP protocol had better results (if we consider only the data flow in experiments realized on Local Area Network (LAN)). The most suitable protocols used for communication in 3DCVE were, however, SCTP and RTP/RTCP, because they combine features of TCP and UDP, optimized for multimedia applications (stream and real-time). According to Boukerche and colleagues [22], SCTP provides a reliable delivery system for key packages and unreliable delivery for normal update packages. Smoothed SCTP, that adds strategies for dealing with jitter, implements a small buffer at the client side.

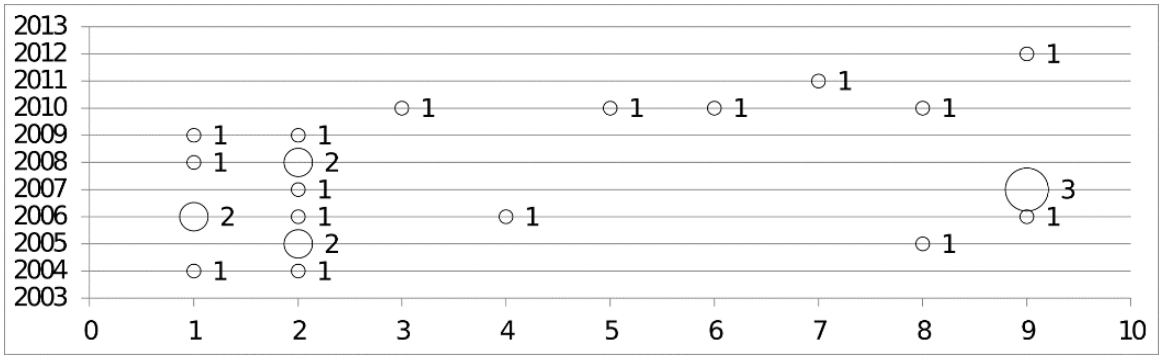


Fig. 2. Frequency of use of protocol (by year)

Some protocols were found only in one primary study. These can be considered "evidence deserts" (which should be performed as new or improved research). Protocols with only one occurrence were: AppTraNet, RTP / RTCP, RTEP and PGM. Some experiments were performed with the RTEP and ICMP. The RTP/RTCP, despite being used in a recent study [13], only occurred once, but it is cited in some textbooks and studies as a trend [23], [24], [25]. PGM was used in conjunction with QUANTA to develop the network module iTILE [26]. ISTP was used to encapsulate the packages to be exchanged among nodes of 3DCVE, although, if the message type is a key state, it needs to be transmitted in a reliable way (ACK and NACK) [9].

Internet Control Message Protocol (ICMP) had two occurrences, but in both studies was used as a benchmark for other protocols [13], [27], because it is not used to exchange data.

The QUANTA framework appeared in some studies [26], [8], [10], [28]. QUANTA is a network framework created by the National Science Foundation (NSF). It provides an easy way to transfer data in distributed systems. The transfer is done at a high level of application, being invisible to the developer. In this review, QUANTA is presented in the bubble plot (Figure 2); although it is not a network protocol, it can be used instead of network protocols.

The majority of primary studies reviewed used data networks with high speed and reliable characteristics, or local and/or regional networks interconnected by optical fibers. The main focus of this review was to find possible solutions for data networks based on the Internet, in order to present the problems and solutions discussed.

An important feature that should be defined at the beginning of implementation of an 3DCVE is the network topology that will be used. Four principal topologies were used in the primary studies. The physical topologies most used were: mesh, star, hybrid and peer-to-peer. Logical topologies were: intra-domain, inter-domain metadata overlay and overlay. This answers RQ2.

Peer-to-peer topology was the most common in recent years in the design of 3DCVE, because it allows full distribution, low latency and efficient management of messages, if implemented with some care. Wang and colleagues [29] used peer-to-peer topology to propose an architecture for a 3DCVE that explores user location in the environment. The user node only receives information about nodes that are close to the user in the environment. Steiner and Biersack [30] used a peer-to-peer topology based on Delaunay triangulation. The algorithm is dynamic with respect to the connecting nodes of a 3DCVE. The basic idea of the algorithm is to classify links that connect nodes in an intra-cluster or inter-cluster.

Implementing 3DCVE in data networks based on the Internet has some challenges, such as package loss, latency and jitter. This answers RQ3. Some primary studies discussed issues related to package loss [8], [28], [31], [32], [9], [5], [29]. Nevertheless, none of them presents experiments in a real environment, but only simulated results. Package loss is not the most harmful problem in the implementation of 3DCVE, however, and can be solved by certain prediction techniques [22].

With respect to network latency, researchers assume that more than 200ms can influence the user experience in a 3DCVE [31]. It is, however, possible to find latency of 600ms in intercontinental connections over the Internet, so this is one of the main challenges to implementation. Among the challenges, jitter is the most harmful, because in 3DCVE it is important to maintain constant speed for the delivery rate of packages. Some researchers have solved jitter problems with buffering techniques [27], [22]. In gigabit networks, such problems were not found; some authors even claimed that the bottleneck caused by the network would not matter in the near future. DeFanti and colleagues [33] claim that the bottleneck is owed to bus, network interfaces and protocols.

3 The Proposed Framework

As shown in the previous sections there are several solutions to problems related to data networks. All solutions are treated in isolation. The integration of these solutions into a single framework can facilitate the development of 3DCVE based on graphics clusters, which are clusters tailored to computer graphics applications. This proposal describes an architecture for a framework designed to help developers and researchers to develop 3DCVE. The systematic review identified problems and solutions to problems in the context of 3DCVE development. It was possible to identify acceptable values, such as package delivery delay.

3.1 Dynamic-Adaptive Framework

Virtual environments are rendered, and thus one of the ways to accomplish that is by using graphics clusters. These are composed of hardware and software combined. The software components that make up a graphic cluster can be classified in four categories: applications, visualization tools, cluster system management and operating system.

In this proposed framework, the focus is on the graphic cluster management layer. This layer is responsible for communication between the nodes and for abstracting the ways of communication between these nodes, including inter-cluster communication (server to server).

This paper proposes a framework for 3DCVE development, which allows adjustments in the runtime of the settings of the connections between remotely located

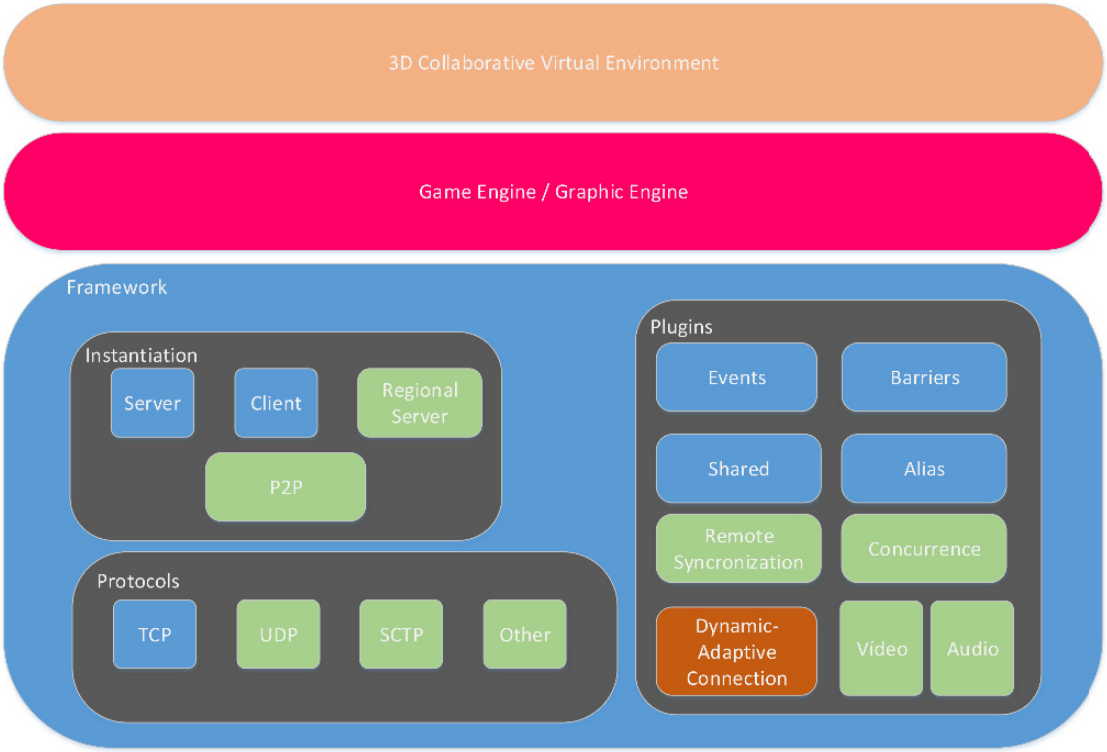


Fig. 3. Architecture of Framework

graphics clusters. We call this layer dynamic-adaptive connection management, given that the adjustments are made at runtime and aim to provide quality of connection. Figure 3 depicts the proposed framework.

The framework was developed by means of libGlass [34]. Some of the modules described in Figure 3 have been extended to provide quality of service to 3DCVE. The framework is divided into three modules:

- The Instantiation module is responsible for implementing the topology connection between clusters and their rendering nodes;
- The Protocol module provides the protocols used in the communication between clusters and their nodes. The protocols developed so far are TCP, UDP and STCP. This component has a packing/unpacking infrastructure for messages, which supports all basic types (integer, float, string, etc.); in addition, one can define one's own types. This packer/unpacker ensures interoperability between operating systems; and
- The plug-ins are intended to provide support for cooperative, synchronous/asynchronous applications, and allow users, geographically distributed, to perform activities or achieve common goals. Some of the plug-ins available in libGlass have been extended. The plug-ins provide services such as: transmission of Events (Events plug-in), data sharing (Shared plug-in), barrier synchronization (Barriers plug-in) and association functions (Alias plug-in).

3DCVE development requires certain characteristics so libGlass has been extended. Some plug-ins were created or modified. The plug-ins are:

- Remote Synchronization: the Shared plug-in has been extended from libGlass, allowing each local server to synchronize with remote servers. The server receives the data from the local nodes and sends them to remote servers;
- Concurrence: the plug-in which provides concurrent access for shared objects for each location. So libGlass is compatible with various types of cooperative applications, various methods should be available. Initially, we implemented the non-optimistic locking method (which ensures that all events are always executed in the correct order for all users, not allowing events to be executed out of order);
- Control of Users: the plug-in that enables the management of users through services such as adding a new user, banning users, storing users in a session and updating the users on all instances of the application;
- Video: the plug-in that allows each site to transmit or receive videos. This plug-in is an extension of plug-in Event;
- Audio: the plug-in that allows each site to transmit or receive audio. Thus, like the Video plug-in, this is also an extension of plug-in Event; and
- Dynamic-Adaptive Connection: the plug-in that allows the checking of network adversities at runtime, allowing different connection configurations to be used. The log files generated during the execution of a 3DCVE are analyzed, enabling adjustments. The analysis of the connection is done through inferences based on fuzzy logic.

These features may not be sufficient to meet all the requirements of cooperative applications, so with the emergence of new services and requirements, new features can be proposed and implemented in the framework.

3.2 Dynamic-Adaptive Connection

This plug-in is responsible for maintaining the quality of the connection between the servers of the graphics clusters. In order to classify the connection, and to maintain the quality of connection. Concepts of fuzzy logic are used to evaluate the state of the connections.

According to results shown in the systematic review, it was possible to estimate the connection quality in relation to the connection delay, package loss, jitter, etc. A connection may have many values for each of the mentioned items, however, and thus the use of a fuzzy inference system assists decisions.

Fuzzy logic allows the subjective concepts of a specialist to be expressed and provides assistance with these concepts mathematically, i.e., translation of human knowledge. It is well suited for certain classes of objects that do not respond to conventional treatment [35], such as empirical systems, vague systems, imprecise systems and some situations which it is difficult to model using conventional methods. Thus, it is possible to model the behavior of the connections, adapting them according to linguistic terms.

Linguistic variables were created in order to assess the quality of interconnection between the servers (P2P) and nodes (client-server) of graphics clusters. Variables were divided into two categories: input and output. Input variables (Table 1) are used to measure the main problems encountered in 3DCVE, according to the papers reviewed. These problems should not be classified as present or not, whereas intermediate values can be taken into account. Many problems can also be evaluated in combination, thus allowing different actions to be taken. The input variables identified so far are:

- Package delivery delay: this variable takes into account the package delivery delay between servers (P2P) and can be classified as Harmful, Not harmful and Highly Harmful;
- Time variation in package delivery: this variable takes into account the average time of package delivery between servers. It is classified as Slightly harmful, Harmful and Highly harmful; and
- Package loss: this variable takes into account the packages that are lost in the connections between servers. It is classified as Low, High, and Without loss.

The output variables (Table 2) are used in order to apply actions in the environment according to the ranked values from input variables. The output variables identified and implemented so far are:

- Change the amount of data of the data packages: this variable determines if the package configuration is consistent with the state of the network at a given time. The following linguistic terms have been classified:
    - No: the packet is not changed. A default size is used (1500 bytes). Only one update message is sent by package;
    - Medium: the packet is modified when necessary being used to sending multiple messages, but the size is still the default; and
    - Big: the packet size and configuration (data) has increased according to need.

- Use another communication protocol: this variable determines which network protocol may be used for interconnection servers and nodes of the graphics clusters. The following linguistic terms were classified:
  - UDP: the standard protocol is maintained;
  - TCP/UDP: according to the combination of values classified by the input variables, different protocols may be used. The TCP/UDP combines TCP and UDP. TCP to transmit key packages and UDP to transmit update packages;
  - SCTP: similar to TCP/UDP, it uses features present in both protocols, but provides other solutions, like buffering.
- Buffer: this variable determines if the use of buffering is required. The following linguistic terms were classified:
  - No Buffer: the default connection mode is maintained;
  - Level 1: state update packages are buffered, in order to keep the refresh rate of environments without sacrificing frame rate presented to the participants of the 3DCVE; and
  - Level 2: a small buffer is used on the client side. We used strategies from SCTP protocol for the buffering.
- Predict package: this variable is responsible for predicting lost packages during the connection between the servers. The following linguistic terms were classified:
  - No Predict: the default connection mode is maintained;
  - Level 1: a linear prediction algorithm has been used, where the lost update packages are predicted from the received packages [22]; and
  - Level 2: here we used reactive latency compensation, the dead reckoning algorithm. It solves two problems simultaneously: reduced bandwidth consumption and mitigation of the effect of network delay.

Table 1. Input Variables

	Input Variables	
Name of variable	**Linguist terms**	**Degree of relevance**
Package delivery delay	Not harmful	[0, 60]
	Harmful	[55, 200]
	Highly harmful	[190, 600]
Time variation in packages delivery	Slightly harmful	[0, 5]
	Harmful	[4, 10]
	Highly harmful	[9, 100]
Loss of package	Without loss	[0, 3]
	Low	[2, 6]
	High	[5, 20]

The membership function used is the trapezoidal (1). It results from the relevance degree where a, b, c and d are the edges of the trapezium. The x is the domain. The membership function is crucial to the use of fuzzy logic [35].

$$trapmf = (x, a, b, c, d) = \max\left(\min\left(\frac{x-a}{b-a}, 1, \frac{d-x}{d-c}\right), 0\right). \quad (1)$$

The combination of different values of input variables generates different outputs. Table 3 presents examples of the Fuzzy Rules created for the inference system. The rules were descripted using the Fuzzy Control Language (FCL). The complete list of rules can be visualized in: http://54.213.29.17/rules.fcl.

Table 2. Output Variables

Output Variables		
Name of variable	**Linguist terms**	**Degree of relevance**
Changes the amount of data of the data packages	No	[0, 0,3]
	Medium	[0.28, 0.6]
	Big	[0.55, 1.0]
Use another communication protocol	UDP	[0, 0.3]
	TCP/UDP	[0.25, 0.6]
	SCTP	[0.55, 1.0]
Buffer	No Buffer	[0, 0.3]
	Level 1	[0.28, 0.7]
	Level 2	[0.68, 1.0]
Predict package	No Predict	[0, 0.3]
	Level 1	[0.25, 0.6]
	Level 2	[0.55, 1.0]

Table 3. Fuzzy Rules – just some examples

Fuzzy Rules
RULE 1: IF delay IS harmful AND time_variation IS slightly_harmful AND loss IS low THEN buffer IS level_1;
RULE 2: IF delay IS harmful AND time_variation IS slightly_harmful AND loss IS low THEN predict IS level_1;
RULE 3: IF delay IS harmful AND time_variation IS slightly_harmful AND loss IS low THEN protocol IS tcp_udp;
RULE 4: IF delay IS harmful AND time_variation IS slightly_harmful AND loss IS low THEN data_packages IS medium;
RULE N: …

Figure 4 depicts the input and output variables of the Dynamic-adaptive module proposed. This figure was plotted by using the values present in both Table 1 and Table 2. The top of trapezoid represents the value 1.0 for the membership function, that is, total inference.

Figure 5 shows the simulated results for three different sets of values. These parameters were found in connections between the graphic clusters of our research partners. The sets of values are: Delay 12ms, Time variation 0ms and Packet loss 0 (left); Delay 100ms, Time variation 3ms and Packet loss 0 (right); and Delay 300ms, Time variation 6ms and Packet loss 5 (bottom). The results confirm the expected values when compared to the output values (Figure 4).

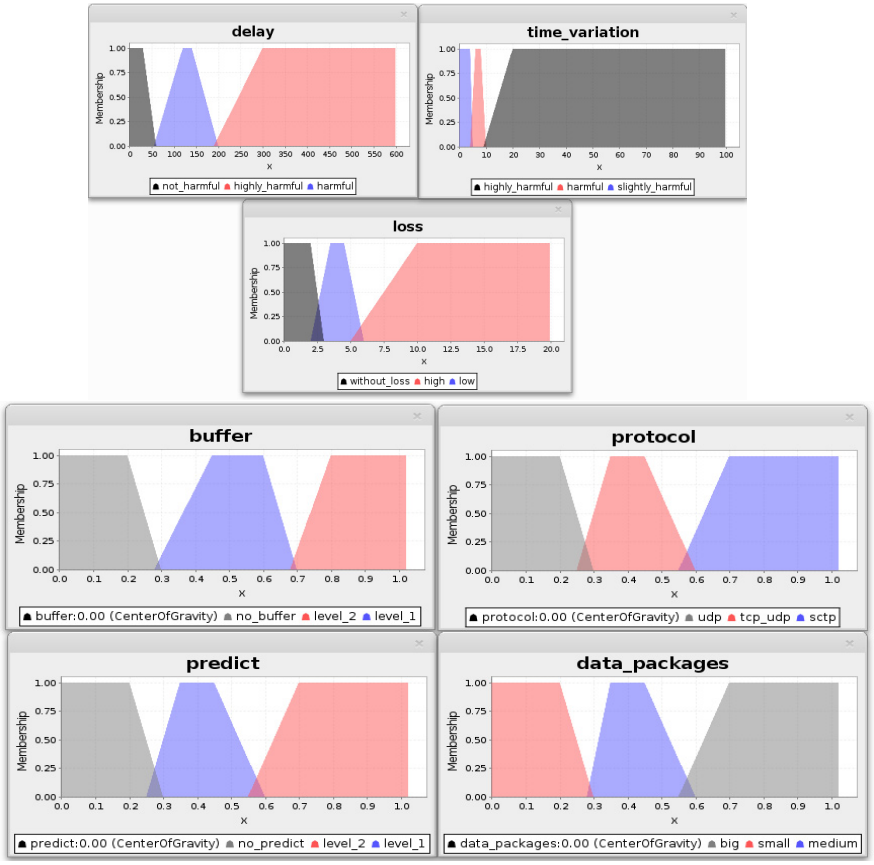


Fig. 4. Input and Output Variables of Dynamic-adaptive solution

4 Final Remarks

The process described by Kichenham and colleagues [2] was followed as standard for the conduct of the review. We examined 132 primary studies and found four physical topologies and nine protocols that had been used in the last 10 years. We also identified some algorithms that aimed to minimize network issues. It is hoped that researchers can use this document as a basis to advance their researches and developers can use it to identify problems and solutions already consolidated. Thus, this systematic review is of relevance to both academic and professional areas. It

presents possible solutions to the implementation of 3DCVE, focusing on a variety of network protocols and topologies.

The review showed that the most commonly used physical topology was peer-to-peer, used to mitigate the problems occurring in networks based on the Internet. For the most part, however, the topology used was not mentioned. In some primary studies the authors mentioned the use of a centralized server; thus, we can assume that they used a star topology or mesh type.

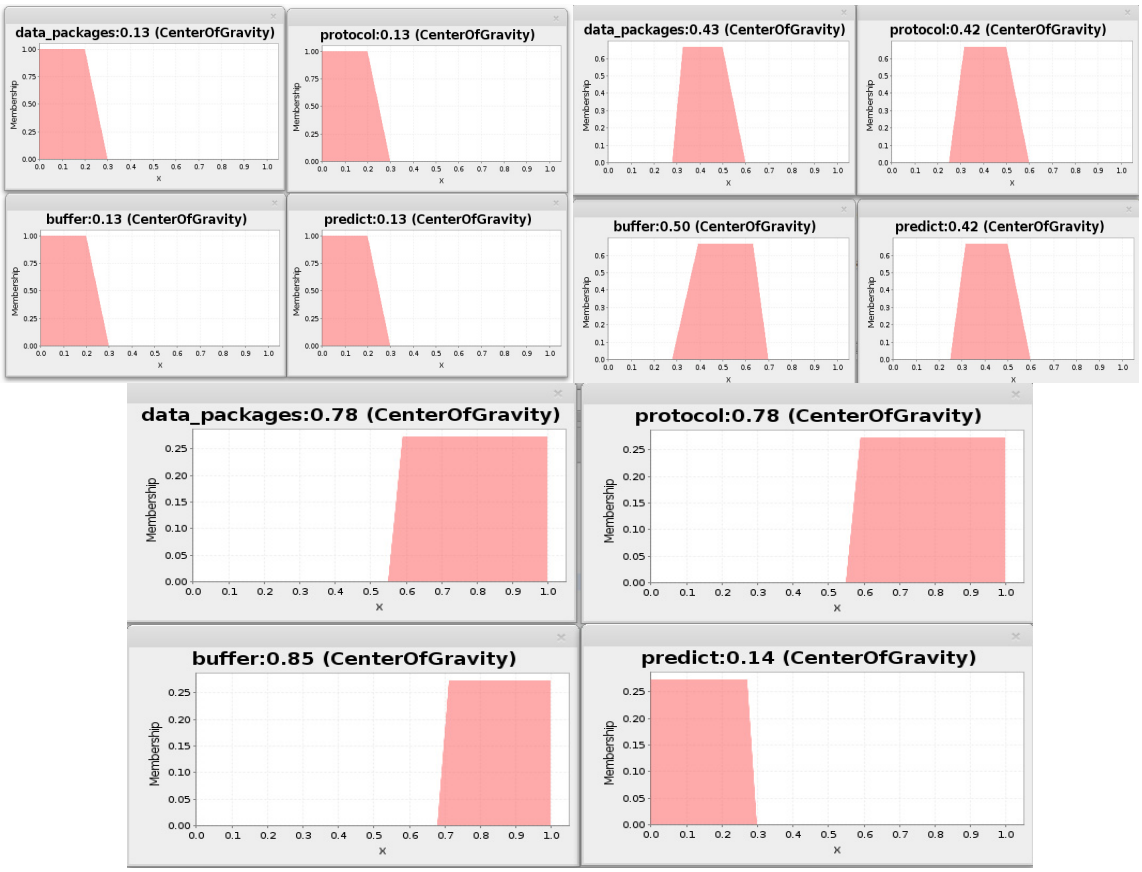


Fig. 5. Output Variables of Dynamic-adaptive solution. A) Data packages (No) / Protocol (UDP) / Buffer (No Buffer) / Predict (No predict). B) Data packages (Medium) / Protocol (TCP/UDP) / Buffer (Level1) / Predict (Level1). C) Data packages (Big) / Protocol (SCTP) / Buffer (Level2) / Predict (No predict).

The UDP protocol was used most often. In architectures based on gigabit networks, was identified the suitable for this kind of application, or just require further research. The SCTP protocol, for example, was cited as popular in some papers, but was found in only one study. The protocol list presents the QUANTA framework; although not a protocol it has been used to resolve various issues and is much in evidence in the last few years.

The proposed framework is under development and testing. Solutions discussed in this systematic review are currently being implemented or extended. It is hoped that those based on peer-to-peer and client-server topologies will be sufficient to meet the challenges in the development of 3DCVE based on graphic cluster over the Internet.

The management adaptive-dynamic connection plug-in is expected to provide quality service to 3DCVE based on graphics clusters in execution time. The use of fuzzy logic is a feasible approach, allowing different configurations to be tailored to the features found in the environment.

References

1. Singhal, S., Zyda, M.: Networked virtual environments: design and implementation. ACM Press/Addison-Wesley Publishing Co., New York (1999)
2. Kitchenham, B., Pearl Brereton, O., Budgen, D., Turner, M., Bailey, J., Linkman, S.: Systematic literature reviews in software engineering- a systematic literature review. Inf. Softw. Technol. 51(1), 7–15 (2009)
3. Dyba, T., Dingsoyr, T., Hanssen, G.K.: Applying systematic reviewsto diverse study types. In: International Symposium on EmpiricalSoftware Engineering and Measurement, ESEM 2007, pp. 225–234. IEEE Computer Society, Washington,DC (2007)
4. Baladi, M., Vitali, H., Fadel, G., Summers, J., Duchowski, A.: A taxonomy for the design and evaluation of networked virtualenvironments: its application to collaborative design. InternationalJournal on Interactive Design and Manufacturing (IJIDeM) 2(1), 17–32 (2008)
5. Yuan, Q., Lu, D.: A latency-adaptive communication architecturefor inter-networked virtual environments. In: 2004 IEEE International Conference on, Systems, Man and Cybernetics, vol. 7, pp. 6296–6301 (2004)
6. Drolet, F., Mokhtari, M., Bernier, F., Laurendeau, D.: A softwarearchitecture for sharing distributed virtual worlds. In: Virtual RealityConference, VR 2009, pp. 271–272. IEEE (2009)
7. Tumanov, A., Allison, R., Stuerzlinger, W.: Variability-aware latencyamelioration in distributed environments. In: Virtual Reality Conference, VR 2007, pp. 123–130. IEEE (March 2007)
8. Steinbach, E., Hirche, S., Kammerl, J., Vittorias, I., Chaudhari, R.: Haptic data compression and communication. IEEE Signal Processing Magazine 28(1), 87–96 (2011)
9. Ling, C., Xiao-Lei, X., Gen-Cai, C., Chuen, C.: An effectivecommunication architecture for collaborative virtual systems. In: International Conference on Communication Technology Proceedings, ICCT 2003, vol. 2, pp. 1598–1602 (2003)
10. Sung, M.Y., Yoo, Y., Jun, K., Kim, N.-J., Chae, J.: Experimentsfor a collaborative haptic virtual reality. In: 16th International Conference on Artificial Reality and Telexistence–Workshops, ICAT 2006, December 29-December 1, pp. 174–179 (2006)
11. Li, L., Li, F., Lau, R.: A trajectory-preserving synchronizationmethod for collaborative visualization. IEEE Transactions on Visualization and ComputerGraphics 12(5), 989–996 (2006)
12. Malheiros, V., Hohn, E., Pinho, R., Mendonca, M., Maldonado, J.C.: A visual text mining approach for systematic reviews. In: Proceedings of the First International Symposium on EmpiricalSoftware Engineering and Measurement, ESEM 2007, pp. 245–254. IEEE Computer Society, Washington, DC (2007)
13. Correa, M., Schpector, J., Trevelin, L., de Paiva Guimaraes, M.: Immersive environment for molecular visualization to interaction between research groups geographically dispersed. In: 2010 3rd International Symposium on Applied Sciencesin Biomedical and Communication Technologies (ISABEL), pp. 1–5 (November 2010)

14. Ubik, S., Navrátil, J., Žejdl, P., Halák, J.: Real-time stereoscopic streaming of medical surgeries for collaborative eLearning. In: Luo, Y. (ed.) CDVE 2012. LNCS, vol. 7467, pp. 73–77. Springer, Heidelberg (2012)
15. Kenyon, R., Leigh, J.: Networked virtual environments and rehabilitation 26(VII), 4832–4835 (2004)
16. Watanabe, M.C., Okuda, M.B., Karube, Y., Matsukura, R., Matsuzawa, T.: Collaborative environment with visualizingmedical volume data by virtual reality, pp. 347–352 (2007)
17. Kadavasal, M., Oliver, J.: Towards sensor enhanced virtual realityteleoperation in a dynamic environment 2(PART B), 1057–1065 (1996); cited By (since 1996)
18. Dong, K., Nan, K., Tilak, S., Zheng, C., Xu, D., Schulze, J., Arzberger, P., Li, W.: Real time biomedical data streamingplatform (rimes): A data-intensive virtual environment 2, 342–346 (2010)
19. Bruns, F., Erbe, H.-H., Muller, D., Schaf, F., Pereira, C., Reichert, C., Campana, F., Krakheche, I.: Collaborative learning andengineering workspaces 1(PART 1), 112–117 (2007)
20. Liang, J.: Modeling an immersive vr driving learning platformin a web-based collaborative design environment. Computer Applications in Engineering Education 20(3), 553–567 (2012)
21. Guo, T.-T., Guo, L., Wang, Z., Lin, S., Pan, J.-H.: A networkedvirtual experiment system based on virtual campus 3, 884–888 (2009)
22. Boukerche, A., Shirmohammadi, S., Hossain, A.: Moderating simulation lag in haptic virtual environments. In: 39th Annual Simulation Symposium 2006, pp. 269–277 (April 2006)
23. Kurose, J., Ross, K.: Computer Networking: A Top-Down Approach, 5th edn. Addison-Wesley (March 2009)
24. Nahrstedt, K., Yang, Z., Wu, W., Arefin, A., Rivas, R.: Nextgeneration session management for 3d teleimmersive interactiveenvironments. Multimedia Tools and Applications 51(2), 593–623 (2010)
25. Ronningen, L., Panggabean, M., Tamer, O.: Toward futuristic near natural collaborations on distributed multimedia plays architecture. In: 2010 IEEEInternational Symposium on Signal Processing and Information Technology (ISSPIT), pp. 102–107 (2010)
26. Cho, Y., Kim, M., Park, K.: Lotus: composing a multi-user interactive tiled display virtual environment. The Visual Computer 28(1), 99–109 (2012)
27. Anthes, C., Haffegee, A., Heinzlreiter, P., Volkert, J.: A scalablenetwork architecture for closely coupled collaboration. Computingand Informatics 24(1), 31–51 (2005)
28. You, Y., Sung, M., Kim, N., Jun, K.: An experimental study on theperformance of haptic data transmission in networked haptic collaboration. In: The 9th International Conference on Advanced Communication Technology, vol. 1, pp. 657–662 (February 2007)
29. Wang, Y., Li, Z., Zhang, W.: A fully distributed p2p communications architecture for network virtual environments. In: 2011 7th International Conference on WirelessCommunications, Networking and Mobile Computing (WiCOM), pp. 1–4 (2011)
30. Steiner, M., Biersack, E.: Ddc: A dynamic and distributed clusteringalgorithm for networked virtual environments based on p2p networks. In: Proceedings of the 25th IEEE International Conference on ComputerCommunications, INFOCOM 2006, pp. 1–6 (2006)
31. Chen, L.: Effects of network characteristics on task performance ina desktop cve system. In: 19th International Conference on Advanced Information Networking andApplications, AINA 2005, vol. 1, pp. 821–826 (March 2005)
32. You, Y., Sung, M.Y.: Haptic data transmission based on theprediction and compression. In: IEEEInternational Conference on Communications, ICC 2008, pp. 1824–(May 1828)

33. DeFanti, T.A., Leigh, J., Renambot, L., Jeong, B., Verlo, A., Long, L., Brown, M., Sandin, D.J., Vishwanath, V., Liu, Q., Katz, M.J., Papadopoulos, P., Keefe, J.P., Hidley, G.R., Dawe, G.L., Kaufman, I., Glogowski, B., Doerr, K.-U., Singh, R., Girado, J., Schulze, J.P., Kuester, F., Smarr, L.: The optiportal, a scalable visualization,storage, and computing interface device for the optiputer. Future Generation Computer Systems 25(2), 114–123 (2009)
34. Gnecco, B.B., Dias, D.R.C.: Glass library (August 2013), http://sourceforge.net/projects/libglass
35. Zadeh, L.A.: Fuzzy logic, neural networks, and soft computing. Commun. ACM 37(3), 77–84 (1994)

Anexo 6: Comprovante da publicação no *IEEE International Conference on Information Reuse and Integration 2014* (IRI 2014)

- Uma cópia do artigo é apresentado a seguir (a partir da próxima página).

Towards a Refactoring Catalogue for Knowledge Discovery Metamodel

Rafael S. Durelli†, Daniel S. M. Santibáñez*, Márcio E. Delamaro† and Valter Vieira de Camargo*
†Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo,
Av. Trabalhador São Carlense, 400, São Carlos – SP – Brazil
Email: {rdurelli, delamaro}@icmc.usp.br
*Departamento de Computação, Universidade Federal de São Carlos,
Caixa Postal 676 – 13.565-905, São Carlos – SP – Brazil
Email: {daniel.santibanez,valter}@dc.ufscar.br

***Abstract*—Refactorings are a well known technique that assist developers in reformulating the overall structure of applications aiming to improve internal quality attributes while preserving their original behavior. One of the most conventional uses of refactorings are in reengineering processes, whose goal is to change the structure of legacy systems aiming to solve previously identified structural problems. Architecture-Driven Modernization (ADM) is the new generation of reengineering processes; relying just on models, rather than source code, as the main artifacts along the process. However, although ADM provides the general concepts for conducting model-driven modernizations, it does not provide instructions on how to create or apply refactorings in the Knowledge Discovery Metamodel (KDM) metamodel. This leads developers to create their own refactoring solutions, which are very hard to be reused in other contexts. One of the most well known and useful refactoring catalogue is the Fowler's one, but it was primarily proposed for source-code level. In order to fill this gap, in this paper we present a model-oriented version of the Fowler's Catalogue, so that it can be applied to KDM metamodel. In this paper we have focused on four categories of refactorings: (*i*) renaming, (*ii*) moving features between objects, (*iii*) organizing data, and (*iv*) dealing with generalization. We have also developed an environment to support the application of our catalogue. To evaluate our solution we conducted an experiment using eight open source Java application. The results showed that our catalogue can be used to improve the cohesion and coupling of the legacy system.**

***Index Terms*—Refactoring, KDM, ADM, Empirical Study**

I. Introduction

Empirical studies show that refactoring can improve maintainability and reusability of systems [1]. Not only existing research which suggests that refactoring is useful, but it also suggests that refactoring is a frequent practice [2].

In a parallel and related research line, Object Management Group has employed a lot of effort to define standards in the refactoring process, creating the concept of ADM. ADM is the next generation of software reengineering, relying on standard models to perform the whole process. ADM follows the Model-Driven Development (MDD) [3] guidelines and comprises two major steps. Firstly a reverse engineering is performed starting from the source code and a model instance is created. Next successive transformations are applied to this model up to reach a good abstraction level in model called KDM. Upon this model, several refactorings, and optimizations can be performed in order to solve problems found in the legacy system. Secondly a forward engineering is carried out and the source code of the modernized target system is generated. According to the OMG the most important artifact provided by ADM is the KDM metamodel, which is a multipurpose standard metamodel that represents all aspects of the existing information technology architectures. The idea behind the standard KDM is that the community starts to create parsers from different languages to KDM. As a result everything that takes KDM as input can be considered platform and language-independent. For example, a refactoring catalogue for KDM can be used for refactoring systems implemented in different languages. The current version of the KDM is 1.3 and it is being adopted by ISO as ISO/IEC 19506 [3].

In the area of object-oriented programming, refactorings are the technique for improving the structure of legacy system without changing its external behavior [2]. Nowadays it is evident that refactorings are useful to improve the quality of source code, and thus, to increase its maintainability. However, although ADM, and mainly KDM, had been proposed to support the modernization of legacy systems, up to this moment there are no proposals of refactoring catalogues for KDM. Therefore, software modernizers/reengineers need to develop your own solutions to transform source KDM instances in target ones. Usually these solutions are proprietary and very difficult to reuse. Besides, available object-oriented refactoring catalogues can not be reusable as they are, because they have been created to source-code refactoring. This forces the software engineer to refactor a legacy system without any kind of dedicated support as using the KDM model.

As refactoring a legacy system can be very complex, manual modifications without any catalogue may lead to unwanted side-effects and result in a tedious and error-prove refactoring process. Therefore, we claim that working out a catalogue of refactoring to the KDM specification is indispensable because software engineers can reduce time and effort during the refactoring of legacy systems once we are using techniques of MDD [4]. Furthermore, we also argue that devising a catalogue of refactoring by means of KDM specification makes this catalogue be both language-independent and standardized [5].

We believe there are two main hurdles to be overcome so that refactoring techniques can be used in the KDM model in an effective and widespread way. The first hurdle is the lack a catalogue of well known refactorings based on the KDM

model. Also, software modernizers would greatly benefit from the possibility to follow a catalogue of refactoring, in practice they mostly rely on experience or intuition because of the lack of approaches providing a catalogue for KDM. In order to address this first hurdle we present a dedicated refactoring catalogue for the KDM metamodel which is based on the catalogue proposed by Fowler [2]. We chose this catalog once it contains well known, basic and fine-grained refactorings. This allows that larger refactorings can be applied when combining a sequence of them, i.e., a chain of refactorings.

A second hurdle is the absence of an Integrated Development Environments (IDEs) to lead engineers to automatically apply refactorings using the KDM model as such exist in others object-oriented languages. The catalogue presented by Fowler et al. [2] provided a basis on which developers could rely to build tool support for object-oriented refactoring: similar catalogue for the KDM models are likely to bring similar benefits to assist software modernizers during the modernization process. Therefore, to overcome the second hurdle we devised a Eclipse plug-in named Modernization-Integrated Environment (MIE), which is an environment that implements all refactoring available in the catalogue herein. The novelty of this environment is not the supporting technologies and tools, but rather its catalogue of refactoring based on KDM.

Moreover, in order to provide some evidence of the our dedicated refactoring catalogue for KDM we performed an experiment using eight open source Java application. More specifically, we conducted a reverse engineering in order to get the KDM model of these eight open source Java application. Then, we applied three different refactorings in their KDM models. Experimental results show that the our dedicated refactoring catalogue improved the legacy system's cohesion.

Therefore, the main contributions of this paper are threefold: (*i*) we show a dedicated refactoring catalogue for KDM; (*ii*) we demonstrate the feasibility of our catalogue by implementing it as an Eclipse plug-in; *(iii)* we show the results of an experiment by applying some refactorings to eight open source programs and their KDM models.

This paper is structured as follows: in Section II ADM and KDM are explained; in Section III, the catalogue of refactoring for KDM is shown; in Section IV, a Eclipse plug-in to support the catalogue is described; Section V summarizes the experimental results; in Section VI, there are related works and Section VII suggests future work and makes concluding remarks.

II. Background

A. ADM and KDM

ADM is the concept of modernizing existing systems with a focus on all aspects of the current systems architecture and the ability to transform current architectures to target architectures by using all principles of MDD [6, p. 60].

To perform a system modernization, ADM introduces several modernization standards: Abstract Syntax Tree Metamodel (ASTM), Knowledge Discovery Metamodel (KDM), Structured Metrics Metamodel (SMM), System Assurance & Evidence, Software Quality and Business Architecture Standards. These standards collectively provide or can provide the universe of metadata that defines existing software environments. However, here we focus on KDM because it is the key cornerstone of ADM and the main ideas of our research. The goal of the KDM standard is to define a metamodel to represent all the different legacy software artifacts involved in a legacy information system (e.g. source code, user interfaces, databases, etc.). The metamodel of the KDM standard provides a comprehensive high-level view of the behavior, structure and data of legacy information systems by means of a set of facts. The main purpose of the KDM specification is not the representation of models related strictly to the source code nature such as Unified Modeling Language (UML). While UML can be used to generate new code in a top-down manner, an ADM-based process using KDM starts from the different legacy software artifacts and builds higher-abstraction level models in a bottom-up manner through reverse engineering techniques.

KDM specification owns some KDM domain, each domain defines an architectural viewpoint. In order to define the catalogue of refactoring for the KDM we need to focus just on the Program Element Layer - more specifically in the Code Package, which represents the code elements of a program (classes, fields and methods) and their associations. Therefore, it is important to dig a little deeper in the Code Package. The Code Package consists of 24 classes and contains all the abstract elements for modeling the static structure of the source code. In Table I is depicted some of them. This table identifies KDM metaclasses possessing similar characteristics to the static structure of the source code. Some metaclasses can be direct mapped, such as Class from object-oriented language, which can be easily mapped to the ClassUnit metaclass from KDM.

Table I
METACLASSES FOR MODELING THE STATIC STRUCTURE OF THE SOURCE-CODE

Source-Code Element	KDM Element
Class	ClassUnit
Interface	InterfaceUnit
Method	MethodUnit
Field	StorableUnit
Local Variable	Member
Parameter	ParameterUnit
Association	KDM RelationShip

III. Refactoring Catalogue for Knowledge Discovery Metamodel

In this section we describe the catalogue of refactoring for the KDM herein proposed. To create this catalogue we adapted some fine-grained refactorings proposed by Fowler [2]. As stated before we chose the Fowler's refactorings once them are well known, basic and fine-grained refactorings. It is worth highlighting that for drafting the catalogue of refactorings in this paper, we also used a format similar to Fowler [2].

The catalogue is structured in four groups as can be seen in the Table II, which contains 17 refactorings followed by

Table II
REFACTORING CATALOGUE FOR KNOWLEDGE DISCOVERY METAMODEL

N	Name of the Refactoring	Description
	Rename Feature	
1	*Rename ClassUnit, StorableUnit and MethodUnit*	A ClassUnit, a StorableUnit or a MethodUnit does not reveal its purpose
	Moving Features Between Objects	
2	*Move MethodUnit*	A MethodUnit is being using by another ClassUnit than the ClassUnit on which it is defined.
3	*Move StorableUnit*	A StorableUnit is used by another ClassUnit more than the ClassUnit on which it is defined.
4	*Extract ClassUnit*	You have one ClassUnit doing work that should be done by two ClassUnit.
5	*Inline ClassUnit*	A ClassUnit is not doing very much.
	Organing Data	
6	*Replace data value with Object*	You have a data item that needs additional data or behavior.
7	*Encapsulate StorableUnit*	There is a public StorableUnit.
8	*Replace Type Code with ClassUnit*	A ClassUnit has a numeric type code that does not affect its behavior.
9	*Replace Type Code with SubClass*	You have an immutable type code that affects the behavior of a ClassUnit.
10	*Replace Type Code with State/Strategy*	You have a type code that affects the behavior of a ClassUnit, but you cannot use subclassing.
	Dealing with Generalization	
11	*Push Down MethodUnit*	Behavior on a superclass is relevant only for some of its subclasses.
12	*Push Down StorableUnit*	A StorableUnit is used only by some subclasses.
13	*Pull Up StorableUnit*	Two subclasses have the same StorableUnit.
14	*Pull Up MethodUnit*	You have MethodUnits with identical results on subclasses.
15	*Extract SubClass*	A ClassUnit has features that are used only in some instances.
16	*Extract SuperClass*	You have two ClassUnits with similar features.
17	*Collapse Hierarchy*	A superclass and subclass are not very different.

a short description. Due to space limitations in the followings subsections is described only three refactorings: **Extract ClassUnit**, **Replace data value with Object** and **Pull Up MethodUnit**. In order to highlight how these refactorings can be applied into KDM we also show two algorithms. These algorithms can assist other software modernizers to create news refactorings once these refactorings show explicitly how to handle the KDM metamodel. In the followings subsections a description of these refactorings are provided.

A. *Extract ClassUnit*

Input: A source ClassUnit to extract responsibilities, a name of the new class, instances of the meta-class that represent StorableUnit to be moved and instances of the meta-class that represent MethodUnits to be moved.

Summary: You have one ClassUnit doing work that should be done by two.

Solution: Create a new ClassUnit and move the relevant StorableUnit and MethodUnits from the old ClassUnit into the new ClassUnit.

Parameters:

- A source ClassUnit to extract responsibilities.
- The name of the new ClassUnit.
- Instances of the meta-class that represent StorableUnits to be moved.
- Instances of the meta-class that represent MethodUnits to be moved.

Refactoring Guidelines:

- Split the responsibilities of the class.:
  - Identify the StorableUnits that should not be in source ClassUnit.
  - Identify the MethodUnits that should not be in source ClassUnit.
- Create a new instance of ClassUnit to express the split-off responsibilities.
- Use **Move StorableUnit** on each StorableUnit you wish to move.
- Use **Move MethodUnit** on each MethodUnit you wish to move.

Algorithm 1: Extract ClassUnit

```
Input: String newName, ClassUnit class, Package p,
       StorableUnit[] fields, MethodUnit[] methods
1  begin
2      ClassUnit extracted =
       CodeFactory.eINSTANCE.createClassUnit() ❶;
3      if extracted != null then
4          extracted.setName(newName);
5          extracted.getSource.add(SourceFactory.
           eINSTANCE.createSourceRef());
6          p.getCodeElement().add(extracted);
7      else
8          else return null
9      end
10     StorableUnit link =
       CodeFactory.eINSTANCE.createStorableUnit() ❷;
11     if link != null then
12         link.setName(extracted.getName().toLowerCase());
13         link.getAttribute().add(KdmFactory.eINSTANCE.
           createAttribute());
14         link.getSource().add(SourceFactory.eINSTANCE.
           createSourceRef());
15         extracted.getCodeElement().add(link);
16     else
17         else return null
18     end
19     foreach f in fields do
20         extracted.getCodeElement().add(f) ❸;
21     end
22     foreach m in methods do
23         extracted.getCodeElement().add(m) ❹;
24     end
25     return extracted
26 end
```

To illustrate how the refactoring **Extract ClassUnit** can be implemented in the KDM model, consider the chunk of

pseudo code depicted in Algorithm 1. Before the line 2 be executed one must give the following inputs: (*i*) a name to set the new ClassUnit, (*ii*) a source ClassUnit to extract either StorableUnit or MethodUnit, (*iii*) a Package to put the new ClassUnit, and (*iv*) a set of StorableUnit and MethodUnit. In line 2 an instance of ClassUnit is created. This ClassUnit is created by means of the Abstract Factory Pattern ❶. If the condition in line 3 evaluates to true, the statements in lines 4, 5 and 6 are executed. During their execution the name of the new ClassUnit is set and in line 6 the new ClassUnit is added to the instance of Package. In line 10 a StorableUnit is created ❷ - it represents a link from the old ClassUnit to the new ClassUnit. If the condition in line 11 evaluates to true, the statements in lines 12, 13, 14 and 15 are executed. In line 12 is set the name of the StorableUnit. Line 15 add the created StorableUnit to the created ClassUnit. Lines 19, 20 and 21 illustrate a loop to move all StorableUnit to the new ClassUnit ❸. Similarly, lines 22, 23 and 24 depict another loop to move all MethodUnit to the new ClassUnit ❹.

B. *Replace Data Value With Object*

Input: Instance of the meta-class that represent the StorableUnit that needs additional data or behavior.

Summary: You have a StorableUnit that needs additional data or behavior.

Solution: Turn the StorableUnit into a ClassUnit.

Parameters:

- Instance of the meta-class that represent the StorableUnit that needs additional data or behavior.

Refactoring Guidelines:

- Create a ClassUnit for the StorableUnit.
- Create a StorableUnit of the same type as the value in the new ClassUnit.
- Create an instance of MethodUnit to represent the operation get that takes the StorableUnit as an argument.
- Change the type of the StorableUnit in the source ClassUnit to the new ClassUnit.
- Change the getter in the source ClassUnit to call the getter in the new ClassUnit.

C. *Pull Up MethodUnit*

Input: Instances of subclasses that own in common the superclass.

Summary: A set of subclasses that have at least one MethodUnit in common.

Solution: Move the commons MethodUnits to the superclass.

Precondition: Identify the commons MethodUnits.

Parameters:

- Instances of subclasses that own in common the superclass.

Refactoring Steps:

- For each instance that represent the subclasses identify if there is some MethodUnit in common between these subclasses.
- For each identified MethodUnit applies the refactoring **Move Method**. As parameter to this refactoring it is necessary the identified MethodUnit and the an instance of the ClassUnit that represents the superclass to move it.

In Algorithm 2 is illustrated how the refactoring **Pull Up MethodUnit** can be implemented in the KDM model. Before to start the refactoring **Pull Up MethodUnit** the line 2 must be executed ❶. The statement in this line inspects all MethodUnits in a set of subclasses to ensure they are identical. If the condition in line 3 evaluates to true, the loop in lines 4, 5 and 6 are executed. In this loop all identical MethodUnit are moved to an instance of ClassUnit that represent the superclass ❷. In line 10 all elements of the superclass are obtained ❸. Then two loop are execute in lines 11 and 12, respectively. The inner loop (line 12) verify if two MethodUnits are equals. If the condition in line 13 evaluates to true, then a cast is made in line 14 and the contained MethodUnit is removed in line 15.

Algorithm 2: Pull Up MethodUnit

```
   Input: ClassUnit[] subClasses, ClassUnit superC
1  begin
2      MethodUnit[] commonMethods =
       identifyCommonMethodUnit(subClasses) ❶;
3      if commonMethods != null then
4          foreach mU in commonMethods do
5              superC.getCodeElement().add(mU) ❷ ;
6          end
7      else
8          else return null
9      end
10     EList[] elements = superC.getCodeElement() ❸ ;
11     foreach c in elements do
12         foreach c2 in elements do
13             if (c instanceof MethodUnit) &&
               (c.getName().equals(c2.getName())) then
14                 MethodUnit mRve = (MethodUnit) c2;
15                 superC.getCodeElement().remove(mRve);
16             end
17         end
18     end
19 end
```

IV. Proof-Of-Concept Implementation

We devised a Eclipse plug-in named Modernization-Integrated Environment (MIE) which is split in three layers, as follows: (*i*) Core Framework, (*ii*) Tool Core, and (*iii*) Graphical User Interface (GUI). This plugin was devised on the top of the Eclipse Platform; The first layer we used both Java and Groovy as programming language. Moreover, the Core Framework layer contains a set of Eclipse plug-ins on which our environment is based on, such as MoDisco and Eclipse Modeling Framework (EMF)[1]. We used MoDisco[2] once it is

[1]http://www.eclipse.org/modeling/emf/

[2]http://www.eclipse.org/MoDisco/

an extensible framework to develop model-driven tools to support use-cases of existing software modernization and provides an Application Programming Interface - (API) to easily access the KDM model. Also, EMF was used to load and navigate KDM models that were generated with MoDisco. The second layer, the Tool Core, is where all refactorings provided by our environment were implemented. It works intensively with KDM models, which are XML files. Therefore, we use Groovy to handle those types of files because of the simplicity of its syntax and fully integrated with Java. Finally, the third layer is the Graphical User Interface (GUI) that consists of a set of SWT windows with several options to perform the refactorings based on the KDM model.

V. Experimental Study

This section describes the experiment used to gauge the catalogue of refactoring for KDM. Moreover, this experiment also evaluate the devised Eclipse plug-in, which was earlier described. Specifically, we investigate the following research question:

RQ$_1$: How much of the program's maintainability and understandability (high cohesion) can be affected by applying a set of refactorings in the KDM model?

A. Goal Definition

We use the organization proposed by the Goal/Question/-Metric (GQM) paradigm, it describes experimental goals in five parts, as follows: (*i*) **object of study:** the object of study is our catalogue of refactoring adapted for KDM; (*ii*) **purpose:** the purpose of this experiment is to evaluate the catalogue of refactoring adapted for KDM; (*iii*)**perspective:** this experiment is run from the standpoint of a researcher; (*iv*) **quality focus:** the primary effect under investigation is the improvement in program's maintainability and understandability (high cohesion) after applying the refactorings; (*v*) **context:** this experiment was carried out using Eclipse 4.3.2 on a 2.5 GHz Intel Core i5 with 8GB of physical memory running Mac OS X 10.9.2. Our experiment can be defined as: **Analyze** the catalogue of refactoring adapted for KDM model, **for the purpose of** evaluation, **with respect to** improvement in program cohesion, **from the point of view of** the researcher, **in the context of** heterogeneous subject programs.

B. Hypothesis Formulation

Our research question (**RQ$_1$**) was formalized into hypotheses so that statistical tests can be performed.

Null hypothesis, H$_0$: there is no difference in cohesion before and after to apply a set of refactoring into the KDM model (measured in terms of the metric CAMC and SCC) which can be formalized as:

H$_0$: $\mu_{CAMC_{Bf}} = \mu_{CAMC_{Af}}$ **and** $\mu_{SCC_{Bf}} = \mu_{SCC_{Af}}$

Alternative hypothesis, H$_1$: there is a significant difference in cohesion before and after to apply a set of refactoring into the KDM model (measured in terms of the metric CAMC and SCC) which can be formalized as:

H$_1$: $\mu_{CAMC_{Bf}} \neq \mu_{CAMC_{Af}}$ **and** $\mu_{SCC_{Bf}} \neq \mu_{SCC_{Af}}$

C. Experimental Design

For our evaluation, we need a set of sample KDM models to apply the catalogue of refactoring. However, due to the scarcity of complete KDM models in the public domain, we adopted a reverse engineering approach and generated KDM models from eight open source Java projects by using MoDisco. During the selection of these programs we focused on covering a broad class of applications. In addition, several of the subject programs have been studied elsewhere, making this study comparable with earlier studies. Table III shows the subject KDM models along with some measures of their size. Notice that these measures were obtained automatically by MoDisco and Eclipse Metrics 1.3.6 [3].

Table III
KDM Models used in the Evaluation

ID	Program	KDM Model		
		ClassUnit	StorableUnit	MethodUnit
1	org.gadberry.jexel	75	199	240
2	Jester	14	10	59
3	apache.commons.cli	99	707	621
4	apache.commons.io	357	643	2453
5	JUnit	1041	712	2552
6	org.jaxen	353	440	2063
7	org.snmp4j	329	1146	2340
8	org.dom4j	469	653	3032
Total		2737	4510	13360

We selected three refactorings for our evaluation: **Extract ClassUnit**, **Push Down MethodUnit**, and **Pull up MethodUnit**. We applied each of the three refactorings to every possible location in each KDM model. It is worth to notice that all refactorings were applied completely automatically by means of our devised proof-of-concept tool. To deal with refactorings that go into infinite loops, we set three minutes timeout interval. More specifically, we applied the **Extract ClassUnit** to every class that had more than 300 LOC (Line of Code); we applied the **Push Down MethodUnit** to every method of a class that had a subclass that was not from a library using every such subclass as the target of the *push-down*; and we applied the refactoring **Pull up MethodUnit** to every method of a class that had a superclass that was not from a library, using every such superclass as the target of the *pull-up*. Then after applied all refactorings we counted whether them were successful, i.e., if the intended refactoring could be performed, and how many constraints were generated on the model and on the code side after to apply the refactorings. We also measured both software quality metrics Cohesion Amongst the Methods of a Class (CAMC) and Similarity-based Class Cohesion (SCC)[4] before applying the refactoring on the KDM models and after applying the refactoring on the KDM models. Notice that in this case we actually measured these metrics in the code instead of the KDM model. This was possible as our proof-of-concept tool provides support for the generating of the code after one finishes to apply the refactorings.

[3] http://metrics.sourceforge.net/

[4] CAMC and SCC both are defined as high-level design quality metrics, and an increase in their value means an improvement in program cohesion.

Table IV
RESULTS OF APPLICATION PER PROJECT AND PER REFACTORING

ID	Extract ClassUnit							Push Down MethodUnit							Pull Up MethodUnit						
	Exec.	Succ.	Cons.	CAMC		SCC		Exec	Succ.	Cons.	CAMC		SCC		Exec	Succ.	Cons.	CAMC		SCC	
				Bf	Af	Bf	Af				Bf	Af	Bf	Af				Bf	Af	Bf	Af
1	75	43	32	0.133	0.189	0.078	0.092	350	245	105	0.133	0.164	0.078	0.094	350	311	39	0.133	0.177	0.078	0.098
2	14	7	7	0.168	0.196	0.084	0.096	267	213	54	0.168	0.171	0.084	0.087	267	213	54	0.168	0.178	0.084	0.097
3	99	48	51	0.163	0.176	0.067	0.081	159	130	29	0.163	0.152	0.067	0.073	159	130	29	0.163	0.187	0.067	0.085
4	357	267	90	0.175	0.185	0.088	0.094	444	124	320	0.175	0.159	0.088	0.098	444	124	320	0.175	0.187	0.088	0.089
5	1041	705	336	0.183	0.199	0.063	0.089	468	284	184	0.183	0.1773	0.063	0.076	468	284	184	0.183	0.194	0.063	0.079
6	353	156	197	0.193	0.167	0.087	0.073	1411	841	570	0.193	0.168	0.087	0.079	1411	851	570	0.193	0.198	0.087	0.087
7	329	298	31	0.128	0.198	0.092	0.099	678	489	189	0.128	0.158	0.092	0.095	678	489	189	0.128	0.165	0.092	0.099
8	469	389	80	0.17	0.159	0.085	0.097	398	368	30	0.17	0.15	0.085	0.091	398	368	30	0.17	0.25	0.085	0.097

D. Results and Empirical Analysis for the Cohesion Values

This section presents the experimental results after to apply the refactorings **Extract ClassUnit**, **Push Down MethodUnit**, and **Pull up MethodUnit**. The results are depicted in Table IV. The columns Exec., Succ., and Cons. stand for Executed, Successful and Constraints, respectively. Columns Exec. shows the number of refactorings applied to the original KDM model. Columns Succ. presents the rate of successful refactoring really applied in the KDM model, otherwise Columns Cons. shows the rate of unsuccessful refactoring applied in the KDM model. As can be seen, success rates, constraints generated and changes induced vary widely for every refactoring. As stated before, we also measured some software quality metrics (CAMC and SCC) before and after applying all refactorings. Therefore, Columns CAMC and SCC are split into two cell, i.e., Bf and Af as Bf stand for Before and Af stand for After. As shown, the quality in terms of cohesion is in some case was gradually degraded. However, it is fairly evident that in some case the applied refactorings improved the cohesion, i.e., in some points we had positive impact on the design quality as shown in Table IV.

In Figure 1 is summarized the sampled data of the metrics CAMC and SCC before and after to apply the refactorings. These figure also provide an overview of the significant gain in cohesion that could be achieved by our catalogue of refactoring. Besides achieving gain in cohesion, in this figure it is fairly evident that for almost every refactoring the median after applying the refactorings of both metrics (CAMC and SCC) is greater than before to apply the refactorings. In addition, in these figures also can be observed that the interquartile ranges are reasonably similar (as shown by the lengths of the boxes), though the overall range of the data set is greater before to apply the refactorings (as shown by the distances between the ends of the two whiskers for each boxplot). Just the data set related to the metric CAMC for **Pull Up Method** shows one suspiciously far out values (outliers) which required a closer look.

In Table IV it is possible to point out by looking at columns Succ. that the rate of successful applied refactoring are better than the the rate of unsuccessful. In order to be aware of some unsuccessful refactorings, we manually inspected some refactorings to find the reason why some refactorings could not

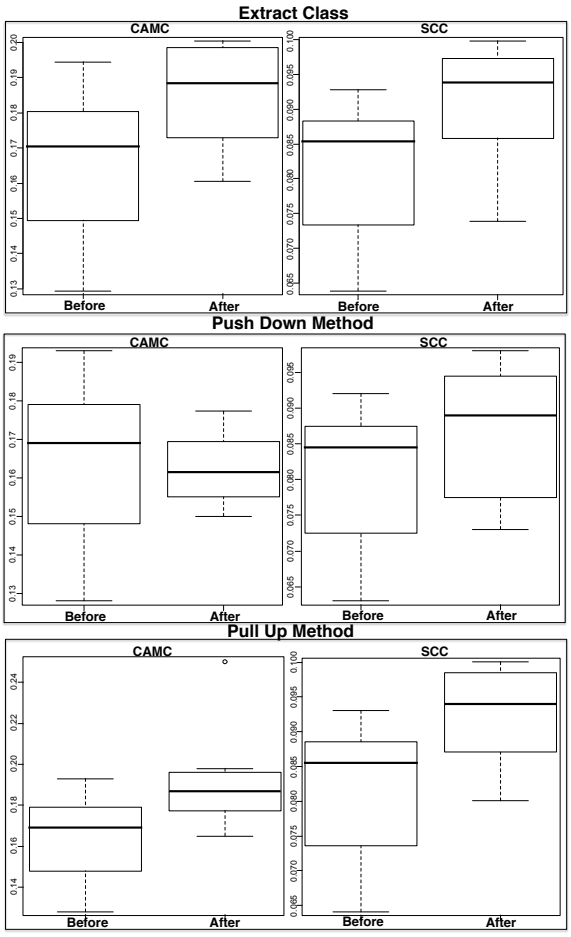


Figure 1. Comparative boxplots for the refactorings.

be applied to the KDM model during the refactoring process. Then we found out that the problem mostly happened when a hierarchy structure of the KDM file was changed radically.

Since some statistical tests only apply if the population follows a normal distribution, before choosing a statistical test we examined whether our sample data departs from linearity. We use Q-Q plots as shown in Figure 2. In these figure one can see that most of the data depart from linearity, indicating normality of the samples. Therefore, we could apply the t-

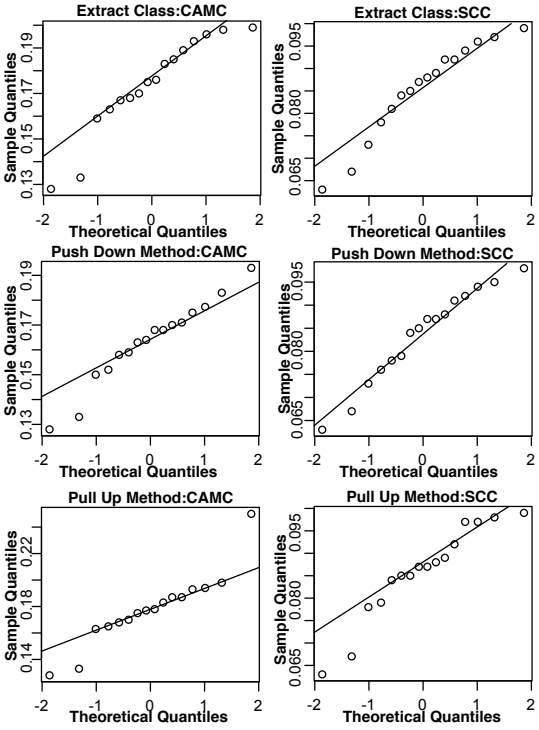


Figure 2. Normal probability plots.

test. In our context, to apply the t-test for CAMC and SCC: (1) the difference between the cohesion values before and after the change has to be calculated, (2) the means and standard deviation values of these differences have to be obtained, and (3) the t-value has to be calculated. We applied the typical significance threshold ($\alpha = 0.05$) to decide whether the differences between the cohesion values were significant.

The results in Tables IV and V lead us to the following observations about the refactoring **ExtractClassUnit**.

The results before and after applying the refactoring **ExtractClassUnit** demonstrate that the cohesion values for the metric CAMC were significantly different. As can be seen in Table V the means before and after applying the refactoring **ExtractClassUnit** were 0.164125 and 0.183625, respectively. The corresponding critical T value was -2.6139, the standard deviation for this refactoring was 0.0211, DOF $= 7$, two-tailed p-value $=> 0.0347$, with 95% confidence intervals (upper $= 0.1817$ and lower$= 0.1465$). Interpreting these data we can remark that the difference in regarding to the metric CAMC for the refactoring **ExtractClassUnit** are significantly different. Therefore, we can conclude with 95% confidence (or a chance of error of 5%) that the refactoring **ExtractClassUnit** helped to increase the cohesion amongst the methods of a class of the evaluated systems. Regarding the results of the metric SCC after applying the refactoring **ExtractClassUnit** we could validate that the evaluated systems had significant change. As can be seen in Table V the T value was -2.5616, the standard deviation for this refactoring was 0.0106, DOF $= 7$, two-tailed p-value $= 0.0375$, with 95% confidence intervals (upper $= 0.0894$ and lower$= 0.0716$). Interpreting these data we can observe that the difference in regarding to the metric SCC for the refactoring **ExtractClassUnit** are significantly different. Thus, we can conclude with 95% confidence (or a chance of error of 5%) that the refactoring **ExtractClassUnit** improved in some way the similarity-based class cohesion of the evaluated systems.

Similarly, the results in Tables IV and V lead us to the following observations about the refactoring **PushDownMethodUnit**. The results before and after applying the refactoring **PushDownMethodUnit** demonstrate that the cohesion values for the metric CAMC were not significantly different. As can be seen in Table V the means before and after applying the refactoring **PushDownMethodUnit** were 0.164125 and 0.1624125, respectively. The corresponding critical T value was 0.2862, the standard deviation for this refactoring was 0.0168, DOF $= 7$, two-tailed p-value $= 0.783$, with 95% confidence intervals (upper $= 0.1781$ and lower$= 0.1501$). Interpreting these data according to the t-test we can remark that metric CAMC for the refactoring **PushDownMethodUnit** are not significantly different. Therefore, it was possible to conclude with 95% confidence (or a chance of error of 5%) that the refactoring **PushDownMethodUnit** did not improve the cohesion amongst the methods of a class of the evaluated systems. As for the metric SCC the refactoring **PushDownMethodUnit** demonstrate that the cohesion values are also not significantly different. Although, in Table IV one can point that after applying the refactoring **PushDownMethodUnit** all the evaluated system had been improved in regarding the metric SCC (see column SCC cell Af) - after perform the t-test we could conclude with 95% confidence (or a chance of error of 5%) that the refactoring **PushDownMethodUnit** did not raise the similarity-based class cohesion of the evaluated systems.

Finally, the results in Tables IV and V lead us to the following observations about the refactoring **PullUpMethodUnit**. Similarly as the refactoring **PushDownMethodUnit** the refactoring **PullUpMethodUnit** demonstrate that the cohesion values for the metric CAMC were not significantly different. As can be seen in Table V the means before and after applying the refactoring **PullUpMethodUnit** were 0.164125 and 0.192, respectively. The corresponding critical T value was -0.1945, the standard deviation for this refactoring was 0.4057, DOF $= 7$, two-tailed p-value $= 0.8513$, with 95% confidence intervals (upper $= 0.5033$ and lower$= -0.1751$). Although it is fairly evident that the refactoring **PullUpMethodUnit** improved in some way the evaluated systems (see columns CAMC cell Af) we cannot prove statistically that it really improved the evaluated systems. Thus, statistically we can remark with 95% confidence (or a chance of error of 5%) that the refactoring **PullUpMethodUnit** did not improve the cohesion amongst the methods of a class of the evaluated systems. Regarding the metric SCC it was not possible to find a significant difference between the cohesion before to apply the refactoring **PullUpMethodUnit** versus the cohesion after to apply the refactoring **PullUpMethodUnit**. In Table V

Table V
T-TEST RESULTS.

T-TEST	Extract ClassUnit				Push Down MethodUnit				Pull Up MethodUnit			
	CAMC		SCC		CAMC		SCC		CAMC		SCC	
	MEAN				MEAN				MEAN			
	0.164125	0.183625	0.0805	0.090125	0.164125	0.1624125	0.0805	0.086625	0.164125	0.192	0.0805	0.091375
	t-value = -2.6139		t-value = -2.5616		t-value = 0.2862		t-value = -1.7083		t-value = -0.1945		t-value = -0.1949	
	SD = 0.0211		SD = 0.0106		SD = 0.0168		SD = 0.0101		SD = 0.4057		SD = 0.1567	

the means before and after applying applying the refactoring **PullUpMethodUnit** were 0.0805 and 0.091375, respectively. The corresponding critical T value was -0.1949, the standard deviation for this refactoring was 0.1567, DOF $= 7$, two-tailed p-value $= 0.851$, with 95% confidence intervals (upper $= 0.2115$ and lower$= -0.0505$). Therefore, by interpreting these data we can draw the conclusion with 95% confidence (or a chance of error of 5%) that the refactoring **PullUpMethodUnit** did not improve the similarity-based class cohesion of the evaluated systems.

E. Threats to Validity

The lack of representativeness of the subject programs may pose a threat to external validity. We argue that this is a problem that all software engineering research, since we have theory to tell us how to form a representative sample of software. Apart from not being of industrial significance, another potential threat to the external validity is that the investigated programs do not differ considerably in size and complexity. To partially ameliorate that potential threat, the subjects were chosen to cover a broad class of applications. Also, this experiment is intended to give some evidence of the efficiency and applicability of our implementation solely in academic settings. A threat to construct validity stems from possible faults in the implementations of the techniques. With regard to our catalogue of refactoring, we mitigated this threat by running a carefully designed test set against several small example programs. Similarly, all the eight open source Java projects used in this experiment have been extensively used within academic circles, so we conjecture that this threat can be ruled out.

VI. RELATED WORK

In [7] an approach to specify generic refactorings is presented. Here, Moha et al. introduce a meta-metamodel (GenericMT), which enables the definition of generic refactorings on the Meta Object Facility (MOF) layer. This meta-metamodel contains structural commonalities of object-oriented models (e.g., classes, methods, attributes and parameters). Generic refactorings are then specified on top of the GenericMT.

Borger et al. [8] developed a plug-in for the CASE tool ArgoUML that support UML model-based refactorings. The refactoring of class, states and activities is possible, allowing the user to apply refactorings that are not simple to apply at source-code level.

VII. CONCLUSIONS

We adapted the traditional notion of fine-grained refactoring to the KDM specification. As far as we know, this paper is the first one to define one catalogue of refactoring for KDM. We argue that devising a catalogue of refactoring for KDM makes it be both language-independent and standardized. To provide some evidence of our catalogue of refactoring, we conducted an experiment using eight open source Java application. More specifically, for these eight application we applied three different refactorings - **Extract ClassUnit**, **Push Down MethodUnit**, and **Pull up MethodUnit**. Experimental results show that the our catalogue of refactoring improved the legacy system.

As a future work, we aim at applying our catalogue in more case studies.From these case studies we can propose more fine-refactorings for KDM. We also aim to create macro-refactorings for the KDM in order to explore the role of configuration knowledge for achieving model-driven refactoring for KDM. Previously, we devised an approach for identifying concerns in KDM models [9]. Therefore, the next step is to create refactorings that take into account Crosscutting Frameworks [10].

ACKNOWLEDGMENTS

Rafael S. Durelli would like to thank the financial support provided by FAPESP, process number 2012/05168-4.

REFERENCES

[1] S. Demeyer, B. Du Bois, and J. Verelst, "Refactoring - Improving Coupling and Cohesion of Existing Code," *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, 2004.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Agosto 2000.

[3] R. Pérez-Castillo, I. G.-R. de Guzmán, and M. Piattini, "Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems," *Comput. Stand. Interfaces*, pp. 519–532, 2011.

[4] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*, ser. FOSE 07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–54.

[5] R. Perez-Castillo, I. G.-R. de Guzman, O. Avila-Garcia, and M. Piattini, "On the use of adm to contextualize data on legacy source code for software modernization," in *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, ser. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 128–132. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2009.20

[6] W. M. Ulrich and P. Newcomb, *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[7] N. Moha, V. Mahe, O. Barais, and J.-M. Jezequel, "Generic model refactorings," in *MoDELS*, pp. 628–643.

[8] M. Boger and T. Sturn, "Tools-support for model-driven software engineering," *Proceedings of Practical UML-Based Rigorous Development Methods*, pp. 490–496, 2002.

[9] D. Santibáñez, R. S. Durelli, B. Marinho, and V. V. de Camargo, "A Combined Approach for Concern Identification in KDM models," in *Latin-American Workshop on Aspect-Oriented Software Development*, ser. LAWASP '7, 2013.

[10] V. V. Camargo and P. C. Masiero, "An approach to design crosscutting framework families," in *Proceedings of the 2008 AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*. ACM, 2008.

Anexo 7: Comprovante da publicação no *IEEE International Conference on Information Reuse and Integration 2014* (IRI 2014)

- Uma cópia do artigo é apresentado a seguir (a partir da próxima página).

A Mapping Study on Architecture-Driven Modernization

Rafael S. Durelli[†‡], Daniel S. M. Santibáñez[*], Bruno Marinho[*], Raphael Honda[*],
Márcio E. Delamaro[†], Nicolas Anquetil[‡] and Valter Vieira de Camargo[*]
[*]Departamento de Computação, Universidade Federal de São Carlos,
Caixa Postal 676 – 13.565-905, São Carlos – SP – Brazil
Email: {daniel.santibanez, bruno.marinho, raphael.honda, valter}@dc.ufscar.br
[‡]RMoD Team - INRIA, Lille – France
Email: {Nicolas.Anquetil}@inria.fr
[†]Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo,
Av. Trabalhador São Carlense, 400, São Carlos – SP – Brazil
Email: {rdurelli, delamaro}@icmc.usp.br

***Abstract*—Background: Perhaps the most common of all software engineering activities is the modernization of software. Unfortunately, during such modernization often leaves behind artifacts that are difficult to understand for those other than its author. Thus, the Object Management Group (OMG) has defined standards in the modernization process, by creating the concept of Architecture-Driven Modernization (ADM). Nevertheless, to the best of our knowledge, there is no a systematic mapping study providing an overview of how researchers have been employing ADM. Thus, we assert that there is a need for a more systematic investigation of the topics encompassed by this research area. Objective: To describe a systematic mapping study on ADM, highlighting the main research thrusts in this field. Method: We undertook a systematic mapping study, emphasizing the most important electronic databases. Results: We identified 30 primary studies, which were classified by their contribution type, focus area, and research type. Conclusion: This systematic mapping can be seen as a valuable initial foray into ADM for those interested in doing research in this field. More specifically, our paper provides an overview of the current state of the art and future trends in software modernization area, which may serve as a road-map for researchers interested in coming up with new tools and processes to support the modernization of legacy systems.**

***Index Terms*—Systematic Mapping, Architecture-Driven Modernization, ADM, Knowledge Discovery Metamodel, KDM**

I. Introduction

Software systems cannot be simply discarded because they incorporate a lot of invaluable knowledge about their organizations. However, the structure of these software systems starts deteriorating when they undergo maintenance. These systems are considered legacy when their maintenance costs are raised to undesirable levels but they are still valuable assets to their organizations. Given the significant business value of these systems, they must be modernized.

In this context, the Object Management Group (OMG) has defined standards in the modernization process, creating the concept of Architecture-Driven Modernization (ADM). ADM follows the Model-Driven Development (MDD) [1] guidelines and comprises three major steps. First, reverse engineering is performed starting from the source-code. This step yields an instance model named Platform-Specific Model (PSM). Second, successive transformations are applied to this model, up to point at each of these transformations reaches an apt abstraction level. These transformations are represented using a model called Knowledge Discovery Metamodel (KDM). Using this model, several modernization activities, optimizations, and modifications can be performed in order to solve problems found in the legacy system. Third, a forward engineering step is carried out, resulting in a modernized version of the source code of the target system. The idea behind the KDM standard is to provide a common representation, thereby increasing portability by allowing the community to create parsers that turn any representation into KDM. As a result, as long as developers stick to such a standard, everything that takes KDM as input can be considered platform and language-independent. For example, a refactoring catalogue for KDM can be used for refactoring systems implemented in different languages.

In order to gauge and report on the current state of the art and research trends in this field, we carried out a systematic mapping study on ADM. Our motivation to conduct such a systematic mapping study is to identify the topics that have been most investigated as well as the topics that have not received much attention. Although, ADM is a relatively new approach, OMG claims that it is a step towards combining two well-known research fields, (i) MDD and *(ii)* software reengineering. Since ADM has come a long way in the last few years and many efforts have emphasized the modernization of legacy systems through this approach, we assert that there is a need for a more systematic investigation of the topics encompassed by this research area. To the best of our knowledge, this is the first systematic mapping study on ADM, providing an initial foray into the current state-of-the-art of this field.

This paper is structured as follows. Section II describes how the systematic mapping methodology has been conducted. Section III presents the main findings of this study. In Section IV the threats to validity of this study are presented. Finally in Section V concluding remarks are made.

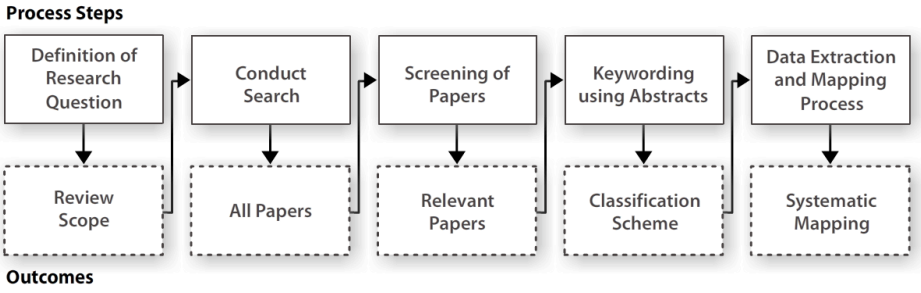


Figure 1. The systematic mapping process (Adapted from [2]).

II. Research Methodology

A mapping study provides a systematic and objective procedure for identifying the nature and extent of the available research relevant to answer a particular research question. Throughout the conduction of this systematic mapping, we followed the all guidelines provided by Petersen et al. [2]. Each step produces an intermediate outcome, the concluding result being the mapping study as shown in Figure 1.

Furthermore, in this paper we have used Visual Text Mining (VTM) technique to support the studies selection [3]. VTM uses text mining algorithms and methods combined with interactive visualisations. Therefore, it can help the user making sense of a collection of primary studies, without actually reading all of them. In this case the studies were reading partially or full. The following sections present details on how we carried out this mapping study.

A. Search Strategy

The review protocol is defined in this step. This protocol contains: *(i)* the research questions (RQs) and *(ii)* the search string.

RQs must embody the mapping study purpose. Thus, given that we set out to give an overview of the current state of the art and research in the ADM field, we formulated three research questions:

- **RQ**$_1$ - Given ADM's standards metamodels, which one has been more used in the literature? In addition, given the identified metamodel, what are the most and least used packages?
- **RQ**$_2$ - What types of studies have been published in the area?
- **RQ**$_3$ - What are the most and least discussed focus areas in the ADM literature? Moreover, what types of contributions have been presented so far?

Afterwards, the search string was defined. The search string was created based upon a set of keywords. Figure 2 shows the search string we used to carry out our systematic mapping.

B. Data Source and Study Selection

The search encompassed electronic databases that are deemed as the most relevant scientific sources [4] and therefore likely to contain important primary studies. We used the search string on the following electronic databases: *ACM*, *IEEE XPLORE*, *Scopus*, *Web of Science* and *Engeneering Village*. It is worth mentioning that since the features provided by these databases, as well as the exact syntax of the search strings to be applied, vary from one database to the other, the string given in Figure 2 was actually used to construct a semantically equivalent string tailored to each database.

("KDM") **OR** ("Knowledge Discovery Metamodel") **AND** ("Knowledge-Discovery Metamodel") **OR** ("Knowledge-Discovery Meta-model") **OR** ("Knowledge Discovery Meta-model") **OR** ("Architecture Driven Modernization") **OR** ("Architecture-Driven Modernization") **OR** ("Model Driven Modernization") **OR** ("Model-Driven Modernization") **OR** ("Model-driven software modernization") **OR** ("Abstract Syntax Tree Metamodel") **AND** ("ASTM") **OR** ("Structured Metrics Metamodel") **OR** ("SMM")

Figure 2. Search string used in our systematic mapping.

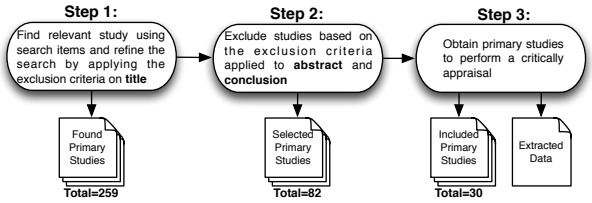


Figure 3. Study Selection Summary.

The search string given in Figure 2 was applied in the digital libraries. An overview of results acquired from these digital libraries is depicted in Figure 3. Moreover, it presents the amount of studies remaining after each step.

In order to determine which primary studies are relevant to answer our research questions, we applied a set of inclusion and exclusion criteria. The inclusion criteria we applied were the following:

- The primary study presents at least one modernization approach that employs ADM;
- The primary study describes an empirical evaluation of an ADM-based approach.

and the following exclusion criteria:

- Papers that mention ADM and its related metamodels only in the abstract;
- Introductory papers for books and workshops;
- The primary study is a short paper (containing up to three pages).

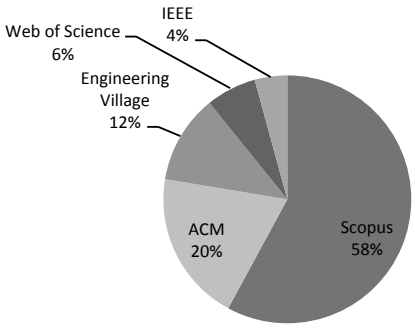


Figure 4. Distribution of primary studies by electronic database.

As can be seen in Figure 4 Scopus was the digital library that returned most primary studies 58% (150). ACM, Engineering Village, Web of Science and IEEE returned 20% (51), 12% (30), 6% (17), and 4% (11), respectively. We surmise that this occurred because Scopus indexes studies from others libraries, such as IEEE and ACM. Summarizing, we obtained 259 primary studies in the first step. After the first step (see Figure 3), 82 papers were selected. We limited the publication venues to international journals and conferences. We applied the aforementioned exclusion criteria to the abstract and conclusion of each primary study. After this step, 229 papers were excluded. So we end up analyzing 30 primary studies.

C. Defining a Classification Scheme

We applied the classification schemes proposed by Petersen et al. [2] and classified the publications into categories from three perspectives: (*i*) **Focus Area**, (*ii*) **Contribution Type** and (*iii*) **Research Type**. The resulting classification schemes are described in the following subsections.

1) Focus Areas: After reading through the primary studies, five focus areas were identified. The first one is **"Software Modernization"**. This focus area is related to primary studies that describe approaches that employ ADM to fully modernize legacy systems either to another platform or architecture. The second focus area is related to **"Business Knowledge Extraction"**, which describes primaries studies on processes, methods, or approaches to extract business-related information of legacy systems. The third one is **"Concern Extraction"** and it comprises primaries studies report on processes, methods, or approaches to extract crosscutting concerns (CC) of legacy systems. The forth one, **"Extension of ADM's Metamodels"**, is concerned with grouping studies taht present approaches, methods, or processes to extend one of the ADM's metamodels. Finally, the last focus area is **"Applicability"** which includes papers that mainly focus on presenting evidence related to applying ADM and its metamodels in practice. In other words, papers that enable researchers and practitioners to get a better understanding and utilization of ADM and its metamodels.

2) Contribution Type: five contribution types were identified. **"Tools"** groups primary studies that focus on providing tools to support the modernization of legacy system by using ADM. The second contribution type is **"Process"**, which refers to primary studies that describe processes to assist the modernization of legacy system by means of ADM. **"Model Transformation"** contains primary studies that describe the use of language transformation such as Query/Views/Transformations (QVT)[1] or ATL Transformation Language (ATL)[2] to realize transformation among the ADM's metamodels. The fourth contribution type is **"Metamodel"**. Studies in this category create or extend the ADM's metamodels to deal with a specific problem as, for instance, providing a KDM light-weight extension in order to either represent the aspect oriented paradigm or supports a component-oriented decomposition. The last contribution type is **"Metrics"** and describes papers that focus on proposing or applying metrics to gauge the effectiveness of ADM and its metamodels.

3) Research Type: The research type reflects the research approach used in the primary study. The research type categories are based on the scheme proposed by Wieringa et al. [5] (RQ_3), as follows: Studies in the **"Validation Research"** aims to examine a solution proposal that has not yet been practically applied. It is conducted in a systematic way and may present any of these: prototypes, math analysis, etc. **"Evaluation Research"** in contrast to validation research, evaluation research aims at examining a solution that has already been practically applied. Studies in this category investigate the practical implementation of the proposed solution and usually present results using empirical strategies (e.g., experiments and study cases). **"Conceptual Proposal"** presents an arrangement to perceive things that already exist, in a novel way. **"Experience paper"** reports on personal experience of the author from one or more real life projects. **"Opinion papers"** report on the personal opinion of the author on suitability or unsuitability of a specific technique or tool.

D. Data Extraction and Synthesis

We elaborated data extraction forms to accurately record the information obtained by the researchers from the primary studies. The form for data extraction provides some standard information, such as a summary of the primary study, highlighting how ADM and its metamodels were used, date when the data extraction took place, title, authors, venue, and a summary of the study's conclusion. During the extraction process, information about each primary study was independently gathered by all reviewers. The review was performed in November, 2013 by three Masters students, a PhD student, and three domain experts.

E. Validation

In validation phase an approach that uses VTM technique and the associated tool - Projection Explorer (PEx) - were applied to support the inclusion and exclusion decisions [3].

Figure 5 presents a document map generated using PEx. This map is composed of 259 primary studies analysed in this review, highlighting them using different shades of gray

[1] http://www.omg.org/spec/QVT/1.1/
[2] www.eclipse.org/atl/

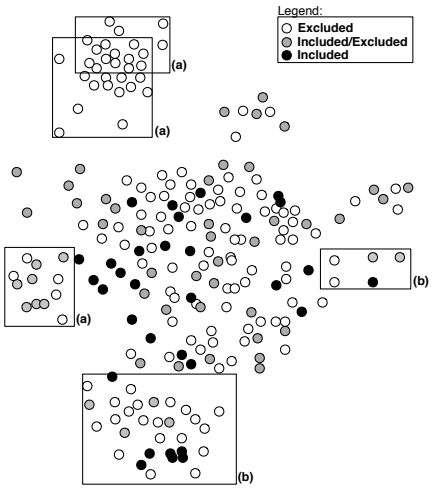


Figure 5. Document map colored with the history of the inclusions and exclusions of the studies.

to differentiate in which of the stages a study was removed from the review. White points are studies excluded in first stage, gray points are the studies excluded in second stage and the black points are the included. The exploration of a document map is conducted in two steps: (*i*) firstly, a clustering algorithm is applied to the document map, creating groups of highly related documents; (*ii*) secondly, the resulting clusters are analysed in terms of: **Pure Clusters** - all documents belonging to a cluster have the same classification (all included or excluded, regardless of exclusion stage). Normally, in this case do not need to be reviewed; and **Mixed Clusters** - which represent documents with different classification on the same cluster. These cases are hints to the reviewer, and the estuaries grouped should be reviewed following the traditional method. To facility the visualisation, in Figure 5 just five clusters generated by PEx are depicted. Examples of pure clusters (all excluded) are identified in Figure 5 using label "(a)" and therefore do not needed to be reviewed. Mixed clusters (clusters containing black (included) and white or gray (excluded) studies) are identified using label "(b)" and they were reviewed by the authors of this paper. At the end, we kept the initial classifications conducted manually, but this technique contributed to a review of studies that could have been wrongly excluded or included previously.

F. Mapping and discussion of research questions

The focus of this section is to present a broad overview of the research on ADM. Apart from creating this overview, we also used the information drawn from the primary studies to answer the research questions.

Instead of using frequency tables we have decided to produce a bubble plot to report the frequencies and distribution of the selected studies according to their categories and publication date, thereby providing a map of research related to ADM. Our resulting map is shown in Figure 7. Bubble plots are essentially two x-y scatter plots with bubbles in category intersections. The size of each bubble is determined by the number of primary studies that have been classified as belonging to the categories corresponding to the bubble coordinates. This visual summary provides a bird's-eye view that enables one to pinpoint which categories have been emphasized in past research along with gaps and opportunities for future research. Figure 7 has three facets: **Contribution Type**, **Focus Area** and **Research Type**. It is worth highlighting that certain primary studies were grouped in more than one category, affecting the frequency count; i.e., the sum of the frequencies shown in each facet can be greater than the total of selected studies presented earlier (30).

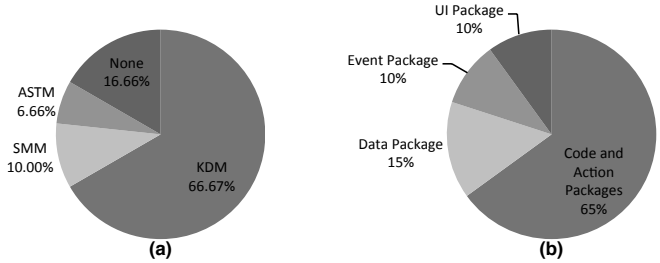


Figure 6. Frequency of ADM's metamodels used in the literature and the most and least used packages.

As for answering the first part of $\mathbf{RQ}_1$ we analyzed all primary studies, focusing on investigating which ADM standard metamodels have more been used in the literature. In Figure 6(a) is depicted a pie chart wherein we plotted the collected data. As can be seen in this figure, KDM seems to be the most used metamodel (66%). A small percentage of primary studies have reported on the use of SMM (10%). While ASTM has been used by a rather small amount of studies (6.66%). We found that 16.66% of the primary studies does not explicitly mention which metamodel has been used during the modernization process. In order to answer the second part of $\mathbf{RQ}_1$ we investigated which are the most and least used packages within the KDM. In Figure 6(b) it is fairly evident that the packages Code and Action are the most used in the literature (65%). We surmise that the reason for this is twofold: (*i*) these packages are often used to represent the source-code of system and since most of the primary studies use the source code as input to start the modernization process; and (*ii*) the absence of a complete parser to instantiate all KDM's layers. The third one most used is the Data package (15%). This package is used to represent relational data, such as databases. As shown in Figure 6(b), the least used packages are Event and UI.

By observing Figure 7 (right side) it is possible to answer $\mathbf{RQ}_2$. The vast majority of the primary studies were classified as **Evaluation Research**, approximately 49%. A small percentage of publications is concerned with **Validation Research** (3.12%). While 30% of the selected studies fall into **Experience Paper** and **Opinion paper**. Finally, **Conceptual Proposals** account for 18% of the selected studies.

In Figure 7 (left side), it can be seen that the majority

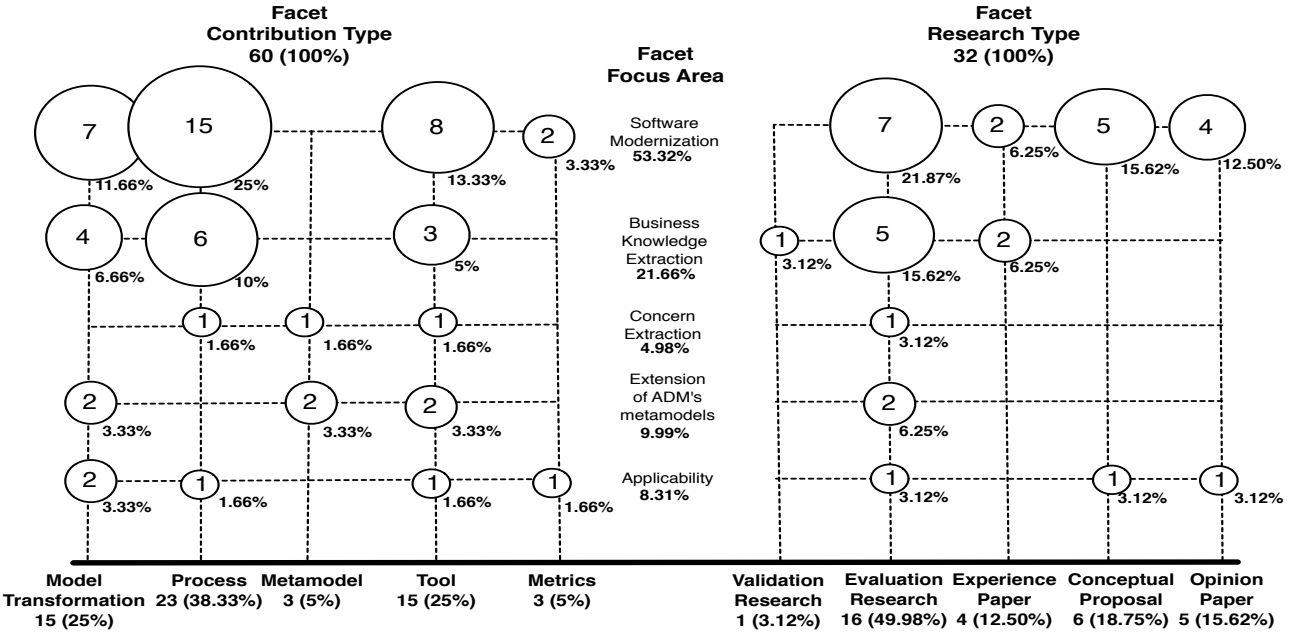


Figure 7. Overview of research on ADM and its metamodels.

of the research papers focus on processes to assist software engineers during the modernization of legacy system. Model transformations and Tools are fields that have also been researched. We believe that these fields have drawn a lot of attention from researchers because most primary studies that describe modernization processes also propose a set of model transformations and a tool that fully or partially automates the proposed process. On the other hand, the contribution type with less studies are **Metamodel** and **Metrics**. Thus, it is argued that primary studies that describe processes to assist the modernization of legacy systems by means of ADM, papers that show a set of rules to be applied during model transformation among the ADM's metamodels (KDM, SMM and ASTM), and papers that devise tools to assist ADM's process can be considered as evidence clusters. In other words, where there may be scope for more complete literature reviews to be undertaken. Whereas metamodels (i.e., papers that explain how to extend ADM's metamodels) and metrics (i.e., papers that describe how to apply metrics in ADM's metamodel) can be regarded as gaps, thus new or better primary studies are required.

In terms of focus area, Figure 7 (middle) shows that previous research has turned much attention to presenting **Software Modernization** (i.e., 53.32%). **Business Knowledge Extraction** has also been significantly covered, 21.66%. While **Concern Extraction**, **Extension of ADM's metamodels** and **Applicability** have been presented collectively by rather small percentage of 25%. As result of this analysis we partially answered $\mathbf{RQ}_3$. We highlighted the main types of contributions that have been proposed in the ADM literature so far. We organized the other part of the analysis related to $\mathbf{RQ}_3$, i.e., (a discussion about the focus area regarding the ADM) in subsections. Each subsection briefly describes the studies selected for each **Focus Area** while highlighting the extent and nature of research.

*1) **Software Modernization**:* Jorge Maratalla et al., propose **GAFEMO** [6], which aims to modernize a legacy systems to the service-oriented approach taking advantage of the features provided by gap-analysis techniques. This approach takes as input a legacy system and then creates KDM representations of it. Afterwards, a set of rules are applied in this model to create the services.

In [7] the authors propose a modernization approach for the modernization of Data warehouses following the concepts of ADM. The approach automatically performs the following tasks: (*i*) obtain a logical representation of data sources (*ii*) mark this logical representation with MD concepts, and (*iii*) derive a conceptual MD model from the marked model. In [8] is defined an approach that is focused on the analysis of legacy systems to discover and create functionalities to be exposed as services using Web Services by means of ADM. It is based in five steps: (*i*) Database reverse engineering: database schema is reversed and a suitable model is built; (*ii*) First service extraction: based on the structure of the database schema, a first service extraction can be undertaken; (*iii*) PIM generation: is obtained from the PSM representation using a model-to-model transformation, CRUD operations are automatically created; (*iv*) Service discovering: abstract objects are identified in the PIM; (*v*) WSDL (Web Service Description Language) generation: using the PIM, a model-to-model transformation and a WSDL metamodel are generated to expose the services discovered and created in the PIM and the PSM.

In [9], [10] is proposed an approach based on ADM named CloudMIG that aims at supporting SaaS (Software as a Service) providers to semi-automatically migrate legacy software systems to the cloud. It is composed of six major steps: (*i*) Extraction: Includes the extraction of architectural and utilization models of the legacy system, the approach uses KDM; (*ii*) Selection: Select an appropriate CEM- compatible cloud profile candidate; (*iii*) Generation: Produces the target architecture and a mapping model; (*iv*) Adaptation: The adaptation activity enables a reengineer to manually adjust the target architecture; (*v*) Evaluation: Realize static analyses and a runtime simulation of the target architecture; (*vi*) Transformation: The actual transformation of the existing system from the generated target architecture to the aimed cloud environment. In [11] the authors propose an approach that uses ADM which is focused on the analysis of legacy systems to discover and create functionalities to be exposed as services using Web Services.

Pérez-Castillo et al., [12]–[14] present approaches to modernize legacy systems together with the legacy relational database. This approach recovers the code-to-data linkages and obtains three kinds of models according to the ADM approach: (*i*) The KDM Code Model, which represents the inventory of legacy source code. It has also the points that link the SQL Sentence Models and Database Schema Models. (*ii*) The SQL Sentence Model for modeling a certain SQL query that was embedded in legacy source code. (*iii*) The Database Schema Model, which represents the specific database fragment derived by an SQL Sentence Model. In [15] presents the XIRUP modernization methodology, which proposes a highly iterative process. This process is feature-driven, component-based, focused on the early elicitation of key information, and relies on a ADM.

Mainetti et al., [16] present an approach that allows developers to automatically modernize the client side of legacy systems. In this approach developers can refactor the Graphical User Interface (GUI) of legacy systems during the modernization, taking the opportunities offered by novel interaction paradigms, i.e., Rich Internet Application (RIA).

In [17] the authors present an approach for the definition of a systematic process for Web Applications (WA) to RIA modernization, by applying ADM principles. The approach presented by the authors consists on generating a RIA client from the legacy WA presentation and navigation layers and its corresponding service-oriented connection layer with the underlying business logic at server side. Boussaidi et al., [18] propose an approach that makes use of the KDM to reconstruct and document software architectural views of the legacy system. They consider an architectural view to be a way of partitioning a system using a specific set of KDM relevant concepts and relations and they propose clustering algorithms that target specific views mainly a layered view that we call horizontal view and a feature based view that we call vertical view. In [1] ADM is used into practice by building a modernization tool to generate metric reports of legacy Oracle Forms applications to assess migration efforts. The authors devised an extractor that generates KDM models from PL-SQL code (PL/SQL-to-KDM) and a metrics report generator for these KDM models.

2) ***Business Knowledge Extraction****:* Pérez-Castillo et al., [19]–[22] present an approach to recover business processes from legacy systems. This approach is based on a set of transformation: (*i*) transformation obtains PSM models from each legacy software artifact using a specific metamodel for each artifact; the traditional reverse engineering techniques such as static analysis, dynamic analysis, and formal concept analysis can be used to extract the needed knowledge; (*ii*) a set of model transformations to obtain a KDM model built from the PSM models at (*i*); (*iii*) a transformation finally obtains the current business process model, this transformation is based on a set of business patterns. In [23] the authors report the results of a family of case studies that were performed to empirically validate this approach. Pérez-Castillo et al., also provides in [24] a semi-automatic technique based on dynamic analysis, combined with static analysis to instrument the source code for obtaining event log models. A set of model transformation to transform the event log into another model following the KDM to depict legacy system, concerning its runtime viewpoint, which can be used in any software modernisation project. In [22] Pérez-Castillo et al., explain the KDM2BPMN model transformation within MARBLE, an ADM-based framework to rebuilt business processes embedded in legacy systems in order to facilitate and improve the evolutionary maintenance.

Normantas and Vasilecas [25] present an approach that facilitates software comprehension by enabling traceability of business rules and business scenarios in software system, i.e., their approach aim to extract business specific knowledge from the knowledge about the existing software system represented within the KDM. Ropero et al., [26] describes a set of rules to transform Mining XML (MXML) metamodel, which is common used to represent the sequence of business activities executed by an enterprise system to KDM. The authors takes an MXML model and obtains an equivalent KDM model at the same abstraction level. The proposed set of rules consist of eight declarative transformation rules.

3) ***Concern Extracting****:* Santibáñez et al., [27] propose an approach called CCKDM for identifying crosscutting concerns by means a combination of a concern library and a K-means clustering algorithm. The input of the approach is a KDM model instance and the result is the same KDM model with annotated concerns. According to the authors this is the first work in concern mining area that use a standardized model in the context of ADM to perform search of concerns. They also believe that ADM standards will be widely used in a near future because is an OMG initiative.

4) ***Extension of ADM's metamodels****:* We identified three papers that address how to perform extension of ADM's metamodels. We provided a brief summary of these paper are follows.

In [28] the authors propose the COMO (Component-Oriented MOdernization) metamodel an KDM's extension, by borrowing recurring concepts from component-based solutions

and software architectures, and to support a proper componentization of the system to assist the modernization of legacy systems. In [24] propose an extension to the KDM that aims to represent all the information registered in a MXML model in the KDM model. In [29] the author proposes an extension of the KDM to represent all elements of the Aspect Oriented Paradigm, i.e., aspect, advice, point-cut, can be represented using the KDM. All these three paper have in common is that the authors claimed the impact of these extensions on well-proven and KDM based tools is not problematic since they are performed with the own extension mechanism of the KDM standard.

*5) **Applicability**:* Pérez-Castillo et al., [21], [24], [30] present how to apply KDM to modernize legacy systems. Also, the authors described each layer of the metamodel KDM, they also presented a set of example of how to use ADM and KDM during the modernization of a legacy systems. The authors claim that the paper enables researchers and practitioners to get a better understanding KDM.

III. Main Findings and Open Issues

Recent proposals in ADM have focused mainly on providing approaches to modernize legacy system to another platform/architecture. However, if we look at overall problem of the integration of modernization into an ADM context, there is still room for improvement. For instance, in order to integrate ADM's metamodels into larger context, the area of discovering knowledge, i.e., parsers needs more attention along with solution to verification of models. Few efforts (e.g., [1], [31]) have dealt with devising parsers to represent instances of KDM, but even these parsers provide limited infrastructure to represent all KDM's layers. Thus, new efforts must be conducted to create a more effective parsers able to represent all KDM's layer. Besides, the processes to discovery of knowledge are often mostly static in a sense that these parsers are unable to obtain knowledge during the executing of the target legacy system. Hence, further research is required to support the discovery of knowledge dynamically.

Another issue is that although KDM had been created to support modernization of legacy systems, the original version of the KDM does not contains metaclasses suitable for representing, for instance, Aspect Oriented Programming concepts; making it difficult to conduct a modernization process whose goal is to remodularize crosscutting concerns. In order to overcome the aforementioned limitation, in [32] we devised a heavyweight extension for KDM called KDM-OA. The goal is to create an extension that allows representing as high-level as low-level AO details, but still respecting the language and platform independence offered by KDM. As result it is possible to apply modernization based on Crosscutting Framework Families [33].

Also, we observed that there are three main hurdles that demand more research so that modernization techniques can be used in the ADM approach in an effective way. The first hurdle is the present lack of a fully developed idea of "good" KDM style. This is an important issue, for a clear notion of style is a fundamental prerequisite for the use of modernization, enabling software engineers to see where they are heading when modernizing their legacy system with KDM. Fowler et al. [34] advocated a specific notion of style for Object-Oriented Programming through a catalog of 22 code smells, compounded by a catalog of 72 refactorings through which those smells can be removed from existing code. The second one – both a cause and a consequence of the first – is the lack of a KDM equivalent of such catalogues. We assume that the process of modernization by using KDM would equally benefit from KDM specific catalogues of smells and refactorings, helping software engineers to detect situations where the KDM could be improved, guiding them through the corresponding transformation processes. The third hurdle is the absence of a tool that supports refactoring by using KDM specification. The catalogue presented by Fowler et al. [34] provides a basis upon which developers can rely on to build tool support for object-oriented refactoring: a similar catalogue for the KDM specification is likely to bring similar benefits to assist software engineers during the modernization process.

IV. Threats to Validity

Primary studies selection: Aiming at ensuring an unbiased selection process, we defined research questions in advance and devised inclusion and exclusion criteria we believe are detailed enough to provide an assessment of how the final set of primary studies was obtained. However, we cannot rule out threats from a quality assessment perspective because we selected studies without assigning any scores.

Missing important primary studies: We conducted the review in several search engines. Nevertheless, it is possible that we some primary studies were left out of our selection. We mitigated this threat by selecting search engines that have been regarded as the most relevant scientific sources [4].

Reviewers reliability: The reviewers of this study are researchers in the software reuse field. So, we are not aware of any bias we may have introduced during the analyses.

Data extraction: Another threat for this review refers to how data was extracted from the digital libraries, since not all the information was obvious to answer the questions and some data had to be interpreted. In order to ensure the validity, multiple sources of data were analyzed, i.e. papers and technical reports. In the event of a disagreement between the two primary reviewers, a third reviewer acted as an arbitrator to ensure that full agreement was reached.

V. Concluding Remarks

Research in the area of ADM can lead to advances in modernization of software systems, resulting in software systems that are more maintainable, extensible, and reusable. To get an overview of the current research in this area we carried out a systematic mapping. After examining 30 primary studies, we answered three research questions.

We have found that the most used ADM's standard metamodel is KDM, which is used in approximately 66% of the primary studies. Also, we found that the most used packages

are Code and Action, which was used in around 65% of the primary studies. We found that most papers in this area fall into the **Evaluation Research** category, 49.98% of the selected studies. A small percentage of publications is concerned with **Validation Research** (3.12%) Papers that fall into the **Experience** and **Opinion** categories account for 30% of all selected papers, and **Conceptual Proposals** account for 18%. Concerning the third research question, we found that the main contributions types are as follows: (1) Process, (2) Model Transformation, (3) Tolls, (4) Metamodels and (5) Metrics.

Another contribution of this paper is the map we created. By observing it is possible to ascertain the extent and form of literature related to ADM, thereby identifying which categories have been emphasized in past research, gaps, and possibilities for future research. Furthermore, it provides additional insight into the frequencies of publication over time.

We confined our analysis mainly to the extent of the evidence available, rather than the content. Thus, as a longer-term future work, we intend to carry out systematic reviews in order to pinpoint the state of evidence in the most prominent categories.

ACKNOWLEDGMENT

Rafael S. Durelli would like to thank the financial support provided by FAPESP, process number 2012/05168-4. We also thanks Vinicius Durelli for his proof-reading of this paper.

REFERENCES

[1] J. Izquierdo and J. Molina, "An architecture-driven modernization tool for calculating metrics," *Software, IEEE*, 2010.

[2] K. Petersen, R. Feldt, and M. Mattsson, "Systematic mapping studies in software engineering," in *12th Inter. Conf. on Evaluation and Assessment in Software Engineering*, 2008.

[3] V. Malheiros, E. Hohn, R. Pinho, M. Mendonca, and J. C. Maldonado, "A visual text mining approach for systematic reviews," in *First Inter. Symposium on Empirical Software Engineering and Measurement*, 2007.

[4] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering," *Information and Software Technology*, 2009.

[5] R. Wieringa, N. Maiden, N. Mead, and C. Rolland, "Requirements engineering paper classification and evaluation criteria," 2005.

[6] J. Moratalla, V. de Castro, M. Sanz, and E. Marcos, "A gap-analysis-based framework for evolution and modernization: Modernization of domain management at red.es," in *SRII Global Conf.*, 2012.

[7] J.-N. Mazón and J. Trujillo, "A model driven modernization approach for automatically deriving multidimensional models in data warehouses," in *26th Inter. Conf. on Conceptual modeling*, 2007.

[8] I. G.-R. d. Guzman, M. Polo, and M. Piattini, "An adm approach to reengineer relational databases towards web services," in *14th Work. Conf. on Reverse Engineering*, 2007.

[9] S. Frey and W. Hasselbring, "An extensible architecture for detecting violations of a cloud environment's constraints during legacy software system migration," in *Software Maintenance and Reengineering, 15th European Conf. on*, 2011.

[10] S. Frey, W. Hasselbring, and B. Schnoor, "Automatic conformance checking for migrating software systems to cloud infrastructures and platforms," *Journal of Software: Evolution and Process*, 2012.

[11] I. Garcia-Rodriguez de Guzman, "Pressweb: A process to reengineer legacy systems towards web services," in *Reverse Engineering, 14th Working Conf. on*, 2007.

[12] R. Perez-Castillo, I. Garcia Rodriguez de Guzman, M. Piattini, and M. Piattini, "On the use of adm to contextualize data on legacy source code for software modernization," in *16th Work. Conf. on Reverse Engineering*, 2009.

[13] R. P. del Castillo, I. García-Rodríguez, and I. Caballero, "Preciso: A reengineering process and a tool for database modernisation through web services," in *ACM SAC*, 2009.

[14] R. Pérez-Castillo, I. G. R. de Guzmán, and M. Piattini, "Database schema elicitation to modernize relational databases," 2012.

[15] R. Fuentes-Fernandez, J. Pavon, and F. Garijo, "A model-driven process for the modernization of component-based systems," *Science of Computer Programming*, 2012.

[16] L. Mainetti, R. Paiano, and A. Pandurino, "Migros: a model-driven transformation approach of the user experience of legacy applications," in *12th Inter. Conf. on Web Engineering*, 2012.

[17] R. Rodríguez-Echeverría, P. J. Clemente, J. C. Preciado, and F. Sanchez-Figueroa, "Modernization of legacy web applications into rich internet applications," in *11th Inter. Conf. on Current Trends in Web Engineering*, 2012.

[18] G. Boussaidi, A. Belle, S. Vaucher, and H. Mili, "Reconstructing architectural views from legacy systems," in *Reverse Engineering, 19th Work. Conf. on*, 2012.

[19] R. Pérez-Castillo, B. de Guzmán, Weber, and A. S. Places, "An empirical comparison of static and dynamic business process mining," in *ACM Symposium on Applied Computing*, 2011.

[20] R. Perez-Castillo, M. Fernandez-Ropero, I. Garcia-Rodriguez de Guzman, and M. Piattini, "Marble. a business process archeology tool," in *27th IEEE Inter. Conf. on Software Maintenance*, 2011.

[21] R. Perez-Castillo, I. Garcia-Rodriguez de Guzman, and M. Piattini, "Mimos, system model-driven migration project," in *17th European Conf. on Software Maintenance and Reengineering*, 2013.

[22] R. Pérez-Castillo, I. G.-R. De Guzmán, and M. Piattini, "Implementing business process recovery patterns through qvt transformations," in *Inter. Conf. on Theory and practice of model transformations*. Springer-Verlag, 2010, pp. 168–183.

[23] R. Perez-Castillo, J. A. Cruz-Lemus, I. G.-R. de Guzman, and M. Piattini, "A family of case studies on business process mining using marble," *Systems and Software*, 2012.

[24] R. Pérez-Castillo, I. G.-R. de Guzmán, M. Piattini, and B. Weber, "Integrating event logs into kdm repositories," in *27th Annual ACM Symposium on Applied Computing*, 2012.

[25] K. Normantas and O. Vasilecas, "Extracting business rules from existing enterprise software system," in *Information and Software Technologies*, ser. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2012.

[26] M. Fernández-Ropero, R. Pérez-Castillo, B. Weber, and M. Piattini, "Empirical assessment of business model transformations based on model simulation," in *Inter. Conf. on Theory and Practice of Model Transformations*, 2012.

[27] D. Santibáñez, R. S. Durelli, B. Marinho, and V. V. de Camargo, "A Combined Approach for Concern Identification in KDM models," in *Latin-American Workshop on Aspect-Oriented Software Development*, ser. LAWASP '7, 2013.

[28] L. Baresi and M. Miraz, "A component-oriented metamodel for the modernization of software applications," in *Engineering of Complex Computer Systems, 16th IEEE Inter. Conf. on*, 2011.

[29] P. M. Shahshahani, "Extending the knowledge discovery metamodel to support aspect- oriented programming," 2011.

[30] R. Perez-Castillo, I. G.-R. de Guzman, and M. Piattini, "Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems," *Computer Standards & Interfaces*, vol. 33, no. 6, pp. 519 – 532, 2011.

[31] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "Modisco: a generic and extensible framework for model driven reverse engineering," in *IEEE/ACM Inter. Conf. on Automated software engineering*, 2010.

[32] B. M. Santos, R. R. Honda, V. V. Camargo, and R. Durelli, "KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel ," in *28th Brazilian Symposium on Software Engineering (SBES),*, 2014.

[33] V. V. Camargo and P. C. Masiero, "An approach to design crosscutting framework families," in *Proceedings of the 2008 AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*. ACM, 2008.

[34] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

Anexo 8: Comprovante da publicação no *28th Brazilian Symposium on Software Engineering* (SBES 2014)

- Uma cópia do artigo é apresentado a seguir (a partir da próxima página).

KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel

Bruno M. Santos, Raphael R. Honda,
Valter V. de Camargo
Departamento de Computação
Universidade Federal de São Carlos - UFSCar
São Carlos – SP – Brazil
{bruno.santos, raphael.honda, valter}@dc.ufscar.br

Rafael S. Durelli
Instituto de Ciências Matemáticas e Computação
Universidade de São Paulo - USP
São Carlos – SP – Brazil
rsdurelli@icmc.usp.br

***Abstract*— Architecture-Driven Modernization is the new generation of software reengineering. The main idea is to modernize legacy systems using a set of standard models. The first step is to obtain, by reverse engineering, an instance of an ISO metamodel called KDM that represents all details of the legacy system. Then, refactorings and optimizations can be applied over this model turning it into a target/modernized KDM. Afterwards the source code of the target system can be generated. In its original form, KDM does not provide aspectual concepts, preventing an aspect-oriented modernization to be properly conducted. In this paper we present KDM-AO, an aspect-oriented heavyweight extension for the KDM metamodel. The extension has been created based on a well known aspect-oriented profile for AspectJ language. To evaluate our extension, we applied it in an aspect-oriented modernization whose goal was to remodularize the persistence concern of an application using a Persistence Crosscutting Framework. The case study showed that KDM-AO is able to represent high-level and low-level aspect-oriented abstractions.**

***Keywords*— KDM profile; Architecture-Driven Modernization; KDM; aspect-oriented modernization; Crosscutting Frameworks**

I. INTRODUCTION

Systems are termed as "legacy" when their maintenance and evolution cost increasingly rise to unbearable levels, but they still deliver great and valuable benefits for companies. In order to make information systems continue satisfying their previously established requirements, they need to be continuously evolved or they probably will fail in fulfilling their goals. Many companies have systems that suffer the phenomena of erosion and aging. These phenomena are result of successive changes systems suffer along years of maintenance, for example, functionalities that were removed, modified or added; hence compromising their overall quality [1][4].

In 2003 the Object Management Group (OMG) created a task force called Architecture Driven Modernization Task Force (ADMTF). It was aimed to analyze and evolve typical reengineering processes, formalizing them and making them supported by models [2]. ADM advocates the conduction of reengineering processes following the principles of Model-Driven Architecture (MDA) [22][2], i.e., all the software artifacts considered along with the process are models.

According to OMG the most important artifact provided by ADM is the Knowledge Discovery Metamodel (KDM). By using KDM, it is possible to represent all system's artifacts, such as configuration files, graphical user interfaces, architectural views and source-code details. The idea behind KDM is to motivate the community to start creating parsers and tools that work over KDM instances; thus, every tool that takes KDM as input can be considered platform and language-independent. For instance, a refactoring catalogue for KDM can be used for refactoring systems implemented in different languages [32]. One of the primary uses of KDM is during reverse engineering processes, in that a parser reads source-code of a system and generates a KDM instance representing it. After that, refactorings and otimizations can be performed over this model, aiming to solve previously identified problems.

Whenever one decides to modernize legacy systems aiming to remodularize concerns, a candidate paradigm is Aspect-Orientation (AO), which provides abstractions to improve the modularization of crosscutting concerns [29]. Although ADM/KDM had been created to support modernization of legacy systems, the original version of the KDM does not contains metaclasses suitable for representing AOP concepts [32]; hampering modernization processes whose goal is to remodularize crosscutting concerns [3].

A possible alternative is to extend KDM using a lightweight solution (Profiles) that is based on set of stereotypes and tag definitions. Profiles are able to impose restrictions on existing metaclasses, respecting the metamodel. However, the lightweight extension mechanism provided by KDM does not guarantee type checking in models, transferring all that responsibility for tools and Software Engineers.

In order to overcome the aforementioned limitation, in this paper we present a heavyweight extension for KDM called KDM-AO. Heavyweight extensions are based on a modified KDM metamodel, including new metaclasses or changing the existing ones. The goal was to create an extension that allows representing both high-level as low-level details, but still respecting the language and platform independence offered by KDM. One important characteristic of our heavyweight extension is that we have not changed the existing KDM metaclasses, we had just added new ones. Therefore, it can be easily incorporated in existing KDM tools.

978-1-4799-4223-7/14 $31.00 © 2014 IEEE
DOI 10.1109/SBES.2014.20

Our KDM-AO is totally based on an existing UML profile for creating class diagrams with AO stereotypes proposed by Evermann [8]. However, although Evermann´s profile is specific to class diagrams, when its stereotypes are mapped to the KDM, the KDM extension inherits all infrastructure available for this metamodel, allowing one to represent all static and dynamic details of a system. To support the creation of KDM-AO instances, we have also created an Eclipse plug-in to facilitate this process.

Another contribution of this paper is to present a preliminary mapping between UML metamodel and KDM metamodel. This mapping is a conceptual tool for converting UML profiles in KDM extensions. The success of modernization processes is heavily dependent on the abstractions which are possible to be represented in KDM. As most of the abstractions of recent domains (web services, embedded systems, aspects, business processes, cloud, etc) are not presented in KDM in an explicit way, we consider the conversion of UML profiles in KDM extensions (either heavy or lightweight) an important activity.

In order to assess our KDM-AO we carried out a Crosscutting Framework-based modernization process in a management system of a CD Shop [3]. The evaluation showed that KDM-AO is able to represent all the details inherent in this type of framework, as well as all AO concepts. In addition, the results show that by using the KDM-AO is possible to modernize a legacy system to AOP. However, it is beyond our scope the forward engineering of the system.

This paper is structured as follows: Section II shows background about ADM/KDM and Aspect-Oriented KDM. In Section III the Aspect-Oriented KDM is described. A case study is shown in Section IV. The related works are shown in Section V. Finally, in Section VI, the discussions and conclusions are presented.

II. BACKGROUND

A. ADM/KDM

In 2003, OMG initiated efforts to standardize the process of modernization of legacy systems using models by means of the ADMTF [2]. The aim of the ADM is the revitalization of existing applications by adding or improving functionalities, using existing OMG modeling standards and also considering MDA principles. In other words, the OMG through ADMTF took the initiative to standardize reengineering processes.

According to ADM [2], ADM does not replace reengineering, but improves it through the use of MDA. The basic process flow of modernization has three phases: Reverse engineering, restructuring and forward engineering. In the reverse engineering, the knowledge is extracted and a Platform-Specific Model (PSM) is generated. The PSM model serves as the basis for the generation of a Platform Independent Model (PIM) called KDM. Then this PIM can serve as basis for the creating of a Computing Independent Model [2].

In order to support the modernization process, in 2006 the KDM metamodel was created. It can be used to represent the system and their operating environments. KDM is language and platform-independent, i.e., a PIM that is able to represent physical and logical artifacts of legacy systems at different levels of abstraction. KDM contains twelve packages and it is structured in a hierarchy of four layers: (i) Infrastructure Layer, (ii) Program Elements Layer, (iii) Runtime Resource Layer and (iv) Abstractions Layer [2]. These layers are created automatically, semi-automatically or manually through the application of various techniques of extraction of knowledge, analysis and transformations [5]. Fig. 1 depicts the architecture of KDM. By observing this figure it is fairly evident that each layer is based on the previous layer, thus, they are organized into packages that define a set of metamodel, whose purpose is to represent a specific and independent interest of knowledge related to legacy systems [2].

Herein, we are especially interested in the Program Elements Layer because it defines the Code package which is widely used by our extension. The Code package possesses a set of metaclasses to represent program elements in implementation level. In other words, this package contains a set of metaclasses that represent the common named elements in the source code supported by different programming languages such as data types, classes, procedures, methods, templates and interfaces [6].

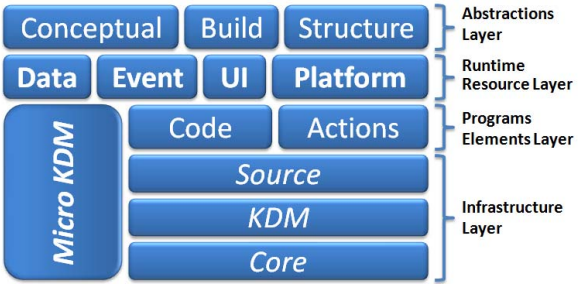


Figure 1. KDM Architecture [2] (Adapted).

As in UML, it is also possible to define either lightweight or heavyweight extension in KDM by means of extension mechanism. Heavyweight extensions are based on a modified KDM metamodel, including new metaclasses or changing the existing ones. On the other hand, lightweight extensions (also known as profiles) are based on set of stereotypes, tag definitions, and constraints, which are basically "notes" over the model. Profiles are able to impose restrictions on existing metaclasses, but they respect the metamodel, without modifying the original semantics of the elements. One of major benefits of profiles is that they can be handled in a natural way by existing tools.

In general, the drawback of heavyweight extensions is that existing tools get no longer compatible with the new metamodel. However, the only way to guarantee model correctness in model level is using heavyweight extensions. This happens because it is possible to relate metamodel elements by their types and not just by their names, as it usually happens in lightweight extensions. Using lightweight extensions, the correctness of the model must be guaranteed by tools. Besides, when heavyweight extensions do not change the original metamodel (just adding new ones), it acts like a

lightweight one, as the extended part can be make available as an independent module and can be easily incorporated in existing KDM tools.

Another important and interesting point here is the following. KDM is not a metamodel intended to serve as base for diagrams, like UML. While UML instances are usually created by humans, KDM instances are system representations created by parsers and processed by tools. So, lightweight profiles make much more sense in the context of UML than in the context of KDM.

B. *Aspect-Oriented Profile*

The main decision before the creation of KDM-AO was to choose an UML profile which was broad enough to represent all the AO concepts. In this sense, we conducted a literature review to identify Aspect-Oriented metamodels and UML profiles that could be considered good candidates. We had analyzed several proposals [10] [11] [13] [14] [15] [16] [17] [18] [19] [31], but the Evermann's profile was considered the most suitable one because it incorporates the level of details that we are interested in [8].

Although this profile has been primarily proposed for AspectJ language, it incorporates all the AO generic concepts. Observing aspect-oriented languages like AspectJ, AspectC++ and AspectS, it is possible to notice that Evermann´s profile involves all of the details presented in these languages, obviously using different terminology. It is like a superset for aspect-oriented programming. This is not a problem because if we want to represent an AspectS program using Evermann´s elements the only possible problem is that some elements will keep empty. However, this is acceptable in the KDM philosophy, since it has an element called *ClassUnit* for representing classes, but we can also create KDM instances for procedural systems. Therefore, this type of element would not be created.

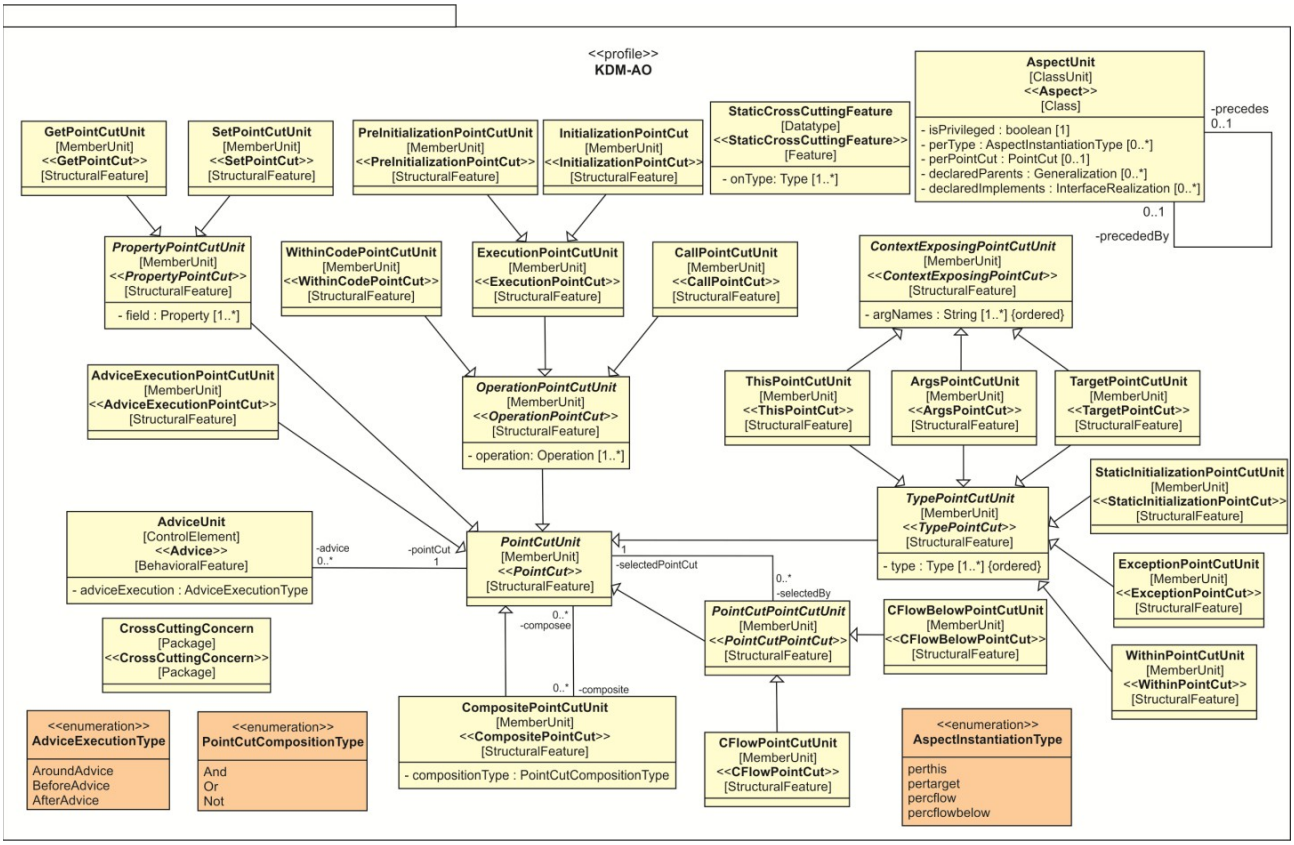


Figure 2. KDM-AO and Evermann's Profile (Adapted).

Fig. 2 shows both the Evermann's profile and the KDM-AO. Each class/element has four words in its first compartment. The first word (in bold) represents the name of the metaclass we have created in our extension, for example, *AspectUnit*. The second word inside the brackets is the KDM superclass we have chosen to make the actual element extends from. For example, we have decided to make our new metaclass *AspectUnit* extends the KDM metaclass *ClassUnit*. Below the mentioned elements, we also have two more words representing the Evermann's profile. For example, the stereotype *<<Aspect>>* created by Evermann extends the UML metaclass Class. These are the third and fourth words. So, in this figure, each element/class represents either a KDM-AO's metaclass, or an Evermann's stereotype, or an *enumeration* of values.

As previously mentioned, the profile proposed by Evermann uses specific AspectJ elements. However, the higher level elements are common to all AO languages, such as Aspect, Advice and Pointcut. Another point that has also guided our decision in choosing Evermann profile was that it had already been reviewed and modified by Gottardi [12], which allowed us to get a better view of its construction.

III. ASPECT-ORIENTED KDM

In this section we present some details of KDM-AO, which can be seen in Fig. 2. The first pair of brackets ([ ]) under the name of the each element exhibits the name of the KDM metaclass that was extended.

One of the biggest challenges when extending metamodels is to know if the metaclasses chosen as base for a new element is the most suitable ones. As Evermann´s profile elements had already previously been mapped to UML metaclasses (extending them through stereotypes), our main task was to identify KDM metaclasses that had similar characteristics to those ones used by Evermann. Due to that, it had been necessary to develop a mapping between both metamodels (UML and KDM), which can be seen in Table 1.

This mapping table identifies KDM metaclasses possessing similar characteristics to UML metaclasses. Some metaclasses can be direct mapped, such as Class from UML, which can be easily mapped to the *ClassUnit* metaclass from KDM. Both present the same goal and characteristics; representing classes in an object-oriented context. However, as KDM aims to represent lower-level details than UML, some UML metaclasses do not have just one candidate in the KDM side. This is the case of Property. This UML metaclass has three possibilities in KDM: *StorableUnit*, *ItemUnit* or *MemberUnit*. *StorableUnit* represents primitive type variables; *ItemUnit* represents registers and *MemberUnit* represents associations with other classes. This abstraction gap occurs because the Code package of KDM is in a lower abstraction level than UML.

TABLE I. KDM-UML MAPPING

UML Element	KDM Element	Differences
Class	*ClassUnit*	The metaclass Class (UML/ Basics package) has four properties: *isAbstract*, *ownedProperty*[*], *ownedOperation*[*] and superClass. The *ClassUnit* element, from *Code Package* encompasses all of these properties through the *AbstractCodeElement* class. A *ClassUnit* may have any attribute whose type is a concrete class of *AbstractCodeElement*, like *StorableUnit*, *MemberUnit*, *ItemUnit*, *MethodUnit*, *CommentUnit*, *KDMRelationships*, etc.
Operation	*MethodUnit*	*Operation* (UML/Basics package) is a behavioral element that has the following properties: class (specifies the owner class), *ownedParameter* (Operation's parameters) and *raisedException* (Operation's exceptions). The *MethodUnit* class is the ideal element to represent Operations because it is a behavioral KDM element capable to represent the most diverse programming languages operations. *MethodUnit* has attributes like *kind* (defines the kind of the operations, for example: abstract, constructor, destructor, virtual, etc.) and *export* (defines the access modifiers, for example: public, private and protected)
Property	*Storable, Member* or *ItemUnit*	*Property* (UML) represents variables in general (local variables, global variables, arrays, associations, etc.), while KDM has an element for each kind of *Property*: primitive type variable (*StorableUnit*), records and arrays (*ItemUnit*) class members (*MemberUnit*)
Package	*Package*	A *Package* on UML (Basics package) is very similarly to a KDM Package (*Code Package*). Both are containers for program elements, like classes, and others code elements. A *Package* could have one or more classes, and a class could have many others elements, like *methods*, *properties*, *comments*, etc.
Structural Feature	*DataElement*	*StructuralFeature* (UML/*Core*::*Abstractions* package) is an abstract metaclass that can be specialized to represent a structural member of a class, like a property. The KDM has the *DataElement* class (*Code package*), that can be specialized to *StorableUnit*, *MemberUnit* or *ItemUnit*.
Behavioral Feature	*Control Element*	*BehavioralFeature* (UML/ *Core::Abstractions* package) is an abstract metaclass that can be specialized to represent behavioral members of a class. The equivalent class on KDM is the *ControlElement*, an abstract class that can be specialized to represent callable elements, including behavioral elements like *MethodUnit*.
Parameter	*Parameter Unit*	*Parameter* (UML/ *Core:Abstractions*) is an abstract metaclass to represent the name and the type of the element that will be passed by parameter in a behavioral element. On the KDM we can use the *ParameterUnit* class. This metaclass can also represent the name, type, position of the parameter in the signature and the kind of paremeter (value or referece)
Relationship	*KDM Relationship*	Both *Relationship* and *KDMRelationship* metaclasses are abstract metaclasses that can be specialized to represent some kind of relationship between two elements, like *Aggregation*, *Generalization*, etc.
...	...	...

In Table 1 it is possible to see the existing relation between the metaclasses and also some comments about it. As KDM is a metamodel much broader than UML, most of the relations just make sense considering the Code Package of KDM, as this package is the one that aims to represent classes, attributes, methods, relationships and other static characteristics. The other KDM packages are more concentrated on details that are absent in UML, like Graphical

User Interface (GUI), architecture and conceptual elements. Because of space limitations, our mapping table shows just the main elements we have used in our KDM-AO extension. However, notice that we mapped all the classes from Evermann's profile.

Based on this mapping, we developed our KDM-AO by creating a new KDM metaclass for every stereotype presented in the Evermann´s profile but exchanging the metaclasses that was extended. For example, if a stereotype in the Evermann´s profile extends the Class metaclass, we then created a new corresponding metaclass in KDM (naming it in a similar way) and make it extends the *ClassUnit* metaclass from KDM.

As can be seen in Fig. 2, the main object-oriented elements (concepts) of Evermann´s profile are represented for higher level classes/stereotypes, which are: CrosscuttingConcern, Aspect, Advice, Pointcut and StaticCrossCuttingFeature. The remainders are subclasses of these higher level elements, representing subtypes. In this section we describe the corresponding elements we have created for each of the main elements. As it was presented earlier, in our KDM-AO, the name of most of our elements ends with the word Unit, for example, *AspectUnit*, *AdviceUnit* and *PointCutUnit*. That is the way we have used to differentiate between our elements from Evermann´s ones.

In Evermann´s profile, the CrosscuttingConcern element extends the Package UML metaclass and aims to represent the existence of a crosscutting concern like persistence, security and concurrency. In our KDM-AO this element extends the *Package* metaclass. This KDM metaclass represents a package in which is possible to encapsulate Aspects, Classes and others elements.

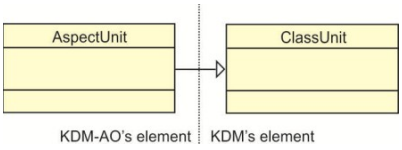


Figure 3. AspectUnit.

AspectUnit is our element for representing aspects, which extends the *ClassUnit* KDM metaclass (Fig. 3). We decided to extend this metaclass because aspects have all the characteristics classes have, besides pointcuts, advices and intertype declarations. From Fig. 3 to Fig. 6, we decided to omit the attributes/properties because all of them can be seen in the corresponding class in Fig. 2.

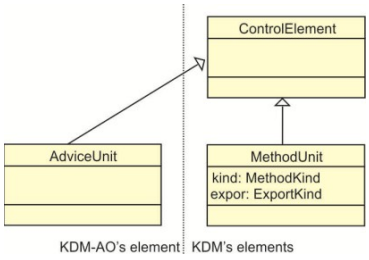


Figure 4. AdviceUnit.

Our element for representing advices is *AdviceUnit* (Fig. 4), which extends the *ControlElement* metaclass. Knowing that advice is an element that specifies behavior, we could consider it like a method. However, advices do not have neither access specifiers (public, private, protected) nor types (constructor, destructor, etc). Because of that we have decided do not make it extends MethodUnit.

PointCutUnit is our element for representing pointcuts. According to Evermann´s profile, pointcut is a structural element and extends the UML metaclass *StructuralFeature*. KDM has also a class for representing structural characteristics called *DataElement*, which is an abstract metaclass. Its descendents are *StorableUnit*, *MemberUnit* and *ItemUnit*. As *StorableUnit* and *ItemUnit* cannot be abstract, *MemberUnit* was chosen to be the superclass of *PointCutUnit*. Besides, another reason for extending *MemberUnit* was that pointcuts can crosscuts other classes and *MemberUnit* is the KDM metaclass used to denote references to other classes. The relations in which these classes are involved can be seen in Fig. 5.

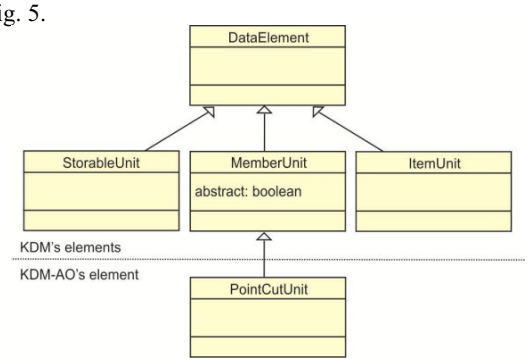


Figure 5. PointCutUnit.

StaticCrossCuttingFeature is our element for representing intertype declarations. In our KDM-AO we have decided to extend two KDM metaclasses: *StorableUnit* e *MethodUnit*. In this way, *StaticCrossCuttingFeature* is able to represent structural and behavioral characteristics. Therefore, an instance of *StaticCrossCuttingFeature* can be an attribute or a method (see Fig. 6).

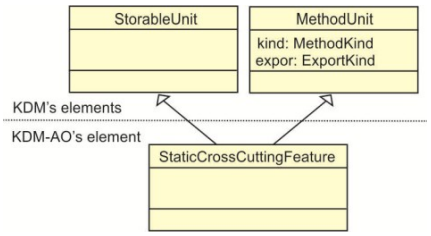


Figure 6. StaticCrossCuttingFeature.

Implementation Details. In order to create the KDM-AO, we have used the Eclipse IDE and Eclipse Modeling Framework (EMF) plug-in, which allows visualizing and editing the KDM metamodel in the Ecore format, available at the OMG website.

Each profile class is represented by means of EMF elements: *Eclass*, *EEnum*, *EPackage*, *EAtribute* and *EReference*. In Fig. 2, almost every class is represented inside

the metamodel for the *EClass* element. The elements denoted as <<*enumeration*>> are represented by the elements *EEnum*. The attributes inside the classes are recreated by the element *EAttribute* and the relationships between the profile classes are specified by the elements *EReference*. Fig. 7 shows one of our *AspectUnit* metaclass represented in the KDM metamodel. It is possible to see in part A the class attributes (*isPrivileged*, *perType*, *perPointCut*, *declaredParents* e *declaredImplements*) and relationships (*precedes* e *precededBy*). Part B shows that metaclass already introduced inside the KDM metamodel along with all of its attributes.

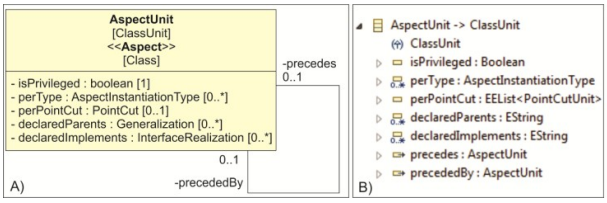


Figure 7. KDM-AO in EMF.

At every new added element there is a set of properties in which some have already default value and other do not, that is, it needs to be fulfilled. For instance, when adding a new *EClass* element, the main properties that must be informed are *Name* and *ESuperTypes* (super classes inherited by the new element). In Fig. 8 we show the properties belonging to *AspectUnit* metaclass. As long as all new metaclasses have been created in KDM, we generated a plug-in called KDM-AO plugin which allows the creation of KDM-AO instances.

Property	Value
Abstract	false
Default Value	
ESuper Types	ClassUnit -> Datatype
Instance Type Name	
Interface	false
Name	AspectUnit

Figure 8. AspectUnit properties.

IV. CASE STUDY

In this section we present a case study showing that the KDM-AO can be used to support a modernization process based on Crosscutting Frameworks (CFs) [3][22]. CFs are aspect-oriented frameworks that encapsulate in a generic way just one crosscutting concern, like persistence, security and cryptography [3][22]. CFs are composed of concrete and abstract aspects and also concrete and abstract classes. Most of them heavily rely on intertype declarations, dynamic crosscutting and well known aspect-oriented idioms like Container Introduction and Marker Interface [27].

The modernization scenario we regard here considers the existence of i) an instance of KDM representing a legacy system (here called “legacy KDM” or "base model") that needs to be modernized; ii) one or more instances of KDM representing CFs available in a repository; and iii) one or more KDM instances representing the elements of instantiation, i.e., concrete classes and aspects created by the application engineer to couple the CFs to a base code or base model (in this case, the legacy KDM).

In this case study, we have modernized a management system of a CD/DVD shop. The modernization goal was to modularize the persistence concern with aspects. As our group has some experience with Crosscutting Frameworks, the idea was to use a Persistence CF previously developed in this process. Doing that, we would be validating the KDM-AO in representing aspects and also in representing CFs.

Note that the focus of this paper is to show that it is possible to represent AO concepts with extended KDM. It is out of our scope mining the legacy KDM looking for crosscutting concerns, remove them or even provide a tool that facilitates the coupling of CFs. Therefore, the first step was to obtain a KDM instance representing the CD/DVD Shop system. This was done using MODISCO [25], which has a parser that automatically transforms Java source code into KDM XMI instances. The second step was concerned in obtaining an instance of our KDM-AO for our Persistence CF. This was done using a plug-in developed in this work, called KDM-AO plug-in, in which classes and aspects were converted into a KDM instance representing the CF.

Because of space limitations, in this section are shown only the main CF´s aspects with the wide variation in the use of elements that were inserted into the KDM metamodel related to the proposed in Evermann´s profile [8].

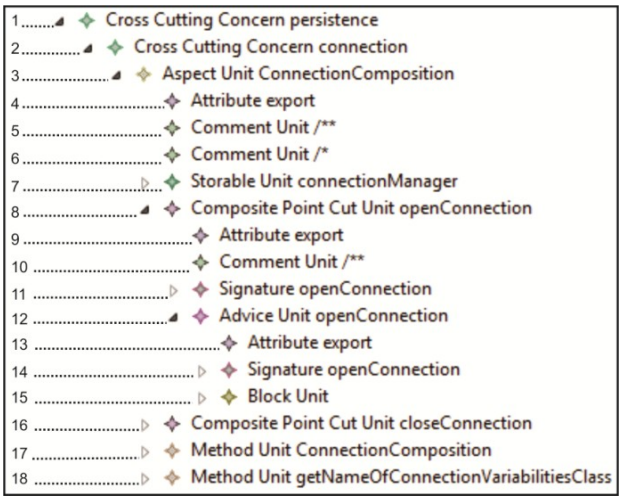


Figure 9. ConnectionComposition.aj in KDM-AO.

In Fig. 9 we can see an abstract aspect of the CF called *ConnectionComposition.aj* which is located inside the package *persistence.connection*. The purpose of this aspect is to provide a base behavior for opening and closing database connections. During instantiation, one needs to provide concrete implementations for the abstract pointcuts `openConnection()` and `closeConnection()`. This aspect has in your body an attribute, two abstract pointcuts, a concrete and one abstract operation and two advices.

The visualization shown in Fig. 9 is possible because of the use of KDM-SDK plug-in that allows one to edit XMI

models in accordance with the KDM metamodel [6]. However, you can also view the file generated by the plug-ins KDM-SDK and KDM-AO-plugin in XMI version.

Each line in Fig. 9 contains the element type and then its value. For example, in the first line we can visualize the existence of a *CrossCuttingConcern* element; whose value is persistence, i.e., this is an instance of the *CrossCuttingConcern* metaclass. Line 3 displays the name of the aspect that is being modeled here; initially the type (*AspectUnit*), then its value (*ConnectionComposition*).

The element *Attribute export* (line 4) is used to store the visibility (Public, Private and Protected), as well as indicate if the element is abstract or concrete. This element is used to represent classes, aspects, methods, pointcuts, advices among others elements that allow this type of statement. The element *StorableUnit* (line 7) is used to declare variables and *PointCutCompositeUnit* (lines 8 and 16) is used to represent concrete or abstract pointcuts of an aspect.

The element *Signature* (lines 11 and 14) receives the same name that the element does and has the function of storing the parameters that were passed in Pointcuts, Methods and Advices. *AdviceUnit* (line 12) represents an advice that was declared in the aspect. It is essential to fill the Advice Execution property because this property declares what kind of advice that element represents (After, Before or Around). In Fig. 9, the element *BlockUnit* (line 15) is the body of advice and you can represent snippets such as `try/catch`, among others. Comment Unit (lines 5, 6 and 10) stores comments that have been made in the source code and *MethodUnit* (lines 17 and 18) allows representing methods in the aspect.

It's Important to say that one source code in AspectJ consists of Java code and aspect code. Elements such as *MethodUnit, CommentUnit*, *Signature*, *BlockUnit* and *Attribute Export* already exist in the KDM metamodel and are being used to make the representation of the common elements of the Java language within the aspect.

```
1  <codeElement xsi:type="code:AspectUnit" name="OORelationalMapping">
2    <attribute tag="export" value="public abstract"/>
3    <attribute tag="export" value="private"/>
4    <ownedElement xsi:type="code:StaticCrossCuttingFeature" name="" ext="">
5      <ownedElement xsi:type="code:StorableUnit" name="tableName" type="/0/@model.0/@codeE
6        <attribute tag="export" value="public"/>
7        <ownedRelation xsi:type="code:HasValue" to="/0/@model.1/@codeElement.42"/>
8      </ownedElement>
9      <onType>PersistentRoot</onType>
10   </ownedElement>
11   <ownedElement xsi:type="code:StaticCrossCuttingFeature" name="">
12     <ownedElement xsi:type="code:MethodUnit" name="getID" type="/0/@model.0/@codeElement
13       <attribute tag="export" value="public abstract"/>
14       <codeElement xsi:type="code:Signature" name="getID">
15         <parameterUnit type="/0/@model.0/@codeElement.2/@codeElement.0" kind="return"/>
16       </codeElement>
17     </ownedElement>
18     <onType>PersistentRoot</onType>
19   </ownedElement>
```

Figure 10. A snippet of the aspect *OORelationalMapping.aj* in XMI format.

In Fig. 10 is shown the *OORelationalMapping* aspect as an instance of KDM-AO in XMI. This aspect aims to introduce (by intertype declaration) dozens of persistence methods in persistent classes of the application. In line 1, there is a declaration of the *OORelationaMapping,* which is an *AspectUnit*. Inside it, there are two Intertype Declarations through *StaticCrossCuttingFeature* element (lines 4 and 11). This kind of statement allows someone to insert properties and operations in other elements, such as interfaces, aspects and classes, just filling in the values in the *onType* attribute (lines 9 and 18). The first *StaticCrossCuttingFeature* (line 4) that appears is inserting a *StorableUnit* (line 5) named *tableName* in *PersistentRoot* interface. The second one (line 11) is inserting a *MethodUnit* element (line 12) named *getID* interface into the same Interface (*PersistentRoot*). In Fig. 11 is the equivalent AspectJ source code represented in the XMI in Fig. 10.

Figs. 9 and 10 showed that it is possible to represent and store KDM instances that represent aspects of CF's or conventional aspects. Another essential activity during the reuse of CFs is to perform the instantiation process and the coupling of the CF to a code base. This is done specializing concrete operations and pointcuts.

```
public abstract aspect OORelationalMapping {
   public String PersistentRoot.tableName = "";
   [...]
   public abstract int PersistentRoot.getID();
```

Figure 11. A snippet of the aspect *OORelationalMapping.aj* source code.

In our case study, it was necessary to create four concrete aspects and one class manually to perform the coupling of the CF to the *CDStore* application. The aspects created were *MyOORelationalMapping*, *MyConnectionCompositionRules*, *MyDirty* and *MyAspect* and the class was *MyConnectionVariabilities*. The *MyConnectionVariabilities* class stores information about the database; the aspect *MyOORelationalMapping* declares classes of the base application that should receive persistence methods; the aspect

MyConnectionCompositionRules specifies the points at the connection to the database will be opened and closed. Finally the aspect *myDirty* and *myAspect* that are abstracts and extend aspects of the CF.

Fig. 12 shows the *MyOORelationalMapping* aspect (lines 1, 2 and 3), whose name can be seen on line 2. Inside this aspect created by the application engineer there are declare parents statements informing application classes that they must extend a CF interface called *PersistentRoot*. This is done so that all classes receive application persistence operations defined in this interface.

```
1  <codeElement xsi:type="code:AspectUnit"
2                  name="MyOORelationalMapping"
3                  isAbstract="false">
4    <attribute tag="export" value="public"/>
5    <ownedRelation xsi:type="code:Imports"
6                     to="CrossCuttingConcern persistence"
7                     from="AspectUnit MyOORelationalMapping"/>
8    <ownedRelation xsi:type="code:Imports"
9                     to="CrossCuttingConcern application"
10                    from="AspectUnit MyOORelationalMapping"/>
11   <ownedRelation xsi:type="code:Extends"
12                    to="AspectUnit OORelationalMapping"
13                    from="AspectUnit MyOORelationalMapping"/>
14   <ownedRelation xsi:type="code:InterfaceRealization"
15                    to="InterfaceUnit PersistentRoot"
16                    from="ClassUnit Music"
17                    Name="Music-PersistentRoot"/>
18   <ownedRelation xsi:type="code:InterfaceRealization"
19                    to="InterfaceUnit PersistentRoot"
20                    from="ClassUnit Purchase"
21                    Name="Purchase-PersistentRoot"/>
22   <declaredParents>Music-PersistentRoot</declaredParents>
23   <declaredParents>Purchase-PersistentRoot</declaredParents>
24 </codeElement>
```

Figure 12. A snippet of the aspect *MyOORelationalMapping*.aj in XMI format.

In lines 5 to 7 and 8 to 10 are shown two *Imports*, the first package is the *persistence* (CF package) and the second is the *application* package (base application package). The lines 11 to 13 represent the *Extends* element that stores the information that the aspect *MyOORelationalMapping* extends the behavior of the *OORelationalMapping* aspect, present in the CF.

In lines 22 and 23 of this aspect can be visualized the use of the *declareParents* element, this element stores the name of a relationship between a base application class and a CF's aspect or interface.

To specify this relationship in a model, it's necessary to use an element that can store the name of the base application class, the name of the CFs aspect or interface and the name of the relationship between them. In KDM, the element capable to do this representation is the *Implements*, however, he does not have a name property, thus it was necessary to extend this element and add the "name" property. This new element created from Implements was called *InterfaceRealization*.

Lines 14 to 17 represent an instance of the element *InterfaceRealization* where line 15 shows the CF *PersistentRoot* interface, line 16 shows the Music basic application class and the name of the relationship between them can be seen in line 17. Another *InterfaceRealization* instance can be seen in lines 18 to 21.

The second snippet in the source code instantiation to be shown are the pointcuts that open and close the connection to the database, present on the aspect *myConnectionCompositionRules*. In Fig. 13 is shown a snippet of a XMI file that contains the element pointcut *openConnection* where is possible to see its main elements, like *CompositePointCutUnit*, *ExecutionPointCutUnit* and *ParameterUnit*. The *CompostitePointCutUnit* element (line 5) is the encapsulation of all pointcuts that represent the *openConnection* (line 6). The *ExecutionPointCut* (Line 9) is the pointcut that crosscutting the *main* method (line 13) from *FindSomeCDs* class (Line 11). Finally, the *ParameterUnit* (lines 16 to 19) store the pointcut parameters.

```
1  <ownedElement xsi:type="code:AspectUnit"
2                  name="myConnectionCompositionRules"
3                  isAbstract="false">
4    <attribute tag="export" value="public"/>
5    <ownedElement xsi:type="code:CompositePointCutUnit"
6                    name="openConnection"
7                    compositeType="OR">
8      <attribute tag="export" value="public"/>
9      <ownedElement xsi:type="code:ExecutionPointCutUnit"
10                     type="//@model.0/@codeElement.1/@codeEleme
11       <codeElement xsi:type="code:ClassUnit" name="FindSomeCDs">
12         <attribute tag="export" value="public"/>
13         <codeElement xsi:type="code:MethodUnit" name="main">
14           <attribute     tag="export" value="public static"/>
15           <codeElement xsi:type="code:Signature" name="main">
16             <ownedElement xsi:type="code:ParameterUnit"
17                           type="//@model.0/@codeElement.1/@c
18                           kind="return"/>
19             <parameterUnit name=".." ext="" kind="unknown"/>
```

Figure 13. A snippet of the aspect *MyConnectionCompositionRules*.aj in XMI format.

With the realization of this case study was possible to ascertain the suitability of the extension developed to represent the most important characteristics and specificities of a CF implemented in AspectJ.

Lower level speficifications. To represent a generic *pointcut* with our extension, it is only needed create an instance of *OperationPointCutUnit* (Fig. 1) and inform the parameters that crosscut the base system. But if a more specific *pointcut* has to be represented, it's possible to create an instance of a more specific *pointcut*. For example, *GetPointCutUnit* and *SetPointCutUnit* are specials kinds of *PointCutUnit* that represent field accesses.

Another example is the control flow of a join point. A control-flow pointcut always specifies another pointcut as its argument. There are two control-flow pointcuts, and in our extension they are represented by *CFlowPointCutUnit* and *CFlowBelowPointCut*. The first pointcut captures all the *OperationPointCutUnit* in the control flow of the specified *PointCutUnit*, including the *OperationPointCutUnit* matching the *PointCutUnit* itself. The second *PointCutUnit* excludes the *OperationPointCutUnit* in the specified *PointCutUnit* [30].

Fig. 14 shows how the mentioned *PointCutUnits* can be represented in the KDM-AO plug-in. Lines 2 and 3 represents the *GetPointCutUnit* and *SetPointCutUnit*, representing that the *accountBalance* field will be crosscut when it is read or write. The *CompositePointCutUnit* (lines 4 and 7)

encapsulates the *PointCutUnits,* allowing the application engineer specify the points of the base system that will be affected by *PointCutUnits*. Line 6 shows a *CallPointCutUnit* that is modified by a *CFlowPointCutUnit* (line 5) and Line 9 shows a *ExecutionPointCutUnit* that is modified by a *CFlowBelowPointCutUnit* (line 8).

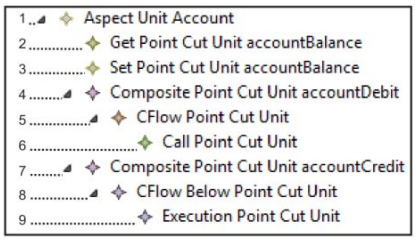


Figure 14. Lower level specifications example in KDM-AO plug-in.

There are others possible representations of pointcuts supported by our extension, and the level of details of a KDM-AO instance will depend mainly on the application engineer and the parser that creates the instance.

V. RELATED WORKS

The research work most related to ours is the KDM AO extension presented by Mirshams [9]. As we have done here, this author also created a heavyweight KDM extension for aspect-oriented programming. There are three main differences between our works. Firstly, while Mirshams has based her extension in an aspect model created by herself, we have created our extension based on a very well known profile for aspect-oriented programming. Evermann´s profile encompasses all the AO concepts presented in AspectJ and in other less known aspect-oriented languages, like Aspect C++ and AspectS [8].

The second difference is the level of abstraction of our extensions. The aspect model used by Mirshams contains much less elements than Evermann´s profile. That means our extension is able to represent both a high level (using the most generic metaclasses) and a low level (using most specific metaclasses) view of the system. In her case, just a higher level view is possible. The third difference is that her work is limited to dynamic crosscutting as there are no elements for representing intertype declarations. However, despite all of these differences, the main similarity is that we have used the same KDM metaclasses she has used too.

Another KDM extension is presented by Baresi and Miraz [28]. They proposed a heavyweight KDM extension to support Component-Oriented MOdernization (COMO). COMO is a metamodel that supports traditional concepts of software architecture, allowing to attach software components in KDM. Using their extension it is possible to replace or add parts of a system. Unlike we have done here, in their paper they had not used an existing profile as the starting point for creating their extension - they combined another metamodel to the KDM.

COMO extends some high level metaclasses of KDM, such as *KDMModel*, *KDMEntity* and *KDMRelationship*. These classes are the base of their extension and provide the link between KDM and COMO metamodels.

The main similarity with our work is that they have also performed a heavyweight extension in KDM. As a main difference, the extension presented by them only extended high level elements of KDM, while in our solution we have use more specific elements such as *ClassUnit* and *MemberUnit*.

VI. DISCUSSIONS AND CONCLUSIONS

As we have commented in Section II, a heavyweight extension can change the original metamodel or simply add new metaclasses. We have opted for this second choice because it facilitates the reuse of our extension in other contexts. Besides, only by using heavyweight extensions is possible to guarantee some level of correctness in model-level. Otherwise this responsibility must be transferred to tools.

By means of our case study, it is fairly evident that our extension can represent all AOP elements. However, as we have not carried out a complete case study to gauge how reliable our extension is to represent aspects concepts in other programming languages, such as AspectC++, we consider this is a limitation of our extension. Nevertheless, to mitigate this limitation, the elements of AspectC++ and AspectS were analyzed. Consequently, we conclude that there are enough elements in our extension that can be used to represent source code in both AspectC++ and AspectS.

Apart from Mirsham´s work [9] we did not find another work that has extended KDM for aspect-oriented programming. Her extension is also a heavyweight solution and does not include inter-type declarations. Besides, her solution does not allow representing lower level concepts.

Another contribution here is to show a preliminary mapping between UML and KDM which can be used to turn UML profiles into to KDM extensions. In our case, we turned an AO UML profile into a KDM heavyweight AO extension. However, considering the mapping shown in Table 1, any UML profile could be transformed. This is quite useful because in Model-Driven Environments, systems that are represented as KDM instances will need to be visualized as class diagrams.

Although our case study has shown just the ability of KDM-AO to represent existing code (reverse engineering), it can also be used in forward engineering for code generation. In this case, it seems to be more appropriate than the Mirsham's profile, since ours includes lower level details.

When conducting our case study using CFs, we have noticed that our extension would be more useful and more expressive if it had also metaelements for representing CF characteristics, like hot spots, frozen spots and other framework characteristics. We intend to perform these modifications in a future work [3].

Another interesting work we intend to conduct in the future is to compare our heavyweight extension with a lightweight one. Currently, we are already developing a lightweight version of our extension. However, we can already anticipate that one of the biggest drawbacks of the lightweight version is the possibility of including erroneous relationships in the model. We are also planning to carry out an experiment to list the vantages and disadvantages of each extension.

As other future works, we aim to conduct others case studies using AspectC++ and AspectS in order to test the

KDM-AO extension, with the objective of evaluating the issue of platform independence. Another future work that can be done is to check if there is some other element to be added to the profile, taking into account new additions to the aspect-oriented programming from 2007 to the current year.

By conducting this research we have noticed that the power of model-driven modernization is greatly influenced by the capacity of representing specific concepts in a proper and suitable way.

As we have shown, the absence of aspectual concepts in the original KDM prevent the realization of aspect-oriented modernizations, or at least, makes it very hard. The same occurs when we consider other fields/domains, such as: web services, embedded systems, business processes, fault tolerance, testing, etc. All of these subareas has already UML profiles, available at OMG [26] [27], aiming to represent specific details/concepts/abstractions in a more precise way. Therefore, our mapping table can be easily employed to create KDM extensions from all of these UML profiles, as we have exemplified here.

ACKNOWLEDGEMENTS

Bruno M. Santos and Raphael R. Honda would like to thank CNPq for sponsoring our research. Rafael S. Durelli and Valter V. de Camargo would like to thank the financial support provided by FAPESP, processes numbers 2012/05168-4 and 2014/14080-9, respectively.

REFERENCES

[1] G. Visaggio, "Ageing of a data-intensive legacy system: symptoms and remedies," Journal of Software Maintenance 13. 2001, pp. 281–308, doi: 10.1002/smr.234.

[2] Architecture-Driven Modernization, 2014. Document omg/ http://adm.omg.org/.

[3] V. V. Camargo and P. C. Masiero, "Frameworks Orientados a Aspectos," XIX Simpósio Brasileiro de Engenharia de Software, Uberlândia. 2005, pp. 200-216.

[4] D. L. Parnas, "Software aging," ICSE '94 Proc. of the 16th international conference on Software engineering, Los Alamitos, CA, USA, 1994, pp. 279-287.

[5] K. Normantas, S. Sosunovas and O. Vasilecas, "An Overview of the Knowledge Discovery Meta-Model," Proc. of the 13th International Conference on Computer Systems and Technologies - CompSysTech'12, 2012, pp. 52-57.

[6] Knowledge Discovery Meta-Model. KDM Guide, August 2011. Document omg/formal/2011-08-04.

[7] V. V. Camargo, P. C. Masiero, "An Approach to Design Crosscutting Framework Families," ACP4IS 08, Brussels, Belgium, 2008.

[8] J. Evermann, "An overview and an empirical evaluation of UML: an UML profile for aspect-oriented frameworks," Workshop AOM '07, Vancouver, British Columbia, Canada, 2007.

[9] P. S. Mirshams, "Extending the Knowledge Discovery Metamodel to Support Aspect-Oriented Programming," 79 p. Dissertation (Master in Applied Science in Software Engineering) – Computer Science Department and Software Engineering, University of Montreal, Quebec, Canada, 2011, unpublished.

[10] M. Kande, J. Kienzle, and A. Strohmeier, "From AOP to UML - a bottom-up approach," Proc. of the AOM with UML workshop at AOSD, 2002, 2002.

[11] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli, "A UML notation for aspect-oriented software design," Proc. of the AOM with UML workshop at AOSD, 2002, 2002.

[12] T. Gottardi, R. A. D. Penteado and V. V. de Camargo, "A Process for Aspect-Oriented Platform-Specific Profile Checking," Proc. of the 2011 International Workshop on Early Aspects. New York, NY , USA. 2011.

[13] D. Stein, S. Hanenberg, and R. Unland, "Designing aspect-oriented crosscutting in UML," Proc. of the AOM with UML workshop at AOSD, 2002.

[14] M. Basch and A. Sanchez, "Incorporating aspects into the UML," Proc. of the AOM workshop at AOSD, 2003.

[15] L. Fuentes and P. Sanchez, "Elaborating UML 2.0 profiles for AO design," Proc. of the AOM workshop at AOSD, 2006.

[16] E. Barra, G. Genova, and J. Llorens, "An approach to aspect modelling with UML 2.0," Proc. of the AOM workshop at AOSD, 2004.

[17] J. Grundy and R. Patel, "Developing software components with the UML, Enterprise Java Beans and aspects," Proc. of ASWEC 2001, Canberra, Australia, 2001.

[18] C. Chavez and C. Lucena, "A metamodel for aspect-oriented modeling," Proc. of the AOM with UML workshop at AOSD, 2002.

[19] H. Yan, G. Kniesel, and A. Cremers, "A meta model and modeling notation for AspectJ," Proc. of the AOM workshop at AOSD, 2004.

[20] A. Rashid and R. Chitchyan, "Persistence as an Aspect," 2nd International Conference on Aspect Oriented Software Development (AOSD), Boston–USA, March, 2003.

[21] C. F. M. Couto, M. T. O. Valente and R. da S. Bigonha, "Um Arcabouço Orientado por Aspectos para Implementação Automatizada de Persistência," 2º. Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP'05), evento satélite do XIX SBES, Uberlândia, MG, Brasil, outubro, 2005.

[22] T. Gottardi, R. S. Durelli, O. P. López and V. V. Camargo, "Model-based reuse for crosscutting frameworks: assessing reuse and maintenance effort," Journal of Software Engineering Research and Development, 2013, pp. 1-34, doi:10.1186/2195-1721-1-4.

[23] S. Soares, E. Laureano and P. Borba, "Implementing Distribution and Persistence Aspects with AspectJ," 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), November, 2002, pp 174-190.

[24] A. Rausch, B. Rumpe and L. Hoogendoorn, "Aspect-Oriented Framework Modeling," 4th AOSD Modeling with UML Workshop (UML Conference 2003) October, 2003.

[25] H. Bruneliere, J. Cabot, F. Jouault and F. Madiot, "MoDisco: A generic and extensible framework for model driven reverse engineering," IEEE/ACM international conference on Automated software engineering, ACM New York, NY, USA, 2010, pp. 173-174.

[26] Object Management Group. OMG Specifications, April 2014. Documents omg/ http://www.omg.org/spec/.

[27] S. Hanenberg, "Multi-Design Application Frameworks," Generative and Component-Based Software Engeneering Young Reaearchers Workshop, Erfurt, October 10, 2000.

[28] L. Baresi and M. Miraz, "A Component-oriented Metamodel for the Modernization of Software Applications," 16th IEEE International Conference on Engineering of Complex Computer Systems. 2011.

[29] G. Kiczales *et al.*, "Aspect Oriented Programming," Proc. of 11 ECOOP. pp. 220-242, 1997.

[30] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications, Greenwich (74° w. long.), 2003, pp.75-77.

[31] J. U. Júnior, R. D. Penteado and V. V. Camargo, "An overview and an empirical evaluation of UML-AOF: an UML profile for aspect-oriented frameworks," Proc. of the 2010 ACM Symposium on Applied Computing. 2010, pp 2289-2296.

[32] R. S. Durelli *et al.*, "A Mapping Study on Architechture-Driven Modernization," 15th IEEE International Conference on Information Reuse and Integration, 2014, pp 1-8.

Anexo 9: Comprovante da publicação no *II Workshop on Software Visualization, Evolution and Maintenance – Brazilian Conference on Software: theory and practice, 2014* (VEM 2014)

- Uma cópia do artigo é apresentado a seguir (a partir da próxima página).

KDM-RE: A Model-Driven Refactoring Tool for KDM

Rafael S. Durelli[1], Bruno M. Santos[2], Raphael R. Honda[2], Márcio E. Delamaro[1] and Valter V. de Camargo[2]

[1]Computer Systems Department University of São Paulo - ICMC
São Carlos, SP, Brazil.

[2]Computing Departament
Federal University of São Carlos - UFSCAR
São Carlos, SP, Brazil.

{rdurelli, delamaro}@icmc.usp.br[1],

{valter, bruno.santos, raphael.honda}@dc.ufscar.br[2]

Abstract. *Architecture-Driven Modernization (ADM) advocates the use of models as the main artifacts during modernization of legacy systems. Knowledge Discovery Metamodel (KDM) is the main ADM metamodel and its two most outstanding characteristics are the capacity of representing both i) all system details, ranging from lower level to higher level elements, and ii) the dependencies along this spectrum. Although there exist tools, which allow the application of refactorings in class diagrams, none of them uses KDM as their underlying metamodel. As UML is not so complete as KDM in terms of abstraction levels and its main focus is on representing diagrams, it is not the best metamodel for modernizations, since modifications in lower levels cannot be propagated to higher levels. To fulfill this lack, in this paper we present a tool that allows the application of seventeen fine-grained refactorings in class diagrams. The main difference from other tools is that the class diagrams uses KDM as their underlying metamodel and all refactorings are applied on this metamodel. Therefore, the modernizer engineer can detect "model smells" in these diagrams and apply the refactorings.*

1. Introduction

Architecture-Driven Modernization (ADM) is an initiative which advocates for the application of Model Driven Architecture (MDA) principles to formalize the software reengineering process. According to the OMG the most important artifact provided by ADM is the Knowledge Discovery Metamodel (KDM). KDM is an OMG specification adopted as ISO/IEC 19506 by the International Standards Organization for representing information related to existing software systems. KDM is structured in a hierarchy of four layers; *Infrastructure Layer*, *Program Elements Layer*, *Runtime Resource Layer*, and *Abstractions Layer*. We are specially interested in the *Program Elements Layer* because it defines the Code and Action packages which are widely used by our tool. The Code package defines a set of meta-classes that represents the common elements in the source-code supported by different programming languages such as: (*i*) `ClassUnit` and `InterfaceUnit` which represent classes and interface, respectively, (*ii*) `StorableUnit` which illustrates attributes and (*iii*) `MethodUnit` to represent methods, etc. The Action package represents behavior descriptions and control-and-data-flow relationships between code

elements. Refactoring has been known and highly used both industrially and academically. It is a form of transformation that was initially defined by Opdyke [Opdyke 1992] as "a change made to the internal structure of the software while preserving its external behavior at the same level of abstraction". In the area of object-oriented programming, refactorings are the technique of choice for improving the structure of existing code without changing its external behavior [Fowler et al. 2000]. Refactorings have been proved to be useful to improve the quality attributes of source code, and thus, to increase its maintainability. It is possible to identify several catalogs of refactoring for different languages and the most complete and influential was published by Fowler in [Fowler et al. 2000]. Nowadays, there are researches been carried out about apply refactoring in model instead of source code[Ulrich and Newcomb 2010]. Nevertheless, although ADM provides the process for refactoring legacy systems by means of KDM, there is a lack of an Integrated Development Environment (IDE) to lead engineers to apply refactorings as such exist in others object-oriented languages. In the same direction, Model-Driven Modernization (MDM) is a special kind of model transformation that allows us to improve the structure of the model while preserving its internal quality characteristics. MDM is a considerably new area of research which still needs to reach the level of maturity attained by source code refactoring [Misbhauddin and Alshayeb 2012].

In order to enable MDM in the context of ADM, refactorings for the KDM metamodel are required. In this context, in a parallel research line of the same group, we developed a catalogue of refactorings for the KDM [Durelli et al. 2014]. We argue that devising a refactoring catalogue for KDM makes this catalogue language-independent and standardized. However, the KDM metamodel was not created with the goal of being the basis for diagrams, as is the case of UML metamodel. Thereby, in order to make possible to apply fine-grained refactoring in the KDM metamodel, it is necessary to devise a way to view the KDM instance graphically. Furthermore, although there exist tools, which allow the application of refactorings in class diagrams, none of them uses KDM as their underlying metamodel. As UML is not so complete as KDM in terms of abstraction levels and its main focus is on representing diagrams, it is not the best metamodel for modernizations, since modifications in lower levels cannot be propagated to higher levels

Hence, the main contribution of this paper is the provision of a plug-in on the top of the Eclipse Platform named **K**nowledge **D**iscovery **M**odel-**R**efactoring **E**nvironment (KDM-RE). This plug-in can be used to lead engineers to apply refactorings in KDM, which are based on seventeen well known refactorings[Fowler et al. 2000]. The IDE as well as the adapted catalogue are based on our experience as model-driven engineering. Also, by using this plug-in the modernizer engineer can visualize the Code package as an UML class diagram, allowing engineers to detect model smells in that diagram. One hypothetical case study was developed in order to exemplify the use of the plug-in. This paper is organized as followed: Section 2 provides the background to fully understand our plug-in - Section 3 depicts information upon the plug-in KDM-RE and an case study - in Section 4 there are related works and in Section 5 we conclude the paper with some remarks and future directions.

2. ADM and KDM

OMG defined ADM initiative [Perez-Castillo et al. 2009] which advocates carrying out the reengineering process considering MDA principles. ADM is the concept of modern-

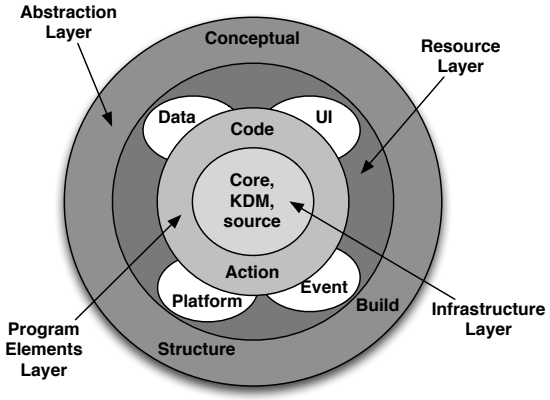


Figure 1. Layers, packages, and separation of concerns in KDM (Adapted from [OMG 2012])

izing existing systems with a focus on all aspects of the current systems architecture. It also provides the ability to transform current architectures to target architectures by using all principles of MDA [Ulrich and Newcomb 2010].

To perform a system modernization, ADM introduces Knowledge Discovery meta-model (KDM). KDM is an OMG specification adopted as ISO/IEC 19506 by the International Standards Organization for representing information related to existing software systems. According to [Perez-Castillo et al. 2009] the goal of the KDM is to define a meta-model to represent all the different legacy software artifacts involved in a legacy information system (e.g. source code, user interfaces, databases, etc.). The KDM provides a comprehensive high-level view of the behavior, structure and data of legacy information systems by means of a set of meta-models. The main purpose of the KDM specification is not the representation of models related strictly to the source code nature such as Unified Modeling Language (UML). While UML can be used to mainly to visualize the system "as-is", an ADM-based process using KDM starts from the different legacy software artifacts and builds higher-abstraction level models in a bottom-up manner through reverse engineering techniques. As outlined before, the KDM consists of four abstraction layers: (*i*) *Infrastructure Layer*, (*ii*) *Program Elements Layer*, (*iii*) *Runtime Resource Layer*, and (*iv*) *Abstractions Layer*. Each layer is further organized into packages, as can be seen in Figure 1. Each package defines a set of meta-model elements whose purpose is to represent a certain independent facet of knowledge related to existing software systems. We are specially interested in the *Program Elements Layer* because it defines the Code and Action packages which are widely used by our catalogue. The Code package defines a set of meta-classes that represents the common elements in the source code supported by different programming languages. In Table 1 is depicted some of them. This table identifies KDM meta-classes possessing similar characteristics to the static structure of the source code. Some meta-classes can be direct mapped, such as Class from object-oriented language, which can be easily mapped to the `ClassUnit` meta-class from KDM.

3. Refactoring for KDM by means of KDM-RE

This sections describes KDM-RE. In Figure 2 we depicted the main window of our plug-in. For explanation purpose, we highlight two main regions, i.e., ⓐ, and ⓑ. It supports 17

Table 1. Meta-classes for Modeling the Static Structure of the Source-code

Source-Code Element	KDM Meta-Classes
Class	ClassUnit
Interface	InterfaceUnit
Method	MethodUnit
Field	StorableUnit
Local Variable	Member
Parameter	ParameterUnit
Association	KdmRelationShip

refactorings adapted to KDM. These refactorings are based on some fine-grained refactorings proposed by Fowler [Fowler et al. 2000]. All the refactorings are shown in Table 2. We chose the Fowler's refactorings because they are well known, basic and fine-grained refactorings. Please, not that KDM-RE uses MoDisco[1] once it provides an extensible framework to transform an specific source-code to KDM models. In Figure 2 is presented

Table 2. Refactorings Adapted to KDM

Rename Feature	Moving Features Between Objects	Organing Data	Dealing with Generalization
Rename ClassUnit	Move MethodUnit	Replace data value with Object	Push Down MethodUnit
Rename StorableUnit	Move StorableUnit	Encapsulate StorableUnit	Push Down StorableUnit
Rename MethodUnit	Extract ClassUnit	Replace Type Code with ClassUnit	Pull Up StorableUnit
	Inline ClassUnit	Replace Type Code with SubClass	Pull Up MethodUnit
		Replace Type Code with State/Strategy	Extract SubClass
			Extract SuperClass
			Collapse Hierarchy

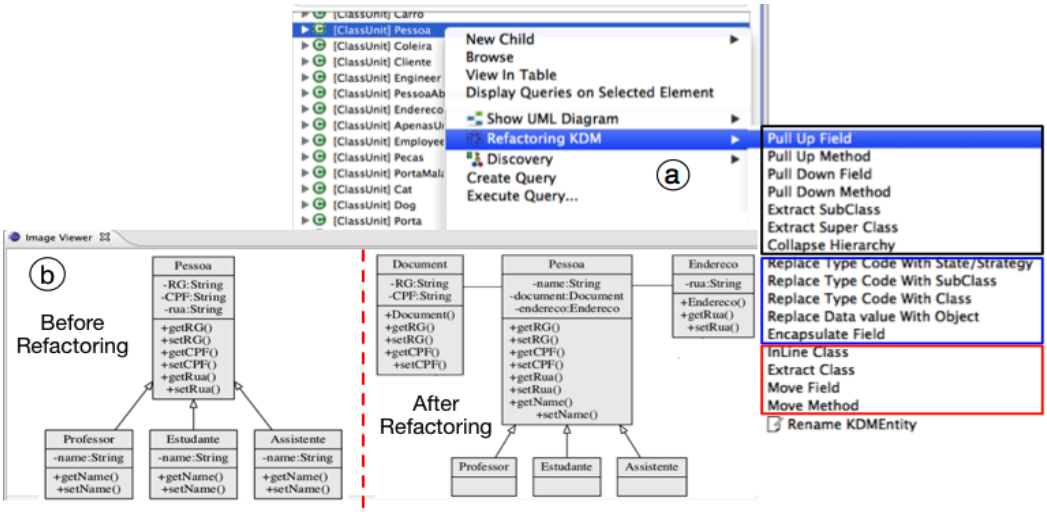


Figure 2. Snippets KDM-RE's Interface

just a snippet of KDM-RE. Starting from the popup menu named "Refactoring KDM", in this model browser, see Figure 2ⓐ, either the software developer or software modernizer can interact with the KDM model and choose which refactoring must be carried out in the KDM. In the region ⓐ can be seen all 17 refactorings that have been implemented in KDM-RE. For illustration purposes only we drew rectangles to separate the refactorings

[1] http://www.eclipse.org/MoDisco/

into three groups. The black rectangle represents refactorings that deal with generalization, the blue rectangle stand for refactorings to organize data and the red one symbolize refactoring to assist the moving features between objects.

The region ⓑ on Figure 2 shows an UML class diagram. This diagram can be used before to apply some refactorings to assist the modernizer to decide where/when to apply the refactorings. This UML class diagram also can be useful as the modernizer performs the refactorings in KDM model. For instance, changes are reproduced on the fly in a class diagram. We claim that the latter use of this diagram is important once it provides an abstract view of the system, hence, the modernizer can visually check the system's changes after applying a set of refactorings. Furthermore, in the context of modernization usually the source-code is the only available artifact of a legacy system. Therefore, creating an UML class diagram makes, both the legacy system and the generated software to have a new type of artifact (i.e., UML class models), improving their documentation.

3.1. Case Study

In this section, we motivate KDM-RE by analyzing one hypothetical case study. This case study is a small part of the university domain. Figure 2 ⓑ (left side) shows a class diagram used for modeling a small part of the university domain. In an university there are several Persons, more specifically Professors, their Assistants, and Students. Each Person has RG, CPF, and address (of type String). Moreover, classes Professor, Assistant, and Student have an attribute name of type String each. The software modernizer or the software developer found out by looking at the UML class diagram (see Figure 2ⓑ left side) this redundantly, i.e., equal attributes in sibling classes. Therefore, he/she must apply the refactoring "Pull Up Field'. Similarly, he/she also found out by looking at the UML class diagram that one class is doing work that should be done by two or more. For example, he/she found that the attributes RG and CPF should be modularized to a class. Similarly, it is necessary to provide more information about they address, such as number, city, country, etc. Therefore, he/she must apply the refactoring "Extract Class" to the attributes "RG", "CPF" and "rua". Due space limitation it is depicted just the extraction of the attributes "RG" and "CPF". The first step is to select the meta-class that he/she identified as a bad smell, i.e., the meta-class to be extracted into a separate one. This step is illustrated in Figure 3(a).

After selecting the meta-class, a right-click opens the context menu where the refactoring is accessible. After the click, the system displays the "RefactoringWizard" to the engineer, Figure 3(b) depicts the Extract Class Wizard. In this wizard, the name of the new meta-class can be set. Also a preview of all detected `StorableUnits` and `MethodUnits` that can be chosen to put into the new meta-class. Further, the engineer can select if either the new meta-class will be a top level meta-class or a nested meta-class. The engineer also can select if the KDM-RE must create instances of `MethodUnits` to represent accessors methods (gets and sets). Finally, the engineer can set the name of the `StorableUnit` that represent the link between the two meta-classes (the old meta-class and the new one). After all of the required inputs have been made, the engineer can click on the button "Finish" and the refactoring "Extract Class" is performed by KDM-RE.

As can be seen in Figure 3(c) a new instance of `ClassUnit` named "Document" was created - two `StorableUnit` from "Pessoa", i.e., "rg" and "CPF" were moved

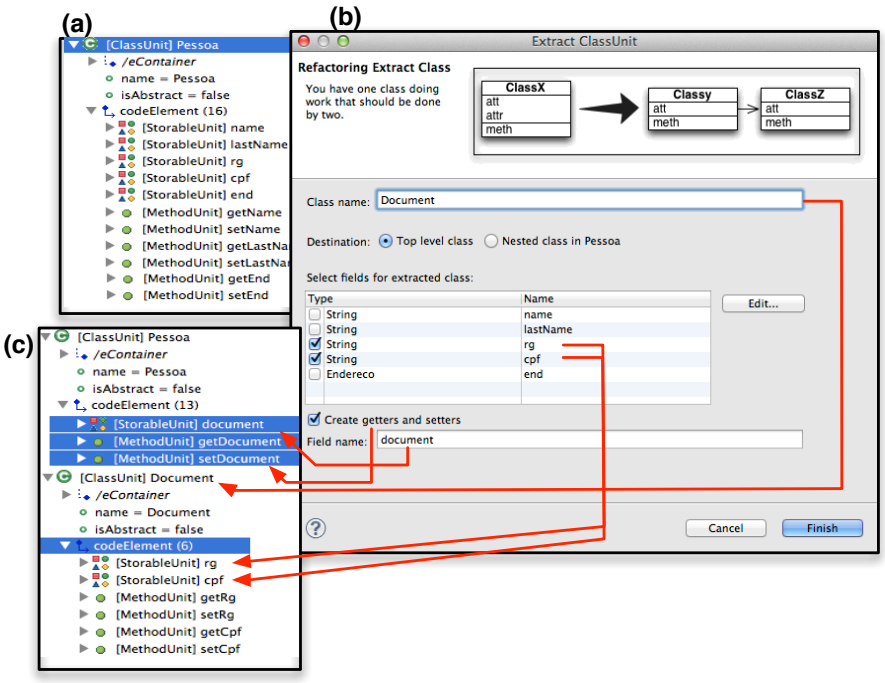


Figure 3. Extract Class Wizard

to the new `ClassUnit` - instances of `MethodUnits` were also created to represent the gets and sets. In addition, the instance of `ClassUnit` named "Pessoa" owns a new instance of `StorableUnit` that represent the link between both `ClassUnits`. Due space limitation the other `StorableUnits` of `ClassUnit` named "Pessoa" are not shown in Figure 3(c). After the engineer realize the refactorings, an UML class diagram is created on the fly to mirror graphically all changes performed in the KDM model, see Figure 2ⓑ right side.

4. Related Work

Van Gorp et al. [Gorp et al. 2003] proposed a UML profile to express pre and post conditions of source code refactorings using Object Constraint Language (OCL) constraints. The proposed profile allows that a CASE tool: (*i*) verify pre and post conditions for the composition of sequences of refactorings; and (*ii*) use the OCL consulting mechanism to detect bad smells such as crosscutting concerns. Reimann et al. [Reimann et al. 2010] present an approach for EMF model refactoring. They propose the definition of EMF-based refactoring in a generic way. Another approach for EMF model refactoring is presented in [Thorsten Arendt 2013]. They propose EMF Refactor [2], which is a new Eclipse incubation project in the Eclipse Modeling Project consisting of three main components. Besides a code generation module and a refactoring application module, it comes along with a suite of predefined EMF model refactorings for UML and Ecore models.

[2]http://www.eclipse.org/emf-refactor/

5. Concluding Remarks

In this paper is presented the KDM-RE which is a plug-in on the top of the Eclipse Platform to provide support to model-driven refactoring based on ADM and uses the KDM standard. More specifically, this plug-in supports 17 refactorings adapted to KDM. These refactorings are based on some fine-grained refactorings proposed by Fowler [Fowler et al. 2000]. As stated in the case study the engineer/modernizer by using KDM-RE can apply a set refactorings in a KDM. Also, on the fly the engineer can check all changes realized in this KDM replicated into a class diagram - the engineer can visually verify the system's changes after applying a set of refactorings. In addition, usually the source code is the only available artifact of the legacy software. Therefore, creating an UML class diagram makes, both the legacy software and the generated software to have a new type of artifact (i.e., UML class models), improving their documentation. Also, we claim that as we have defined all refactoring based on the KDM, they can be easily reused by others researchers.

It is important to notice that the application of refactorings in UML class diagrams is not a new research as stated before. However, all of the works we found on literature perform the refactoring directly on the UML metamodel. Although UML is also an ISO standard, its primary intention is just to represent diagrams and not all the characteristics of a system. As KDM has been created to represent all artifacts and all characteristics of a system, refactorings performed on its finer-grained elements can be propagated to higher level elements. This propitiates a more complete and manageable model-driven modernization process because all information is concentrated in just one metamodel. In terms of the the users who uses modernization tools like ours, the difference is not noticeable; that is, whether the refactorings are performed over UML or KDM. However, there are two main benefits of developing a refactoring catalogue for KDM. The first one is in terms of reusability. Other modernizer engineers can take advantage of our catalogue to conduct modernizations in their systems. The second benefit is that, unlikely UML, a catalogue for KDM can be extended to higher abstractions levels, such as architecture and conceptual, propitiating a good traceability among these layers.

We believe that KDM-RE makes a contribution to the challenges of Software Engineering which focuses on mechanisms to support the automation of model-driven refactoring. Future work involves implementing more refactorings and conducting experiments to evaluate all refactorings provided by KDM-RE. Doing so, we hope to address a broader audience with respect to using, maintaining, and evaluating our tools. Currently, KDM-RE generates only class diagrams to assist the modernization engineer to perform refactorings, however, as future work, we intend to: (*i*) extend this computational support to enable the achievement of other diagrams, e.g., the sequence diagram, (*ii*) perform structural check of the software after the application of refactorings; and (*iii*) carry out the assessment tool, as well as refactorings proposed by controlled experiments. A work that is already underway is to check how other parts of the highest level of KDM are affected after the application of certain refactorings. For example, assume that there are two packages P1 and P2. Suppose there is a class in P1, named C1, and within the P2 there is a class named C2. Assume that C1 owns an attribute A1 of the type C2., i.e., there is an association relationship between these classes of different packages. P1 and P2 represent architectural layers, i.e., P1 = Model and P2 = View. Thus, the relationship that exists is undesirable. When we make a fine-grained refactoring such as moving the attribute A1

of the class C1, it should be reflected to the architectural level, eliminating the existing relationship between the two architectural layers.

6. Acknowledgements

Rafael S. Durelli would like to thank the financial support provided by FAPESP, process number 2012/05168-4. Bruno Santos and Raphael Honda also would like to thank CNPq for sponsoring our research.

References

Durelli, R. S., Santibáñez, D. S. M., Delamaro, M. E., and Camargo, V. V. (2014). Towards a refactoring catalogue for knowledge discovery metamodel. In *IEEE 15th International Conference on Information Reuse and Integration (IRI)*.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (2000). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gorp, P. V., Stenten, H., Mens, T., and Demeyer, S. (2003). Towards automating source-consistent uml refactorings. In *International Conference on UML - The Unified Modeling Language*, pages 144–158. Springer.

Misbhauddin, M. and Alshayeb, M. (2012). Model-driven refactoring approaches: A comparison criteria. In *Sofware Engineering and Applied Computing (ACSEAC), 2012 African Conference on*.

OMG (2012). Object Management Group (OMG) Architecture-Driven Modernisation. Disponível em: *http://www.omgwiki.org/admtf/doku.php?id=start*. (Acessado 2 de Agosto de 2012).

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. Ph.D. Thesis, University of Illinois.

Perez-Castillo, R., de Guzman, I. G.-R., Avila-Garcia, O., and Piattini, M. (2009). On the use of adm to contextualize data on legacy source code for software modernization. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 128–132, Washington, DC, USA. IEEE Computer Society.

Reimann, J., Seifert, M., and Abmann, U. (2010). Role-based generic model refactoring. In *In ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013)*. Springer.

Thorsten Arendt, Timo Kehrer, G. T. (2013). Understanding complex changes and improving the quality of uml and domain-specific models. In *In ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013)*.

Ulrich, W. M. and Newcomb, P. (2010). *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Anexo 10: Comprovante da publicação no *International Conference on Enterprise Information Systems, 2013* (ICEIS 2013)

- Uma cópia do artigo é apresentado a seguir (a partir da próxima página).

Concern-based Refactorings Supported by Class Models to Reengineer Object-Oriented Software into Aspect-Oriented Ones

Paulo Afonso Parreira Júnior[1], Matheus Carvalho Viana[1], Rafael Serapilha Durelli[2], Valter Vieira de Camargo[1], Heitor Augustus Xavier Costa[3] and Rosângela Aparecida Dellosso Penteado[1]
[1]*Departament of Computer Science, Federal University of São Carlos, São Carlos, Brazil*
[2]*Computer Systems Department, University of São Paulo, São Carlos, Brazil*
[3] *Departament of Computer Science, Federal University of Lavras, Lavras, Brazil*
{paulo_junior, matheus_viana, valter, rosangela}@dc.ufscar.br, heitor@dcc.ufla.br, rdurelli@icmc.usp.br

Keywords: Concern-based Refactorings, Class Models, Aspect-Orientation, Reengineering.

Abstract: Reengineering Object-Oriented Software (OO) into Aspect-Oriented Software (AO) is a challenging task, mainly when it is done by means of refactorings in the code-level. The reason for it is that direct transformations from OO code to AO one involve several design decisions due to syntactic and semantic differences of both paradigms. To make this task more controlled and systematic, we can make use of concern-based refactorings supported by models. This type of refactorings concentrates on transforming broader scenarios into a set of context-dependent scenarios, rather than specific ones, as in code-level refactorings. In this paper we propose a set of concern-based refactorings that allows design decisions to be made during the reengineering process, improving the quality of the final models. Two of them are presented in more details in this paper. An example is presented to assess the applicability of the proposed refactorings. Moreover, we also present a case study, in which AO class models created based on the refactorings are compared with AO class models obtained without the aid of these refactorings. The data obtained in this case study indicated to us that the use of the proposed refactorings can improve the efficacy and productivity of a maintenance group during the process of software reengineering.

1 INTRODUCTION

Aspect-Orientation (AO) can be used in the revitalization of Object-Oriented (OO) legacy software. AO allows encapsulating the so-called "crosscutting concerns" (CCC) - software requirements whose implementation is tangled and scattered by functional modules - in new abstractions such as pointcuts, aspects, advices and inter-type declarations (Kiczales et al., 1997).

Reengineering from OO to AO in code-level is not an easy task due to existing differences between concepts related to both approaches. However, if the reengineering process was supported by models, it could facilitate future maintenance. In this paper we propose the use of concern-based refactorings on OO class models annotated with information of CCC to obtain AO models. In the context of this paper, annotated OO class models are UML OO class models whose elements (classes, interfaces, attributes and methods) are annotated with stereotypes corresponding to the CCC that exist in the software. The main idea is that concern-based refactorings can be applied to transform these models into AO models.

There are many studies in the literature that present code-based refactorings (Silva *et al*., 2009; Monteiro and Fernandes, 2006; Hannemann *et al*., 2005; Marin *et al*., 2004; Iwamoto and Zhao, 2003). Our main reasons to create and apply **concern-based refactorings supported by models** ("model-based refactorings" in the rest of this paper) are:

i) code-level refactorings can be applied to transform OO software in AO ones. However, this transformation is usually done in one step, which has as input an OO code and as output an AO one. It makes the reengineering process less flexible, because the responsibility to generate a code that follows good design practices of AO is on the refactorings. The transformation based on model-based refactorings introduces at least one

more step in the process before generating the final code. Thus, to ease the inflexibility of the process, in this step the outcome AO model can be modified by the software engineer according to the environment and stakeholder requirements;

ii) generally, the source code is the only available artifact of the legacy software. Applying model-based refactorings, both the legacy software and the generated software will have a new type of artifact (*i.e.*, UML class models), improving their documentation; and

iii) unlike the code-based refactorings, model-based ones are platform independent. Thus, models can be transformed and good designs can be produced regardless of programming language.

A set of nine model-based refactorings was developed. It is subdivided into: i) three generic refactorings, which are concern-independent refactorings; and ii) six specific refactorings to the following concerns: persistence (subdivided into connection, transaction and synchronization management), logging and Singleton and Observer design patterns (Gamma et al., 1995). Due to the limitation of space, only two of them are discussed in more details in this paper. The AO class models presented in this paper are based on AOM (Aspect-Oriented Modeling) approach proposed by Evermann (Evermann, 2007).

The remainder of this paper is structured as follows. Some concepts related to the AOM approach proposed by Evermann, the annotated OO class models and the computational support DMAsp (Costa *et al*., 2009), used to generate automatically annotated OO class models, are discussed in Section 2. The generic and specific refactorings are presented in Section 3. An example that illustrates the use of these refactorings is shown in Section 4 and an evaluation of them is presented in Section 5. Some related works are summarized in Section 6. Finally, conclusions and suggestions for future work are presented in Section 7.

2 BACKGROUND

ProAJ/UML (UML Profile for AspectJ) is one of the most used approaches to model AO software (Evermann, 2007). This approach consists of a set of stereotypes that can be applied on UML class models, such as:

- `<<CrossCuttingConcern>>`: it is an extension of the *Package* meta-class, in the UML meta-model. Its aim is to encapsulate aspects related to the same crosscutting concern.
- `<<Aspect>>`: it extends the UML *Class* meta-class. Its goal is to cluster *pointcuts* and *advices* in an aspect and to allow aspects to extend classes or aspects and implement interfaces.
- `<<Advice>>`: it is a *BehavioralFeature* meta-class extension. Its aim is to associate advices with aspects.
- `<<PointCut>>`: it is a *StructuralFeature* meta-class extension, whose goal is to specify a static behavior. Its modelling is performed by concrete subclasses of *PointCut*, such as *CallPointCut* and *ExecutionPointCut*.

These stereotypes are used in the AO class models generated with the application of the refactorings proposed in this work. Furthermore, OO class models annotated with information of CCC are used in the proposed refactorings. These annotations are represented using stereotypes on the left side of the classes, interfaces, attributes and methods identifiers. Figure 1 illustrates a class annotated with indications of persistence CCC. The DMAsp (Design Model to Aspect) tool (Costa *et al*., 2009), developed in a previous work, is used to generate automatically the annotated OO class models.

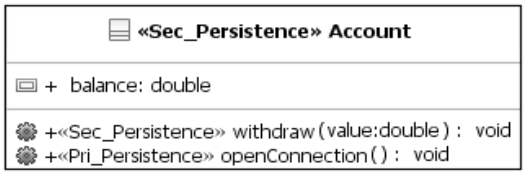


Figure 1: An UML Class Annotated with Information about Persistence CCC.

Based on the concept of annotated OO class models, the following concepts, proposed by Figueiredo *et al*. (2009), were adapted to the context of this work and are commented in the refactoring descriptions.

- **Components affected by a concern** are software elements such as classes, interfaces, attributes and methods which have indications of this concern. These elements are annotated with stereotypes of the concern that affect them.
- **Primary concern** is the main concern of a component and it is related to the reason by which it was created. For example, the `openConnection` method (Figure 1) was created to open database connections. Then "Persistence" is the primary concern of this method. The primary concerns are identified by the prefix "Pri_" in the stereotypes.
- **Secondary concern** of a component corresponds to functions that this component plays. However,

these functions are not directly related to the reason for which it was created. In Figure 1, the `Account` class and its method `withdraw` were created to perform the business rules of a hypothetical banking system. Thus "Persistence" is a secondary concern in these components. The secondary concerns are identified by the prefix "Sec_" in the stereotypes.

- **Well-modularized components** are software elements composed only by the primary concern for which they were created. For example, the `openConnection` method (Figure 1) is considered well-modularized, because the only type of stereotype of this method is a primary concern related to "Persistence" concern.

3 MODEL-BASED REFACTORINGS

Hannemann (2006) proposed the following classification of AO software refactorings:

i) **Conventional OO Refactorings adapted for AO Software**. These refactorings only involve OO elements. The difference between these refactorings and the well-known OO refactorings is they are aware of the existence of AO elements;

ii) **Specific Refactorings for AO Software**. These refactorings involve OO and AO elements and they are specific to lead to the AO abstractions, such as aspects, pointcuts, etc; and

iii) **Crosscutting Concerns Refactorings**. Also called concern-based refactorings, they should take all the elements (classes, aspects, interfaces, etc) that participate in a crosscutting concern and their relationships into consideration. This happens, because concerns usually are manifested in several components.

A set of nine concern-based refactorings is shown in Table 1, but only two of them are presented in this paper in more details. The remaining refactorings were omitted for reasons of limitation of space and can be found in (Parreira Júnior, 2011).

The following information of refactorings are presented: i) Acronym and Name of the Refactoring; ii) Application Scenario, which defines the situations the refactoring can be applied; iii) Motivation, which presents some problems caused by tangling and scattering of CCC; and iv) ProAJ/UML Mechanism, which is a set of steps to obtain an AO class model from an OO one, according to the ProAJ/UML profile.

Table 1: Model-based refactorings.

Generic Refactorings	
Name	**Description**
R-1	Encapsulating a Primary Concern that does not have Generalization / Specialization relationships.
R-2	Encapsulating a Primary Concern that has Generalization / Specialization relationships.
R-3	Extracting a Primary Concern.
Specific Refactorings	
Name	**Description**
R-Connection	Encapsulating the CCC responsible for managing database connections.
R-Transaction	Encapsulating the CCC responsible for managing database transactions.
R-Sync	Encapsulating the CCC responsible for managing database synchronization.
R-Logging	Encapsulating the CCC responsible for controlling the application of logging record.
R-Singleton	Encapsulating the CCC corresponding to the Singleton design pattern.
R-Observer	Encapsulating the CCC corresponding to the Observer design pattern.

3.1 Generic Refactorings

The generic refactorings are responsible for transforming an annotated OO class model to a partial AO class model. The generated model is named "partial", because the existing CCC may not be well-modularized yet. In this case, there still can exist classes/interfaces, methods and/or attributes affected by crosscutting concerns. One of the generic refactorings, *R-3*, is presented as follows.

R-3. Extracting a Primary Concern.
Application Scenario: when there are classes with Secondary Concerns, which are Crosscutting Concerns and these ones are not Primary Concerns in any classes.

Motivation: some crosscutting concerns can be scattered in several classes and there are not specific classes that implement them. One concern of this type is not a Primary Concern in any class of the application. This scenario represents a high level of concern tangling and a low level of software modularization.

ProAJ/UML Mechanism: 1) Create a *CrossCuttingConcern* element called "CCC", in which "CCC" represents the concern name that is being modularized; **2)** Inside the element created previously, add an *Aspect* element called "CCCAspect"; and **3)** Move each well-modularized attribute and method from the classes affected by the concern to the "CCCAspect" element.

Looking at the application scenario of *R-3* refactoring, we understand that: "no matter what concern we are dealing, the scenario described

above represent a low level of modularization". Thus, we already can apply a modularization strategy to this concern, whatever it is, putting all the well-modularized elements (attributes and methods) related to this concern in a specific module, in this case, an aspect. Only well-modularized elements are moved to the aspect, avoiding problems related to the dependence of other concerns.

3.2 Specific Refactorings

The specific refactorings are responsible for transforming partial AO class models in final ones. These refactorings are named "specific", because they only can be applied to a specific type of concern. For example, there is a specific refactoring to the transaction management concern that generates an AO class model with the modularization of this concern using aspects. Six specific refactorings were developed, as presented in Table 1.

These refactorings were created based on the most common strategies for implementing these types of crosscutting concerns. For example, the database connection concern is usually implemented with a class responsible for creating connections and each persistent method must open the connection at the beginning of its execution and close it at the end. In another example, the singleton pattern is generally implemented as follows (Gamma et al., 1995): i) create an attribute of the same type of the Singleton class; ii) become private the constructor of the Singleton class; and ii) create a method responsible for keeping only one instance of the Singleton class. Therefore, it is possible to define some steps for modularization of this type of concern, based on the most common strategies for implementing them.

The specific refactorings are applied on the models generated by the generic refactorings. Thus, in ProAJ/UML Mechanism description, aspects created previously are mentioned. To illustrate this case the refactoring *R-Singleton* is presented.

Unlike the generic refactorings, in this case, we can use some more specific steps to modularize the CCC, because Singleton pattern is a commonly known concern. Thus the aspect created by a generic refactoring has been transformed into an abstract one and for each class affected by the Singleton concern one aspect has been created. This strategy follows a good practice for AO design suggested by Piveta *et al*. (2007). Furthermore, it is similar to Hanneman and Kiczales' (2002) solution and was adapted to the context of annotated OO class models.

***R-Singleton*. Encapsulating the CCC corresponding to the Singleton design pattern.**
Application Scenario: when there are classes dedicated to implementation of Singleton pattern.
Motivation: the Singleton pattern can cause problems of tangling and scattering of concerns in OO application. The modularization using AO is one alternative to solve these problems (Hannemann and Kiczales, 2002).
ProAJ/UML Mechanism: 1) Identify the "CCCAspect" aspect related to implementation of the singleton concern and verify if this aspect is abstract. If not, transform it into abstract one; **3)** Create, inside the element "CCC" related to the singleton concern, an empty interface called `Singleton`; **4)** Define an execution pointcut called `instance` that intercepts the calls to constructor of the classes that realize the `Singleton` interface and add it to the "CCCAspect"; **5)** Identify the set of classes, "S", that implement the Singleton pattern, *i. e*., the classes whose instance must be unique in the application. If the constructor of these classes be *private*, transform it into *public*; **6)** For each class "N" ϵ "S", create an aspect "CCCAspectN", where "N" corresponds to the class name and create inheritance relationships from the aspects "CCCAspectN" to the aspect "CCCAspect"; **7)** Each aspect "CCCAspectN" created previously must declare an interface realization relationship between the class "N", represented by this aspect, and the interface `Singleton`; and **8)** Create an around advice that returns a `Singleton` object. This advice implements the logic of the Singleton pattern: if there exists an instance, return it; otherwise, create one instance and return it. Associate the around advice to the `instance` pointcut.

3.3 Considerations about the Refactorings

Some of the main reasons to apply generic refactorings are: **i) the application of generic refactorings can facilitate the achievement of a better AO model**: wrong decisions made by software engineers, due to their inexperience, can prejudice the AO model quality. Thus, an initial modularization strategy offered by these refactorings can minimize this problem; and **ii) generic refactorings can be applied to any type of concern, even to those concerns that are not widely known as crosscutting concerns**: it is not easy to identify whether a particular concern is or

not a crosscutting concern. Thus, with the help of generic refactorings, we can identify scenarios that demonstrate or provide evidence of the existence of crosscutting concerns in software. For example, the application scenario for the refactoring *R-3* states a configuration that can evidence the existence of a crosscutting concern (many classes of software related to a secondary concern in these classes).

There is not a specific sequence to apply generic refactorings proposed in this work. The steps created for refactoring are applied when a specific element is well-modularized, *i.e.*, when there is no interference of other concerns in this element. Moreover, some modularization strategies described in the steps of the refactoring were considered to avoid interference in the order of execution of the refactoring. Similarly to what happens with the generic refactorings, the order in which the specific refactorings are applied does not interfere in the final AO class model. It happens because each refactoring acts only on a particular concern at a time, not compromising elements related to other concerns.

The manual execution of the steps described in the refactoring presented on class models of software for medium and large scale can be hard and error-prone. Thus, an Eclipse plug-in called MoBRe (Model-Based Refactorings) was developed to perform tasks related to refactoring of crosscutting concerns in a semi-automatic way. MoBRe (Parreira Júnior et al., 2011) allows transforming an annotated class model into a partial AO class model, when the generic and specific refactorings are applied. The AO class models generated can be visualized within the Eclipse.

4 EXAMPLE OF USE

To verify the applicability of the proposed refactorings, an example, using the Health Watcher software (Soares et al., 2002), is presented. This software registers complaints in the health area and it was chosen because it: i) has an OO and an AO version; and ii) was modularized by expert software engineers by using best practices of AO design.

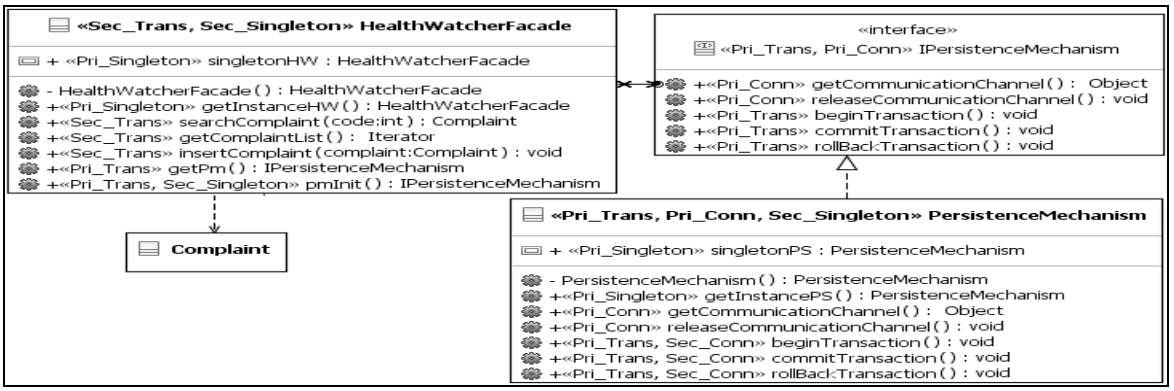


Figure 2: A UML Class Stereotyped with CCC Indications.

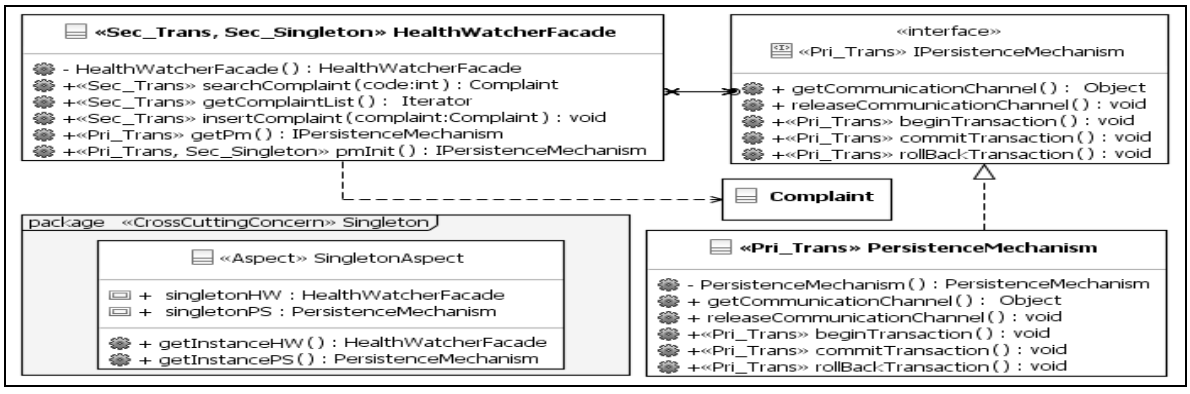


Figure 3: Health Watcher AO Class Model obtained through *R-3* Refactoring.

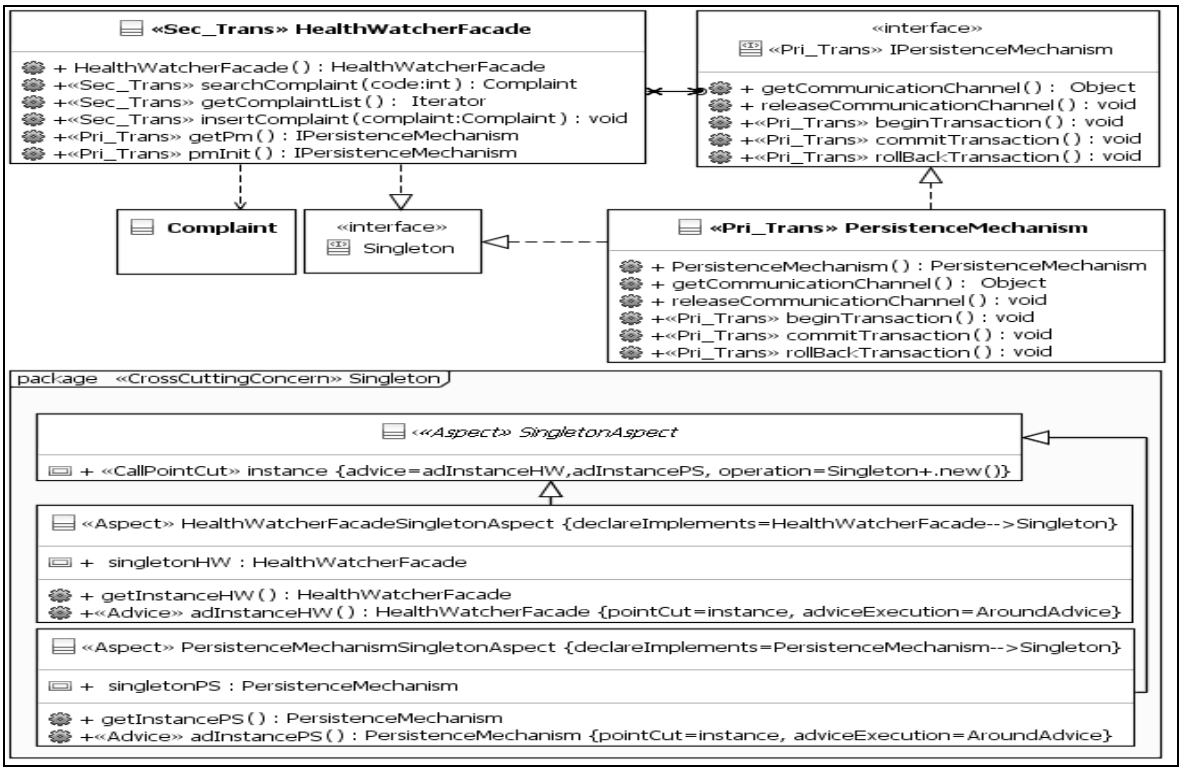


Figure 4: Health Watcher AO Class Model obtained through *R-Singleton* Refactoring.

The crosscutting concern modularized in this example is the Singleton pattern, represented by the "Singleton" stereotype. Other crosscutting concerns affect this application, such as connection and transaction management, represented by the "Conn" and "Trans" stereotypes, but the modularization of them is not performed in this paper because of limitations of space.

One part of Health Watcher OO class model, responsible for the maintenance of the patient complaints, is presented in Figure 2. This model is annotated by using stereotypes of the concerns that affect the software classes, according to the information provided by Soares et al. (2002).

The `HealthWatcherFacade` class provides methods necessary for execution of the business logic of the application, as complaints registration, diseases, and symptoms. The `singletonHW` and `singletonPS` attributes and the `getInstanceHW` and `getInstancePS` methods have "Singleton" as Primary Concern, because they were created specifically for implementing the Singleton pattern. The same way, "Conn" and "Trans" are Primary Concerns of the `IPersistenceMechanism` interface and the `PersistenceMechanism` class, because they were created for implementing these concerns. The `HealthWatcherFacade` class has "Trans" and "Singleton" as Secondary Concerns, because this class was not created for implementing these concerns, but it is affected by them. This information about what concerns are primary or secondary one was provided by the Health Watcher developers.

According to the scenario of tangling/scattering of the model presented in Figure 2, the singleton concern can be initially refactored by the *R-3* refactoring. This happens because "Singleton" is a Secondary Concern in some classes of this model and it is not a Primary Concern in none other classes.

After applying the *R-3* refactoring to the "Singleton" concern, the partial AO class model presented in Figure 3 was obtained. The changes made were: i) the `SingletonAspect` aspect was created; and ii) the `singletonHW` and `singletonPS` attributes and the `getInstanceHW` and `getInstancePS` methods were moved to the `SingletonAspect` aspect. It is because these elements are well-modularized in the `Health-WatcherFacade` and `PersistentMechanism` classes.

The `pmInit` method of the `HealthWatcherFacade` class continues being affected to the "Singleton" concern. To eliminate this influence, the *R-Singleton* refactoring has been used on the model in Figure 3 and the resulting model is presented in Figure 4.

The changes were:

- An interface called `Singleton` was created; The `SingletonAspect` aspect became abstract and a pointcut called `instance` was added to it. This pointcut intercepts the calls to constructor of the classes whose instances must be unique;
- Two new aspects that extend `Singleton-Aspect` were created: `SingletonAspect-HealthWatcherFacade` and `Singleton-AspectPersistenceMechanism`;
- For each aspect created: i) attributes and methods corresponding to each Singleton class were moved to it; ii) advices responsible for returning an instance of the Singleton class when the pointcut `instance` is reached were created; and iii) declare parents structures were created to associate each class related to Singleton concern with `Singleton` interface; and
- The modifiers of the constructors of the `HealthWatcherFacade` and `Persistence-Mechanism` classes were changed from `private` to `public`. It was done so that the instance of a Singleton class is obtained using its constructor and not by `getInstance` method.

5 EVALUATION

The crosscutting concern modularization may be performed with or without the assistance of refactorings. In the second case, the process of modularization is extremely dependent on the expertise of the software engineer. He/She must have knowledge about the crosscutting concerns to be modularized and best practices and strategies for the modularization of these concerns. Refactorings minimize this dependence, making the final product (modularized software) more standardized and improving its quality.

The question we want to answer with this case study is: *how much can the refactorings affect the efficacy of the modularization process and the productivity of the maintenance group?* In this context, productivity is defined as the time that a group takes to modularize the crosscutting concerns of a software product. Besides, efficacy consists in verifying whether all crosscutting concerns were suitably modularized or not. Thus, the case study was carried out and it is shown in the next subsections.

5.1 Case Study Definition

The efficacy and productivity evaluation of the refactorings was performed in two ways:

i) comparing the generated AO class models with another version of them, obtained from a reverse engineering using the AO code found in the literature (in this study, we use the JSpider AO code available in (JSpider, 2013)). To do this, a set of seven **Metrics for Modularization** were used to compare both versions of the application AO class model (Table 2). All of them, except *MQ* and *AVG(MQ)*, accept the following values: 1.0 – Completely Compliant; 0.5 - Partially Compliant; and 0.0 – Not Compliant. These values are assigned to the metrics for modularization by specialists after comparing the models created by the participants of this experiment to the models obtained from the literature. The metrics *MQ* and *AVG(MQ)* accepts values between [0.0; 5.0] and the higher the value of them, the better is the modularization of a concern; and

ii) comparing the time spent by each participant to complete the modularization of a given OO class model. For this, we used the metric *Productivity* (*Pr*), given to the Formula (1). The higher the value of *Pr*, the better is the productivity of a participant.

5.2 Case Study Planning

a) Selection of Context and Formulation of Hypothesis. The study was carried out with graduate students at the Federal University of São Carlos. The system used as object of study was JSpider (2013), a highly configurable and customizable Web Spider engine. The participants had to modularize the Logging and Singleton crosscutting concerns and generate an AO class model from the OO model classes of the JSpider application.

Four hypotheses were elaborated (Table 3), two of which refer to the efficacy and two ones refer to the productivity. Besides, the metrics *MQ* and *Pr* were used for formulating the hypotheses.

b) Selection of Variables and Participants. Independent variables are those manipulated and controlled during the experiment. In this context, they are the way how the participants performed the

modularization: with or without the use of the refactorings.

Dependent variables are those under analysis, whose variations must be observed. In this experiment, they are the efficacy and productivity. The participants were selected through a convenient non-probabilistic sampling.

Table 2: Metrics for Modularization.

Metric	Metric Description
CC_As	Correctly Created Aspects: specifies if all needed aspects were correctly created.
CM_AM	Correctly Modularized Attributes and Methods: specifies if all attributes and methods affected by a concern were correctly modularized.
CC_PA	Correctly Created Pointcut and Advices specifies if all needed pointcuts and advices were correctly created.
CC_GSR	Correctly Created Generalization and Specialization Relationships: specifies if all needed relationships were correctly created.
CS_PC	Correctly Specified Profile Concepts: specifies if all ProAJ/UML concepts were correctly used.
MQ	Modularization Quality: CC_As + CM_AM + CC_PA + CC_GSR + CS_PC.
AVG(MQ)	Average of the Metric *MQ*.

$$Pr = AVG(MQ) / T, \quad (1)$$

where *AVG(MQ)* is the average of the metric *Modularization Quality* and *T* is the time (in hours) spent by a participant to modularize the crosscutting concerns.

Table 3: Hypotheses of the case study.

Hypotheses for Efficacy	
H_{0Ef}	There is no difference of using refactorings or not using them, regarding the efficacy. H_{0Ef}: $MQ_{WR} = MQ_{WOR}$
H_{1Ef}	There is difference of using refactorings or not using them, regarding the efficacy. H_{1Ef}: $MQ_{WR} \neq MQ_{WOR}$
Hypotheses for Productivity	
H_{0Pr}	There is no difference of using refactorings or not using them, regarding the productivity. H_0: $Pr_{WR} = Pr_{WOR}$
H_{1Pr}	There is no difference of using refactorings or not using them, regarding the productivity. H_0: $Pr_{WR} \neq Pr_{WOR}$
Legends: • X_{WR}, where X is a metric, means: the value of X obtained by a specific participant using the refactorings proposed in this paper (WR = With Refactorings). • X_{WOR}, where X is a metric, means: the value of X obtained by a specific participant that did not use the refactorings proposed in this paper (WOR = Without Refactorings).	

c) Design of the Experiment. The distribution of the participants in groups was done by using a profile characterization questionnaire. The questions were about their level of experience in OO and AO, modularization and UML profile to modelling AO software. All questions had the possible answers: 1 - None; 2 - Basic; 3 - Medium; 4 - Advanced; 5 - Expert. The obtained values are plotted in the graph shown in Figure 5.

The groups were created as follows: Group A – participants P1 to P5; Group B – participants P6 to P10. The average of expertise of Group A is approximately 1.86 and Group B, 1.80, representing that the groups were balanced. To separate the experts and novices we have defined the value 1.75 (horizontal line in Figure 5). This value was defined according to our experience with the required knowledge to perform the modularization of CCCs. Above this value the participants were considered experts (P1, P2, P6 and P9) and below novices (P3, P4, P5, P7, P8 and P10). It is important to notice that both groups have the same number of expert and novice participants.

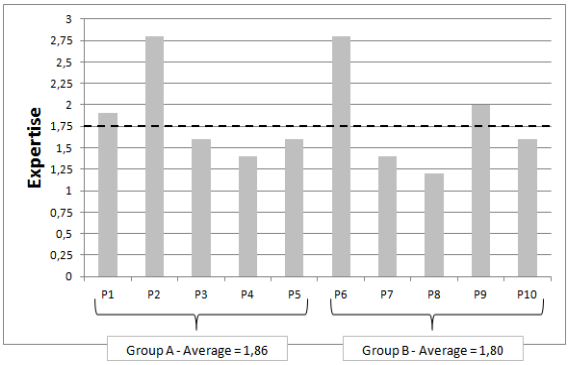


Figure 5: Participants Expertise.

The documents used in this experiment were: i) a registry form to be filled out with information related to the execution of the study; ii) a script of execution with the steps to be followed to perform the experiment; and iii) the description of the proposed refactorings. The registry form contained the participant name, the application to be modularized, the starting time and the observations and/or problems noticed by the participant. The script of execution contained a list of tasks that the participants should carry out and had the goal of assisting them and minimizing the possibility of failures during the execution. The description of the proposed refactorings presents the refactoring according the template used in Section 3.

Table 4 shows the experimental design. The experiment was divided in three phases. In the first phase (*Training*), we conducted a training aimed at homogenizing the knowledge of the participants on the modularization of crosscutting concerns using hypothetical applications.

In the second phase (*Pilot*) all participants had to discover how to modularize the persistence concern that crosscuts pieces of the HealthWatcher application manually and using the proposed refactorings. The goal of the pilot was to minimize the difficulties of following the steps described in

the refactorings. Besides, the pilot also was intended to avoid that problems related to the filling out of the forms could interfere in the results of the experiment.

Table 4: Design of the case study.

Phases	Software Used	With Refactorings	Without Refactorings
1st. Phase (Training)	Toy Applications	All participants	All participants
2st. Phase (Pilot)	Health Watcher	All participants	All participants
3nd. Phase (Execution)	JSpider	Group A	Group B

In the third phase (*Execution*) the goal was to modularize the Logging and Singleton concerns in the JSpider application. Different types of concern between *Execution* and *Pilot* phases were used to avoid that the knowledge on the persistence concern obtained in the previous phase (*pilot*) to influence the results.

d) Collected Data. Table 5 and 6 show the data obtained in third phase of the experiment (*Execution*) by the Groups A and B, respectively (AoE means *Average of the Experts* and AoN means *Average of the Novices*).

The participants, assigned by "P#", and the titles of the table columns are presented in first line. The time used by the participants for performing the modularization is presented in lines from 2 to 3. The concern names are presented in lines 3 and 10 and the values of the metrics described in Table 2 are presented in lines from 5 to 10 (for the Singleton concern) and from 12 to 17 (for the Logging concern). The average of the metric *MQ* and the value of the metric *Pr* are presented in lines 19 and 20. The columns that contain the data of participants classified as experts were highlighted in gray color.

e) Data Analysis. Regarding the *Efficacy* and *Productivity*, Tables 5 and 6 show that most of participants that used the proposed refactorings got better results than those ones that do not used them.

Regarding the *Productivity*, just one of the participants that used the proposed refactorings was less productive. Although there was an extra task to be carried out (to follow the steps described in the refactorings), the developer will need to modify the models during the process of refactoring less times, thus minimizing the final time to perform the activity.

As it can be observed in Figure 6, the average of metric *MQ*, when the participants had the aid of the refactorings was higher than the one when they did not have this aid. According to this chart, the value of the metric *MQ* was 206% higher, in average, for all participants, and 344% higher, in average, for participants classified as experts and 150% higher, in average, for participants classified as novices).

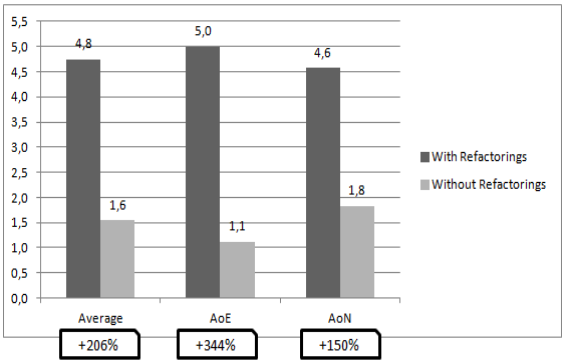


Figure 6: Average of the Metric *MQ*.

Table 5: Execution of the case study – Group A (with refactorings).

Data	P1	P2	P3	P4	P5	Average	AoE	AoN
Time (min)	35	47	60	60	60	52	41	60
Time (hours)	0,58	0,78	1,0	1,0	1,0	0,87	0,68	1,0
Singleton Pattern								
CC_As	1,0	1,0	1,0	1,0	1,0			
CM_AM	1,0	1,0	1,0	1,0	1,0			
CC_PA	1,0	1,0	0,5	1,0	0,5			
CC_GSR	1,0	1,0	1,0	1,0	1,0			
CS_PC	1,0	1,0	1,0	1,0	1,0			
MQ	**5,0**	**5,0**	**4,5**	**5,0**	**4,5**	**4,8**	**5,0**	**4,7**
Logging								
CC_As	1,0	1,0	1,0	1,0	1,0			
CM_AM	1,0	1,0	1,0	1,0	1,0			
CC_PA	1,0	1,0	0,5	0,5	0,5			
CC_GSR	1,0	1,0	1,0	1,0	1,0			
CS_PC	1,0	1,0	1,0	1,0	1,0			
MQ	**5,0**	**5,0**	**4,5**	**4,5**	**4,5**	**4,7**	**5,0**	**4,5**
Results								
AVG(MQ)	**5,0**	**5,0**	**4,5**	**4,8**	**4,5**	**4,8**	**5,0**	**4,6**
Pr	**8,57**	**6,38**	**4,5**	**4,8**	**4,5**	**5,75**	**7,48**	**4,6**

Analogously, in Figure 7, the average of metric *Pr* also was better when the participants used the refactorings. This chart presents productivity 124% higher, in average, for all participants, and 115% higher, in average, for participants classified as experts and 134% higher, in average, for participants classified as novices).

It is also possible to notice in Figure 6 and 7 that the refactorings helped more expert participants than non-expert ones. It happened maybe because the description of the proposed refactorings was not well detailed enough to guide non-expert participants to modularize the concerns correctly.

Table 5: Execution of the case study – Group B (without refactorings).

Data	P6	P7	P8	P9	P10	Average	AoE	AoN
Time (min)	38	45	60	60	30	46	49	45
Time (hours)	0,63	0,75	1,0	1,0	0,5	0,78	0,69	0,83
Singleton Pattern								
CC_As	1,0	1,0	0,0	0,0	1,0			
CM_AM	1,0	1,0	0,0	0,0	1,0			
CC_PA	0,0	0,0	0,0	0,0	0,0			
CC_GSR	0,0	0,0	0,0	0,0	0,0			
CS_PC	0,0	0,0	0,0	0,0	1,0			
MQ	**2,0**	**2,0**	**0,0**	**0,0**	**3,0**	**1,4**	**1,0**	**1,7**
Logging								
CC_As	1,0	1,0	0,5	0,0	1,0			
CM_AM	1,0	1,0	0,0	0,0	0,5			
CC_PA	0,5	1,0	0,0	0,0	0,0			
CC_GSR	0,0	0,0	0,0	0,0	0,0			
CS_PC	0,0	0,0	0,0	0,0	1,0			
MQ	**2,5**	**3,0**	**0,5**	**0,0**	**2,5**	**1,7**	**1,3**	**2,0**
Results								
AVG(MQ)	**2,3**	**2,5**	**0,3**	**0,0**	**2,8**	**1,6**	**1,1**	**1,8**
Pr	**3,63**	**3,33**	**0,3**	**0,0**	**5,6**	**2,57**	**3,48**	**1,97**

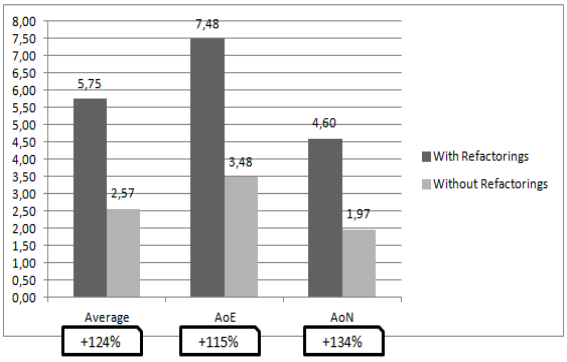


Figure 7: Average of the Metric *Pr*.

f) Hypothesis Testing. After outlier analysis, it was noticed that none outlier was identified and the hypotheses tests were performed. The verification of the normality of the distribution sample data was made using the non-parametric test called Shapiro-Wilk (Montgomery, 2000).

The aim of the hypothesis test is to verify if the null hypothesis (H_{0Ef} and H_{0EPr}) can be rejected, with some significance degree, in favor of an alternative hypotheses (H_{1Ef} or H_{1Pr}) based in the set of data obtained.

The *t-test* was applied to the set of sample data in two stages, because of the existence of two dependent variables, *Efficacy* and *Productivity* were observed. In first stage, the sample relative to the values of the metric *Pr* was compared. In second one, the comparison was made using samples referring to the values of *MQ* metric. For the purpose of this study, the minor degree of significance α was used in both stages to reject the null hypothesis and the maximum degree of significance equal to 5% was considered.

First stage. Based in two independent samples (Pr_{WR} and Pr_{WOR}) with averages equals to 5.75 and 2.57, respectively in Tables 5 and 6, the null hypothesis (H_{0Pr}) could be rejected with 0.0151% of significance. In others words, it is possible to assure with 99.9% of accuracy that the average of the values of the productivity obtained by the participants that used the refactorings is different.

Second stage. Based in two independent samples (MQ_{WR} and MQ_{WOR}) with averages equals to 4.8 and 1.6, respectively, the null hypothesis (H_{0Ef}) could be rejected with 0.0007% of significance. In others words, it is possible to assurance with 99.9% of accuracy that the average of the values of the efficacy obtained using the refactorings is different as compared to not using the refactorings.

With the rejection of H_0, it can be stated that the observed differences in the average of *efficacy* and *productivity* of the participants who used the refactorings and participants who have not use them, have statistical significance. Thus, the change in efficacy and productivity of the groups was due to the strategies for software modularization adopted in the experiment, *i.e.*, with or without refactorings.

As presented in Figures 6 and 7, the average value of the metric *MQ* of the participants who used the refactorings was higher than that of the participants who have not used ($MQ_{WR} > MQ_{WOR}$). These data show that the use of refactorings for modularization of crosscutting concerns is generally more effective than when such refactoring are not used.

Analogously, with respect to productivity, it was expected that the systematic description of the steps of refactorings becomes more agile the execution of the participants' tasks. Based on the data and hypothesis test, there are evidences that the use of refactorings can increase the productivity of a group.

The analysis of the data was accomplished using a statistical plug-in to the Excel called Analyse-it (2013).

g) Threats to the Validity of the Study. *Concluding Validity*: the *t-test* was adopted because our study was a project with one factor with two treatments. This is the most suitable test for projects with this configuration, which the aim is to compare the obtained averages from two distinct treatments. The *t-test* usually requires normally distributed data. So, the Shapiro-Wilk test was applied and the result

was positive for our study.

Internal Validity: a point that may have influenced the results of the experiment is the use of graduate students as participants. However, they were not influenced by the conductors of this study and we did not show any expectation in favor or against the refactorings proposed in this paper. Besides, the students were properly grouped according to their experience levels in order to have homogeneous groups. This was done to avoid that a group could finish the tasks much earlier than other group. The participants did not receive any grade for the participation.

External Validity: an important bias is the choice of the concerns to be modularized in the experiment. Different types of concern were used to avoid that the knowledge on a specific concern obtained in the training phase to influence the results in other phases of the experiment. Another bias in this case study is that the proposed refactorings have been applied in software of fairly small size that cannot reflect the real scenario of a company that develops/maintains software. It is intended to replicate such experiment with different participants, concerns and applications, in order to isolate the obtained results from these possible influences.

6 RELATED WORKS

Many works have been proposed for refactoring of OO software to AO ones and the refactorings are only applied at source-code level, from OO to AO (Silva et al., 2009); (Monteiro and Fernandes, 2006); (Hannemann et al., 2005); (Marin et al., 2005); (Iwamoto and Zhao, 2003). Moreover, it was noted a lack of related works related to model-based refactorings.

Borger et al. (2001) developed a plug-in for the CASE tool ArgoUML that support UML model-based refactorings. The refactoring of class, states and activities is possible, allowing the user to apply refactorings that are not simple to apply at source-code level. Van Gorp *et al*. (2003) proposed a UML profile to express pre and post-conditions of source-code refactorings using Object Constraint Language (OCL) constraints. The proposed profile allows that a CASE tool: i) verify pre and post-conditions for the composition of sequences of refactorings; and ii) use the OCL consulting mechanism to detect bad smells.

The differential of this work in relation to others is the proposal to construct an AO model considering OO class models annotated with stereotypes representing crosscutting concerns.

From the conducted case study was performed an evaluation of the obtained results with the support of AO metrics. It was realized that the use of proposed refactorings allows to obtain high quality AO models because: i) it provided a step by step guide to modularization of certain CCC; and ii) the proposed refactorings were elaborated considering good design AO practices. Therefore, the use of these refactorings can lead to build high quality AO models, because it prevents software engineers to choose inappropriate strategies for modularization of crosscutting concerns. The limitations of this study is considered: i) lack of a more quantitative evaluation of the computational support and the proposed refactorings; ii) the need for new metrics to improve the evaluation process of the refactorings; iii) lack of studies about the security semantics of legacy software after the application of refactorings; and iv) a little amount of refactorings for CCC.

7 FINAL CONSIDERATIONS AND FUTURE WORKS

The idea of using annotated OO class models to build AO models was adopted because they can bring the following benefits: i) it helps to visualizing possibilities for modularization without using AO; ii) provides higher level of abstraction by helping the software understanding; iii) the generated models serves as documentation for the AO software and legacy ones and are independent of programming language.

As future works we intend: i) to determine if, by means of a controlled experiment, the AO project model generated with the use of refactorings has better benefits than an AO project only obtained with code refactorings; ii) to develop new specific refactorings for other types of concerns such as security, exception handling, among others; iii) to create a module for detecting the impacts that can cause a refactoring on a particular software before being applied; and iv) to proposed strategies for guarantee the behavior-preservation of OO and AO models after using the refactorings;

ACKNOWLEDGEMENTS

The authors would like to thank CNPq for the financial support (Proc. No. 133140/2009-1 and 560241/2010-0).

REFERENCES

Analyse-it. Statistical analysis software for researchers in environmental & life sciences, engineering, manufacturing and education. Available in: http://www.analyse-it.com/. Accessed on: Jan/2013.

Boger, Marko and Sturm, Thorsten. "Tools-support for Model-Driven Software Engineering", *In Proceedings of Practical UML-Based Rigorous Development* Methods. (2001).

Costa, H. A. X. ; Parreira Júnior, P. A. ; Camargo, V. V. ; Penteado, R. A. D. . "Recovering Class Models Stereotyped With Crosscutting Concerns". In: Session Tool of *XVI Working Conference on Reverse Engineering (WCRE), Lille, França*. (2009).

Evermann, J. "A MetaLevel Specification and Profile for AspectJ in UML". Victoria University Wellington, Wellington, New Zealand. *AOSD - Aspect Oriented Software Development.* (2007).

Figueiredo E., Sant'Anna C., Garcia A., and Lucena C. Applying and Evaluating Concern-Sensitive Design Heuristics. *In Proceedings of the 23rd Brazilian Symposium on Software Engineering* Fortaleza, 2009.

Gamma, E., Helm, R., Johnsn, R., Vlisside, J. "Design Patterns: Elements of Reusable Object-Oriented Software". *Addison-Wesley*, (1995).

Hannemann, J., and Kiczales, G. Design Pattern Implementation in Java and AspectJ. In Proceedings of the *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA 2002) SIGPLAN Notices, Vol. 37, N°11, ACM 161–173 (2002).

Hannemann, J.; Murphy, G. C.; Kiczales, G. Role-Based Refactoring Of Crosscutting Concerns. *In Procedings of the Aspect-Oriented Software* Development, New York, Usa. P.135–146. (2005).

Hannemann, J. "Aspect-Oriented Refactoring: classification and challenges". In Procedings of the *International Workshop On Linking Aspect Technology And Evolution, Bonn*, Germany. p. 1-5 (2006).

Iwamoto, M.; Zhao, J. Refactoring Aspect-Oriented Programs. In Procedings of the *International Workshop On Aspect-Oriented Modeling* With UML, Boston, USA P. 1-7 (2003).

JSpider. The Open Source Web Robot. Available in*: j-spider.sourceforge.net/.* Accessed at: Jan. 2013.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J. "Aspect-Oriented Programming". *11th European Conference on Object-Oriented Programming*. v. 241 de LNCS, p. 220-242. Springer-Verlag (1997).

Marin, M.; Moonen, L.; Van Deursen, A. An Approach To Aspect Refactoring Based On Crosscutting Concern Types. *Sigsoft Software Engineering Notes*, V.30, N.4, P.1–5. (2005).

Monteiro, M. P.; Fernandes, J. M. L. Towards A Catalogue Of Refactorings And Code Smells For Aspectj. *Transactions On Aspect Oriented Software* Development (Taosd) - Lecture Notes In Computer Science, N.3880, P.214–258 (2006).

Montgomery, D. C., Design and Analysis of Experiments, 5 ed., Wiley, 2000.

Parreira Júnior, P. A. "Recovering Aspect-Oriented Class Models from Object-Oriented Systems by Model-based Refactorings". *Master Dissertation. UFSCar. São Carlos. Brazil*. 2011 (available only in portuguese).

Parreira Júnior, P. A.; Penteado, R. A. D.; Camargo, V. V.; Costa, H. A. X. "Mobre: Refactoring from Annotated OO Class Models to AO Class Models". In: CBSoft Tools Session, 2011, São Paulo/SP. *II Brazilian Conference on Software: Theory and Practice (CBSoft)*, 2011 (to be publish in portuguese).

Piveta, E. K. *et al*. "Avoiding Bad Smells In Aspect-Oriented Software". In: International Conference On Software Engineering And Knowledge Engineering - Seke, Boston, Usa, 2007. *Proceedings... Boston: Seke* 2007. P.81–87.

Silva, B. *et al*. Refactoring Of Crosscutting Concerns With Metaphor-Based Heuristics. Electronic Notes In *Theoretical Computer Science (Entcs), Vol*. 233, P. 105-125. (2009).

Soares, S., Laureano, E., and Borba, P. "Implementing distribution and persistence aspects with AspectJ". In *Procedings of the 17th ACM Conference OOPSLA'02*, 174 –190. (2002).

Van Gorp, P. Stenten, H. Mens, T. and Demeyer, S., "Towards Automating Source Consistent UML Refactorings," *Proc. Unified Modeling Language* Conf. (2003).

Anexo 11: Comprovante da publicação no *11th Workshop on Software Modularity (WMod) – Brazilian Conference on Software: theory and practice* (WMod 2014)

- Uma cópia do artigo é apresentado a seguir (a partir da próxima página).

Investigating Lightweight and Heavyweight KDM Extensions for Aspect-Oriented Modernization

Bruno M. Santos[1], Rafael S. Durelli[2], Raphael R. Honda[1], Valter V. Camargo[1]

[1]Departamento de Computação Universidade Federal de São Carlos - UFSCar
São Carlos – SP – Brazil.

[2]Instituto de Ciências Matemáticas e Computação Universidade de São Paulo - USP
São Carlos – SP – Brazil.

{bruno.santos, raphael.honda, valter}@dc.ufscar.br,
rsdurelli@icmc.usp.br

Abstract. *Architecture-Driven Modernization is the new generation of software reengineering. The main idea is to modernize legacy systems using a set of standard metamodels. The first step is to obtain an instance of an ISO metamodel, called KDM, through reverse engineering. Then, refactorings and optimizations can be performed over this model transforming it into a target KDM. Further, a modern source code can be generated from the target KDM. Although KDM intentionally does not provide support for Aspect-Oriented Programming (AOP), it is possible to extend it by either creating/changing its existing metaclasses (heavyweight) or using its extension mechanism, creating stereotypes (lightweight). There are some few works in the literature that present KDM extensions. However, all of them present just one extension without providing a discussion or a comparison about the suitability of them. This is important because such comparison can support modernizers in choosing the best alternatives to meet their needs. Therefore, in this paper we present two KDM extensions for AOP in order to enable an AO modernization. We also present a preliminary comparison between them trying to characterize their vantages and disadvantages and we conclude that to perform an AO modernization the heavyweight one is the most indicated.*

1. Introduction

Architecture Driven Modernization (ADM) was proposed by Object Management Group (OMG) in 2003. ADM uses the concepts of Software Reengineering (SR) and the principles of Model-Driven Architecture (MDA) in order to perform modernizations in a respective system based on models [OMG 2014]. ADM introduces several standards metamodel mainly to support the whole process of model-driven modernization. Its main metamodel is Knowledge Discovery Metamodel (KDM), which is language and platform independent and allows representing various artifacts of a system at different abstraction levels, such as configuration files, database, source-code, etc [ADM 2014].

In a parallel research line, researchers have been using Aspect-Oriented Programming (AOP) as an alternative for the modularization of systems [Kiczales, *et al.* 1997]. They argue that modularize the system is an important kind of modernization because it allows, for instance, the separation of crosscutting concerns in the legacy

source code [Laddad 2003]. In this context, our main goal is to enable the Aspect-Oriented (AO) modernization to be carried out according to the ADM standards. To fulfill this goal an extension of AOP concepts has to be performed in KDM metamodel.

The latest version of KDM is 1.3 [KDM 2012], in this version it is not possible to represent the concepts of AOP. Therefore, it is necessary to extend the metamodel to allow the representation of AOP. Similarly, as UML, nowadays there are two ways to extend the KDM metamodel, either by (*i*) performing a lightweight (LW) extension or (*ii*) performing a heavyweight (HW) extension. Studies published in literature [Mirshams 2011] [Baresi and Mirazi 2011] have performed extensions in KDM, but, most of them do not report the advantages and disadvantages of each extension mechanism.

An important point to perform metamodel extensions is to use a representation of the new concepts that should be represented. In the lightweight extension this representation is called as profile. To support our extensions, we carried out a search in the literature to find a profile that was as comprehensive as possible and which would be able to represent not only the concepts of AspectJ language, but also the concepts of other languages that implement AOP.

One of the most important steps in a modernization process is the refactoring of the models. In this sense, modernization engineers usually need to write source code to refactoring legacy KDM into target KDM. When the modernization is performed in a specific domain, such as AOP, the refactoring code involves to instantiate an extended KDM, which can be a LW or HW extension. Depending on the extension type (light or heavy) the productivity of the modernization engineers can be different because these two types of extensions encompass many particularities [Magableh, Shukur and Ali 2012]. In this paper, we perform these two extensions both named KDM-AO. In other words we have created both LW and HW. Moreover, we present a preliminary comparison between the two extensions we have developed. This comparison aims at providing a base for modernization engineers in choosing the most suitable one according to their needs. The comparison was performed based on some criteria we have chosen and applied to a case study. Our results show that the heavyweight extension is the most appropriate mechanism to deal with domain-specific concepts with a large number of concepts because it helps the productivity and ensure the quality of the models. Also a fully list that contains the advantages and disadvantages of each extension was created. We claim that by means of it the engineer can pinpoint which extension is better for she/he context. In order to assess our KDM-AO we carried out a Crosscutting Framework-based modernization process in a management system of a CD Shop [Camargo and Masiero 2005] using the two extensions. The evaluation showed that both extensions are able to represent all the details inherent in this type of framework, as well as all AO concepts. In addition, the results show that by using the KDM extensions is possible to modernize a legacy system to AOP.

This paper is structured as follows: In Section II, the background about ADM/KDM is shown. In Section III the Metamodel Extensions are described. A case study is shown in Section IV. In Section V we present a preliminary qualitative comparison. The related works are shown in Sections VI. Finally, in Section VII, the discussions and conclusions are presented.

2. Background

In 2003, OMG initiated efforts to standardize the process of modernization of legacy systems using models by means of the ADM Task Force (ADMTF) [ADM 2014]. The aim of the ADM is the revitalization of existing applications by adding or improving functionalities, using the OMG modeling standards and also considering MDA principles. In order to support the modernization process, in 2006 the KDM metamodel was created. KDM is language and platform-independent.

KDM contains twelve packages and it is structured in a hierarchy of four layers: (i) Infrastructure Layer, (ii) Program Elements Layer, (iii) Runtime Resource Layer and (iv) Abstractions Layer [KDM 2012]. These layers are created automatically, semi-automatically or manually through the application of various techniques of extraction of knowledge, analysis and transformations [5]. Each layer is organized into packages that define a set of metamodel, whose purpose is to represent a specific and independent interest of knowledge related to legacy systems [KDM 2012]. In the context of this paper, we are going to focus on the Infrastructure Layer and Program Elements Layer, especially on the *KDM* and *Code* packages. The KDM package has a set of metaclasses that allows the creation of stereotypes in instantiation time. This means that KDM package provides the metaclasses to perform a LW extension, so it is not needed the creation of new metaclasses in KDM metamodel.

The *Code* package contains a set of metaclasses to represent program elements in implementation level. This package owns a set of metaclasses that represent the common named elements in the source code supported by different programming languages such as data types, classes, procedures, methods and interfaces [KDM 2012].

The KDM metamodel provides the base metaclasses that allows performing both light and heavyweight extensions, the decision of which kind of extension will depend on the modernization context. When the LW extension is performed, the original metamodel is not changed, so this guarantees the tools compatibility. Otherwise, the HW extension changes the original metamodel adding new metaclasses, so new tools have to be made to support this kind of extension. These points are valid for LW and HW extension when performed in KDM or Unified Modeling Language (UML) metamodels, even the goal of these two metamodels being different [Pérez-Castillo, Guzmán and Piattini 2011].

Another important point is that KDM is not a metamodel intended to serve as base for diagrams, like UML [Chavez and Lucena 2002] [Fuentes and Sanchez 2006]. While humans usually create UML instances, KDM instances are created by parsers and processed by tools. Therefore, profiles make more sense in the context of UML than in the context of KDM. Anyway, this paper aims to perform both extensions in KDM metamodel and list the advantages and disadvantages, providing guidance on how to perform them.

3. Metamodel Extensions

As in UML, it is also possible to define either LW or HW extension in KDM by means of extension mechanism. HW extensions are based on a modified KDM metamodel, including new metaclasses or changing the existing ones. In other hand, LW extensions (profiles) are based on set of *Stereotypes* and *TagDefinitions*, which are basically

"notes" over the model. Profiles are able to impose restrictions on existing metaclasses, but they respect the metamodel, without modifying the original semantics of the elements. One of major benefits of profiles is that they can be handled in easier way by existing tools.

In general, the drawback of HW extensions is that existing tools get no longer compatible with the new metamodel. However, the only way to guarantee model correctness in model level is using HW extensions. This happens because it is possible to relate metamodel elements by their types and not just by their names, as it usually happens in LW extensions. Using LW extensions, the correctness of the model must be guaranteed by tools.

The main decision before performing KDM extension was to choose an UML profile which was broad enough to represent all the AO concepts. In this sense, we conducted a literature review to identify Aspect-Oriented metamodels and UML profiles that could be considered good candidates. We had analyzed several proposals, but the Evermann's profile was considered the most suitable one because it incorporates the level of details that we are interested in **Erro! Fonte de referência não encontrada.**. Both extensions presented in this paper have been created based on Evermann's profile. In Figure 1 is presented both extensions. We would like to emphasize that a more detailed version of the HW extension can be found in **Erro! Fonte de referência não encontrada.**. Each class/element has four words (four lines) in its first compartment. The first word (in bold) represents the name of the metaclass we have created in our heavyweight extension, for example, *AspectUnit* is a term that exists in the HW extension. The second word, inside the brackets, is the KDM metaclass we have chosen to make the current element extends from. For example, the new metaclass *AspectUnit* that belongs to our heavyweight extension extends the *ClassUnit* KDM metaclass.

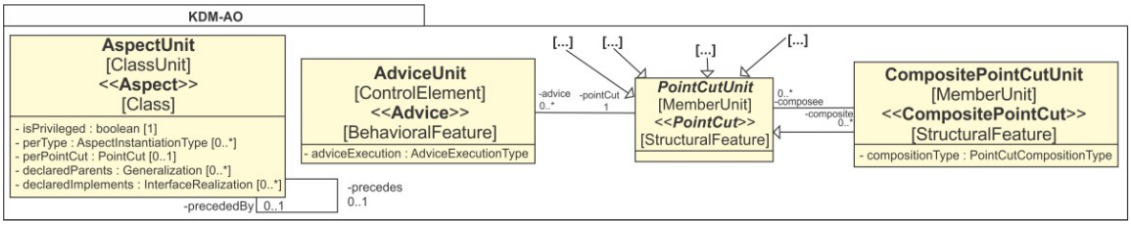


Figure 1. KDM-AO and Evermann's Profile (Adapted)

In our lightweight extension, we do not have metaclasses, but stereotypes. Thus, below the first two elements already described there is a <<stereotype>>. Therefore, all the stereotypes shown in this figure exist in our lightweight extension. Stereotypes are applied over existing elements, so the name of the element in which the stereotype can be applied on is the metaclass shown in the second line. For example, the stereotype <<aspect>> can only be applied over *ClassUnit* instances. The last line contains the name of the UML metaclass used in the original version of Everman's profile.

3.1. Lightweight extension

KDM metamodel provides a set of metaclasses in *KDM* package to allow the creation of stereotype families; these metaclasses are shown in Figure 2. The *ExtensionFamily* element acts as a container for a set of related stereotypes and their corresponding *TagDefinitions*. The *Stereotype* concept provides a way of branding model elements so

that they behave as if they were the instances of new virtual metamodel constructs. Thus, a stereotype may be used to indicate a difference in meaning or usage between two model elements with identical structure.

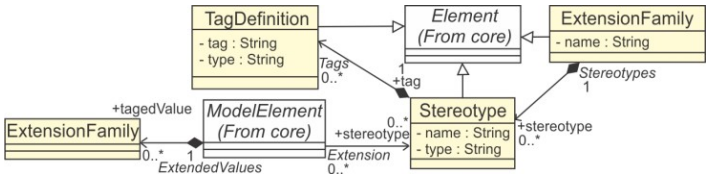


Figure 2. Extensions Class Diagram (Adapted)

Each *Stereotype* owns the optional set of *TagDefinitions* and each one provides the name of the tag and the name of the KDM type of the corresponding value. The *ExtensionFamily*, *Stereotype* and *TagDefinition* elements are the main elements to represent lightweight extensions in KDM. Herein, we are representing the LW extension programmatically. Therefore, to perform this extension the user has to create a new Java class and create instances of the *ExtensionFamily*, *Stereotype* and *TagDefinition* elements with the values that represent the new elements. In Figure 3 is represented a snippet of the programmatic lightweight extension in KDM. Line 1 depicts the creation of *ExtensionFamily's* element instance and Line 2 shows an instance of *Stereotype* element. Line 3 adds the *Stereotype* created on Line 2 into the *ExtensionFamily* created on Line 1. Lines 4 and 5 the values of *Stereotype's Name* and *Type* are set. In the Line 6 a *TagDefnition* is created and in the Line 7 this tag is attached to the *Stereotype*. Finally, in the Lines 8 and 9 the *TagDefinition's* properties *Tag* and *Type* are defined.

```
1..ExtensionFamily AspectConcepts =
KdmFactory.eINSTANCE.createExtensionFamily();
2..Stereotype AspectUnit = KdmFactory.eINSTANCE.createStereotype();
3..AspectConcepts.getStereotype().add(AspectUnit);
4..AspectUnit.setName("AspectUnit");
5..AspectUnit.setType("ClasUnit");
6..TagDefinition IsPrivileged =  KdmFactory.eINSTANCE.createTagDefinition();
7..AspectUnit.getTag().add(IsPrivileged);
8..IsPrivileged.setTag("isPrivileged");
9..IsPrivileged.setType("boolean"); [...]
```

Figure 3. Snippet of LW extension programmatically

Once all the elements in Figure 1 have been added programmatically it is possible reuse it by means of a class with all the *Stereotypes* and *TagDefinitions* programmed. The usage of light and heavyweight extension is shown in Section 4. In Section 4 we present a modernization case study that uses both extensions to make the representation of the main AOP concepts.

3.2. Heavyweight extension

In Evermann's profile, the *CrossCuttingConcern* element extends the Package UML metaclass and aims to represent the existence of a Crosscutting Concern like Persistence, Security and Concurrency. In our KDM-AO this element extends the Package metaclass. This KDM metaclass represents a package in which is possible to deposit Aspects, Classes and other elements.

AspectUnit is our element for representing aspects, which extends the *ClassUnit* KDM metaclass. We decided to extend this metaclass because aspects have all the characteristics that classes have, but pointcuts, advices and intertype declarations. The element for representing advices in our heavyweight extension is *AdviceUnit*, which

extends the *ControlElement* metaclass. Knowing that advice is an element that specifies behavior, we could consider it like a method. However, advices do not have neither access specifiers (public, private, protected) nor types (constructor, destructor, etc). Because of that we have decided do not make it extends *MethodUnit*, instead, we make it extends from its parent metaclass *ControlElement*.

PointCutUnit is our element for representing pointcuts. According to Evermann's profile, pointcut is a structural element and extends the UML metaclass *StructuralFeature*. KDM has also a class for representing structural characteristics called *DataElement*, which is an abstract metaclass. Its descendents are *StorableUnit*, *MemberUnit*, and *ItemUnit*. As *StorableUnit* and *ItemUnit* cannot be *abstract*, *MemberUnit* was chosen to be the supertype of *PointCutUnit*. Moreover, another reason for extending *MemberUnit* was that pointcuts can crosscut other classes and *MemberUnit* is the KDM metaclass used to denote references to other classes. *StaticCrossCuttingFeature* is our element for representing intertype declarations. In our KDM-AO we have decided to extend two KDM metaclasses: *StorableUnit* e *MethodUnit*. In this way, *StaticCrossCuttingFeature* is able to represent structural and behavioral characteristics.

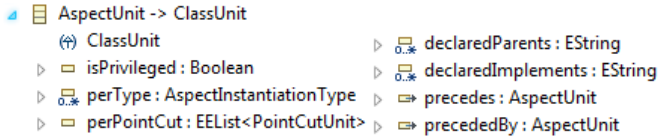


Figure 4. A snippet of the heavyweight extension

In Figure 4 is possible to see the *AspectUnit* element expanded with all its attributes and relationships. As we can see, *ClassUnit* is the superclass of *AspectUnit* and all the attributes and relationship added are present in our KDM-AO.

4. Case Study Overview

In this section is presented a case study in which were applied both extensions in a modernization process based on Crosscutting Frameworks (CFs) **Erro! Fonte de referência não encontrada.**. CFs are AO Frameworks that encapsulate in a generic way just one crosscutting concern, like Persistence, and Security **Erro! Fonte de referência não encontrada.**. In this case study, we modernized a management system of a CD shop. The modernization goal was to modularize the persistence concern with aspects. As our group has some experience with Crosscutting Frameworks, the idea was to use a Persistence CF previously developed in this process. To represent this application using our KDM extensions we have to create 4 Aspects and 2 Pointcuts in both extensions.

Because of space limitations, in this paper we show the usage of our extensions to represent just two aspects, one is from the Persistence CF and the other is from the instantiation model. The name of the first chosen aspect is *ConnectionComposition*. Its purpose is to provide a base behavior for opening and closing database connections. During instantiation, application engineers need to provide concrete implementations for the abstract pointcuts *openConnection*() and *closeConnection*(). This aspect has in its body an attribute, two abstract pointcuts, a concrete and one abstract operation and two advices. In Figure 5 there are parts A and B in which part A represents an instance of LW extension and part B represents an instance of HW extension. Each line contains the

element type and then its value for both extensions. The following paragraphs describe only the AOP elements, the elements from Object-Orientation are out of our scope.

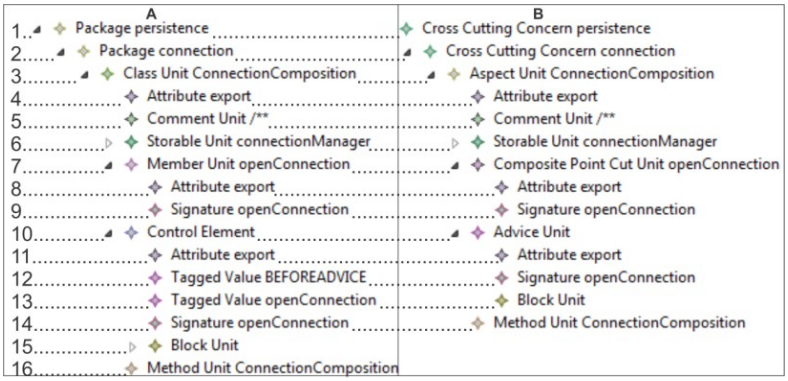


Figure 5. A snippet of the aspect *ConnectionComposition*.aj in KDM-AO LW and HW

In Line 1 we can visualize a Package (Part A) and *CrossCuttingConcern* (Part B) whose value is persistence, i.e., in LW this is an instance of the package metaclass and in HW an instance of *CrossCuttingConcern* metaclass. The main difference between them is that besides they are from different metaclasses, the LW one is stereotyped with *CrossCuttingConcern*. Line 3 displays the name of the aspect that is being modeled here; in LW the stereotype *AspectUnit* is applied in an instance of *ClassUnit* and in the HW there is no stereotype, instead of this we have an instance of the metaclass *AspectUnit*. To declare a Pointcut in LW you have to apply a stereotype in the *MemberUnit* element, and this is shown in Line 7 part A. *CompositePointCutUnit* (Line 7 part B) is used to represent concrete or abstract pointcuts of an aspect.

The element in KDM that can represent an Advice in LW is ControlElement when the stereotype *AdviceUnit* is applied (Line 10). In HW *AdviceUnit* (Line 10) represents the advice that was declared in the aspect. It is essential to fill the Advice Execution property because this property declares what kind of advice that element represents (After, Before or Around). This kind of declaration is very different in both extensions, for example, while in HW this information is set in the properties of the *AdviceUnit* element, in LW you have to create a *TaggedValue* element and apply a TagDefinition in it. In Lines 12 and 13 (part A) we have two *TaggedValues* elements, the first one store the *TagDefinition* adviceExecution and the second one stores the name of the poincut that this *AdviceUnit* (*ControlElement*) belongs.

The second Aspect that has been chosen is from the instantiation model and its name is *myConnectionCompositionRules*. This aspect stores the name of the method in the base source code that will be affected by the Persistence CF and it is composed by two Pointcuts and one method that store the name of the class that will implement the persistence concern. In Figure 6 is depicted a snippet of the aspect *myConnectionCompositionRules* in heavyweight extension. This figure also shows an *ExecutionPointCutUnit*. The *ExecutionPointCutUnit* intercepts the execution of the method main in the class *FindSomeCDs* and is bounded to the *CompositePointCut openConnection*. Line 1 represents the source code to create an instance of *ExecutionPointCutUnit*, according to the heavyweight extension. Lines 2 and 3 represent the declaration of some properties of the *ExecutionPointCutUnit* element, in Line 2 the *operation* is set and in Line 3 is set the *CompositePointCutUnit* that the Execution belongs.

```
1..ExecutionPointCutUnit myExecution =
CodeFactory.eINSTANCE.createExecutionPointCutUnit();
2..myExecution.getOperation().add(FindSomeCDs.main);
3..myExecution.getComposite().add(openConnection); [...]
```

Figure 6. Snippet of *myConnectionCompositionRules* in heavyweight

Figure 7 shows a LW instance of KDM that represents the same source code in Figure 6 but now we are using only metaclasses from the original KDM.

```
1..MemberUnit myExecution = CodeFactory.eINSTANCE.createMemberUnit();
2..myExecution.getStereotype().add(Profile.executionPointCutUnit);
3..TaggedValue operation = KdmFactory.eINSTANCE.createTaggedValue();
4..operation.setTag(Profile.operation);
5..operation.setValue("FindSomeCDs.main");
6..TaggedValue composite = KdmFactory.eINSTANCE.createTaggedValue();
7..composite.setTag(Profiles.composite);
8..composite.setValue("openConnection");
9..myExecution.getTaggedValue().add(operation);
10..myExecution.getTaggedValue().add(composite); [...]
```

Figure 7. Snippet of *myConnectionCompositionRules* in lightweight

To create an *ExecutionPointCutUnit* we have to apply the *execution* stereotype in a *MemberUnit* element, this is represented in Lines 1 and 2. There are two main properties in an *ExecutionPointCutUnit* that are the *operation* and the *composite*, but these properties do not exist in *MemberUnit* element. To declare additional property in an element it is necessary to create instances of *TaggedValue* element and apply a *TagDefinition*. After this it is time to set the value of the property and this value is always a *String Type*. The *operation* property is declared in Lines 3, 4 and 5 and the *composite* property is declared in Lines 6, 7 and 8. Once we have created a *TaggedValue* and set its properties we have to bind the *TaggedValue* to its respective element and the source code that implements this is declared in Lines 9 and 10.

5. A Preliminary Qualitative Comparison

In order to compare the two extensions mechanisms we defined some evaluation criteria shown in Table I. These criteria were created to compare the extensions, trying to characterize if one is better than another one.

The first criterion is intuitiveness of creating instances. This criterion tries to compare the intuitiveness of instantiating the extensions. In this sense, intuitiveness is related to how straightforward is to use the extension elements. For example, to create an aspect in the heavyweight extension it is only necessary to create an instance of the *AspectUnit* class. However, in the lightweight version, it is necessary to create an instance of the *ClassUnit* class and apply the stereotype *AspectUnit* on it. In this case it is much more intuitive to use the HW than the LW one. Notice that this criterion impacts both the creation of new instances of the extensions and also the maintainability of those instances. As HW is more intuitive, it is more straightforward of creating and also changing (adding, removing) existing extension instances. This can be seen if we compare Figure 6 and Figure 7.

The second criterion is the correctness of creating instances. This criterion is related to the likelihood of creating erroneous instances. The heavyweight is better than the light because the values passed in the lightweight extension are only String types, so if the programmer set wrong information this will not be validated. Otherwise, heavyweight extensions are strongly typed, so if a property receives a Boolean type, the

only values possible to set are "*true*" or "*false*", anything else will be considered as a type error and the model will not be able to compile.

The third criterion is tooling impact, which is regarding how easy is to reuse the extension in different tools. The lightweight extension is easier to reuse because tools may be previously designed for considering profiles. In this case, the task would only to import the profile into the tool and use it. However, heavyweight extensions are usually proprietary solutions. Most of the times they are not designed for being reused in other contexts. Although it is not impossible to make a heavy extension reusable, it is harder when compared to the lightweight version. For example, if a tool has been designed for mining KDM instances, it may not work properly if a modified KDM is provided. The impact can be minimized when the heavyweight extension has just added new elements in the standard KDM, rather than modified existing ones.

The fourth criterion is the productivity in creating the extensions. This criterion tries to characterize the effort in creating LW and HW extensions. Clearly, the creation of LW ones is much easier than HW. As creating LW you do not need to worry with types because basically every value is set with String Type. The creation of heavyweight ones is much more time consuming because every additional property have to be set in a *TaggedValue* element. One important point to highlight here is that the productivity in creating extensions is not as important as the maintainability of it. This happens because, in general, the extension will be created once and used several times. Therefore, if the creation takes two months or two hours is not that important. Nevertheless, although the creation of heavy extensions is harder, its usage intuitiveness is better, as result, fewer errors are expected in its instances.

The fifth criterion is about the productivity for maintaining the extension. This criterion tries to characterize the effort in adding, removing and changing existing instances elements. As for maintaining the extension, LW showed itself quicker and easier than HW. This came about once that to perform any change it is necessary just to apply the updates and the LW extension is be already to use. Otherwise the HW extension after applying all changes it is necessary to recompile and update the plug-in.

The sixth criterion is the complexity of the XMI. The idea here is to identify which XMI is more complex. Complex XMI is harder to process and will require much more effort in writing the algorithm. Again, HW extension presents a more clear and intuitive XMI. In this extension the algorithm just need to search for the correct term, such as *AspectUnit*. However, in the LW version, the identification of elements required the identification of the metaclass that represents the element, the identification of the *stereotype* and its *tags*. In Table 1 is possible to see a comparison between LW and HW. There is a ↑ when the extension is better and a ↓ when the extension is worse.

Table 1. Comparison between the extension mechanisms

Criteria	Lightweight	Heavyweight
1. Intuitiveness of Creating Instances	↓	↑
2. Instances Correctness	↓	↑
3.Tooling impact	↑	↓
4.Productivity for Creating the Extensions	↑	↓
5. Productivity for Maintaining the Extension	↓	↑
6. Complexity of the XMI	↓	↑

In terms of reusability of the extension, the LW version is a better alternative because tools can be more easily prepared for working with profiles. For example, consider the existence of a tool that applies refactoring over the KDM. This tool can be

easily extended to consider stereotypes. Although it is not impossible to make a HW extension to be reused, the impact in tools will required more effort.

If you are intending to make your extension available to be reused, than the LW one seems to be a better alternative. This happens because you can make available just the *Stereotypes* and *TaggedValues*. If existing tools were already designed for accepting KDM profiles, it is straightforward to reuse it.

If you are intending to use the extension into an organization or proprietary tool, the HW can be a better solution in terms of productivity and quality of the instances. It's important because the quality of the modernized instance will have a direct impact in the source code generated. Based on our experience in performing LW and HW extensions we consider the HW extension the best mechanism to support AOP modernizations because this kind of modernization involves many concepts and the HW one ensures a higher productivity and quality of the models. The set of criteria presented aims to support software engineers in choosing a type of extensions. The final depends on several other details, context and scenarios in which the extension will be used.

6. Related Works

We found some works in the literature that presents KDM extensions, but we are considering only the two most relevant. The first one is proposed by Mirshams [Mirshams 2011] and another one is proposed by Baresi and Miraz (2011). However, as they have developed just one extension, either light or heavy, they do not provide any comparison in order to characterize which one is better.

Mirshams (2011) created a heavyweight KDM extension for AOP. There are two main differences between our works. The first difference is the level of abstraction of our extensions. The aspect model used by Mirshams contains much less elements than Evermann´s profile. That means our extension is able to represent both a high level (using the most generic metaclasses) and a low level (using most specific metaclasses) view of the system. The second difference is that her work is limited to dynamic crosscutting as there are no elements for representing intertype declarations. Baresi and Miraz (2011) describe another KDM extension. They proposed a HW KDM extension to support Component-Oriented MOdernization (COMO) that is a metamodel with traditional concepts of software architecture, allowing attaching software components in KDM. This paper performs a KDM extension but the main difference is that they don't use AOP concepts and its goal is just to allow KDM to perform COMO modernizations.

There is another related work that compares LW and HW extension but in UML metamodel. Magableh, Shukur and Ali (2012) gather fourteen papers that executed LW or HW and apply some comparison criteria. As we did here, they list some criteria to evaluate the papers. The criteria used by them were used to inspire us to create ours, but we didn't use these criteria because some of them were too much specific for UML or considered if the tool created to support the extension were complete enough. Because of this we had to create a new set of criteria to support our discussions.

7. Discussions and Conclusions

The main points in this paper are on how to perform these extensions and list the advantages and disadvantages of using them. In section III we showed some guidelines

that can help performing these extensions and in section V we present a list with some criteria that can help deciding which extension mechanism a software engineer should use, of course, depending on his necessities and context.

By means of our case study, it is fairly evident that our extensions can represent all elements of AOP. However, as we have not carried out a complete case study to gauge how reliable our extension is to represent aspects concepts in other programming languages, such as AspectC++, we argue that this is a limitation of our extensions. Nevertheless, to mitigate this limitation the elements of AspectC++ and AspectS were analyzed. Consequently, we conclude that there are enough elements in our extension that can be used to represent source code in both AOP languages. The main contribution of the case study is to prove that the AOP concepts can be represented in LW and HW, so we conclude that even if we perform this modernization in another system or use another CF with a different concern we would reach the same results. To prove this we are planning to perform AOP modernizations in other systems with different CFs.

When conducting our case study using CFs, we have noticed that our extension would be more useful and more expressive if it had also metaelements for representing CF characteristics, like hot spots, frozen spots and other framework characteristics. We intend to perform these modifications in a future work [Camargo and Masiero 2008]. As we are proposing a preliminary qualitative comparison our set of criteria may not be enough to compare the extensions. To solve this gap we intend to search in the literature more criteria to ensure a fairer comparison. As other future works, we aim to conduct others case studies using AspectC++ and AspectS in order to test the KDM-AO extensions, to evaluate the issue of platform independence. Another future work that we are already performing is to evaluate these two extension mechanism by means of a controlled experiment in the context of Computer Science graduation students at UFSCar. It consists in dividing the class in two groups and giving them some evolution and maintenance activities to be performed with LW and HW extension. The main point is to evaluate which approach allows the creation and maintenance of model with the less time and lowest number of errors. We intend to publish these results in a paper after we finish all the programmed steps.

By conducting this research we have noticed that the power of model-driven modernization is greatly influenced by the capacity of representing specific concepts in a proper and suitable way. We also noticed that KDM is a powerful metamodel that can be adapted in almost every context, so it will depend on the software engineer to create the solution the best serves to his proposals.

Acknowledgements

Bruno M. Santos and Raphael R. Honda would like to thank CNPq for sponsoring our research. Rafael S. Durelli and Valter V. de Camargo would like to thank the financial support provided by FAPESP, processes numbers 2012/05168-4 and 2014/14080-9, respectively.

References

Architecture-Driven Modernization. (2014) Document omg/ http://adm.omg.org/.

Baresi, L. and Miraz, M. (2011) “A Component-oriented Metamodel for the Modernization of Software Applications”, 16th IEEE International Conference on Engineering of Complex Computer Systems.

Camargo, V. V. and Masiero, P. C. (2005) “Frameworks Orientados a Aspectos”, XIX Simpósio Brasileiro de Engenharia de Software, Uberlândia. pp. 200-216.

Camargo, V. V. and Masiero, P. C. (2008) “An Approach to Design Crosscutting Framework Families”, ACP4IS 08, Brussels, Belgium.

Chavez, C. and Lucena, C. (2002) “A metamodel for aspect-oriented modeling”, In Proceedings of the AOM with UML workshop at AOSD.

Evermann, J. (2007) “An overview and an empirical evaluation of UML: an UML profile for aspect-oriented frameworks”, Workshop AOM ’07, Vancouver, British Columbia, Canada.

Fuentes, L. and Sanchez, P. (2006) “Elaborating UML 2.0 profiles for AO design”, In Proceedings of the AOM workshop at AOSD.

Kiczales, G. *et al.* (1997) “Aspect Oriented Programming”, In: Proceedings of 11 ECOOP. pp. 220-242.

Knowledge Discovery Meta-Model. KDM Guide, April 2012. Document omg/formal/2012-05-08.

Laddad, R. (2003), AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications, Greenwich (74° w. long.). pp.75-77.

Magableh, A., Shukur, Z. and Ali, N. (2012) “Heavy-Weight and Light-weight UML Modelling Extensions of Aspect-Orientation in the Early Stage of Software Development”, Journal of Applied Sciences.

Mirshams, P. S. (2011) “Extending the Knowledge Discovery Metamodel to Support Aspect-Oriented Programming”, 79 f. Dissertation (Master of Applied Science in Software Engineering) – University of Montreal, Quebec, Canada, unpublished.

Object Management Group. (2014) OMG Specifications, April 2014. Documents omg/ http://www.omg.org/spec/.

Pérez-Castillo, R., Guzmán, I. G. and Piattini, M. (2011) “Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems”, Computer Standards & Interfaces 33, Elsevier B.V. pp. 519–532.

Santos, B. M., Honda, R. R., Durelli, R. S. and Camargo, V. V. (2014) “KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel”, In 28th Brazilian Symposium on Software Engineering (SBES). Publishing Press.

Anexo 12: Carta do orientador estrangeiro atestando a realização das atividades

- A carta é apresentado a seguir (a partir da próxima página).

INRIA - Lille Nord Europe
USTL - LIFL - CNRS UMR 8022

Rafael Serapilha Durelli
Computer Systems Dept
University of São Paulo in São Carlos

To whom it may concern

I, Dr. Nicolas Anquetil, associate professor at University Lille-1, France, declare that Rafael Serapilha Durelli realized all activities planned for the internship as a visiting Ph.D. student in Computer Science at INRIA Lille – Nord Europe during his stay in our group (RMod)

Yours sincerely

Dr. Nicolas Anquetil
Associate professor, INRIA Lille – Nord Europe

Dr. Nicolas Anquetil
Associate professor, INRIA Lille – Nord Europe
nicolas.anquetil@inria.fr

INRIA
40, avenue Halley, Parc scientifique de la haute borne
FR-59650 Villeneuve d'Ascq, France

Tel. +33 (0)3 59 35 87 47
Fax +33 (0)3 59 57 78 50
Secr. +33 (0)3 59 35 86 17