Universidade Federal de Campina Grande
Centro de Ciências e Tecnologia
Curso de Mestrado em Informática
Coordenação de Pós-Graduação em Informática

SAMOA - Sistema de **A**poio à **M**odelagem Orientada a **O**bjetos de **A**plicações

Dissertação de Mestrado

Edemberg Rocha da Silva

Campina Grande – PB
Dezembro – 2003

Universidade Federal de Campina Grande
Centro de Ciências e Tecnologia
Curso de Mestrado em Informática
Coordenação de Pós-Graduação em Informática

SAMOA - **S**istema de **A**poio à **M**odelagem Orientada a **O**bjetos de **A**plicações

Edemberg Rocha da Silva

Dissertação submetida à Coordenação de Pós-Graduação em Informática do Centro de Ciências e Tecnologia da Universidade Federal de Campina Grande como requisito parcial para a obtenção do grau de Mestre em Ciências (MSc).

Área de concentração: Bancos de Dados

Prof. Dr. Ulrich Schiel
(Orientador)

Campina Grande – PB
Dezembro – 2003

FICHA CATALOGRÁFICA

SILVA, Edemberg Rocha da

S586S

SAMOA – **S**istema de **A**poio a **M**odelagem Orientada a **O**bjeto de **A**plicações

Dissertação (Mestrado), Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande – PB, Dezembro de 2003.

108 p. Il.

Orientador: Ulrich Schiel, Dr.

Palavras Chave:

1. Engenharia de Software
2. Padrões de Projeto
3. Detecção automática de Padrões de Projeto
4. Geração de Código
5. Geração de Críticas
6. UML
7. XMI

CDU – 519.683

“*SAMOA* - SISTEMA DE APOIO A MODELAGEM ORIENTADA A OBJETO DE APLICAÇÕES”

EDEMBERG ROCHA DA SILVA

DISSERTAÇÃO APROVADA EM 22.12.2003

PROF. ULRICH SCHIEL, Dr.
Orientador

PROF. DALTON DARIO SEREY GUERRERO, Dr.
Examinador

PROF. ARTURO HERNÁNDEZ DOMÍNGUEZ, Dr.
Examinador

CAMPINA GRANDE – PB

A minha grande mãe, Suzana

Agradecimentos

É muito gratificante terminar mais uma etapa em nossa vida e poder reconhecer a ajuda, a compreensão e o carinho de tantas pessoas. Melhor ainda, é poder dar-lhes sinceros agradecimentos.

Agradecimentos mais que especiais que desejo à (aos):

- **Deus** por ter me dado uma vida saudável, repleta de felicidades e de pessoas especiais.
- Meus pais Edmilson e Suzana, em especial minha grande mãe, pelo carinho, apoio, tolerância, incentivo e fé em mim. Sinto a presença de vocês ao meu lado principalmente nos momentos de dificuldades.
- Meus irmãos Edman, Savana e Sonaly (também afilhada) e a minha avó Inês pelo carinho.
- Sinceros agradecimentos ao meu orientador Ulrich Schiel, pelo empenho, compreensão e seriedade no decorrer de todo este trabalho.
- Glória Coeli e Lúcia Melo pelo incentivo e fé que depositam em mim.
- Minhas amigas, Carol de Jesus e Pasqueline, que começaram comigo esta empreitada e que sempre me ajudaram nos momentos mais difíceis.
- Aos meus grandes amigos Thiago Palmeira e Ericke Ramalho, pela paciência e apoio solidário. Pessoas como vocês me fazem crer que o sentimento de amizade, realmente, existe.
- Funcionária da COPIN, Aninha, pelo apoio e incentivo.
- CAPES pelo suporte financeiro durante todo o trabalho de mestrado.

Sumário

Capítulo 1 – Introdução..1

1.1.Propósito da dissertação...3
1.2. Motivações...4
1.3. A organização da dissertação...6

Capítulo 2 – Fundamentação Teórica...7

2.1. Modelagem Orientada a Objetos..7
2.2. A linguagem UML...9
2.2.1. Propósito de se trabalhar com a UML....................................10
2.2.2. Diagramas propostos pela UML...11
2.2.3. OCL – Object Constraint Language……………………………17
2.2.4. Conclusão...18
2.3. O Padrão XMI..19
2.3.1. Características do XMI...19
2.3.2. Os requisitos do projeto XMI...20
2.3.3. Mapeamento de modelos e meta-modelos MOF em documentos e DTDs XML...21
2.3.4. Cenários de utilização do XMI..22
2.4. Padrões de Projeto...26
2.4.1. Origem dos Padrões de Projeto..27
2.4.2. Que é um padrão de projeto?..28
2.4.3. Como padrões de projeto solucionam problemas de projeto..29
2.4.4. Com selecionar um padrão de projeto....................................29
2.4.5. Como usar um padrão de projeto...32

2.4.6. Padrões mais comuns..33
2.5. Assistentes Automatizados..39
2.6. Meta-modelagem e Padrões..41
2.7. Trabalhos Relacionados...44
2.8. Conclusão...45

Capítulo 3 – Sistema de Apoio a Modelagem Orientada a Objetos de Aplicações – SAMOA...46

3.1. O meta-modelo de Meijers..48
3.2. Arquitetura do SAMOA...49
3.3. Considerações para detectar padrões...53
3.4. Processo de detecção de padrões..56
3.4.1. Detecção de padrões em classes Java.................................56
3.4.2. Detecção de padrões em diagramas de classe UML.............60
3.5. Geração de Críticas..64
3.6. Processo de Instanciação de Padrões..65
3.7. Conclusão...71

Capítulo 4 – Desenvolvimento do SAMOA..72

4.1. Modelo de domínio..72
4.2. Levantamento dos Requisitos...73
4.3. Planejamento dos Incrementos...75
4.4. Implementação e testes..77
4.4. Diagrama de classes e de sequencia..78
4.5. Conclusão...82

Capítulo 5 – Estudo de Caso...84
Capítulo 6 – Conclusões...95
Anexo..104

Lista de Abreviaturas

DTD	Document Type Definition
GoF	Gang of Four
ISO	International Organization for Standardization
MOF	Meta Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OO	Orientação a Objetos
UML	Unified Modeling Language
W3C	World Wide Web Consortiun
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Lista de Figuras

FIGURA 2.1: exemplo de um modelo UML com restrições OCL............................18
FIGURA 2.2: uma junta esquadrilha..30
FIGURA 2.3: uma junta "rabo de pomba"..30
FIGURA 2.4: Abstract Factory utilizado para o layout de um jardim.......................34
FIGURA 2.5: exemplo em Java do padrão Singleton...36
FIGURA 2.6: algoritmos para detecção de padrões e para reconhecimento
das entidades de cada tipo.............................. 43
FIGURA 3.1: interoperação entre o SAMOA e uma ferramenta CASE WEB.........47
FIGURA 3.2: diagrama de classe UML simplificado do meta-modelo de Meijers..48
FIGURA 3.3: arquitetura do SAMOA..50
FIGURA 3.4: PatternsBox como núcleo do SAMOA...57
FIGURA 3.5: detecção de padrões em classes Java pelo SAMOA.......................58
FIGURA 3.6: representação UML de uma instancia do padrão Composite...........59
FIGURA 3.7: arquivo XML com os padrões detectados nas classes submetidas..59
FIGURA 3.8: relacionamento do processo de detecção em diagramas UML vs. classes Java... 62
FIGURA 3.9: exemplo de uma realização do padrão Observer.............................62
FIGURA 3.10: XMI simplificado do diagrama de classe...63
FIGURA 3.11: processo de detecção em diagramas de classe UML.....................64
FIGURA 3.12: processo de instanciação do padrão Composite............................66
FIGURA 3.13: estrutura do padrão *Composite*..67
FIGURA 3.14: seqüência do processo de instanciação..69
FIGURA 4.1: modelo conceitual do SAMOA...73
FIGURA 4.2: diagrama de casos de uso...74

FIGURA 4.3: diagrama de classes...................................... 79
FIGURA 4.4: seqüência do processo de detecção em diagramas UML................80
FIGURA 4.5: seqüência do processo de instanciação de um padrão....................81
FIGURA 5.1: simplificado diagrama de classes para um editor de texto com figuras 84
FIGURA 5.2: XMI representando o diagrama da figura 5.1....................................85
FIGURA 5.3: página principal do SAMOA..86
FIGURA 5.4: página para capturar o arquivo XMI..86
FIGURA 5.5: buscando o XMI para upload...87
FIGURA 5.6: fazendo upload do XMI..88
FIGURA 5.7: visualização do padrão detectado..89
FIGURA 5.8: conteúdo do XML gerado...89
FIGURA 5.9: link para instanciar padrões...90
FIGURA 5.10: lista dos padrões cadastrados...91
FIGURA 5.11: fornecendo dados para o formulário...91
FIGURA 5.12: disponibilizando fontes Java...92
FIGURA 5.13: arquivos fontes gerados a partir dos dados do formulário..............92

Lista de Tabelas

TABELA 3.1: classes submetidas a detecção de padrões pelo SAMOA...............58
TABELA 3.2: definição do modelo abstrato do Composite.....................................67
TABELA 3.3: construção do modelo concreto Composite......................................69
TABELA 3.4: fontes Java gerados...70
TABELA 4.1: requisitos funcionais..74
TABELA 4.2: pacotes de serviço...77
TABELA 4.3: relacionamento entre os pacotes e os módulos do sistema.............78

Resumo

Padrões de projeto são considerados uma das mais valiosas tecnologias para produzir software de qualidade. Uma técnica para melhorar o uso de padrões é identificar suas realizações e inferir um conhecimento para melhorá-las. Esta tarefa de encontrar todas as realizações de padrões em um projeto caracteriza-se por ser tediosa para o engenheiro de software. Nessa dissertação apresentamos um sistema assistente para programadores e arquitetos de software, chamado SAMOA (**S**istema de **A**poio a **M**odelagem Orientada a **O**bjetos de **A**plicações). Este sistema é um assistente interativo para automatizar o trabalho de detecção de realizações de padrões de projetos. Basicamente, o SAMOA é capaz de automaticamente (i) encontrar padrões aplicados em diagramas de classes UML e em fontes JAVA; (ii) produzir possíveis criticas sobre esses padrões. Depois que esses são detectados, um conjunto de críticas de projetos são verificadas para testar se a realização dos padrões pode ser melhorada. E (iii) instanciar padrões visando à geração de código do mesmo, na linguagem de programação Java. Foi implementado um protótipo do sistema que realiza as atividades (i) e (iii). Abordamos, também, quais diferenciais nosso sistema tem em relação aos demais existentes.

Abstract

Design patterns are considered one of the most valuable technologies to produce quality software. A technique to improve the use of patterns is to identify their realizations and to induce a knowlege to enhance their use. This work to find all pattern realizations in a software design can be tedious for the software engineer. In this dissertation we show an assistant system for programmers and software architets, called SAMOA (**S**istema de **A**poio a **M**odelagem Orientada a **O**bjetos de **A**plicações). This system is an interactive assistant to automate the work of detection of the realizations of design patterns. On principle, SAMOA is able to automatically (i) find patterns applied in UML diagrams and in JAVA;sources (ii) make critiques about these patterns. If a pattern has been detected, a set of design critiques are verified to test if the realization of the pattern can be improved. And (iii) instanciate patterns to aim an automatic code generation in the JAVA programming language. We have implemented a prototype of the system that realizes activities (i) and (iii). We also discuss , wich properties differentiate our system from existing others.

Capítulo 1 - Introdução

Recentemente, padrões de projeto (*design patterns*) têm-se tornado um forte auxílio para a comunidade de programação orientada a objetos. Eles descrevem uma solução geral para um problema típico de projeto de software. A solução é descrita em um formato padrão que consiste de uma estrutura genérica de projeto para uma solução (expressa em alguma terminologia de elementos de projeto) junto a uma descrição textual de padrões, indicando, por exemplo, quando usar a solução e como aplicá-la numa determinada situação. Daqui por diante, quando falarmos em padrões, estaremos nos referindo aos padrões de projeto.

Embora muitos dos padrões não sejam específicos para orientação a objetos - *OO* (isto é, eles podem ser usados em projetos não OO), as estruturas de projeto são, geralmente, expressas em uma terminologia orientada a objetos, ou seja, em termos de classes, interfaces, métodos, atributos e relacionamentos. Isto significa que aplicar padrões no desenvolvimento de um sistema OO é, a principio, relativamente direto, desde que a terminologia possa ser mapeada diretamente para a linguagem de construção.

Padrões oferecem vários (potenciais) benefícios ao desenvolver software OO. Primeiro, há a possibilidade de reuso. Aplicar uma solução que tem sido desenvolvida e utilizada antes, pode evitar trabalhos de projetos que normalmente ocorrem, especialmente o trabalho investido em procurar por soluções que não satisfazem um problema em mãos. Usar padrões também permite que as

discussões sejam concentradas em decisões importantes, tais como "deveríamos permitir em tempo de execução variações deste comportamento?" ou "onde deverá ser a criação desses objetos?". Numa forma similar, padrões fazem comunicações entre os desenvolvedores mais eficazes. Usando padrões, poderemos evitar discussões sobre o porquê que certas classes estão organizadas e programadas numa certa maneira. Também, a tarefa de entender um programa escrito por outros se torna mais fácil quando temos conhecimentos sobre e onde certos padrões são usados.

Problemas com o uso de padrões

Usar padrões no desenvolvimento de um software OO não é um processo trivial. É preciso identificar a necessidade para um certo padrão pelo reconhecimento de um problema e escolher uma solução particular. Uma vez que isso é feito, o padrão tem de ser integrado com o projeto que já se encontra disponível. Em geral, isto significa que os elementos de projeto da descrição do padrão têm de ser mapeado e integrado com os elementos de projeto. Mais especificamente, um desenvolvedor deve decidir quais classes em um programa desempenharão os papéis definidos pelas classes na descrição do padrão, quais métodos desempenharão os papéis dos métodos definidos nas classes dos padrões, etc. É natural que isso poderá, também, conduzir a situações onde os elementos de projeto em um programa desempenhem múltiplos papéis (correspondendo aos elementos de projeto em vários padrões). Por exemplo, uma classe assumindo o papel do "*abstract factory*" no padrão Abstract Factory[1], poderá também desempenhar o papel de uma classe *singleton* no padrão Singleton. As ocorrências de padrões, em um projeto, afetam toda a organização de um programa (que classes existem, como elas estão associadas e como as hierarquias de classes estão organizadas). Contudo, o inverso também é verdade; a estrutura de programa existente influencia a forma como aplicaremos um

[1] Todos os padrões discutidos nesta dissertação foram retirados de [Gamma et al. - 2000].

padrão. Por exemplo, muitos padrões definem heranças hierárquicas. Se quisermos combinar dois desses padrões no mesmo local do nosso programa (por exemplo, combinando um *composite* com um *observer*), teremos que decidir qual dessas duas hierarquias está liderando e encontrar uma solução diferente para a integração de outro padrão.

Então, ao aplicar um padrão, o desenvolvedor deverá estar ciente da organização geral do projeto, e preparado para reorganizá-lo baseado em novas visões. Finalmente, usar um padrão em algum lugar em um projeto poderá impor restrições no desenvolvimento futuro dos elementos envolvidos. Por exemplo, se tivéssemos aplicado um padrão Proxy em algum lugar, devemos assegurar que a classe *proxy* implementa todas as operações definidas na sua super classe e delegá-las - sempre que necessário - ao objeto que a representa. Se, mais tarde, as operações forem adicionadas à super classe (possivelmente por um outro desenvolvedor) a classe *proxy* terá de ser adaptada também. Contudo, se nenhuma precaução for tomada (através de uma documentação, por exemplo), será fácil esquecê-las e, em alguns casos, será difícil encontrar os erros que foram introduzidos.

1.1. Propósito da dissertação

Nesta pesquisa, desenvolvemos uma ferramenta para dar suporte ao processo de projeto de software baseado em padrões. Chamamos, então, essa ferramenta de **SAMOA** - ***S****istema de* ***A****poio à* ***M****odelagem Orientada a* ***O****bjetos de* ***A****plicações*.

A idéia base é que padrões podem ser vistos como um tipo de blocos de construções para projetistas OO. Uma ferramenta que possa prover uma habilidade para detectar, instanciar e auxiliar na melhor construção desses blocos, poderá ajudar a construir projetos mais consistentes e implementações inteiras.

Nossos requisitos determinam que o SAMOA, ferramenta para projetos baseados em padrões, possa ser capaz de:

- identificar a realização de padrões tanto em um processo de reengenharia em classes Java, quanto em diagramas de classes UML (*Unified Modeling Language*) [Booch et al. - 2002];
- gerar código fonte para cada instância de um padrão;
- sugerir melhorias para uma melhor realização desses padrões;
- servir como um auxílio às ferramentas *CASE* que trabalham com edições de diagramas UML. Uma extensão funcional no tocante ao emprego de padrões.

1.2. Motivações

Esses requisitos a serem atendidos pelo SAMOA, deram-se pelas seguintes motivações:

- a detecção de padrões em classes *Java* auxilia o engenheiro de software em um processo de engenharia reversa. Dado um conjunto de classes *Java,* o SAMOA pode extrair conhecimentos dele. Tais conhecimentos tratam-se de exibir ao engenheiro padrões que foram usados para a construção das classes, ajudando na documentação do sistema.

- já no caso da detecção em diagramas UML, depois de editado um diagrama de classes, o usuário poderá utilizar o processo de detecção do SAMOA para verificar a coerência entre o projeto concreto e as intenções do engenheiro. Uma discrepância entre o resultado deste processo e as intenções do engenheiro poderá acarretar as seguintes conseqüências: se o engenheiro quis empregar um padrão mas este não pôde ser detectado, provavelmente o projeto está errado ou pode ser melhorado. Outro caso seria a detecção de um padrão onde o

engenheiro não planejou usar, podendo auxiliar em uma melhor compreensão e documentação do projeto. As ferramentas que existem hoje para o processo de detecção, foram desenvolvidas apenas para detectar padrões em códigos fontes, ou seja, só em processos de reengenharia. Nenhuma preocupam-se com projetos em construção, que é um dos requisitos do SAMOA.

- no processo de instanciação, o usuário poderá solicitar a implementação de um padrão específico, tendo para isto uma interface, um formulário, de fácil manuseio. Atualmente, nenhuma ferramenta dispõe de facilidades para se fazer esta construção.

- no processo de geração de críticas, existem ferramentas capazes de criticar/sugerir melhorias a projetos. É o caso, por exemplo, da ferramenta CASE POSEIDON [Poseidon - 2003]. Mas ela não dispõe de um conhecimento sobre padrões. Ela não é capaz de detectar padrões, tampouco sugerir críticas para melhor realizá-los. O SAMOA pode então ser agregado às ferramentas CASEs, que editam diagramas UML, para que estas possam trabalhar com padrões de projeto.

Até aqui, o leitor poderia estar se perguntando: se um padrão foi detectado, para que serve esta informação para o projetista? Explicaremos baseados na seguinte analogia, um compilador de um programa verifica se existe erros de sintaxe e de semântica em um específico programa, retornados ao usuário caso sejam detectados. O que acontece com o SAMOA é um processo um pouco semelhante. Nosso sistema executa uma espécie de checagem para verificar a existência de um padrão em seu projeto. O sistema fará uma varredura nos fontes do usuário a fim de extrair conhecimento com relação ao uso de padrões. Ele poderá acreditar que está utilizando um padrão, mas o SAMOA poderá não detecta-lo e essa não detecção servirá para abrir os olhos do projetista para reavaliar seu projeto. No momento, o SAMOA não é capaz de informar o que

estaria faltando para um determinado diagrama ou código Java está faltando para, realmente, representar um determinado padrão desejável pelo usuário.

1.3. A organização da dissertação

O restante da dissertação está organizado da seguinte forma: no Capítulo 2 são discutidas as tecnologias relacionadas com esta pesquisa e cruciais para o entendimento da construção do SAMOA, assim como as razões pelas quais elas foram escolhidas. No 3 são apresentadas as funcionalidades do sistema. Veremos o SAMOA de forma mais aprofundada. No Capítulo 4 discutimos aspectos relevantes desde a fase de análise até a de implementação e teste do SAMOA. No Capítulo 5 é exibido um estudo de caso. No 6 concluímos, comentando a importância deste trabalho, assim como propostas para trabalhos futuros a partir da germinação deste.

Capítulo 2 – Fundamentação Teórica

Neste presente capítulo abordamos as tecnologias utilizadas para o desenvolvimento do sistema proposto neste trabalho de pesquisa. O entendimento delas é essencial para uma melhor compreensão do nosso contexto. Também analisamos, no final co capítulo, os trabalhos relacionados à presente dissertação.

2.1. Modelagem Orientada a Objetos

Os engenheiros civis constroem vários tipos de modelos. Com maior freqüência, encontram-se modelos estruturais que ajudam as pessoas a visualizar e a especificar partes de sistema e os relacionamentos existentes entre essas partes. Dependendo de ser mais importante o interesse comercial ou a questão de engenharia, os engenheiros também poderão elaborar modelos dinâmicos – por exemplo, com a finalidade de ajudá-los a estudar o comportamento de determinada estrutura em relação a tremores de terra. Cada tipo de modelo é organizado de modo diferente e cada um tem seu próprio foco.

No caso de software, existem várias maneiras de se definir um modelo. As duas maneiras mais comuns são provenientes da perspectiva de um algoritmo ou da perspectiva orientada a objetos.

A visão tradicional do desenvolvimento de software adota a perspectiva de um algoritmo. Nessa visão, o principal bloco de construção do software é o procedimento ou a função. Essa perspectiva conduz os desenvolvedores a voltar

seu foco de atenção para questões referentes ao controle e à decomposição de algoritmos maiores em outros menores. Não existe nenhuma grande desvantagem nessa solução, com exceção da tendência a permitir sistemas instáveis. À medida que os requisitos se modificam (e isso certamente ocorrerá) e o sistema cresce (o que também acontecerá), será difícil fazer a manutenção de sistemas construídos a partir do foco em algoritmos.

A visão contemporânea do desenvolvimento de software adota uma perspectiva orientada a objetos. Nessa visão, o principal bloco de construção de todos os sistemas de software é o objeto ou a classe. Explicando de uma maneira simples, um objeto é alguma coisa geralmente estruturada a partir do vocabulário do espaço do problema ou do espaço da solução; uma classe é a descrição de um conjunto de objetos comuns. Todos os objetos têm uma identidade (você pode atribuir-lhes nomes ou diferenciá-los dos demais objetos de alguma maneira), um estado (costuma haver dados a eles associados) e um comportamento (você poderá fazer algo com o objeto ou ele poderá fazer algo com os outros objetos).

Por exemplo, considere uma arquitetura simples com três componentes para um sistema de cobrança, incluindo a interface para o usuário, você encontrará objetos concretos, como botões, menus e caixas de diálogo. No banco de dados, haverá objetos concretos como tabelas, que representam entidades provenientes do domínio do problema, incluindo clientes, produtos e pedidos. Na camada intermediária, você encontrará objetos como transações e regras de negócios, além de visões de alto nível relacionadas às entidades do problema, como clientes, produtos e pedidos.

O método orientado a objetos para o desenvolvimento de software é, com certeza, uma parte do "*main stream*", simplesmente porque tem sido provado seu valor para a construção de sistemas em todos os tipos de domínios de problemas, abrangendo todos os graus de tamanho e de complexidade. Além disso, muitas linguagens, sistemas operacionais e ferramentas contemporâneas são, de alguma forma, orientadas a objetos, fortalecendo a visão de mundo em termos de objetos. O desenvolvimento orientado a objetos fornece os fundamentos conceituais para a

montagem de sistemas a partir de componentes com a utilização de tecnologias como *Java Beans* ou *Enterprise Java Beans* [SunJava].

Várias conseqüências decorrem da escolha da visão de mundo orientada a objetos: o que é a estrutura de uma boa arquitetura de objetos? Quais artefatos o projeto deverá criar? Quem deverá criá-los? Como esses artefatos poderão der medidos? Como esses artefatos poderão se tornar reusáveis?

2.2. A linguagem UML

A Origem da *Unified Modeling Language* (UML) se deu a partir da fusão da metodologia de Grady Booch e da Metodologia OMT de James Rumbaugh, e posteriormente com a inclusão da metodologia OOSE de Ivar Jacobson. Grady Booch e James Rumbaugh juntaram forças através da Rational Corporation para forjar uma unificação completa de seus trabalhos. Em outubro de 1995, lançaram um rascunho do Método Unificado na versão 0.8, sendo esse o primeiro resultado concreto de seus esforços [Booch et al. - 2002].

Também em outubro de 1995, Ivar Jacobson juntou-se à equipe de unificação incluindo algumas idéias do método OOSE (Object-Oriented Software Engineering). Como autores, Booch, Rumbaugh e Jacobson estavam motivados em criar uma linguagem de modelagem unificada que tratasse assuntos referentes a sistemas complexos e de missão crítica, que se tornasse poderosa o suficiente para modelar qualquer tipo de aplicação de tempo real, cliente/servidor ou outros tipos de software padrões [Furlan - 1998].

A UML vai além de uma simples padronização em busca de uma notação unificada, uma vez que contém conceitos novos que não são encontrados em outras linguagens. A UML recebeu influência das técnicas de modelagem de dados (diagrama de entidade e relacionamento), modelagem de negócio (work flow), modelagem de objetos e componentes, e incorporou idéias de vários autores, dentre eles Peter Coad, Derek Coleman, Ward Cunningham, David Embley, Eric Gamma, David Harel, Richard Helm, Ralph Johnson, Stephen Mellor, Bertrand Meyer, Jim

Odell, Kenny Rubin, Sally Shlaer, John Vlissides, Paul Ward, Rebecca Wirfs-Brock, Ed Yourdon e Martin Fowler.

Buscou-se unificar as perspectivas entre os diversos tipos de sistemas e fases de desenvolvimento de forma que permitissem levar adiante determinados projetos, o que não era possível pelos métodos existentes.

2.2.1. Propósito de se trabalhar com a UML

A UML pode ser usada para [Rational et al. - 1997b]:

a) mostrar fronteiras de um sistema e suas funções principais utilizando atores e casos de uso;
b) ilustrar a realização de casos de uso com diagramas de interação;
c) representar a estrutura estática de um sistema utilizando diagramas de classe;
d) modelar o comportamento de objetos com diagramas de comportamento (diagrama de estado, diagrama de atividades, diagrama de seqüência e diagrama de colaboração);
e) revelar a arquitetura de implementação física com diagramas de implementação (diagrama de componentes e diagrama de distribuição);
f) estender sua funcionalidade através de estereótipos.

A UML consolida um conjunto de conceitos essenciais para modelagem visual que geralmente são utilizados por vários métodos atuais e ferramentas do mercado. Tais conceitos são encontrados em uma variedade de aplicações, embora nem todos sejam necessários nas diversas partes da aplicação, ou seja, a UML por oferecer uma linguagem de modelagem visual, apresenta três benefícios:

- **visualização**: os relacionamentos existentes entre os diversos componentes da aplicação podem ser visualizados de forma a antever o produto final;
- **gerenciamento da complexidade**: cada aspecto do sistema é desenhado à parte em um modelo específico para que se possa estudar e compreender a

estrutura, o comportamento e os possíveis particionamentos físicos, bem como identificar oportunidades de reutilização de componentes;

- **comunicação**: através da utilização de símbolos padrões torna-se possível uma comunicação direta e não ambígua entre os participantes do projeto com relação aos detalhes de comportamento do sistema.

A UML fornece mecanismos de extensibilidade e de especialização para apoiar conceitos essenciais.

Ela pode e deve apoiar linguagens de programação, bem como métodos e processos de modelos de construção. Pode ainda dar suporte a múltiplas linguagens de programação e métodos de desenvolvimento sem dificuldade excessiva.

2.2.2. Diagramas propostos pela UML

A UML possui em sua extensão 8 diagramas para modelagem de sistemas, tanto na fase de análise como na fase de projeto que são os diagramas de classe, Caso de Uso, Comportamento (Diagramas de Estado, Atividade, Interação (diagramas de Seqüência, Colaboração)), Implementação (diagramas de Componente e Distribuição).

2.2.2.1. Diagramas de Classe

O Diagrama de classes expressa de uma forma geral a estrutura estática de um sistema, contém classes e associações entre elas. Uma classe descreve um conjunto de elementos [Muller - 1997]. Uma associação é um conjunto de ligações. Objetos são instâncias das classes. O relacionamento entre as classes é chamado de associação e o relacionamento entre objetos é chamado de ligação.

2.2.2.2. Diagrama de Caso de Uso

Os diagramas de caso de uso fornecem um modo de descrever a visão externa do sistema e suas interações com o mundo exterior. Representa uma visão de alto nível de funcionalidade intencional, mediante o recebimento de um tipo de requisição do usuário, ou seja, eles permitem modelar o sistema de forma que se consiga ver como o sistema vai se comportar com a interação do mundo exterior, mediante a função (caso de uso) que o usuário possa executar, não sendo importante como funciona o sistema internamente [Furlan - 1998]; [Rational et al. - 1997a].

Os propósitos primários dos casos de uso são:

- descrever os requisitos funcionais do sistema de maneira que haja uma concordância de visão do sistema entre usuários e desenvolvedores;
- fornecer uma descrição consistente e clara sobre as responsabilidades que devem ser cumpridas pelo sistema, além de formar a base de como vai ser feita a interface do sistema;
- oferecer as possíveis situações do mundo real para o teste do sistema, ou seja, na fase de teste pode-se utilizar as situações de interação do usuário com o sistema que é modelado no diagrama de caso de uso e fazer teste com o sistema.

Um diagrama de caso de uso é um gráfico de atores, um conjunto de casos incluído por um limite de domínio, comunicação, participação e associações entre atores, assim como generalizações entre casos de uso.

O diagrama de caso de uso é formado basicamente por quatro elementos: ator, caso de uso, interação e sistema.

O propósito de um caso de uso é especificar um serviço que o sistema fornece a seus usuários, ou seja, um caso de uso é a função que uma classe pode desempenhar sem que se preocupe como essa função foi implementada internamente.

Um caso de uso é uma unidade coerente de funcionalidade fornecida por um sistema, manifestada por seqüências de mensagens trocadas entre o sistema e uma ou mais interações externas (chamadas de atores), junto com ações executadas pelo sistema.

2.2.2.3. Diagramas de Comportamento

São diagramas que representam a estrutura dinâmica ou comportamental de um sistema.

2.2.2.3.1. Diagrama de Estados

Diagramas de estados de David Harel foram uns avanços às tradicionais máquinas de estado planas. Uma máquina de estado finita pode ser pensada como uma caixa-preta que recebe um número finito de entradas (estímulos) e está preparada para tratá-las oferecendo respostas [Furlan - 1998].

Uma desvantagem do diagrama de estado é ter de definir todos os possíveis estados de um sistema, o que pode tornar a análise complexa e de difícil coordenação. A UML propõe o emprego do diagrama de estado de maneira individualizada para cada classe, com o objetivo de tornar o estudo simples o bastante para se ter um diagrama de estado compreensível.

O diagrama de estado foi introduzido por Rumbaugh, utilizado por Booch e adotado posteriormente na UML. Uma característica particularmente valiosa do enfoque é sua habilidade em generalizar estados, que permitem fatorar transições comuns. Modelos de estado são idéias para descrever o comportamento de um único objeto, mas não para descrever adequadamente o comportamento que envolve vários objetos, sendo melhor a utilização de diagramas de interação.

2.2.2.3.2. Diagrama de Atividade

Um diagrama de atividades é uma variação da máquina de estado no qual os estados são atividades representando o desempenho das operações e as transições são encadeadas por uma conclusão das operações. Ele representa uma máquina de estado, conforme apresentado anteriormente no diagrama de estado, de um procedimento dele mesmo; o procedimento é a implementação de uma operação na própria classe.

O propósito deste diagrama é dar enfoque ao fluxo dirigido pelo processamento interno (oposto aos eventos externos). Utiliza-se o diagrama de atividades em situações onde todos ou a maioria dos eventos representam o término das ações geradas internamente [Furlan - 1998]. Também são usados para detalhar eventos externos. Um evento externo surge quando o objeto está em estado de espera, durante o qual não há qualquer atividade interna no objeto permanecendo no aguardo de algum evento resultante de uma atividade de outro objeto. Pode haver mais de um evento que tire o objeto do estado de espera, sendo que o primeiro que acontecer dispara a transição [Rational et al. - 1997a].

Um diagrama de atividades é formado por estados de ações, decisões, raias (swimmlanes), itens de controle e relacionamentos. Um diagrama de atividades é utilizado para vários propósitos como:

- capturar o funcionamento interno de um objeto;
- capturar ações que serão desempenhadas quando uma operação é executada;
- mostrar como um processo de negócio funciona em termos de atores, fluxos de trabalho, organização e objetos;
- mostrar como uma instância de caso de uso pode ser realizada em termos de ações e mudanças de estado de objetos;
- mostrar como um conjunto de ações relacionadas pode ser executado e como afetará objetos ao redor.

2.2.2.3.3. Diagramas de Interação

Os diagramas de Interação são utilizados para representar a interação entre os objetos de um sistema com a finalidade de realizar uma certa tarefa.

2.2.2.3.3.1. Diagrama de Seqüência

O diagrama de seqüência mostra a interação entre objetos de um ponto de vista temporal. Diferente dos diagramas de colaboração, onde o contexto dos objetos não é representado explicitamente [Muller - 1997].

Em modelagem orientada a objetos, diagramas de seqüência são usados de duas diferentes maneiras, de acordo com a fase do ciclo de vida e o nível de detalhe que se quer obter.

Seu uso corresponde à documentação dos casos de uso que enfoca a descrição da interação entre os objetos, freqüentemente em termos que são transparentes para o usuário, sem levar em conta os detalhes da sincronização. Neste caso, a informação acarretada pelas setas, corresponde ao evento que ocorre dentro do domínio da aplicação. Até este estágio de modelagem, as setas não correspondem ainda a um *'broadcast* de mensagem' no sentido de linguagem de programação, e a diferença entre os fluxos de controle e os fluxos de dados não é geralmente estabilizado.

2.2.2.3.3.2. Diagrama de Colaboração

Diagramas de colaboração ilustram interações entre objetos, usando uma estrutura semelhante ao diagrama de classes, que facilita a ilustração da colaboração de um grupo de objetos, para realizar uma determinada tarefa. Diagramas de colaboração expressam ambos, o contexto de um grupo de objetos (através de objetos e ligações) e a interação entre esses objetos. Esses diagramas são uma extensão do diagrama de objetos [Muller - 1997].

O contexto de uma interação compreende os argumentos, as variáveis locais criadas durante a execução, e as ligações entre os objetos que participam na interação.

Uma interação é implementada por um grupo de objetos que colaboram pela troca de mensagens. Essas mensagens são representadas ao longo das

ligações, que conectam os objetos, usando uma seta apontando em direção ao recipiente da mensagem.

Diferente de um diagrama de seqüência, o tempo não aparece explicitamente no diagrama de colaboração, e como resultado as várias mensagens são numeradas para indicar a ordem de envio.

Diagramas de colaboração mostram as interações entre objetos e a estrutura de associações que facilita essas interações simultâneas.

2.2.2.3.4. Diagramas de Implementação

São utilizados para mostrar os aspectos de implementação.

2.2.2.3.4.1. Diagrama de Componentes

O diagrama de componentes descreve componentes de software e seus relacionamentos dentro do meio de implementação, ele indica a seleção feita no tempo de implementação [Muller - 1997].

Componentes

Os componentes representam todos tipos de elementos que pertencem à parte da aplicação do software. Entre outras coisas, eles podem ser simples arquivos, ou bibliotecas buscadas dinamicamente.

Dependências entre Componentes

Relacionamentos de dependência são usados dentro do diagrama de componentes para indicar que um componente refere-se a serviços oferecidos por outro componente. Este tipo de dependência reflete seleção de implementação. Um relacionamento de dependência é representado por uma seta tracejada desenhada do cliente ao supressor.

Em um diagrama de componente, relacionamentos de dependência geralmente representam dependência de compilação. A ordem de compilação é dada pelo gráfico de relacionamento de dependência.

2.2.2.3.4.2. Diagrama de Distribuição

Diagrama de distribuição mostra o layout físico dos vários componentes (nós) do hardware que compõe um sistema, bem como a distribuição dos programas executáveis neste hardware [Muller - 1997].

Cada recurso do hardware é representado por um cubo, evocando a presença física do equipamento dentro do sistema. Qualquer sistema pode ser descrito por um número pequeno de diagramas distribuídos, e um único diagrama é freqüentemente suficiente.

Diagramas de distribuição podem mostrar classes nós ou instâncias nós. Como outro tipo de diagrama, a diferença gráfica entre classes e objetos é implementada sublinhando o nome do objeto.

2.2.3. OCL – Object Constraint Language

OCL permite a especificação de três tipos de restrições: invariantes, os quais são estáticos, e pré e pós-condições, os quais são dinâmicos. Um invariante está associado a uma classe, uma interface ou um tipo. Ele especifica uma condição que deve ser verdadeira para todas as instancias de uma classe, interface ou tipo associado, em algum tempo. Em contraste, uma pré ou pós-condição está associada a uma operação UML e especifica uma condição que necessita ser verificada imediatamente antes ou depois de sua execução.

A Figura 2.1 ilustra um resumo de um diagrama de classe UML de um modelo de uma companhia. O diagrama inclui duas restrições OCL. A primeira, *managerConstraint* é um invariante que requer que todas as instancias da classe *Company* que constituem seu contexto a fim de ter sua associação *manager*

estabelecida com uma instancia da classe *Person* com o valor do atributo inteiro *age* dentro do intervalo e o atributo booleano *isUnemployed* contendo o valor *false*. O segundo, *resultOkConstraint* é uma pós-condição que requer o método *income* da classe *Person* que constitui seu contexto para sempre retornar um valor superior a 5000.

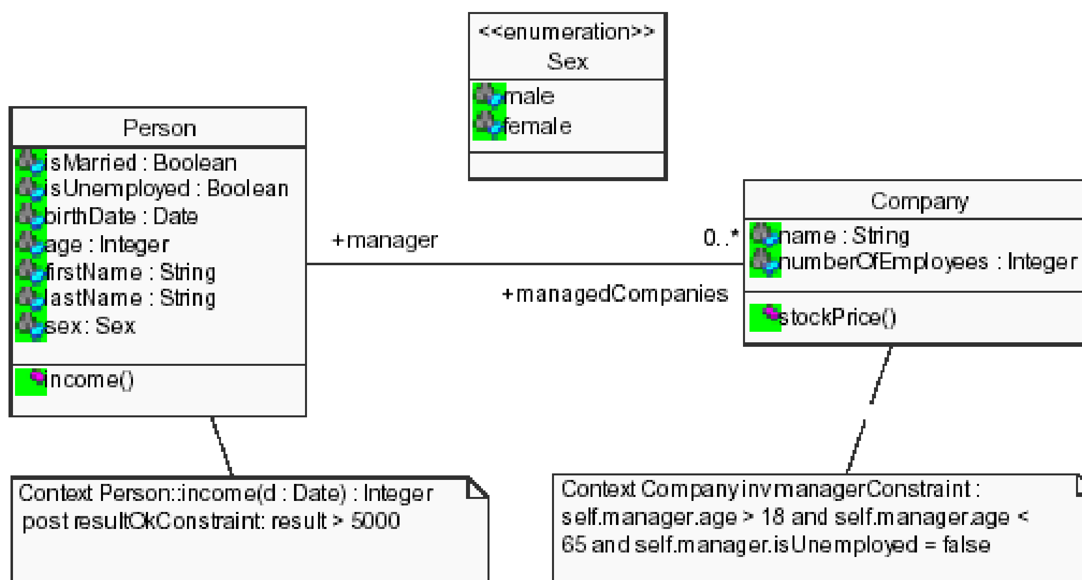

FIGURA 2.1: exemplo de um modelo UML com restrições OCL [Ramalho et al. - 2003]

Esses dois exemplos de restrições ilustram o argumento de projeto base do OCL: alcançar o balanceamento prático entre a precisão formal e intuitiva para atender as necessidades do usuário.

2.2.4. Conclusão

UML é uma linguagem de descrição de aplicações orientada a objetos, que pode ser utilizada para modelagem em vários paradigmas. Um de seus pontos fortes é o fato dela ser mais completa e mais amigável que as outras técnicas de modelagem de um modo geral, pois UML fornece vários diagramas para modelagem de sistemas, os quais apresentam um resultado passível de compreensão pelo usuário e programador, facilitando assim o desenvolvimento da análise, projeto e implementação do sistema.

2.3. O Padrão XMI

Existe um problema com as ferramentas de desenvolvimento orientado a objetos atuais: a troca de meta-dados entre as ferramentas ainda é uma tarefa muito difícil (suponha duas ferramentas CASE que precisem trocar informações baseadas nos modelos descritos em cada uma delas, teríamos uma incompatibilidade no formato dos dados. Isso tornaria difícil o compartilhamento/troca de tais informações). Não apenas as ferramentas codificam e armazenam os seus meta-dados de maneiras distintas, como os próprios esquemas conceituais que as ferramentas disponibilizam para a criação de modelos também são, na maior parte das vezes, bastante distintos. Então, o que temos, é um problema de interoperabilidade em dois níveis: no nível da codificação e no nível do esquema conceitual.

O XMI (XML Metadata Interchange) [XML – OMG]; [XMI - IBM] é um padrão para troca de modelos de sistemas orientados a objeto que tenta solucionar o problema de interoperabilidade nesses dois níveis através da definição de um padrão de codificação genérico, o XML [XML – W3C], e da definição de um padrão para os esquemas conceituais, chamado MOF [MOF - OMG].

O padrão XMI foi criado com o objetivo de permitir a interoperabilidade entre ferramentas CASE, repositórios de meta-dados e ferramentas de desenvolvimento, através da troca de meta-dados em um arquivo ou fluxo de dados baseado no padrão XML.

2.3.1. Características do XMI

O XMI é um padrão para codificação de meta-dados de ferramentas de desenvolvimento orientadas a objeto. No nível conceitual, o XMI é baseado em outro padrão da OMG chamado MOF. O MOF é um padrão para definição de interfaces de programação CORBA para repositórios de modelos, mas é também um padrão para

descrever meta-modelos. Em princípio, o MOF é genérico e rico o suficiente para ser capaz de descrever adequadamente qualquer meta-modelo orientado a objeto como, por exemplo, a UML. No entanto, embora voltado para a orientação a objeto, é possível descrever outros meta-modelos , como o entidade-relacionamento.

No nível da codificação, o XMI é baseado em XML. Mas o XMI não é uma linguagem XML. O XMI é uma especificação de como gerar linguagens XML adequadas para modelos de dados e como codificar esses meta-dados em um documento XML. Então, uma especificação XMI nada mais é do que um conjunto de regras que normalizam a geração de XML a partir de MOF. A especificação XMI compõe-se de dois conjunto de regras: um conjunto de regras de produção de DTDs XML e um conjunto de regras de produção de documentos XML. O 1º conjunto de regras ensina como derivar a gramática da linguagem XML correspondente ao meta-modelo. Os DTDs são documentos que descrevem a gramática de uma linguagem baseada em XML. Portanto, descrevem regras para a construção do documento XML correspondente a um modelo. Mas, devido a pouca expressividade da linguagem dos DTDs, é necessário um nível adicional de regras para guiar a geração dos documentos XML. Essa é a função do 2º conjunto de regras.

2.3.2. Os requisitos do projeto XMI

Para entender como o XMI veio a ter as características que possui, é importante saber quais requisitos foram levantados na fase inicial desse projeto e como a especificação final atendeu ou deixou de atender a esses requisitos.

O 1º requisito era que o XMI deveria servir para trocar meta-dados para qualquer meta-modelo MOF. O 2º requisito era que o XMI deveria definir como gerar a sintaxe de transferência para um modelo, baseado unicamente no meta-modelo. Ou seja, se existir uma especificação MOF para um meta-modelo, então o padrão deve permitir a geração automática de um mapeamento desse meta-modelo em XML. Ainda sobre o MOF, outro requisito do projeto XMI era que cada tipo de dados MOF deveria ser mapeado em um elemento distinto na DTD do XMI, quer seja esse elemento uma entidade ou um atributo. Outro requisito importante é que a utilização

do XMI deveria ser insuficiente para utilização efetiva dos meta-dados. Ou seja, o XMI deve especificar a sintaxe, mas a semântica é dada pelo MOF. O produtor e o consumidor do XMI precisam ter conhecimento do meta-modelo MOF para utilizá-lo efetivamente. Essa informação semântica sobre o meta-modelo não está incorporada na DTD gerada, mas deve ser de conhecimento das ferramentas. Isso se deve ao fato de a DTD ser bem mais simples que o meta-modelo MOF.

O XMI sozinho não é suficiente para a utilização dos meta-dados, mas um outro requisito importante é que XMI e MOF devem ser suficiente para recuperar todos os meta-dados. Outros requisitos também importantes:

- XMI deve permitir troca de fragmentos de modelos assim como de modelos completos.
- XMI não necessita que o modelo transmitido esteja completamente validado para que o metadado seja transmitido.
- XMI deve suportar versões de modelos.
- XMI deve permitir a codificação de extensões de meta-dados junto com os meta-dados do meta-modelo padrão.
- XMI deve ser usado para transmitir dados assim como meta-dados. As camadas modelo e meta-modelo podem ser suprimidas, mas o MOF deve permanecer como a única camada “meta”, descrevendo os dados, em vez de descrever o meta-modelo.
- XMI deve prover DTDs padrões para os meta-modelos MOF e UML. As DTDs estão localizadas no apêndice da especificação do XMI.

2.3.3. Mapeamento de modelos e meta-modelos MOF em documentos e DTDs XML

O mapeamento de meta-modelos MOF para DTDs e de modelos MOF para documentos XML é o cerne da especificação do XMI. Cada mapeamento está definido através de um conjunto de regras. Não é nossa intenção detalhar o mapeamento, por que isso a especificação já faz, mas dar uma noção geral de como são essas regras. A 1ª característica importante do mapeamento é definir para que

elementos da linguagem XML são mapeados os elementos do MOF. Uma classe MOF é sempre mapeada em uma tag XML. Um atributo MOF também é mapeado em uma tag XML. Cada associação é mapeada em 2 tags XML.

Uma DTD XMI deve conter os seguintes elementos:

- declarações XML obrigatórias;
- cabeçalho XMI;
- as declarações dos meta-dados de um meta-modelo específico;
- as declarações incrementais dos meta-dados de um modelo específico;
- declarações de extensões a meta-modelos.

2.3.3.1. Extensões do XMI para dados de ferramentas

Os meta-modelos existentes permitem definir características abstratas dos modelos, que independem da implementação. Entretanto, há meta-dados, que são específicos de cada ferramenta de desenvolvimento, e que não fazem e não devem fazer parte de um meta-modelo genérico. Por exemplo: uma ferramenta CASE precisa armazenar outras informações além das definições das classes, associações, atributos, e pacotes, como, por exemplo, as coordenadas das representações gráficas dos elementos do modelo. É para isso que servem as extensões do XMI.

Existem 2 tipos de extensões: extensões de uma classe, através de uma ou mais ocorrências da tag XMI.*extensions* dentro da declaração de uma classe e, extensões globais, através de uma ou mais ocorrências de XMI.*extensions* embaixo da declaração <XMI>.

2.3.4. Cenários de utilização do XMI

2.3.4.1. Interoperabilidade de meta-dados entre ferramentas

Não existe uma ferramenta única que suporte toda a necessidade de documentação e modelagem de sistemas de uma organização. Freqüentemente, as organizações precisam utilizar ferramentas diferentes para modelar cada uma das etapas de cada um dos seus processos. Fazer com que essas ferramentas troquem informações de modelagem é um grande desafio.

Para que a ferramenta A troque meta-dados de modelagem com a ferramenta B, é preciso que exista um componente na ferramenta A, que importe e exporte modelos de B, ou vice-versa. Esse módulo de importação e exportação, que chamaremos ponte de meta-dados, deve existir para cada par de ferramentas, para as quais seja necessário a troca de modelos. O XMI deverá ser utilizado como ponte universal entre ferramentas de desenvolvimento orientadas a objeto. Com a sua utilização será necessário apenas que cada ferramenta importe e exporte meta-dados em XMI, para que a troca de meta-dados entre todas as ferramentas seja possível.

2.3.4.2. Projetos de pesquisa que utilizam o XMI

Devido a sua adoção pela indústria ser recente, o padrão XMI ainda é utilizado em apenas alguns poucos projetos. Nas linhas abaixo, apresentamos alguns projetos que já estão utilizando XMI.

SPOOL

O projeto SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) [Keller et al. - 2000] é um projeto conjunto do grupo de engenharia de software da Universidade de Montreal [XML4SE] e da equipe de certificação de qualidade da Bell Canada. O objetivo do projeto é identificar e promover boas práticas de engenharia de software.

O projeto está desenvolvendo um ambiente para engenharia reversa de software chamado ambiente SPOOL. Esse ambiente compões-se de 4 componentes principais: ferramentas de gerência, visualização e análise de meta-dados, um

extrator de modelos a partir de código fonte C++, um importador de modelos, e um repositório de modelos, que armazena os modelos em um SGBD orientado a objeto, e que pressupões um meta-modelo UML. O importador de modelos transporta modelos, codificados no formato XMI, para dentro do SGBDOO.

Metadata Mediator

Esse projeto tem 3 objetivos principais [XML Schema]; [Ontogenics]: criar um mediador de meta-dados baseado na Web para converter modelos de e para XMI, criar uma representação XML independente de fabricante para representar regras de negócios e, criar extensões à UML para integrar regras de negócios na metodologia e no repositório do modelo.

2.3.5. Ferramentas que utilizam XMI

Argo/UML

O Argo/UML [ArgoUML - 2003] é uma ferramenta CASE gratuita, de fonte aberta, desenvolvida pela Universidade da Califórnia, que utiliza a linguagem de notação UML e está disponível para a plataforma Java. O Argo suporta 5 dos 8 diagramas da UML.

Os modelos são armazenados em 3 linguagens diferentes, baseadas em XML. O projeto é armazenado na linguagem ARGO. A linguagem PGML (Precision Graphics Markup Language), também baseada em XML, é utilizada para armazenar os objetos gráficos do projeto. A linguagem XMI é utilizada para armazenar os modelos e as extensões da ferramenta.

Poseidon

Poseidon para UML[Poseidon - 2003] é uma ferramenta de modelagem de sistemas da empresa alemã **Gentleware AG**. Trata-se de uma ferramenta completa de modelagem UML, evoluída a partir da ferramenta de código-aberto ArgoUML. Com mais de 350.000 instalações ela está entre as ferramentas de modelagem mais conhecidas. Seu principal foco está na usabilidade que a torna uma ferramenta simples de aprender e usar. Assim como no Argo/UML, a linguagem XMI também é utilizada para armazenar os modelos e as extensões da ferramenta.

Rational Rose

O Rational Rose [Rational et al. 1997a] é uma ferramenta CASE comercial para a linguagem de notação UML. A linguagem XMI não é a linguagem de armazenamento dos modelos.

Para importar e exportar modelos XMI, é preciso instalar um componente externo chamado Rose XMI Add-on [RoseXMI].

Unisys Universal Repository

O Unisys Universal Repository [Unisys];[UnisysXMI] é um repositório de meta-dados baseado no padrão MOF, que permite a importação e exportação de modelos baseados no padrão XMI.

XMIToolkit

É um software gratuito desenvolvido pela IBM [XMIToolkit] para a plataforma Java, que permite a importação e exportação de meta-dados em XMI de e para código fonte Java, código binário Java e o formato padrão de modelos do Rational Rose, o MDL. O XMIToolkit, por ter uma biblioteca de código fonte aberto, é uma boa escolha para incorporação do recurso de exportação e importação de formato XMI por ferramentas de desenvolvimento.

Outros

Além das ferramentas citadas acima, há notícias na Internet de que a ferramenta de desenvolvimento Visual Age for Java, o servidor de aplicações WebSphere, e o framework de componentes São Francisco utilizem XMI. No entanto, não encontramos nenhuma informação oficial a respeito.

2.3.6. A ausência de OCL no XMI

Então, observamos que o padrão XMI define textualmente elementos para todos os diagramas UML, mas até o presente momento, ele não define as restrições feitas em OCL nos diagramas. Porém, existe uma proposta de trabalho que propõe uma linguagem de marcação para especificações OCL, chamada de XOCL (linguagem baseada em XML para representar OCL). Podemos até visualizar como uma extensão ao XMI. O XOCL está sendo proposto por um projeto de doutorado em [Ramalho et al. - 2003].

2.4. Padrões de Projeto

Os padrões de projeto aplicam-se a, desde cidades, organizações, construções, até programas de computador. O estudo de padrões, sejam eles arquitetônicos ou relacionados ao desenvolvimento de software, mostra como é possível reutilizar idéias anteriores em novos projetos.

Nessa seção será mostrada a importância do uso de padrões de projeto na informática. Através de alguns exemplos de implementação destes padrões em *Java*, visa-se oferecer uma maneira de demonstrar uma aplicação prática e eficiente dos padrões de projetos mais utilizados.

2.4.1. Origem dos Padrões de Projeto

Nos idos da década de 70, o arquiteto Christopher Alexander buscava reconhecer quando um projeto arquitetônico era bom e por quê.

Através de observações, Alexander constatou que se a qualidade em um projeto é o objetivo, então seria possível quantificar o que torna um desenho bom ou ruim.

Alexander descobriu que diminuindo o foco, ou seja, procurando estruturas que resolvam problemas similares, ele pode discernir similaridades entre projetos de alta qualidade. Ele chamou essas similaridades de "padrões" [Shalloway et al. - 2000].

Observamos o mundo que nos rodeia e aprendemos suas estruturas abstraindo detalhes particulares, e documentando soluções análogas obtidas sobre condições diferentes.

Tais regras empíricas, representando regularidades de comportamento ou de estruturas, são chamadas "padrões" [Salingaros - 1999].

Alexander define um padrão como "a solução para um problema em um contexto".

"Cada padrão descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira." [Alexander - 1977].

Padrões visuais são a expressão mais simples do conceito de padrões [Salingaros - 1999]. Muitos padrões estão embutidos em nossas mentes: nós herdamos ações e relações que garantem nossa sobrevivência. Outros padrões devem ser aprendidos, e formam uma extensão artificial da mente humana. A habilidade para observar padrões nos dá a vantagem humana tanto de adaptar quanto mudar nosso ambiente. É claro, a complexidade que envolve um padrão em

cada configuração específica, deve ser parcialmente esclarecida para que possamos entender seus mecanismos básicos.

2.4.2. Que é um padrão de projeto?

De acordo com as observações de Cristopher Alexander, um padrão é composto das seguintes partes [Gamma et al. - 2000]:

O *nome* do padrão é uma referência que se pode usar para descrever um problema de projeto, suas soluções e conseqüências em uma ou duas palavras. Dar nome a um padrão aumenta imediatamente o vocabulário de projeto. Isso permite projetar em um nível mais alto de abstração. Ter um vocabulário para padrões permite-nos conversar sobre eles com nossos colegas, em nossa documentação e até com nós mesmos. O nome torna mais fácil pensar sobre projetos e a comunicá-los, bem como os custos e benefícios envolvidos, a outras pessoas.

O *problema* descreve quando aplicar o padrão. Ele explica o problema e seu contexto. Pode descrever problemas de projetos específicos. Algumas vezes, o problema incluirá uma lista de condições que deve ser satisfeita para caracterizar o padrão.

A *solução* descreve os elementos que compõem o projeto, seus relacionamentos, suas responsabilidades e colaborações. A solução não descreve um projeto concreto ou uma implementação em particular porque um padrão é como um gabarito que pode ser aplicado em muitas situações diferentes. Em vez disso, o padrão fornece uma descrição abstrata de um problema de projeto e de como um arranjo geral de elementos resolve o mesmo.

As conseqüências são os resultados e análises das vantagens e desvantagens da aplicação do padrão. Embora as conseqüências sejam raramente mencionadas quando se descreve decisões de projeto, elas são críticas para a avaliação de alternativas de projetos e para a compreensão dos custos e benefícios da aplicação do padrão. As conseqüências para o software freqüentemente

envolvem compromissos de espaço e tempo. Elas também podem abordar aspectos sobre linguagens e implementação.

2.4.3. Como padrões de projeto solucionam problemas de projeto

O foco principal do uso de padrões de projeto na informática é no campo de software orientado a objetos. De acordo com Gamma [Gamma et al. - 2000], a parte difícil sobre projeto orientado a objetos é a decomposição do sistema em objetos. A tarefa é difícil porque muitos fatores entram em jogo: encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução, reutilização, e assim por diante. Todos influenciam a decomposição, freqüentemente de formas conflitantes.

Muitos objetos num projeto provêem do modelo de análise. Porém, projetos orientados a objetos freqüentemente acabam tendo classes que não têm contrapartida no mundo real. As abstrações que surgem durante um projeto são as chaves para tornar um projeto flexível. Os padrões de projeto ajudam a identificar abstrações menos óbvias bem como os objetos que podem capturá-las. Por exemplo, objetos que representam processos ou algoritmos não ocorrem na natureza, no entanto, eles são uma parte crucial de projetos flexíveis. Esses objetos são raramente encontrados durante a análise ou mesmo durante os estágios iniciais de um projeto; eles são descobertos mais tarde, durante o processo de tornar um projeto mais flexível e reutilizável [Gamma et al. - 2000].

2.4.4. Como selecionar um padrão de projeto

A escolha de um padrão de projeto é mais subjetiva do que se imagina. Alan Shalloway [Shalloway et al. - 2000] apresenta como exemplo a discussão de dois carpinteiros sobre a melhor forma de se juntar a madeira para a construção de gavetas para um armário.

Dois carpinteiros estão discutindo sobre qual a melhor forma de se fazer as juntas para as gavetas de um armário. Cada um deles apresenta um tipo de solução: uma junta do tipo "esquadrilha" e uma junta do tipo "rabo de pomba".

A junta "esquadrilha" possui as seguintes características:

- pega-se as peças de madeira que se deseja juntar e cortam-se as pontas em 45 graus. Para juntá-las podemos usar pregos, parafusos ou cola;

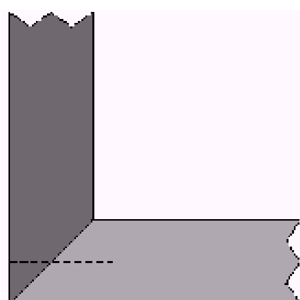

FIGURA 2.2: uma junta esquadrilha [Shalloway et al. - 2000]

A junta “rabo de pomba” é feita do seguinte modo:

- através de cortes consecutivos em 45 graus criam-se “dentes” os quais permitem o encaixe das peças.

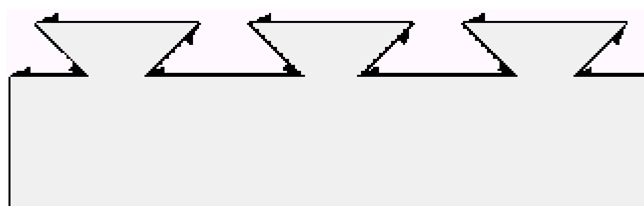

FIGURA 2.3 - Uma junta "rabo de pomba" [Shalloway et al. - 2000]

Quando o carpinteiro falou a respeito da junta "esquadrilha" ele tinha as seguintes características em mente:

- é uma solução de simples implementação, com custo reduzido. Porém é um dos tipos mais fracos de junção.

O outro carpinteiro, quando falou a respeito da junta “rabo de pomba” tinha em mente uma solução que possuía outras características:

- é uma solução mais evoluída, porém mais cara;
- é independente das alterações climáticas pois devido a configuração das juntas elas sofrerão a alteração conjuntamente, tornando, assim, a junta mais forte;
- não depende de cola ou qualquer outro tipo de mecanismo de união, porém é mais difícil de ser feita;
- é mais agradável esteticamente.

Qual é mais eficiente? Qual é melhor para se trabalhar?

Não se pode afirmar qual das soluções foi escolhida pelos carpinteiros, mas seja ela qual for, foi baseada em suas experiências próprias e disposição para o cumprimento da solução.

A seleção de um padrão de projeto é um dos passos mais difíceis para iniciar o uso de padrões para um projeto particular. Gamma [Gamma et al. - 2000] apresenta vários itens que devem ser observados quando se deseja escolher o padrão que melhor se encaixa ao problema:

- considerar como padrões de projeto solucionam problemas de projeto
- examinar qual a intenção do projeto, ou seja, o que faz o padrão de projeto, quais seus princípios e que tópico ou problema particular de projeto ele trata
- estudar como os padrões se inter-relacionam
- estudar padrões de finalidades semelhantes
- examinar uma causa de reformulação de projeto
- considerar o que deveria ser variável no seu projeto, ou seja, ao invés de considerar o que pode forçar uma mudança em um projeto, considerar o que você quer ser capaz de mudar sem reprojetá-lo.

2.4.5. Como usar um padrão de projeto

[Gamma et al. - 2000] recomenda seguir os seguintes passos quando for usar um padrão de projeto efetivamente:

1. leia o padrão por inteiro uma vez, para obter sua visão geral. Preste atenção em particular à aplicabilidade e conseqüências do padrão, para assegurar-se que ele é correto para seu problema;
2. volte e estude a estrutura, os participantes e as colaborações do padrão. Assegure-se que compreende as classes e objetos envolvidos e como se relacionam entre si;
3. olhe um exemplo de código para ver um exemplo concreto do padrão codificado. O estudo do código ajuda a aprender como implementar o padrão.
4. escolha nomes para os participantes do padrão que tenham sentido no contexto da aplicação. Os nomes para os participantes do projeto são, geralmente, muito abstratos para aparecerem diretamente em uma aplicação. No entanto, é útil incorporar o nome do participante no nome que aparecerá na aplicação. Isso ajudará a tornar o padrão mais explícito na implementação;
5. defina as classes. Declare suas interfaces, estabeleça seus relacionamentos de herança e defina as variáveis de instância que representam dados e referências a objetos. Identifique as classes existentes em sua aplicação que serão afetadas pelo padrão e modifique-as de acordo.
6. defina nomes específicos da aplicação para as operações no padrão. Os nomes em geral dependem da aplicação. Use as responsabilidades e colaborações associadas com cada operação como guia. Seja consistente, também, nas suas convenções de nomenclatura.
7. implemente as operações para se portar as responsabilidades e colaborações presentes no padrão.

É preciso salientar que os padrões de projetos não devem ser usados indiscriminadamente. Conforme [Gamma et al. - 2000], um padrão deverá apenas ser aplicado quando a flexibilidade que ele oferece é realmente necessária. Muitas vezes um projeto simples pode ser complicado caso seja adotado um padrão de projeto onde será necessária a adição de níveis de endereçamento indireto para

alcançar um certo grau de flexibilidade, aumentando o custo em termos de desempenho.

2.4.6. Padrões mais comuns

2.4.6.1. Padrões de Criação

Conforme [Gamma et al. - 2000], os padrões de criação abstraem o processo de instanciação. Eles ajudam a tornar um sistema independente de como seus objetos são criados, compostos e representados.

Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto que um padrão de criação de objeto delegará a instanciação para outro objeto.

Consequentemente, os padrões de criação dão muita flexibilidade no que é criado, quem cria, como e quando é criado. Eles permitem configurar um sistema com objetos "produto" que variam amplamente em estrutura e funcionalidade. A configuração pode ser estática (isto é, especificada em tempo de compilação) ou dinâmica (em tempo de execução).

Existem alguns tipos de padrões de criação que já foram definidos [Gamma et al. - 2000]. Entre eles destacam-se: *Abstract Factory, Builder, Factory Method, Singleton*.

2.4.6.1.1. Abstract Factory

Esse padrão fornece uma interface para a criação de famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas. Também conhecido como kit.

Heyworth [Heyworth - 1996] afirma que esse padrão é ideal quando se deseja isolar sua aplicação da implementação da classe concreta. A figura a seguir ilustra um exemplo, em Java, do uso deste padrão:

```
public abstract class Jardim
{
    public abstract Planta getCentro();
    public abstract Planta getBorda();
    public abstract Planta getSombra();
}
```

```
public class Planta{
   String nome;
   public Planta(String pnome) {
      nome = pnome; //salva nome }
   public String getName() {
      return name;  }
}
```

```
public class Horta extends Jardim {
   public Planta getSombra() {
      return new Planta("Brócolis");
   }
   public Planta getCenter() {
      return new Planta("Milho");
   }
   public Plant getBorder() {
      return new Planta("Ervilha");
   }
}
```

```
class Produtor
{
   //Abstract Factory o qual retorna um dos
  //três jardins
  private Jardim jd;
  public Jardim getJardim(String jtipo)
  {
   jd = new Horta(); //default
   if(jtipo.equals("Permanente"))
      jd = new JardimPermanente();
   if(gtipo.equals("Anual"))
      jd = new JardimAnual();
   return jd;
  }
}
```

FIGURA 2.4: Abstract Factory utilizado para o layout de um jardim

De acordo com [Heyworth - 1996], , a aplicação cliente instancia a Abstract Factory com uma classe concreta em tempo de execução e então usa a interface abstrata. Partes da aplicação cliente que usa a Factory não necessitam saber qual classe concreta está em uso atualmente.

Gamma [Gamma et al. - 2000] aconselha a usar o padrão Abstract Factory quando:

- um sistema deve ser independente de como seus produtos são criados, compostos ou representados;
- um sistema deve ser configurado como um produto de uma família de múltiplos produtos;

- uma família de objetos-produto for projetada para ser usada em conjunto, e você necessita garantir essa restrição;
- você quer fornecer uma biblioteca de classes de produtos e quer revelar somente suas interfaces, não suas implementações.

Gamma [Gamma et al. - 2000] indica que o padrão Abstract Factory possui as seguintes conseqüências:

- ele isola as classes concretas.
- ele torna fácil a troca de famílias de produtos.
- ela promove a harmonia entre produtos.
- é difícil de suportar novos tipos de produtos.

2.4.6.1.2. Builder

A função deste padrão é separar a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.

Heyworth [Heyworth - 1996] afirma que um Builder parece similar em conceito ao Abstract Factory. A diferença é que o Builder refere-se a simples objetos complexos (objetos que são construídos a partir da composição de outros objetos. Por exemplo, o objeto Carro é complexo, pois ele é composto dos objetos Pneu, Direção, Chassi etc.) de diferentes classes concretas, mas contendo múltiplas partes, enquanto o Abstract Factory permite criar famílias inteiras de classes concretas.

O uso do padrão Builder dá-se, quando:

- o algoritmo para criação de um objeto complexo deve ser independente das partes que compõem o objeto e de como elas são montadas;
- o processo de construção deve permitir diferentes representações para o objeto que é construído.

As conseqüências do uso deste padrão são [Gamma et al. - 2000]:

- permite variar a representação interna de um produto.

- isola o código para construção e representação.
- oferece um controle mais fino sobre o processo de construção.

2.4.6.1.3. Singleton

Esse método, segundo Gamma [Gamma et al. - 2000], garante que uma classe tenha somente uma instância e fornece um ponto global de acesso para a mesma.

Heyworth [Heyworth - 1996] afirma que esse é um dos métodos mais fáceis de ser implementados. Esse padrão é útil quando se deseja um objeto global simples na aplicação. Outros usos podem incluir um tratamento de erros global, segurança da aplicação ou um ponto simples de interface para outra aplicação.

```
class ISpooler
{
    //Este é um protótipo para um spooler para impressão
   //tal que apenas uma instancia poderá existir
   static boolean instancia_flag = false; //true se a instancia for 1
   //o construtor é privado
    private ISpooler() { }
    //o método estático Instance returna uma instancia ou null
   static public iSpooler Instance()
  {
     if (! instancia_flag)
    {
        instancia_flag = true;
        return new ISpooler();
     }
     else return null;
   }
   //-------------------------------------------
   public void finalize()
  {
       instancia_flag = false;
   }
}
```

FIGURA 2.5: exemplo em Java do padrão Singleton

Recomenda-se usar esse padrão quando [Gamma et al. - 2000]:

- deve haver apenas uma instância de uma classe, e essa instância deve dar acesso aos clientes através de um ponto bem conhecido;
- quando a única instância tiver de ser extensível através de subclasses, possibilitando os clientes de usarem uma instância estendida sem alterar o seu código.
-

O padrão Singleton apresenta vários benefícios [Gamma et al. - 2000]:

- acesso controlado à instância única;
- espaço de nomes reduzido;
- permite um refinamento de operações e da representação;
- mais flexível do que operações de classe.

2.4.6.2. Padrões estruturais

Os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores. Os padrões estruturais de classes utilizam a herança para compor interfaces ou implementações [Gamma et al. - 2000].

Em lugar de compor interfaces ou implementações, os padrões estruturais de objetos descrevem maneiras de compor objetos para obter novas funcionalidades. A flexibilidade obtida pela composição de objetos provê da capacidade de mudar a composição em tempo de execução, o que é impossível com a composição estática de classes [Gamma et al. - 2000].

Existem alguns tipos de padrões estruturais que já foram definidos [Gamma et al. - 2000]. Entre eles destacam-se: *Adapter*.

2.4.6.2.1. Adapter

A função desse padrão é converter a interface de uma classe em outra interface, esperada pelos clientes. O Adapter permite que classes com interfaces

incompatíveis trabalhem em conjunto – o que, de outra forma, seria impossível. É também conhecido como Wrapper [Gamma et al. - 2000].

Um caso típico é quando se deseja construir uma interface única para velhos ou novos sistemas.

Deve-se usar o padrão Adapter quando [Gamma et al. - 2000]:

- se quiser usar uma classe existente mas sua interface não corresponde à interface que necessita;
- se quiser criar uma classe reutilizável que coopera com classes não-relacionadas ou não-previstas, ou seja, classes que não necessariamente tenham interfaces compatíveis;
- se necessitar usar várias subclasses existentes, porém, for impraticável adaptar estas interfaces criando subclasses para cada uma.

Dentre as conseqüências do uso deste padrão, indicadas por [Gamma et al. - 2000], destaca-se que um objeto adaptado não oferece a interface do objeto original, por isso ele não pode ser usado onde o original o for. Adaptadores de dois sentidos podem fornecer esta transparência. Eles são úteis quando dois clientes necessitam ver um objeto de forma diferente.

2.4.6.3. Padrões Comportamentais

Os padrões comportamentais se preocupam com algoritmos e a atribuição de responsabilidades entre objetos. Os padrões comportamentais não descrevem apenas padrões de objetos ou classes, mas também os padrões de comunicação entre eles. Estes padrões caracterizam fluxos de controle difíceis de seguir em tempo de execução. Eles afastam o foco do fluxo de controle para permitir a concentração somente na maneira como os objetos são interconectados [Gamma et al. - 2000].

Os padrões comportamentais de classe utilizam uma herança para distribuir o comportamento entre classes. Os padrões comportamentais de objeto utilizam a composição de objetos em vez da herança. O *Template Method* é o mais simples e o mais comum dos padrões comportamentais.

2.4.6.3.1. Template Method

Um Template Method é uma definição abstrata de um algoritmo. Ele define um algoritmo passo a passo. Cada passo invoca uma operação abstrata ou uma operação primitiva. A intenção do uso do Template Method é definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para subclasses. Ele permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo [Gamma et al. - 2000].

O padrão Template Method pode ser usado, segundo Gamma [Gamma et al. - 2000]:

- para implementar as partes invariantes de um algoritmo uma só vez e deixar para as subclasses a implementação do comportamento que pode variar;
- quando o comportamento comum entre subclasses deve ser fatorado e concentrado numa classe comum para evitar a duplicação de código;
- para controlar extensões de subclasses.

2.5. Assistentes Automatizados

Assistentes automatizados são considerados ferramentas valiosas na engenharia de software. Um número considerável de assistentes automatizados tem sido desenvolvido para auxiliar o engenheiro em muitas fases do ciclo de produção de software. Esses assistentes podem ser classificados entre sistemas generativos ou de críticas. Os sistemas generativos ajudam os engenheiros a produzir novos artefatos usando técnicas generativas, ou seja, técnicas que geram soluções de um

template de soluções gerais. Já os sistemas de críticas provêem críticas sobre artefatos existentes para melhorar a sua realização, baseando-se na analise do que esta sendo desenvolvido e sugerindo melhorias baseadas em regras já predefinidas.

Padrões de projeto podem ser usados para descrever um sistema de software complexo em termos de uma abstração em mais alto nível do que classes, objetos e mensagens. Descrevendo situações típicas de componentes de soluções para sistemas de software. Contudo, pouquíssimos assistentes automatizados empregam padrões como abstrações básicas para projeto e engenharia reversa, embora esses padrões derivem de uma experiência concreta. Num projeto, um engenheiro de software pode encontrar uma solução de projeto similar a um padrão sem notar a semelhança. Também, apenas a experiência pode encaminhar o engenheiro ao uso de padrões, e regras aonde utilizar um padrão particular não são, geralmente, triviais. Essas características sugerem que os engenheiros possam se beneficiar de um assistente automatizado capaz de criticar seus projetos em relação ao uso de padrões de projeto.

Padrões podem ser explorados tanto por sistemas generativos quanto por de críticas, pois os padrões representam soluções para um propósito geral.

O mais famoso assistente automatizado para o engenheiro de software é o *Programmer's Apprendice* [Rich - 1990] que foi desenvolvido no Laboratório de Inteligência Artificial do MIT por Rich e Wanters. Esse sistema faz uso de um conhecimento base sobre projeto e implementação de software a fim de detectar erros cometidos por programadores ou selecionar automaticamente a implementação escolhida. Este conhecimento base é expresso em termos de *clichès, isto é, "usando combinações de elementos com nomes familiares"* [Rich - 1990]. A diferença primária entre um *clichè* e um padrão está no nível de abstração. Um *clichè* pode representar um fragmento de algoritmo ou um tipo de dado abstrato, mas não diz nada a respeito sobre interação entre objetos. Esta é a razão pela qual *clichès* podem ser usados para descrever a implementação de um objeto mas não podem representar a arquitetura de vários outros objetos.

A ferramenta CASE POSEIDON [Poseidon – 2003] traz consigo um assistente automatizado. POSEIDON suporta diagramas UML e provê ao

engenheiro críticas sobre os modelos em construção. Estas críticas vão de uma convenção de nomes a sugestões sobre possíveis melhorias nos projetos. Essa ferramenta seleciona críticas pesquisando estruturas particulares em um modelo UML corrente. As críticas selecionadas são, então, propostas ao engenheiro em uma lista - *"to-do list"*. Contudo, o POSEIDON não incorpora um conhecimento base sobre padrões de projeto. Como um exemplo, essa ferramenta verifica classes sem métodos ou sugere ao engenheiro a adotar nomes, estritamente relacionados ao domínio da aplicação, evitando nomes tais como *adapter* ou *proxy*.

Padrões e estruturas similares são largamente utilizados em ferramentas de engenharia reversa para extrair um alto nível de conhecimento de um código fonte. Uma ferramenta capaz de explorar estas informações foi desenvolvida por [Meijers - 1997], o qual extrai informações a partir de fontes *Java*, o *PatternsBox* [PatternsBox]. Discutiremos mais sobre ele no capítulo 3.

2.6. Meta-modelagem e Padrões

Técnicas baseadas em meta-modelagem consistem em definir um conjunto de meta-entidades das quais a descrição de um padrão é obtida pela composição dessas meta-entidades. Esta composição segue regras semânticas, fixadas pelos relacionamentos entre meta-entidades. Deste ponto de vista, meta-modelagem é um meio significativo para formalizar padrões.

Um meta-modelo para um padrão não captura o que um padrão é no geral, mas como ele é utilizado em um ou mais casos específicos, por exemplo, aplicação, representação estrutural, etc. Cada tipo de uso implica a definição de um meta-modelo dedicado. Por exemplo, um meta-modelo baseado em fragmentos para representar a estrutura de padrões [Meijers - 1997] ou baseado em meta-entidades para instanciação e validação [Sunyé - 1999].

Um meta-modelo padrão nunca produz padrões. Ao invés disto, ele produz modelos de padrões. Estes resultantes modelos são aproximações de padrões no caso de uso considerado. Existem vários meta-modelos, por exemplo em [Winter et

al. - 1996] é introduzido um meta-modelo para instanciação e validação, mas sem suporte a geração de código. Em [Sunyé - 1999], o meta-modelo não suporta geração de código e não oferece detecção de padrões. Em [Meijers - 1997], o sistema proposto, baseado em fragmentos, permite a geração de código e a detecção de padrões. Como os objetivos desta pesquisa estão refletidos nessa proposta de Meijers, adotamos o meta-modelo dele para o projeto do *SAMOA*. Daqui por diante, quando mencionarmos o meta-modelo, estaremos nos referindo ao proposto por Meijers.

O meta-modelo incorpora um conjunto de entidades e regras para interações entre elas. Todas as entidades precisam descrever a estrutura dos padrões introduzidos em [Gamma et al. - 2000]. Um modelo de padrão consiste em uma coleção de entidades, representando a noção de participantes como definido em [Gamma et al. - 2000]. Cada entidade contém uma coleção de elementos, representando os diferentes relacionamentos entre entidades. Se necessário, novas entidades ou elementos podem ser adicionados pela especialização das classes originais. Veremos mais detalhes desse meta-modelo no Capítulo 3.

2.7. Framework PatternsBox

Desenvolvido por Hervé Albin-Almiot [PatternsBox], na École des Mines de Nantes na França, com o intuito de implementar o meta-modelo proposto por [Meijers - 1997], visando a detecção de padrões em classes Java [SunJava]. O framework não utiliza nenhum sistema de marcações e não requer informação adicional a ser embutida no código do usuário, para tornar possível a detecção de padrões. Com isso, o usuário não terá que se preocupar em inserir nenhuma marcação no seu código para que seja encontrado algum padrão nele. Assim fica mais fácil adicionar novos padrões que possam ser detectados nos fontes dos usuários.

Seu funcionamento é baseado na informação estrutural das classes, utilizando um repositório de todos os constituintes de um padrão (elementos e

entidades). Esse repositório, instância da classe *TypesRepository* contém todos os *PEntities* e *PElements* definidos no meta-modelo [Meijers - 1997].

A detecção é decomposta em dois passos:

(1) a classe *PatternIntrospector*, encarregada de realizar a detecção, submete todos os elementos sintáticos (classes, interfaces, métodos...) encontrados no código fonte do usuário e os armazena no *TypesRepository*.

Cada tipo é submetido ao método que detecta os seus constituintes. Esse método foi chamado de “reconhecer*”* – *recognize*. A Figura 2.6, [Amiot et al. - 2001], ilustra o algoritmo do método para tal reconhecimento. Esses algoritmos foram propostos por [Meijers - 1997] e implementados no *PatternsBox*.

Reconstruction algorithm, class `PatternIntrospector`

Let `C`: list of the user's classes
Let `P`: pattern being recognized
Let `E`: list of existing entities (`Pentity`)
Let `L`: list of existing elements (`PElement`)
Let `S`: list of the syntactic elements (classes, interfaces…) found on the user code.
Let `T`: list of instanced of `Pelement`
Let `U`: class being examined

```
S = C
T = ∅
For each e of E, while size(S) > 0
        S = e.recognize(S, P)
T = P.listPEntities()
For each t of T
        U = Class.forName(t.getName())
        S = U.getDeclaredConstructors() + U.getDeclaredMethods() + U.getDeclaredFields()
        For each l of L, while size(S) > 0
                S = l.recognize(S, P)
```

Method `recognize` of `Pentity` (`e.recognize(S,P)`)

Let `N`: list of non-recognized entities

```
N = S
For each s of S
    If s = e Then
        P.addPEntity(new(s))
        N = N - {s}
Return N
```

Method `recognize` of `Pelement` (`l.recognize(S,P)`)

Let `N`: list of non-recognized elements

```
N = S
For each s de S
    If s = l Then
        P.getActor(s.getDeclaringClass().
                getName()).addPElement(new(s))
        N = N - {s}
Return N
```

FIGURA 2.6: algoritmos para detecção de padrões e para reconhecimento das entidades de cada tipo

O *PatternIntrospector* constrói um modelo concreto que representa o código do usuário contendo apenas os constituintes definidos no meta-modelo. Finalmente, ele solicita cada modelo abstrato no *PatternsRepository* para

determinar quais padrões foram detectados. Os padrões detectados são então retornados em uma lista na qual o usuário deverá iterá-la para visualizar o resultado.

(2) Cada modelo abstrato é examinado e a determinação de quais entidades (instâncias de *PEntity*) do modelo submetido podem se associadas a diferentes papéis. Os seguintes critérios devem ser considerados:

- as classes dos usuários devem conter (no mínimo) tantas entidades possíveis quanto o modelo abstrato;
- para cada papel atribuído do modelo abstrato, as entidades correspondentes no modelo sendo construído devem conter entidades as quais existem no modelo abstrato;
- tanto os relacionamentos de herança, associação e realização devem ser representados.

2.8. Trabalhos Relacionados

Pouquíssimos trabalhos têm sido realizados no campo da detecção automática de padrões. A maioria deles concentram seus esforços em mecanismo de reengenharia.

Em [Keller et al. - 1999] está descrito uma analise estática para descobrir padrões de sistemas escritos em C++. O sistema PAT - Program Analysis Tool – [Prechelt et al. - 1998] detecta padrões estruturais pela extração de informações de projeto nos cabeçalhos de arquivos C++ e os armazena em fatos Prolog. Os padrões são armazenados como regras e a pesquisa é feita pela execução de consultas Prolog. Esta técnica não é muito vantajosa pois força o programador a inserir marcações em seus programas.

Já em [Brown - 1997] a detecção de padrões está restrita a sistemas escritos em Smalltalk. Nele o processo detecção está "amarrado" a programas escritos nessa linguagem.

Um meta-modelo para representar padrões de projeto num modelo OMT foi proposto por [Meijers - 1997]. Já [PatternsBox] realizou a implementação, deste modelo em Java [SunJava], assim como os algoritmos de detecção de padrões, também propostos por [Meijers - 1997].

Existe outro meta-modelo para representar padrões, é o *PatternGen Tool* [Sunyé - 1999], mas ele não é capaz de dar suporte a geração de código nem à detecção de padrões.

O sistema que propomos aqui, o SAMOA, se destaca por poder realizar a detecção de padrões em diagramas de classe UML. O projetista ao construir seus diagramas poderá constatar a existência de um padrão, sem que para isto ele precise programar algumas linhas de código (caso das ferramentas mencionadas em linhas anteriores) e nem adicionar alguma informação a mais em seus diagramas. Outra peculiaridade do SAMOA é a de poder instanciar um padrão para um específico contexto do usuário e gerar seu código fonte, equivalente, na linguagem de programação Java.

2.9. Conclusão

Utilizamos este capítulo para descrever as tecnologias relacionadas direta e indiretamente envolvidas no nosso trabalho, dentro do escopo de padrões de projeto.

Antes de chegarmos ao projeto e conseqüentemente conclusão da ferramenta SAMOA, vários aspectos imprescindíveis ligados a padrões de projetos foram analisados. O embasamento teórico, descrito de forma bem sucinta nesse capítulo, foi indispensável para que pudéssemos traçar os caminhos que deveríamos seguir nas etapas posteriores (projeto e implementação) até a conclusão do nosso objetivo com êxito.

Capítulo 3 – SAMOA - Sistema de Apoio à Modelagem Orientada a Objetos de Aplicações

O sistema, a ser descrito neste capítulo, é um assistente automatizado na WEB para auxiliar programadores e projetistas de software, com relação ao uso de padrões de projeto. Esse auxílio consiste no processo de detecção da realização de padrões (tanto em diagramas de classe exportadas no formato XMI quanto em código fonte *Java*) assim como a instanciação dos mesmos. Se a informação necessária para detectar um padrão puder ser obtida dos diagramas ou fontes *Java*, chamamos estes padrões de "detectáveis". Alguns padrões estão fora dessa categoria, visto que eles não podem ser completamente definidos em termos de classes, objetos e interações, chamamos estes padrões de "não detectáveis". Este é o caso, por exemplo, dos padrões *Façade* e *Interpreter* [Gamma et al. - 2000]. Estes padrões são caracterizados por definir seus papéis tendo em vista o entendimento do projeto como um todo e não por suas estruturas isoladas, portanto o processo de detecção desses padrões necessitaria de uma compreensão geral de todo o modelo. Faz-se necessário um melhor aprofundamento do entendimento dos diagramas e dos fontes *Java*. Aqui, nos detemos aos padrões que classificamos por serem detectáveis.

O SAMOA também é composto de um sistema de geração de críticas que visa trabalhar em uma interação direta com seus usuários propondo críticas a padrões específicos. Veremos, nas seções posteriores que, uma vez detectada a realização de um padrão, o sistema listará críticas sugerindo melhorias no uso desses padrões.

A idéia de termos um sistema dessa na natureza na WEB é pensando, também, nas equipes de software que se encontram distribuídas geograficamente, mas que possam gozar de uma ferramenta disponível em um ambiente público, no caso, nada melhor do que a WEB para tal objetivo. Mas a proposta do SAMOA vai além de ser apenas um simples sistema WEB. Uma vez que uma das entradas do SAMOA consiste em um arquivo no padrão XMI e tendo como um das saídas, um arquivo XML de padrões detectados (no caso do processo de detecção). Com essa interoperabilidade, o SAMOA tem uma idéia avançada de engajar-se em um sistema de *WEB Services*. O sistema poderá comunicar-se não só apenas com uma ferramenta CASE mono-usuário, mas sim com uma outra ferramenta CASE WEB. A figura a seguir demonstra a forma de interoperação entre ferramentas CASEs e o SAMOA, representando a comunicação entre os sistemas no intuito de detecção de padrões em diagramas UML.

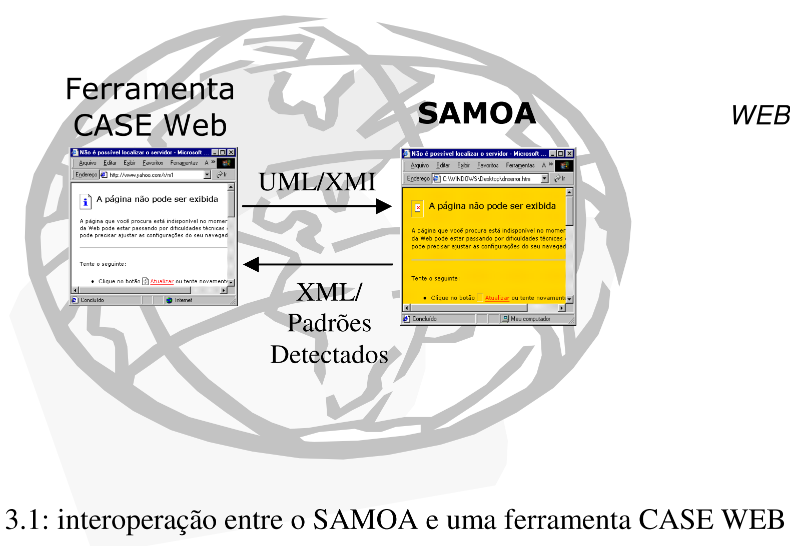

FIGURA 3.1: interoperação entre o SAMOA e uma ferramenta CASE WEB

O usuário edita diagramas de classe UML e a ferramenta a converte para o formato XMI, entregando este ao SAMOA. O SAMOA efetua a detecção de padrões no UML/XMI e retorna o resultado da detecção em formato XML (os detalhes deste processo encontram-se em seções posteriores).

Neste capítulo nos dedicamos a descrever, detalhadamente, as funcionalidades do SAMOA.

3.1. O meta-modelo de Meijers

O meta-modelo proposto por [Meijers - 1997] incorpora um conjunto de entidades e associações entre elas. Todas as entidades necessárias para descrever as estruturas dos padrões de projeto introduzidas em [Gamma et al. - 2000] estão representadas. A Figura 3.2 mostra um fragmento deste meta-modelo.

Um modelo de padrão é tratado como uma instancia da subclasse da classe *Pattern*. Ela consiste de uma coleção de entidades, classes ou interfaces, (instancias de *PEntity*), representando a noção de participantes como definido em [Gamma et al. - 2000]. Cada entidade (*PEntity*) contém uma coleção de elementos (*PElement*), representando os diferentes relacionamentos entre entidades. Se necessário, novas entidades ou elementos podem ser adicionados pela especialização das classes *PEntity* ou *PElement*.

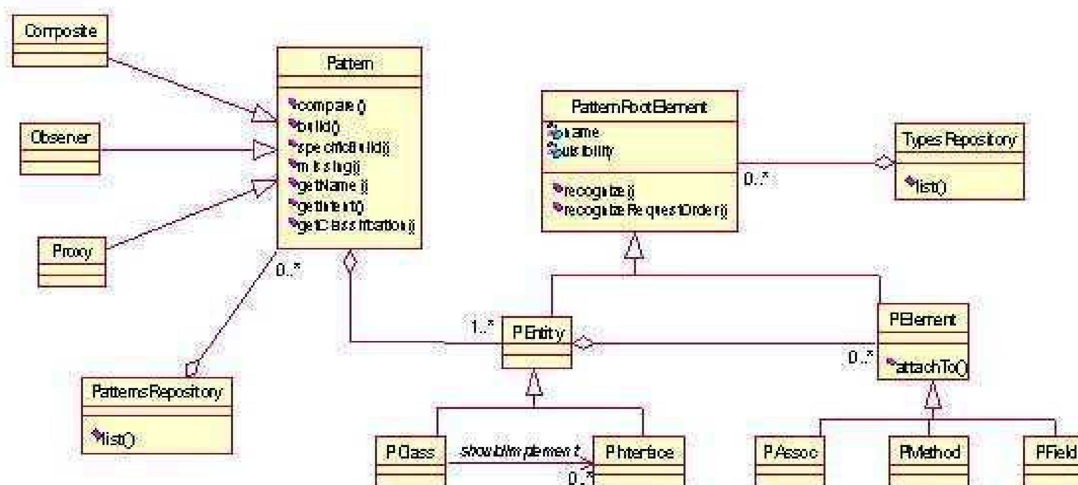

FIGURA 3.2: diagrama de classe UML simplificado do meta-modelo de Meijers

O meta-modelo define a semântica dos padrões. Um padrão é composto de uma ou mais classes ou interfaces, instancias de *PClass* e *PInterface* que são subclasses de *PEntity*. Uma instancia de *PEntity* contém métodos e atributos, instâncias de *PMethod* e *PField*, respectivamente.

Uma associação (classe *PAssoc*) representa um relacionamento entre entidades (é uma simplificação que representa simples relacionamentos como associações mono-direcionais e binárias encontradas em [Gamma et al. - 2000]). Por exemplo, uma associação que liga uma classe *B* a uma classe *A* é definida usando duas instancias da classe *PClass*, *A* e *B*, e uma instância da classe *PAssoc*. A instância da classe *PAssoc* de *A* e para *B*.

Adotamos este meta-modelo devido ao fato de ter sido elaborado visando apenas os elementos estruturais dos padrões do [Gamma et al. - 2000], e por sua forma fragmentada (que proporciona facilidade para geração de código) que é um dos nossos objetivos.

3.2. Arquitetura do SAMOA

Para termos o SAMOA funcionando na WEB, (um dos requisitos deste trabalho), projetamo-lo de modo a produzir e comportar-se em uma arquitetura em multicamadas. Adotamos uma arquitetura em três camadas. Aplicamos a presença de um Servidor de Aplicativos. Ele gerencia o reaproveitamento de recursos e a conectividade - tanto com os repositórios de dados como com a camada de aplicação. Sua presença, entretanto, cria dois importantes fatores a considerar: aumento de custo e complexidade do desenvolvimento.

Propomos, então, a seguinte arquitetura para o SAMOA:

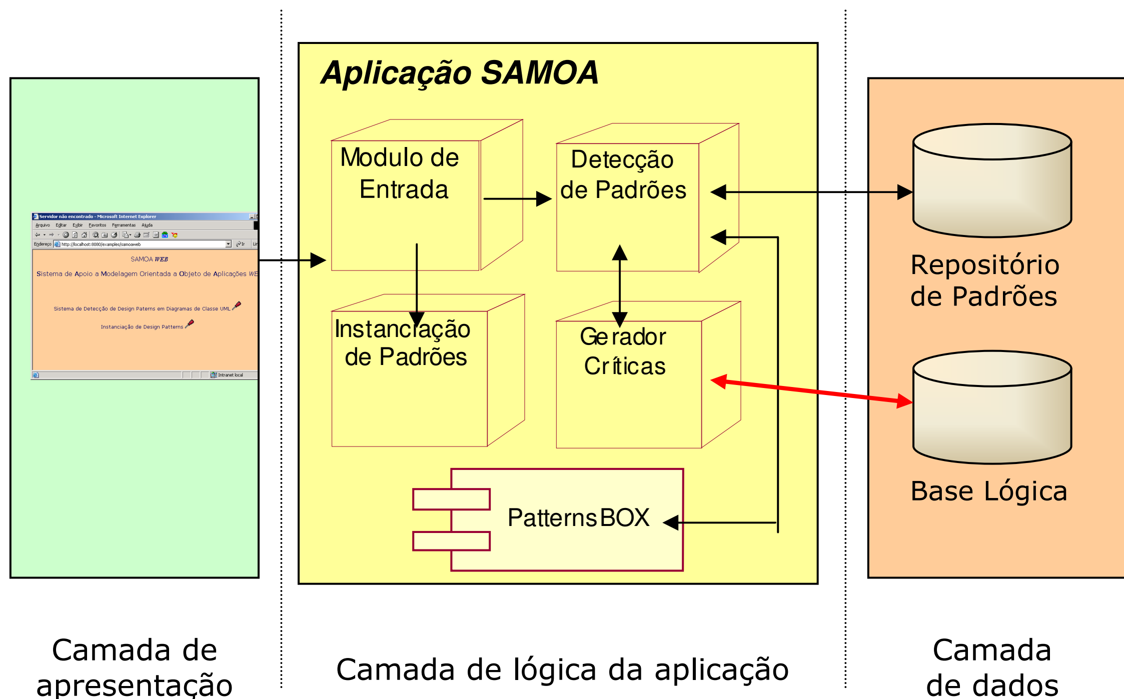

FIGURA 3.3: arquitetura do SAMOA

Discutiremos agora detalhes dessas camadas e seus constituintes.

3.2.1. Camada de Apresentação

Esta camada é representada pelo navegador e pelo servidor WEB. Ela organiza e exibe as informações para o usuário. Recebe requisições (uma instanciação ou detecção de padrões) e organiza a forma de entrada e saída (um link para os fontes instanciados ou XML dos padrões detectados).

3.2.2. Camada de Dados

Representada pelo servidor e pelas abstrações dos dados (no nosso caso, padrões descritos em arquivos no formato texto, seguindo o meta-modelo de [Meijers1997]. Acesso a programas legados e sistema de arquivos. Somente pode ser acessada pela camada de lógica de aplicação, o que aumenta a segurança do sistema. É nesta camada que se encontram armazenados os padrões para efeitos de detecção e instanciação dos mesmos. E também, a base

lógica de conhecimento, base esta que contém críticas, pré-definidas, em relação a alguns padrões.

3.2.3. Camada de lógica da aplicação

Representada por um conjunto de módulos que realizam a lógica de negócios do aplicativo; esses objetos ficam hospedados num **Servidor de Aplicativos**. Pode ser reusada por várias formas de apresentação diferentes, e se comunica com a camada de serviços de dados para prover informações e acionar operações.

A camada de lógica é composta por quatro módulos:

- Módulo de entrada

Recupera um corrente diagrama de classes UML, editado em uma ferramenta CASE e exportado no formato XMI. Com isso, o sistema fica independente de qualquer outra ferramenta, e o uso de XMI irá prover uma interoperabilidade entre o SAMOA e qualquer editor de diagramas UML.

Alternativamente, em um processo de engenharia reversa, o SAMOA poderá receber, como entrada, arquivos fontes *Java*. Com isso o sistema irá inferir conhecimento desses códigos para descobrir a realização de padrões. A seção posterior, elucidará todo o processo de detecção para ambas as entradas mencionadas em linhas anteriores.

- Detecção da realização de padrões

Módulo responsável pela detecção de padrões tanto em classes *Java* quanto em diagramas UML/XMI. Existe um framework chamado *PatternsBox* [PatternsBox], que implementou todo o meta-modelo proposto por [Meijers - 1997] e, também, o algoritmo de detecção de padrões. Esse framework foi

integrado ao SAMOA, em particular na detecção de padrões em fontes Java. Na próxima seção discutiremos detalhes do *PatternsBox*.

- Geração de críticas sobre a aplicação dos padrões encontrados

Uma vez que os padrões foram detectados, inicia-se um processo de inferência de conhecimento sobre tais padrões, visando estabelecer críticas sobre a má utilização dos padrões e provendo sugestões para sua melhoria. Um conjunto de críticas associado aos padrões, deverá estar armazenado em uma base lógica de conhecimento, tendo suas regras descritas em uma linguagem lógica orientada a objetos. A necessidade de uma linguagem deste tipo, consiste em aproveitar a definição dos conceitos de orientação a objetos já definidos nesse tipo de linguagem. Uma linguagem de descrição lógica que possui recursos para tal finalidade é a linguagem *F-Logic* (formalismo utilizado para representar entidades em uma linguagem lógica orientada a objetos) [Kifer et al. - 1995].

As críticas sugerem:

- nomes para classes, atributos e operações: por exemplo, o nome de um *factory method* no padrão *Factory Method* [Gamma et al. - 2000] deverá terminar com o sufixo “Factory”;
- o escopo para operações: por exemplo, métodos *hook* no padrão *Template Method* [Gamma et al. - 2000] devem ser declarados *protected* para acessá-los apenas via métodos *template*;
- operações que são prováveis de serem perigosas para reuso, tais como prover um acesso direto ao *subject* de um objeto *proxy* no padrão *Proxy* [Gamma et al. - 2000];
- técnicas que podem ser usadas para resolver problemas de projetos: por exemplo, o padrão *Iterator* [Gamma et al. - 2000] pode ser usado para acessar componentes de um objeto *composite* no padrão *Composite* [Gamma et al. - 2000].

As regras para realizar essas críticas formarão o conjunto de conhecimento base do SAMOA.

- Instanciação e geração de código

O SAMOA não só abrange os processos de detecção e geração de críticas sobre os padrões encontrados, como também o usuário poderá selecionar um dos padrões armazenados no sistema e solicitar a geração dele, apenas na linguagem *Java*. A seção 3.5 descreve este mecanismo.

3.3. Considerações para detectar padrões

Nesta seção serão discutidos os critérios utilizados para a detecção dos padrões elaborados pela *GoF* [Gamma et al. - 2000], restringindo-nos aos padrões considerados no SAMOA devido a suas características estruturais. Por exemplo, o padrão *Proxy* requer que a classe *proxy* estenda a classe *subject*, mas ela não impede o engenheiro de adicionar uma outra classe entre essas duas. Então a classe *proxy* herdará a classe adicionada e esta à classe *subject*. Esse grau de liberdade é a razão pela qual, no resto desta seção, a palavra herança não referencia diretamente herança, mas referenciará a possibilidade de herança em cascata.

O padrão *Adapter* pode ser realizado usando duas diferentes estruturas, uma baseada em herança múltipla, e a outra é baseada em herança simples. Esse padrão pode ser detectado usando uma estrutura *template* que seleciona tercetos de classes: *target*, *adapter* e *adaptee*. Na realização do *Adapter*, baseada na herança simples, o adapter herda do *target* e é associado com o *adaptee*. No de uma herança múltipla, o *adapter* herda tanto do *target* quanto do *adaptee*. Em ambos os casos o *target* deve conter no mínimo uma operação abstrata implementada pelo *adapter* e o template da colaboração requer a implementação desta operação a ser parcialmente delegada ao *adaptee*.

A detecção do padrão *Bridge* requer encontrar o terceto de classes: *abstraction*, *implementor* e *concrete implementor*. O concrete implementor herda do implementor e implementam, no mínimo, uma das operações abstratas. O abstraction e o implementor são ligados pelo relacionamento um para um existente. O template da colaboração declara que, no mínimo, uma das operações na abstração deve delegar suas responsabilidades a uma operação abstrata do implementor.

O padrão Composite pode ser detectado por todos os tercetos das classes: *component*, *composite* e *leaf*. Composite e leaf herdam da classe component e cada objeto composite contém um conjunto de componentes sem restrições de cardinalidade. A classe composite deve implementar, no mínimo, uma operação abstrata do component e esta operação deve delegar parte de suas responsabilidades aos objetos "*components*".

O padrão *Decorator* é quase similar ao Composite e pode ser detectado através de regras similares. O *Decorator* requer uma associação um para um. Quando o projeto contém mais do que uma classe decorator, o padrão Decorator é composto de quatro classes para a introdução de uma classe abstrata decorator, mas esta modificação não troca, substancialmente, as regras de detecção.

O padrão *Factory Method* envolve quatro classes: o product, o concrete product, o creator e o concrete creator. O *concrete product* herda do *product* e o *concrete creator* herda do *creator*, mas apenas o creator é abstrato pois ele contém uma operação abstrata que é implementada no *concrete creator*. Essa operação é o *factory method* e apenas ele deve retornar o objeto *product*. A primeira sugestão sobre a presença de um padrão *Factory Method* em um diagrama de classe é a presença de uma associação *create* entre o *concrete product* e o *concrete creator* que permite os objetos *concrete creator* criarem objetos *concrete product*.

A detecção do *Abstract Factory* pode ser baseada na detecção dos *factory methods*. De fato, uma das possibilidades para implementar um *abstract*

factory é criar uma classe abstrata contendo o *factory method* para cada família de produtos. Isto sugere que um *abstract factory* pode ser detectado procurando por todas as classes abstratas contendo um *factory method*.

O padrão *Iterator* é composto de quatro classes: o *aggregate*, *concrete aggregate*, *iterator* e o *concrete iterator*. As classes *aggregate* e *iterator* são abstratas e implementadas, respectivamente, pelo *concrete aggregate* e *concrete iterator*. O *aggregate* provê um *factory method*, implementado pelo concrete *aggregate*, que pode ser usado para produzir objetos *iterator*. O template estrutural pode ser completado com uma associação entre o iterator e o *aggregate* ou entre suas duplicatas concretas. Essa associação está direcionada do *iterator*, ou do *concrete iterator*, ao *aggregate*, ou ao *concrete aggregate*. O objeto *iterator* é usado, no mínimo, para dois propósitos: acessar o corrente elemento do *aggregate* e mover para o próximo na travessia. Geralmente, o objeto iterator é também utilizado para a condição de termino do laço, mas isto não é, necessariamente, uma condição e não pode ser usado para detectar esse padrão.

O padrão *Observer* requer a presença de quatro classes: *subject*, *observer* e suas implementações, *concrete subject* e o *concrete observer*. O *subject* está associado com o conjunto de *observers*, permitindo alcançar o *observer* do *subject*. O *concrete observer* está associado com o *concrete subject* e esta associação é um para um e permite o *concrete observer* acessar o *concrete subject*. Essa associação pode ser substituída com uma associação, com as mesmas propriedades, entre o *observer* e o *subject*. O *concrete observer* deve implementar, no mínimo, uma das operações abstratas do *observer*.

A detecção do padrão *Prototype* requer encontrar duas classes, a *prototype* e a *concrete prototype*, conectada por uma relação de herança. A *prototype* deve definir uma operação abstrata *clone* e a *concrete prototype* deverá implementá-la. Essa operação pode ser identificada na linguagem Java. De fato, a esta operação em Java é chamada de "*clone*" e todas as classes a possuem por causa da herança da classe "*Object*" [SunJava].

O padrão Proxy é composto por três classes: *subject*, *real subject* e *proxy*. A *real subject* e a *proxy* estão associadas e ambas herdam da classe *subject*. Todas as operações públicas na *subject* são abstratas. Além disso, cada operação herdada na classe *proxy* deve ser, parcialmente, delegada à real *subject*.

3.4. Processo de detecção de padrões

O sistema de detecção foi projetado para trabalhar tanto com códigos fonte em Java quanto em diagramas UML, de um projeto. Temos, então, duas opções para detecção de padrões pelo SAMOA. Uma delas é no tocante a um processo de reengenharia onde o projetista informará o caminho onde se encontram suas classes Java e, automaticamente, o SAMOA irá resgata-las e inferir conhecimento a partir delas com o intuito de determinar a realização de padrões. A outra opção visa que o projeto UML de classes seja exportado, em formato XMI e, em seguida, carregado pelo sistema que determinará a realização de padrões no diagrama. Nas seções a seguir, elucidaremos esse processo de detecção para cada opção mencionada acima.

3.4.1. Detecção de padrões em classes Java – No caminho da reengenharia

Apresentamos, no Capítulo 2, o framework que implementou o meta-modelo adotado pelo SAMOA e o algoritmo de detecção de padrões ambos propostos por [Meijers - 1997]. Trata-se do framework chamado *PatternsBox*. Devido ao fato deste ter implementado o modelo de Meijers (adotado nessa pesquisa como mencionado no capítulo anterior), faremos o seu uso, mas também por ele realizar o trabalho de detecção de padrões em código fonte Java (uma de nossas metas).

O núcleo do SAMOA passou a ser o *PatternsBox*. A figura a seguir representa a composição desse framework para o universo do nosso sistema.

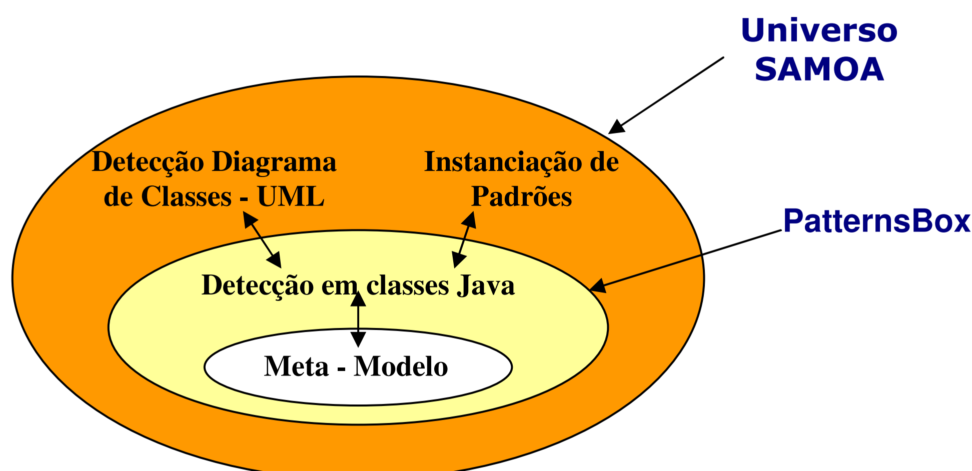

FIGURA 3.4: PatternsBox como núcleo do SAMOA

Então reutilizamos esse framework para o processo de detecção de padrões em códigos fonte Java. Mas fizemos uma pequena alteração com relação à saída do PatternsBox dos padrões encontrados. Após efetuar uma introspecção dos padrões encontrados em determinado pacote de classes Java, a classe *PatternIntrospector* retorna em uma lista, os padrões encontrados e a retorna para o usuário para que este faça uma iteração com essa lista para visualizar os resultados.

A pretensão do SAMOA é a de ser um sistema interoperável com qualquer outro. Para isto, foi alterado o formato da saída do resultado da detecção da realização de padrões. Não queremos retornar uma lista para o usuário iterar. A solução é o próprio SAMOA iterar essa lista, gerando um arquivo XML com marcações especificando os padrões detectados e o papel que cada classe desempenhou. Neste caso, o sistema retornaria ao usuário o arquivo XML gerado. Então almejamos o seguinte:

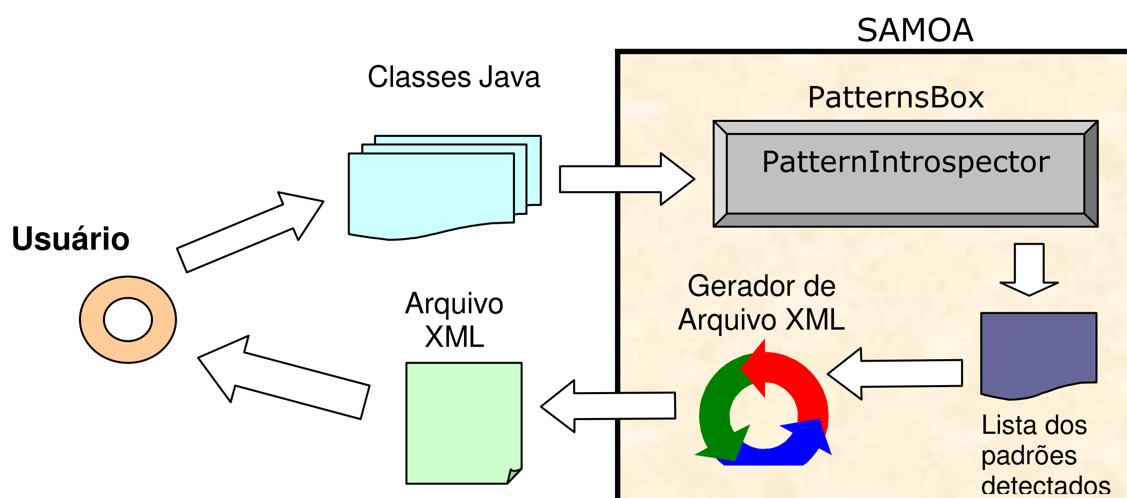

FIGURA 3.5: detecção de padrões em classes Java pelo SAMOA

Observemos, então, o seguinte exemplo. O pacote de classes ilustrados na Figura 3.7 representa um conjunto de classes Java que foram submetidas ao SAMOA, para o processo de detecção de padrões.

```
public abstract class Grafico {
        abstract void operacao();
}
```

```
public class Figura extends Grafico {
        // Association: graficos
        private Vector graficos =
                new Vector();
        public void addComponent(Grafico aComponent) {
                graficos.addElement(aComponent);
        }
        public void removeComponent(Grafico aComponent) {
                graficos.removeElement(aComponent);
        }
        public void operacao() {
        }
}
```

```
public class Retangulo extends Grafico {
      public void operacao() {
        }
}
```

```
public class Linha extends Grafico {
      public void operacao() {
        }
}
```

```
public class Circulo extends Grafico {
      public void operacao() {
        }
}
```

TABELA 3.1: classes submetidas a detecção de padrões pelo SAMOA

As classes representam a realização do padrão Composite [Gamma et al. - 2000]. A figura a seguir representa, na linguagem UML, essas classes.

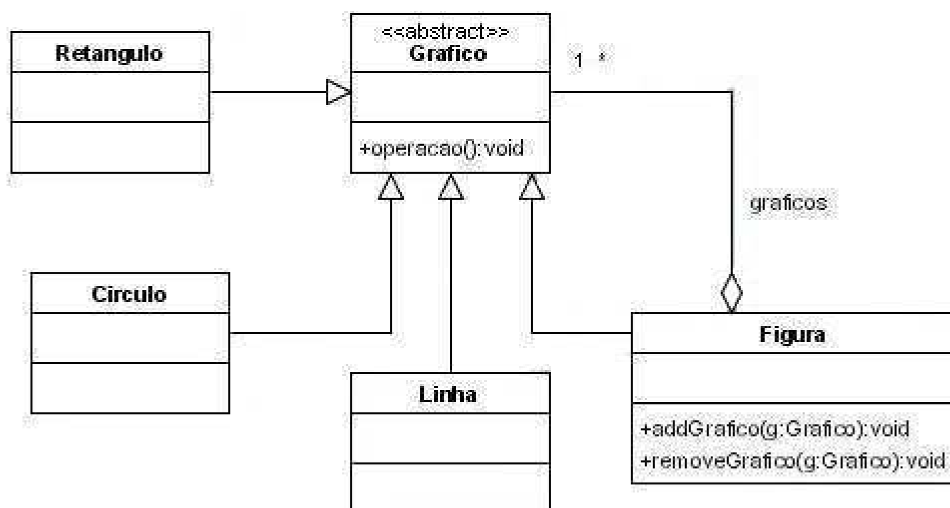

FIGURA 3.6: representação UML de uma instancia do padrão Composite

As classes são passadas ao *PatternsBox* que efetuará a detecção da realização dos padrões. Em seguida, ele retornará uma lista dos padrões encontrados, repassando para o pacote do *samoa.xml* (responsável pela geração do arquivo XML) que irá iterar essa lista e construir o arquivo XML com marcações informando os padrões encontrados e o papel que cada classe desempenha no padrão. Por fim, o SAMOA fornecerá, ao usuário, a seguinte saída:

```
<?xml version = '1.0' encoding = 'ISO-8859-1' ?>
<SAMOA>
	<PadroesDetectados>
		<Composite>
			<Composite>Figura</Composite>
			<Component>Grafico</Component>
			<Leaf>Retangulo</Leaf>
			<Leaf>Linha</Leaf>
			<Leaf>Circulo</Leaf>
		</Composite>
	</PadroesDetectados>
</SAMOA>
```

FIGURA 3.7: arquivo XML com os padrões detectados nas classes submetidas

O Patternsbox é capaz de detectar semelhanças com determinados padrões desde que, no mínimo, as classes e os seus relacionamentos estejam bem definidos. Questões relacionadas à visibilidade e atributos das classes, podem ser descartados pelo framework com intuito de detectar, tanto quanto possível, a semelhança com um determinado padrão.

3.4.2. Detecção de padrões em diagramas de classe UML

Um outro meio de detectar a realização de padrões é através da analise de diagramas de classe UML. A razão para se detectar padrões nesse tipo de diagrama é que o processo de detecção, como mencionado em seções anteriores, é baseado em fragmentos. As classes, interfaces e relacionamentos são fragmentados em entidades e em elementos pelo *PatternsBox* (que analisa apenas classes Java), representando suas formas estruturais. E nessa pesquisa, trataremos apenas de padrões detectáveis (que podemos encontrar suas realizações através de uma estrutura definida através de diagramas de classes).

O mecanismo de detecção, neste caso, consiste nas seguintes etapas:

1. o projetista constrói o projeto de seu sistema, no caso o diagrama de suas classes, em uma ferramenta CASE qualquer (desde que esta o exporte em um formato XMI);

2. depois de feito o diagrama, o projetista deverá solicitar à ferramenta CASE que o exporte em formato XMI.

3. em seguida, o SAMOA receberá como entrada o arquivo XMI gerado. O sistema recupera os constituintes do XMI fornecido. Trata-se das classes, interfaces, métodos e relacionamentos representados nesse arquivo em termos de marcações que se encontram definidas, estruturalmente, por uma DTD (neste caso, uma DTD específica para representar elementos de diagramas UML, ver [XMI - 2003]). Na medida que cada elemento for extraído

do XMI fornecido, esse pacote irá mapear cada um desses e irá instanciar objetos de algumas classes definidas pelo meta-modelo de [Meijers - 1997] – tais classes encontram-se implementadas pelo *PatternsBox*.

Se for extraído um elemento do XMI que representa uma classe específica , o SAMOA irá instanciar um objeto da classe *PClass*[1] e adicionará a ele, informações referente à classe que está sendo extraída. Essas informações a serem adicionadas são referentes a métodos, atributos, associações, generalizações e realizações. Sendo, então, instanciados objetos da classe *Pmethod*[1]*, PAttribute*[1]*, PAssoc*[1]*, PClass*[1] *e PInterface*[1], respectivamente. De forma semelhante acontece quando o SAMOA, ao ler um arquivo XMI encontra uma marcação discriminando uma interface (*UML:Interface*). Neste caso o sistema irá criar uma instância da classe *PInterface*[1] e varrer as informações restantes dessa interface a adicioná-las ao objeto criado. Na medida que cada um desses objetos, extensões da classe *PEntity*[1] forem criados, são armazenados em uma coleção. Esta será repassada ao sistema que irá percorrê-la extraindo os objetos nela contidos, e para cada um desses que for recuperado será gerado um código fonte *Java*.

Ressaltamos que nem todos os constituintes do XMI serão mapeados para Java, apenas as informações necessárias para definir, estruturalmente, as classes, interfaces, métodos e seus relacionamentos. Ou seja, apenas o necessário para definir as suas assinaturas.

4. Após o mapeamento, as informações extraídas do XMI são transformadas em arquivos no formato da linguagem *Java* e, em seguida, são repassados para o *PatternsBox*. Então, seguem os passos semelhantes ao processo de detecção como no processo de reengenharia descrito na seção anterior. Reusamos o processo de detecção, que no final, retornará um arquivo XML com os padrões encontrados.

Essa forma de detecção pode ser vista como uma camada acima do processo de detecção descrito na seção anterior. Observe a figura:

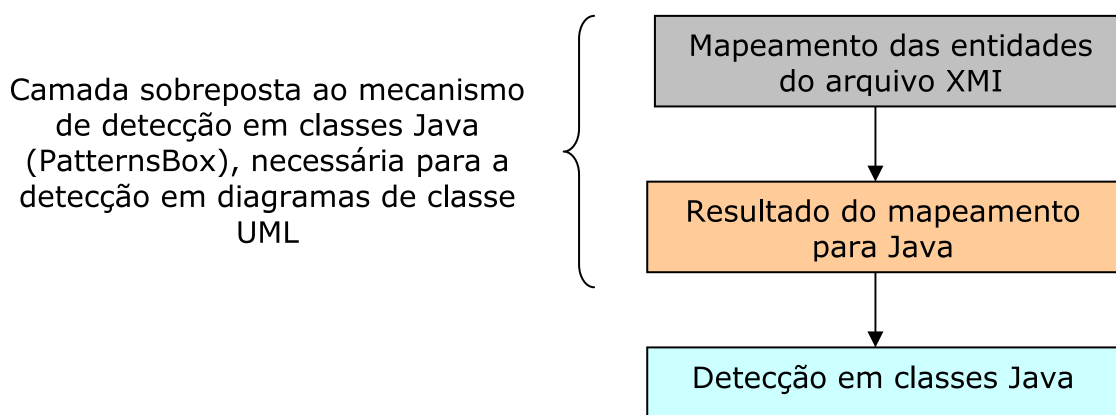

FIGURA 3.8: relacionamento do processo de detecção em diagramas UML vs. classes Java

O exemplo a seguir trata de um projeto no qual as classes representam a realização do padrão *Observer* [Gamma et al. - 2000]. Elas foram modeladas na ferramenta CASE Poseidon [Poseidon - 2003].

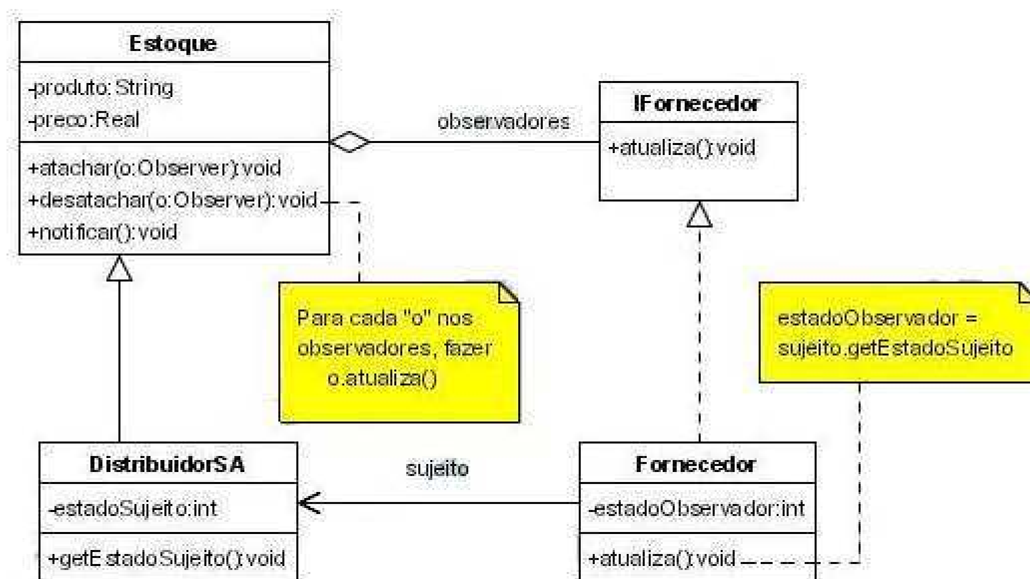

FIGURA 3.9: exemplo de uma realização do padrão Observer

[1] As classes foram modeladas por Meijers[Meijers - 1997] e implementadas no framework *PatternsBox*

Esse diagrama, foi exportado para o formato XMI e fornecido ao SAMOA. A figura a seguir ilustra um trecho do conteúdo do arquivo XMI gerado.

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' ...>
     ...
    <UML:Class xmi.id = '...' name = 'Estoque' visibility = 'public' isRoot = 'false' isLeaf =
      'false'          isAbstract = 'false' isActive = 'false'> ...
    <UML:Classifier.feature>
     <UML:Attribute xmi.id = 'lsm:14c0275:-7ff4' name = 'produto' visibility = 'private' ...>
      <UML:StructuralFeature.type>
       <UML:Class xmi.idref = 'lsm:14c0275:f93bd838dd:-7ff3'/>
      </UML:StructuralFeature.type>
     </UML:Attribute> ...
     <UML:Operation xmi.id = 'lsm:14c0275:-7ff1' name = 'atachar' visibility = 'public' ...>
      <UML:BehavioralFeature.parameter>
       <UML:Parameter xmi.id = 'lsm: f93bd838dd:-7ff0' name = 'o' .../>
        <UML:Parameter.type>
         <UML:DataType xmi.idref = 'lsm:14c0275:f93bd838dd:-7fed'/>
        </UML:Parameter.type>
       </UML:Parameter>
      </UML:BehavioralFeature.parameter>
     </UML:Operation>
    <UML:Interface xmi.id = '...' name = 'IFornecedor' visibility = 'public' ...>
          <UML:Classifier.feature>
                <UML:Operation xmi.id = '...' name = 'atualiza' visibility = 'public' ...>
                         ...
                </UML:Operation>
            ...
</XMI>
```

FIGURA 3.10: XMI simplificado do diagrama de classe

Como mencionado anteriormente, o SAMOA mapeia um subconjunto dos elementos XMI para Java. Repassando-os, em seguida, para o *PatternsBox*. Observe a ilustração abaixo, ela estende a Figura 3.6 que representa o processo de detecção em classes Java.

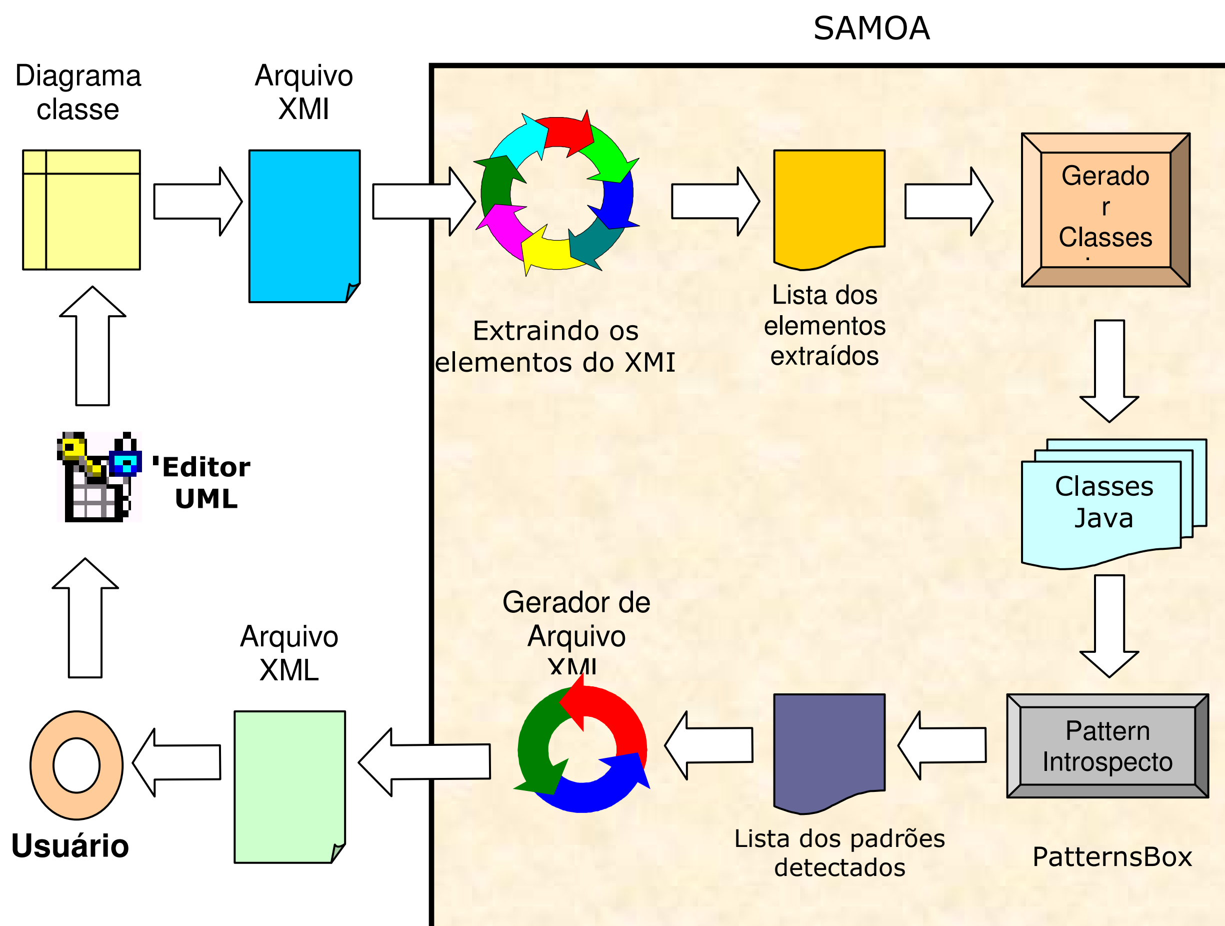

FIGURA 3.11: processo de detecção em diagramas de classe UML

Utilizamos o *PatternsBox* apenas nos processos de detecção de padrões, tanto em diagramas de classes quanto em fontes Java.

3.5. Geração de Críticas

Este mecanismo representa o sistema de críticas do SAMOA. Enquadra-se na fase posterior ao processo de detecção de padrões.

Vimos que uma vez detectado padrões, tanto em diagramas UML quanto em classes Java, o sistema constrói um arquivo, no formato XML, com marcações informando os padrões detectados e seus participantes (qual classe ou interface assumiu um determinado papel no padrão detectado). Esse arquivo seria passado ao sistema de críticas para que este proponha melhorias ao diagrama editado.

Essas críticas encontram-se armazenadas em uma base lógica de conhecimento e implementadas em alguma linguagem lógica orientada a objetos.

Exemplificando, suponha que o diagrama da Figura 3.6 foi submetido ao SAMOA e este acusou a realização do padrão *Composite*. O arquivo XML seria recuperado pelo sistema de críticas para que este saiba qual foi o padrão detectado, a fim de selecionar um conjunto de críticas para utilizar sobre o padrão encontrado. Neste caso específico, o padrão encontrado foi o *Composite*, logo o sistema de críticas irá recuperar de sua base lógica as críticas que podem ser aplicadas sobre ele e listar o resultado de sua aplicação para o usuário. No caso do diagrama da Figura 3.6, as críticas poderiam ser:

- inserir o método getGraficos para retornar os componentes do *Composite*;
- Considerar o uso do padrão *Iterator* para acessar aos objetos gráfico em figura;
- em Gráfico deverão ser definidas tantas operações para Figura, Retângulo, Linha e Texto quanto possíveis.
- não colocar referencia para os filhos em Gráfico.

Observemos que algumas restrições(*constraints*) não podem ser definidas utilizando a linguagem UML, mas sim com OCL. Essas restrições podem ser úteis para auxiliar ao sistema de críticas no processo de críticas aos padrões encontrados. Por exemplo, suponhamos que queremos detectar a realização do padrão *Singleton*, como representar em UML uma classe que tenha somente uma instância? Essa restrição seria especificada em OCL e o SAMOA poderia critica-la.

3.6. Processo de Instanciação de Padrões

Nesta seção elucidamos o processo de instanciação de padrões. Para sermos claros na explicação desse processo, tomaremos por base um exemplo

de instanciação do padrão *Composite* [Gamma et al. - 2000]. Os passos para instanciação desse padrão são os seguintes:

(1) o primeiro passo consiste em especializar o meta-modelo adicionando todos os elementos de comportamentos e de estruturas.

(2) O segundo passo consiste na instanciação do meta-modelo para o padrão Composite (este meta-modelo encontra-se já especificado no repositório de padrões), construído de acordo com a semântica do meta-modelo. Chamamos, então, o modelo resultante de modelo abstrato devido ao fato deste não conter informações sobre o contexto da aplicação do usuário. Esse modelo abstrato corresponde a uma "definição" do padrão Composite e contém todas as informações necessárias a construção deste específico padrão.

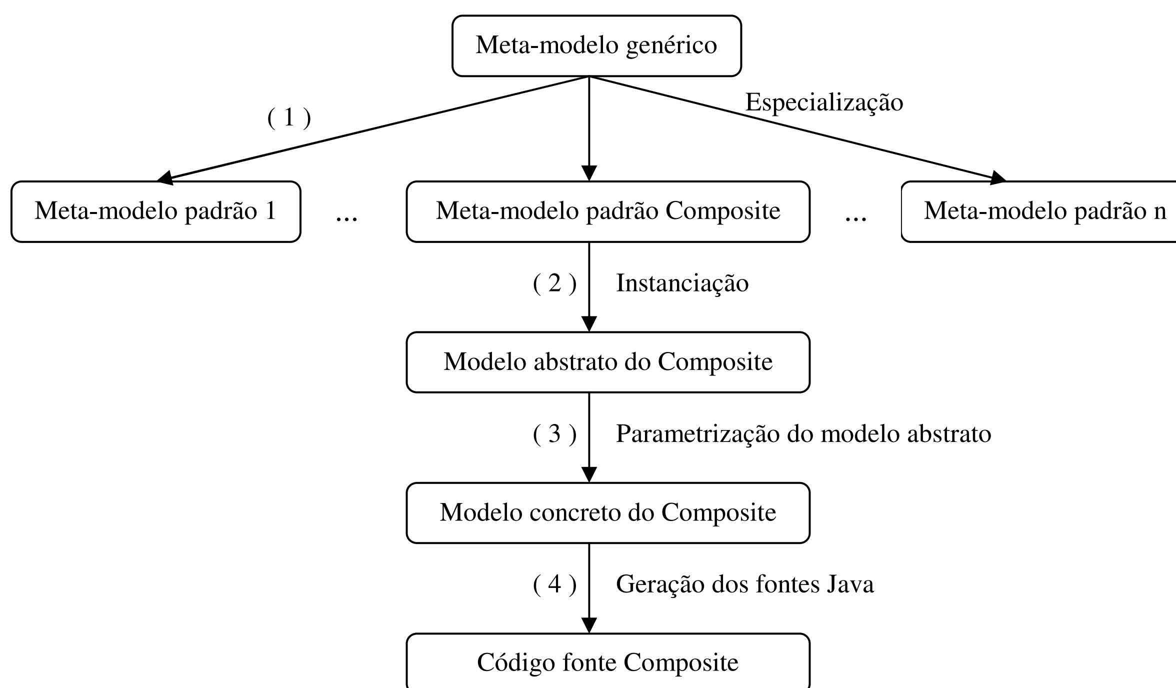

FIGURA 3.12: processo de instanciação do padrão Composite

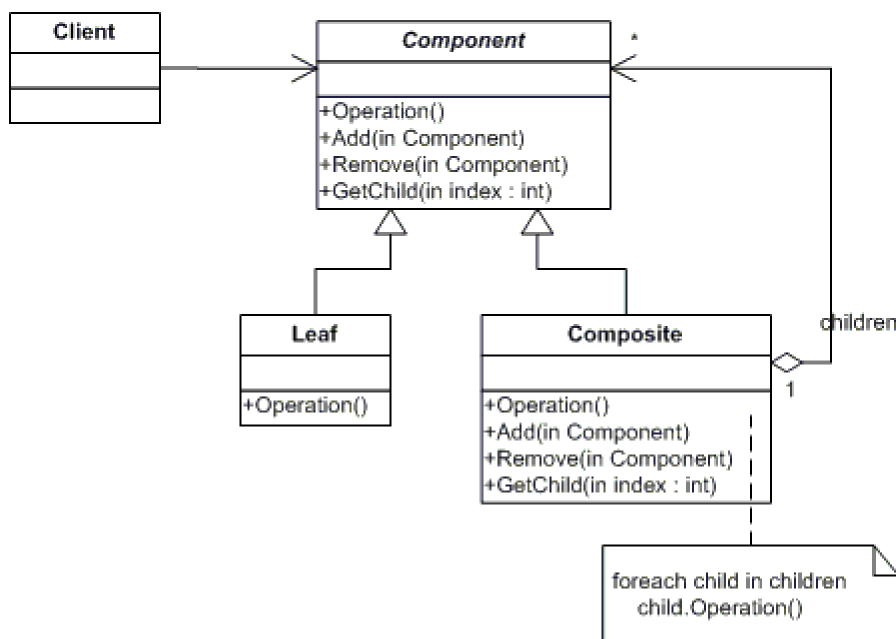

FIGURA 3.13: estrutura do padrão *Composite*

O modelo abstrato é expresso em uma classe *Java* chamada *Composite*, subclasse de *Pattern*. A tabela a seguir ilustra um fragmento dessa declaração baseado no diagrama da Figura 3.14.

Definição do modelo abstrato do padrão *Composite*
class Composite extends Pattern {
Declaração do construtor
Composite(...) { ...
Declaração do ator (participante) *Componente*
component = new Pinterface("Component") operation = new Pmethod("operation") component.addPElement(operation) this.addPEntity(component)
Definição da associação *"children",* com cardinalidade n
children = new Passoc("children", component, n)
Declaração do ator (participante) *Composite*
composite = new PClass("Composite") composite.addShouldImplement(component) composite.addPElement(children)
O método *"operation"* definido em *Composite* implementa o método *"operation"* de *Componente* e está ligado a associação *"children"*
aMethod = new PdelegatingMethod("operation", children) aMethod.attachTo(operation) composite.addPElement(aMethod)

this.addPEntity(composite)
Declaração do ator (participante) *Leaf*
leaf = new PClass("Leaf") leaf.addShouldImplement(component) leaf.assumeAllInterfaces() this.addPEntity(leaf) }
O serviço *addLeaf* serve para adicionar "*Leafs*" a corrente instancia do padrão *Composite*

void addLeaf(String leafName) {
PClass newPClass = new PClass(leafName)
newPClass.addShouldImplement(
(PInterface)getActor("Component"))
newPClass.assumeAllInterfaces()
newPClass.setName(leafName)
this.addPEntity(newPCclass)
}

TABELA 3.2: definição do modelo abstrato do Composite

Os modelos abstratos são armazenados dentro de um repositório (classe *PatternsRepository*, Figura 3.2). Esse repositório ajuda a acessar, facilmente, o padrão previamente definido.

(3) Neste passo, há a transformação do modelo abstrato para o concreto. Este representa o padrão aplicado para se encaixar aos requisitos da aplicação do usuário. Para ilustrar o procedimento de instanciação nós adotamos um exemplo introduzido em [Gamma et al. - 2000] e reusado em [Winter et al. - 1996]. Esse exemplo define uma hierarquia de objetos gráficos como mostra a Figura 3.7.

A instanciação do modelo abstrato é realizada pela instanciação da classe *Composite* definida no passo (2). Então, a instancia é parametrizada (cada participante do padrão é, por exemplo, nomeado) para construir a aplicação concreta (Figura 3.13). O código fonte exibido na Tabela 3.3 é dado como um exemplo. Obtivemos do modelo abstrato do padrão e o transformamos para o contexto do usuário. Então, o SAMOA dinamicamente produz o código baseado no contexto do usuário, gerando as seguintes saídas.

Declaração de um novo modelo concreto Composite
Composite p = new Composite();
p.getActor("Component").setName("Grafico");
p.getActor("Component").getActor("operation").setName("desenhar");
p.getActor("Leaf").setName("Texto");
p.getActor("Composite").setName("Figura");
p.addLeaf("Linha");
p.addLeaf("Retangulo");

TABELA 3.3: construção do modelo concreto Composite

(4) O passo final consiste na geração de código. Trata-se de um processo automático desempenhado em cima do modelo concreto gerado. O pacote *samoa.util* encarrega-se de gerar os fontes em *Java*. Ele recebe os modelos concretos gerados, na forma de uma lista e em seguida, começa a geração das classes e interfaces na linguagem *Java*. Feito isto, os arquivos gerados são compactados e disponibilizados ao usuário.

A figura a seguir resume os passos para a instanciação de um padrão específico.

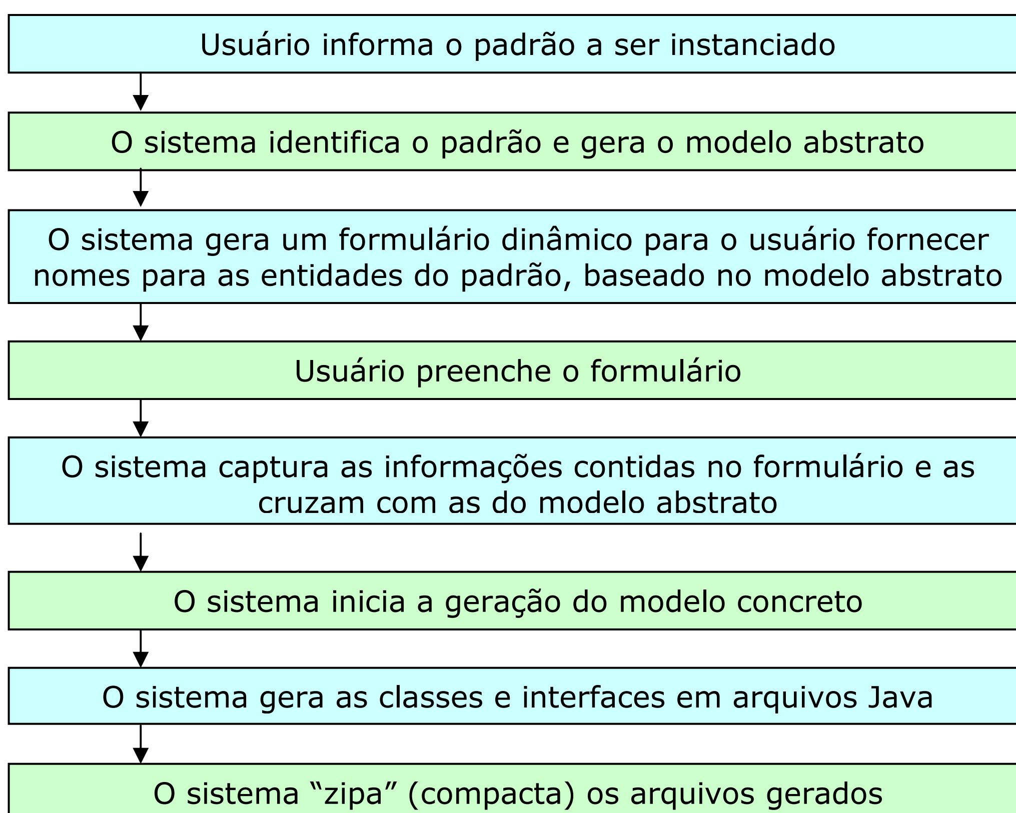

FIGURA 3.14: seqüência do processo de instanciação

O código fonte em *Java* obtido do modelo concreto referente ao diagrama da Figura 3.7, é o seguinte:

Classe abstrata Gráfico

```
public abstract class Grafico{
       public abstract void operacao();
}
```

Classe Figura

```
public class Figura extends Grafico{
       //Associação graficos
       private Vector graficos = new Vector();
      public void addGrafico(Grafico grafico)
      {filhos.add(grafico);}
      public void removeGrafico(Grafico grafico)
      {filhos.remove(grafico);}
      public void operacao( )
      {for (Enumeration enum = graficos.elements();
                                    enum.hasMoreElements();
             ((Grafico)enum.nextElement()).operacao());
       }
}
```

Classe Círculo

```
public class Texto extends Grafico{
      public void operacao(){}
}
```

Classe Linha

```
public class Linha extends Grafico{
      public void operacao(){}
}
```

Classe Retângulo

```
public class Retangulo extends Grafico{
      public void operacao(){}
}
```

TABELA 3.4: fontes Java gerados

3.7. Conclusão

Foram descritos nesse capítulo todos os aspectos envolvidos com o projeto da ferramenta SAMOA. Mostramos, basicamente, o que é o SAMOA, como ele foi idealizado e projetado e quais funcionalidades ele se propõe atender.

Fizemos uma abordagem geral da ferramenta, através da ilustração e descrição da sua arquitetura, detalhando-a, posteriormente, para descrever a forma como a mesma foi concebida, quais os seus propósitos e quais as funcionalidades oferecidas pelo SAMOA seriam disponibilizadas aos seus usuários, ou seja, como elas seriam apresentadas.

Capítulo 4 – Desenvolvimento do SAMOA

O nosso projeto contempla a concepção e o desenvolvimento de uma ferramenta para manipular a detecção, instanciação, assim como críticas para uma melhor utilização de padrões de projeto: o Sistema de Apoio a Modelagem Orientada a Objetos de Aplicações - SAMOA.

Em nosso projeto, adotamos o desenvolvimento iterativo e incremental definido por [Larman - 1998].

4.1. Modelo de domínio

No SAMOA temos um processo de detecção, instanciação e crítica a padrões de projeto. Para efetuar a instanciação de um padrão, se faz necessário ter seu meta-modelo descrito em algum repositório, assim como, para detectar um padrão em alguma classe Java ou diagrama UML. Uma vez detectado um padrão, o sistema deverá ter a habilidade de construir críticas a respeito da realização do padrão detectado. Assim como os meta-modelos de padrões, as críticas relacionadas a determinados padrões deverão estar, também, em um repositório.

A Figura 4.1 ilustra o modelo de domínio do SAMOA:

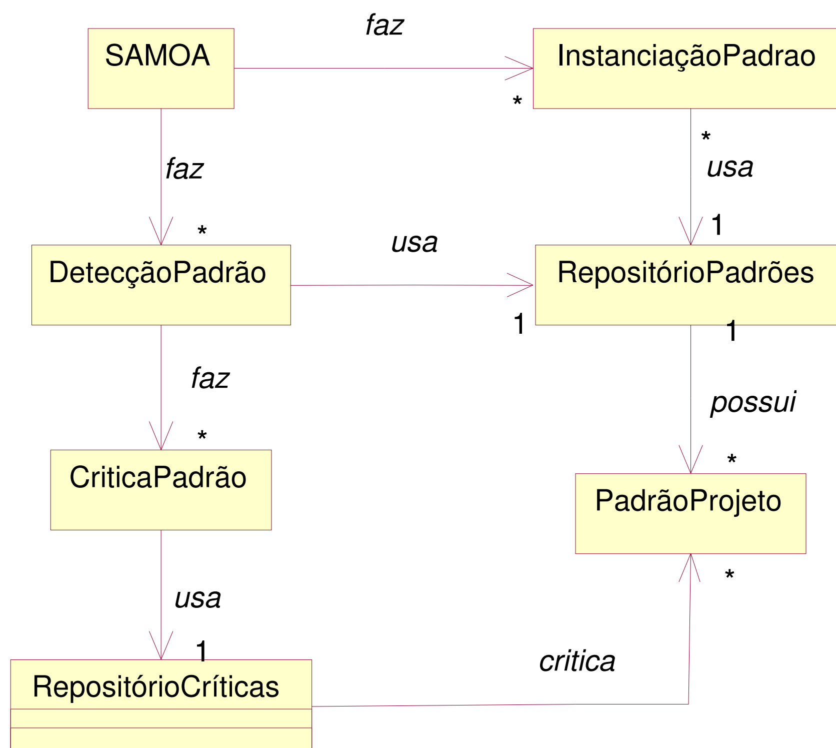

FIGURA 4.1: modelo conceitual do SAMOA

4.2. Levantamento dos Requisitos

Na análise e no projeto do SAMOA avaliamos dois tipos de usuários para a ferramenta:

- usuários que desejam verificar a existência de um padrão em um determinado diagrama de classe UML. Neste caso, o usuário poderá ser um simples projetista ou uma ferramenta CASE fazendo uso da nossa ferramenta. Deverá receber alguma sugestão para melhoria do uso do padrão detectado;
- usuários que desejam instanciar um padrão para um contexto específico, e ter um código fonte gerado na linguagem Java.

Por meio do diagrama de casos de uso mostrado abaixo (Figura 4.2), descreveremos como os usuários (**Ator**) poderão interagir como o SAMOA.

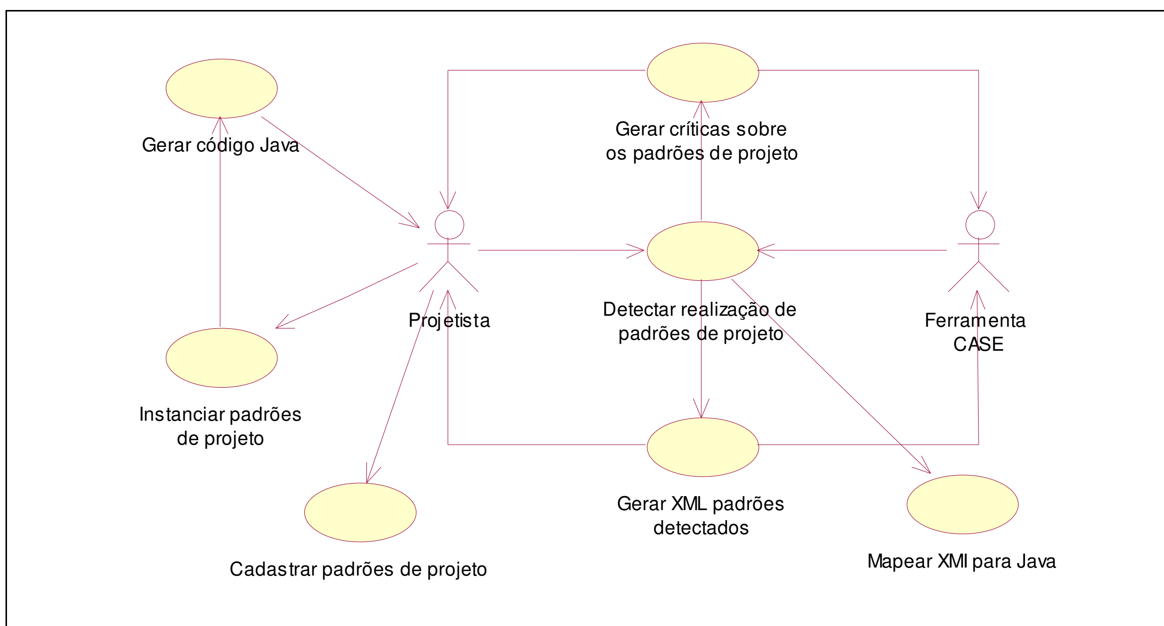

FIGURA 4.2: diagrama de casos de uso

4.2.1. Requisitos Funcionais do SAMOA

Caso de Uso	Descrição
Detectar a realização de padrões de projeto	Permite detectar a realização de padrões em diagramas de classe UML exportado no formato XMI.
Mapear XMI para Java	Permitir mapear os elementos constuintes do modelo UML /XMI para arquivos fonte Java.
Gerar XML dos padrões detectados	Permitir a construção de um arquivo XML contendo os padrões detectados no diagrama fornecido pelo usuário. Os padrões encontrados, assim as entidades que representam cada um representam, deverão estar delimitado por marcações, *tags*, no arquivo XML.
Gerar críticas sobre os padrões de projeto	Uma vez detectado a realização de padrões, os que forem detectados são repassados a um sistema de críticas para que este elabore críticas/sugestões que contribuam para uma melhor utilização do padrão.

Instanciar padrões de projeto	Permitir que o projetista instancie um determinado padrão que ele desejar para um específico contexto.
Gerar código Java	Permitir a geração de código, em Java, do padrão instanciado.
Cadastrar padrões de projeto	Permitir que o projetista armazene os modelos abstratos dos padrões de projeto, no repositório de padrões do SAMOA.

TABELA 4.1: requisitos funcionais do SAMOA

4.3. Planejamento dos Incrementos

Em nosso projeto, adotamos o desenvolvimento iterativo e incremental [Larman - 1998].

Cada ciclo (iteração) trata um conjunto relativamente pequeno de requisitos, realizado através da análise, do projeto, da construção e do teste. A conclusão de cada ciclo permitirá que a ferramenta cresça incrementalmente [Larman - 1998].

Definimos quatro incrementos para a construção da ferramenta. O planejamento dos incrementos foi baseado na relevância das funcionalidades, às quais o SAMOA se propunha.

Determinamos que o primeiro incremento do SAMOA (sem ser na WEB), deveria atender a principal finalidade do nosso projeto, que seria a construção de um aplicativo para detectar padrões em diagramas UML, exportados em XMI. Consideramos tal finalidade primordial, visto que ela é o alicerce da arquitetura do projeto e, a partir do seu desenvolvimento, nos deparamos com diversas questões relacionadas às dificuldades de implementação que nos permitiram definir posteriormente todas as outras funcionalidades da ferramenta, abrangendo o que foi disposto na arquitetura.

Nesse incremento também foi desenvolvida a tradução de XMI para a linguagem Java que é o processo intermediário entre o diagrama UML/XMI (projetado pelo usuário em uma ferramenta CASE), e o processo de detecção pelo framework [PatternsBox]. Concluímos esse incremento desenvolvendo as funcionalidades de importar UML/XMI e exportar para fontes Java. Como resultado desse incremento, temos um detector de padrões capaz de detectar padrões em diagramas UML, reutilizando o mecanismo de detecção do [PatternsBox]. E, para testes, criamos um repositório de dados com definições de padrões, baseados no meta-modelo proposto por [Meijers - 1997]. Esse repositório consiste em simples arquivos de texto, cada um tendo a representação de um padrão. Mais tarde, transpomos os objetivos alcançados nesse incremento, para uma plataforma WEB.

O segundo incremento contemplou os requisitos de funcionalidade exigidos para a instanciação de um padrão específico requisitado pelo desenvolvedor. Também, mais tarde, transpomos esse incremento para a plataforma WEB.

Em um terceiro incremento, transpomos os resultados das 1ª e 2ª interações para a plataforma WEB.

Um quarto incremento foi especificado, mas não concluído. Trata-se do processo de selecionar críticas para a melhoria do uso de um específico padrão detectado pelo SAMOA. Elaboramos, textualmente, pontos a serem checados quando forem encontradas realizações de padrões. Porém não chegamos a implementar este incremento devido ao fator tempo e deixamos aqui apenas um projeto arquitetural de como ele seria. Deixamos então a implementação deste como sugestão para trabalhos futuros.

4.4. Implementação e testes

Destacaremos nesta seção alguns aspectos no tocante ao processo de implementação e testes do SAMOA.

O sistema foi totalmente desenvolvido na linguagem de programação Java e fizemos a utilização de dois frameworks, o *PatternsBox* (o qual discutimos no Capítulo 3) e o *JUnit* [JUnit Java] usado no intuito de desenvolver testes de unidades para as classes desenvolvidas (por exemplo, testar os métodos). Mas não apenas testes de unidades foram desenvolvidos. Também fizemos testes funcionais para testar se o sistema atingiu os nossos requisitos.

A camada de apresentação construímos com a tecnologia JSP*(Java Server Pages)* para a construção de páginas WEB dinâmicas [JSP Java]. E na camada de dados utilizamos, apenas, arquivos que contêm a descrição de padrões baseados no meta-modelo de [Meijers - 1997]. Já na camada de dados, no momento, utilizamos apenas arquivos.

Para que a arquitetura do sistema funcione, vários pacotes de serviços foram desenvolvidos. A seguir descreveremos os pacotes criados:

Pacote	Objetivos
samoa.xmi.util	Desenvolvido para extrair os elementos XMI e gerar objetos do meta-modelo para cada entidade encontrada.
samoa.util.arquivo	Após terem sido criados os objetos, estes são entregues ao pacote *samoa.util.arquivo* que se encarregará em gerar os arquivos fontes *Java*.
	Após terem sido instanciados os padrões, os fontes gerados são compactados e um arquivo no formato *.zip* .
samoa.util.xml	Pacote que recebe como entrada uma lista dos padrões encontrados. Em seguida, gera um arquivo XML com marcações definindo os padrões encontrados e o papel de cada classe assume no padrão encontrado.

<table>
<tr><td>Samoa.util.instancia</td><td>Pacote que recebe uma requisição de um padrão a ser instanciado e em seguida recupera as informações, do repositório de padrões, e constrói uma instancia do padrão.</td></tr>
</table>

TABELA 4.2: pacotes de serviço do SAMOA

Ilustramos na tabela abaixo o relacionamento entre cada pacote com os módulos do sistema, discutidos no Capítulo 3.

<table>
<tr><th>Módulo</th><th>Pacote(s) necessário(s)</th></tr>
<tr><td>Instanciação e geração de código</td><td>• samoa.util.instancia
• samoa.util.arquivo</td></tr>
<tr><td>Módulo de entrada</td><td rowspan="2">• samoa.util.arquivo
• samoa.util.xml
• samoa.xmi.util
• PatternsBox</td></tr>
<tr><td>Detecção da realização de padrões</td></tr>
</table>

TABELA 4.3: relacionamento entre os pacotes e os módulos do sistema

4.5. Diagrama de classes e de seqüência

Um diagrama de classes ilustra as especificações para as classes desenvolvidas para o SAMOA. Mostramos, na Figura 4.3, o diagrama desenvolvido.

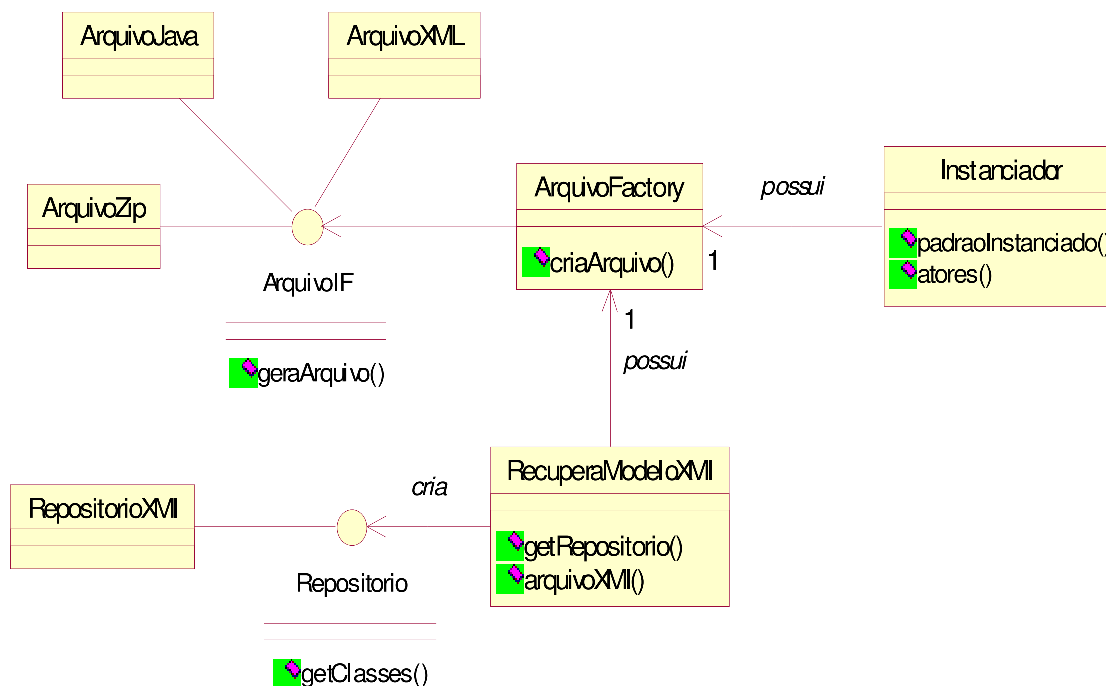

FIGURA 4.3: diagrama de classes

A classe *RecuperaModeloXMI* é responsável pela recuperação dos elementos XMI descritos em um arquivo deste formato. Ela procura identificar as classes e interfaces descritas, assim como seus métodos, relacionamentos, atributos (no caso de uma classe etc.) e armazena essas informações encontradas no objeto da classe *RepositorioXMI*. Em seguida, este objeto é passado para um outro objeto da classe *ArquivoJava*, para que sejam construídas as classes Java. Depois de geradas as classes Java, elas são passadas para o framework *PatternsBox*. Depois detectado os padrões, um objeto da classe *ArquivoXML* encarrega-se de construir um arquivo XML com os padrões detectados.

No parágrafo acima, a forma de interação entre as classes diz respeito ao processo de detecção de padrão. No caso de um processo de instanciação, o sistema recupera os padrões do repositório através do PatternsBox e em seguida o objeto da classe *Instanciador* receberá as informações referentes ao contexto do qual o usuário deseja instanciar o padrão. Feito isso, o objeto da classe Instanciador faz uso de um outro objeto da classe *ArquivoJava* para a geração de

código, em Java, do padrão instanciado. Em seguida, um outro objeto entra em cena. Trata-se do objeto da classe ArquivoZip que irá gerar um arquivo compactado, no formato *zip*, dos fontes Java gerados. As Figuras 4.4 e 4.5 ilustram, através de diagramas de seqüências, o as iterações existentes no processo de detecção e instanciação de padrões.

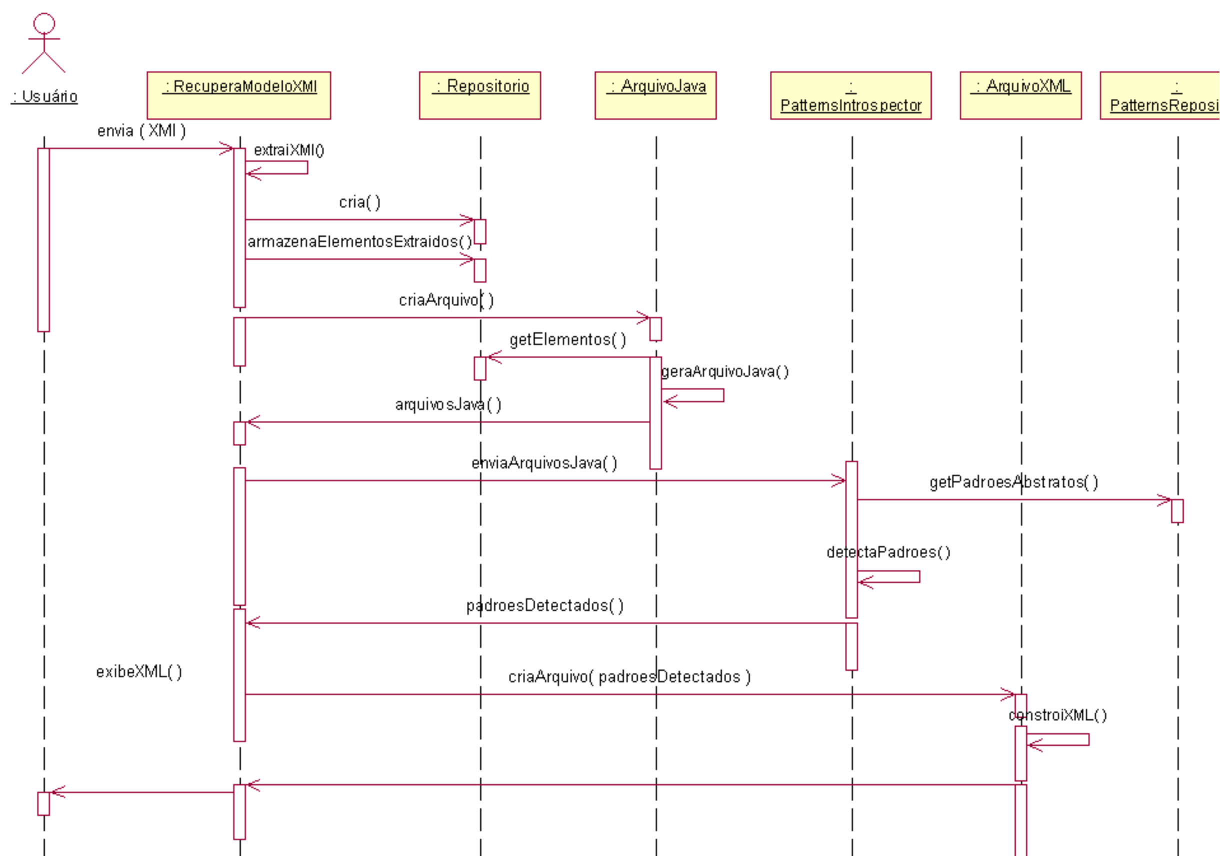

FIGURA 4.4: seqüência do processo de detecção em diagramas UML

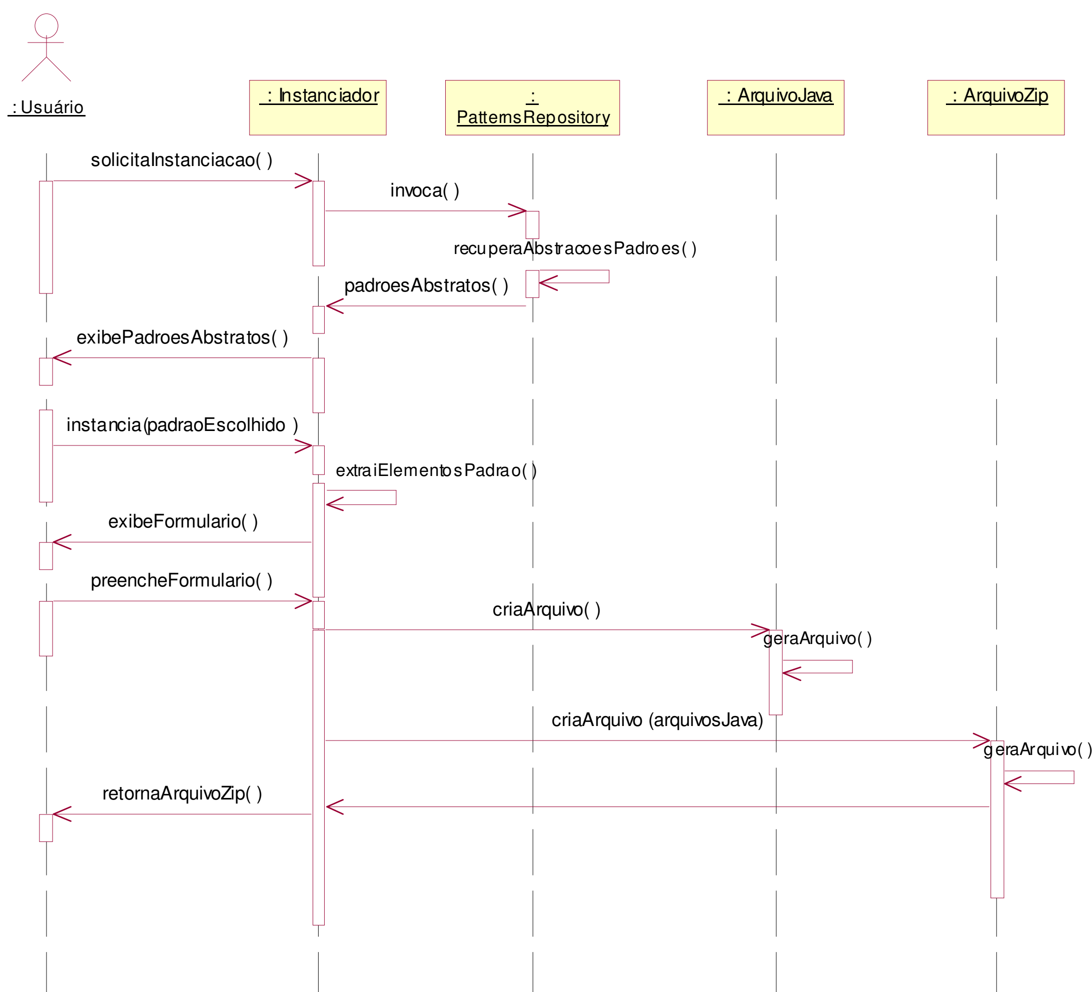

FIGURA 4.5: seqüência do processo de instanciação de um padrão

A classe *RecuperaModeloXMI* possui os seguintes métodos:

- arquivoXMI – método que recebe o caminho onde esta localizado o arquivo XMI;
- getRepositorio – método que cria e retorna um objeto da classe *RepositorioXMI*, que contém as informações contidas no XMI fornecido.

A classe *RepositórioXMI* possui os seguintes métodos:

- getClasses – retorna as classes encontradas no XMI.
- getInterfaces - retorna as interfaces encontradas no XMI.

A classe *ArquivoFactory* é uma classe cujo objetivo representa uma fábrica de outros objetos. Por exemplo, ela poderá criar um objeto da classe *ArquivoXML*, *ArquivoJava* ou *ArquivoZip*. Ela foi baseada no padrão *AbstractFactory* [Gamma et al. - 2000]. Ela possui o seguinte método:

- criaArquivo – retorna um objeto do tipo *ArquivoIF*.

A interface *ArquivoIF* possui o seguinte método:

- geraArquivo – as classes que realizam essa interface deverão implementar esse método para gerar o arquivo compatível com o formato desejado. Por exemplo, o objeto da classe *ArquivoJava*, ao invocar o seu método geraArquivo, deverá gerar um arquivo *Java* de acordo com a sintaxe desta linguagem de programação. O mesmo se aplica às classes *ArquivoXML* e *ArquivoZip*, que deverão gerar arquivos no formato XML e Zip, respectivamente.

A classe instanciador possui os seguintes métodos:

- padraoInstanciado – método que informa o padrão que esta sendo instanciado;
- atores – método que retorna os atores participantes do padrão, no contexto desejado pelo usuário.

4.6. Conclusão

Descrevemos neste capítulo, as principais características, da análise até a implementação, envolvidas para construir o SAMOA.

Descrevemos, segundo a metodologia adotada para o desenvolvimento do nosso sistema, os requisitos funcionais e os diagramas de classe

representando as classes projetadas para o SAMOA. Definimos, também, os pacotes desenvolvidos e visamos um cruzamento entre as funcionalidades previstas e os pacotes que as atendem.

Foram descritas de forma individualizada as soluções computacionais utilizadas para atender as funcionalidades do SAMOA.

Capítulo 5 - Estudo de Caso

Mostramos um estudo de caso onde utilizamos o SAMOA para detectar a realização de padrões em um diagrama de classes UML e um processo de instanciação de um específico padrão. Começaremos, então, pelo processo de detecção de padrões.

Detecção de Padrões em diagramas UML

Suponhamos que o projetista utilizou uma ferramenta CASE para a definir um diagrama de classe UML, para um editor de documentos que possui figuras. E projetou o seguinte diagrama:

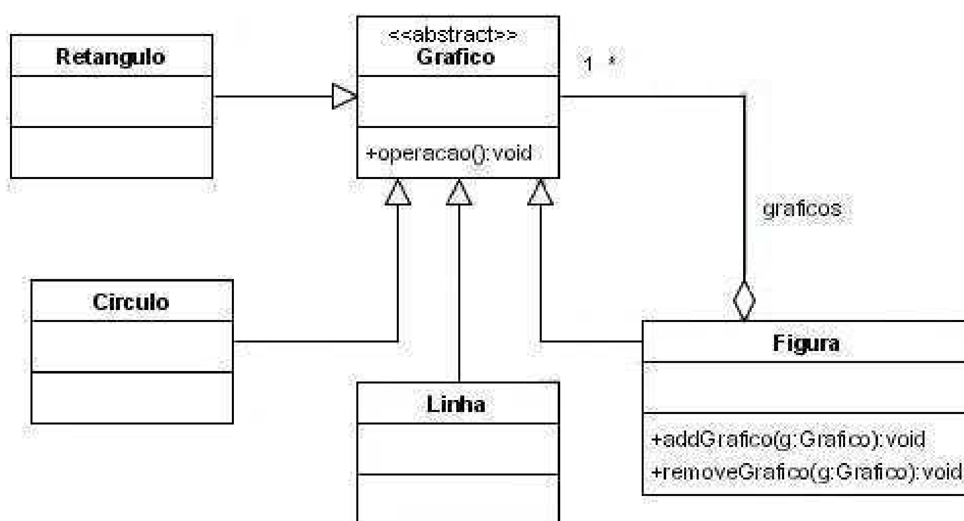

FIGURA 5.1: simplificado diagrama de classes para um editor de texto com figuras

O projetista acredita que ele projetou suas classes utilizando o padrão Composite, mas a ferramenta CASE que ele utilizou não pode lhe dar esta

certeza, pois atualmente, nenhuma ferramenta CASE é capaz de detectar a realização de padrões em diagramas UML (como mencionamos em capítulos anteriores). Então o projetista submete o diagrama ao SAMOA para checar a existência do padrão o qual ele acredita está utilizando. O SAMOA compara o diagrama fornecido com todos os padrões armazenados no seu repositório de padrões.

Mas para que o SAMOA faça o processo de detecção de padrões em diagramas UML, faz necessário que estes sejam exportados no formato XMI (vimos que hoje em dia, a maioria das ferramentas CASEs possuem mecanismo para exportação nesse formato). A figura a seguir ilustra um pequeno trecho do XMI do diagrama apresentado na Figura 5.1. Ele foi gerado pela ferramenta *Poseidon* [Poseidon - 2003]. A definição completa desse XMI gerado encontra-se no anexo.

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Wed
Nov 12 22:15:42 GMT-03:00 2003'>
    ....
    <UML:Class xmi.id = 'lsm:161b9b2:f8d2a07169:-7ff9' name = 'Grafico' visibility =
'public'  isSpecification = 'false' isRoot = 'false' isAbstract = 'true' isActive = 'false'>
        <UML:Classifier.feature>
          <UML:Operation xmi.id = 'lsm:161b9b2:f8d2a07169:-7ff8' name = 'operacao'
            visibility = 'public' isSpecification = 'false' ownerScope = 'instance'
            isQuery = 'false' concurrency = 'sequential' isRoot = 'false' isLeaf = 'false'
            isAbstract = 'true'>
      ....
       </UML:Class>
       <UML:Class xmi.id = 'lsm:161b9b2:f8d2a07169:-7ff0' name = 'Retangulo' visibility
= 'public'  isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'
        isActive = 'false'>
        ...
       </UML:Class>
     ...
  </XMI>
```

FIGURA 5.2: XMI representando o diagrama da Figura 5.1

Feito isto, basta fornecer este XMI ao SAMOA.

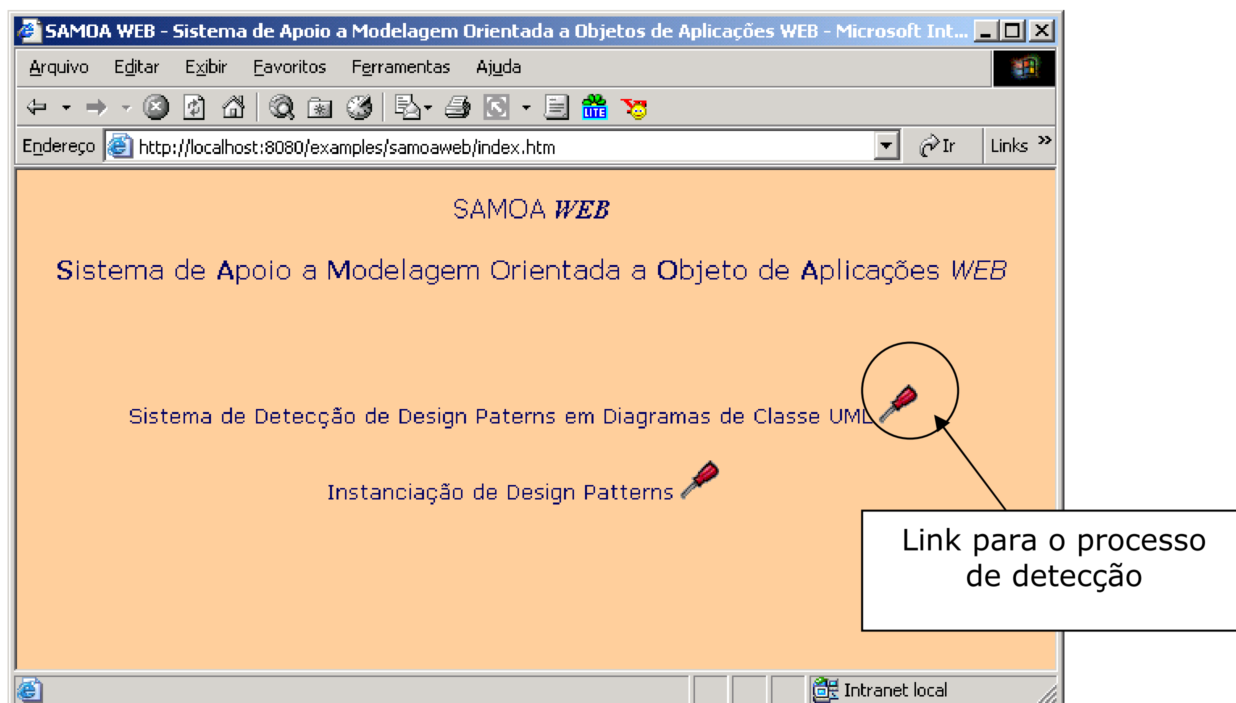

FIGURA 5.3: página principal do SAMOA

Após clicar no link ,que requisita ao sistema o processo de detecção de padrões, o sistema abrirá a seguinte página:

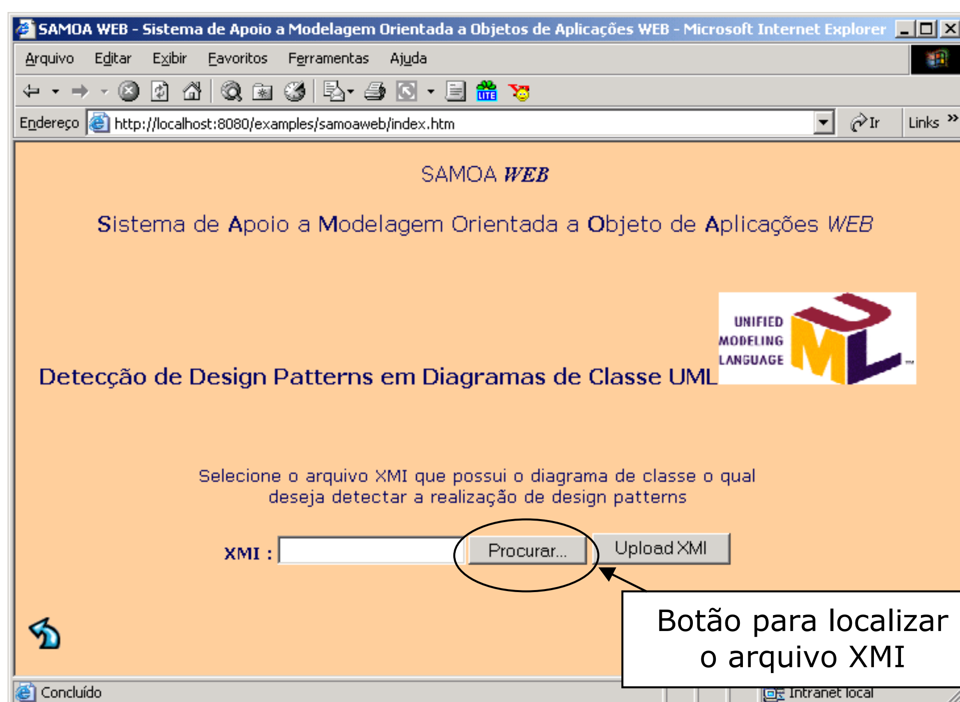

FIGURA 5.4: página para capturar o arquivo XMI

Após clicar no botão, o seguinte caixa de diálogo se abrirá, com intuito do usuário localizar o arquivo XMI.

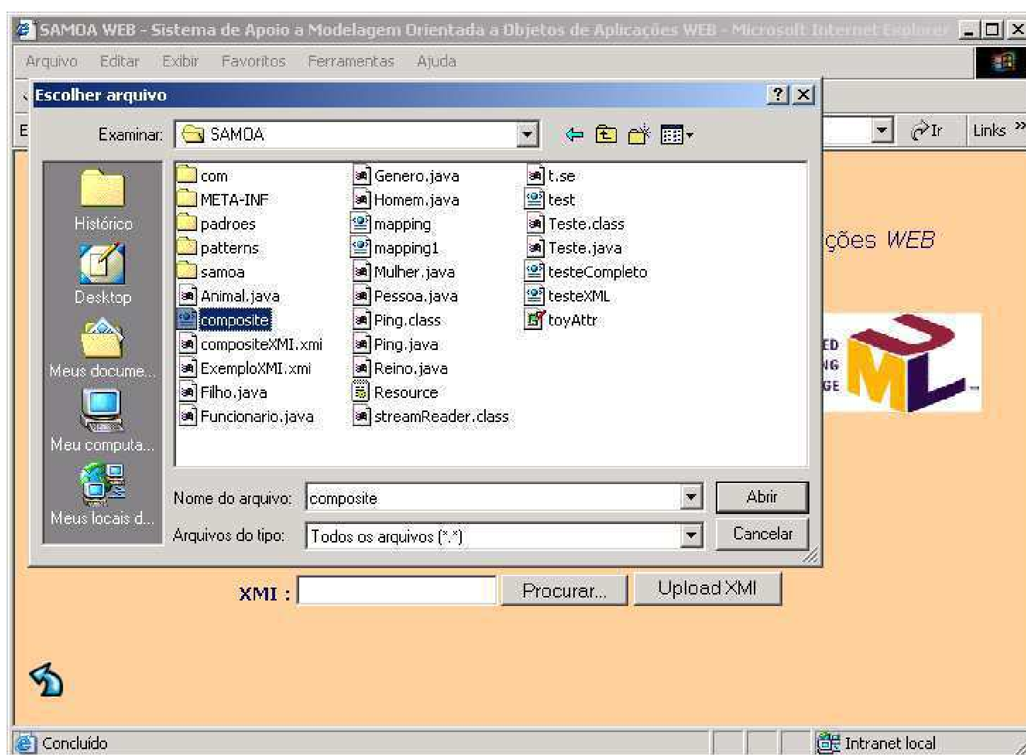

FIGURA 5.5: buscando o XMI para upload

Após o usuário selecionar o arquivo XMI, basta agora ele clicar no botão para fazer upload do arquivo e iniciar o processo de detecção de padrões.

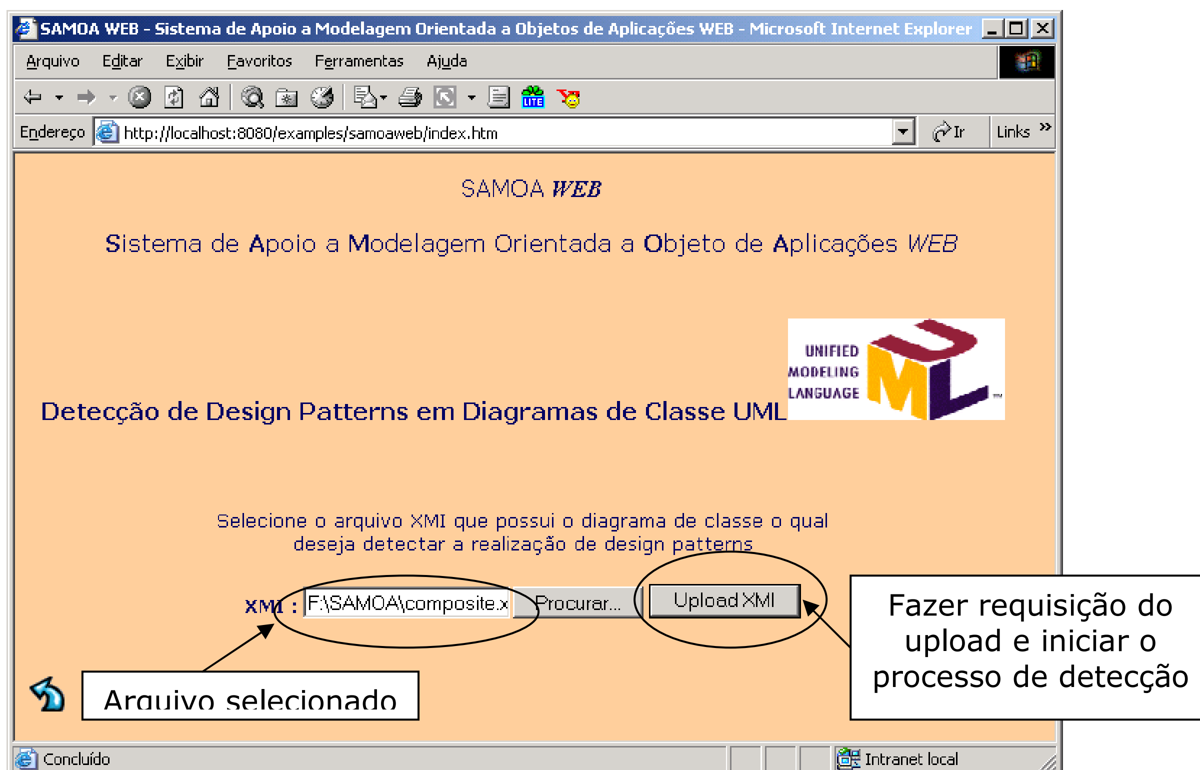

FIGURA 5.6: fazendo upload do XMI

Começa, então, o processo de detecção de padrões como mencionado no capítulo 3. O SAMOA procurará pela realização ou alguma aproximação de padrões, no XMI fornecido. Em seguida o sistema retornará um arquivo XML informando os padrões detectados e os papeis que cada classe ou interface desempenha. Só que não exibiremos o XML gerado. Colocamos uma interface por cima desse arquivo para uma melhor visualização pelo usuário, porém um link para o XML é gerado caso o ele queira visualiza-lo. Ver Figura 5.7.

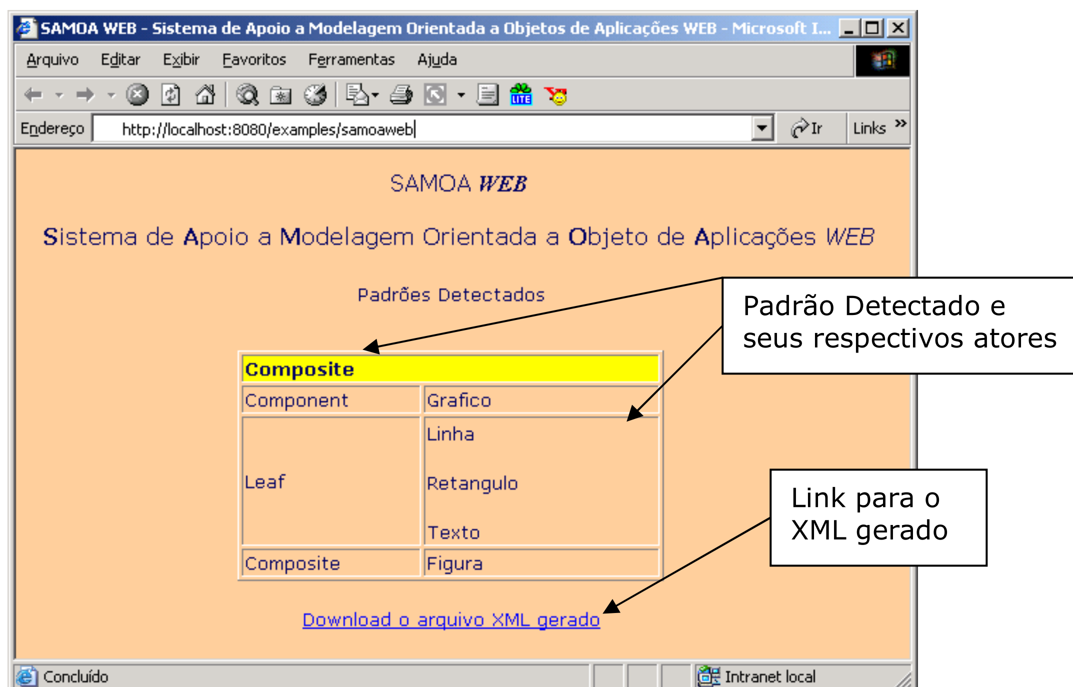

FIGURA 5.7: visualização do padrão detectado

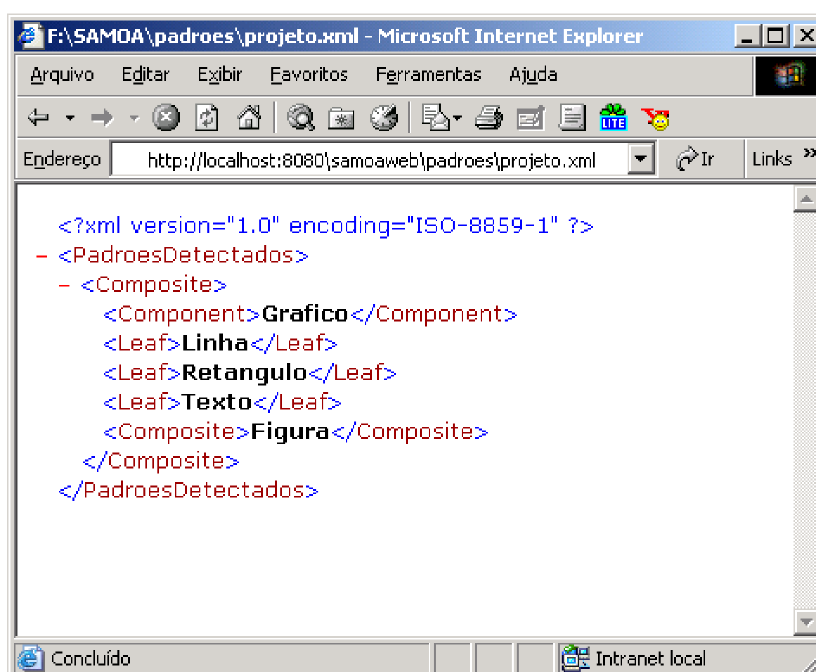

FIGURA 5.8: conteúdo do XML gerado

A figura acima ilustra, através de tags, o padrão Composite [Gamma et al. – 2000] detectado e os papeis exercidos por cada classe. Caso nenhum padrão tivesse sido detectado, o sistema informaria apenas uma mensagem informando a não detecção de padrões.

Instanciação de Padrões

O usuário do SAMOA não só pode se beneficiar do processo de detecção de padrões, mas pode instanciá-los. Suponha que um determinado usuário decidiu utilizar o padrão *Proxy* [Gamma et al. - 2000] no seu contexto específico. Ela já sabe quem são os participantes desse padrão, mas gostaria de fornecer apenas nomes para participantes e que o sistema gerasse o código fonte equivalente em *Java*, para esse padrão específico. Isso já com os relacionamentos e atributos necessários embutidos no fonte gerado (com todo o *template* da estrutura do padrão requerido).

Observe as seguintes ilustrações:

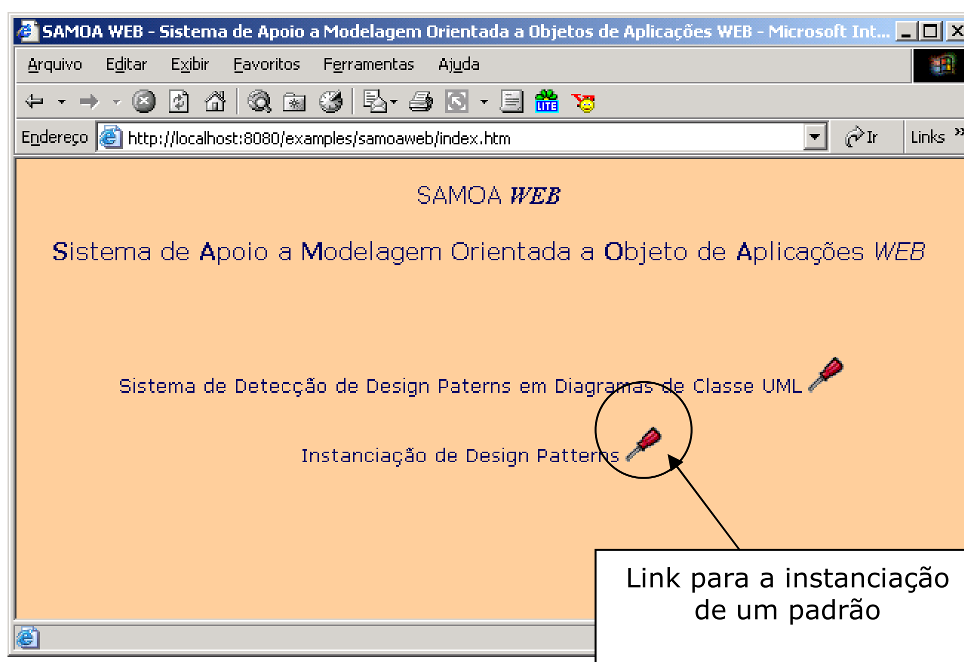

FIGURA 5.9: link para instanciar padrões

Após clicar no link para instanciar padrões, o sistema requisitará que seja exibida uma lista dos padrões cadastrados.

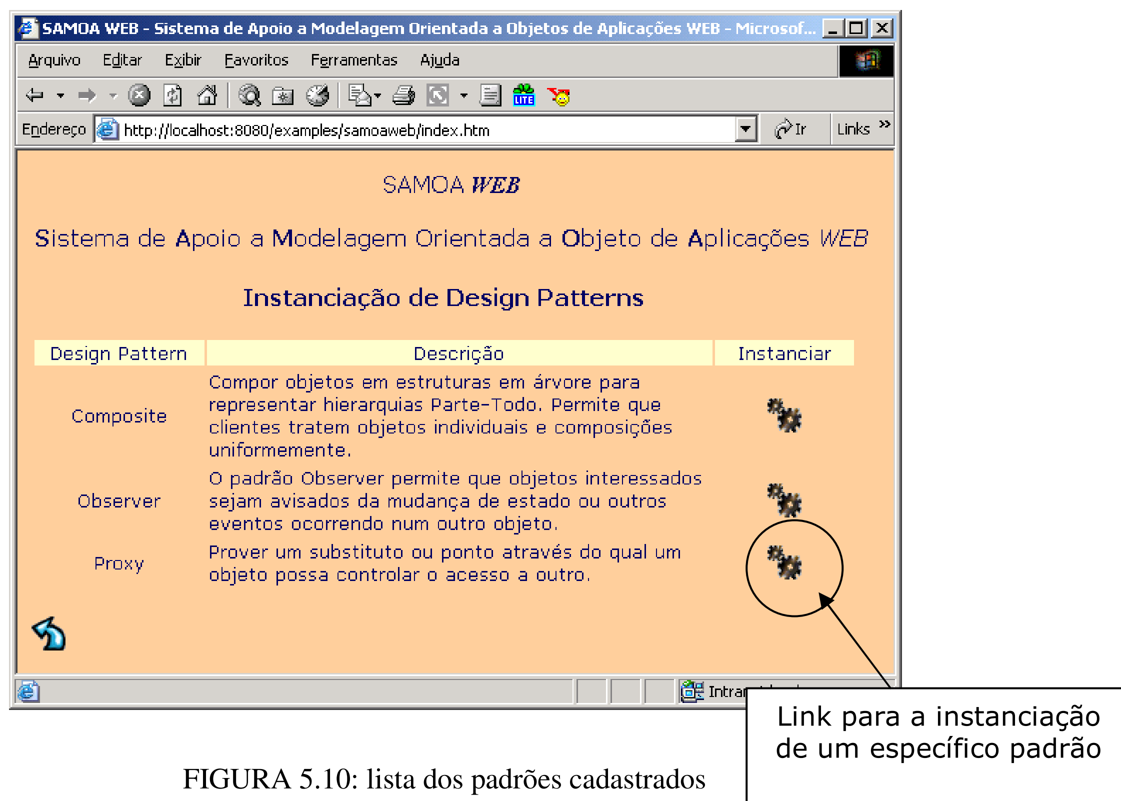

FIGURA 5.10: lista dos padrões cadastrados

Caberá então ao usuário clicar no link para instanciar o padrão Proxy. Depois uma página com um formulário, dinamicamente construído, será exibida para que este seja preenchido.

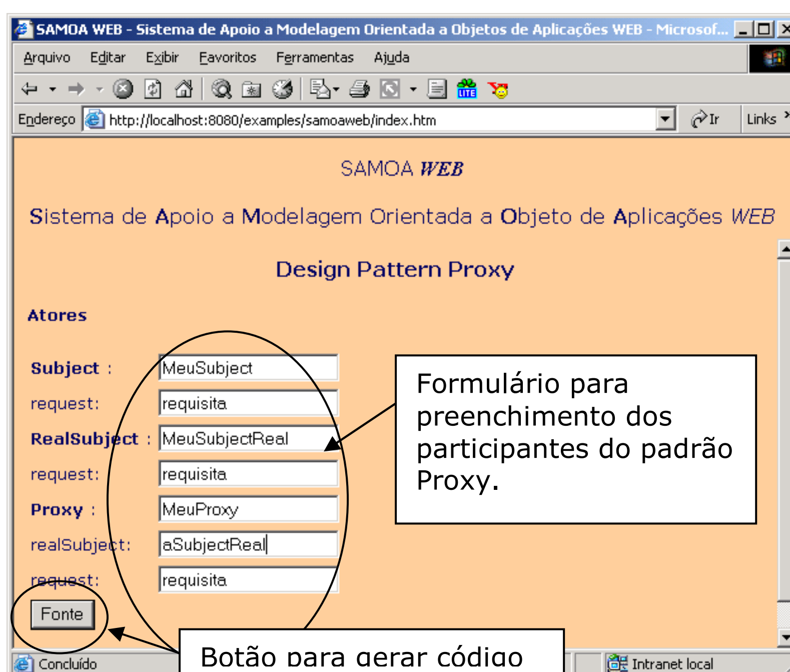

FIGURA 5.11: fornecendo dados para o formulário

Em seguida, basta clicar no botão “Fonte”, para que o sistema gere o código fonte em Java. Os arquivos são gerados, compactados no formato *zip* e disponibilizados para download. Veja a figura a seguir:

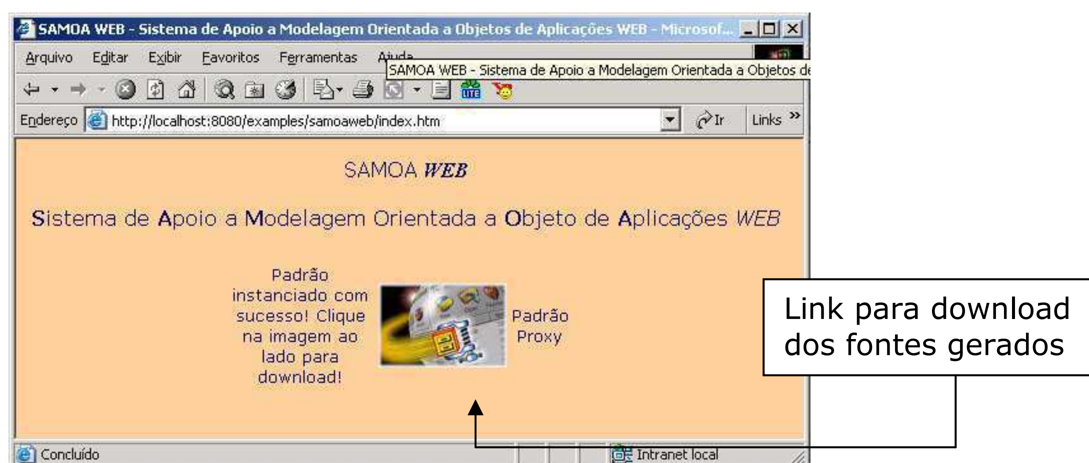

FIGURA 5.12: disponibilizando fontes Java

Veremos a abaixo, os fontes gerados:

```
//MeuSubject.java
public class MeuSubject {
    public void requisita() {
    }
}
//MeuProxy.java
public class MeuProxy extends MeuSubject {
    // Association: aSubjectReal
    private MeuSubjectReal aSubjectReal =
        null;
    public MeuSubjectReal getMeuSubjectReal() {
        return aSubjectReal;
    }
    public void setMeuSubjectReal(MeuSubjectReal aMeuSubjectReal) {
        aSubjectReal = aMeuSubjectReal;
    }
    // Method linked to: aSubjectReal
    public void requisita() {
        aSubjectReal.requisita();
    }
}
//MeuSubjectReal.java
public class MeuSubjectReal extends MeuSubject {
    public void requisita() {
    }
}
```

FIGURA 5.13: arquivos fontes gerados a partir dos dados do formulário

Geração de Críticas

Embora não tenhamos implementado os sistema de críticas, ilustraremos aqui como ele se comportaria no processo de detecção de padrões de projeto.

Continuando o caso de detecção de padrões em diagramas UML, vimos que o sistema gera um arquivo XML contento os padrões detectados e seus participantes (Figura 5.8). Esse arquivo seria passado ao sistema de críticas para que este proponha melhorias ao diagrama editado (Figura 5.1).

Vimos que o sistema acusou o padrão *Composite* no diagrama da Figura 5.1. Então, o sistema de críticas irá criticar o modelo submetido ao processo de detecção e irá procurar sugerir melhoras em tal diagrama. No caso do diagrama da Figura 5.1, as críticas poderiam ser:

- inserir o método getGraficos para retornar os componentes do *Composite*;
- Considerar o uso do padrão *Iterator* para acessar aos objetos gráfico em figura;
- em Gráfico deverão ser definidas tantas operações para Figura, Retângulo, Linha e Texto quanto possíveis.
- não colocar referencia para os filhos em Gráfico.

Conclusão

Foram descritos nesse capítulo todos os aspectos envolvidos com o projeto da ferramenta SAMOA. Mostramos, basicamente, o que é o SAMOA e a quais funcionalidades ele se propõe a atender.

Fizemos uma abordagem geral da ferramenta, através da ilustração e descrição de sua arquitetura, detalhando-a, posteriormente, para descrever a forma como a mesma foi concebida, quais os seus propósitos e quais as funcionalidades que ela se propunha a disponibilizar. Concluímos com uma descrição de como as funcionalidades oferecidas pelo SAMOA são disponibilizadas aos seus usuários. Utilizamos para tanto um estudo de caso, onde o SAMOA foi usado para detectar e instanciar padrões.

Capítulo 6 - Conclusões

O uso de padrões de projeto facilita a compreensão dos sistemas existentes. Observa-se que os padrões fornecem desde soluções a problemas comuns a uma base para a compreensão de grandes sistemas orientados a objetos.

O público alvo do estudo de padrões de projeto engloba desde pessoas que estão aprendendo a programação orientada a objetos e que irão adquirir um conhecimento maior sobre os fluxos de controle e herança envolvidos na construção de programas, até profissionais da área que irão refinar seus projetos através do uso destes padrões.

Através de padrões de projeto é possível identificar os pontos comuns entre duas soluções diferentes para um mesmo problema. Conhecer esses pontos comuns nos permite desenvolver soluções cada vez melhores e mais eficientes que podem ser reutilizadas.

Então, fazer uso de padrões de projeto é fundamental para a produção de software de qualidade. E uma das técnicas para melhorar um projeto existente é identificar todas as realizações de padrões, para a aplicação de regras para melhorá-las. Tal técnica visa encontrar todas as realizações de padrões de projeto empregadas num projeto, sendo esta tarefa tediosa para o engenheiro de

software. Esta pesquisa trata de uma ferramenta de assistência automatizada, para programadores e arquitetos de software para detectar a realização de padrões tanto em diagramas de classe UML quanto em fontes Java.

No processo de detecção em fontes Java serve para extrair um conhecimento maior a fim de documentação sobre a realização de padrões em um conjunto implementado de classes. No caso da detecção em diagramas UML, serve para checar a coerência entre o esperado pelo usuário e a abstração de um padrão. A discrepância entre essas duas parcelas resultará em:

- o sistema poderá detectar um padrão, no diagrama, e o projetista não percebeu. Isso lhe dará uma melhor percepção do uso de padrões, ou;
- o sistema não detectar a realização de um padrão, embora o projetista acredite ter utilizado. Isso servirá para atentar ao projetista que existe alguma incoerência no projeto para que ele tome medidas corretivas, até que seu padrão seja detectado pelo sistema. Em sua versão atual, o SAMOA ainda não é capaz de informar onde se encontra essa inconsistência, apenas detectar padrões corretamente definidos.

Projetamos o SAMOA para prover críticas sobre modelos UML, especificamente diagramas de classe ou arquivos fontes *Java*, uma vez detectadas as realizações de padrões. Neste caso, o sistema auxilia com sugestões de projeto para a melhoria do uso de padrões. Seriam sugestões com relação a nomenclaturas, visibilidade de métodos, atributos etc.. No momento, propomos a arquitetura para esta funcionalidade específica e exemplificamos algumas sugestões que podem ser realizadas. Deixamos aqui a implementação desse módulo para trabalhos futuros, assim como a implementação de uma interface para cadastrar essas sugestões e novos padrões no SAMOA.

O SAMOA, tendo a capacidade de instanciar padrões de projeto, dotou-se da tarefa de adaptar e implementar um padrão em um contexto particular. E

procurando facilitar o trabalho do desenvolvedor, o sistema propõe uma interface baseada em um formulário dinâmico construído de acordo com as informações necessárias à instanciação de um padrão específico, para um contexto específico (números de atores e seus nomes, relacionamentos, cardinalidade, etc.).

Atualmente, o SAMOA está limitado a reconhecer um subconjunto dos padrões do GoF [Gamma et al. - 2000], padrões os quais definimos como *"detectáveis"*, em capítulos anteriores. Para detectar os outros padrões seria necessário um estudo mais aprofundado de padrões, levando em consideração suas colaborações e não só suas formas estruturais.

Referências Bibliográficas

[Alexander - 1977] Alexander, C. et al., " *Pattern Language"* Oxford University Press, New York, 1977.

[Amiot et al. - 2001] Amiot, H. et al., *"Meta-modeling Design Patterns: Application to pattern detection and code synthesis"*. ECOOP'2001.

[ArgoUML – 2003] ArgoUML, 2003. Homepage: http://argouml.tigris.org/, acessado em 04/2003.

[Booch - 1994] Booch, G. "*Object-Oriented Analysis and Design with Application"*. Benjamin/Cummings, 1994.

[Booch et al. - 2002] Booch, G., Rumbaugh, J. e Jacobson, I.. *"UML: Guia do Usuário"*. Editora Campus, 2002.

[Brown - 1997] Brown, K.. *"Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk"*. Dissertação de Mestrado. Universidade de Illinois, 1997.

[Buschmann et al. - 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad P. and Stal, M.. *"A System of Patterns – Patterns Oriented Software Architecture"*. Wiley, 1996.

[Eriksson et al. - 1998] Eriksson, H. E. ; Penker, M. "*UML Toolkit*". United States of America, JOHN WILEY & SONS, 1998.

[Furlan - 1998] Furlan, José Davi. "*Modelagem de Objetos através da UML.*". São Paulo, Makron Books, 1998.

[Gamma et al. - 2000] Gamma, Erich et al., *"Design Patterns: Elements of Reusable Object Oriented Software".* Addison-Wesley, Massachusetts, 2000.

[Heuzeroth - 2003] Heuzeroth, D., Holl, T., Högström, G. and Löwe, W.. *"Automatic Design Pattern Detection".* Universidade de Karlsruhe, Alemanha, 2003.

[Heyworth - 1996] Heyworth, James, *"Introduction to Design Patterns in Delphi".D*isponível:http://www.obsof.com/dephi_tips/pattern.html. Acessado em 06/2003.

[Jacobson – 1992] Jacobson, I. "*Object-Oriented Software Engineering.*" Addison-Wesley, 1992.

[JSP - Java] Informações sobre Java Server Pages (*JSP)* disponível em: http://java.sun.com/products/jsp/

[JUnit Java] Informações sobre o framework *JUnit* disponível em: http://junit.sourceforge.net/

[Keller et al. - 1999] Keller, R., Schauer, R., Robitaille, S. and Page, P.. *"Pattern-Based Reverse-Engineering of Design Components"*, ISCE, pág. 226-235, 1999.

[Keller et al. - 2000] Keller, Rudolf K., Schauer, Reinhard, Denis, Guy. Selecting a model interchange format: the SPOOL case study In: Proceedings of the Thirty-Thrid Annual Hawaian International Conference on System Sciences, January 2000.

[Kifer et al. - 1995] Kifer, M., Lausen, G. and Wu, J. *"Logical Foundations of Oriented-Objected and Frame-Based Languages"*. Jornal of Association for Computing Machinery, 1995.

[Kifer et al. – 1995] Kifer, M., et al..*"Logical Foundations of Oriented-Objected and Frame-Based Languages"*. Jornal of Association for Computing Machinery, 1995.

[Larman - 1998] Larman, C.. *"Applying UML and Patterns"*. Prentic-Hall, USA, 1998.

[Meijers - 1997] Meijers, M.. *"Tools Support for Object-Oriented Patterns"*. Tese de Doutorado, Universidade de Utrecht, 1997.

[MOF - OMG] Meta Object Format. Url: http://www.omg.org

[Muller - 1997] Muller, P. A. "*Instant UML"*. Canada, Wrox Press Ltd.,1997.

[Ontogenics] Repository Translation Service. Homepage: http://www.ontogenics.com/repository/translator

[PatternsBox] PatternsBox. Disponível em http://www.emn.fr/albin.

[Poseidon - 2003] Poseidon for UML, Maio 2003. Homepage: http://www.gentleware.com/

[Prechelt et al. - 1998] Prechelt, L. and Krämer, C.. *"Functionality versus Practicality: Employing Existing Tools for Recovering Strutural Design Pattern"*, J. UCS, 1998.

[Rational et al. – 1997a] Rational software, "*UML Notation Guide"*. version 1.1, september/1997a.

[Rational et al. – 1997b] Rational software, "*UML Summary"*. version 1.1, september/1997b.

[Rich et al. – 1990] Rich, C., et al. . Recognizing a Program's Design: A Graph-Parsing Approach. IEEE Software, Vol. 7, No. 1, 1990, 82-89.

[Robbins – 2000] Robbins, J.. *"Design Critiquing Systems"*. Disponível em http://www.ics.uci.edu, 2000.

[RoseXMI] Plug-in Rose XMI disponível em http://zuse.esnig.cifom.ch/analyse/rationalrose/xmi/xmi_addin.htm

[Rumbaugh – 1991] Rumbaugh, J. *"Object-Oriented Modeling and Design.*" Prentice Hall, 1991.

[Salingaros - 1999] Salingaros, Nikos A., The Structure of Pattern Languages. Disponível:www.math.utsa.edu/sphere/salingar/StructurePattern.html, 1999.

[Shalloway et al. - 2000] Shalloway, Alan, et al., Pattern Oriented Design: Using Design Patterns From Analysis to Implementation, Net

Objectives,2000.Dissponível:www.netobjectives.com/download/what_are_design_patterns.pdf

[Somerville - 1996] Somerville, I.. *"Software Engineering".* Addison-Wesley, 1996.

[SunJava] Site oficial da linguagem de programação Java, disponível no site http://java.sun.com/

[Sunyè – 1999] Sunyé, G.; Génération de code à l´aide de patrons de conception; LMO´99, pp. 163-178, 1999. Na França.

[UML – 2003] Unified Modeling Language 1.4 specification . Available at site Object Management Group. URL: http://www.omg.org/technology/documents/formal/uml.htm. Consulted in February, 2003.

[Unisys] Unisys Universal Repository. Homepage: http://www.unisys.com/marketplace/urep

[UnisysXMI] Unisys News Release: "Unisys Demonstrates XMI Interoperability Between UREP Universal Repository and Microsoft Repository". Url: http://www.unisys.com/news/releases/1998/dec/12076622.html

[Winter et al. - 1996] Winter, M, et al., Towards Pattern-Based Tools. EuropLop´96, 1996.

[XMI – 2003] XML Metadata Interchange (XMI) – Version 1.2. Disponível no site da OMG. URL:

http://www.omg.org/technology/documents/formal/xmi.htm - Consulted in February, 2003.

[XMI - IBM] Homepage do XMI Toolkit 1.2. Url: http://www.alphaworks.ibm.com/tech/xmitoolkit

[XML - OMG] XML Metadata Interchange 1.1. Url: http://www.omg.org/

[XML – W3C] eXtended Markup Language 1.1. Url: http://www.w3c.org/XML

[XML Schema] XML Schema Home Page. Homepage: http://www.w3c.org/XML/Schema

[XML4SE] Homepage do grupo de pesquisa "XML for Software Engineering".Url:http://www.iro.umontreal.ca/labs/gelo/xml4se/en/docs.html

Anexo

Este anexo mostra o conteúdo, completo, de um arquivo XMI gerado pela ferramenta Poseidon, referente ao diagrama de classes exibido na figura 5.2.


```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp
= 'Wed Nov 12 22:15:42 GMT-03:00 2003'>
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>Netbeans XMI Writer</XMI.exporter>
      <XMI.exporterVersion>1.0</XMI.exporterVersion>
    </XMI.documentation>
  </XMI.header>
  <XMI.content>
    <UML:Model xmi.id = 'lsm:161b9b2:f8d2a07169:-7ffa' name = 'model 1'
isSpecification = 'false'
      isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
      <UML:Namespace.ownedElement>
        <UML:Class xmi.id = 'lsm:161b9b2:f8d2a07169:-7ff9' name = 'Grafico'
visibility = 'public'
          isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
isAbstract = 'true'
          isActive = 'false'>
          <UML:Classifier.feature>
            <UML:Operation xmi.id = 'lsm:161b9b2:f8d2a07169:-7ff8' name =
'operacao'
              visibility = 'public' isSpecification = 'false' ownerScope =
'instance'
              isQuery = 'false' concurrency = 'sequential' isRoot = 'false'
isLeaf = 'false'
              isAbstract = 'true'>
              <UML:BehavioralFeature.parameter>
                <UML:Parameter xmi.id = 'lsm:161b9b2:f8d2a07169:-7ff7' name
= 'return' isSpecification = 'false'
                  kind = 'return'/>
              </UML:BehavioralFeature.parameter>
            </UML:Operation>
            <UML:Method xmi.id = 'lsm:161b9b2:f8d2a07169:-7ff5'
isSpecification = 'false'
```

```
              isQuery = 'false'>
              <UML:Method.body>
                <UML:ProcedureExpression xmi.id = 'lsm:161b9b2:f8d2a07169:-
7ff4' language = 'java'
                  body = ''/>
              </UML:Method.body>
              <UML:Method.specification>
                <UML:Operation xmi.idref = 'lsm:161b9b2:f8d2a07169:-7ff8'/>
              </UML:Method.specification>
            </UML:Method>
          </UML:Classifier.feature>
        </UML:Class>
        <UML:Class xmi.id = 'lsm:161b9b2:f8d2a07169:-7ff0' name =
'Retangulo' visibility = 'public'
          isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
isAbstract = 'false'
          isActive = 'false'>
          <UML:GeneralizableElement.generalization>
            <UML:Generalization xmi.idref = 'lsm:161b9b2:f8d2a07169:-
7fef'/>
          </UML:GeneralizableElement.generalization>
        </UML:Class>
        <UML:Class xmi.id = 'lsm:161b9b2:f8d2a07169:-7fee' name = 'Texto'
visibility = 'public'
          isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
isAbstract = 'false'
          isActive = 'false'>
          <UML:GeneralizableElement.generalization>
            <UML:Generalization xmi.idref = 'lsm:161b9b2:f8d2a07169:-
7fed'/>
          </UML:GeneralizableElement.generalization>

        </UML:Class>
        <UML:Class xmi.id = 'lsm:161b9b2:f8d2a07169:-7fec' name = 'Linha'
visibility = 'public'
          isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
isAbstract = 'false'
          isActive = 'false'>
          <UML:GeneralizableElement.generalization>
            <UML:Generalization xmi.idref = 'lsm:161b9b2:f8d2a07169:-
7feb'/>
          </UML:GeneralizableElement.generalization>
        </UML:Class>
        <UML:Stereotype xmi.id = 'lsm:161b9b2:f8d2a07169:-7fea' name =
'realize'
          isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
isAbstract = 'false'>
          <UML:Stereotype.baseClass>Abstraction</UML:Stereotype.baseClass>
        </UML:Stereotype>
        <UML:Generalization xmi.id = 'lsm:161b9b2:f8d2a07169:-7fef'
isSpecification = 'false'>
          <UML:Generalization.child>
            <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7ff0'/>
          </UML:Generalization.child>
          <UML:Generalization.parent>
            <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7ff9'/>
          </UML:Generalization.parent>
        </UML:Generalization>
```

```
        <UML:Generalization xmi.id = 'lsm:161b9b2:f8d2a07169:-7fed'
isSpecification = 'false'>
          <UML:Generalization.child>
            <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7fee'/>
          </UML:Generalization.child>
          <UML:Generalization.parent>
            <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7ff9'/>
          </UML:Generalization.parent>
        </UML:Generalization>
        <UML:Generalization xmi.id = 'lsm:161b9b2:f8d2a07169:-7feb'
isSpecification = 'false'>
          <UML:Generalization.child>
            <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7fec'/>
          </UML:Generalization.child>
          <UML:Generalization.parent>
            <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7ff9'/>
          </UML:Generalization.parent>
        </UML:Generalization>
        <UML:Class xmi.id = 'lsm:161b9b2:f8d2a07169:-7fe9' name = 'Figura'
visibility = 'public'
          isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
isAbstract = 'false'
          isActive = 'false'>
          <UML:GeneralizableElement.generalization>
            <UML:Generalization xmi.idref = 'lsm:161b9b2:f8d2a07169:-7fe8'/>
          </UML:GeneralizableElement.generalization>
          <UML:Classifier.feature>
            <UML:Operation xmi.id = 'lsm:161b9b2:f8d2a07169:-7fe7' name =
'addGrafico'
              visibility = 'public' isSpecification = 'false' ownerScope =
'instance'
              isQuery = 'false' concurrency = 'sequential' isRoot = 'false'
isLeaf = 'false'
              isAbstract = 'false'>
              <UML:BehavioralFeature.parameter>
                <UML:Parameter xmi.id = 'lsm:161b9b2:f8d2a07169:-7fe6' name
= 'g' isSpecification = 'false'
                  kind = 'in'>
                  <UML:Parameter.type>
                    <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7ff9'/>
                  </UML:Parameter.type>
                </UML:Parameter>
                <UML:Parameter xmi.id = 'lsm:161b9b2:f8d2a07169:-7fe5' name
= 'return' isSpecification = 'false'
                  kind = 'return'/>
              </UML:BehavioralFeature.parameter>
            </UML:Operation>
            <UML:Method xmi.id = 'lsm:161b9b2:f8d2a07169:-7fe4'
isSpecification = 'false'
              isQuery = 'false'>
              <UML:Method.body>
                <UML:ProcedureExpression xmi.id = 'lsm:161b9b2:f8d2a07169:-
7fe3' language = 'java'
                  body = ''/>
              </UML:Method.body>
              <UML:Method.specification>
                <UML:Operation xmi.idref = 'lsm:161b9b2:f8d2a07169:-7fe7'/>
```

```
              </UML:Method.specification>
            </UML:Method>
            <UML:Operation xmi.id = 'lsm:161b9b2:f8d2a07169:-7fe2' name =
'removeGrafico'
              visibility = 'public' isSpecification = 'false' ownerScope =
'instance'
              isQuery = 'false' concurrency = 'sequential' isRoot = 'false'
isLeaf = 'false'
              isAbstract = 'false'>
              <UML:BehavioralFeature.parameter>
                <UML:Parameter xmi.id = 'lsm:161b9b2:f8d2a07169:-7fe1' name
= 'g' isSpecification = 'false'
                  kind = 'in'>
                  <UML:Parameter.type>
                    <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7ff9'/>
                  </UML:Parameter.type>
                </UML:Parameter>
                <UML:Parameter xmi.id = 'lsm:161b9b2:f8d2a07169:-7fe0' name
= 'return' isSpecification = 'false'
                  kind = 'return'/>
              </UML:BehavioralFeature.parameter>
            </UML:Operation>
            <UML:Method xmi.id = 'lsm:161b9b2:f8d2a07169:-7fdf'
isSpecification = 'false'
              isQuery = 'false'>
              <UML:Method.body>
                <UML:ProcedureExpression xmi.id = 'lsm:161b9b2:f8d2a07169:-
7fde' language = 'java'
                  body = ''/>
              </UML:Method.body>
              <UML:Method.specification>
                <UML:Operation xmi.idref = 'lsm:161b9b2:f8d2a07169:-7fe2'/>
              </UML:Method.specification>
            </UML:Method>
          </UML:Classifier.feature>
        </UML:Class>
        <UML:Generalization xmi.id = 'lsm:161b9b2:f8d2a07169:-7fe8'
isSpecification = 'false'>
          <UML:Generalization.child>
            <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7fe9'/>
          </UML:Generalization.child>
          <UML:Generalization.parent>
            <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7ff9'/>
          </UML:Generalization.parent>
        </UML:Generalization>
        <UML:Association xmi.id = 'lsm:161b9b2:f8d2a07169:-7fdd' name =
'filhos'
          isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
isAbstract = 'false'>
          <UML:Association.connection>
            <UML:AssociationEnd xmi.id = 'lsm:161b9b2:f8d2a07169:-7fdc'
visibility = 'public'
              isSpecification = 'false' isNavigable = 'true' ordering =
'unordered' aggregation = 'aggregate'
              targetScope = 'instance' changeability = 'changeable'>
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity xmi.id = 'lsm:161b9b2:f8d2a07169:-7fdb'>
                  <UML:Multiplicity.range>
```

```
                    <UML:MultiplicityRange xmi.id =
'lsm:161b9b2:f8d2a07169:-7fda' lower = '1'
                      upper = '1'/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
              <UML:AssociationEnd.participant>
                <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7fe9'/>
              </UML:AssociationEnd.participant>
            </UML:AssociationEnd>
            <UML:AssociationEnd xmi.id = 'lsm:161b9b2:f8d2a07169:-7fd9'
visibility = 'public'
              isSpecification = 'false' isNavigable = 'true' ordering =
'unordered' aggregation = 'none'
              targetScope = 'instance' changeability = 'changeable'>
              <UML:AssociationEnd.multiplicity>
                <UML:Multiplicity xmi.id = 'lsm:161b9b2:f8d2a07169:-7fd8'>
                  <UML:Multiplicity.range>
                    <UML:MultiplicityRange xmi.id =
'lsm:161b9b2:f8d2a07169:-7fd7' lower = '0'
                      upper = '-1'/>
                  </UML:Multiplicity.range>
                </UML:Multiplicity>
              </UML:AssociationEnd.multiplicity>
              <UML:AssociationEnd.participant>
                <UML:Class xmi.idref = 'lsm:161b9b2:f8d2a07169:-7ff9'/>
              </UML:AssociationEnd.participant>
            </UML:AssociationEnd>
          </UML:Association.connection>
        </UML:Association>
      </UML:Namespace.ownedElement>
    </UML:Model>
  </XMI.content>
</XMI>
```