2
Trabalhos Relacionados

Este capítulo apresenta um estudo sobre algumas soluções, abordagens e técnicas existentes na literatura para a geração de um diagrama comportamental para linguagens estruturadas.

Entende-se como linguagem estruturada a linguagem que permite a quebra do programa em blocos. Cada pedaço pode ser compilado separadamente e todos eles agem isolados um dos outros. Eles trabalham em conjunto cada vez que um bloco necessita de um código presente em outro bloco. Exemplos de linguagens estruturadas são C, C++ e Pascal.

O problema foi atacado a partir de três óticas diferentes: a primeira buscou projetos que chegassem perto de uma solução completa. A segunda focou em encontrar abordagens para o problema da engenharia reversa de um código, sem necessariamente olhar para a camada de exibição. E uma terceira abordagem priorizou soluções para o problema da exibição dos dados obtidos através da engenharia reversa.

São descritos neste documento os trabalhos encontrados mais relevantes. Para cada um existe um breve resumo da abordagem utilizada e uma crítica sobre a solução proposta. Além disso, foram citados alguns projetos e tecnologias que não possuem uma ligação direta com o problema apresentado, porém servem como base para a solução.

2.1.
Rational Data Architecture

A motivação inicial do trabalho foi gerar um diagrama de comportamento de uma *procedure* escrita na linguagem SQL. Pensando nisso, foi feito um estudo para verificar em que nível estão as ferramentas para manipulação e desenvolvimento de banco de dados. Não foi encontrada nenhuma capaz de produzir o diagrama esperado. Procurando em fóruns de internet e em referências encontradas no site de busca *Google*, o Rational Data Architect (IBM, 2008) estava sempre muito bem cotado, citado constantemente como uma das melhores ferramentas existentes na atualidade. Como era necessária uma

análise profunda de uma ferramenta com esta finalidade, a escolha foi natural. O estudo foi realizado em uma versão temporária fornecida pela IBM.

O Rational Data Architect, RDA, é um software pertencente à plataforma de desenvolvimento da IBM. Ele é parte do conjunto de produtos para gerenciamento de informações da empresa. O RDA é compatível com vários sistemas de gerenciamento de banco de dados (SGBD), como DB2, Sybase, Oracle, SQL Server, entre outros. Ele se integra com os outros produtos da marca Rational, como o Rational RequisitePro, para controle de requisitos, e o ClearCase, para controle de versão. O propósito principal do produto é prover um ambiente de desenvolvimento para tarefas relacionadas a banco de dados, como criação de tabelas, *procedures,* entre outros. A sua principal característica é fornecer um meio de transformar um modelo UML (*Unified Modeling Language*) em um modelo lógico e vice-versa. Além disto, representa uma ferramenta importante no trabalho de documentação e engenharia reversa de banco de dados.

Como dito anteriormente, o RDA é compatível com outros produtos da Rational. Como exemplo de ganho derivado disso, pode-se citar a tarefa de gerenciamento de requisitos e gerência de configuração. Toda funcionalidade mapeada no RequisitePro poderá ser associada a um elemento do modelo lógico, que por sua vez está relacionado por um mapeamento de transformação a um diagrama UML. No final da ponta, este ainda poderá estar associado com o arquivo que representa determinada classe na linguagem utilizada. Além disso, todos estes modelos e diagramas estão versionados no ClearCase. O cenário anterior é apenas uma demonstração da capacidade que este ambiente Rational oferece.

Partindo do tradicional problema da existência de sistemas legados ou sem documentação, ele provê um gerador de modelos físicos a partir de uma simples conexão JDBC com um banco de dados. Além deste tradicional modelo, outras três variações são apresentadas: modelo de domínio, modelo de glossário e modelo lógico. Basicamente, em cada um destes diferentes modelos, existe uma alteração nos elementos gráficos apresentados. O modelo lógico, por exemplo, se aproxima muito de um MER.

Vale lembrar que a qualquer momento pode-se gerar um *script* para alterar ou criar o banco de dados a partir de um modelo, ou fazê-lo diretamente usando a conexão da própria ferramenta.

Ainda no contexto de software legado, uma funcionalidade muito útil é a ajuda na migração da estrutura e dos dados de um banco para a sua versão

mais nova. É possível fazer uma função de mapeamento de uma versão para a outra, e automaticamente a conversão é feita. Além disso, é possível comparar dois bancos diferentes, gerando um relatório com todos os campos que se diferem.

Caso o RBA seja usado para uma conexão com um banco DB2, é disponibilizada uma ferramenta para desenvolvimento de p*rocedures*. O editor de texto colore o corpo do código de acordo com a sintaxe da linguagem utilizada no banco DB2. No modo de *debug* estão presentes todas as tradicionais funcionalidades de um ambiente de depuração, como inspeção de conteúdo de variáveis e opção de reiniciar a execução de um ponto específico. Não existe nenhum tipo de apoio, neste sentido, para outros gerenciadores de banco de dados, como *Oracle*.

Por último, vale ressaltar que existem dezenas de outras funcionalidades menos importantes, como a importação e exportação de modelos para o formato ERwin, rastreamento de dependências de tabelas, geração de *scripts* ou atualizações de bancos de uma versão específica para outra, entre outras.

O RDA se mostrou uma ótima ferramenta para documentação. A sua capacidade de geração de diferentes tipos de modelos mostrou-se eficaz. A atenção e importância dadas às funcionalidades de engenharia reversa motivaram a continuação da dissertação, pois ressaltaram a importância de um aplicativo que auxilie na manutenção e especificação de bancos legados. Está claro no uso desta ferramenta que o problema apresentado nesta dissertação realmente existe, pois a interface chama a atenção a todo o momento para opções de engenharia reversa. Apesar disso, ela reforçou a idéia que a questão motivadora deste documento não está disponível nas ferramentas conhecidas do mercado. O RDA é um sistema complexo, caro e compatível com os seus concorrentes. O único suporte dado a *procedures* é uma ferramenta de *debug* para usuários do banco de dados DB2. Em nenhum momento são disponibilizadas opções para a engenharia reversa deste artefato.

Outros bancos de dados também possuem ferramentas próprias para desenvolvimento de *procedures*, porém não foi encontrada a opção de geração de modelos a partir do código fonte.

2.2.
Artigos Diversos

2.2.1.
Towards The Reverse Engineering of UML Sequence Diagrams (Briand, 2004)

São comuns, hoje em dia, ferramentas que analisam código, principalmente se escrito na Linguagem Java, para gerar diagramas UML.

O objetivo do artigo é definir uma metodologia para gerar, através de engenharia reversa, um diagrama de seqüência partindo do rastreamento da execução de um código. A parte conceitual é pouco explorada, dando assim mais ênfase à prática e a exemplos.

Foi feita uma análise detalhada de algumas outras técnicas parecidas, levantando suas vantagens e desvantagens em determinadas situações. Pelo menos com relação a defeitos, há uma preocupação de não repetir os erros das concorrentes. Segundo o panorama apresentado, percebe-se claramente a vantagem em usar a metodologia do artigo. Um exemplo disto é que nenhuma das concorrentes exibe condicionais no diagrama de seqüência gerado.

A estratégia utilizada consiste basicamente em instrumentar o código, executá-lo e analisar os dados obtidos do rastreamento para gerar o diagrama final. O último passo é feito com a ajuda de dois metamodelos. O primeiro é adaptado da própria definição da UML e representa o modelo de construção dos artefatos presente em um diagrama de seqüência. O segundo é um esquema genérico para armazenar e organizar as informações obtidas do rastreamento: que método é chamado por quem, as condicionais, os iteradores presentes, etc. O código executado preenche o modelo de rastreamento, com a aplicação de algumas regras de transformação, para produzir um modelo de diagrama.

O artigo deixa claro que não tem por objetivo mostrar como gerar um diagrama de seqüência do modelo de diagrama apresentado. Isso porque ele considera fácil, além de não ser este o objetivo do trabalho, e sim facilitar e servir de infraestrutura para uma ferramenta CASE, por exemplo. Com relação às regras de mapeamento, elas são simples e servem não só para transformação de modelos, como também para validação dos diagramas gerados por ferramentas que seguem esta metodologia.

No final é apresentado um estudo de caso em que esta metodologia foi aplicada. A instrumentação do código foi feita dinamicamente por um aplicativo

escrito em linguagem de script e o código instrumentado era baseado em um software escrito por outro grupo que produziu também um diagrama de seqüência na fase de especificação, podendo ser assim comparado com o diagrama gerado pela engenharia reversa. Foi observado um resultado satisfatório no experimento, com a única diferença sendo um maior grau de detalhamento do diagrama gerado, pois ele representa uma visão fiel da execução.

No trabalho apresentado nesta dissertação, este artigo foi útil para mostrar quais caminhos devem ser seguidos, e quais caminhos podem ser evitados. A experiência passada pelo autor ajudou a delinear a estratégia utilizada no estudo de caso, como a criação de padrões e métodos de extração e leitura do código fonte.

O artigo tem como ponto forte os modelos apresentados. O modelo usado no rastreamento é genérico o suficiente para ser utilizado por qualquer linguagem, sendo adaptável para técnicas que utilizam analise estática de código. As regras apresentadas parecem ser realmente corretas e funcionais.

Como ponto fraco está o estudo de caso. Não foi mostrado como instrumentar o código para gerar o modelo de rastreamento. Isto não aparenta ser uma tarefa fácil, ainda mais com instrumentação gerada automaticamente. Na abordagem utilizada. o código precisa ser alterado para que a metodologia funcione. Existem técnicas em que isto não é necessário, como as que se utilizam ferramentas de apoio a *debug*. E mesmo instrumentado, não foi comentado como ler todas as entradas da tabela de rastreamento e produzir o modelo em questão.

Apesar de estar bem fundamentada, a seção de trabalhos relacionados não comenta a fundo técnicas de análise estática, como a utilizada pelo software Together[1]. Falta uma comparação de vantagens e desvantagens da opção de uso de rastreamento em detrimento à leitura estática do código. Uma vantagem clara é o seu uso em linguagens em que o comportamento da classe pode ser alterado dinamicamente, tendo assim instâncias de objetos distintos com comportamentos diferentes, como acontece com a linguagem *Ruby*.

[1] www.togethersoft.com

2.2.2.
A study on the current state of the art in tool-supported UML-based static reverse engineering (Kollmann, 2004)

Como citado no artigo da sessão 2.2.1, a UML surgiu e rapidamente se tornou padrão para representação e especificação de sistemas orientados a objetos.

O objetivo do artigo é comparar o modelo gerado por quatro diferentes ferramentas através da técnica de engenharia reversa, com análise estática do código. Cada ferramenta, indiferente do fabricante ou preço, fornece uma interpretação única para um mesmo código fornecido como entrada. Exemplo disto é a modelagem de relacionamentos, como agregação. São estas funcionalidades básicas, e as diferentes interpretações, que este artigo visa comparar.

As ferramentas escolhidas foram: Together, Rational Rose[2], IDEA[3] e FUJABA[4]. As duas últimas são ferramentas acadêmicas, desenvolvidas em projetos de pesquisa, sendo a IDEA um pouco mais conhecida atualmente, devido ao seu plug-in para a IDE de desenvolvimento IntelliJ[5]. Apesar de não ser tão recente, o artigo ainda é bastante atual, dado que nos últimos anos o foco da evolução destas ferramentas, foi principalmente na usabilidade e integração com novas IDEs, como o Eclipse e o NetBeans.

A avaliação foi feita examinando apenas o diagrama de classe gerado e os critérios de comparação utilizados foram: número de classes, número de associações, detecção de interfaces, multiplicidades e regras de nomenclaturas. Um sistema interno foi escolhido para servir de caso de estudo e no final os diagramas gerados eram comparados por uma ferramenta desenvolvida pelos autores do artigo. Esta ferramenta recebe como entrada arquivos XMI (gerados por todas as ferramentas utilizadas no estudo) e produz como saída uma listagem das diferenças existentes entre dois diagramas.

No que diz respeito às funcionalidades básicas, todas as ferramentas conseguiram cumprir o seu papel. As diferenças existentes entre os números encontrados são justificadas pela política utilizada pela ferramenta, o que na

[2] www.rational.com

[3] http://idea-uml.qarchive.org/

[4] www.fujaba.de

[5] www.jetbrains.com/idea/

prática não representa um problema, desde que exista um comportamento padrão e o usuário de cada ferramenta o saiba.

Nas funcionalidades avançadas, como reconhecimento de multiplicidade de relacionamentos e detecção de interfaces, por exemplo, houve uma discrepância grande das ferramentas comerciais e das ferramentas acadêmicas. Isso influencia muito o resultado final, pois o entendimento da lógica da aplicação fica muito prejudicado, principalmente para reconhecimento de padrões de projeto.

O artigo teve sucesso ao comparar o diagrama gerado, porém tem como ponto fraco não explorar outras características e outros modelos da UML suportados. Foi ignorado o diagrama de seqüência, por exemplo.

Não foi comentada a capacidade de geração do código através do modelo. E, principalmente, a maior falha do artigo foi não comentar a capacidade das ferramentas de adaptar o diagrama depois de uma mudança drástica ou refatoramento. Este ponto costuma ser problemático em ferramentas case.

Em nenhum momento foi evidenciada alguma capacidade de adaptação da máquina de gerar diagramas pelo programador, porque certamente não é uma funcionalidade disponível. Não é possível utilizar essas ferramentas como arcabouços que permitam reescrever parte de um código, ou algum *script* numa linguagem proprietária, por exemplo, para adaptar ou reaproveitar as funcionalidades existentes para alguns casos especiais. Resumidamente, as ferramentas apresentadas não se mostraram extensíveis.

O artigo serviu para confirmar que as ferramentas case comerciais tendem a estar um passo à frente das ferramentas acadêmicas, além de reforçar que nenhuma delas disponibiliza pontos de extensão para customizações de comportamento ou reutilizações de código.

Neste trabalho, o artigo foi útil para corroborar a teoria que não é possível fazer a aplicação desejada a partir das ferramentas disponíveis.

2.2.3. A reverse engineering methodology to reconstruct hierarchical data flow diagrams for software maintenance (Benedusi, 1989)

O artigo descreve a metodologia utilizada para gerar um diagrama de fluxo de dados a partir de um código escrito na linguagem Pascal. Dos artigos encontrados, é o que se aproxima mais do objetivo final deste documento.

Na primeira metade do artigo são discutidos conceitos genéricos da área de engenharia reversa. Ele apresenta quais são as estruturas da linguagem que

podem ser extraídas para o diagrama automaticamente, porém sem entrar em detalhes ou dar exemplos. Fica restrito ao campo conceitual. Comenta, por exemplo, que nem todo o conhecimento, decisões e experiências implementadas no código podem ser automaticamente extraídas. Mesmo com a afirmação anterior, é dito que sempre se pode melhorar e aproveitar as informações obtidas. Esse tipo de discussão perdurou durante boa parte do artigo, claramente para justificar as decisões tomadas na solução utilizada. Um exemplo disto foi o comentário, de que baseado na experiência dos autores, uma metodologia de engenharia reversa deve ser guiada pelo documento a ser produzido e pelo artefato de entrada, num processo que se inicia com a análise do documento desejado e termina com o diagnóstico do que se pode retirar das informações de partida, gerando uma regra de transformação.

Na segunda metade do artigo foi apresentada a regra de transformação utilizada para gerar um DFD a partir de um código Pascal. O primeiro passo é a geração de um “Structure Chart” (SC). Isso nada mais é, do que um grafo em que os métodos são vértices e as chamadas entre eles são arestas. Outras informações básicas são necessárias, como as variáveis utilizadas nas chamadas. Feito isso, basta aplicar algumas regras e no final é gerado um DFD. Essas regras são fáceis de serem aplicadas, apesar de serem de difícil entendimento, devido à complexidade do texto escrito usando um padrão mais voltado para a matemática. A primeira regra, por exemplo, fala que todo nó folha, vira um processo no último nível do DFD gerado. Além dessas, outras regras são fornecidas para definição do que seria considerado um depósito de dados, entidade externa e os parâmetros passados nos fluxos de dados. Ao final, realmente foi mostrado que foi possível atingir o objetivo inicial.

Como dito anteriormente, em nenhum momento houve uma preocupação em generalizar o processo para outros casos parecidos. O foco sempre foi Pascal e DFD. A linguagem de entrada poderia ser apenas um detalhe na aplicação modelo desenvolvida, se fosse produzido um *framework* para transformação de SCs em DFDs. Apesar disso, as regras apresentadas podem ser utilizadas em outras linguagens facilmente. Também não foi apresentada nenhuma estrutura ou modelo de dados utilizado para fazer a transformação.

2.2.4.
Extracting Entity Relationship Diagram from a Table-based Legacy Database (Yeh et al., 2005)

Este artigo relata um trabalho sobre extração de um modelo de entidades e relacionamentos (MER) a partir da leitura das informações presentes no banco de dados.

Ele descreve, sem entrar em detalhes práticos, um processo genérico para Engenharia Reversa de Banco de Dados (ERBD). Este processo é composto por três passos: preparação do projeto; extração dos dados; e geração dos modelos.

A preparação do projeto consiste simplesmente em obter e preparar os artefatos a serem utilizados, como arquivos DDL, scripts de criação de banco, entre outros. O segundo passo é o mais complicado, pois engloba uma heurística de extração dos dados, que pode ser simples ou complexa, dependendo do nível de detalhe e padronização do material recolhido no primeiro passo. É neste momento que todos eles serão agrupados e sintetizados em uma estrutura computacional para dar início ao último processo, que é justamente a geração do modelo de entidades e relacionamentos. Abaixo, o processo é detalhado especificadamente para o trabalho realizado pelo grupo.

Percebeu-se que não seria possível extrair toda a semântica necessária apenas olhando para o banco de dados, pois muitas das vezes os nomes dos campos, assim como os nomes das tabelas, não eram coerentes com o domínio da aplicação. Para resolver isto, na fase de preparação foi feito um mapeamento de formulários com os valores preenchidos. Preenchiam-se os formulários do sistema com seus respectivos valores de teste. A fase de preparação era a ação descrita acima, junto com a extração dos esquemas das tabelas com os dados para um meta-modelo. Este meta-modelo consiste em algumas tabelas descritivas genéricas, nas quais todo o banco de dados é resumido e convertido para este novo esquema. Isto permite um padrão na hora de leitura dos dados, independente do modelo físico real.

Feita a preparação, a extração dos dados é realizada através de uma heurística de correlação entre os dados obtidos do formulário e os dados obtidos da tabela. São apresentadas em detalhes as regras usadas para tal. No final são descobertas as tabelas e os atributos do modelo de domínio, assim como as chaves primárias e chaves estrangeiras.

O próximo passo é o mais simples. Com todas as informações devidamente extraídas, só resta a escolha do estilo final do MER, como o modo de tratar um relacionamento N para N, por exemplo. No final, um diagrama

praticamente pronto é gerado. Só restam ajustes finos de nomenclatura que estão devidamente documentados.

Apesar de curto, o artigo colaborará muito com alguns pontos importantes na dissertação. O primeiro deles é o modo de descrever o processo de engenharia reversa. Foi possível notar um estilo acadêmico de descrever os fatos, que serve de exemplo para alguns tópicos adotados nesta dissertação. Nisto se inclui o modo como foi dividido o processo de trabalho. Os três passos adotados facilitam a organização do raciocínio, o que permite maior facilidade na definição de um cronograma e as tarefas a realizar.

Além do processo de trabalho, foi utilizada a idéia da criação de um meta-modelo do passo de preparação para a extração dos dados. Só assim era possível montar um meta-ambiente que fosse indiferente às fontes obtidas, dando, portanto, flexibilidade.

2.2.5. Inverse Transformation of Software from Code to Specification (Sneedm, 1988)

Como transcrito do próprio artigo: “Este documento descreve o esquema suportado por ferramentas automáticas para transformação reversa do código para especificação pelo processo de engenharia reversa”.

Na prática, ele não apresenta nenhuma técnica ou metodologia para fazer a transformação e sim conceitos básicos, porém essenciais para quem quer realizar um trabalho nesta área, principalmente com linguagens estruturadas, já que foi escrito para este tipo de tecnologia. Pelo fato de ter sido produzido no final da década de 80, são usados muitos termos e situações que estão em desuso atualmente, com exceção de desenvolvedores que dão manutenção, ou programam módulos, para sistemas antigos. Um exemplo disso é o uso de arquivos físicos para armazenarem dados e informações ao invés de gerenciadores de banco de dados.

Um software é apresentado como uma composição de três níveis de abstração com objetivos distintos: conceitual, lógica e física. O primeiro é a visão da rede de entidade com seus atributos e seus relacionamentos. O segundo é a visão da organização dos dados e o terceiro é uma visão mais baixo nível, perto da implementação, chegando próximo a um pseudocódigo.

São apresentados alguns conceitos, como motivos para se realizar a engenharia reversa, afirmações, como a que diz que é mais fácil alterar programas com modelos conceituais existentes, e regras para se ter uma boa

especificação. A especificação é dividida em duas categorias: especificação de dados e de programa. Essas categorias são subdivididas em outras subcategorias. O código então é partilhado em grupos de elementos, como arquivos de execução, arquivos de armazenamento de dados, entre outros. No final são apresentadas regras de mapeamento, que na verdade servem como um guia de boas práticas para derivar uma subcategoria da especificação de um grupamento originado do código. São regras que estão no campo conceitual, sem nenhuma aplicabilidade prática. Isto porque não são apresentados modelos reais de especificações a serem produzidos e sim categorias de artefatos e as regras que mapeiam o código para elas.

O artigo é muito útil para organizar os conceitos, apesar de ser um tanto datado. Com ele é possível expressar mais facilmente o objetivo e que tipo de artefato se deseja produzir quando se está imaginando um processo ou programa para engenharia reversa. Talvez por isso seja citado por praticamente todos os outros artigos referenciados anteriormente, tornando sua leitura obrigatória.

2.3. Frameworks para geração de diagramas

Existem alguns produtos que possuem objetivos semelhantes à parte deste documento que objetiva a criação de um *framework* para diagramação de gráficos. Seguem abaixo os principais.

2.3.1. Eclipse Modeling Framework (EMF)

O EMF (Eclipse Foundation, 2008) é um *framework* Java para tratar aplicações que fazem uso de modelo, como digramas de classes, diagramas de estados, entre outros. Basicamente, o foco do projeto são linguagens orientadas a objeto e modelos definidos na UML. Com isso é fácil criar um editor ou um diagrama com opções de *drag and drop* de elementos e outras funcionalidades típicas de ferramentas *case*, como geração de código.

A entrada para a criação dos modelos pode ser feita de várias formas, desde arquivos XML até classes escritas na linguagem Java. Isto permite que outras aplicações consigam se integrar facilmente a ela. Na prática basta entrar com os dados do diagrama em um editor básico fornecido pelo *framework* e então obter algo parecido com a figura.

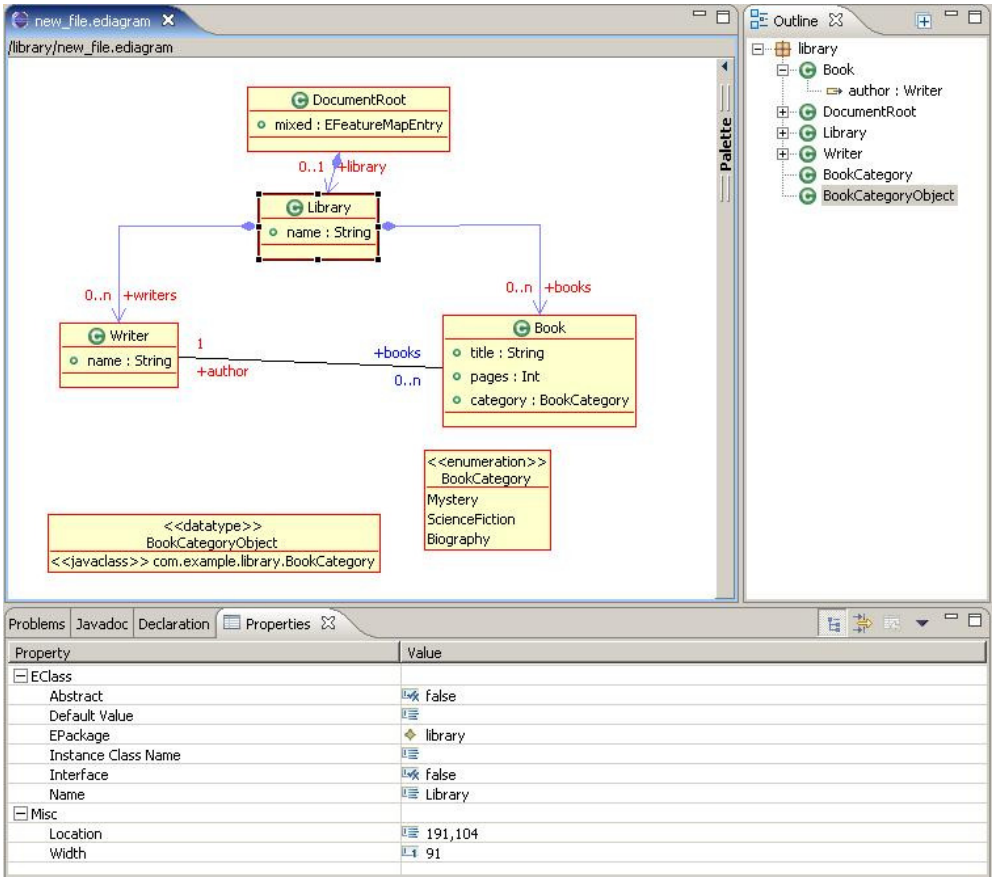

Figura 2: Exemplo de um diagrama gerado com o EMF

Caso se deseje criar um aplicativo para gerar diagramas de seqüência a partir de código Java, por exemplo, basta criar um interpretador que leia o código e produza os metadados necessários para criar o diagrama. O trabalho de gerar os dados seria o mais complicado, pois o EMF se encarregaria de exibir o diagrama. Porém, já não se torna tão confortável quando se deseja adicionar ações específicas, como dois cliques em uma classe. Todas essas ações são chamadas seguindo o modelo tradicional de eventos do Java. Não existe um modelo de interação, em que seja permitido configurar no mesmo XML o modelo e as ações sobre ele. Isto dificulta um pouco a criação de algo genérico acoplado a um aplicativo de terceiros. As facilidades citadas para criação de elementos UML começam a desaparecer quando se deseja usar outros modelos, como por exemplo um DFD.

Outro ponto crítico é não ser possível usar o EMF fora do ambiente Eclipse. Para utilizá-lo é necessário obrigatoriamente ser um *plug-in* da IDE citada. Em parte, isto se justifica pelo fato dele usar internamente o GEF para desenhar os elementos na tela, o que obriga por conseqüência o uso do SWT.

Ou seja, não é possível criar uma aplicação Swing[6] fora do Eclipse e fazer uso do GEF. Isto restringe e muito o público alvo do *framework*.

O EMF se mostrou um ótimo *framework* para criação de *plug-ins* para o Eclipse com o objetivo de manipular editores de modelos orientados a objetos, provenientes de engenharia reversa ou não, e gerar códigos na linguagem Java. Como são poucos, senão nulos, os concorrentes e *frameworks* com objetivos semelhantes, ele deve sempre ser levado em consideração em projetos que envolvam engenharia reversa e geração de diagramas.

2.3.2. Graphviz

Graphviz (AT&T, 2008) é um projeto de código aberto para visualização de grafos. Na figura abaixo é possível ver um exemplo de um diagrama produzido pelo produto.

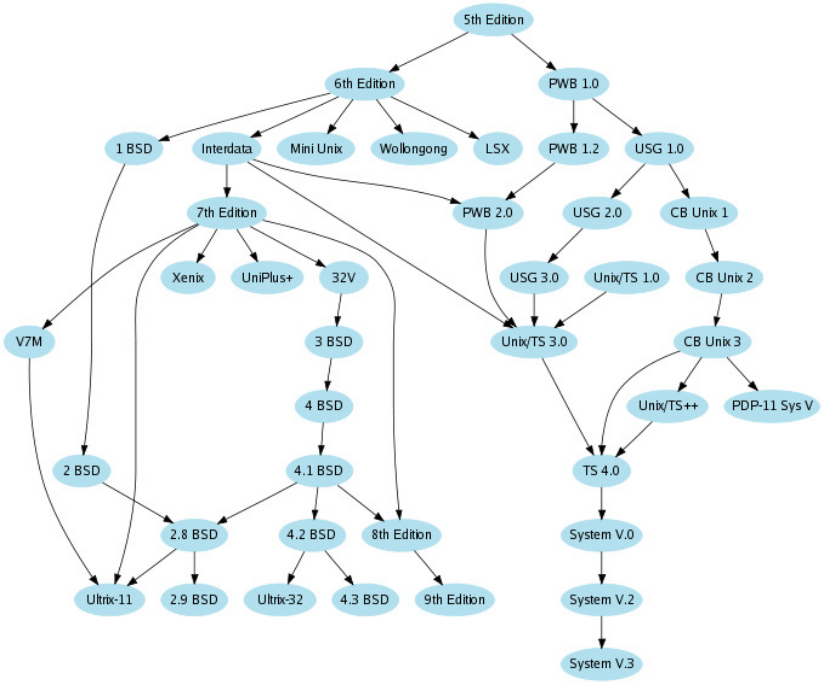

Figura 3: Diagrama gerado pelo Graphviz

Ele não se propõe a ser uma alternativa a programas como o Visio, da Microsoft. Isto porque não existe um editor amigável para criar o diagrama. Ao invés disso, é utilizado um arquivo de texto simples escrito na linguagem *Dot* (AT&T, 2008). Ela foi desenvolvida pelo projeto para ser uma linguagem voltada

[6] Swing é a API padrão do Java para manipulação e criação de interfaces, ou aplicativos.

para a criação de gráficos. O segredo da linguagem é ser simples e de fácil aprendizado.

O Graphviz não é um software, e sim um projeto que engloba diversas tecnologias e ferramentas focadas na visualização de grafos. Existem vários aplicativos, como o que transforma um arquivo escrito em *Dot* em uma imagem SVG, ou mesmo *Postscript.* O mais conhecido, porém, é o que permite a visualização direta do diagrama a partir de uma especificação escrita em *Dot*, sem a necessidade de gerar imagens intermediárias.

Ele é uma ótima ferramenta para visualização, pois é possível modificar completamente o layout do grafo, com customização de cores e imagens dos nós, o que o torna capaz de exibir diagramas complexos, como um DFD.

O seu ponto fraco é não ser possível interferir no diagrama gerado. Ele fornece a disposição dos nós, e não é permitido alterar isto. Entretanto, o algoritmo utilizado é muito bom e consegue organiza os nós numa disposição aceitável. Além disso, ele é estático, o que não torna possível utilizá-lo dentro de uma aplicação maior com um modelo de interação dinâmico.

As limitações acima impedem o projeto de ser usado em uma ferramenta com o foco em engenharia reversa, pois os diagramas gerados por estes produtos nem sempre são fieis ao que o usuário deseja ter como documento final. Então a necessidade de alteração e customização dinâmica faz com que o Graphivz não seja uma boa alternativa, existindo melhores opções.

2.4. Ferramentas Utilizadas

2.4.1. Visual Library

Para exibição do diagrama no *Framework* apresentado no capítulo 4, foi usada a API de desenho Visual Library (Netbeans, 2008). “Visual Library é a próxima geração de bibliotecas para manipulação gráfica”. Essa frase, exposta no site principal do projeto, mostra bem como a ferramenta é poderosa. Ela foi concebida inicialmente como parte da IDE de desenvolvimento NetBeans. A sua criação foi motivada pelos diversos módulos que exigiam manipulação de diagramas e gráficos. Seguem como exemplo duas telas da IDE que usa a biblioteca gráfica.

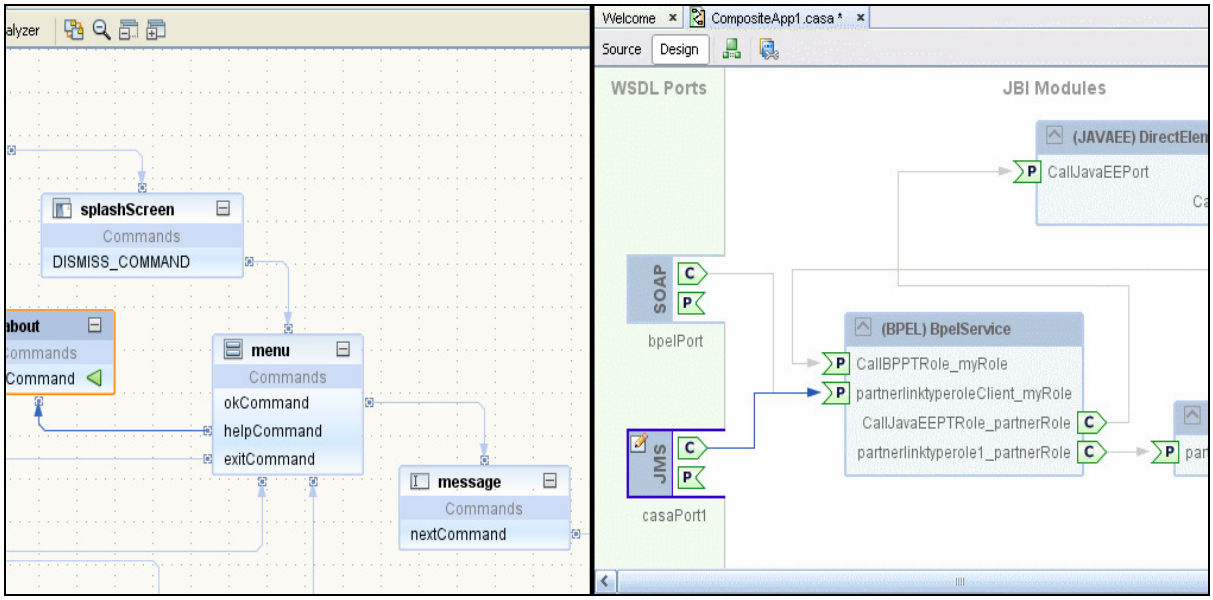

Figura 4: Desenvolvimento móvel e criação de aplicativos SOAP

Na Figura 4, à esquerda é visualizada a ferramenta de criação de aplicativos voltados ao mercado de aparelhos portáteis, como celulares e PDAs. Um exemplo do módulo de desenvolvimento de aplicativos SOAP é apresentado à direita. Além desses dois, existe o módulo de diagramação e geração de documentos UML, que também lança mão dessa tecnologia. Como pode ser observado, uma necessidade interna ao projeto *NetBeans* fez nascer a *Visual Library*.

A biblioteca é de código aberto como é comum em artefatos ligados ao projeto *NetBeans*, podendo ser alterada e usada gratuitamente em todas as situações, sem restrições.

O seu uso não está limitado a aplicativos internos ao ambiente de desenvolvimento. Qualquer processo Java, sem nenhuma restrição de máquina ou ambiente de desenvolvimento, pode se aproveitar de suas funcionalidades.

Basicamente o que a ferramenta faz é auxiliar a criação de softwares que utilizam diagramas em sua interface gráfica. Todo o trabalho de criar conexões entre os elementos, organizar o layout e editar as posições de todos os artefatos apresentados, pode ser realizado sem nenhuma dificuldade prática. O modo de programar é muito parecido com o Swing, o que facilita quando existe um conhecimento prévio da API de desenho default do Java. Além disso, é intuitivo e rápido. Todo o estilo dos diagramas, assim como o comportamento dos seus elementos na interface, são altamente customizáveis. Existe uma configuração padrão, e programaticamente é possível mudá-la.

Neste trabalho, esta API foi de fundamental importância. Ela diminuiu o tempo de desenvolvimento da geração dos diagramas, além de fornecer um acabamento profissional ao projeto. Vale lembrar que ela não facilita a geração do diagrama propriamente dito, e sim a implementação de uma ferramenta que objetive isso. Diferente do EMF (Capítulo 2.3.1), ele é apenas uma *API*, e não uma aplicação.

Para compará-la com outras ferramentas do gênero, podemos citar o GEF, do projeto Eclipse. O objetivo é exatamente o mesmo, porém com um grau de complexidade maior. Por ser mais antiga, muito das suas dificuldades de uso foram retiradas pela *Visual Library*, que possui em sua *API* elementos similares aos do GEF, além de outras referências que deixam nítido de onde veio a inspiração do projeto. O desenvolvimento deve ser feito usando a *API* gráfica SWT, que é menos conhecida que o Swing. Além disso, mesmo sendo código aberto, é difícil de baixar e compilar. Perde-se muito tempo ao tentar fazer isso. Com a *Visual Library*, este trabalho é mínimo. Essa característica dificulta o entendimento de como ela funciona internamente, exercício fundamental no uso de recursos avançados.