UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

ReacT-MDD – Rastreabilidade Reactiva no Desenvolvimento de Sistemas de Informação

Marco Bruno Correia Costa

(Mestre)

Dissertação para o Grau de

Doutor em Engenharia Informática e de Computadores

DOCUMENTO PROVISÓRIO

Março de 2010

Dissertação orientada por:

Prof. Doutor Alberto Rodrigues da Silva

Professor Associado no

Departamento de Engenharia Informática do

Instituto Superior Técnico

Lisboa, 2 de Março de 2010

Resumo

A aproximação emergente MDD (*Model Driven Development*) preconiza o desenvolvimento de código a partir de modelos, eventualmente descritos pela linguagem UML. Tanto o código produzido como todos os restantes artefactos são parte integrante do projecto de desenvolvimento, operação e manutenção da aplicação.

É necessário um mecanismo que permita a rastreabilidade de todos os artefactos ao longo de todas as fases do projecto, independentemente do seu nível conceptual. As relações que tornam os diferentes artefactos coerentes têm de ser mantidas, preferencialmente de forma automática.

O modelo QVT (Queries-Views-Tranformations) define transformações entre modelos, usando metamodelação. Propõe-se neste trabalho de investigação um modelo complementar ao QVT, com os aspectos considerados necessários à *rastreabilidade reactiva* de artefactos de forma genérica. Com esta abordagem promove-se a alteração automática de artefactos documentais a partir de alterações ao código e vice-versa.

Palavras-chave:

UML, MDD, QVT, Rastreabilidade, Modelação, ReacT-MDD

Abstract

The MDD (*Model Driven Development*) approach, favors the development of code starting at models, eventually described by the UML language. Not only the produced code, but also the remaining artefacts are part of the solution, at the development, operation and maintenance stages.

However, it is necessary a mechanism that allows traceability between all artefacts during all project stages, not regarding its conceptual level. The artefact coherence relations must be maintained and this task should be automatically accomplished, as possible.

The QVT (Queries-Views-Tranformations) model defines model transformations, using metamodelling. It is proposed, in this research work, a complementar model to QVT, dealing several issues about *reactive traceability,* which is extensivelly defined. This approach takes QVT as a starting point to accomplish the construction of automated transformations between models and implements a way of maintaining traces between artefacts (models and code) as well as reacting to changes for the sake of system coherence.

Keywords:

UML, MDD, QVT, Traceability, Modelling, ReacT-MDD

Para a Rosinda,

Agradecimentos

Ao Prof. Doutor Alberto Rodrigues da Silva este trabalho deve a sua tão incansável como sábia orientação, para além da paciência e simpatia com as quais me honrou.

À Rosinda pelo amor e apoio inultrapassável que me deu durante todos estes anos.

Índice

1 Introdução

1.1 Enquadramento

O desenvolvimento dos sistemas de informação tem sofrido alterações tecnológicas e conceptuais profundas ao longo dos últimos cinquenta anos. As diversas evoluções de hardware incrementaram em várias ordens de grandeza o desempenho e a memória dos sistemas informáticos permitindo novas evoluções no campo do software. Por outro lado, o surgimento de diferentes paradigmas de programação e modelação levou a alterações metodológicas importantes no processo de desenvolvimento.

À medida que os sistemas de informação se foram tornando mais complexos e de maior dimensão, surgiu a necessidade de encontrar formas de os desenvolver sistematicamente. O aparecimento das metodologias de desenvolvimento de sistemas de informação tornou estes mais simples de produzir e manter, bem como reduziu os custos a médio e longo prazo, melhorando ainda a sua qualidade. Em geral, as tarefas que anteriormente se realizavam *ad-hoc,* passaram a ser desempenhadas através de um processo definido de desenvolvimento. As vantagens da utilização de muitas destas metodologias (e.g., sistematização, documentação, melhoria na qualidade e redução de custos a longo prazo) têm sido reconhecidas ao longo do tempo pela indústria.

A par da criação de novas metodologias foram concebidas ferramentas de apoio à modelação de sistemas, denominadas CASE *(Computer Aided Systems*[1] *Engineering)*. Durante a década de 1980 surgiram diversos produtos que, após um sucesso inicial, acabaram por não ter grande aceitação. O hardware parecia não acompanhar o desenvolvimento do software, tornando estas aplicações lentas e difíceis de utilizar. Actualmente as ferramentas CASE atingiram já a fase em que a sua utilização passou a ser reconhecida como importante, mas ainda não como essencial. Encontram-se num estádio que permite já passar à fase de geração automática de código [Herrington 2003] a partir de modelos, embora esta característica ainda não esteja vulgarizada.

[1] A sigla CASE é também descrita como Computer Aided Software Engineering. Embora exista um conjunto de ferramentas que estão próximas da especificação de código, sendo neste caso correcta a descrição anterior, existem também ferramentas centradas na modelação do negócio ou do contexto em que os programas funcionam. Nesse sentido é perfeitamente possível que estas ferramentas sirvam para especificar áreas do negócio que nem sequer estão automatizadas.

A aceitação gradual destas tecnologias pode ser ilustrada por diferentes estudos realizados nos últimos anos. Em 1992 [Nosek et al., 92] apenas 24% das empresas de sistemas de informação usavam ferramentas CASE. Em 2003, segundo um inquérito realizado a 183 empresas, na área do desenvolvimento de sistemas de informação [Welsh, 03], mais de metade (55%) utilizavam já ferramentas de modelação. Por outro lado, menos de um terço (31%) referiram que usavam capacidades de geração automática (que se limita normalmente à geração da estrutura das classes ou dos esquemas de bases de dados relacionais).

Entretanto, algumas das tecnologias envolvidas no processo de desenvolvimento chegaram já a um ponto de considerável maturidade. São exemplos destas tecnologias os sistemas de bases de dados relacionais, as interfaces gráficas, os compiladores, todos eles com mais de 30 anos de investigação e desenvolvimento. No entanto, a Engenharia Informática ainda está longe de atingir a maturidade relativa das Engenharia Civil ou da Engenharia Mecânica que têm uma história mais alargada. Não é de estranhar, por isso, que o erro, em maior ou menor escala, esteja presente em grande parte dos projectos informáticos actuais. Este facto leva a que devam ser utilizadas técnicas propiciadoras de uma acção correcta naquilo que diz respeito a todas as condicionantes de um projecto de sistemas de informação (e.g., tempo, custo, qualidade, risco) de forma a conseguir resultados satisfatórios.

Neste trabalho são abordadas técnicas de Engenharia, mas são naturalmente reconhecidos outros tipos de técnicas igualmente importantes para que um projecto possa ser considerado bem sucedido. Dificuldades em áreas como a Gestão de Recursos Humanos, o Marketing ou a Gestão Financeira, exteriores ao âmbito deste trabalho, podem ser responsáveis, directa ou indirectamente, pelo sucesso ou insucesso dos projectos. Por exemplo, uma gestão financeira errada pode levar a que o projecto esgote o orçamento antes do previsto, tal como uma campanha de marketing errada pode afastar o mercado esperado para um certo produto, levando a que este não seja vendido suficientemente e terminando o seu ciclo de vida antes do tempo esperado. Por isso não são apenas necessárias técnicas de Engenharia para o processo de desenvolvimento dos sistemas de informação, tal como não o são para qualquer outra área ou produto. Pode-se porém indicar, de uma forma genérica, algumas características comuns às melhores técnicas de Engenharia [Berard, 05]: (1) podem ser avaliadas quantitativamente, tal como qualitativamente; (2) podem ser usadas repetidamente, com resultados semelhantes; (3) podem ser ensinadas num período razoável; (4) podem ser aplicadas por outras pessoas com um nível de sucesso aceitável; (5) atingem melhores resultados que outras técnicas, de forma consistente e significativa, ou que uma abordagem *ad hoc* e (6) são aplicáveis numa percentagem significativa de casos. Desta lista de características, facilmente se observa que grande parte das práticas que normalmente associamos à actividade de desenvolvimento de sistemas de informação estão mais na categoria da Arte do que da Técnica, por não terem as características indicadas. Este facto é mais evidente nas actividades que envolvem níveis elevados de abstracção e de criatividade, como é o caso do planeamento do sistema de informação, da análise e concepção de sistemas de informação, ou da captura de requisitos. Apesar de existirem algumas linguagens e ferramentas que apoiam estas actividades, bem como a criação de padrões de desenho ou arquitecturais, tornando-as mais objectivas, os resultados estão ainda muito aquém daquilo que seria de esperar de uma actividade de Engenharia.

Entre o momento em que uma tecnologia se torna disponível e a sua adopção pela indústria existe geralmente um intervalo de tempo que é tanto maior, quanto menos evidentes são os ganhos no seu uso. Com as ferramentas CASE tem-se verificado alguma dificuldade na tomada de consciência da importância destas questões pelas equipas de sistemas de informação [Iivari, 96; Chervany et al., 98]. Foi apontado o facto de que as equipas em que este tipo de ferramentas não são obrigatórias tendem a não usá-las, mesmo se estas foram sugeridas e disponibilizadas. Existem diversas causas para este facto, que vão desde a dificuldade em aprender a operar as ferramentas [Zettel, 05], uma resistência à mudança na própria forma de trabalhar, ou um papel menor atribuído às estruturas organizacionais que gerem os sistemas de informação [Albizuri-Romero, 00]. O facto de estas ferramentas implicarem investimentos significativos em recursos financeiros e humanos e em tempo de aprendizagem não ajuda também a sua adopção. Em [Kline, 02] é ainda referido o facto de programadores experientes terem uma atitude mais positiva do que programadores inexperientes, apesar de ambos os grupos concordarem na dificuldade de aprendizagem do produto. Uma ferramenta, seja qual for, pode resolver um certo problema correctamente, mas se a sua utilização implicar um nível de conhecimento demasiadamente alto (medido em tempo e custo de aprendizagem) esta deixa de fazer sentido. Por enquanto é necessário um esforço na formação em duas vertentes: a conceptual e a operacional. Se é importante os utilizadores destas ferramentas conhecerem a forma de trabalhar com elas, neste caso, em particular, é tanto ou mais adquirirem os conhecimentos teóricos e metodológicos necessários. Apesar de envolver custos significativos, tanto financeiros como em tempo, a formação académica e profissional desempenha aqui um papel essencial para a consciencialização, por parte dos profissionais do sector, da necessidade de utilização das boas práticas anteriores. No âmbito das ferramentas CASE, a sua utilização só tem efeitos práticos positivos quando os seus utilizadores possuem o conhecimento teórico inerente às notações e metodologias adoptadas. Por outro lado, tal como tem acontecido com a generalidade das aplicações correntemente usadas em diferentes domínios, também as ferramentas CASE deverão tornar-se mais simples de utilizar, com a produção de interfaces gráficas cada vez mais intuitivas.

A par com a produção de novas linguagens e ferramentas de programação e modelação, tem existido uma alteração dos níveis conceptuais tratados por estas [Schmidt, 06]. Os aspectos técnicos relevantes para a modelação e programação de um programa abarcam progressivamente elementos sistémicos de níveis mais elevados de abstracção e complexidade. Assim, o surgimento de DSLs (*Domain Specific Languages*), a par com geradores de código, está a levar a que o MDE (*Model Driven Engineering*) ou mais especificamente o MDD (*Model Driven Development*) se torne cada vez mais uma realidade.

Neste contexto, supondo a verificação de um cenário em que: (a) os processos de desenvolvimento e manutenção recorrem a modelos (MDE); (b) estes modelos são produzidos através de ferramentas automáticas e (c) existe conhecimento adequado, por parte dos intervenientes no projecto, tanto do processo como das ferramentas envolvidas – subsistem ainda os problemas que se descrevem em seguida.

1.2 Identificação dos Problemas

Os problemas subjacentes a este trabalho podem ser analisados, pelo menos, segundo as seguintes perspectivas: artefactos e entropia; artefactos operacionais e conceptuais; ferramentas e artefactos.

Artefactos e entropia

O desenvolvimento de sistemas de informação é um campo suficientemente complexo para permitir a existência de uma grande diversidade de ferramentas usadas ao longo do ciclo de vida das aplicações, nomeadamente, compiladores, editores de texto, editores gráficos, controladores de versões, instaladores, produtores de instaladores, geradores de código, geradores de documentação de código, gestores de requisitos. Todas estas ferramentas produzem elas próprias novos dados que devem ser mantidos, com as suas possíveis alterações. Infelizmente a sua integração está longe de ser total e, mesmo os produtos mais completos, não cobrem todas as necessidades. Os artefactos gerados (i.e. os produtos resultantes da utilização de cada ferramenta) tornam-se assim em mais um aspecto a ter em conta no próprio processo de desenvolvimento. Considerando o processo de desenvolvimento de uma perspectiva sistémica, o aumento de artefactos pode indiciar um aumento da entropia do sistema. Porém, o número de artefactos, por si só, não é um problema. Se existirem ferramentas que os interligam de forma automática, relacionando-os inequivocamente (Figura 1.1), os conjuntos entretanto criados comportam-se como um elemento integrado naquilo que diz respeito à sua coerência (mantendo-se constante a entropia do sistema, segundo a mesma analogia).

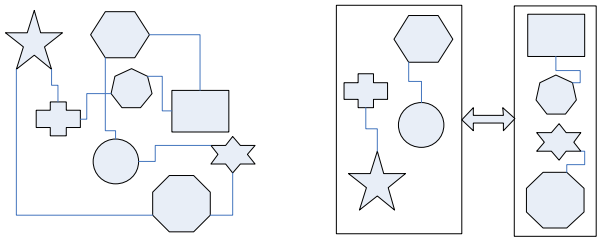


Figura 1.1 – Diagrama com elementos relacionados explicitamente e diagrama com elementos relacionados implicitamente

Normalmente, se o número de artefactos não constitui por si só um problema, já o mesmo não se pode dizer dos seus tipos. Ao introduzir-se um novo tipo de artefactos pode ser necessário assegurar a relação com os demais tipos. Neste ponto podem surgir dificuldades se a relação for mantida de uma forma manual, i.e., com a intervenção de um operador humano. Tome-se como exemplo um cenário em que existe uma ferramenta de gestão de requisitos através da qual são definidos os requisitos técnicos de uma aplicação, e.g., ter que usar os dados de uma tabela pertencente a uma base de dados existente. Apesar de os requisitos serem tratados automaticamente pela ferramenta (que os relaciona entre si) a sua relação com os demais é mantida manualmente (no exemplo dado, a alteração da estrutura da tabela, ou da semântica das suas colunas, teria de ser verificada aquando da utilização da dita ferramenta de gestão de requisitos).

Artefactos operacionais e conceptuais

Uma outra forma (porventura mais específica e concreta) de analisar o problema anteriormente descrito será considerar uma aplicação desenvolvida através de um processo envolvendo o uso de modelos, ao longo de diversas fases. Os modelos criados podem referir-se a diversos níveis conceptuais, sendo o sistema modelado subjacente o mesmo. Considere-se a divisão dos artefactos do sistema aplicacional em dois tipos: artefactos conceptuais e artefactos operacionais. O primeiro tipo representa todos os artefactos que não estão directamente envolvidos na operação do sistema, tais como: classe de domínio, tabela relacional contida num diagrama de tabelas, descrição de um requisito. Por outro lado, os artefactos operacionais representam todos os elementos que contribuem directa ou indirectamente para a operação da aplicação, e.g. classe C# da aplicação, em código fonte (indirectamente) ou binário (directamente), tabela relacional numa base de dados usada pela aplicação. Desta classificação resulta que as alterações ao sistema (i.e. aos artefactos que constituem a aplicação) podem ser realizadas sobre qualquer um dos dois tipos anteriores. Essas alterações, sobre um dado artefacto, podem implicar a alteração de outros artefactos que estejam com ele relacionados (mesmo que estes sejam de diferentes tipos de artefactos). Quando as alterações são feitas sobre artefactos conceptuais é muito provável que estas tenham de ser propagadas para os artefactos operacionais correspondentes. Os processos de geração automática favorecem este tipo de alterações permitindo em muitos casos uma actualização simples dos artefactos operacionais subjacentes.

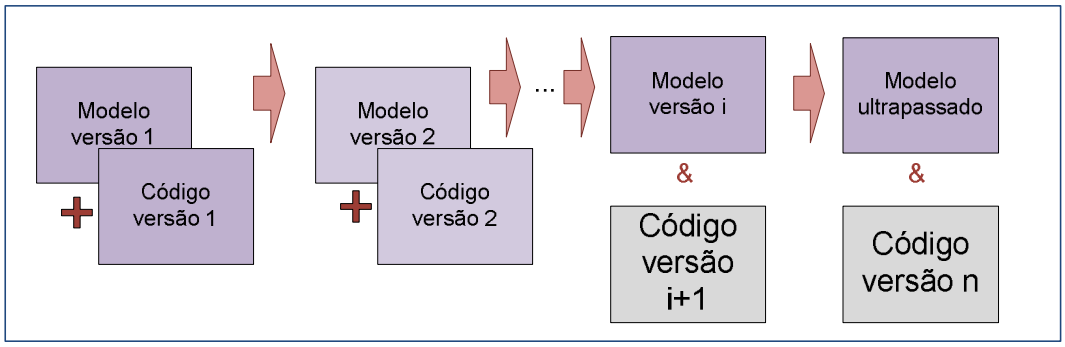


Figura 1.2 – Desactualização dos modelos por actualização do código

No entanto, e por outro lado, das alterações sobre artefactos operacionais, normalmente mais frequentes, podem advir incoerências entre estes e os artefactos conceptuais, caso não sejam acauteladas as acções necessárias. Destas incoerências resulta um óbvio desalinhamento entre a documentação do sistema (os artefactos conceptuais) e os respectivos artefactos operacionais. Desta forma, fica comprometida a **rastreabilidade** dos artefactos (neste contexto introdutório simplificadamente identificada como a capacidade de encontrar os artefactos relacionados com um dado artefacto base, em diferentes fases do desenvolvimento). Por outro lado, quando as aplicações entram na fase de operacionalização, a geração automática de código pode ser posta em causa pela existência de dados de produção. A interrupção, parcial ou total, da geração tem de ser acompanhada por restrições equivalentes nos artefactos pertencentes aos níveis conceptuais superiores. Da mesma forma a integração de uma dada aplicação num conjunto mais vasto de aplicações legadas tem de ter em conta as limitações de evolução das mesmas.

Ferramentas e artefactos

Devido à heterogeneidade das fontes dos artefactos envolvidos, produzidos a partir de todas as ferramentas necessárias ao desenvolvimento de sistemas de informação, a sua integração constitui um problema para as equipas de desenvolvimento. Mesmo em pequenos projectos, caso se utilize um grande número de ferramentas, o esforço de manter actualizada a informação de suporte ao desenvolvimento pode revelar-se imprevistamente alto, ou mesmo impraticável. Por outro lado, mesmo com a operação de uma única ferramenta, e.g. num IDE *(Integrated Development Environment)* podem existir diversos artefactos cuja coerência tem de ser mantida, e.g., referência a informação vinda de uma fonte externa. A forma como esta verificação é feita é usualmente um aspecto interno à própria ferramenta, pelo que, dificilmente se consegue acrescentar novas funcionalidades que permitam a sistematização, senão a automatização, desta actividade. Em vez da manutenção de coerência através de um qualquer componente *caixa-preta,* é importante que esta actividade possa ser parametrizada adequando-se às necessidades específicas de cada equipa de desenvolvimento.

Análise de sensibilidade

A alteração à estrutura de uma aplicação pode ter consequências não evidentes à partida. Como exemplo tome-se uma alteração ao nome de uma classe Java. Não só todas as referências da aplicação a essa classe têm de ser alteradas (o que é suportado usualmente pela ferramenta de desenvolvimento) como pode ser necessário alterar todos os modelos que referenciam essa classe, bem como toda a documentação resultante. Alternativamente pode também ser necessário deixar inalterados os nomes correspondentes em modelos que lhe dizem repeito, estabelecendo-se uma relação entre eles. Essa relação entre artefactos com conteúdo diferente (na medida em que podem ser outros atributos para além do *nome* a serem alterados, e.g., o tipo de acesso), após ser estabelecida, pode ser verificada e mantida. Uma aparente pequena alteração pode dar origem a um vasto conjunto de alterações o que pode ser evitado à partida se existir análise de sensibilidade, i.e, neste contexto, um estudo do impacto de uma alteração ao estado do sistema.

1.3 Tese e Objectivos de Investigação

Tese

Para ultrapassar os problemas identificados na secção 1.2 a tecnologia existente não é suficiente. É necessário algo mais, designadamente um modelo de coerência comum e uma nova classe de aplicações, ou pelo menos uma funcionalidade nova nas ferramentas de desenvolvimento existentes, que permita definir as regras de rastreabilidade necessárias entre os diversos artefactos da aplicação em causa, mantendo assim a sua coerência. Deverá preferencialmente ser providenciado um mecanismo que consiga verificar automaticamente essa coerência, permitindo tomar as decisões necessárias para que o estado da aplicação se possa manter coerente com o menor esforço possível. A implementação das funcionalidades referidas deverá estar sujeita a

normas que as uniformizem permitindo a sua utilização, estudo e melhoramento por uma comunidade alargada.

Objectivos

De forma a validar a tese enunciada propõe-se neste trabalho os seguintes objectivos de investigação:

- Identificar e enumerar factores metodológicos, humanos e tecnológicos que facilitem ou que obstem à rastreabilidade de artefactos;
- Definição de um modelo teórico de rastreabilidade de artefactos no contexto do desenvolvimento de aplicações que aborda os problemas descritos;
- Definir uma *framework* de software que permita a implementação de uma aplicação de rastreabilidade reactiva e restrinja esse desenvolvimento ao modelo teórico proposto;
- Validar e discutir os resultados obtidos pela realização de casos de estudo.

1.4 Descrição do Trabalho de Investigação

O presente trabalho centra-se na área do MDE, em particular centra-se nos mecanismos da geração automática no contexto do desenvolvimento de sistemas de informação, propondo uma abordagem que permita o uso real dos diversos artefactos produzidos, de forma coerente, pelas ferramentas intervenientes. Para tal, estabelece-se um modelo teórico que fundamenta os conceitos usados pela abordagem.

Propõe-se ainda uma nova classe de ferramentas de apoio ao desenvolvimento deste tipo de sistemas. Estas ferramentas, denominadas de "**rastreabilidade reactiva"**, permitem manter as referências entre artefactos conceptuais e operacionais presentes em diferentes contextos (e.g., SGBD Oracle, SGBD SQLServer, projecto C#), reagindo a alterações a cada um dos artefactos produzidos, de forma automática ou assistida. É possível fazer propagar, a diferentes contextos, uma alteração a um artefacto através de um conjunto de regras definidas sobre ele. A definição dos contextos e dos seus mapeamentos fazem também parte deste trabalho, sendo um dos aspectos que permitem ao utilizador criar e manter a rastreabilidade.

A rastreabilidade reactiva pode ser tratada através de ferramentas dedicadas a esse fim, operadas autonomamente, ou integrada em ferramentas já existentes. O protótipo criado usa a primeira abordagem tratando uniformemente artefactos de diferentes tipos (conceptuais e operacionais), bem como as relações entre estes.

São ainda tratados dois casos de estudo que consubstanciam e validam preliminarmente a abordagem permitindo retirar conclusões sobre a prática da mesma. O trabalho resultante está suportado por um estudo das metodologias e notações historicamente mais relevantes neste contexto, sendo a sua síntese um contributo para o enquadramento da solução encontrada. A

utilização de diferentes ferramentas ao longo da referida investigação contribuiu para a compreensão das potencialidades de cada uma, bem como para as dificuldades resultantes do seu uso.

1.5 Organização da Dissertação

Na escrita da dissertação são tomadas diversas opções de fundo que são contextualizadas e justificadas progressivamente ao longo dos seus diferentes capítulos. Genericamente, a dissertação evolui dos conceitos para a prática e do geral para o particular.

No Capítulo 2 é feito um enquadramento histórico e conceptual do trabalho. Devido ao facto das actividades inerentes ao desenvolvimento deste tipo de sistemas serem relativamente recentes nota-se por vezes, em trabalhos técnicos e científicos, uma tendência para ser ocultada a sua evolução a nível metodológico e notacional. Dá-se assim, na primeira secção deste capítulo, uma perspectiva da evolução histórica das notações e metodologias de desenvolvimento de sistemas de informação.

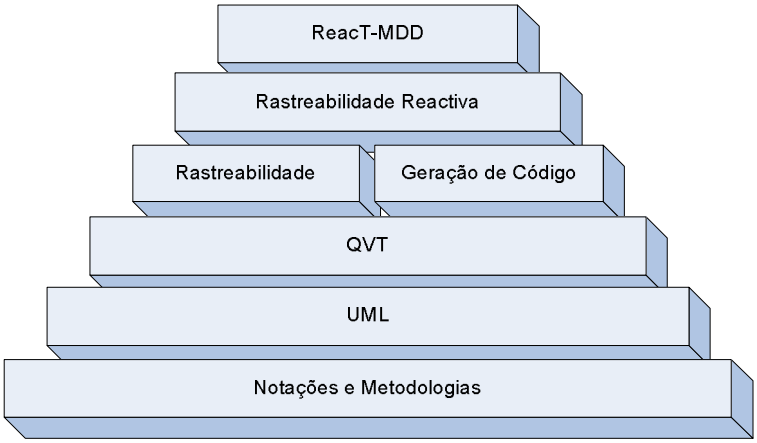


Figura 1.3 - Enquadramento conceptual dos diferentes capítulos da dissertação

Actualmente, o UML (*Unified Modelling Language*) é reconhecidamente a linguagem padrão na modelação de sistemas de informação. Este facto tem levado a um grande desenvolvimento de técnicas e ferramentas que gravitam em torno dos padrões estabelecidos pela OMG (*Object Management Group*). O modelo proposto tem por base alguns destes padrões que são abordados na segunda secção do mesmo capítulo. A actividade de desenvolver sistemas de informação envolve diferentes componentes conceptuais e operacionais apoiando-se em diversos tipos de ferramentas. Estas ferramentas são usadas por tipos de utilizadores diferentes, consoante a sua participação num projecto. Uma grande variedade de necessidades, por parte dos utilizadores, levou a uma oferta variada de ferramentas, com caracerísticas e potencialidades específicas. Na terceira secção do mesmo capítulo descreve-se sumariamente alguns destes tipos de ferramentas. Esta é apenas uma referência introdutória que visa contextualizar alguns conceitos referidos posteriormente. Para uma descrição mais exaustiva destas aplicações, bem como das suas características consulte-se o Anexo A. Tal como outras ferramentas de apoio ao desenvolvimento

de sistemas de informação, os CASE representam, por si só, um mercado competitivo onde operam algumas das mais importantes organizações da indústria informática. No contexto do desenvolvimento dirigido por modelos (MDE), estas ferramentas têm um papel essencial. No Anexo B é proposta uma metodologia para a avaliação e selecção de ferramentas CASE.

Muitos dos artefactos produzidos ao longo do desenvolvimento[2] são-no, total ou parcialmente, automaticamente. Usualmente a produção automática de artefactos, a partir de outros, denomina-se de geração automática. A geração automática recorre sempre a um conjunto de artefactos para gerar outros. Caso os artefactos que servem de inicío ou de fim sejam referentes a modelos, a geração produz uma transformação entre modelos ou entre modelos e código. Assim sendo, os diferentes padrões de geração automática relacionam-se com as transformações entre modelos (que na prática resultam normalmente na criação ou alteração de artefactos a partir de outros), no contexto da rastreabilidade. Na terceira secção do segundo capítulo são apresentados os diversos padrões de geração automática e é indicada uma classificação de transformações automáticas. Ainda no segundo capítulo é apresentado o modelo QVT *(Queries, Views and Transformations)* que é uma das propostas existentes na transformação entre modelos e constituiu a base para o presente trabalho.

No Capítulo 3 é apresentada uma proposta que concretiza a investigação realizada no que diz respeito aos aspectos teóricos. No início do capítulo são apresentados os conceitos básicos, no que diz respeito à rastreabilidade reactiva, sendo estes aprofundados gradualmente ao longo do capítulo. O texto envolve a utilização de diferentes tipos de expressão (e.g., diagramas UML, expressões formais e descrições textuais) para introduzir e definir os conceitos referidos. A apresentação dos conceitos envolve uma definição alternativa a outras existentes, sendo estas referidas e comparadas. São ainda explicadas, no mesmo capítulo, as diferenças entre as perspectivas usualmente adoptadas e a abordagem ReacT-MDD.

No Capítulo 4 são mostrados aspectos técnicos e arquitecturais da *framework* ReacT-MDD, bem como de um protótipo construído em conformidade, constituindo esta uma das contribuições deste trabalho de investigação.

No Capítulo 5 são explicados dois casos de estudo que permitiram retirar conclusões quanto à viabilidade da abordagem ReacT-MDD. Os dois casos têm propósitos diferentes; no primeiro caso é desenvolvido um exemplo simples que permite a compreensão da abordagem, enquanto no segundo caso é referida uma aplicação com um número maior de artefactos que pode ser considerada um exemplo de aplicação de algumas das práticas aqui referidas (e.g., geração automática de código, desenvolvimento conduzido por modelos, utilização de notações padronizadas como o UML).

No Capítulo 6 são apresentadas as conclusões e são indicadas algumas propostas de trabalho futuro.

---

[2] Quando não for explicitado entende-se “desenvolvimento” por "desenvolvimento do sistema de informação".

2 Estado de Arte

Neste capítulo é realizada uma síntese da evolução das notações e metodologias na área da modelação de sistemas de informação. A compreensão desta evolução, ainda que muito resumida, contextualiza as opções tomadas, nomeadamente quanto à escolha do UML e padrões associados, como linguagem de modelação. São ainda caracterizadas as ferramentas usadas no domínio do desenvolvimento aplicacional e da modelação. Estas ferramentas produzem os artefactos que têm de ser relacionados entre si, tendo sido por isso estudadas no âmbito deste trabalho de investigação. O UML é a notação usada ao longo do trabalho de investigação sendo abordado, juntamente com os padrões associados na secção seguinte. A geração automática, usada por vezes nas ferramentas estudadas, é uma das técnicas usadas para a produção de artefactos no contexto do desenvolvimento aplicacional. São explicitadas, por essa razão, as diferentes abordagens de geração automática. O QVT tem, em seguida, uma secção dedicada visto constituir um ponto de partida para conseguir atingir o objectivo da rastreabilidade entre artefactos.

2.1 Notações, Metodologias e Tecnologias de Desenvolvimento de Sistemas de Informação

As notações utilizadas para a modelação de sistemas de informação evoluíram notavelmente nos últimos quarenta anos do século XX. Como quase todas as notações surgiram como suporte a uma metodologia, a sua evolução está intimamente ligada ao desenvolvimento destas. Nesta secção dar-se-á uma perspectiva histórica desta evolução com o intuito de melhor enquadrar o aparecimento do UML e a sua utilização actual no âmbito deste trabalho.

Até à década de 1970, as metodologias de desenvolvimento de sistemas de informação tinham o seu foco essencial na produção de código de baixo nível através de requisitos bem conhecidos. A codificação de algoritmos em programas era feita seguindo-se uma dada metodologia, ou como ainda é comum, muitas vezes através de processos ad hoc. À medida que os sistemas de informação foram-se tornando mais complexos, mercê por um lado do aumento das expectativas dos utilizadores e por outro do progresso tecnológico na área do hardware, tornou-se notório que as metodologias tradicionais existentes não podiam servir para produzir e manter facilmente os novos sistemas.

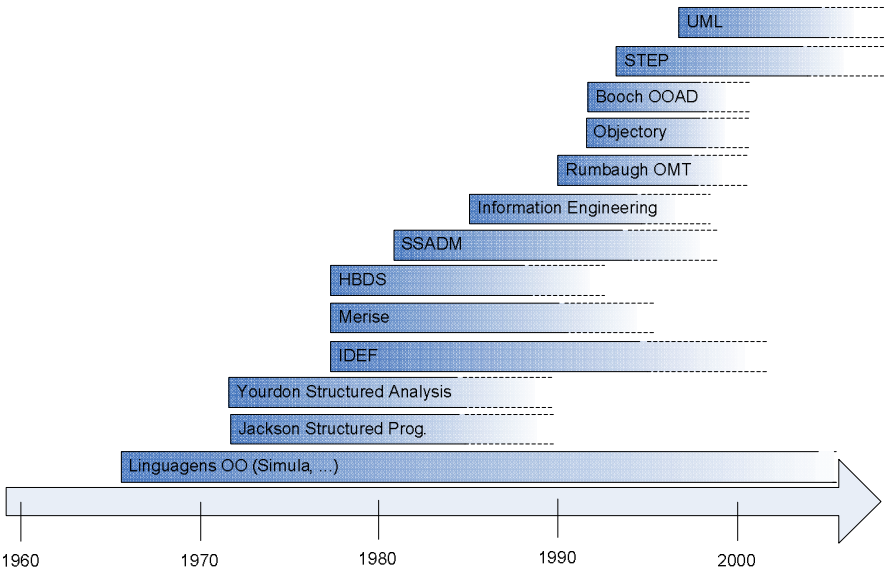


Figura 2.1 – Alguns marcos importantes na área do desenvolvimento de sistemas de informação

Durante as décadas de 1970 e 80 surgiram inúmeras metodologias de desenvolvimento de sistemas de informação, muitas das quais com notações associadas. Cada um dos proponentes destas metodologias tentou produzir a melhor notação para os artefactos produzidos ao longo do desenvolvimento de sistemas de informação (fossem eles esquemas contendo elementos gráficos designados por diagramas, tabelas e descrições textuais). No entanto, como cada uma das metodologias propunha diferentes tipos de artefactos, envolvendo notações diferentes, o resultado foi o surgimento de um grande número de notações. A partir da década de 1990, com a massificação das linguagens orientadas por objectos que vinham a ser desenvolvidas desde a década de 1960 [Dahl et al., 66] alguns teóricos começaram a reformular as metodologias clássicas tendo em vista este novo paradigma, enquanto outros tentavam criar novas metodologias. Embora com resistência e inércia [Ramsin et al., 08], as metodologias orientadas por objectos foram sendo desenvolvidas acabando por estabilizar e evoluir em meados da década de 1990.

São abordadas em seguida algumas das metodologias mais importantes na história, ainda curta, da modelação dos sistemas de informação. Desta lista constam metodologias cujas notações associadas se tornaram relevantes na respectiva época. Não sendo exaustiva, tanto em número como no tratamento de cada uma delas, esta secção coloca em relevo os aspectos importantes no contexto do estudo dos artefactos existentes e das suas possíveis relações.

2.1.1 Metodologias e notações clássicas

Jackson Structured Programming

O Método de Programação Estruturada de Michael Jackson [Jackson, 75] [King, 88], desenvolvido no início da década de 1970, inicialmente como proposta de extensão ao Cobol, tinha a sua aplicação na programação de processos batch, muitas vezes codificados nesta linguagem. Neste tipo de programas em que existe uma forte correspondência entre a estrutura do programa e as estruturas de dados de input e output, por um lado, e, por outro, as operações admitidas são apenas as operações básicas sobre dados, o método permite uma geração da totalidade do código.

O desenvolvimento pode ser apoiado por uma ferramenta existente [KCSL, i] e pode-se gerar código não só em Cobol como noutras linguagens como C, Pascal ou noutras linguagens imperativas. O método teve grande sucesso tendo sido adoptado como standard pelo governo do Reino Unido. No entanto, tinha o seu campo de aplicação muito limitado ao tipo de programas produzido na época que excluíam a interactividade hoje existente na maior parte das aplicações. O âmbito reduzido de aplicação, a par com o surgimento de outras abordagens limitou a utilização posterior deste método. A noção de rastreabilidade de artefactos está implícita à aplicação do MPE visto que cada artefacto depende dos artefactos produzidos numa fase precedente. O artefacto final (eventualmente o código) é o resultado de uma sequência de transformações sobre os artefactos anteriores. Se forem respeitadas convenções de nomeação dos conceitos intervenientes é possível identificar relações entre elementos dos modelos em diferentes artefactos.

Structured Systems Analysis and Design Methodology

Em 1981 o Governo do Reino Unido propôs-se a iniciar uma informatização em grande escala do seu serviço público, até aí com predominância de processos manuais. Foi então solicitada à *Central Computing and Telecommunications Agency* (CCTA) que criasse uma metodologia de análise e concepção de sistemas de informação padrão a ser aplicada em todos os projectos da administração pública de sistemas de informação [Hutchings, i]. A LBMS criou para a CCTA o SSADM (*Structured Systems Analysis and Design Methodology*) que rapidamente foi estudado e adoptado por diversos intervenientes da indústria de desenvolvimento, públicos e privados. O SSADM inclui uma notação para diversos diagramas que facilitam a especificação precisa de requisitos e a análise e concepção de sistemas de informação, tendo como preocupação o entendimento por parte tanto do produtor como do cliente desses mesmos diagramas. A metodologia inclui a realização de três tipos de modelos principais:

- Modelo Lógico de Dados – Realiza-se a identificação, a modelação e a documentação dos requisitos de negócio de um sistema de informação. Esta visão inclui um modelo de entidade-relação denominado Estrutura de Dados Lógica.

- Modelo de Fluxo de Dados – Realiza-se a identificação, a modelação e a documentação da forma como os dados circulam no sistema de informação da organização. Um Modelo de Fluxo de Dados é composto de um conjunto de Diagramas de Fluxo de Dados interligados com documentação textual. Os DFD representam Processos, Arquivos de Informação, Entidades Externas e Fluxos de Dados.

- Modelo de Eventos de Entidades – Realiza-se a identificação, a modelação e a documentação dos eventos que afectam cada entidade e a sequência em que esses eventos ocorrem. Um Modelo Entidade/Evento consiste num conjunto de Ciclos de Vida de cada entidade e em documentação textual.

Em 1995 o SSADM era a metodologia mais utilizada na Europa com mais de 5000 utilizadores certificados [Model Systems, 02]. O SSADM era uma norma aberta que foi revista até finais da década de 1990. Em 2000 o nome foi mudado passando a chamar-se “Business System Development” e actualmente existem adaptações desta metodologia a notações mais recentes

como o UML e aos conceitos de análise e concepção orientada por objectos com o OO-SSADM [Robinson et al., 94]. Existem diversas relações entre alguns elementos conceptuais de diferentes modelos (e.g., um Arquivo de Informação de um DFD está associado a uma Estrutura de Dados Lógica que representa um subconjunto da Estrutura de Dados Lógica do sistema completo).

Considerando a versão original do SSADM, verifica-se que tal como noutras abordagens do mesmo período, esta centra-se na compreensão e documentação de um domínio tendo em vista a implementação de uma ou mais aplicações nesse âmbito. A documentação pode englobar um número considerável de descrições textuais, em língua natural. Em consequência, previligia o sentido da criação/geração de artefactos dos modelos para o código e não o inverso. É possível considerar alguma rastreabilidade dos artefactos produzidos, embora, na prática, este aspecto fosse tratado de forma manual.

Merise

O método Merise nasceu igualmente das necessidades de um governo, neste caso o francês. O Ministério da Indústria francês procurava uma metodologia que permitisse ao serviço público desenvolver projectos de sistemas de informação dentro do tempo e do orçamento previstos [Tardieu et al., 83]. O projecto foi iniciado em 1977 e culminou num método que inclui diversas visões do sistema de informação, como o Modelo Conceptual de Dados (CDM) para conceber bases de dados, envolvendo uma notação própria para o diagrama de entidade-associação. Nesta metodologia são seguidas seis etapas: Sistema de Informação Manual, Especificação de Requisitos, Modelo Conceptual, Modelo Lógico, Modelo Físico, Sistema de Informação Automatizado [CommentCaMarche, 08]. O Merise foi a metodologia (com a sua notação original) mais usada em França, tendo sido substituída nos últimos anos por metodologias orientadas por objectos e pela notação do UML.

Embora existam relações implícitas entre diferentes artefactos produzidos em cada uma das etapas, a falta de ferramentas CASE suficientemente evoluídas condicionou o tratamento da rastreabilidade, transformando-o numa tarefa essencialmente manual.

Information Engineering

A metodologia *Information Engineering* (IE) foi criada por James Martin, considerado o pai das ferramentas CASE (*Computer Aided Software Engineering*) [SCE, i] e de algumas técnicas de RAD (*Rapid Application Development*) concretizadas numa metodologia denominada RIPP (*Rapid Iterative Production Prototyping*) em meados da década de 1980. A metodologia IE aborda o sistema de informação através de três visões: dados, processos e tecnologia; ao longo de quatro níveis ou fases: planeamento, análise, concepção e implementação. A IE pode ser expressa através da conhecida "Pirâmide do Sistema de Informação" [Martin, 89a] e foi criada para o desenvolvimento de grandes e complexos sistemas de informação em que a troca de informação entre as partes do sistema é uma característica essencial. A informação já tinha um papel fundamental nas metodologias anteriores, mas no IE os aspectos de integração e a relação com os processos funcionais são fundamentais. O IE foi definido por James Martin como "…um conjunto interligado de técnicas formais na qual os modelos empresariais, os modelos de processos e os

modelos de dados são construídos numa base xde conhecimento e são usados para criar e manter os sistemas de processamento de dados." [Titan, i].

O IE inicia-se com o desenvolvimento do Modelo da Organização [Martin, 89b], o qual é usado para descrever e compreender a missão e os objectivos da organização. O modelo estabelece as razões pelas quais as acções são realizadas e a forma de atingir a missão. O passo seguinte é realizar os modelos de Dados e de Processos que servem para satisfazer os requisitos dos utilizadores e comunicar os conceitos de análise e concepção entre as equipas de projecto. Além do estudo da organização, durante o ciclo de vida do sistema são constituídos e mantidos três modelos: Conceptual, Lógico e Físico, correspondentes a três níveis de abstracção através dos quais se pode analisar o sistema de informação.

O IE foi suportado por diversas ferramentas como o Information Engineering Workbench (IEW) da KnowledgeWare, Information Engineering Facility (IEF) da Texas Instrument [TI, 87] e o Excelerator da Intersolv. As duas primeiras aplicações podiam gerar código Cobol a partir de modelos de processos e esquemas de bases de dados a partir dos diagramas entidade-associação. O IE define o metamodelo de cada um dos artefactos produzidos, sendo as relações entre cada um dos elementos conceptuais presentes nos artefactos tratadas exaustivamente através de tabelas. Desde que sejam mantidas convenções de nomeação nos artefactos produzidos, é possível definir explicitamente relações como "Entidade é tratada em Processo" ou "Função contribui para Objectivo". A criação de alguns artefactos de código (e.g., estruturas de dados) era conseguida através de um processo de geração (automática, no caso da ferramenta IEW, ou eventualmente manual). Assim, considerando apenas os artefactos produzidos recorrendo-se à metodologia, o IE estimulou a rastreabilidade, embora sem ter tido as ferramentas automáticas necessárias que a concretizassem na prática.

Análise Estruturada

A Análise Estruturada (AE) é na verdade um grupo de metodologias que inclui, além de outros diagramas, o Diagrama de Fluxo de Dados. Ed Yourdon desenvolveu estes diagramas no início da década de 1970 [Yourdon et al, 78] sendo este o instrumento conceptual fundamental da AE. Alguns outros teóricos dos sistemas de informação, como Tom DeMarco (responsável pela AE), Paul Ward e Steve Mellor (criadores da Análise Estruturada de Tempo Real) contribuíram para a criação deste tipo de metodologias, num período em que a modelação de sistemas de informação se encontrava ainda num estado muito inicial. Na década de 1970, o desenvolvimento era na sua maioria feito por equipas da própria organização em projectos que tentavam resolver problemas muito específicos. Era comum utilizar-se um grande número de aplicações (normalmente em processos *batch*) que tinham de ser integradas de forma a constituírem a solução do problema. Algumas linguagens utilizadas então (e ainda hoje) não promoviam a modularização (e.g., Cobol) o que tornava difícil a manutenção de grandes aplicações. À medida que os sistemas de informação foram chegando a uma fase de maturidade, os processos de desenvolvimento começaram a assumir um menor peso nos custos dos projectos, enquanto a manutenção das aplicações começava a tomar a maior fatia dos orçamentos. A Concepção Estruturada de Yourdon tentou melhorar a modularização das aplicações e introduziu as noções de acoplamento e coesão que permanecem actuais trinta anos depois. Na Análise Estruturada de Tom DeMarco não foram

incluídos aspectos como a modelação lógica de dados e a modelação de eventos [Yourdon, 10], que só mais tarde surgiriam noutras metodologias. Nesta metodologia foi tratado o problema da mudança através da criação do *modelo lógico actual*, *modelo lógico novo, modelo físico actual* e *modelo físico novo*, por esta sequência. Usava-se então um processo sequencial, também denominado em *cascata,* em que a cada fase sucede uma outra que tem de estar terminada.

Como noutras metodologias, na AE encontram-se relações conceptuais entre alguns dos artefactos produzidos (e.g., DFDs Lógicos e DFDs Físicos, DFDs e descrição dos respectivos processos e arquivos de dados). No entanto o tratamento da rastreabilidade, como tal, não foi sequer abordado em ferramentas automáticas.

STEP

O desenvolvimento do STEP (*Standard for the Exchange of Product model data*) [Schenck et al, 94] foi iniciado em 1984 tendo sido tornado uma norma (ISO 10303) em 1994. Foi criado com o objectivo de permitir a descrição e partilha de informação de um produto, ao longo do seu ciclo de vida, usualmente em indústrias como a automóvel, naval, aeroespacial e construção civil, no entanto o seu âmbito cedo foi alargado a diferentes áreas. Neste tipo de indústrias, que envolvem um grande número de organizações em cada projecto, é muito importante que todos os intervenientes partilhem os mesmos modelos (principalmente de dados), independentemente da tecnologia que cada um utiliza. O STEP permite que um modelo criado numa dada ferramenta, com uma notação escolhida, através de uma certa metodologia, seja partilhado com outra equipa, fora ou dentro da organização, que utilize ou não a mesma metodologia, notação ou ferramenta. Para isso foi criada a linguagem textual Express, também ela uma norma (ISO 10303-11), para suportar o STEP. O Express tornou o STEP como uma das primeiras linguagens de modelação a ter um equivalente textual. Excluindo a informação gráfica, inerente aos diagramas produzidos, toda a informação produzida pelo STEP pode ser descrita pelo Express, tornando o primeiro independente da plataforma ou da ferramenta CASE (desde que esta consiga ler uma descrição Express). O âmbito do STEP não se limita à partilha de informação produzida por uma ferramenta CASE abrangendo domínios como as ferramentas CAE (Computer Aided Engineering Analysis), CAM (Computer Aided Manufacturing) e CAD (Computer Aided Design). A maior parte das ferramentas de CAD actualmente existentes implementam os protocolos AP-203 e AP-214 que servem para transferir informação para modelos independentes da tecnologia, segundo a visão STEP.

O conceito de artefacto é central ao STEP, embora seja considerado apenas como veículo para a transmissão de um conceito, ou visão parcial deste (dois artefactos podem incluir perspectivas complementares de um mesmo conceito).

IDEF

O IDEF (*ICAM*[3] *Definition Language*) é um grupo de métodos utilizados na modelação de sistemas de informação, criado pela Força Aérea Americana no fim da década de 1970. Inclui entre outros

[3] ICAM (Integrated Computer-Aided Manufacturing)

métodos: IDEF0 (Modelação funcional), IDEF1 (Modelação de informação), IDEF1X (Modelação de dados) e IDEF3 (Captura de descrições de processos). Existem na verdade dezasseis métodos IDEF, do IDEF0 a IDEF14 e IDEF1X. Estes incluem uma notação própria e são suportados actualmente pela ferramenta SmartER [KBSI, i]. O IDEF faz parte de um esforço de normalização levado a cabo pelo Departamento de Defesa dos EUA (DoD). Este grupo de métodos foi criado no âmbito dos sistemas de produção industrial sendo posteriormente generalizado e adoptado para o desenvolvimento de sistemas de informação. O IDEF pode ser usado isoladamente ou em conjunto com outras metodologias (no segundo caso para modelar conceitos mais afastados da tecnologia envolvida [Kim et al., 02]).

Os artefactos conceptuais produzidos pelo IDEF podem ser relacionados entre si. A manutenção destas relações, bem como das relações com (e entre) artefactos operacionais dependerá do desenvolvimento futuro de ferramentas.

2.1.2 Metodologias e notações orientadas por objectos

Hypergraph Based Data Structure

Em 1977, François Bouillé iniciou o desenvolvimento de um método de modelação de conhecimento [Bopearachchi, 02] orientado por objectos. No modelo HBDS (Hypergraph Based Data Structure) foram incluídos os conceitos das linguagens orientadas por objectos (p.ex. Classe, Objecto, Atributo e Operação) além de conceitos próximos das bases de dados, como a Ligação (equivalente à Associação do modelo entidade-associação de Chen [Chen,75]), eventos, lógica *fuzzy* e padrões de desenho [Bouillé, i]. Devido ao facto de o modelo conter todas essas características, existe uma grande uniformidade no seu tratamento. O HBDS continuou a ser utilizado e desenvolvido numa perspectiva académica tendo sido entretanto afastado em detrimento de outras metodologias.

Quanto aos artefactos, é possível integrar numa mesma ferramenta [Costa, 95] diferentes aspectos tradicionalmente tratados em ferramentas distintas (e.g., armazenamento de dados conceptuais e operacionais, lógica de negócio, geração de interfaces).

Objectory

A análise de casos de utilização, conhecida por método *Objectory*, foi criada por Jacobson em 1992 como forma de aproximar as metodologias de desenvolvimento de sistemas de informação às necessidades dos utilizadores. São essenciais a esta metodologia o conceito de *actor* e *caso de utilização*. Um caso de utilização pode ser definido como um "documento que descreve uma sequência de eventos de um actor (um agente externo) usando um sistema para completar o processo" segundo o seu criador [Jacobson, 92]. O Objectory representa uma tentativa de encontrar uma representação orientada por objectos de um sistema de informação, sob a perspectiva dos utilizadores desse sistema. Actualmente é muito utilizado para captar os requisitos dos utilizadores ao longo do desenvolvimento do sistema [Woolridge, 04] e deve estar sintonizada com a execução final para que o sistema atinja o objectivo de cumprir as expectativas dos utilizadores. A notação de Jacobson foi integrada no UML (Secção 2.1.3).

O *Objectory* envolve a utilização de artefactos com diagramas, bem como descrições textuais (usualmente em língua natural) para a documentação dos casos de utilização. A coerência entre os artefactos usados pelo Objectory pode ser conseguida com a utilização de uma ferramenta de gestão de requisitos, a par de uma revisão periódica das descrições definidas. No entanto, a coerência entre estes artefactos e os restantes é mais difícil de manter por faltarem relações implícitas que facilitariam essa actividade.

Object-Oriented Analysis and Design (Booch)

Booch [Booch, 94] desenvolveu uma metodologia de desenvolvimento de sistemas de informação que usa conceitos similares aos das linguagens orientadas por objectos a um nível de análise e concepção. A análise do sistema de informação percorre os seguintes passos:

- Análise de requisitos na perspectiva do utilizador através de ferramentas textuais;
- A análise de domínio é iniciada com a criação de modelos estáticos com diagramas de classes e suas relações;
- Descrição das classes, atributos e operações no Dicionário (é um repositório de informação acessível por todos os intervenientes no processo de desenvolvimento);
- Criação de máquinas de estados do tipo Harel [Harel, 87] para expressar o comportamento do sistema;
- Criação de diagramas de objectos, com notação específica, que demonstram diversos detalhes do sistema;
- Criação de modelos de interacção demonstrando relações para um determinado cenário;
- Uma vez a fase de análise de domínio terminada, é iniciada a fase de concepção que desenvolve a arquitectura do sistema. A fase de concepção é iterativa e consiste no mapeamento entre a concepção lógica e a concepção física onde são estabelecidos detalhes como processos, desempenho, local, tipos de dados, estruturas de dados específicas, visibilidade e distribuição [Burback, 98]. É então produzido um protótipo que será testado. O processo é iterado entre a concepção lógica, concepção física, protótipo e teste.

Apesar da metodologia indicar a implementação do protótipo e o seu teste, estes aspectos não são explicitados com grande detalhe. Existem relações lógicas entre alguns dos artefactos produzidos (e.g., os modelos de interacção e modelos de objectos partilham da definição das classes) mas não é indicada nenhuma forma de manter a coerência entre os artefactos produzidos.

Object Modelling Technique

A metodologia OMT foi criada por James Rumbaugh [Rumbaugh et al., 90] sendo precursora do UML, tal como as duas metodologias anteriores. A metodologia compreende as seguintes fases:

- Análise – Definição do problema e construção dos modelos de Objectos, Dinâmico e Funcional;

- Concepção do sistema – Organização do sistema em subsistemas, definição de questões relacionadas com o armazenamento da informação, concorrência e gestão de recursos.

- Concepção dos objectos – Concepção de algoritmos e estruturação das classes.

A fase de análise OMT inclui três modelos: Modelo de Objectos, Modelo Dinâmico e Modelo Funcional. O Modelo de Objectos na verdade representa normalmente classes e as suas relações estáticas com outras classes, podendo incluir também objectos recorrendo-se a uma notação especial. O Modelo Dinâmico é representado através de diagramas de sequência e máquinas de estados, análogos aos utilizados em UML. Neste modelo são identificados os aspectos da sequência do controlo eventualmente tendo em atenção aspectos temporais. No Modelo Funcional dá-se relevância às transformações ocorridas nos dados do sistema, que são modeladas através de diagramas de fluxo de dados.

2.1.3 Ferramentas de apoio ao desenvolvimento de sistemas de informação

Existem actualmente inúmeras ferramentas de apoio ao desenvolvimento de sistemas de informação. Os ambientes de desenvolvimento (IDE, Integrated Development Environment) são utilizados para as actividades que normalmente dizem respeito à criação e manutenção do código do programa, tal como à sua compilação. Este tipo de ferramenta até meados da década de 1990 estava normalmente associado a apenas uma linguagem de programação de terceira geração (e.g., Borland Turbo Pascal para a linguagem Pascal, Cincom VisualWorks para a linguagem Smalltalk). Posteriormente os ambientes de programação começaram a integrar diversos compiladores o que permite utilizar o mesmo editor e a mesma interface gráfica para programar em diversas linguagens (e.g., Microsoft Visual Studio .NET, Eclipse).

As ferramentas CASE (*Computer Aided Software Engineering*), um outro tipo de aplicações, foram criadas com o intuito de facilitarem o desenho e a manutenção dos diferentes artefactos (e.g., diagramas, texto) inerentes às metodologias de desenvolvimento de sistemas de informação existentes. Além de auxiliarem o desenho dos sistemas de informação, nas suas diversas vertentes, estas ferramentas permitem a utilização de um repositório comum de informação que elimina a redundância e facilita a actualização automática dos artefactos produzidos (e.g., relatórios, esquemas de bases de dados, pedaços de código de linguagens de programação).

Consoante as ferramentas CASE estão mais ou menos próximas das linguagens de programação, isto é, as suas actividades estão associadas à implementação propriamente dita, são designadas por *Lower* ou *Upper CASE,* respectivamente. Os *Upper CASE* estão mais vocacionados para as actividades de alto nível (i.e., desenho dos diagramas, geradores de documentação a partir do repositório comum, ferramentas de análise) enquanto os *Lower CASE* suportam a implementação (e.g., gerando código a partir de modelos e *vice versa*).

Actualmente as diferenças apontadas entre as ferramentas tendem a esbater-se com a permuta de características entre os dois tipos. Existem assim IDEs com características das ferramentas CASE (e.g., Microsoft Visual Studio .NET, IBM Eclipse) ou ferramentas CASE com capacidades anteriormente características dos IDEs (e.g., Borland Together, IBM Rational Rose).

Ferramentas CASE

Segundo o Software Engineering Institute [SEI, 05] uma ferramenta CASE pode ser definida como *um produto de software destinado a suportar uma ou mais actividades de engenharia no âmbito de um processo de desenvolvimento.* Esta definição é suficientemente genérica para abarcar ferramentas tão diferentes como compiladores, aplicações de gestão de configurações, sistemas de gestão de bases de dados, etc. No entanto se se definir os constituintes e funcionalidades de uma ferramenta CASE a distinção deste tipo de ferramentas torna-se evidente. Uma ferramenta CASE tem obrigatoriamente de conter um repositório comum de informação que guarda toda o conhecimento existente sobre o sistema em causa. O principal objectivo deste componente é o de evitar a redundância e a duplicação de esforços, permitindo a utilização de um elemento conceptual, definido através da ferramenta, em diversos diagramas, descrições, listas, tabelas, etc. Os elementos conceptuais podem relacionar-se entre si, num ou mais diagramas, sem que exista redundância no repositório.

Existem várias formas deste repositório ser guardado (e.g., através de um SGBD, em ficheiros de formato proprietário, em ficheiros XML) e disponibilizado, consoante a ferramenta permite o acesso ao repositório por diversos utilizadores ou apenas por um único. A ferramenta CASE pode por isso ter de gerir o acesso à informação de uma forma transaccional, considerando a possibilidade de as alterações estarem a ser feitas concorrentemente por diversos utilizadores. A gestão do repositório comum pode igualmente tratar aspectos como a segurança (e.g., criando perfis de utilizadores, níveis e tipos de acesso) ou a interacção com outras fontes de informação (e.g., importação e ligação a ficheiros com dados criados por outras ferramentas). Uma ferramenta CASE trata um conjunto de elementos conceptuais que podem ser gráficos ou textuais, segundo uma ou mais notações. Usualmente a ferramenta trata apenas uma linguagem de modelação (que nos últimos anos tem sido cada vez mais o UML) ou um subconjunto desta. No caso de uma linguagem extensa e com revisões constantes, como é o caso do UML, pode acontecer que uma ferramenta CASE não trate todos os elementos da linguagem, apenas os considerados mais importantes. A criação dos diagramas é feita através de um editor gráfico**,** com maiores ou menores potencialidades, para cada tipo de diagrama. Usualmente estes editores não permitem a inclusão de elementos estranhos a cada diagrama obrigando o utilizador a seguir as normas da linguagem. Evidentemente esta verificação é importante pois garante que os diagramas produzidos cumprem as normas definidas pela linguagem.

É também frequente este tipo de ferramentas conseguirem apresentar relatórios ou tabelas que relacionam de uma forma útil e actualizada os elementos guardados no repositório. Estes artefactos podem servir para validar os modelos criados, levando esta validação a uma nova iteração no processo de desenvolvimento. Podem também ser usados para documentar o sistema modelado, facilitando a sua compreensão por parte de quem ainda não o conhece completamente ou servindo de fonte à produção de outros documentos.

No entanto talvez a característica mais importante deste tipo de ferramentas seja o facto de elas lidarem com os modelos, não com as ocorrências destes, especificando-os através dos seus elementos constituintes e das relações entre estes, a diversos níveis de abstracção.

Ambientes de Desenvolvimento Integrado (IDE)

Um IDE é um conjunto integrado de ferramentas que permite a codificação e compilação de um programa, ou parte dele, recorrendo a uma ou mais linguagens de programação e compiladores associados. Um IDE possui um editor de texto que permite realizar a introdução e alteração do código numa linguagem de programação (e.g., C#, Java, C). Este editor pode ter uma maior ou menor complexidade, consoante utiliza características gráficas (e.g., tipo de letra, cores, botões para esconder/mostrar blocos de código). Um dos objectivos deste componente é o de facilitar a edição, disponibilizando automatismos que aceleram ou simplificam a produção de código (e.g., atalhos de teclas, *macros* prédefinidas ou definidas pelo utilizador, criação de elementos a partir de *templates*). Existem ainda editores gráficos que facilitam a construção das interfaces gráficas, ao permitirem o desenho dessas interfaces mostrando-as com o aspecto final. Os editores gráficos podem também facilitar a gestão dos eventos relacionados com a interface gráfica, associados à aplicação e incluir elementos não gráficos que reduzem o tempo de codificação (e.g., ligação a uma tabela de uma base de dados). Note-se que os editores gráficos geram automaticamente o código que representa cada um dos objectos gráficos visíveis. Para além destes componentes pode-se considerar o ambiente propriamente dito que é normalmente constituído por uma aplicação com uma ou mais áreas de trabalho responsável pela gestão dos ficheiros produzidos pelos editores.

2.2 Unified Modelling Language (UML) e Padrões Associados

Além das metodologias e notações anteriormente referidas, existiam ainda muitas outras, menos conhecidas por diversos motivos, e.g. terem aplicações específicas a um dado conjunto de contextos, ou terem sido desenvolvidas para o uso próprio de uma organização (cada grande empresa de consultoria tem normalmente uma metodologia própria ou adaptada de uma existente). Em muitos casos as notações eram completamente diferentes, mesmo estando de acordo em relação aos conceitos representados, e.g. uma classe poderia ser representada por uma nuvem na metodologia de Booch, por um rectângulo na metodologia de Rumbaugh, ou por uma elipse no HBDS.

Tendo em vista a uniformização da notação, independentemente da metodologia adoptada, Rumbaugh, Booch e Jacobson acordaram criar uma notação que servisse para as suas três metodologias, bem como para qualquer outra, como língua universal de modelação, denominada *Unified Modelling Language*. A iniciativa foi acolhida pelo *Object Management Group* [OMG] em 1997, uma organização não governamental de intervenientes na área das tecnologias da informação, tendo sido criado um standard. A especificação já teve diversas revisões encontrando-se actualmente na versão 2.1.1 [OMG, 07]. O facto dos teóricos de três das metodologias mais importantes na altura se terem juntado para um esforço comum de uniformização das suas notações, abriu campo à aceitação do UML por parte da indústria e do meio académico.

O UML compreende um modelo que distingue a representação dos conceitos dos próprios conceitos. Sendo assim, um modelo UML pode ser visto como um conjunto de esquemas ou

diagramas com uma notação pré-definida ou pode ser lido como uma descrição textual dos conceitos representados.

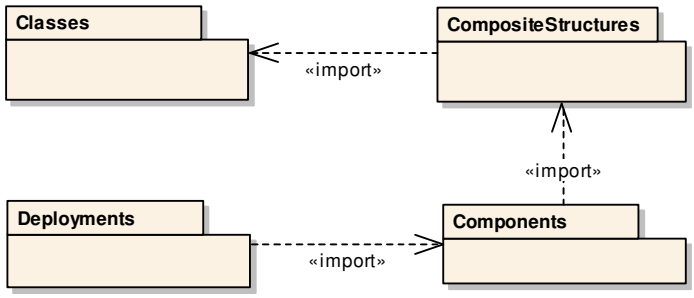


Figura 2.2 – Pacotes UML2 que suportam os modelos estruturais do UML

Tal como noutras linguagens de modelação, existem no UML dois tipos de diagramas: os estruturais (estáticos) e os comportamentais (dinâmicos) (Figuras 2.2 e 2.3). Na especificação do UML 2.1, no primeiro grupo incluem-se os diagramas de classes, os diagramas de pacotes, os diagramas de objectos, os diagramas de componentes, os diagramas de instalação e os diagramas de estrutura composta (UML2). No segundo grupo incluem-se os diagramas de casos de utilização, de sequência, de colaboração (ou comunicação), de estados, de actividade e de tempo (UML2).

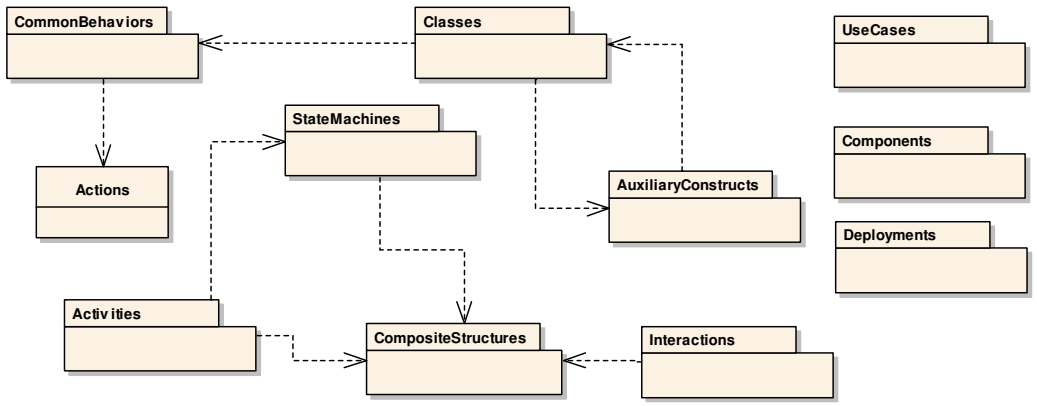


Figura 2.3 – Pacotes UML2 que suportam os diferentes tipos de diagramas em UML

O diagrama de classes é o mais usado e aquele que se considera acrescentar mais informação à modelação [Dobing et. Al, 06]. Sendo o seu elemento básico a classe, é compreensível que este diagrama seja adequado para o desenvolvimento através de metodologias e linguagens de programação orientadas por objectos. Segundo o mesmo estudo os outros dois diagramas mais usados são o diagrama de casos de utilização e os diagramas de sequência de mensagens. Contra aquilo que seria de esperar, os diagramas de sequência e os diagramas de colaboração, apesar de serem isomorfos (são ambos diagramas de interacção), são usados ambos nos mesmos projectos. Neste caso duas formas de ver a mesma informação parecem ser úteis para a documentação do sistema. Por outro lado, só 55% dos inquiridos consideraram que com o UML a comunicação com os clientes no máximo é moderadamente bem sucedida. Este dado põe em relevo o papel do UML

como linguagem para a documentação técnica do sistema. O UML não é uma linguagem universal para ser entendida por qualquer pessoa sem quaisquer conhecimentos de modelação, nem foi criada para tal. A própria especificação do UML envolve o conhecimento dos conceitos que define (e.g., na Figura 2.2 os elementos estruturais, representados graficamente, que constituem o diagrama são definidos no interior dos pacotes definidos no próprio diagrama).

O conhecimento do UML, tal como de qualquer outra notação, não é suficiente para que os modelos criados reflictam correctamente o problema em causa. Só um conhecimento profundo do domínio, aliado a uma boa técnica de modelação pode atingir esse objectivo. Embora existam críticas quanto aos artefactos produzidos pela modelação em UML (e.g., [Bell, 08]) esta apresenta-se como a opção mais conhecida e usada neste sector.

O UML está organizado numa arquitectura com quatro níveis conceptuais, na qual cada nível conceptual descreve os elementos do nível inferior. P. ex., os elementos de um sistema de informação podem ser descritos por um conjunto de diagramas escritos em UML; os diagramas UML são definidos através do metamodelo UML; o metamodelo UML é explicitado através do meta-metamodelo MOF, descrito em seguida.

2.2.1 Meta Object Facility (MOF)

A especificação MOF 2.0 [OMG, 06], que substituiu o MOF 1.4 [OMG, 02], define uma linguagem abstracta e uma infra-estrutura para suportar, construir e gerir metamodelos independentemente da tecnologia adoptada. Um metamodelo é uma linguagem abstracta para um determinado tipo de metainformação. O MOF define também uma infra-estrutura para implementar repositórios que guardam metainformação (e.g., modelos) descrita por metamodelos. Além de outros elementos, a especificação MOF inclui uma definição do meta-metamodelo MOF, isto é, uma linguagem abstracta para especificar os metamodelos MOF, bem como um formato XML Metadata Interchange (XMI) para permitir a transferência de metainformação entre diversos sistemas.

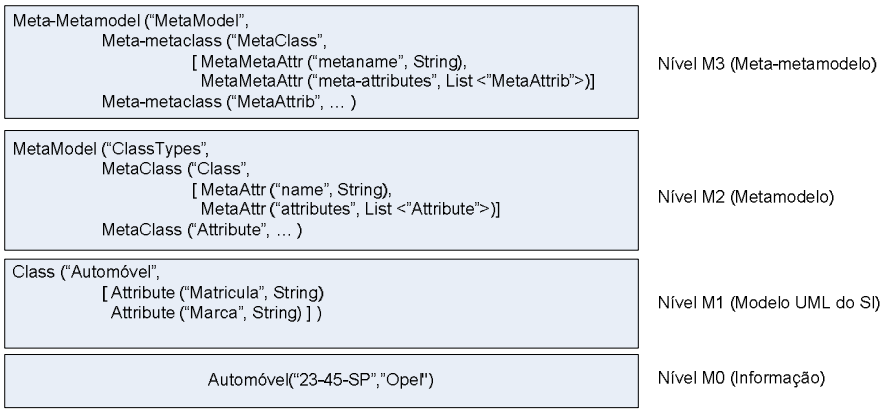


Figura 2.4 – Arquitectura do MOF em quatro níveis

A especificação do MOF inclui um conjunto limitado de conceitos de forma a tornar o standard suficientemente genérico para abarcar todas as implementações. No entanto o MOF é extensível por herança e composição permitindo-se assim definir um modelo de informação mais rico que suporte outros tipos de conceitos.

Com uma arquitectura com diversos níveis pode ser utilizado um mecanismo de reflexão através do conhecimento do nível superior ao nível que contém os objectos a serem tratados. Pode-se assim extrair toda a metainformação dos elementos de um diagrama (e.g., os nomes e tipos de atributos das classes existentes), conhecendo o metamodelo subjacente. Esta questão será explorada com maior detalhe posteriormente sendo da maior importância para o presente trabalho.

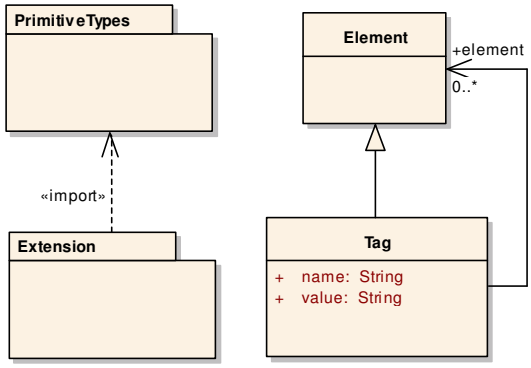


Figura 2.5 – Metamodelo MOF dos elementos que permitem associar marcas de valor aos elementos de qualquer diagrama MOF

Os modelos MOF permitem definir elementos de metamodelos que possuem propriedades e operações, de uma forma coerente com um diagrama de classes UML. Permitem ainda anotar cada um dos elementos do metamodelo com informação que pode ser relevante para a documentação do mesmo. Um exemplo de metamodelo MOF encontra-se na Figura 2.5, onde o diagrama apresentado define alguns elementos dos próprios diagramas MOF. Neste caso foi definido o próprio mecanismo que permite associar informação textual a cada elemento de um metamodelo MOF. Assim sendo, o diagrama é um metamodelo que define os próprios conceitos que podem ser utilizados nele próprio (o que neste caso particular não acontece).

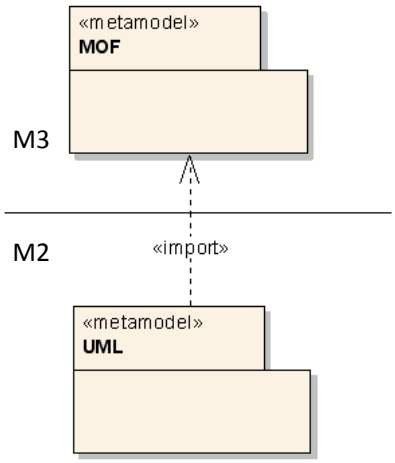


Figura 2.6 – Relação conceptual entre as especificações do UML e do MOF

A especificação do UML utiliza os conceitos definidos pelo MOF para definir os metamodelos correspondentes (Figura 2.6). A especificação do UML, na medida em que define os elementos conceptuais que são usados para definir modelos, encontra-se no nível M2. Como o MOF define os conceitos que são instanciados na própria definição do UML, situa-se no nível M3.

Os níveis de especificação do MOF não dizem nada sobre o nível conceptual dos elementos que estão a ser modelados. Assim, é possível ter um modelo A do nível M2 que representa elementos de *hardware,* coexistindo com um modelo B do nível M1 que representa elementos conceptuais. Nesse caso o modelo A poderia definir uma linguagem usada para modelar elementos de *hardware*, enquanto o modelo B seria um dado modelo de negócio. Foi por isso necessário identificar uma outra abstracção que permitisse classificar os diversos modelos do tipo M1, naquilo que respeita à sua dependência com a tecnologia.

2.2.2 Abordagem MDA

O standard *Model Driven Architecture* (MDA) da OMG tem como objectivo definir uma abordagem para o desenvolvimento de sistemas de informação, separando a modelação do domínio de negócio do seu modelo específico para uma plataforma ou tecnologia. Este conceito não é propriamente novo, como foi referido anteriormente. Já Yourdon [Yourdon et al, 78] tinha separado em quatro fases as actividades de modelação (modelo físico actual, modelo lógico actual, modelo lógico novo, modelo físico novo). A separação tinha como propósito conseguir realizar uma nova implementação (modelo físico novo) numa tecnologia diferente da anteriormente utilizada desde que o domínio se mantivesse inalterado. No entanto o MDA representa um passo significativo no contexto actual na medida em que o UML é reconhecidamente a linguagem padrão para a modelação de sistemas de informação.

O MDA divide o sistema em quatro níveis (Figura 2.7):

- *Modelos independentes da computação* (*Computation Independent Models* – CIM): São especificações de requisitos nos quais não existe qualquer opção tecnológica, seja ela de hardware ou software. Estes modelos são uma descrição da situação em que o sistema é usado assim como dos requisitos necessários à sua execução. Os modelos CIM englobam aquilo a que na metodologia *Information Engineering* de James Martin se chama de modelos de Planeamento. As especificações neste nível são expressas através de diagramas, em vez de texto, aproveitando-se a sua semântica mais rigorosa, no entanto sem ser formal.

- *Modelos independentes da tecnologia* (*Platform Independent Models* – PIM): São descrições do sistema sem serem tomadas decisões quanto à tecnologia envolvida, seja ela de hardware ou software. Esta decisões nem sempre são reconhecidas muito facilmente. Tome-se como exemplo um diagrama de tabelas relacionais. Apesar de não estar indicada nenhuma tecnologia à partida, foi escolhido já um tipo de tecnologia, neste caso os sistemas de bases de dados relacionais (estaríamos a afastar a hipótese de o sistema poder conter uma base de dados orientada por objectos ou hierárquica). Um modelo independente da tecnologia deverá sê-lo não só explícita como implicitamente, isto é, podendo ser implementado com qualquer tecnologia disponível.

- *Modelos dependentes da tecnologia* (*Platform Specific Models* – PSM) – Por exclusão são aqueles domínios onde foi tomada uma qualquer opção tecnológica, seja ela de software ou hardware.

- *Código* – Considera-se não só o texto que constitui as aplicações antes de serem compiladas (nos casos em que esta exista) como também outras descrições necessárias à execução do sistema (e.g., ficheiros de configuração) ou apenas documentos de carácter técnico, entre outros.

Para além destes três níveis de modelos e do nível de código, o MDA inclui também o conceito de mapeamento entre modelos (e entre modelos e código). Assume-se como mapeamento entre modelos o conjunto de dependências que existem entre elementos de modelos distintos e que serão tratadas com maior detalhe na secção 2.4 (Modelo QVT). Pode considerar-se o mapeamento entre modelos e código como um subconjunto do mapeamento entre modelos.

O conceito de mapeamento é fundamental para a geração automática de artefactos que se pretende que seja uma das principais vantagens do MDA. Segundo esta abordagem existem vários tipos de mapeamentos entre modelos de diferentes níveis, nomeadamente:

- Mapeamento CIM – PIM: Neste caso os modelos PIM satisfazem os requisitos expressos nos modelos CIM.

- Mapeamento CIM – CIM: Dois modelos, ou elementos de modelos, CIM podem estar relacionados de alguma forma.

- Mapeamento PIM – PIM: Neste caso os dois modelos são independentes da tecnologia mas estão a níveis diferentes de abstracção ou explicitam visões diferentes sobre o sistema, existindo, porém, dependências entre alguns elementos dos dois modelos.

- Mapeamento PIM – PSM: Neste caso um dos modelos é independente da tecnologia e o outro é dependente. Os elementos que estão relacionados admitem um mapeamento que se traduz normalmente numa geração de código no sentido PIM --> PSM. Como exemplo tome-se a derivação de um diagrama de classes para um diagrama de tabelas.

- Mapeamento PSM – PSM: Ambos os modelos são dependentes da tecnologia (eventualmente diferente) mas existe uma relação entre alguns elementos dos dois modelos (e.g., os nomes de algumas classes de um componente escrito em C# têm de ser iguais aos das classes escritas em Java num outro componente).

- Mapeamento PSM – Código: A derivação de um diagrama de tabelas para o código SQL correspondente à criação dessas mesmas tabelas é um exemplo deste tipo de mapeamento. Neste caso as alterações feitas à estrutura da base de dados têm um mapeamento perfeito no diagrama de tabelas podendo este ser feito nos dois sentidos.

Existem consequências do MDA a diversos níveis [OMG, 03]:

- Implementação: uma nova infra-estrutura de implementação pode ser integrada através da concepção já feita recorrendo-se aos modelos CIM e PIM;

- Integração: Existe não só a implementação como a concepção do sistema actual quando se realiza a integração por isso a produção de pontes de integração de dados pode ser automatizada, tal como a ligação a outros elementos a serem integrados;

- Manutenção: existe o modelo de concepção numa forma automática o que dá aos programadores acesso directo à especificação do sistema, tornando a manutenção mais simples;

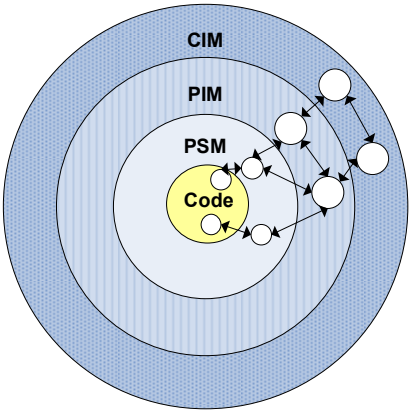


Figura 2.7 – Representação das diferentes camadas do MDA e dos mapeamentos possíveis entre elementos de diferentes camadas

- Testes e simulação: como os modelos desenvolvidos podem servir para gerar código, podem igualmente ser validados quanto a requisitos, testados em diversas infra-estruturas e ainda ser usados para simular o comportamento do sistema a ser modelado.

Como em qualquer linguagem de modelação, também no UML poderia ocorrer o problema do excesso/falta de especificação. Quando existe excesso de especificação, os diagramas produzidos recorrendo à linguagem de modelação identificam correctamente os conceitos envolvidos. No entanto, nesse caso, visto que o excesso de especificação resulta normalmente num aumento dos elementos conceptuais disponíveis, podem ocorrer os seguintes problemas: a linguagem pode tornar-se difícil de aprender (devido à sua especificidade); os diagramas podem tornar-se difíceis de ler para quem não tem um conhecimento específico da área em questão; a sua utilização noutro contexto pode ser difícil ou impraticável. Pelo contrário, quando uma linguagem é demasiado abstracta acaba por não ter expressividade suficiente para que os diagramas produzidos sejam de alguma utilidade.

O UML está definido de uma forma suficientemente genérica para permitir a utilização dos seus diagramas em diferentes contextos. Existem ainda mecanismos de extensão, abordados em seguida, que permitem enriquecer os diagramas UML, dotando-os assim de uma semântica mais rica.

2.2.3 Extensões ao UML

Os diagramas UML podem ser utilizados na maior parte dos casos sem ser necessário recorrer a qualquer alteração à sintaxe definida. No entanto, por vezes um diagrama pode não transmitir facilmente a semântica do problema em causa. O UML providencia um mecanismo de extensão que determina a forma como a linguagem pode ser enriquecida semanticamente. Sendo assim, desde que sejam cumpridas as normas [OMG, 03a] um diagrama pode expressar melhor um certo domínio e ao mesmo tempo assegura-se que o diagrama continua a ser escrito/desenhado em UML. As extensões ao UML são na verdade especializações, ou refinamentos, do seu metamodelo e representam uma adição à semântica base (não é possível alterar um elemento do metamodelo padrão).

O mecanismo de extensão mais importante é o *estereótipo* que permite definir subclasses virtuais das metaclasses UML com novos metaatributos e uma semântica adicional. O estereótipo é ele próprio uma metaclasse em UML. Sendo assim um utilizador de uma ferramenta UML pode definir um estereótipo «seguro» que pode ser associado às classes do modelo de forma a posteriormente poderem ser tomadas decisões quanto à segurança de um subconjunto da informação.

As *marcas de valor* especificam novos tipos de propriedades que podem ser associados aos elementos do modelo. Os valores podem ser simples como inteiros ou cadeias de caracteres, ou mais complexos como referências a outros elementos do modelo.

As *restrições* são o terceiro e último elemento de extensão ao modelo e podem ser associadas a qualquer elemento do modelo permitindo que todas as instâncias desse elemento possam herdar a restrição definida. As restrições podem ser aplicadas ao metamodelo, nomeadamente associadas a estereótipos, sendo definidas através de uma linguagem como o OCL.

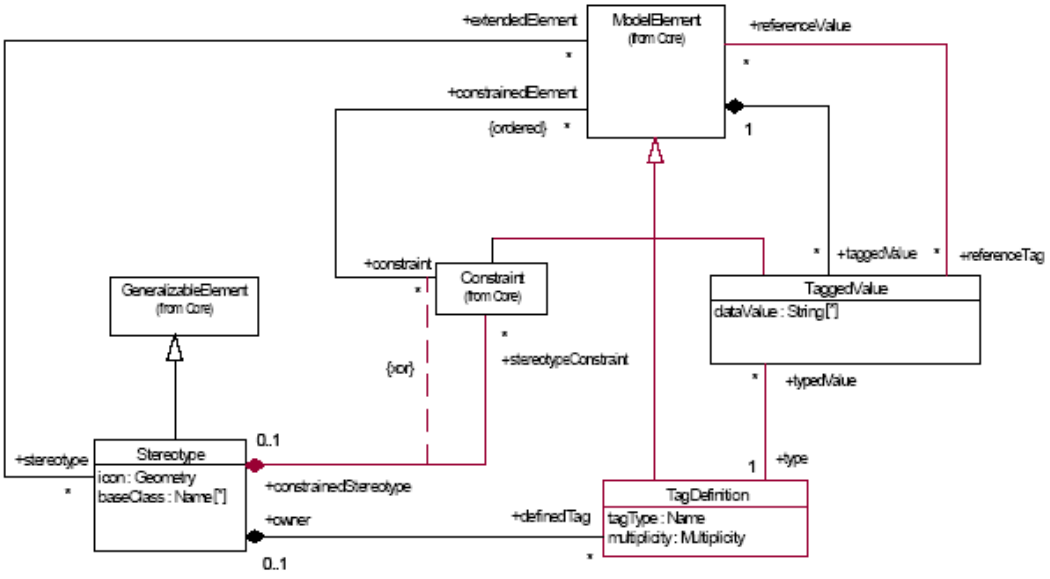


Figura 2.8 – Metamodelo dos mecanismos de extensão do UML ([OMG, 03a])

Por vezes é criado um número significativo de extensões passíveis de serem utilizadas noutros sistemas. Para que esta metainformação possa ser sistematizada foi definido o conceito de *perfil*

que é um *package* estereotipado que contém elementos do modelo especializados para um dado domínio através de estereótipos, marcas de valor e restrições.

2.2.4 OCL

Um diagrama UML, como o diagrama de classes, nem sempre consegue explicitar todos os aspectos relevantes de uma especificação. Pode ser necessário, por exemplo, especificar determinadas restrições acerca dos elementos no modelo. Estas restrições podem ser escritas numa língua natural com as ambiguidades inerentes a este tipo de expressão. Por isso surgiu a necessidade de definir uma linguagem formal para este efeito que conseguisse ser simples de utilizar (o que usualmente não é o caso nas linguagens formais). A *Object Constraint Language* (OCL) [OMG, 03b] foi criada com este fim, como uma linguagem formal para descrever expressões em modelos UML. Estas expressões especificam normalmente condições que devem ser satisfeitas para partes dos sistemas modelados ou pesquisas sobre os elementos descritos no modelo. As expressões OCL são completamente independentes da tecnologia e podem ser mapeadas para outras linguagens. No entanto a avaliação de uma expressão OCL não pode alterar por si só o estado do sistema ou qualquer outro elemento do modelo. O OCL não é uma linguagem de programação pelo que não é possível descrever o fluxo de controlo ou a lógica de um programa. Também não é possível invocar processos ou activar operações que não sejam pesquisas, dentro do OCL. Um conjunto de expressões OCL não é, por definição, directamente executável, considerando-se a sua avaliação instantânea (i.e., os estados dos objectos do modelo não podem mudar durante a avaliação). No entanto, o OCL pode ser usado, em conjunto com outra linguagem (nomeadamente o QVT), para especificar operações ou acções que, quando executadas, alteram o estado do sistema (este aspecto será tratado com maior profundidade na Secção 2.4). A linguagem OCL é *tipificada*, i.e., os objectos OCL têm de pertencer a um dos tipos existentes, sejam estes predefinidos ou definidos no modelo. Existem diferentes projectos (e.g., OCLE [Chiorean, 05], KMF [Akehurst et al., 04], MMT [Clark et al. 2001]) que implementam subconjuntos do OCL tendo por fim a construção de metamodelos, a validação e a análise de coerência entre estes, a conformidade dos modelos com os metamodelos, entre outras características, consoante os casos. A complexidade da sistematização das actividades de metamodelação usando o MOF tem levado a que as ferramentas nesta área estejam ainda numa fase de investigação, apesar de já existir um esforço considerável nesta área pelo menos desde o 2000 (Dresden [Hussman et al. 00] e USE [Richters et al., 00] [Gogolla et al., 07]).

2.3 Transformações Entre Modelos e Geração Automática

À medida que têm emergido novas linguagens de programação, descrição de dados e processos, modelação, bem como de outras actividades inerentes ao desenvolvimento de aplicações, tem surgido a necessidade de em certos casos fazer a conversão entre artefactos descritos em diferentes linguagens. A solução mais comum, no início, era encontrar um conjunto de regras de tradução entre cada duas linguagens e produzir uma ferramenta para executar essa tarefa. O próprio processo de compilação clássica de uma linguagem de programação de alto nível envolve a tradução de um programa escrito nessa linguagem para uma outra linguagem de nível tecnológico

mais baixo. Evidentemente, esta tradução é feita automaticamente na medida em que a linguagem de nível mais alto está completamente especificada na linguagem de nível mais baixo. Com o aparecimento de diversos paradigmas de programação (e.g., linguagens imperativas, funcionais, lógicas) a tradução automática tornou-se mais difícil devido a não existir um mapeamento directo entre os conceitos das diversas linguagens. Nos casos em que as linguagens partilham grande parte dos seus metamodelos (e.g. C# e Java) esta tarefa torna-se mais simples existindo ferramentas que realizam a tradução automática entre programas, embora os resultados desta transformação não sejam, pelo menos por enquanto, completamente fiáveis. Estas ferramentas não excluem a intervenção humana na medida em que para programas razoavelmente complexos (i.e., que envolvam características específicas às linguagens) estas ferramentas geram alguns erros de tradução que têm de ser tratados manualmente. Por outro lado, devido ao facto de poderem existir diversas formas de produzir a tradução, mesmo que o código gerado seja aparentemente correcto, podem existir melhores soluções aplicáveis caso a caso.

Um outro âmbito em que se aplica a tradução é a conversão de dados entre formatos diferentes. Ao longo da década de 1990 a linguagem XML (*Extensible Markup Language)* [W3C, 04], derivada do standard SGML (*Standard Generalized Markup Language*, ISO 8879), tornou-se na linguagem mais usada para a descrição de dados. O HTML (*HyperText Markup Language*), outra instanciação do SGML conforme com o XML, criada para a descrição de documentos por via electrónica, é uma das linguagens que fazem parte da infraestrutura da Internet. A manipulação de dados de ficheiros no formato XML motivou a especificação de uma outra linguagem conforme com o XML, designada por XSL (*Extensible Stylesheet Language,* [Patterson, 01]). Esta linguagem tem duas facetas para manipular e tratar documentos em XML. Por um lado, pode servir para formatar um certo ficheiro preparando-o para a apresentação ou para qualquer outro fim. Pode ainda ser usado para realizar transformações em ficheiros de XML de forma automática, recorrendo a uma ferramenta externa que executa essa tarefa. Nesse caso utiliza-se uma parte da linguagem denominada XSLT (*Extensible Stylesheet Language Transformation*). A OMG promoveu também um formato de ficheiro, baseado em XML, para a partilha de informação entre ferramentas CASE, denominado XMI (*XML Metadata Interchange, [OMG, 02a]*). Devido ao facto de o formato XML implicar que o ficheiro possa ser lido e entendido, não apenas por um programa, mas também por uma pessoa, os ficheiros XML (e XMI) incluem uma redundância (com todos os problemas inerentes como performance e espaço de memória) que deve ser evitada na implementação de um repositório de uma ferramenta CASE. Por este motivo, numa ferramenta CASE, o XMI pode servir para exportar dados para outras aplicações (eventualmente outras ferramentas CASE) mas não para guardar o próprio repositório. Eventualmente, a passagem de informação de uma ferramenta para outra, através de ficheiros XMI, pode implicar uma transformação dos dados. Nesse caso o XSLT pode ser usado como linguagem de transformação. Tem vindo a notar-se, no entanto, que o XSLT como linguagem de transformação apenas é viável para projectos em que a transformação é razoavelmente simples [Dollard, 04]. O código XSLT tem uma manutenção difícil e apenas é usado quando os ficheiros de input estão no formato XML.

As transformações de modelos e a geração automática aplicam-se, segundo a presente abordagem, a artefactos. Assim sendo, estes dois tipos de actividades relacionam-se com a rastreabilidade de diversas formas. Por um lado, podem condicionar a rastreabilidade se não for

mantido um registo das tranformações executadas, bem como dos processos de geração já realizados. Por outro lado, quando bem aplicadas, podem constituir ferramentas para a manutenção da coerência entre diferentes artefactos do sistema, facilitando até a compreensão do mesmo.

Tendo em vista a contextualização das abordagens de transformação de modelos, bem como a caracterização da solução apresentada, é apresentada nesta secção uma classificação baseada em Czarnecki e Helsen [Czarnecki et al., 03].

2.3.1 Classificação das transformações entre modelos

Uma **regra de transformação** pode ser vista como uma aplicação sobre um conjunto partida (*left hand side* ou LHS) a um conjunto chegada (*right hand side* ou RHS)[4]. Tanto o LHS como o RHS podem ser representados usando-se uma combinação dos seguintes elementos: (1) variáveis que guardam elementos dos modelos de partida e/ou chegada, (2) padrões que são fragmentos dos modelos com zero ou mais elementos, (3) expressões lógicas declarativas e imperativas que permitem formalizar restrições aos modelos, no primeiro caso, e realizar cálculos e operações sobre os modelos, no segundo.

Tanto as variáveis como os padrões podem não ser tipificados, ou sê-lo semântica ou sintacticamente. Caso seja sintacticamente tipificada a variável é associada ao elemento do metamodelo cujas instâncias deverá conter. Com variáveis tipificadas semanticamente é possível realizar mapeamentos mais complexos. Por exemplo, sendo o tipo sintáctico de uma variável "expressão" o seu tipo semântico poderia ser "expressão que é avaliada sobre um tipo inteiro".

Por outro lado, as duas partes da transformação (LHS e RHS) podem ou não estar separadas sintacticamente. É mais comum que estejam separadas, ocorrendo a transformação sobre diferentes metamodelos. As regras de transformação podem ou não ser bidireccionais, se a aplicação da transformação inversa resultar no conjunto inicial. Pode igualmente ser necessário criar estruturas intermédias à transformação (e.g., VIATRA [Varro, 02] e GreAT [Agrawal, 03]), usualmente quando o LHS e o RHS se referem ao mesmo modelo.

O **âmbito de aplicação** das regras de transformação permite restringir as partes do modelo que participam na transformação. Algumas abordagens suportam âmbitos flexíveis no LHS (e.g., XDE e GreAT), onde se pode limitar a transformação a um âmbito mais reduzido que o do modelo fonte completo. Esta característica pode ser importante para o desempenho da transformação.

No que diz respeito à **relação entre modelo inicial e modelo alvo**, algumas abordagens requerem a criação de um novo modelo alvo que tem de ser diferente do modelo inicial (e.g., AndroMDA). Noutras abordagens o LHS e o RHS referem-se sempre ao mesmo modelo (e.g., VIATRA, GreAT) fazendo com que a transformação seja dentro do modelo. Existem ainda outras abordagens (e.g. XDE) que permitem que o modelo alvo seja existente ou já existente. Se o modelo alvo

[4] Na verdade é possível conceber uma transformação que envolva mais do que dois conjuntos. Este tipo de transformação composta é um dos requisitos do QVT, como será mostrado na secção 2.4.

corresponde ao inicial podem ser realizadas operações sobre um subconjunto dos elementos existentes ou a criação de novos elementos. De qualquer forma, uma aplicação deste tipo pode permitir dois tipos de operações sobre os elementos: (1) operações de criação, modificação e eliminação dos elementos existentes ou (2) apenas a sua extensão, não permitindo assim a alteração ou eliminação.

Um regra tem de ser aplicada a um conjunto de elementos do modelo inicial. Como esse conjunto pode ter mais do que um elemento a aplicação da regra a cada um dos elementos deve submeter-se a uma **estratégia**. Esta estratégia pode ser determinística, não determinística ou interactiva. Uma estratégia determinística pode utilizar uma técnica específica de percorrer cada um dos elementos necessários, considerando-os como um grafo e aplicando um algoritmo de travessia. São também aplicadas regras escolhendo os elementos através de heurísticas ou através de processamento concorrente. O conjunto de elementos alvo é usualmente determinístico. No caso de uma modificação no próprio modelo inicial (quando este coincide com o modelo alvo) os elementos iniciais da transformação podem tornar-se os elementos alvo.

Os mecanismos de **agendamento da execução de regras de transformação** determinam a ordem pela qual as regras individuais são aplicadas. Os mecanismos de agendamento podem variar segundo quatro tipos:

- Forma: O agendamento pode ser expresso implicitamente ou explicitamente. O primeiro faz com que o utilizador não tenha intervenção no algoritmo de agendamento (e.g., BOTL, OptimalJ). A única forma pela qual o utilizador pode influenciar o agendamento é definindo padrões e a lógica das regras para garantir uma certa ordem de execução (e.g., se uma regra verifica a existência de um resultado que apenas outra regra produza, a sequência é garantida). O agendamento explícito envolve a existência de estruturas de controlo definidas numa linguagem de tratamento das transformações. O agendamento explícito pode ser ainda interno ou externo. No agendamento externo, existe uma separação clara entre as regras e a lógica do agendamento (e.g., no VIATRA [Varro, 02], o agendamento das regras é realizado através de uma máquina de estados finita). Em contraste, o agendamento interno é um mecanismo que permite a uma regra invocar directamente outras regras (e.g., diversas abordagens baseadas em *templates*);

- Selecção de regras: As regras podem ser seleccionadas através: de uma condição explícita (e.g, Jamda), de uma escolha não determinística (e.g., BOTL), ou interactivamente (e.g., XDE). Poderá ser interessante implementar ainda um mecanismo de resolução de conflitos, resultante da selecção de regras contraditórias;

- Iteração sobre regras: Os mecanismos de execução iterativa de regras incluem a recursividade, o ciclo iterativo imperativo e a aplicação repetida até ser cumprida uma condição;

- Faseamento: O processo de transformação pode ser organizado em diversas fases, onde cada uma das fases tem um propósito específico e apenas algumas regras podem ser invocadas em cada fase. Por exemplo, aproximações orientadas para a estrutura (e.g., OptimalJ) têm uma

fase separada para criar a hierarquia que contém o modelo alvo e uma outra fase para indicar os atributos e referências no modelo alvo.

Num projecto que envolva um número significativo de regras é importante encontrar abstracções que permitam uma **organização das regras.** Esta pode normalmente ser realizada através das seguintes técnicas:

- Mecanismos de modularidade: Algumas abordagens permitem o agrupamento de diversas regras em *módulos* (e.g., AST+ e VIATRA). Um módulo pode igualmente importar outro módulo para aceder ao seu conteúdo;

- Mecanismos de reutilização: Estes mecanismos permitem definir uma regra baseada em uma ou mais regras. Em geral, os mecanismos de agendamento podem ser usados para regras de transformação compostas; no entanto, algumas abordagens oferecem mecanismos dedicados como herança entre regras (e.g., herança de regras em AST+, derivação em IOTP, extensão em CDI, especialização em QVT), herança entre módulos (e.g., herança de unidades em AST+) e composição lógica (e.g., QVT);

- Organização estrutural: As regras podem estar organizadas de acordo com a linguagem fonte, a linguagem alvo, ou podem ter a sua estrutura independente destas.

Quanto à **direcção da transformação**, as transformações podem ser apenas unidireccionais[5] ou bidireccionais. As transformações unidireccionais partem de um modelo fonte e criam, ou alteram, um modelo alvo. As transformações bidireccionais podem ser executadas em ambas as direcções. As transformações bidireccionais podem ser conseguidas definindo regras bidireccionais ou usando duas regras complementares unidireccionais, uma para cada sentido da transformação.

As regras de transformação são concebidas normalmente numa perspectiva funcional, i.e., a partir de um *input* no modelo inicial obtem-se um *output* no modelo final. Um regra declarativa (i.e., uma regra que utiliza apenas lógica declarativa e/ou padrões) pode ser aplicada por vezes na direcção inversa. Este caso é pouco comum, comparativamente, sendo o mais usual que a regra não possa ser executada nas duas direcções. Por outro lado, a invertibilidade da transformação depende não só das regras de transformação como da própria invertibilidade da sequência de execução das regras. A inversão de um conjunto de regras pode resultar numa execução sem fim ou sem sentido.

Resumidamente as abordagens de transformação de modelos podem ser divididas em dois grupos: abordagens modelo-código e abordagens modelo-modelo. Estes dois grupos são agregam os seguintes tipos de abordagens:

[5] A palavra *direcção,* de onde são derivadas as palavras unidireccional e bidireccional é tomada no seu sentido lato: "lado para onde alguém se dirige" [Texto, 2007], tendo por isso o mesmo significado, neste contexto, da palavra *sentido*.

Abordagens Modelo-Código

- Abordagens baseadas na travessia de modelos: Consistem em implementar um mecanismo que percorre cada um dos elementos do modelo, gerando código para um ficheiro de texto. (Ex. Jamda, Epsilon [Rose et al, 08]);

- Abordagens baseadas em *templates:* Grande parte das ferramentas MDA suportam geração de código a partir de modelos, através de *templates*.

Abordagens Modelo-Modelo

- Abordagens de manipulação directa: Estas abordagens fornecem uma representação interna dos modelos e uma API de manipulação. São usualmente implementadas como uma *framework* orientada por objectos que pode conter definições genéricas para as transformações (e.g., classes abstractas ou interfaces para as transformações). As regras de transformação são, por isso, implementadas pelo próprio utilizador utilizando uma linguagem de programação como o Java. O código produzido é compilado juntamente com o código da *framework* produzindo o gerador (e.g., Jamda).

- Abordagens relacionais: Nestas abordagens são usadas definições declarativas das transformações. A definição usa um modelo fonte e um modelo alvo da transformação, e restrições sobre esses modelos. As regras assim definidas podem ou não ser "executáveis" (i.e., a execução da regra pode ter um efeito sobre o modelo alvo) ou servir apenas para a validação de um conjunto de condições.

- Abordagens por transformação de grafos: As regras de transformação de grafos consistem num padrão LHS e num padrão RHS que são aplicados, quando se dá a transformação, ao modelo sobre o qual vai ser feita a transformação. As ferramentas que utilizam esta abordagem (e.g., VIATRA, Atom, GreAT, UMLX, BOTL) encontram um modelo (ou parte dele) que verifica o padrão LHS e realizam as operações necessárias sobre esse modelo, de forma a que passe a ser verificado o padrão RHS.

- Abordagens orientadas para a estrutura: Nesta abordagem a transformação é dividida em duas fases: a primeira fase cria a estrutura hierárquica do modelo alvo, a segunda preenche os atributos e referencia os elementos necessários do modelo alvo. (Exemplo: OptimalJ)

- Abordagens híbridas: Estas abordagens combinam diversas técnicas das categorias anteriores. P.ex., a Transformation Rule Language (AST+) junta as abordagens declarativa e imperativa.

- Outras abordagens modelo-modelo: Como é possível serializar os modelos através de uma descrição textual como o XML, neste caso, a transformação pode ser vista como uma geração de código, eventualmente a partir de *templates.*

2.3.2 Geração automática

Para que a abordagem MDA possa dar frutos práticos é necessário que existam ferramentas que produzam automaticamente uma parte significativa dos artefactos de código necessários. Sem que tal aconteça, as vantagens do uso de modelos no processo de desenvolvimento são diluídas no esforço de implementar os diagramas no código final.

A geração automática de código é um processo utilizado para retirar ao programador (ou ao utilizador) actividades que podem ser feitas automaticamente. O facto de o código ser gerado automaticamente não implica necessariamente uma perda de qualidade do mesmo, tanto no que diz respeito a desempenho, legibilidade ou qualquer outro parâmetro de análise do mesmo. Na verdade, para alguns tipos de geração, o código gerado terá mesmo qualidade igual ou superior ao que seria produzido por uma pessoa. Um compilador é um bom exemplo. Devido ao facto de existirem optimizações padronizadas para as linguagens mais conhecidas, o código gerado é tanto ou mais eficiente do que o código máquina produzido manualmente (caso alguém ainda o faça).

O produto final da geração automática pode não ser apenas um texto escrito numa linguagem de programação. Caso seja necessário, a geração automática pode criar artefactos de diversos tipos (e.g., ficheiros de configuração, diagramas, definições de bases de dados). Por isso, nesta secção, a expressão *código gerado* não designa apenas o significado usual, anteriormente indicado. Será, em vez disso aplicada, para incluir qualquer artefacto passível de ser gerado automaticamente, desde que este tenha uma descrição textual[6].

A geração de código pode ser representada por um mapeamento entre duas ou mais linguagens. Existem diversas formas de se implementar este mapeamento que se traduzem em processos de geração distintos. Existem diversos padrões já definidos [Voelter, 03] para a geração automática que serão abordados em seguida. O estudo das formas de integração do código gerado com o código não gerado não se encontra ainda muito aprofundado pelo que se propõe igualmente uma classificação das mesmas.

Geração automática básica

Segundo uma abordagem simples, existe uma aplicação que transforma um *input* com o código inicial num *output* com o código final. O *input* pode ser um ficheiro com um modelo inicial escrito numa linguagem de modelação ou de programação. O modelo de *output* pode ser um programa, ou parte dele, um esquema de base de dados, um modelo, ou qualquer outro artefacto necessário ao desenvolvimento.

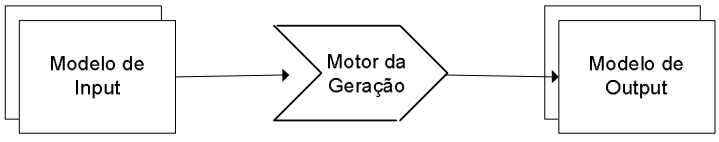


Figura 2.9 - Elementos da geração automática básica

[6] Um diagrama, desde que possa ser representado textualmente, é assim considerado como código.

Esta abordagem apresenta uma debilidade evidente na medida em que a lógica da geração está no motor da geração. Assim, se se quiser alterar a lógica da geração é necessário ter acesso ao código fonte do motor da geração e recompilá-lo. Esta abordagem é muito limitativa não sendo relevante para o presente trabalho.

Geração automática com *templates*

Neste tipo de geração a forma como a transformação se dá é determinada num conjunto de ficheiros externos ao motor da geração. Assim sendo, qualquer alteração que seja necessário realizar sobre a transformação deverá implicar uma alteração dos *templates* de geração. Os *templates* de geração são construídos usualmente com linguagens declarativas como o XSLT e podem ser alterados com qualquer processador de texto. No entanto a sua construção e alteração são processos quase totalmente manuais (com alguma possível assistência dos processadores de XML) o que leva a uma edição lenta e passível de erro.

Neste tipo de geração os templates podem incidir sobre um subconjunto do modelo de input, existindo uma filtragem prévia ao processo de geração.

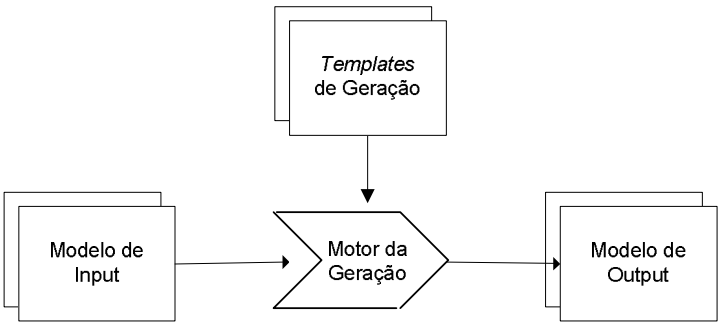


Figura 2.10 - Elementos da geração automática com *templates*

A produção de um gerador deste tipo é relativamente simples para exemplos de reduzida complexidade, no entanto torna-se difícil de conceber para exemplos complexos na medida em que os templates de geração, sendo realizados em XSLT, são difíceis de manter.

Geração automática com templates e um metamodelo

Por vezes utiliza-se uma geração automática de código a partir de um modelo de input em que o seu metamodelo é suficientemente abrangente para orientar a geração de código. Este metamodelo pode ser constituído por um conjunto limitado de elementos bem definidos e com um significado preciso que quando instanciados determinam a geração de código através de *templates* já definidos.

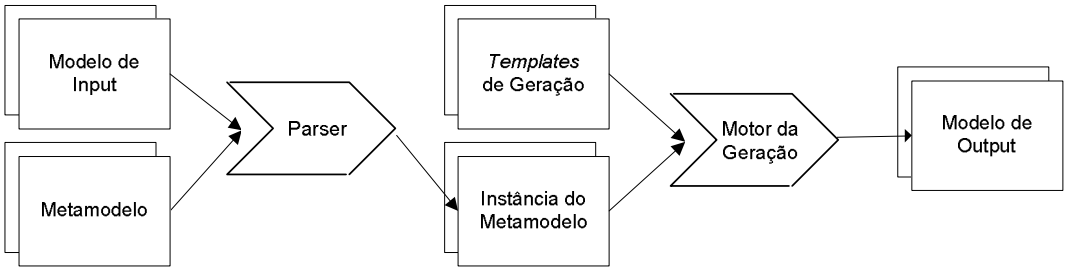


Figura 2.11 - Geração automática com *templates* e um metamodelo

O processo está dividido em duas fases. Primeiro realiza-se o *parsing* do modelo de input (p.ex., um ficheiro XMI correspondente a um modelo UML), tendo em atenção o metamodelo respectivo (p.ex., o Metamodelo do UML, ou modelo MOF de nível M2). Desta fase resulta uma instância do metamodelo. Aplicando os templates de geração para uma linguagem ou modelo alvo, sobre esta instância, obtém-se o modelo de output.

Este tipo de geração automática é na verdade uma extensão do tipo anterior sendo frequentemente utilizado para gerar código a partir de modelos UML.

Geração automática por uma API

Em alguns casos uma aplicação necessita de gerar um pequeno trecho de código que serve para cumprir uma função muito específica (P. Ex., na plataforma .NET é possível, utilizando o Reflection.Emit, gerar uma classe em tempo de execução do programa, utilizável por si próprio).

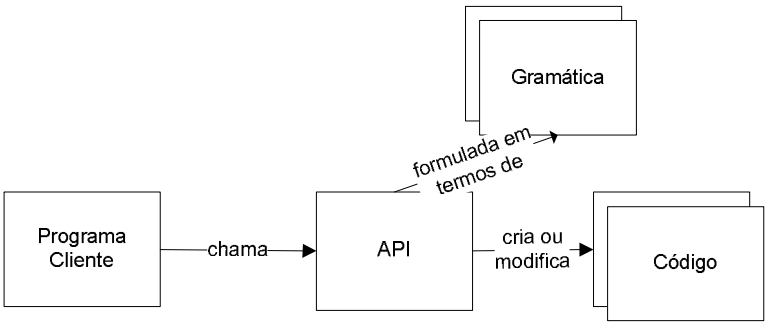


Figura 2.12 - Geração automática através de uma API

Neste tipo de geração não são utilizados *templates* e não existe qualquer espécie de modelo explicitamente. Existe porém um programa que utiliza a API para criar ou modificar o código em questão. Este tipo de geração tem uma utilização limitada a um número reduzido de casos.

Geração automática por código incluído

A geração automática por código incluído é usada quando existe uma parte do programa (ou outro artefacto) que é comum a um certo conjunto de gerações (p. ex., pode-se compilar um código C++ para diversas plataformas). Neste caso utiliza-se uma pré-compilação que acrescenta o código necessário dentro dos pontos de variabilidade definidos. A proporção de código variável deve ser

reduzida tanto quanto possível, para que o desempenho da geração e posterior compilação não sofra uma degradação significativa.

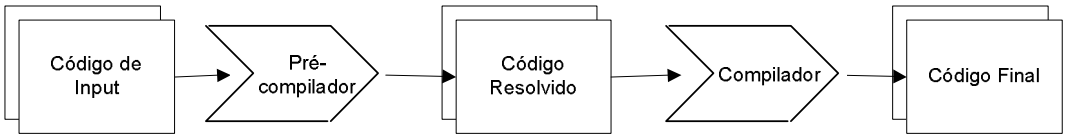


Figura 2.13 - Geração automática por código incluído

Para além dos aspectos de desempenho é importante notar que a manutenção deste tipo de código é difícil para exemplos complexos devido a incluir omissões resolvidas apenas em tempo de (pré)compilação.

Geração automática por atributos

Por vezes é necessário gerar um conjunto de artefactos a partir de uma única fonte. Um bom exemplo é a geração da documentação de uma classe em Java. O código Java é enriquecido com comentários especiais que possuem atributos. Quando é necessário gerar a documentação, existe uma análise do código total e os comentários servem de input à criação dos ficheiros da documentação (Javadocs, no caso da linguagem Java). Neste caso o mesmo ficheiro usado como código fonte do programa é igualmente o código fonte da documentação do mesmo.

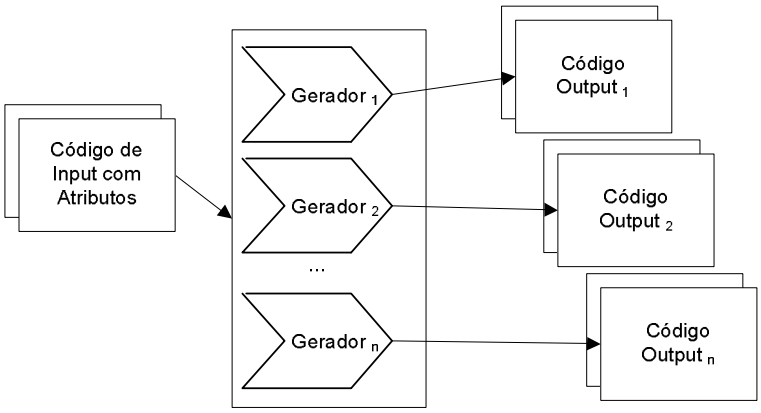


Figura 2.14 - Geração automática por atributos

Geração automática por fusão de código

Este tipo de geração automática por fusão de código (*code weaving*) utiliza-se quando é necessário fundir diversos artefactos num único que é o resultado final. Por vezes estes artefactos representam diversos aspectos do resultado final, daí este tipo de geração ser frequentemente relacionado com a programação orientada por aspectos (AOP, [Laddad, 03], [O'Regan, 04]), embora os dois conceitos sejam independentes. A fusão de código dá-se tendo em atenção um conjunto de meta-artefactos que constituem os meta-modelos dos artefactos de *input*.

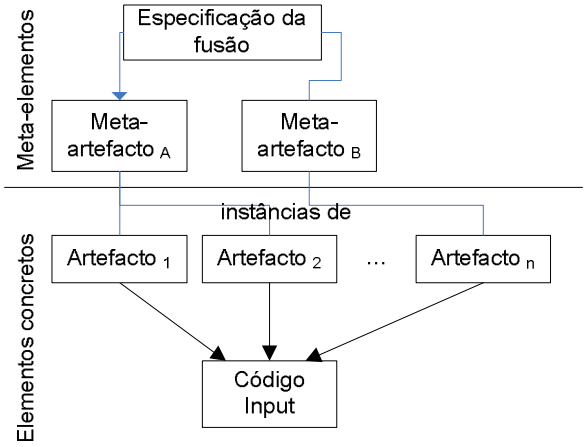


Figura 2.15 - Geração Automática por fusão de código

Integração de código gerado com código não gerado

Após o código ser gerado coloca-se o problema de o integrar com código não gerado. Existem diversas formas de o fazer dependendo de que tipo de código se considere. Como o código gerado pode não ser apenas uma parte de um programa, escrito numa qualquer linguagem de programação, pode pertencer a um dos níveis de abstracção a que o sistema está representado. Consoante se trata de código de uma linguagem de programação, ou uma descrição de um modelo, a forma de o integrar com elementos não gerados pode variar.

Considerando apenas a integração de código de uma linguagem de programação orientada por objectos, esta pode ser feita recorrendo a duas formas básicas com alguns subcasos: código gerado ligado e código gerado preenchido. No caso do código gerado ligado, as classes que constituem o código gerado são mantidas inalteradas, podendo existir uma referência das classes escritas manualmente a estas e vice-versa. A geração de código ligado inclui os seguintes subcasos:

- As classes geradas chamam as classes não geradas: É necessário que o código não gerado referenciado pelas classes não geradas exista antes da geração. Normalmente este tipo de integração aplica-se a uma geração de código que pode chamar bibliotecas ou pacotes de classes já existentes.

- As classes não geradas chamam as classes geradas: Neste caso presume-se que o código gerado é completado com outro não gerado através da sua chamada por outras classes escritas manualmente.

- As classes não geradas herdam as classes geradas: O código gerado pode definir um conjunto de propriedades e comportamentos que podem ser herdados pelas classes não geradas.

- As classes geradas herdam as classes não geradas: Esta é a situação inversa da anterior.

- É ainda possível em linguagens como o C# a criação de classes parciais que podem servir para separar a parte da classe gerada automaticamente do restante conteúdo.

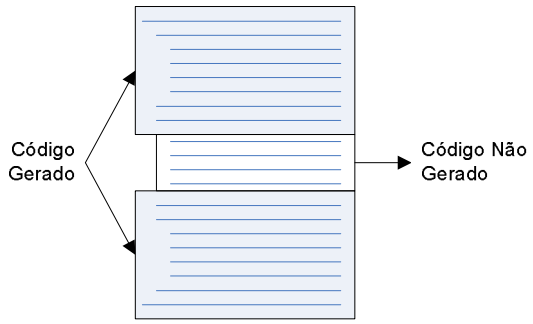


Figura 2.16 - Código gerado preenchido com código não gerado

Por outro lado, a aproximação de código gerado preenchido é relevante quando é útil acrescentar o código manual dentro das próprias classes. Esta abordagem é seguida em alguns ambientes de desenvolvimento (e.g. Sun Netbeans, Eclipse, Microsoft Visual Studio[7]). Nestes ambientes o programador define o modelo da interface gráfica através de um editor gráfico. Simultaneamente existe uma geração das classes, métodos, atributos e eventos necessários à execução dessa interface gráfica. Quando é necessário acrescentar um comportamento preenche-se um método que já tem a sua assinatura gerada, ou define-se um novo. O código gerado e o código não gerado estando presentes num ficheiro têm de continuar a poder ser distinguidos, de forma a se assegurar a manutenção do código não gerado aquando de uma regeração (p.ex. por alteração de um atributo de um objecto gráfico como a localização de uma caixa de texto). Esta abordagem é mais difícil de implementar do que a integração por código ligado se se pretender manter o código não gerado no momento da regeração. Nesse caso o ficheiro tem de ser analisado para se extrair a informação não gerada, guardando-se a sua localização relativa.

2.4 Modelo QVT

Com a visão MDA da OMG verificou-se a necessidade de encontrar linguagens que permitissem a transformação entre modelos, bem como a sua manipulação. Os modelos podem ou não ser guardados em XMI e não é a manipulação a esse nível que é mais importante de explicitar. Na verdade para isso já existia o XSLT, como anteriormente referido. A manipulação dos elementos conceptuais que fazem parte do metamodelo do UML é, essa sim, necessária para este novo tipo de manipulação. O QVT (Queries, Views and Transformations) foi mais uma concretização, por parte da OMG, de criar um standard que identificasse uma maneira de se expressar transformações entre modelos, de forma simples de entender para os utilizadores e executável para os computadores [Tratt, 03]. A sigla QVT denomina três conceitos:

- Pesquisa (Query) – Tendo um artefacto como input pode-se seleccionar um subconjunto dos seus elementos através de uma dada condição.

[7] Alguns ambientes de desenvolvimento como o Microsoft Visual Studio usam múltiplas abordagens de geração automática, consoante o o tipo de artefacto em questão.

- Vista (View) – É um modelo derivado de outros modelos.

- Transformação (Transformation) – Tendo como input um modelo pode-se alterá-lo ou criar um novo a partir deste.

Apesar de a denominação do QVT estar dependente destes três conceitos, o terceiro é o aspecto essencial. A definição das transformações deve ter algumas características que são importantes para a sua aceitação. Os artefactos gerados (diagramas, código, descrições, etc.) devem ser legíveis, tanto quanto possível, i.e., as transformações devem usar uma notação simples de escrever e de ler, tanto no que diz respeito ao texto como aos diagramas produzidos. Características da modelação orientada por objectos como a herança, a composição e a agregação devem ser possíveis de utilizar dentro das próprias definições do QVT. Deve ser possível realizar uma composição de duas ou mais transformações de forma a produzir-se uma transformação mais complexa. As transformações podem igualmente existir entre diversos níveis de abstracção tal como podem existir diversos níveis de abstracção para uma transformação. Assim sendo é necessário que exista um mecanismo de generalizar e especificar transformações. Deve também ser permitido reutilizar transformações, seja no mesmo projecto ou noutros.

Em Novembro de 2005 a OMG adoptou uma versão final do QVT [OMG, 05a], cujos os conceitos essenciais são explicados em seguida.

2.4.1 Arquitectura do QVT

A especificação do QVT tem uma natureza declarativa a par de uma natureza imperativa. A parte declarativa está dividida numa arquitectura de dois níveis que consistem em:

- Um metamodelo, denominado Relações (*Relations),* e uma linguagem, ambos suportando o mapeamento de padrões de objectos e os *templates* de criação de objectos. Os rastos[8] entre elementos do modelo, envolvidos numa transformação, são criados implicitamente.

- Um Núcleo *(Core),* constituído por um metamodelo e uma linguagem, foi definido com extensões mínimas ao EMOF e ao OCL.

A própria especificação do QVT dá como analogia para os elementos da Figura 2.17, a arquitectura Java. Segundo esta analogia a linguagem do Núcleo seria como o *Java Byte Code*, e a semântica do Núcleo seria como a especificação do comportamento para a *Java Virtual Machine.* A linguagem Relações desempenha o papel da linguagem *Java,* e a transformação padrão *RelationsToCore* seria similar à especificação do compilador que produz o *Byte Code.*

Para além da linguagem Relações e da linguagem Núcleo que possuem a mesma semântica, a níveis diferentes de abstracção, existem dois mecanismos que invocam implementações imperativas de transformações. Por um lado, existe a linguagem padrão, Mapeamentos Operacionais *(Operational Mappings)* e, por outro, implementações não normalizadas,

[8] *Trace* neste contexto indica uma relação que se estabelece entre elementos de dois modelos e que persiste ao longo do tempo.

denominadas Operações Caixa-preta MOF *(Black Box MOF Operations).* Cada relação define uma classe que será instanciada para permitir a rastreabilidade entre elementos do modelo a serem transformados, e tem um mapeamento "de um-para-um" para a definição de uma Operação implementado por um Mapeamento Operacional ou por uma caixa-preta.

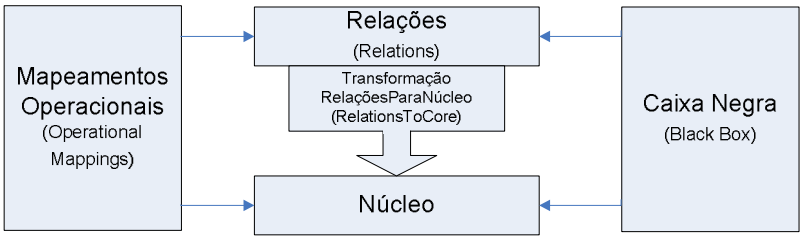


Figura 2.17 – Relações entre os metamodelos QVT

A linguagem de Mapeamentos Operacionais foi criada para permitir a implementação de uma forma imperativa. Fornece extensões ao OCL que produzem alterações nos valores tratados, tornando esta linguagem mais próxima das linguagens imperativas.

As implementações caixa-preta podem ser derivadas das Relações e podem ser usadas para enriquecer a linguagem padrão com novas funcionalidades específicas a um dado contexto de modelação. Por outro lado permitem que algumas partes da transformação possam ser opacas, i.e., que os detalhes da sua implementação não sejam visíveis para o utilizador. De acordo com a analogia anterior, a capacidade de invocar implementações *caixa-preta* e Mapeamentos Operacionais podem ser consideradas equivalentes a invocar operações via o *Java Native Interface (JNI).*

Podem ser considerados os seguintes cenários de execução do QVT: transformações para a verificação que os modelos estão relacionados de uma forma especificada; transformações num único sentido; transformações em dois ou mais sentidos[9]; capacidade de estabelecer relações entre modelos pré-existentes, quando desenvolvidos manualmente, ou através de outra ferramenta ou mecanismo; modificações incrementais (em qualquer direcção) onde um modelo relacionado é alterado após a execução inicial; e por fim, a capacidade de criar e eliminar objectos e valores, para além de especificar quais os objectos e valores que não devem ser modificados.

2.4.2 A linguagem de relações

Na linguagem de relações, a transformação entre modelos candidatos é especificada através de um conjunto de relações que devem existir para que a transformação seja bem sucedida. Um modelo candidato é qualquer modelo conforme com um *tipo de modelo*. Cada tipo de modelo possui um conjunto de elementos e um nome, de forma semelhante a um tipo de dados que num programa limita o conjunto de valores que uma variável pode possuir. Considere-se a seguinte declaração:

[9] São possíveis transformações em mais do que dois sentidos, considerando-se aquelas que relacionam mais do que dois elementos. São, porém, menos usuais pela dificuldade acrescida, tanto da sua especificação como da sua manutenção posterior.

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {
```

Nesta declaração, denominada *umlRdbms,* existem dois tipos de modelos candidatos: *uml* e *rdbms.* O modelo *uml* declara o *SimpleUML* como seu metamodelo, assim como o modelo *rdbms* declara o *SimpleRDBMS* como seu metamodelo. Uma transformação pode ser invocada tanto para verificar dois modelos quanto à coerência, como para modificar um dos modelos de forma a assegurar a coerência. As transformações que asseguram a coerência devem ser realizadas num sentido, i.e., devem seleccionar um dos modelos candidatos como alvo. A transformação é executada seleccionando primeiro os elementos que não cumprem as restrições e posteriormente tentando fazer com que estas sejam verificadas. Tal é conseguido através de eliminações, criações e modificações do modelo alvo.

As relações, numa transformação, definem restrições que devem ser satisfeitas pelos elementos dos modelos candidatos. Uma relação é definida por um ou mais domínios e um par de predicados *when* e *where*. Um domínio possui um *padrão* que pode ser visto como um grafo de objectos, das suas propriedades e das ligações entre eles, tendo como origem uma instância do tipo de domínio. No exemplo seguinte, são declarados dois domínios para os modelos *uml* e *rdbms* respectivamente. Cada um destes domínios especifica um padrão simples, neste caso um *package* com um nome e um *schema* com um nome, em que ambos os nomes estão ligados à variável *pn,* implicando por isso que devem ter o mesmo nome:

```
relation PackageToSchema /* realiza o mapeamento entre cada package
                         e um schema */
{
   domain uml p: Package {name=pn}
   domain rdbms s: Schema {name=pn}
}
```

Uma relação pode ser restringida por dois conjuntos de predicados, uma condição *when* e uma condição *where,* como demonstrado no exemplo *ClassToTable* em seguida:

```
relation ClassToTable /* map each persistent class to a table */
{
   domain uml c:Class {
      namespace = p:Package {},
      kind='Persistent',
      name=cn
   }
   domain rdbms t:Table {
      schema = s:Schema {},
      name=cn,
      column = cl:Column {
         name=cn+'_tid',
         type='NUMBER'
      },
      primaryKey = k:PrimaryKey {
         name=cn+'_pk',
         column=cl
      }
   }
   when {
      PackageToSchema(p, s);
   }
   where {
      AttributeToColumn(c, t);
   }
}
```

Neste exemplo o bloco *when* especifica quais as condições que devem ser verificadas (neste caso, existir a relação *PackageToSchema*) para que a relação *ClassToTable* seja ela própria válida. O bloco *where* especifica quais as condições que devem ser satisfeitas por todos os elementos que participam na relação, podendo restringir qualquer uma das variáveis tanto na relação como nos seus domínios. Neste caso sempre que a relação *ClassToTable* seja válida, também a relação *AttributeToColumn* deve ser também ela válida.

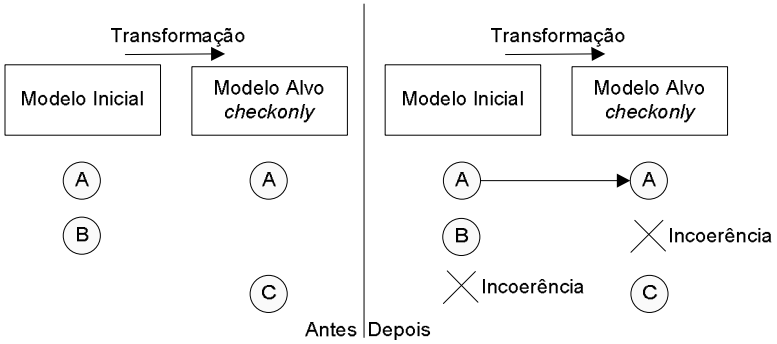


Figura 2.18 - Efeitos de uma transformação sobre um modelo alvo QVT marcado com a palavra reservada *checkonly*

Uma transformação possui dois tipos de relações: de nível mais alto (*top-level)* e de nível abaixo (*non-top-level).* A execução de uma transformação necessita que todas as relações de nível mais alto sejam verificadas, assim como todas as relações abaixo devem ser verificadas sempre que são invocadas directamente ou transitivamente por um bloco *where* de uma outra relação. Uma relação de nível mais alto tem a palavra reservada *top* que a distingue sintacticamente:

```
transformation umlRdbms (uml : SimpleUML, rdbms : SimpleRDBMS) {
   top relation PackageToSchema {…}
   top relation ClassToTable {…}
   relation AttributeToColumn {…}
}
```

O domínio alvo determina se uma relação deve ou não ser garantida, tal é concretizado através das palavras-chave *checkonly* e *enforced.* Quando uma transformação é garantida, no sentido de um domínio *checkonly,* este é simplesmente verificado para se saber se satisfaz a relação. Quando a transformação é executada no sentido de um modelo *enforced*, se a verificação falha o modelo alvo é alterado para que esta passe a ser verificada. No exemplo seguinte, o domínio para o modelo *uml* está marcado como *checkonly* e o domínio para o modelo *rdbms* está marcado *enforce.*

```
relation PackageToSchema /* map each package to a schema */ {
   checkonly domain uml p:Package {name=pn}
   enforce domain rdbms s:Schema {name=pn}
}
```

Se a execução for no sentido do *uml* (Figura 2.18) e existir um *schema* em *rdbms* para o qual não existe um *package* com o mesmo nome no *uml,* tal deve ser reportado como uma incoerência. Assim sendo não é criado um *package* porque o modelo *uml* não é garantido, apenas a sua existência é verificada. No entanto se a transformação for executada no sentido inverso (Figura 2.19), para cada *package* existente em *uml*, a relação verifica se existe um *schema* com o mesmo

nome, e caso não exista, é criado um novo *schema* nesse modelo com o nome dado *(pn).* Se a transformação for executada também no sentido do *rdbms* mas não existe um *package* com o mesmo nome em *uml,* esse esquema é eliminado do modelo *rdbms.* Genericamente, numa transformação, caso existam acções, estas são sempre produzidas sobre o domínio alvo.

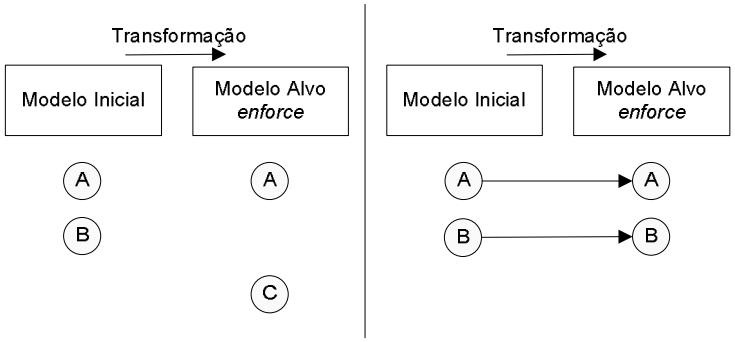


Figura 2.19 – Efeitos de uma transformação sobre um modelo alvo QVT marcado com a palavra reservada *enforced*

2.4.3 Correspondências entre padrões

No exemplo dado da relação *ClassToTable* existem diversas *expressões de templates de objectos (object template expressions)* que são usadas para realizar comparações nos domínios respectivos. A seguinte expressão de template de objectos está associada ao domínio *uml:*

```
domain uml c:Class {
   namespace = p:Package {},
   kind='Persistent',
   name=cn
}
```

Uma correspondência entre expressões de *templates* resulta numa ligação de elementos do modelo candidato a variáveis declaradas pelo domínio. Uma correspondência entre expressões pode ser realizada num contexto em que algumas das variáveis podem já ter ligações a elementos do modelo (e.g. resultantes da avaliação de um bloco *when* ou de outras expressões de templates). Nesse caso a correspondência é realizada encontrando-se as ligações necessárias para que todas as variáveis tenham uma ligação resolvida. No exemplo dado, todas as variáveis da expressão *(c, p* e *cn)* serão ligadas, iniciando-se pela variável *c* que é raiz do domínio. No mesmo exemplo, a variável *p* deve já ter uma ligação resultante da avaliação da expressão *PackageToSchema(p, s)* pertencente ao bloco *when*. A avaliação continua filtrando todos os objectos do tipo *Class* no modelo *uml*, eliminando todos os que não tenham nas suas propriedades os valores dados na expressão de template. Neste caso, todas as classes que não tiverem a propriedade *kind* marcada como "Persistent" são excluídas da comparação.

Para propriedades que são comparadas com variáveis, tais como *"name=cn"*, podem surgir dois casos: 1) Se a variável *cn* já tem uma ligação a um valor então cada classe que não tenha o mesmo valor para a propriedade *name* é excluída; 2) Se a variável *cn* está livre, i.e. ainda não tem o seu valor resolvido, terá uma ligação ao valor da propriedade *name*, para todas as classes que ainda não foram filtradas por outras comparações de propriedades. O valor de *cn* poderá ser usado noutro domínio, ou podem ser acrescentadas restrições no bloco *where*.

A comparação prossegue então para propriedades cujos os valores são comparados com expressões de templates encadeadas. O padrão de propriedade "*namespace = p:Package {}*" só encontrará classes cuja propriedade *namespace* seja uma referência não nula para um *Package.* Ao mesmo tempo a variável *p* será ligada a esse *Package.* No entanto, como no exemplo dado a variável *p* está já resolvida pelo bloco *when*, o padrão encontrará apenas as classes cuja propriedade *namespace* tem uma referência para o mesmo *package* que está ligado a *p*.

Podem ser realizados encadeamentos entre padrões tão complexos quanto o necessário, e a comparação e ligação de propriedades é feita recursivamente até existir um conjunto de tuplos correspondendo às variáveis do domínio e à sua expressão. No exemplo dado as variáveis *c, p* e *cn* formam um tuplo de ordem 3, e cada correspondência válida resulta num único tuplo que representa a ligação.

Numa invocação de uma relação dada, podem existir diversos valores que cumprem as restrições para cada expressão de *template*. A forma como esta multiplicidade de valores é tratada depende da direcção da execução.

Se a relação *ClassToTable* é executada com o *rdbms* como modelo alvo, então para cada resultado obtido (i.e. cada tuplo de ligações válidas) do domínio *uml,* tem de existir pelo menos uma correspondência do domínio *rdbms* que satisfaz o bloco *where.* Se para uma dada correspondência válida no domínio *uml*, não existe uma correspondência válida no domínio *rdbms,* como o domínio *rdbms* é garantido com a palavra reservada *enforced,* neste caso os objectos são criados e as propriedades são especificadas tal como indicado na expressão de template associada ao domínio *rdbms.* Da mesma forma, para cada correspondência válida do domínio *rdbms,* deve existir pelo menos uma correspondência válida do domínio *uml* que satisfaz o bloco *where* (tal acontece porque o domínio *uml* está marcado com a palavra reservada *checkonly).* Se essa correspondência no domínio *uml* não existir terá de se eliminar os objectos *rdbms* presentes na correspondência.

2.4.4 Chaves e criação de objectos usando padrões

Uma expressão de templates de objectos pode servir para a criação de objectos num modelo. Quando, para uma correspondência válida no padrão do domínio fonte, não existe uma correspondência válida no padrão do domínio alvo, então as expressões de templates de objectos do domínio alvo são usadas como base para a criação de objectos no modelo alvo. Por exemplo, quando a relação *ClassToTable* é executada com o *rdbms* como modelo alvo, a seguinte expressão de templates de objectos serve parcialmente como base para a criação de objectos no modelo *rdbms:*

```
t:Table {
   schema = s:Schema {},
   name = cn,
   column = cl:Column {name=cn+'_tid', type='NUMBER'},
   primaryKey = k:PrimaryKey {name=cn+'_pk', column=cl}
}
```

Este template indica que uma tabela deve ser criada com as propriedades *schema, name, column* e *primaryKey* com os valores indicados. Da mesma forma os templates associados a *Column* e *PrimaryKey* especificam a forma como os objectos correspondentes são criados.

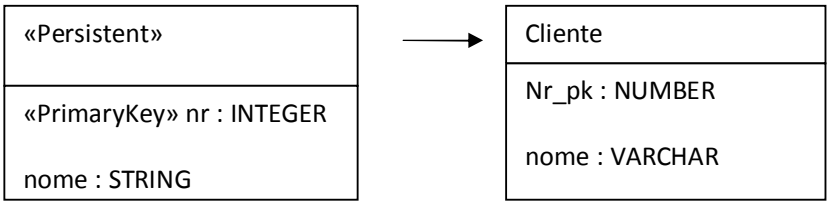


Figura 2.20 – Efeitos de uma transformação sobre um modelo alvo QVT

A criação das colunas, propriamente dita, é feita com a relação QVT especificada da seguinte forma:

```
relation AttributeToColumn
{
   checkonly domain uml c:Class
   {
      attribute=a:Attribute
      {
         name=an,
         type=p:PrimitiveDataType {name=pn}
      }
   }
   checkonly domain rdbms t:Table
   {
      column=cl:Column { name=an, type=sqltype}
   }
   where
   {
      sqltype = if (pn ='INTEGER')
                then 'NUMBER'
                else 'VARCHAR' endif
   }
}
```

No entanto, é necessário assegurar que não se criam objectos duplicados quando os objectos necessários já existem. Nesses casos apenas se pretende realizar uma alteração dos objectos existentes. O MOF permite que cada propriedade de uma classe seja nomeada com identificador. No entanto, para muitos modelos, tal é insuficiente para identificar os objectos univocamente. O metamodelo das relações QVT introduz o conceito de chave (*Key*) que define um conjunto de propriedades de uma classe que identificam univocamente uma instância de um objecto de uma classe do modelo. A classe pode ter diversas chaves (tal como no modelo relacional).

No exemplo dado, na relação *ClassToTable*, pode ser necessário especificar que nos modelos *simpleRDBMS* uma tabela é univocamente identificada por duas propriedades que podem ser o seu nome (propriedade *name)* e o esquema à qual pertence (propriedade *schema),* conforme a expressão:

```
key Table {schema, name};
```

As chaves podem ser usadas no momento da criação do objecto, e.g. se uma expressão de templates de objectos tem propriedades correspondentes a uma chave da classe associada, então a chave é usada para localizar um objecto no modelo. Um novo objecto apenas é criado quando um objecto correspondente não existe.

No exemplo dado, considere-se que existe uma classe persistente com o nome "foo" no modelo *uml* e uma tabela com o mesmo nome no *schema* correspondente do modelo *rdbms,* não tendo a tabela, no entanto, valores para as propriedades *column* e *primaryKey.* Neste caso o modelo *rdbms* não tem uma correspondência para o padrão associado a *Table,* na medida em que duas das suas propriedades não correspondem, sendo necessário criar objectos que satisfaçam a relação. No entanto, como a tabela existente corresponde no que diz respeito às propriedades chave (*name* e *schema*), não há necessidade de criar uma nova tabela; basta actualizar as ocorrências das propriedades *column* e *primaryKey* da tabela.

2.4.5 Restrições sobre expressões e propagação de alterações

De forma a garantir que as transformações são executadas, as expressões que ocorrem numa relação têm de satisfazer as seguintes condições:

- Deve ser possível organizar as expressões que ocorrem no bloco *when,* no domínio inicial e no bloco *where,* numa ordem sequencial que contém apenas os seguintes tipos de expressões:

Uma expressão da forma

```
<object>.<property> = <variable>
```

Onde <variable> é uma variável livre e <object> pode ser uma variável ligada a uma expressão de template do domínio oposto ou uma variável que obtém a sua ligação de uma expressão anterior na ordem das expressões.

Uma expressão da forma:

```
<object>.<property> = <expression>
```

Onde <object> pode ser tanto uma variável ligada a uma expressão de templates de objectos de um padrão de domínio ou uma variável que obtém a sua ligação de uma expressão anterior na ordem das expressões. Não existem ocorrências de variáveis livres em <expression> (todas as variáveis têm de estar ligadas nas expressões anteriores).

Nenhuma outra expressão tem variáveis livres, sendo que todas as variáveis foram já ligadas em expressões anteriores.

- Deve ser possível organizar as expressões que ocorrem no domínio alvo numa ordem sequencial que contém os seguintes tipos de expressões:

Uma expressão da forma:

```
<object>.<property> = <expression>
```

Onde <object> pode ser tanto uma variável ligada a um objecto como uma expressão de template do padrão de domínio ou uma variável que obtém a sua ligação de expressões anteriores. Não existem ocorrências de variáveis livres em <expression>.

Nenhuma outra expressão tem variáveis livres, sendo que todas as variáveis foram já ligadas em expressões anteriores.

Nas relações, o efeito de propagar uma alteração de um modelo inicial para um modelo alvo é semanticamente equivalente a executar a transformação sem que o modelo alvo exista. A semântica da criação e eliminação de objectos garante que apenas as partes necessárias de um modelo alvo são afectadas pelas alterações: 1) A semântica da selecção de objectos assegura que apenas os elementos do modelo que satisfazem as relações é que são tocados; 2) A selecção de objectos baseada em chaves assegura que os objectos existentes são alterados quando necessário; 3) A semântica da eliminação garante que um objecto é eliminado apenas quando nenhuma outra regra necessita que ele exista.

Para as transformações que são realizadas sobre o mesmo modelo (i.e. o modelo candidato inicial e o modelo candidato final estão ligados ao mesmo modelo em tempo de execução) garante-se que: 1) Uma relação é reavaliada depois de cada modificação induzida numa instância do modelo; 2) Uma avaliação de uma relação termina quando todas as instâncias do padrão satisfazem a relação.

2.4.6 Integração de operações "caixa-preta" com relações

Uma relação pode ter uma implementação "caixa-preta" operacional associada para garantir um domínio[10]. A operação "caixa-preta" é invocada quando a relação é executada, no sentido do domínio marcado como *enforced* e a relação é avaliada com o valor *false* de acordo com a semântica de avaliação. A operação invocada é responsável por realizar as alterações necessárias ao modelo de forma a satisfazer a relação especificada. É criada uma excepção em tempo de execução se uma relação é avaliada com o valor *false* após a operação terminar. A assinatura da operação pode ser derivada da especificação do domínio da relação, e.g. um parâmetro de saída correspondente ao domínio *enforced,* e um parâmetro de entrada correspondente a um domínio de entrada.

As relações que podem ser implementadas por operações de mapeamento e operações "caixa-preta" têm as seguintes restrições: 1) O seu domínio deve ser primitivo ou conter um *template* de objectos simples (sem sub-elementos); 2) Os blocos *when* e *where* não devem definir variáveis. Estas condições garantem que não é feita nenhuma avaliação prévia ou posterior das restrições depois da invocação da operação.

2.4.7 Semântica das relações

Uma relação tem a seguinte estrutura abstracta:

```
Relation R
{
   Var <R_variable_set> // declaration of variables used
                        // in the relation
   [checkonly | enforce] Domain:<typed_model_1>
   <domain_1_variable_set> // subset of <R_variable_set>
   {
      <domain_1_pattern> [<domain_1_condition>]
   }
   ...
```

[10] Esta utilização simultânea de dois paradigmas de programação (i.e., imperativo e declarativo) é muito útil sendo já usada noutras linguagens (e.g., C# ou VB.NET, ambos com LINQ e com expressões regulares)

```
   [checkonly | enforce] Domain:< typed_model_n>
   <domain_n_variable_set> // subset of <R_variable_set>
   {
      <domain_n_pattern> [<domain_n_condition>]
   } // n >= 2
   [when <when_variable_set> <when_condition>]
   [where <where_condition>]
}
```

Verificam-se as seguintes propriedades:

- <R_variable_set> é o conjunto de variáveis que ocorre na relação.
- <domain_k_variable_set> é o conjunto de variáveis que ocorrem num domínio k. É um subconjunto de <R_variable_set>, sendo k=1..n.
- <when_variable_set> é o conjunto das variáveis que ocorrem no bloco *when.* É um subconjunto de <R_variable_set>.
- A intersecção dos conjuntos de variáveis dos dois domínios não é necessariamente nula, i.e. uma variável pode ocorrer em múltiplos domínios.
- A intersecção de um conjunto de variáveis dum domínio e de um conjunto de variáveis de um bloco *when* não é necessariamente nula.
- O termo <domain_k_pattern> refere-se ao conjunto de restrições implicadas pelo padrão do domínio k. Note-se que um padrão pode ser visto como um conjunto de variáveis e um conjunto de restrições que os elementos do modelo, ligados a essas variáveis, devem satisfazer de forma a que a correspondência possa ser válida.

No exemplo dado, existe o seguinte padrão:

```
c: Class {kind=Persistent, name=cn, attribute=a:Attribute{}}
```

A seguinte restrição está implícita:

```
c.kind = 'Persistent' and c.name = cn
                      and c.attributes->includes(a)
```

Pode afirmar-se ainda que a avaliação de uma relação no sentido do modelo k (i.e. tendo o modelo k como modelo alvo) tem a seguinte semântica: para cada ligação válida entre variáveis do bloco *when* e de variáveis de domínios diferentes de *k*, satisfazendo a condição *when* e padrões e condições do domínio inicial, tem de existir uma ligação válida entre as restantes variáveis do domínio *k* sem ligação a valores e que satisfazem o padrão *k* e a condição *where.*

A obrigação da relação no sentido do modelo alvo *k* com a palavra reservada *enforced* tem a seguinte semântica: para cada ligação válida de variáveis da condição *when* e de variáveis de domínios diferentes do domínio alvo *k,* tais que satisfazem a condição *when* e as condições e padrões do domínio inicial, se não existe uma ligação válida das restantes variáveis do domínio *k* não ligadas a valores que satisfazem o padrão do domínio *k* e a condição *where,* então dever-se-á realizar as operações necessárias sobre os objectos (criar ou então seleccionar e modificar caso os objectos já existam) e alterar as propriedades necessárias. Se um objecto é seleccionado do

modelo ou criado de raiz, depende se o modelo contém já um objecto que corresponde às propriedades chave, caso existam, especificadas no *template* definição de objectos. É um erro, se a avaliação da expressão de *template* resulta numa atribuição de um valor a uma propriedade já atribuída por outra regra dentro da execução da mesma transformação, indicando uma especificação incoerente. Para tipos primitivos, os valores entram em conflito quando são diferentes. Uma atribuição de um objecto com uma ligação de multiplicidade "1" entra em conflito se o objecto a ser atribuído é diferente daquele que já existe.

Da mesma forma, para cada ligação válida de variáveis do padrão do domínio *k* que satisfazem a condição do domínio *k,* se não existe uma ligação válida entre as variáveis do bloco *when* e os domínios iniciais que satisfazem essa condição *when,* os padrões dos domínios iniciais e a condição *where,* e pelo menos um dos domínios está marcado como *checkonly (*ou *enforce*, que garante a avaliação), então devem ser eliminados os objectos ligados a variáveis do domínio *k* quando a seguinte condição é satisfeita: apagar um objecto apenas se não for necessária a sua existência por outra ligação válida nos domínios iniciais.

2.4.8 Semântica da correspondência entre padrões

De forma a simplificar a descrição desta semântica considerar-se-ão alguns padrões explicitados através dos seus modelos gráficos.

Figura 2.21 – Um modelo UML (nível 2 MOF)

A Figura 2.21 representa um modelo UML em que são definidos de forma simplificada classes e atributos. Trata-se na verdade de um metamodelo, ou seja um modelo de nível M2, segundo a arquitectura MOF. As suas instâncias são classes e atributos UML, podendo ter eles próprios, por isso, instâncias.

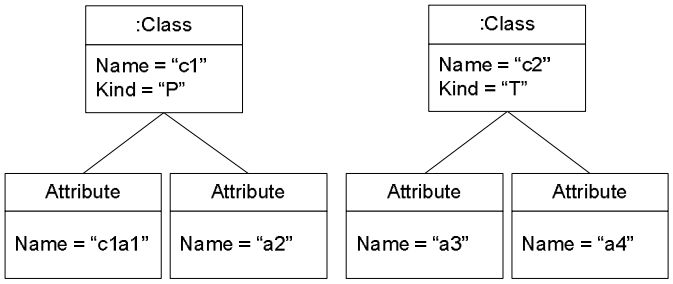


Figura 2.22 – Um modelo com instâncias UML (nível 1 MOF)

Tanto a Figura 2.22 como a Figura 2.23 representam os mesmos conceitos embora com diagramas diferentes. No primeiro caso trata-se de um diagrama de objectos derivado do diagrama de classes da Figura 2.21. Como as instâncias representadas são elas próprias conceitos UML o diagrama é na verdade um isomorfismo da representação da Figura 2.23.

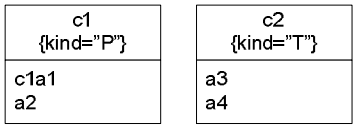


Figura 2.23 – Um modelo UML (nível 1 MOF)

Para simplificar a definição desta semântica considerar-se-á que um padrão tem a seguinte estrutura abstracta:

```
Pattern =
{
   e1: <classname1>, e2: <classname2> … en:<classnameN>
   l1: <assoc1> (ei, ej) … lm:<assocM>(eu, ew)
   where <predicate> }
```

Um padrão pode ser visto como um grafo onde os elementos do padrão, e_1, e_2, …e_n, com tipos <*classname*$_1$>, <*classname*$_2$>, … <*classname*$_N$> respectivamente, são os nós e as ligações do padrão l_1, l_2, …l_m são as arestas. Os predicados são expressões booleanas que se podem referir aos elementos do próprio padrão. Os predicados podem igualmente referir-se a variáveis diferentes dos elementos do padrão; estas são as variáveis livres do padrão. Um padrão é usado para encontrar correspondências entre sub-grafos num modelo. Um sub-grafo de um modelo consistindo em objectos o_1, o_2, … o_n, corresponde ao padrão descrito se e só se:

- o_1 é do tipo <*classname*$_1$> ou um dos seus subtipos, e o_2 é do tipo <*classname*$_2$> ou um dos seus subtipos, e assim sucessivamente…
- o_i e o_j estão ligados por uma associação <*assoc*$_1$>, assim como o_u e o_w estão ligados por uma associação <*assoc*$_M$>, e assim sucessivamente…
- Existe uma ou mais ligações dos valores para variáveis livres tais que <*predicate*> é avaliado com o valor *true* quando referências a e_1, e_2, …e_n são substituídas por o_1, o_2, … o_n, respectivamente.

Uma vez que o padrão tenha sido correspondido, cada e_i é ligado ao objecto do modelo o_i correspondente e cada variável livre v_i é ligada ao valor com o qual o <*predicate*> foi avaliado enquanto se realizava a comparação.

```
Pattern
{
   c1: Class, a1: Attribute
   l1: attrs (c1, a1)
   where c1.name = X and a1.name = X + Y
}
```

No exemplo anterior, *X* e *Y* são variáveis livres. O único sub-grafo da Figura 2.22 que corresponde ao padrão é <*c1*, *c1a1*>. Esta correspondência liga *X* com *c1* e *Y* com *a1.*

Os elementos de um padrão podem ser colecções (e.g. *Set*, *OrderedSet*, *Bag* e *Sequence*). Se o tipo de e_i é uma colecção do tipo <*classname*$_i$> então um sub-grafo do modelo corresponde ao padrão se e só se:

o_i é uma colecção de objectos do modelo com o tipo <*classname*$_i$>;

Não existe nenhuma colecção de objectos do modelo, de tipo <*classname*$_i$>, denominada o_j tal que cada o_i é uma sub-colecção de o_j e cada substituição de o_i por o_j satisfaça o outro critério de comparação;

Se l_j: <*assocname*>(e_m , e_i) é uma ligação no padrão e o tipo de e_m é <*classname*$_m$> então cada elemento de o_i deve estar ligado a o_m pela associação <*assocname*>;

Se l_j: <*assocname*>(e_m , e_i) é uma ligação no padrão e o tipo de e_m é também um conjunto de <*classname*$_m$> então cada elemento de e_i deve estar ligado a cada elemento de o_i tem de estar ligado a cada elemento de o_i pela associação <*assocname*>

No exemplo seguinte os dois sub-grafos <c1, {c1a1, a2}> e <c2, {a3, a4}> do modelo de instâncias da Figura 2.22 correspondem ao padrão:

```
Pattern
{
   c1: Class, a1: Set(Attribute)
   l1: attrs (c1, a1)
   where TRUE
}
```

Note-se que os padrões têm de ter pelo menos um elemento. Não é válida, por isso, uma definição vazia de um padrão. Igualmente só são admitidas colecções de primeira ordem, i.e. os elementos do padrão não podem ser conjuntos de conjuntos.

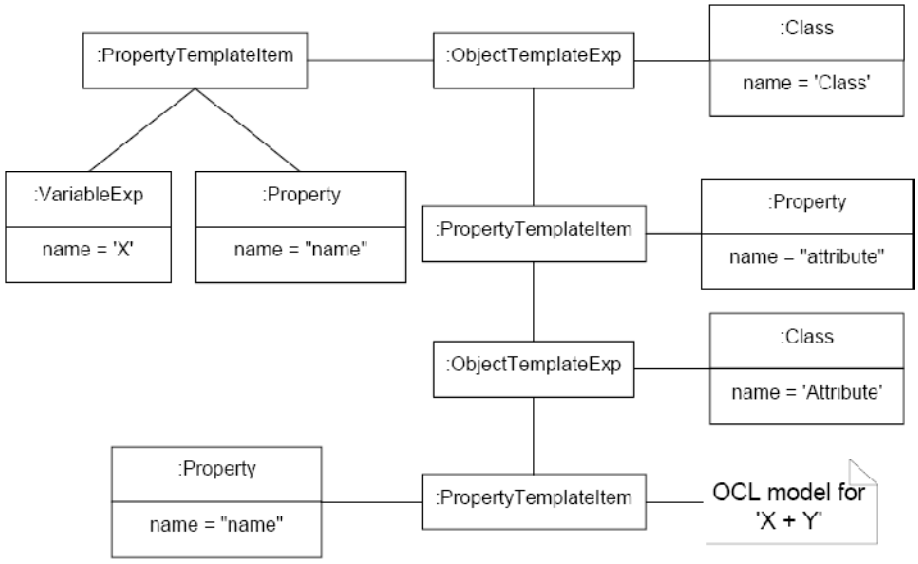


Figura 2.24 – Exemplo de instância de um padrão

Quanto ao metamodelo das expressões de templates pode dizer-se que uma *expressão de template de objectos* constitui o modelo para um *elemento do padrão* com o seu tipo como *classe referenciada*. Da mesma forma uma *expressão de template de colecção* realiza o modelo de um *elemento do padrão* cujo o tipo é a colecção da *classe referenciada*. Por sua vez, um *item de template de propriedade* ligando duas *expressões de templates de objectos* realiza o modelo da ligação.

A parte do padrão que diz respeito ao predicado é uma conjunção das seguintes expressões:

- Uma expressão da forma "referredProperty.name = value";
- A expressão das partes associadas à expressão do template de colecção;
- Uma expressão indicando que nenhuma colecção é vazia;
- A expressão where associada a cada expressão de template.

Considerando-se o seguinte padrão especificado através da sintaxe concreta:

```
Class {name=X, attribute = Attribute {name = X + Y}}
```

O modelo de instâncias para o exemplo acima é dado pela Figura 2.24. Para simplificar o diagrama, a estrutura detalhada da expressão OCL foi substituída por uma *expressão de variáveis* e uma *nota* representando a expressão completa. O modelo de instâncias apresenta duas *expressões de templates de objectos* correspondendo a *Class* e a *Attribute.* O nó correspondente a *Class* está associado a dois nós que são *items de template de propriedade* correspondentes às duas propriedades *name* e *attribute.*

A estrutura do padrão correspondente à Figura 2.24 pode ser dada textualmente por:

```
Pattern
{
   dummy: Class, dummy1: Attribute
   dummy2: attribute (dummy, dummy1)
                freevars X, Y
   where dummy.name = X and dummy1.name = X + Y
}
```

No exemplo anterior é evidente a maior simplicidade da descrição textual quando comparada com a estrutura gráfica do padrão correspondente. A definição de uma DSL para o tratamento de padrões é um dos possíveis desenvolvimentos futuros desta área.

A descrição do QVT dada até este ponto é independente da tecnologia usada para implementar uma ferramenta deste tipo. Podem ser tomadas diferentes opções quanto a aspectos tão importantes como: transacções, repositório de dados, cálculo concorrente e quanto à rastreabilidade dos artefactos produzidos. Este último aspecto é central a este trabalho sendo desenvolvido no capítulo seguinte.

2.5 Comentários Finais

O QVT não é a única proposta existente no campo da transformação de modelos aplicando metamodelação. Existem outras como o Tropos [Perini et al. 05], RM-ODP [Akehurst, 04], DSMDA [Agrawal, 03], CDIF-EXPRESS [Davis, 98], sNets [Bézivin, 98b] [Lemesle, 98], VMTS [Levendovszky, 04] menos abrangentes ou com menor aceitação pela indústria. Sendo esta uma área de investigação ainda sem consenso quanto ao rumo a seguir, tem-se verificado uma evolução dos conceitos referentes à transformação de modelos. A própria metamodelação pode ser enquadrada

na área das ontologias[11] [Bézivin, 98a]. Existem ainda linguagens com um âmbito semelhante ao QVT que usam diferentes aproximações ao problema, e.g. Epsilon [Rose et al, 08] usa *templates* para realizar as transformações após percorrer os elementos dos modelos.

No que diz respeito à execução de transformações sobre modelos, esta produz relações entre elementos do modelo fonte e elementos do modelo alvo. Algumas aproximações permitem realizar a rastreabilidade até certo ponto (e.g., CDI, IOPT) enquanto outras assumem que estas relações são semelhantes a quaisquer outras relações entre elementos dos dois modelos (e.g., VIATRA [Varro, 02], GreAT [Agrawal, 03]). Algumas aproximações com suporte para o tratamento da rastreabilidade necessitam que o utilizador codifique manualmente estas relações nas regras de transformação (e.g., CDI) enquanto outras criam estas ligações automaticamente (e.g., IOPT). No caso do suporte automático, esta aproximação pode ainda dar como opção o controlo do número de relações que são criadas (de forma a limitar o volume de dados envolvidos). Existe ainda a escolha do local onde são guardadas as relações de rastreabilidade, e.g., no modelo fonte e/ou no modelo alvo, ou em separado.

O modelo QVT não inclui uma definição da forma como a coerência é mantida entre os diferentes artefactos. O próprio conceito de artefacto não é claro, na medida em que o QVT não tem como propósito indicar uma forma de implementar a coerência. Apenas define uma linguagem gráfica e uma linguaguem textual que, sendo implementadas, fornecem a forma de interagir com a ferramenta. A formalização, definição e implementação dos diferentes aspectos do tratamento da coerência e da capacidade de realizar a rastreabilidade constitui a principal justificação para esta proposta.

[11] Bézivin [Bézivin, 98] define ontologia como "uma descrição explícita e precisa dos conceitos e relações que existem num domínio particular tal como uma dada organização, uma área de estudo, etc".

3 Rastreabilidade Reactiva

Neste capítulo são introduzidos os conceitos fundamentais para o estudo da rastreabilidade reactiva. O capítulo evolui gradualmente de uma perspectiva conceptual, através de uma descrição informal recorrendo a diagramas UML, para uma formalização, e desta para aspectos mais próximos dos detalhes de implementação que serão explicitados no capítulo seguinte.

3.1 Introdução

O processo de desenvolvimento de aplicações produz um número elevado de artefactos. Estes não devem ser considerados apenas meios para atingir um fim, neste caso a aplicação produzida. Todos os diagramas, descrições textuais, ou outros artefactos necessários à compreensão do sistema, são também parte da solução do problema. Assim sendo, não devem ser desprezados após a entrada da aplicação em fase de produção. Os artefactos devem, pelo contrário, permanecer actualizados apesar de todas as alterações produzidas pela utilização da aplicação ou por outros factores externos (e.g., novos requisitos resultantes de mudança no negócio). A justificação para a afirmação anterior é evidente se se considerar que a vida útil de uma aplicação pode envolver um número elevado de versões e, em particular, se o seu desenvolvimento e manutenção forem dirigidos por modelos. Se os modelos não reflectirem o estado actual do código implementado, o desenvolvimento ou manutenção podem ser feitos sem que os modelos sejam actualizados. As consequências para o futuro da aplicação são o desfasamento entre os modelos que deveriam traduzir uma realidade, entretanto alterada, e o código que deveria instanciar uma solução, entretanto desactualizada. Nesse caso, estando os modelos desactualizados, de nada servem, constituindo apenas um conjunto de esquemas e textos desalinhados com a realidade. Porém, à medida que forem sendo produzidas novas alterações, a complexidade das aplicações for aumentando, as soluções encontradas tornarem-se cada vez mais particulares, e as equipas forem enquadrando novos elementos, será cada vez mais questionável a opção tomada. Neste ponto pode já ser tarde demais para se voltar a actualizar os modelos anteriormente criados. Pode até ser mais simples criar uma nova aplicação de raiz, eliminando os erros de concepção entretanto encontrados. Apesar desta constatação, a experiência mostra que nem sempre existe uma consciência por parte dos profissionais do sector para a importância da totalidade dos artefactos [Iivari, 96]. O problema não se resume a tornar a documentação de uma aplicação coerente com a aplicação propriamente dita (tanto no que diz respeito às suas funcionalidades como ao código executável respectivo). Pode acontecer igualmente que duas aplicações partilhem uma parte de um modelo conceptual que deve também ser mantido e ambas devem reflectir eventuais alterações ao modelo comum, ou pelo menos deve existir uma forma de entender em que é que uma aplicação não está de acordo com o modelo. De uma forma

genérica, é importante ser assegurado que cada um dos artefactos existentes não entra em contradição com os restantes, ou seja que se mantém coerente.

3.2 Conceitos Básicos

No contexto do presente trabalho, designa-se por **projecto** não só um dado programa, ou conjunto de programas, como a entidade que agrega todos os artefactos que estão relacionados directamente com ele, como por exemplo: a) a sua justificação, b) o tratamento de dados que realiza, c) os aspectos tecnológicos que lhe estão inerentes, d) o envolvimento humano necessário e e) a documentação necessária à sua produção, manutenção e operação. O projecto pode incluir, e.g., todos os documentos produzidos sobre o domínio que trata, o código fonte dos programas necessários, os dados tratados por um SGBD, documentos sobre recursos e a sua afectação.

Designa-se assim por **elemento** do projecto**,** qualquer conceito que seja necessário à definição, justificação, ou operação, da aplicação. Pode considerar-se um elemento do projecto como um conceito abstracto concretizado num ou mais **artefactos**. Designa-se por **artefacto** uma qualquer realização física[12] de um ou mais elementos do projecto**.** Um elemento do projecto pode existir em mais do que um artefacto e um artefacto pode realizar mais do que um elemento do projecto. Alternativamente, pode considerar-se artefacto qualquer conjunto de símbolos, organizado segundo uma estrutura conhecida, que existe num dado período de tempo, num dado suporte, com o propósito de instanciar um conjunto de elementos do projecto. Considera-se que um artefacto pode incluir outros artefactos. Por exemplo, a representação duma classe "Pessoa" num diagrama de classes "Domínio A" pode ser considerada ela própria um artefacto, incluído noutro artefacto, nesse caso. No mesmo exemplo, o elemento do projecto "classe de domínio Pessoa" é realizado através do artefacto "classe de domínio Pessoa no diagrama de classes Domínio A".

Um **suporte** é qualquer meio em que possa existir um certo tipo de artefactos (e.g., ficheiro de texto, tabelas de definição de uma base de dados num SGBD, repositório de uma ferramenta CASE). A granularidade do suporte pode variar, consoante o tipo de artefacto em causa.

Considera-se **modelo** um conjunto de elementos do projecto, que podem incluir relações entre eles (elas próprias sendo conceitos). Um modelo é uma simplificação da realidade constituída por um domínio. Neste contexto, um modelo deve incluir todos os elementos relevantes (do problema tratado pelo projecto) e apenas estes.

Considera-se **vista** (ou perspectiva) uma forma de seleccionar, relacionar ou representar, um conjunto de conceitos da aplicação. A vista poderá ser indicada resumidamente como qualificador de um modelo (e.g., *modelo físico* que representaria abreviadamente uma *perspectiva física de alguns elementos da aplicação,* ou ainda *um modelo dos elementos da aplicação com realização física).*

[12] Obviamente a "realização física" é aqui indicada metaforicamente visto indicar apenas a existência de uma ocorrência de um conceito que muitas vezes é intangível.

Designa-se por **diagrama** uma representação gráfica de uma parte de um modelo, segundo uma dada vista. Um diagrama é também um artefacto com a informação gráfica e lógica dos conceitos representados.

Um **metamodelo** é um modelo que define um conjunto de elementos (conceitos) presentes em modelos. Um metamodelo pode também ser considerado um modelo (herdando a necessidade de ser definido[13]).

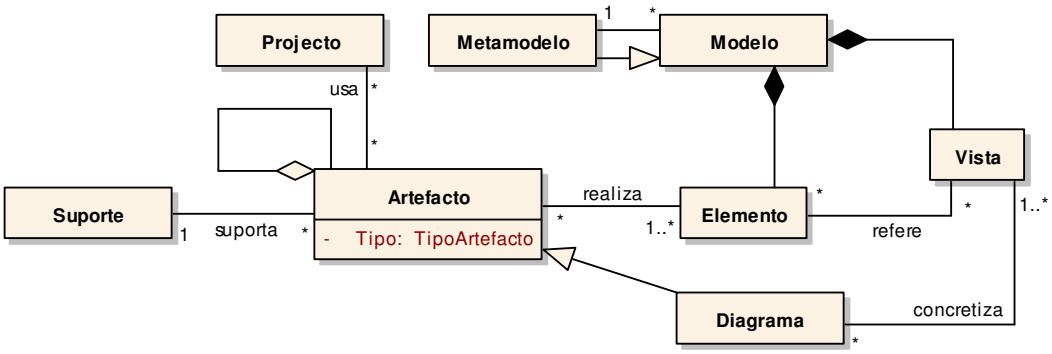


Figura 3.1 – Modelo dos conceitos básicos necessários ao âmbito da rastreabilidade

Note-se que o diagrama não é o único tipo de artefactos existente. Como foi anteriormente referido, todas as descrições textuais e tabelas relevantes para o projecto, mesmo que apenas para a sua documentação, também são artefactos. Podem igualmente ser artefactos outros objectos num contexto aplicacional distinto (e.g., um certo parágrafo incluído num documento, uma tabela num SGBD relacional, uma imagem numa ferramenta de edição gráfica).

Existe actualmente uma infraestrutura arquitectural IEEE-1471 [Hilliard, 07] cujo modelo se apresenta na Figura 3.2 e que tem alguns pontos de contacto com a proposta do ReacT-MDD. De forma a permitir entender a relação entre os dois modelos foram mantidos os termos em Língua Inglesa no primeiro caso.

Uma *Architectural Description* representa a descrição de uma *Architecture* de um *System,* sendo um "conjunto de produtos que documentam a arquitectura" [Hilliard, 07]. Assim sendo uma *Architectural Description* é um artefacto, tal como o é uma *View,* um *Rationale* e o próprio *Model.* Os conceitos estão representados por *System, Architecture, Mission* e *Environment,* dos quais os dois primeiros estão contidos na noção de modelo e os restantes são irrelevantes neste contexto.

[13] A definição pode parecer cíclica se se considerar o metamodelo mais geral. A solução é considerar um metamodelo suficientemente genérico que consiga descrever o modelo implícito. Esta solução diferencia o significado do modelo do conjunto de símbolos através do qual é representado. Note-se ainda que apesar de uma definição de um modelo se considerar completa, esta recorre eventualmente a um conjunto de símbolos definidos por outra linguagem (e.g., pertencentes a uma língua natural, símbolos gráficos como linhas e polígonos) definida implicitamente.

Com a comparação anterior demonstra-se a necessidade da criação do modelo da Figura 3.1.

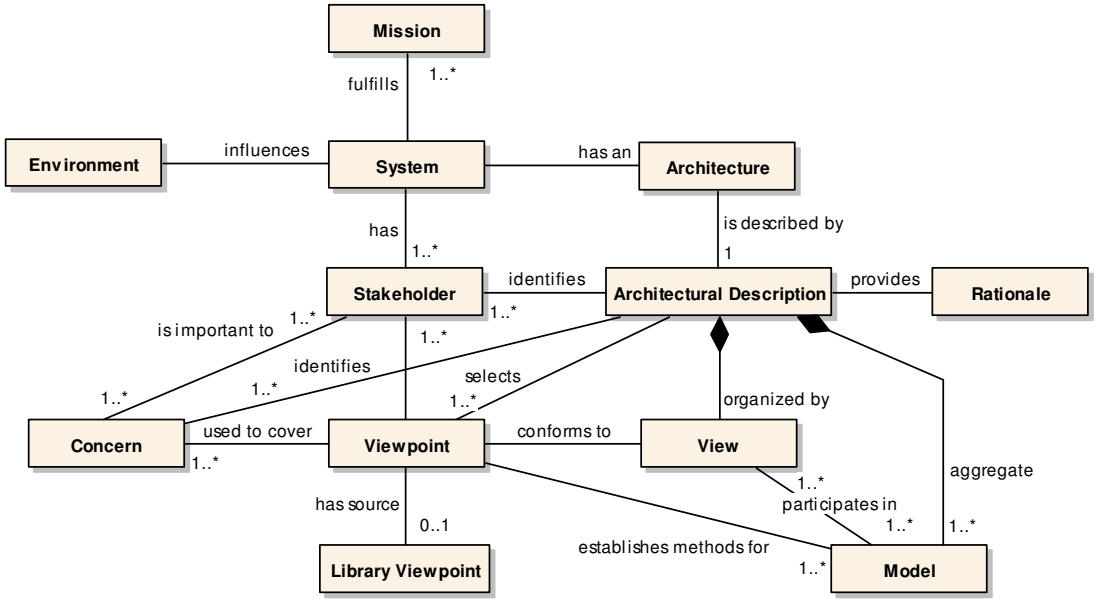


Figura 3.2 – Infraestrutura conceptual IEEE-1471 (adaptado de [Hilliard, 07])

3.3 Relações de dependência entre artefactos

Conforme foi considerado anteriormente, um artefacto pode conter outros artefactos. Importa agora considerar as diversas relações que se podem estabelecer entre esses artefactos tendo em vista acrescentar ao modelo da Figura 3.1 os respectivos conceitos.

Considera-se que dois artefactos são **equivalentes** quando têm a mesma representação dos respectivos elementos do modelo (conceitos). Na Figura 3.3, mesmo considerando que as classes *Person* e *Human* representam o mesmo elemento, pode-se afirmar que os artefactos 1 e 2 não são equivalentes, visto terem o meta-atributo *nome* com valores diferentes ("Person" no primeiro caso e "Human" no segundo).

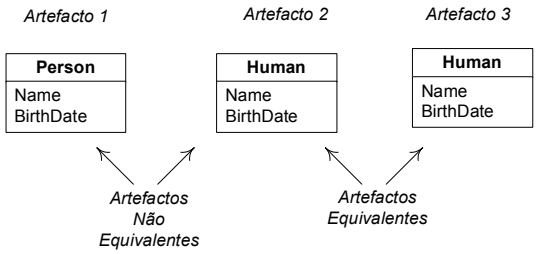


Figura 3.3 – Equivalência entre artefactos

A relação de equivalência é pouco interessante na medida em que é demasiado restritiva caso se queira modelar domínios de alguma complexidade. Neste tipo de domínios existe por vezes a necessidade de se criarem sinónimos para conceitos que são usados sob vistas diferentes. Nesse

caso, o facto de os artefactos não serem equivalentes não significa que os conceitos não estão representados de forma **coerente**. Na Figura 3.4 os artefactos 4 e 5 representam duas classes, em diferentes artefactos, que apesar de parecerem representar o mesmo conceito não têm informação associada que permita justificar essa afirmação. Não existe nenhuma **relação de coerência** sobre os dois artefactos que os permita relacionar quanto ao elemento que representam. Assim sendo não podem ser consideradas coerentes. Na verdade nem sequer estão relacionadas quanto à coerência o que é diferente de não serem coerentes[14].

A relação de identidade é definida, por comparação, através do conceito de variável numa linguagem de programação como Java ou C#. Quando duas variáveis têm como valor o mesmo artefacto, estas são idênticas, i.e., representam o mesmo artefacto.

Os artefactos 6 e 7 possuem informação associada[15] que permite afirmar a coerência entre eles. Pelo contrário, os artefactos 8 e 9, sendo diferentes perspectivas da mesma classe, ao terem uma definição dos atributos diferente não são coerentes.

Para que a relação de coerência seja melhor definida, serão considerados informalmente alguns conceitos.

Seja **A** o conjunto dos artefactos (a_1, a_2, ..., a_n) presentes num dado domínio Ω, definido como o conjunto dos projectos tratados. Considera-se a ausência de artefactos designada por $\mathbf{a}_{\varnothing}$, ou *artefacto nulo*, sendo $\mathbf{a}_{\varnothing} \in \mathbf{A}$. (1)

Considera-se uma **expressão** $E(a^*)$ **sobre um conjunto de artefactos a*** qualquer expressão possível no contexto da definição desses artefactos, sendo o seu valor calculável. Uma expressão $E(a^*)$ compreende usualmente os valores dos tipos básicos (e.g., *int, string* ou *bool)*, embora possa conter igualmente qualquer objecto cuja definição exista nesse âmbito, bem como operações aritméticas, lógicas ou outras que estejam definidas (e.g, GetID(c), representaria uma chamada a uma função "GetID" que devolve a identificação do artefacto "c"). Uma expressão $E(a^*)$ pode ser igualmente um valor constante (numérico ou outro qualquer). Considera-se por V o conjunto de todos os valores possíveis para as expressões produzidas sobre os artefactos de um contexto. Assim sendo uma expressão é uma aplicação de artefactos em valores:

$$E: A \rightarrow V \quad (2)$$

Considere-se a relação binária de comparação, **identidade** entre valores designada por $\leftrightarrow$ e que resulta num valor booleano *true* caso ambas as expressões se refiram ao mesmo valor. Se, neste contexto, os elementos relacionados são objectos o valor refere-se à identidade dos objectos, ou seja, se eles são o mesmo objecto. A semântica desta relação é semelhante à da comparação de

[14] Considera-se que dois elementos estão num estado não coerente, em oposição ao estado de coerência, quando negam uma regra implícita ou explícita de coerência definida entre ambos.

[15] De forma a simplificar a explicação a informação associada não está completa porque não estão referidos os contextos em que os conceitos são usados.

identidade numa linguagem de programação orientada por objectos (i.e., se uma variável é comparada com outra, o resultado é o valor lógico *true* se ambas as variáveis referenciam o mesmo objecto ou possuem o mesmo valor pertencente a um tipo básico).

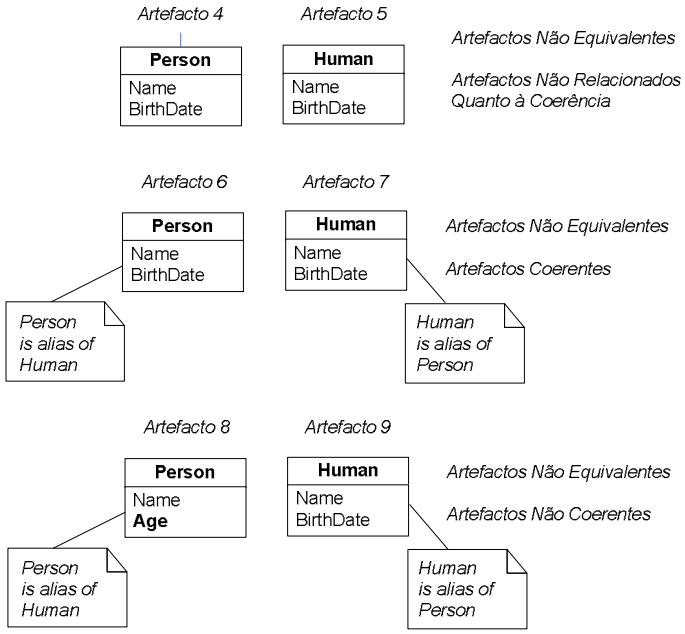


Figura 3.4 – Coerência entre artefactos

Apesar da relação binária identidade relacionar expressões pode dizer-se que estas são sempre convertidas em valores, após a sua avaliação, pelo que é possível dizer-se que $\leftrightarrow$ verifica as seguintes propriedades:

a) $\leftrightarrow$ é reflexiva, ou seja, $\forall a \in V: a \leftrightarrow a$

b) $\leftrightarrow$ é simétrica, ou seja, $\forall a, b \in V: a \leftrightarrow b \Rightarrow b \leftrightarrow a$ (3)

c) $\leftrightarrow$ é transitiva, ou seja, $\forall a, b, c \in V: a \leftrightarrow b \text{ e } b \leftrightarrow c \Rightarrow a \leftrightarrow c$

Como simplificação, considere-se agora apenas as expressões sobre um artefacto $E(a)$.

Para cada artefacto a_x existe sempre uma ou mais expressões $E_{\varnothing}$ definidas da seguinte forma

$$E_{\varnothing}(a_x) \leftrightarrow a_x \; ; \; a_x \in A \quad (4)$$

Facilmente se verifica que o conjunto de casos possíveis para $E_{\varnothing}(a_x)$ é infinito visto ser sempre possível encontrar uma expressão mais complexa do que a anterior, com o mesmo resultado.

Uma relação de coerência entre os artefactos a_x e a_y é definida por uma expressão:

$$E_m(a_x) \leftrightarrow E_n(a_y) \; ; \; a_x, a_y \in A \quad (5)$$

Generalizando (5) para mais de dois artefactos, é possível definir uma **relação de coerência entre dois conjuntos de artefactos a^*_x e a^*_y** como sendo:

$$E_m(a^*_x) \leftrightarrow E_n(a^*_y)\ ;\ a^*_x,\ a^*_y \subseteq A \tag{6}$$

Embora seja possível considerar relações de coerência entre mais de dois conjuntos de artefactos, estas podem ser descritas por um conjunto de relações simples entre dois conjuntos de artefactos. Por sua vez, de forma a simplificar a definição dos conceitos subsequentes serão consideradas normalmente relações de coerência que envolvem apenas dois artefactos. Quando dois artefactos a_x e a_y possuem uma relação de coerência, esta pode ser usada para definir um **rasto simples**:

$$\begin{aligned} &\forall r_i \in R\ \exists a_x, a_y \in A: r(a_x,a_y) \Leftrightarrow E_m(a_x) \leftrightarrow E_n(a_y) \\ &\text{sendo } R: V \times V \rightarrow \{true, false\} \\ &\text{e } a_x \neq a_y \end{aligned} \tag{7}$$

e pode ler-se como: o rasto simples r entre os artefactos a_x e a_y está definido pela relação de coerência $E_m(a_x) \leftrightarrow E_n(a_y)$ e o seu valor para esses artefactos, num dado momento, é true se a avaliação das duas expressões $E_m(a_x)$ e $E_n(a_y)$ resulta em valores idênticos, ou false caso contrário.

É necessário distinguir a expressão do seu valor, depois de esta ser avaliada. Para tal considera-se como **definição de** E_m a sequência dos símbolos de um vocabulário, organizados segundo uma certa ordem e sujeitos a uma sintaxe, que define a expressão. A formalização da sintaxe de uma linguagem de expressões sai do âmbito deste trabalho, tendo sido estudada em numerosos trabalhos [Aho, 06]. Em consequência, o valor da avaliação de E_m é diferente da sua definição sendo esta identificada por $D(E_m)$.

Os artefactos que fazem parte da relação de coerência são normalmente diferentes, embora seja possível um **rasto reflexivo** envolvendo diferentes expressões para um mesmo artefacto. Assim, sendo $r_i \in R$, diz-se que este é um **rasto reflexivo** sobre o artefacto a_x se e só se:

$$\begin{aligned} &\exists r_i \in R,\ a_x \in A: r(a_x, a_x) \Leftrightarrow E_m(a_x) \leftrightarrow E_n(a_x) \wedge D(E_m(a_x)) \neq D(E_n(a_x)) \\ &\text{sendo } R: V \times V \rightarrow \{true, false\} \end{aligned} \tag{8}$$

Generaliza-se a noção de *valor do rasto,* para quaisquer artefactos, através de uma aplicação algébrica denominada val tal que:

$$\begin{aligned} &val(r, a_x, a_y) \\ &\text{sendo } val: R \times V \times V \rightarrow \{true, false, nil\} \end{aligned} \tag{9}$$

À aplicação anterior $val(r, a_x, a_y)$ designa-se como **valor do rasto r, para dados dois artefactos a_x, a_y**, num dado momento, resultando da avaliação da relação de coerência respectiva, tomando para o cálculo esses dois[16] artefactos a_x e a_y. O rasto pode ser ou não válido, tal como pode não estar

[16] Na verdade, usa-se aqui uma simplificação óbvia pelo facto de o rasto poder ser definido entre diferentes conjuntos de artefactos (a^*_x e a^*_y seguindo a definição dada de relação de coerência). No entanto, o lado

definido, para os artefactos em questão. Por isso o seu valor, para dois artefactos em concreto, pertence ao conjunto dos valores *{true, false, nil}.*

A definição (9) é na verdade um caso particular visto representar uma avaliação sobre dois artefactos explicitamente identificados. A noção de valor tem de ser ampliada para conter relações entre artefactos implicitamente identificados, i.e., pertencentes a um nível conceptual inferior ao explicitado. E.g, uma avaliação de uma expressão sobre as ocorrências do artefacto "Classe UML" é diferente de uma avaliação da mesma expressão sobre o próprio artefacto. O nível conceptual é um valor numérico inteiro dado pela arquitectura MDA. Para tal considera-se uma aplicação:

$elem^{n}(a)$

sendo $elem: A \times Z \rightarrow A^{*}$

$elem$ = {...,

$elem^{-1}(a)$ = {ocorrências do artefacto a a um nível imediatamente inferior}, (10)

$elem^{0}(a) = a$,

$elem^{1}(a)$ = {definição de um artefacto a a um nível imediatamente superior },

...}

Sendo Z o conjunto dos números relativos usualmente considerados, na prática $elem^{0}(a)$ significa o próprio artefacto, $elem^{-1}(a)$ representa todas as ocorrências do artefacto ao nível conceptual imediatamente inferior e assim sucessivamente. É ainda possível entender $elem^{1}(a)$ como sendo o artefacto que define o artefacto a.

A definição de um rasto, para dois artefactos em particular, resulta evidente com as definições já realizadas. A definição de um rasto entre dois grupos de artefactos é igualmente possível se se considerar que um artefacto pode ser uma agregação de outros artefactos. Assim sendo a operação $elem^{n}(a)$ deve ser estendida para conter a enumeração dos artefactos agregados. Complementa-se assim a definição de $elem^{n}(a)$ com a noção de **granularidade**:

$elem^{n}_{m}(a)$

sendo $elem: A \times Z \times N \rightarrow A^{*}$

$elem$ = { $elem^{n}_{0}(a) = elem^{n}(a)$, $elem^{n}_{1}(a)$ = { artefactos agregados por a }, ...} (11)

e diz-se que $elem^{n}_{m}(a)$ representa os artefactos dependentes de **nível** n e g**ranularidade** m

---

esquerdo e direito da relação de coerência têm uma semântica diferente, sendo formados cada um deles por um ou mais artefactos.

O operador $elem^n_m(a)$ enquanto enumeração é útil para definir uma forma mais abrangente de valor para o rasto entre dois artefactos. Considera-se por val^n_m um **valor implícito de nível n e granularidade m de um rasto** r **sobre dois artefactos** a_x **e** a_y um operador que relaciona dois artefactos:

$val^n_m: R \times Z \times N \times V \times V \rightarrow$ *{true, false, nil}*

sendo $a_x \in A' \subseteq A$, $a_y \in A'' \subseteq A$:

$$val^n_m(r, a_x, a_y) = \begin{cases} true : \forall a_i \in elem^n_m(a_x)\ \exists' a_j \in elem^n_m(a_y): val(r, a_i, a_j) = true; \\ false : \exists a_i \in elem^n_m(a_x): \forall a_j \in elem^n_m(a_y): val(r, a_i, a_j) = false; \\ nil : \forall a_i \in elem^n_m(a_x), a_j \in elem^n_m(a_y): val(r, a_i, a_j) = nil; \end{cases} \quad (12)$$

Existe uma dependência evidente entre as definições (9), (10) e (12) que pode levar a uma avaliação recursiva, estando a condição de paragem contida na definição (12).

Como caso particular de (12) tome-se um rasto entre um artefacto simples e um conjunto de artefactos. Para que a definição seja válida, considera-se a existência de um artefacto fictício, agregador do primeiro (e tendo-o como único elemento).

Designa-se por **termo direito** e **termo esquerdo** da relação de coerência às respectivas expressões. De forma a realizar pesquisas num dado conjunto já existente de rastos, e em diferentes conjuntos de artefactos considera-se:

$R^?(a_x, a_y)$

sendo $R^?: A \times A \rightarrow R^*$ (13)

e pode ler-se como: "O conjunto de rastos entre os artefactos a_x e a_y"

Na definição anterior o operador $R^?(a_x, a_y)$, a partir de dois artefactos, devolve um conjunto de rastos R^* que pode ser vazio, caso não existam rastos que relacionem ambos os artefactos, na ordem especificada. Pode considerar-se ainda, como subcaso do anterior:

$R^?(a_x, a_\varnothing)$

sendo $R^?: A \rightarrow R^*$ (14)

que se lê como: "O conjunto de rastos entre o artefacto a_x e outro qualquer"

Na definição anterior, $R^?(a_x, a_\varnothing)$ representa um conjunto de rastos que pode ser vazio, caso não existam rastos que relacionem o artefacto a_x com outros artefactos, sendo a_x o artefacto que faz parte do termo esquerdo da relação de coerência. De forma análoga representa-se:

$R^?(a_\varnothing, a_y)$

sendo $R^?: A \rightarrow R^*$ (15)

E pode ler-se como: "O conjunto de rastos entre qualquer artefacto e o artefacto a_y"

A expressão anterior representa um conjunto de rastos que pode ser vazio, caso não existam rastos que relacionem qualquer artefacto com o artefacto a_y, sendo a_y o termo direito da relação de coerência.

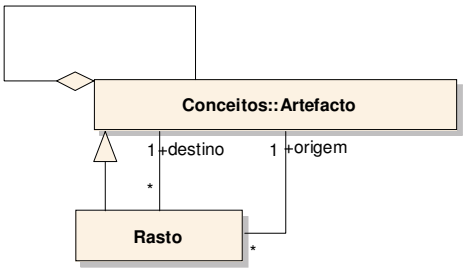


Figura 3.5 – Metamodelo do rasto

Um rasto r_z pode ser considerado o **rasto transitivo** de dois rastos r_x e r_y se as expressões que os definem verificam a seguinte relação:

$$\text{Se } r_x(a_i,a_m) \wedge r_y(a_m,a_j): r_x(a_i,a_m) \Leftrightarrow E_r(a_i) \leftrightarrow E_s(a_m) \wedge$$

$$r_y(a_m,a_j) \Leftrightarrow E_s(a_m) \leftrightarrow E_t(a_j) \qquad (16)$$

$$\text{então } \exists^1 r_z(a_i,a_j): r_z(a_i,a_j) \Leftrightarrow E_r(a_i) \leftrightarrow E_t(a_j)$$

Pode ainda considerar-se que dois artefactos a_i e a_j são **coerentes quanto ao rasto** r_x quando, num dado momento, a relação de coerência que o define é válida.

Verifica-se que $r(a_i, a_j)$ pode ser eventualmente diferente de $r(a_j, a_i)$. Como exemplo, considere-se o rasto:

$$r(a_i,a_j) \Leftrightarrow \text{NrClasses}(a_i) \leftrightarrow \text{NrTabelas}(a_j)\,;\, a_i \in A' \subseteq A,\, a_j \in A'' \subseteq A \qquad (17)$$

sendo: a_i uma colecção de classes Java, a_j uma base de dados, e as operações que calculam o número de classes e o número de tabelas presentes no artefacto respectivo.

No exemplo anterior é evidente que o rasto $r(a_j,a_i)$ com a mesma definição não seria idêntico ao anterior, caso pudesse sequer ser definido. Logo pode dizer-se que, apesar da definição das relações de coerência ser simétrica, os rastos nem sempre o são.

$$\exists r \in R: \text{val}(r,a_i,a_j) \neq \text{val}(r,a_j,a_i) \qquad (18)$$

Dois artefactos podem ter mais do que um rasto definido entre eles, sendo definidas diferentes relações de coerência.

Pode considerar-se que dois artefactos são **genericamente coerentes de** a_i para a_j**,** num dado momento**,** quando todos os rastos existentes de a_i para a_j são válidos:

$$\forall r_k \in R^{?}(a_{i,}\, a_j) : val(r_k(a_i,a_j)) = true \quad (19)$$

Por extensão, pode considerar-se que dois artefactos são **genericamente coerentes entre si,** num dado momento**,** quando todos os rastos existentes envolvendo simultaneamente os dois artefactos, são válidos:

$$\forall r_k \in R^{?}(a_{i,}\, a_j): (val(r_k(a_i,a_j)) = true) \wedge \quad (20)$$
$$\forall r_p \in R^{?}(a_{j,}\, a_i): (val(r_p(a_j,a_i)) = true)$$

Um rasto $r(a_x,a_y)$ pode ocorrer entre artefactos que realizam conceitos de um metamodelo (e.g., *classe C#* e *tabela relacional*). Quando tal acontece diz-se que $r(a_x,a_y)$ é um **meta-rasto** entre a_x e a_y. Nesse caso $val^{-1}{}_0(r(a_x,a_y))$ é dado pela verificação simultânea do rasto sobre todas as realizações de ambos os artefactos que cumprem a relação de coerência.

Considera-se ainda uma aplicação sobre um dado rasto $r(a_x,a_y)$ e denominada **função peso de um rasto** r **de grau n e granularidade m:**

$$W: R \times Z \times N \rightarrow Z$$

Sendo indicada por $w^n{}_m(r(a_{i,}\, a_j))$ onde n é o **grau da função peso** e definida como o número de artefactos envolvidos no rasto r_k no nível conceptual MDA correspondente ao da definição, somado com o grau da função peso. (21)

Tome-se, e.g., um meta-rasto entre cada *classe UML* e *tabela de uma base de dados relacional* que faz com que para cada classe tenha de existir uma (e apenas uma) tabela com o mesmo nome:

$$r \in R,\ a_x,\ a_y \in A,\ x, y, i, j \in N: r(a_x,a_y) \Leftrightarrow \quad (22)$$
$$(\forall a_i \in classesUML\ \exists^1\ a_j \in tabelasRelacionais : Nome(a_i) \leftrightarrow Nome(a_j))$$

No exemplo anterior verifica-se que $val^0{}_0(r) = false$ visto o nome do meta-artefacto ser diferente. No mesmo exemplo o valor interessante seria $val^{-1}{}_0(r)$ que significa o valor do rasto entre os artefactos dependentes (i.e, classes e tabelas relacionais, no mesmo exemplo).

É ainda possível afirmar que:

Para um certo rasto r_k é sempre possível saber qual o valor da sua *função peso* de grau zero ($w^0(r_k)$) sendo o valor da contagem do número de artefactos que fazem parte da definição do rasto (usualmente dois). (23)

Quanto ao grau, para valores diferentes de zero, o peso dos rastos pode ter valores iguais ou superiores a zero, consoante o nível conceptual dos artefactos (e.g., no exemplo dado em (7) é possível calcular o valor de $w^{-1}(r_k)$, sendo o seu valor a soma do número de classes com o número de tabelas). A **função peso** é útil para entender o impacto que um rasto pode ter num sistema, quantificando o número de artefactos directa ou indirectamente afectados por este. Define-se assim uma medida denominada **estimativa do impacto de alterações com alcance n e granularidade m**, para um artefacto a_x como sendo:

$$c^{n}_{m}(a_x)=\frac{\sum_{i=0}^{n} w^{i}_{m}(R^{?}(a_x))}{2}+1 \qquad (24)$$

A **sincronização**, como chamar-se-á à actividade de manter a coerência entre os diversos artefactos, deverá ser realizada de forma a que as alterações realizadas sobre um elemento sejam propagadas a todos os artefactos relacionados quanto à coerência. Como foi mostrado, as acções a tomar após a alteração do estado do sistema, não têm como fim manter os elementos equivalentes, porque tal não é importante em muitos casos, mas sim garantir a sua coerência (tanto quanto possível na sua forma genérica, como definido em (20)). Para ser utilizável na prática, esta actividade deve ser feita recorrendo a meios automáticos que permitam uma tomada de decisão mais rápida, bem como uma actualização dos artefactos correspondentes.

3.4 Classificação das Operações de Sincronização

Através da abordagem MDA (Secção 2.2.2) caracterizam-se os artefactos produzidos em quatro níveis (Figura 3.6). Quanto ao nível dos artefactos envolvidos pode-se assim dividir a sincronização nas seguintes formas:

Sincronização Interna – O conjunto de artefactos que descreve o modelo é coerente entre si, i.e., não existe uma parte de um deles que contradiga qualquer uma das partes dos restantes. Diz-se que nestas condições o modelo, e em consequência os seus artefactos, estão *internamente sincronizados*.

Sincronização Horizontal – Os artefactos dentro de cada nível são coerentes entre si, i.e., não existe nenhuma relação de coerência entre artefactos do mesmo nível que tenha o valor falso.

Sincronização Vertical de Diagramas – Dois artefactos A e B de níveis diferentes estão sincronizados, i.e., não existe nenhum artefacto X pertencente a A que negue uma relação de coerência com outro artefacto Y pertencente ao artefacto B. Diz-se que nestas condições dois diagramas estão *verticalmente sincronizados.*

Sincronização Vertical de Níveis – Dois níveis estão sincronizados, i.e., qualquer diagrama de um dos níveis é coerente com qualquer outro diagrama pertencente a outro nível. Analogamente dois níveis podem desta forma estar *verticalmente sincronizados*.

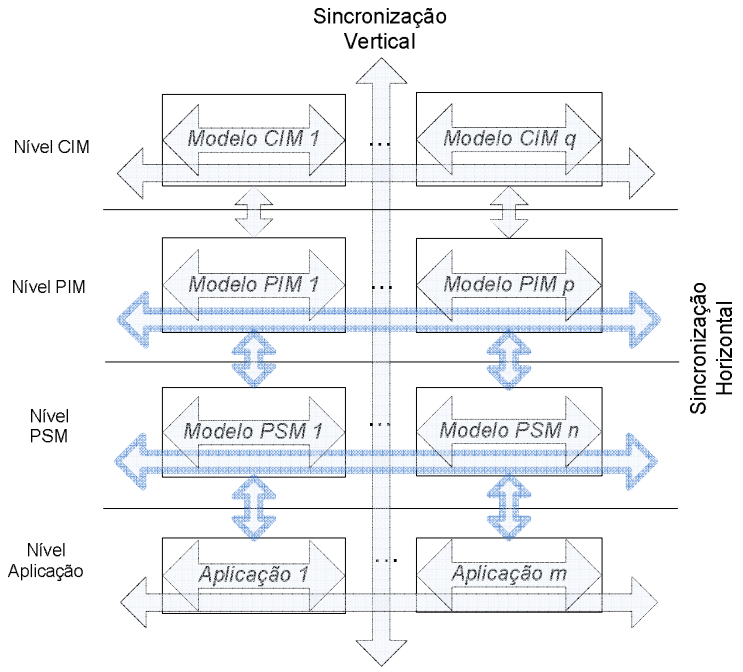


Figura 3.6 – Dimensões da sincronização de artefactos decorrentes da abordagem MDA

Para fins de implementação, os artefactos são divididos em dois tipos: aqueles que dizem respeito aos modelos e os que dizem respeito ao código. A sincronização pode assim ser caracterizada quanto ao tipo de artefactos:

- Intermodelos: Um modelo pode ter elementos dependentes de outros elementos de outros modelos. Não se considera aqui a representação duplicada do mesmo conceito em diagramas distintos que é tratada trivialmente pelas ferramentas CASE. Estas dependências podem ser apenas semânticas, sem que exista qualquer relação sintáctica entre os dois conceitos. Nestes casos a ferramenta necessita de intervenção humana para conseguir determinar este tipo de dependências.

- Intramodelos: Dentro de um modelo podem existir relações de dependência entre diferentes elementos do modelo. Por exemplo, no diagrama de sequência, uma mensagem para um objecto deve corresponder à definição de uma operação na classe correspondente do diagrama de classes. Esta dependência pode ser verificada automaticamente sem grande dificuldade mas podem existir também outras dependências definidas manualmente (e.g. um parâmetro de entrada de um método de uma classe pode estar dependente do nome de uma classe do mesmo modelo).

- Código-Modelo: Um elemento do modelo pode estar dependente de um elemento do código e vice versa. Neste tipo de dependência não basta tratar a representação do modelo, é necessário também que exista uma representação em memória do próprio código. Isso pode ser conseguido recorrendo-se a um *parser* que a partir de uma gramática da linguagem e do ficheiro fonte produz uma representação em objectos dos elementos sinctáticos existentes no código. Estes elementos do código podem então ser mapeados em elementos do modelo recorrendo-se a um conjunto de regras de coerência já definidas. Estas regras podem ser instanciações de outras

mais genéricas ou podem ser definidas caso a caso. Este tipo de dependência poderia ser considerado um caso particular das dependências intermodelos, se não se distinguisse o código dos modelos e,

- Código-Código: Este tipo de relações podem ser transformados em duas relações do tipo anterior facilitando-se a sua implementação.

A sincronização, envolvendo geração de código e transformações entre modelos, consoante o(s) nível(is) MDA em que se situa, apresenta diferentes aspectos que podem levantar problemas de implementação de maior ou menor grau.

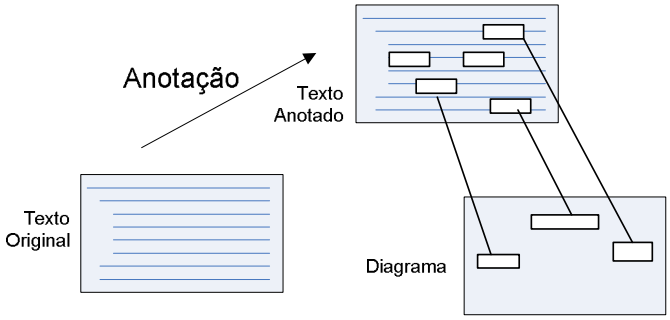


Figura 3.7 – Anotação de textos e utilização posterior em operações de sincronização

Os artefactos correspondentes ao nível CIM são normalmente pouco formais englobando textos em língua natural. Quando os artefactos incluem um metamodelo, a sincronização faz-se normalmente, como se de um artefacto PIM ou PSM se tratasse. O problema surge nos artefactos que incluem uma parte significativa escrita em língua natural (como é usual no caso da descrição de requisitos). O facto destes textos, por si mesmos, não terem uma estrutura (ou metamodelo) inviabiliza a sincronização das suas partes com outros artefactos. Porém, é possível anotar os textos com marcas que podem servir para identificar ocorrências de uma estrutura já definida. Pode usar-se para tal uma linguagem como o XML, mas os textos originais passam a não ter qualquer valor. Os novos artefactos assim criados têm de substituir os anteriores. O metamodelo dos artefactos entretanto criados pode ser suficientemente rico que permita o estabelecimento de relações de coerência entre estes e os restantes, quer pertençam ao mesmo nível MDA ou a outro.

O problema inverso surge ao nível do código, devido ao facto das linguagens de programação terem normalmente uma formalização correcta mas envolvendo um conjunto muito grande de conceitos. A abordagem usual é a de limitar o tratamento dos artefactos aos conceitos estritamente relevantes. Por exemplo, numa aplicação C# podem ser tidos em conta conceitos como *Classe, Variável de instância, Método.* Todos estes conceitos têm uma relação evidente com os elementos correspondentes dos diagramas de classes UML. Pelo contrário, conceitos como *atribuição* ou *decisão* podem ser ignorados, pelo menos numa primeira fase. As consequências negativas deste tipo de simplificação dizem respeito à eventual produção de código não compilável. Tome-se como exemplo o seguinte segmento de programa em C#:

```
public class Pessoa {
   public string nome = “”;
   public int idade = 0;
```

```
}
public class ColeccaoPessoas {
   public List<Pessoa> lista = new List<Pessoa>();
   public void CriarPessoa()
   {
        Pessoa p = new Pessoa();
        p.nome = Console.ReadLine();
        lista.Add(p);
        Console.WriteLine("Acrescentou {0} à lista: ", p.nome);
   }
}
```

Considere-se ainda o diagrama da Figura 3.8, no qual estão representados alguns dos conceitos anteriormente indicados.

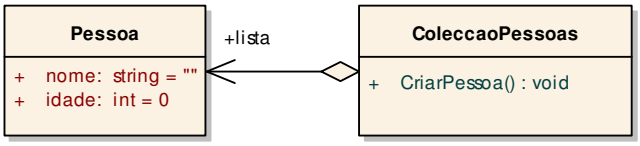


Figura 3.8 – Diagrama de classes UML referente a um segmento de progama em C# (1)

Suponha-se que o diagrama de classes passa a ser o da Figura 3.9, através da mudança do identificador da classe *Pessoa* para *Cliente.* Após a substituição do novo identificador no código, usando uma qualquer técnica que preserve o corpo dos métodos, este pode deixar de ser compilável. Basta para tal que não sejam actualizadas as referências ao nome da classe, no interior do corpo dos métodos, para que o código mantenha referências a uma classe entretanto inexistente.

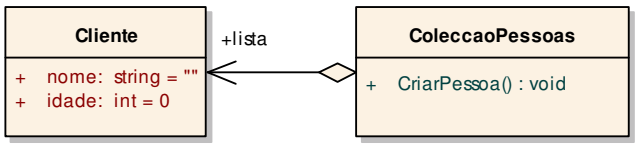


Figura 3.9 - Diagrama de classes UML referente a um segmento de progama em C# (2)

Apesar da simplicidade do exemplo abordado, torna-se evidente que uma simplificação no metamodelo da linguagem de programação é razão suficiente para que a qualidade dos artefactos produzidos possa diminuir. No mesmo exemplo, uma ferramenta IDE poderia facilmente indicar os erros criados na medida em que as alterações violam as regras de compilação da linguagem. Nesse caso, uma simples substituição monitorizada da palavra antiga pela nova poderia resolver o problema em pouco tempo, mesmo com centenas de ocorrências. Porém, as consequências podem ser mais difíceis de gerir se se consider um contexto em que a aplicação possui diferentes classes com o mesmo nome em contextos privados. Neste caso o processo de substituição com um IDE poderia criar ainda mais erros do que aqueles que já existiam.

É por isso desejável que o metamodelo das linguagens de programação envolvidas seja tão completo quanto possível. No entanto, nem sempre as linguagens de programação[17] utilizadas são orientadas por objectos, ou mesmo imperativas [Dershem et al, 95]. As aplicações podem também conter segmentos de código escritos em linguagens declarativas (e.g., SQL), funcionais (e.g., Lisp), ou lógicas (e.g., Prolog). As relações de coerência encontradas entre os diagramas e o código podem ser mais ou menos abrangentes, consoante as linguagens de programação têm elas próprias conceitos análogos aos metamodelos dos modelos usados.

Conforme foi indicado, naquilo que diz respeito à sincronização, os artefactos CIM podem ser tratados como os artefactos PIM ou PSM, desde que possuam um metamodelo suficientemente bem definido. No entanto o caso de estudo realizado focou-se nos tipos de sincronização que estão diferenciados na Figura 3.6, nomeadamente: PIM-PIM, PIM-PSM e PSM-Code.

3.5 Rastreabilidade

O significado usual do termo **rastreabilidade** é demasiado vago, quando aplicado ao contexto dos sistemas de informação. Definições como "*a possibilidade de identificar a origem de um produto e de reconstituir o seu percurso desde a produção até à distribuição*" [Porto, 08] ou "*a capacidade de relacionar cronologicamente entidades univocamente identificáveis e de uma forma relevante*" [Wikipedia, 08] têm de ser contextualizadas para que o seu significado possa ser compreendido.

Considerando os conceitos anteriormente definidos (em 3.2 e 3.3) é possível definir **rastreabilidade** como a acção de verificar[18] os rastos existentes para um grupo de artefactos, tendo em vista um dos seguintes objectivos:

- Análise de sensibilidade – Quando se realiza uma modificação ou eliminação (e em casos especiais uma criação) de um artefacto a rastreabilidade pode ser usada para se entender, quais as implicações que essa acção terá nos outros artefactos, i.e., quais os artefactos afectados pela mesma;
- Validação de requisitos – A validação de um requisito pode ser feita seguindo os rastos necessários, do artefacto que contém o requisito até aos testes e respectivos resultados (eles próprios igualmente artefactos);
- Verificação de conformidade – Os artefactos de código podem estar ou não de acordo com os modelos respectivos, sendo essa verificação feita a partir da validação dos rastos entre artefactos de código e artefactos dos modelos;

[17] Por vezes o código gerado é um ficheiro de configuração, ou qualquer outro artefacto submetido a um *formato* que não representa uma parte de um programa. Nesse caso o dito *formato* tem de ser completo, i.e., todos os constituintes do texto têm de poder ser expressos inequivocamente através dessa definição.

[18] Usando a mesma analogia, *verificar um rasto* é semelhante a *seguir um rasto.*

- Pesquisa documental – A rastreabilidade pode ser usada para o conhecimento técnico da estrutura de uma aplicação, seguindo-se os rastos entre um dado artefacto e os demais.

Assim sendo, tomada como uma actividade, a rastreabilidade não tem como fim realizar alterações a uma aplicação, mas sim obter um determinado tipo de informação sobre a estrutura dessa aplicação. É também possível dizer-se que *uma aplicação é rastreável*[19] ou *um grupo de artefactos é rastreável,* usando-se a rastreabilidade como uma propriedade. Na verdade, quando se usa a rastreabilidade como um qualificador, está-se a afirmar que sobre essa *aplicação* ou *grupo de artefactos* é possível realizar uma operação de rastreabilidade.

Estabelecendo uma analogia com a linguagem SQL, a rastreabilidade seria comparável a uma operação *select,* enquanto a sincronização seria comparável às operações *insert, update* e *delete.* Na primeira operação o estado do sistema não é alterado, enquanto nas restantes podem ser produzidas alterações.

A rastreabilidade pressupõe uma sincronização prévia visto que para que uma aplicação seja rastreável é necessário que os seus artefactos estejam num estado coerente, i.e., que as relações de coerência definidas sejam válidas nesse momento. Caso contrário os resultados obtidos não serão exactos. A qualidade da rastreabilidade está por isso dependente da qualidade da sincronização. Por sua vez, a sincronização pressupõe que as relações de coerência entre os artefactos estejam criadas correctamente. A qualidade da sincronização depende da exactidão da definição das relações de coerência entre os artefactos.

3.6 Perspectivas Relacionais e Generativas

É possível analisar os rastos entre artefactos segundo duas perspectivas que têm implicações muito diferentes. Segundo a **perspectiva relacional**, um dado conjunto de artefactos existe num determinado momento. As relações de coerência que vão sendo encontradas ao longo do tempo servem para definir os rastos entre eles. Na **perspectiva generativa** os rastos definidos servem para criar novos artefactos, a partir de meta-rastos já definidos. Nesta perspectiva, os artefactos gerados existem para que o valor do rasto que lhes dá origem seja verdadeiro.

A perspectiva generativa usa operações de sincronização para criar, modificar ou destruir artefactos. Numa perspectiva generativa é comum considerar-se que determinados elementos são gerados a partir de outros, sendo substituídos cada vez que se processa a geração. Algumas instanciações desta abordagem (e.g., [Vanhooff et al, 05]) usam anotações do modelo (marcas de valor e estereótipos UML) para manterem a informação sobre a transformação. As referidas anotações servem de "mnemónicas" para chamar rotinas de transformação predefinidas noutra linguagem.

---

[19] Seguindo a definição dada, como a rastreabilidade diz respeito directamente aos artefactos da aplicação deveria dizer-se *os artefactos de uma aplicação são rastreáveis*. Porém, usar-se-á a forma indicada, enquanto simplificação.

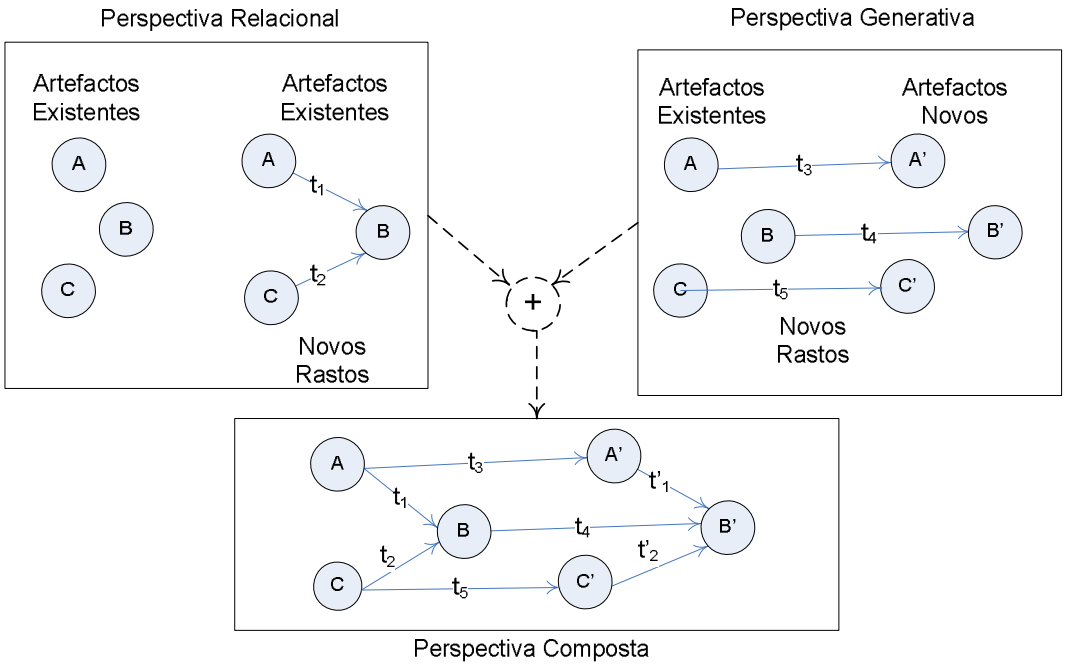


Figura 3.10 – Perspectivas sobre a criação de rastos e artefactos

A utilidade da geração automática de artefactos a partir de rastos é evidente. No entanto, esta perspectiva levanta problemas práticos se se considerar aplicações que já existem, com artefactos bem definidos e estáveis, inclusivamente relacionados com tecnologias legadas. Neste contexto pode ser necessário definir relações de coerência particulares a determinados artefactos. Tome-se o seguinte caso:

Entre outras fontes de dados, uma dada aplicação recorre a uma tabela X que é também utilizada por outras aplicações. É necessário que o nome desta tabela não mude visto não ser já possível manipular a estrutura da base de dados.

Por si só, a perspectiva generativa não resolve o problema referido. É necessário que a definição da *tabela X* não seja novamente gerada. É por isso necessário criar um rasto entre um elemento conceptual e a tabela referida (e eventualmente o rasto inverso). Esse rasto é criado, numa perspectiva relacional, na medida em que passa a relacionar artefactos já existentes.

No entanto, as duas perspectivas são necessárias, visto ambas terem efeitos desejáveis neste contexto. Propõe-se por isso uma *perpectiva composta* que agrega as duas anteriores, ou seja, através da qual é possível (a) criar novos artefactos a partir de rastos existentes, (b) criar novos rastos a partir de artefactos existentes através da descoberta de novas relações de coerência, (c) propagar relações de coerência existentes para artefactos entretanto criados.

3.7 Componente Reactiva

Como foi referido, tanto a rastreabilidade como a sincronização são operações que podem ser realizadas num determinado momento. Existem três formas de agendar a execução destas operações: execução num intervalo de tempo, execução em momentos pré-determinados e execução através de eventos.

Na execução num intervalo de tempo esta pode ser repetida a cada intervalo de tempo constante. O utilizador pode configurar o período da execução para que a execução da aplicação de sincronização não implique um peso significativo no desempenho total do sistema. Este período dependerá da quantidade de artefactos para os quais existem relações de coerência definidas e do desempenho da aplicação.

Com a execução em momentos pré-determinados é possível ao utilizador desencadeá-la manualmente, nesse momento, ou num conjunto de momentos pré-determinados explicitamente. Este mecanismo resume-se a um mecanismo de agendamento de execuções.

Na execução através de eventos, a execução é realizada quando se dá uma alteração ao estado do sistema. Essa alteração, denominada de **evento**, é uma qualquer alteração ao estado de um artefacto abrangido por, pelo menos, uma regra de coerência. Nas linguagens de programação actualmente usadas (e.g., C#, Java, VB.NET) os eventos ocorrem quando o valor de uma variável é alterado, desencadeando uma operação. Define-se **rastreabilidade reactiva** como a actividade de rastrear e sincronizar artefactos através de um mecanismo de execução por eventos. As operações de sincronização são usadas a par das operações de rastreabilidade, pelo que se optou por limitar a designação desta actividade conjunta à mais conhecida e referida. Assim sendo, a rastreabilidade reactiva é um conjunto de operações que visam reagir a alterações provocadas sobre a estrutura da aplicação em causa.

Se num dado momento τ_0 ocorre uma acção (criação, modificação e eliminação) sobre um artefacto a_i, então todos os rastos $R<a_i>$ têm de ser verificados para que possam ser propagadas as alterações necessárias. Como simplificação, considera-se que essa verificação e as eventuais alterações são atómicas em τ_0.

Considera-se que uma **acção** α sobre um artefacto t_i dada por $\alpha(t_i)$ pertence ao conjunto A. Considera-se que $A: \mathbf{T} \rightarrow \mathbf{T}$

Quanto ao tipo de cada acção é possível indicar dois casos particulares:

Uma acção de criação implica uma relação $\varnothing \rightarrow \mathbf{T}$ sendo indicada por $\alpha(t_{\varnothing})$.

Uma acção de eliminação implica uma relação $\mathbf{T} \rightarrow \varnothing$ sendo indicada por $\alpha^{\varnothing}(t_i)$.

De a) torna-se evidente que uma acção, genericamente, não é uma aplicação no sentido algébrico do termo. Um **evento** é materializado após existir conhecimento de uma acção sobre um artefacto. Esse conhecimento pode ser *a priori* ou *a posteriori.* Como muitos dos artefactos são produzidos por ferramentas específicas e exteriores à própria aplicação de rastreabilidade, esse conhecimento

variará consoante o nível de controlo sobre os próprios artefactos. Caso exista um mecanismo que permita conhecer a *intenção* de realizar uma acção sobre um artefacto, bem como de interromper essa acção, esse conhecimento pode ser *a priori.* Porém, na generalidade dos casos é possível assumir que o evento acontecerá *a posteriori.*

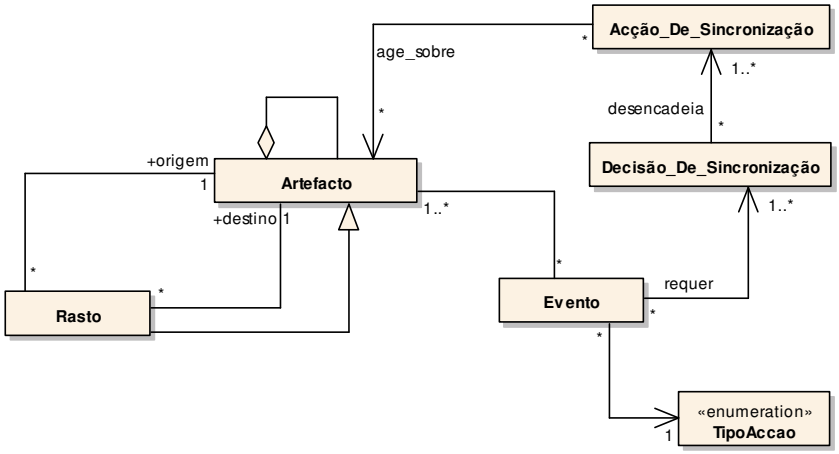


Figura 3.11 – Metamodelo do mecanismo de eventos sobre artefactos

Como um rasto é ele próprio um artefacto é possível considerar rastos sobre rastos, bem como eventos sobre rastos.

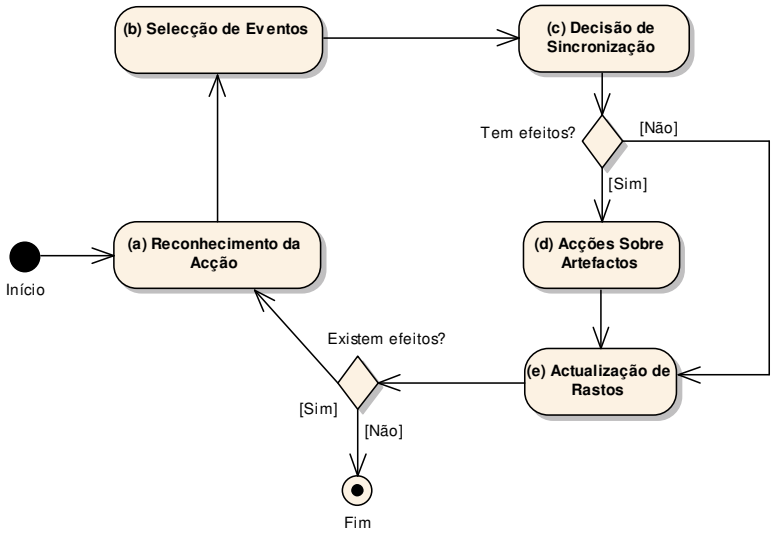


Figura 3.12 – Tratamento dos eventos

O algoritmo de tratamento do evento envolve as seguintes actividades: (a) reconhecimento da acção, (b) selecção de eventos, (c) decisão de sincronização, (d) acções sobre artefactos, (e) actualização de rastos.

Em (a) existe a tomada de conhecimento que foi produzida uma alteração ao estado do sistema, materializada por uma acção sobre um artefacto conhecido. O artefacto é então acrescentado a uma *lista de artefactos alterados (LAA).* Em sequência, para cada artefacto da *lista de artefactos alterados* (b) verifica-se se existe um evento associado a esse artefacto, se não existir a acção é ignorada e o artefacto sai da LAA. Quando existe um evento (e este cumpre as condições de activação, i.e., a acção que o desencadeia é válida) em (c) são invocadas as decisões de sincronização. Neste momento pode ser necessário o utilizador tomar uma decisão, caso a decisão de sincronização implique uma escolha.

Em alguns casos a decisão é determinística e (d) actua imediatamente. Em resultado de eventuais acções sobre o artefacto podem existir efeitos sobre outros artefactos que têm de ser propagadas. Esta actividade pode levar a que o processo seja repetido para os artefactos entretanto alterados. Em (e) é verificada a lista de rastos existentes, bem como os artefactos correspondentes. Quando, entretanto, houve uma acção sobre esse artefacto este é acrescentado à LAA e o processamento continua em (a). A decisão de sincronização (c), não tendo efeitos, pode levar a um estado de incoerência que tem de ser resolvido em (e). Quando a LAA está vazia o sistema voltou ao estado de coerência. Pode então dizer-se que todos os artefactos tratados são genericamente coerentes, conforme a definição dada.

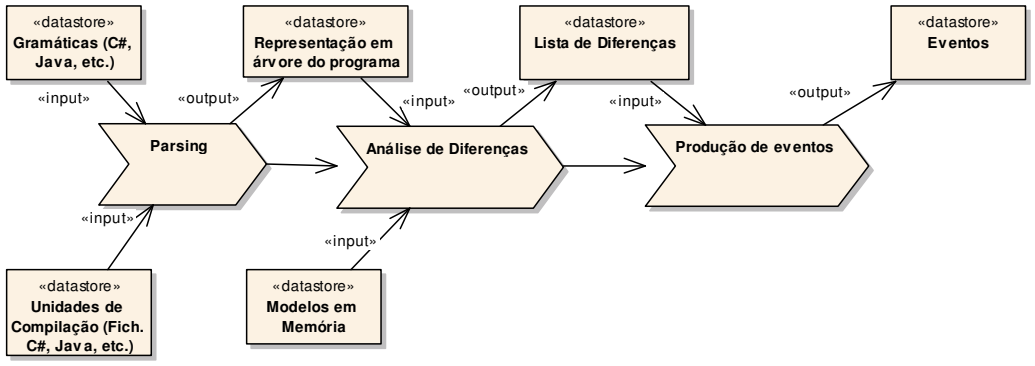


Figura 3.13 – Metamodelo do processo de detecção de eventos

O processo de detecção de eventos (Figura 3.13), ou seja, a tomada de conhecimento da existência de uma alteração ao sistema, produz os dados necessários para o tratamento de eventos anteriormente indicado. O processo é iniciado com a análise sintáctica e lexical *(parsing)* do código existente e que produz uma representação do programa em árvore, através de um modelo de objectos. Este processo é explicado com maior detalhe no Anexo H – Parsers. A análise de diferenças é o processo através do qual é possível saber que existem diferenças entre os artefactos actualmente existentes e aqueles que existiam desde que o sistema foi considerado pela última vez em *estado coerente*. Para cada diferença encontrada é criado um evento que representa a acção sobre o artefacto respectivo (existente ou já inexistente).

4 Framework ReacT-MDD

Neste capítulo são abordados alguns aspectos tecnológicos considerados relevantes. As propostas apresentadas resultam da verificação prática dos conceitos introduzidos no capítulo anterior. A criação da *framework* ReacT-MDD representa uma extensão ao modelo QVT no que diz respeito à implementação da rastreabilidade entre artefactos. É apresentado um protótipo aplicacional denominado ReacT-Workbench que permitiu validar a abordagem.

4.1 Introdução

A *framework* ReacT-MDD envolve diferentes níveis de especificação consoante a descrição está mais próxima ou mais afastada da implementação:

- Especificação conceptual - A especificação conceptual foi realizada no Capítulo 3, através da descrição dos conceitos envolvidos e das suas relações e propriedades;
- Especificação arquitectural – Representa uma descrição dos tipos de elementos tecnológicos existentes e das suas relações, bem como das actividades que de forma genérica são necessárias.
- Especificação aplicacional – Representa uma certa instanciação de uma especificação arquitectural, i.e., uma ferramenta sujeita às restrições da *framework*.
- Especificação de rastreabilidade – Representa a utilização de uma ferramenta que implemente a *framework* ReacT-MDD num certo caso prático.

É fácil comparar os níveis de especificação descritos e os níveis conceptuais MDA. Assim sendo, é possível comparar a especificação de rastreabilidade com o Nível M0 (e.g., as ocorrências do modelo); a especificação aplicacional com o nível M1 (e.g., as definições das instâncias); a especificação arquitectural com o nível M2 (e.g., a estrutura dos conceitos); e a especificação conceptual com o nível M3 (e.g., os conceitos que nos permitem definir todos os conceitos do nível M2).

Na Secção 4.2 é descrita a forma de realizar a especificação arquitectural, através de um conjunto de componentes que têm de existir, segundo a *framework* ReacT-MDD. Seguidamente, na Secção 4.3 são apresentados alguns aspectos de um protótipo realizado em conformidade com esta *framework.*

4.2 Arquitectura

A descrição conceptual dada até este ponto é independente de qualquer tecnologia. Todavia, a tecnologia pode ser um agente facilitador de uma implementação de um protótipo, ou até de um produto final. Os conceitos apresentados foram colocados em prática através de uma *framework*[20] cuja descrição resumida dos aspectos técnicos aqui se apresenta.

Genericamente, uma aplicação de rastreabilidade reactiva tem de possuir alguns componentes, instanciáveis em diversas tecnologias:

- Modelador - Através de uma interface gráfica a ferramenta permite constituir um conjunto de diagramas que representam os conceitos guardados num repositório comum, segundo uma certa notação.

- *Parsers* – A ferramenta possui um conjunto de *parsers* que permite constituir uma representação própria, de um conjunto de artefactos de texto, existentes fisicamente em memória estável. Esta representação é obtida a partir de uma sintaxe definida para a linguagem em questão.

- Motor de metamodelação – É necessário um componente que permita não só definir os metamodelos necessários à compreensão dos artefactos conhecidos como relacionar os artefactos existentes com o tipo correcto de metamodelo. O motor de metamodelação pode ser ainda responsável pela verificação de conformidade dos modelos em relação aos metamodelos. O motor de metamodelação pode ainda ter associada a função de gerar artefactos de ligação entre os metamodelos criados (e.g., XML e código correspondente)

- Fornecedores Externos – Os fornecedores externos permitem efectuar operações sobre artefactos criados por outras aplicações. Estes elementos tornam a *framework* compatível com outras aplicações . A definição dos fornecedores externos deve ser levada a cabo através da implementação de Interfaces genéricas. Os fornecedores podem comunicar com diferentes aplicações, eventualmente executadas em diferentes sistemas operativos.

- Motor de Eventos – O motor de eventos realiza a operação de detecção de eventos e notifica o motor de rastreabilidade quando um evento acontece.

[20] Entende-se por *framework* um padrão ou infraestrutura conceptual reutilizável em diferentes casos, podendo incluir aplicações, bibliotecas de código, linguagens interpretadas, ou outros componentes, tendo em vista relacionar os elementos existentes num dado projecto de desenvolvimento aplicacional.

- Motor de Rastreabilidade – O motor de rastreabilidade verifica o estado de coerência dos artefactos da aplicação, após a ocorrência de um evento e permite tomar decisões (assistidas ou automáticas) quanto às implicações que esse evento tem nos mesmos. O motor de rastreabilidade tem associado um módulo de edição e parsing de uma linguagem usada para definir os rastos entre artefactos.

Os componentes referidos, bem como as relações entre eles, formam a *framework* ReacT-MDD *(Reactive Traceability Model Driven Development).* Quando esta é implementada chama-se à ferramenta resultante **aplicação de rastreabilidade reactiva**.

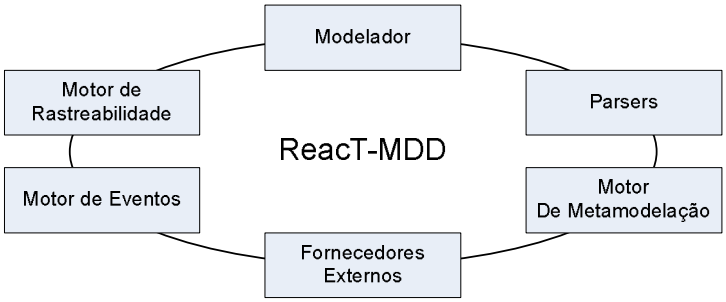


Figura 4.1 – Elementos da *framework* ReacT-MDD

As relações entre os componentes referidos podem variar, consoante a sua implementação, o tipo de artefactos criados pelas tecnologias usadas e as funcionalidades consideradas relevantes. O componente modelador produz artefactos que podem ser tratados por fornecedores externos, caso o componente modelador seja representado por uma ferramenta CASE externa à ferramenta. Pelo contrário, se este componente for implementado através de um módulo próprio, o motor de rastreabilidade e o motor de metamodelação podem interagir mais facilmente com ele, visto poderem partilhar mais facilmente os artefactos produzidos.

Seguidamente será descrita uma implementação desta *framework* que usou um subconjunto destes elementos, de forma simplificada, para testar a abordagem.

4.3 ReacT-Workbench

Foi realizada a instanciação dos elementos descritos, num protótipo que funciona na plataforma *.Net* denominado ReacT-Workbench. Como foi referido anteriormente, o conceito apresentado é independente da tecnologia em que é implementado. No entanto, como a rastreabilidade reactiva diz respeito ao próprio processo de desenvolvimento de aplicações, bem como à sua operacionalização e manutenção, a plataforma usada torna-se relevante para que possa existir conhecimento sobre o estado actual dos artefactos. Se a plataforma em que for implementada a aplicação de rastreabilidade reactiva for diferente da plataforma de algumas das ferramentas que produzem os artefactos da estrutura da aplicação, a interacção pode tornar-se mais difícil de implementar.

A escolha do ambiente de desenvolvimento Microsoft *Visual Studio* (ao longo de diversas versões) deve-se à integração já existente entre diversas ferramentas nele incluídas (e.g., editor, *intellisense, DSL Designer*). Uma dessas ferramentas, o *DSL Designer,* permitiu implementar o componente ReacT-Designer e integrá-lo com a própria operação do ambiente de desenvolvimento. O ReacT-Designer é usado para construir graficamente os metamodelos incluídos nos fornecedores externos implementados. O ReacT-Designer funciona como um *template* de solução dentro do Visual Studio tendo uma forma de interagir semelhante aos restantes.

Foram produzidos diversos *templates* necessários à operação do ReacT-Designer para diferentes metamodelos usados na própria implementação (Anexo I). A implementação dos exemplos estudados foi levada a cabo usando o ReacT-Workbench*,* que representa a ferramenta propriamente dita*.* O próprio ReacT-Workbench pode ser visto como uma *plataforma industrial* [Cusumano, 10] visto fornecer as fundações comuns e a tecnologia de base que uma organização pode usar em diferentes projectos ou produtos (neste caso aplicações). A inclusão dos módulos denominados *fornecedores externos* permite que a ferramenta consiga tratar um maior número de artefactos constituindo essa uma das áreas de desenvolvimento futuro. Os *fornecedores externos* podem ser construídos de uma forma modular em diferentes linguagens de programação sendo independentes entre si.

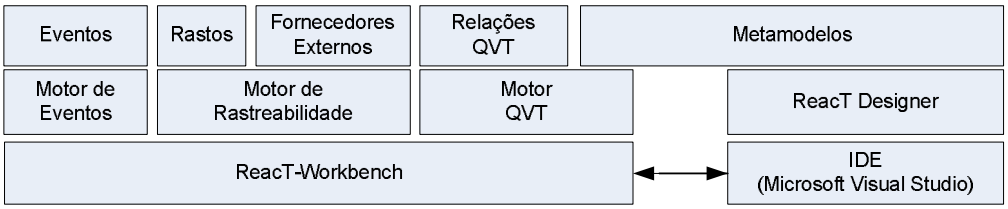


Figura 4.2 – Arquitectura da *framework* ReacT-MDD na plataforma .NET através do ReacT-Workbench

Os fornecedores externos podem incluir módulos de acesso a outras ferramentas como SGBDs, IDEs, processadores de texto, ferramentas de desenho gráfico, ferramentas CASE, entre outras. Podem igualmente servir apenas de ligação aos artefactos produzidos por cada uma destas ferramentas quando uma interacção com as ferramentas que produzem os artefactos é mais dificil. No que diz respeito aos artefactos de código, i.e. produzidos através de uma linguagem de programação, é normalmente mais fácil tratar o artefacto directamente do que interagir com uma ferramenta (neste caso um IDE). Por outro lado, no que diz respeito a uma base de dados relacional é normalmente mais simples interagir com o SGBD do que com os dados propriamente ditos. Na abordagem ReacT-MDD em qualquer um dos casos anteriores existe um meta-modelo que documenta e uniformiza o tratamento dos dados e estruturas de dados envolvidos. O tratamento da conformidade dos artefactos em relação aos metamodelos é um aspecto que sai do âmbito deste trabalho existindo já um estudo aprofundado nesta área [Paige, 07].

O ReacT-Workbench integra um analisador de QVT (correspondente ao componente da *framework* Motor QVT) que permite explicitar as restrições e os rastos existentes entre os diferentes artefactos. O analisador foi programado em C#, bem como a quase totalidade do ReacT-Workbench, e está em conformidade com a sintaxe definida pela OMG [OMG, 05a]. A definição da sintaxe foi realizada segundo uma BNF [Aho, 06] sendo geradas as tabelas de conceitos necessárias. Posteriormente, foi

implementado o comportamento de cada uma das construções sintácticas definidas, levando à construção de uma linguagem que permite o comportamento reactivo.

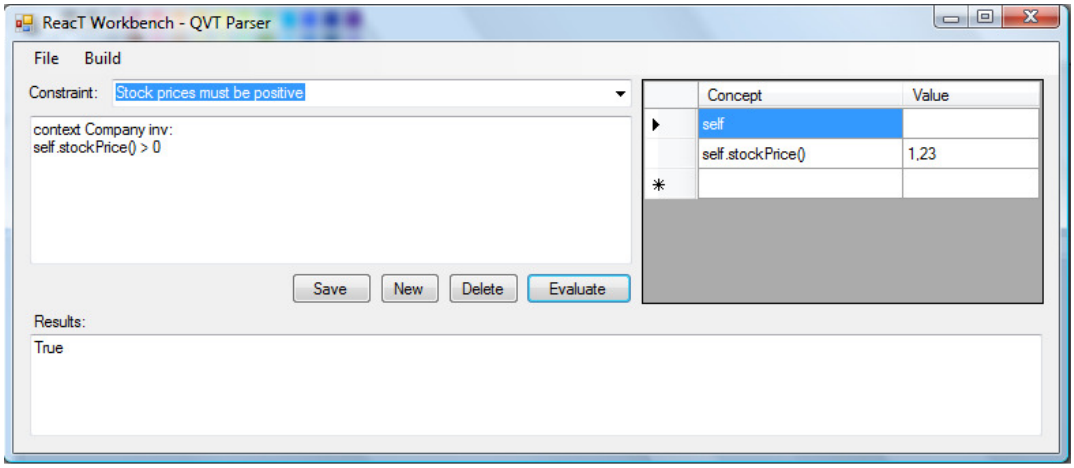


Figura 4.3 – Definição de uma restrição simples e teste com avaliação de uma subexpressão

Os artefactos existentes em diferentes suportes são acedidos fisicamente através de **contextos.** Cada contexto é definido através do nome pelo qual será referido em código, da localização de um artefacto ou grupo de artefactos (que pode estar eventualmente vazio, num dado momento) e do tipo de fornecedor externo. O contexto representa a ligação do grupo de artefactos sobre o qual se pretende realizar a rastreabilidade reactiva à aplicação, através do respectivo suporte, instanciado pela sua localização física. O fornecedor externo inclui o metamodelo do grupo de artefactos que é acedido. Cada contexto tem de referir o tipo de artefacto (incluído no metamodelo incluido no fornecedor externo) ao qual pertence. Só depois de existir uma representação do artefacto através de um contexto é que se pode referenciá-lo no código de rastreabilidade. No exemplo dado na Figura 4.3 é ilustrada uma avaliação de coerência em relação a uma restrição indicada. A ferramenta usada faz parte do ReacT-Workbench.

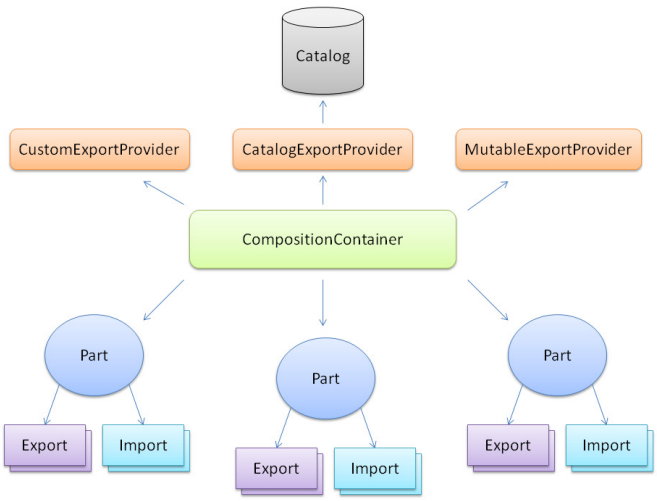


Figura 4.4 – Relações entre os elementos da MEF [CodePlex, 10]

Os fornecedores externos foram implementados de acordo com a MEF *(Managed Extensibility Framework)* [CodePlex, 10; Blumhardt, 09] que é uma tecnologia que permite integrar dinamicamente componentes autónomos numa aplicação. Segundo esta *framework c*ada fornecedor externo implementa uma *ComposablePart* e existe num *Catalog*. O facto de ser possível implementar diferentes fornecedores externos no ReacT Workbench, tornam-no numa ferramenta extensível. A criação de um fornecedor externo implica, entre outras actividades, a implementação da *interface ProviderContract* indicada em seguida:

```
public interface IProviderContract
{
    String ProviderName { get; }
    String Name { get; set; }

    void Fill(params string[] values);

    List<String> GetAttributeNames();
    void SetAttribute(string attKey, List<String> value);
    List<String> GetAttributeValue(string attKey);

    List<String> GetOperations();
    void InvokeOperation(string name, params string[] values);
    List<String> GetArtefactNames();
}
```

Um fornecedor externo é implementado por uma biblioteca que depois é usada num dos outros módulos da ferramenta. Dentro de cada fornecedor externo, cada uma das classes referentes aos conceitos do metamodelo implementa a *interface IProviderContract.* Dentro de cada biblioteca criada, podem existir diversos conceitos que servem de raiz para a navegação, funcionando como a porta de entrada da biblioteca. Nesse caso, o componente ReacT-Designer (Figura 4.6) permite marcar o conceito como *EntryPoint* através de uma metapropriedade denominada *Kind.* O grafo criado pode ser uma árvore ou não, consoante as relações existentes no respectivo metamodelo.

A *interface IProviderContract* implica que sejam implementadas propriedades para identificar cada um dos tipos de artefactos, bem como a sua instância. O método *Fill()* permite actualizar a representação do artefacto. O acesso aos dados específicos ao tipo de artefacto é feito através da invocação de métodos que acedem e actualizam uma estrutura de dados que relaciona cada atributo com o seu valor. Esta estrutura de dados deve ser implementada preferencialmente usando uma colecção dinâmica (e.g., Dictionary na plataforma .Net). As operações sobre cada tipo de artefacto são invocadas através do método *InvokeOperation()* que recebe um número variável de parâmetros. No entanto a implementação de cada uma das estruturas de dados dos conceitos *EntryPoint*, bem como das respectivas operações, deve tomar em conta a possível existência de ciclos. A estrutura de dados usada é suficientemente genérica para resolver referências entre grupos de conceitos.

O metamodelo dos artefactos tratados por um fornecedor externo é definido através de uma extensão ao Microsft Visual Studio realizada através do *template* aplicacional Domain Specific Language que faz parte do Microsoft Visual Studio SDK 2008 [Microsoft, 08]. Foi definido um *template* aplicacional próprio, baseado no diagrama de classes que inclui, para além de conceitos como ModelClass e ModelAttibute, outros conceitos como o *EntryPoint* já referido ou uma sintaxe própria para as associações entre as metaclasses. Ao ser criado o metamodelo (Figura 4.7) é gerado o código XML que o representa e que serve de *input* ao fornecedor externo. Esta geração produz-se

através de uma linguagem de geração automática a partir de *templates* denominada Text Templating Transformation Toolkit (T4) [Giesenow, 08]. O *template* do metamodelo dado na Figura 4.7 encontra-se no Anexo I. No mesmo metamodelo, existe um conjunto comum de conceitos a diversos fornecedores externos simplificados (e.g., MS Access, Oracle). O próprio código gerado constitui um grupo de artefactos que podem ser relacionados com o restante código.

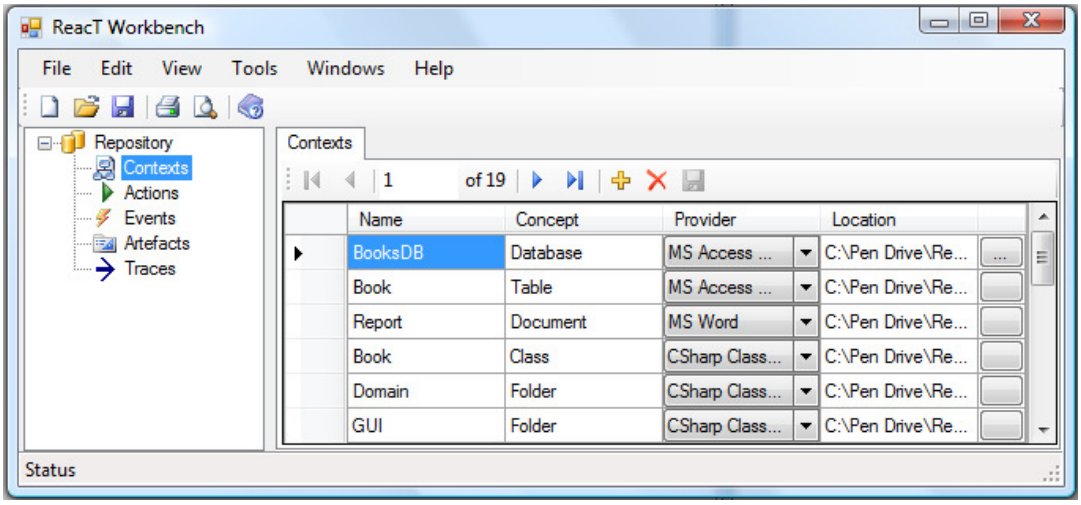


Figura 4.5 – Definição de um contexto através de um fornecedor externo no ReacT-Workbench

Como seria de esperar, o ReacT-Workbench envolve uma definição de conceitos em conformidade com o ReacT-MDD sendo por isso uma implementação válida. Por outro lado, apesar de o ReacT-Workbench ser uma ferramenta, tem igualmente características de uma *framework* visto permitir a sua extensão através do reconhecimento de um conjunto variável de fornecedores externos.

A definição de rastos pressupõe a identificação dos artefactos, através dos seus conceitos. Após os artefactos estarem disponíveis para serem relacionados, estes são escolhidos (Figura 4.6) e é-lhe atribuído uma descrição que permite documentar o rasto entretanto criado. Caso os fornecedores externos sejam suficientemente bem definidos, i.e., a sua implementação contenha as acções consideradas necessárias para um grupo de artefactos, é possível colocar a restrição adicional *IsEnforced*. Esta restrição permite que os artefactos em questão possam ser alterados automaticamente, caso o estado de coerência entre eles passe a ser incoerente. Nessa situação, o valor do rasto é *falso* e é invocado um evento que trata a acção respectiva.

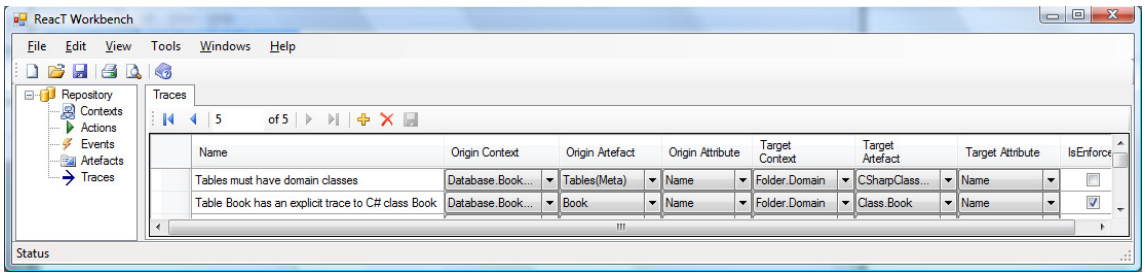


Figura 4.6 – Definição de rastos no ReacT-Workbench

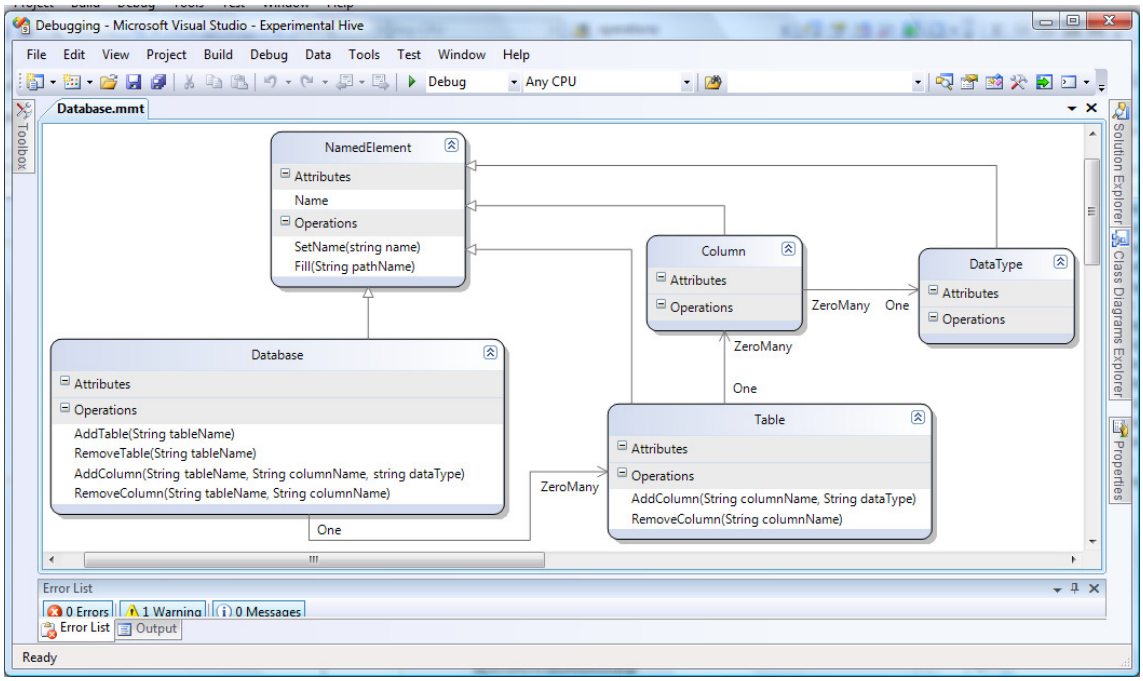


Figura 4.7 – Metamodelo do contexto Database.

O metamodelo criado utiliza a noção de herança entre tipos de artefactos, representados por conceitos do metamodelo. A herança é posteriormente seguida na geração dos artefactos que realizam a relação entre o metamodelo e os fornecedores externos.

O ReacT-Workbench, enquanto protótipo, permitiu conhecer as potencialidade e as dificuldades de implementação de uma ferramenta em conformidade com a plataforma ReacT-MDD. Entre as maiores dificuldades, talvez a mais relevante seja o facto de integrar tecnologias de áreas tão diversas como os analisadores de linguagens de programação, os SGBDs relacionais, as ferramentas CASE e a metamodelação.

Foram realizados dois casos de estudo que serão explicitados em seguida, tendo em vista a compreensão da utilização desta *framework,* no âmbito do desenvolvimento de aplicações.

5 Casos de Estudo

Com o intuito de validar o trabalho realizado, são apresentados dois casos práticos, com características muito diferentes um do outro. No primeiro caso é apresentado um domínio de aplicação denominado “Biblioteca”, com um número reduzido de artefactos e de fácil compreensão. No segundo caso é apresentado um domínio real, com um grande número de artefactos. Embora de forma diferente, ambos os casos de utilização contribuiram para entender as potencialidades e as dificuldades na utilização do ReacT-Workbench.

5.1 Caso de Estudo "Biblioteca"

Considere-se um domínio simplificado dos empréstimos de uma biblioteca onde é importante conhecer quais as obras existentes, as suas edições, os exemplares existentes fisicamente, bem como os sócios e os empréstimos realizados. O diagrama (Figura 5.1) revela as estruturas de dados, a nível conceptual, do domínio descrito. Os identificadores das classes foram propositadamente deixados com acentos e caracteres normalmente não usados na implementação. Trata-se de um diagrama conceptual, em que o conhecimento do problema deve ser o mais evidente possível. A legibilidade deste diagrama é importante na medida em que pode servir para comunicar com intervenientes com pouco conhecimento técnico.

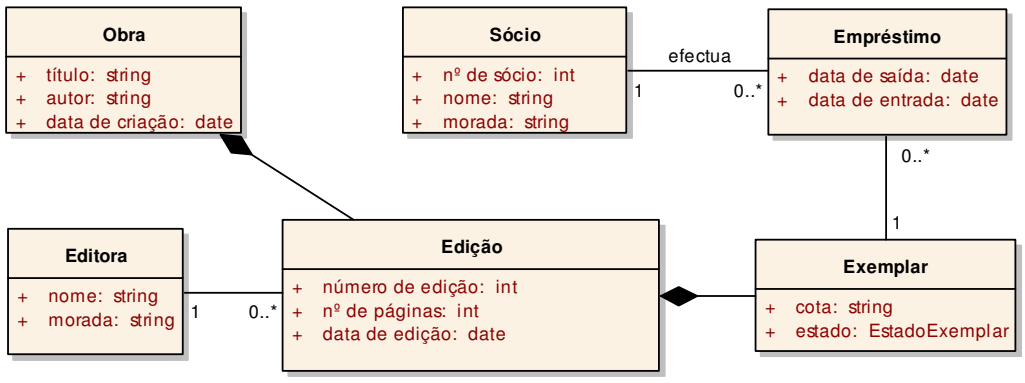


Figura 5.1 – Diagrama de classes conceptual dos dados do empréstimo (1)

O diagrama foi construído na ferramenta Enterprise Architect que avisa o utilizador repetidamente cada vez que um identificador é construído com um carácter de espaço (que dificulta a geração directa). Nesta ferramenta, caso seja feita a geração de artefactos de código C# directamente a partir

deste diagrama, o gerador inclui os espaços nos identificadores criando erros sintácticos óbvios. Como regra, a geração de artefactos de código deve ser feita sobre os diagramas correspondentes aos modelos PSM, senão a sua qualidade está à partida comprometida.

O diagrama de classes conceptual serve para que o utilizador possa compreender os conceitos, independentemente da tecnologia a adoptar. Pode considerar-se que o diagrama apresentado poderia estar enquadrado num modelo PIM, segundo a terminologia MDA referida na secção 2.2.2. Considere-se agora um outro modelo PIM, com os identificadores alterados para que a implementação possa ser mais simples, ou para que utilizadores já conhecedores dos pormenores do projecto possam ter uma ideia conceptual do problema.

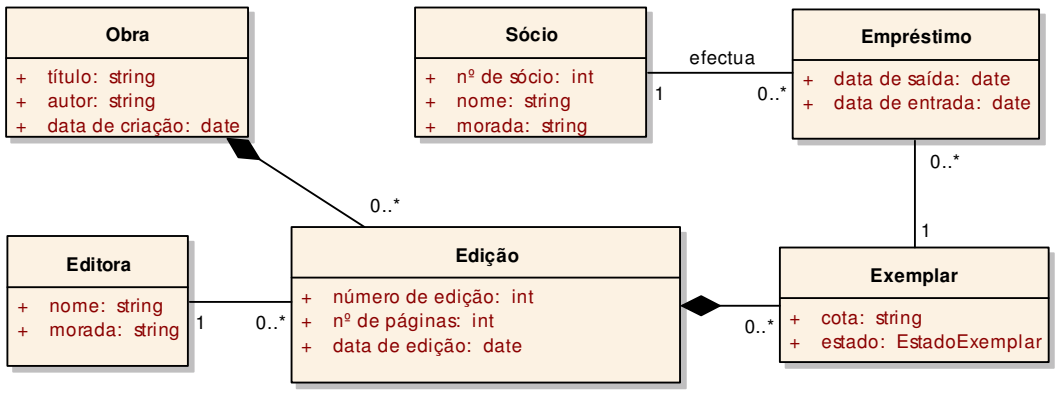


Figura 5.2 - Diagrama de classes conceptual dos dados do empréstimo (2)

Este diagrama (Figura 5.2) continua a corresponder a um modelo PIM, visto ser independente da tecnologia. Podem coexistir diversos modelos PIM, sobre o mesmo domínio, reflectindo diferentes perspectivas ou vistas do mesmo. Como neste caso, podem mesmo coexistir diversos diagramas do mesmo tipo, cada qual com objectivos específicos (e.g., variando a língua, o nível de detalhe, o tipo de elementos descritos).

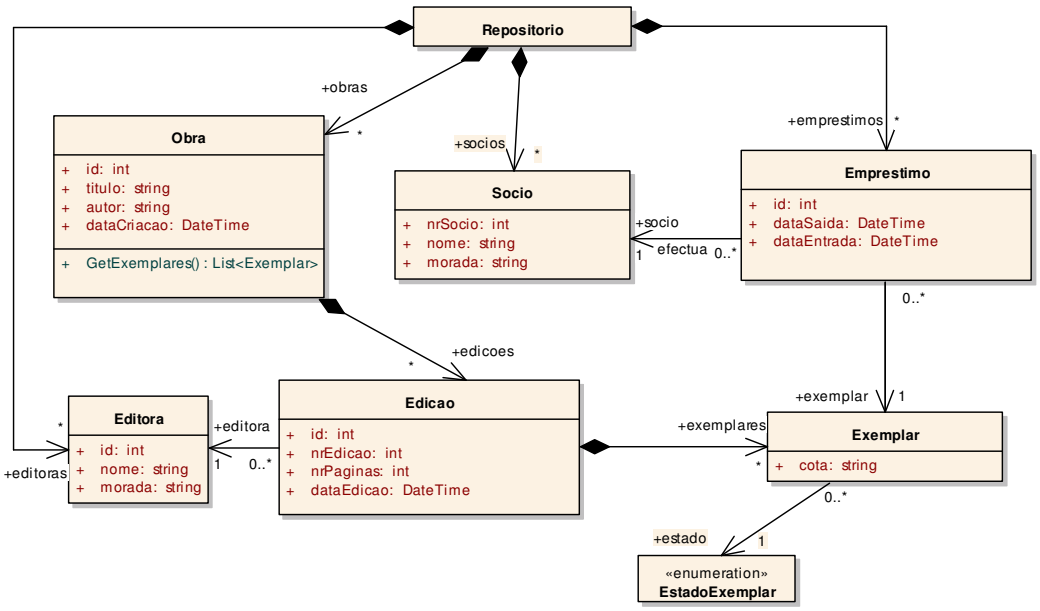


Figura 5.3 - Diagrama de classes dependente da plataforma .NET e da linguagem C#

O diagrama da Figura 3.3 representa uma implementação específica da linguagem C#. Como é visível: o tipo *date* foi substituído por *DateTime*; acrescentou-se uma operação *GetExemplares()* que devolve uma lista de objectos da classe *Exemplar*; estão visíveis os pormenores de implementação das relações entre as classes e destas com os tipos enumerados; a forma como os objectos são guardados é visível através da classe *Repositorio* que só existe neste diagrama; as classes que não têm identificadores explícitos (como *nrSocio* na classe *Socio*) têm um identificador *id* que facilita a relação destas classes com uma base de dados relacional. Esta implementação é apenas uma de muitas que é possível concretizar com este tipo de tecnologia. Alterando a forma como a navegação entre as classes se processa, ou como são guardados os objectos, mesmo sem alterar a tecnologia é possível alterar completamente a solução.

Como o diagrama reproduz já uma solução, é possível gerar automaticamente código C# a partir deste. O código produzido directamente pela ferramenta apresenta ainda diversos erros de geração que têm de ser resolvidos para que a implementação seja correcta. Tome-se como exemplo a geração da classe *Edicao* da qual em seguida se reproduz um segmento (após reformatação):

```
...
using BibliotecaPSM1;
namespace BibliotecaPSM1 {
    public class Edicao {
           public int id;
           public int nrEdicao;
           public int nrPaginas;
           public DateTime dataEdicao;
           public Exemplar exemplares;
           public Editora editora;
           public Edicao(){
           }
    }//end Edicao
}//end namespace BibliotecaPSM1
```

O gerador não respeitou a implementação do atributo *Exemplares* como uma lista, dando-lhe o tipo *Exemplar.* A própria geração deste atributo como lista poderia ser feita de diversas formas (e.g., *List<Exemplar>, ArrayList*, *Exemplar[]*, ou outro agregado de objectos), nenhuma delas como a que foi realizada.

Este facto significa que cada vez que o código for gerado o erro será repetido, tendo de ser revisto e corrigido manualmente. Só então a coerência dos artefactos gerados com os artefactos conceptuais é assegurada. Com este exemplo, de evidente simplicidade, foram gerados cinco tipos de artefactos (Figura 5.4) que na verdade representam três diagramas e oito ficheiros com classes C#, e um ficheiro com código MSIL (*Microsoft Intermediate Language* [Microsoft, 10]) eventualmente incoerentes.

Foi criado um *fornecedor* EnterpriseArchitect que actua como uma ligação aos artefactos criados por esta ferramenta. O *fornecedor*, neste caso, permite apenas conhecer o estado actual dos artefactos criados. Uma implementação mais abrangente poderia interagir com a ferramenta provocando alterações nos diagramas pretendidos. No entanto, naquilo que diz respeito ao problema em causa, o *fornecedor* implementado mostrou-se suficiente para que se consiga analisar a coerência entre os diversos artefactos e avisar o utilizador, quando é necessário tomar alguma medida.

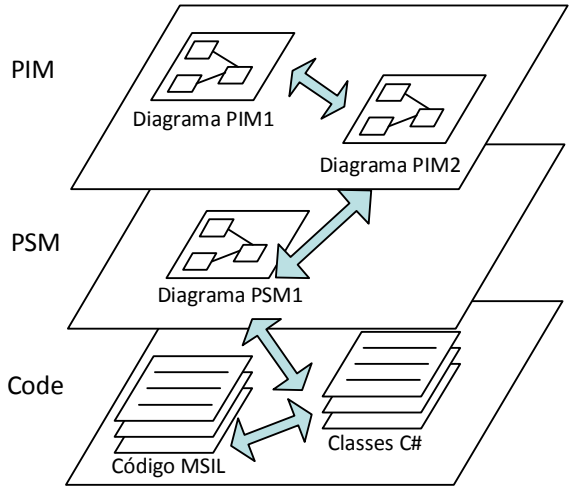


Figura 5.4 – Artefactos produzidos e as suas relações

Foi definida a relação QVT que representa a dependência entre o conceito de *composição* em UML e a implementação proposta em C#.

```
relation UMLComposition2CSharpClasses {
    checkonly domain umlEAClassDiagramPSM1 comp:Composition {
                        s = source: EAClass { name = n1 },
                        t = target: EAClass { name = n2 } }
    checkonly domain cSharpPSM1 c:Class {
                        name = n2,
                        att = list:Variable { name = n1 + 's',
                                    type = 'List' + n1 + '>' }
}
```

Adicionalmente, foram acrescentados dois eventos que permitem monitorizar as alterações produzidas sobre o referido diagrama de classes, bem como sobre as classes C#.

```
event domain umlEAClassDiagramPSM1 d: Diagram {
          actionType = ActionTypes::CRUD }

event domain cSharpPSM1 c: Class {
          actionType = ActionTypes::CRUD }
```

Embora com limitações inerentes a um protótipo aplicacional, foi atingido o objectivo de relacionar os diversos artefactos, quanto à sua coerência. O exemplo descrito, após alterações nos diagramas ou no código, comporta-se como um único artefacto, apesar de ter sido produzido em duas ferramentas diferentes (Enterprise Architect e Visual Studio).

A ferramenta permite igualmente realizar a rastreabilidade dos artefactos, nomeadamente através das relações de coerência que foram introduzidas. A verificação do código MSIL não foi feita por este ser gerado e verificado pelo IDE Microsoft Visual Studio a partir dos artefactos de código fonte escritos em C#.

5.2 Caso de Estudo "gestBarragens"

O projecto gestBarragens - Sistema Integrado de Gestão da Informação para o Controlo de Segurança de Barragens [Silva et al., 08], serviu também para verificar o interesse prático do problema tratado, bem como para fornecer dados de um contexto real a este trabalho. Trata-se de um conjunto de aplicações integradas, que gerem informação técnica sobre as barragens existentes em Portugal permitindo conhecer o estado e a evolução das condições físicas das mesmas. O gestBarragens [Silva et al., 08] suporta os seguintes processos: o processo de observação de grandezas físicas a partir da exploração, manual ou automática, dos sistemas de observação instalados nas barragens; o processo inerente às inspecções visuais realizadas periodicamente por intervenção humana; o processo de gestão e consulta geral da informação patrimonial e documental relacionada com as barragens; e o processo de detecção de situações de anomalia nos dados observados que pode implicar a tomada de decisões de modo a garantir a segurança das barragens. Em complemento, o gestBarragens apresenta mecanismos avançados de consulta e análise da informação (e.g., suportados por sistemas de informação geográficos e por sistemas de reporting) e mecanismos de sincronização da informação entre as diferentes instâncias do sistema mantidas por entidades distintas (e.g., LNEC, INAG, donos de obra).

O gestBarragens tem uma *interface web* e está disponível para um número restrito de utilizadores que fazem parte das organizações intervenientes na operação, manutenção e fiscalização das barragens no território nacional (e.g., LNEC, EDP, INAG).

O gestBarragens envolve diferentes tecnologias como um SGBD Oracle, um SGBD Microsoft SQL Server, um servidor Web MS-IIS, um servidor SIG ESRI-ArcIMS. Naquilo que diz respeito ao presente caso de estudo foram apenas considerados os artefactos referentes ao código (classes C#), às tabelas geridas pelo SGBD Oracle e aos modelos criados em Enterprise Architect.

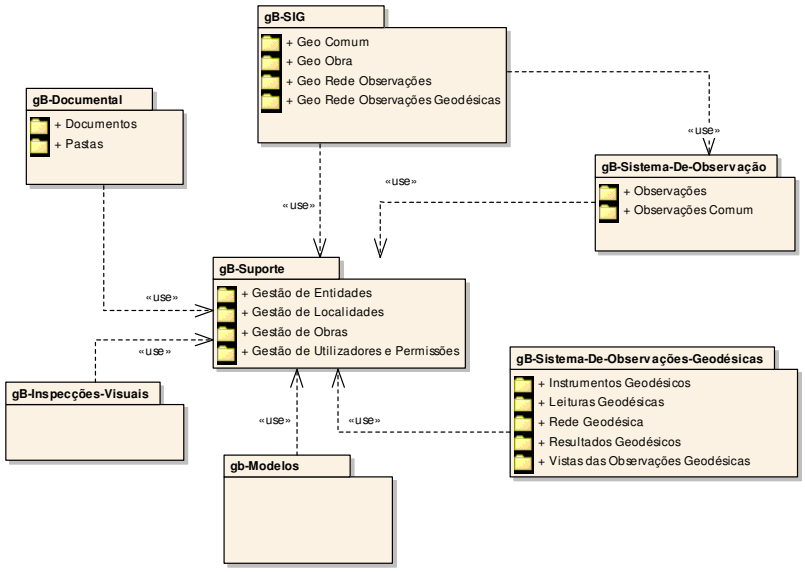


Figura 5.5 – Visão geral do "gestBarragens" [Silva et al., 08]

O gestBarragens é concebido e desenvolvido de forma modular e baseado numa arquitectura de dados comum, sendo evidenciados os seguintes módulos aplicacionais (Figura 5-5): gB-Suporte, gB-Sistema-De-Observação, gB-SIG, gB-Documental, gB-Sistema-De-Observações-Geodésicas, gB-Inspecções-Visuais e gB-Modelos. O módulo gB-Suporte é um sistema de gestão e configuração de vários aspectos comuns e de suporte ao gestBarragens, envolvendo a gestão de entidades transversais ao sistema (e.g., gestão de utilizadores, entidades, obras, elementos de obra), isto é, que podem ser utilizados pelos restantes módulos (Silva et al., 08).

Mostram-se em seguida, a título de exemplo, alguns diagramas que fazem parte da documentação do gestBarragens.

O diagrama de pacotes da Figura 5.5 representa as áreas aplicacionais mais importantes do gestBarragens através de uma visão de muito alto nível. O gestBarragens está desenvolvido com uma arquitectura de quatro camadas que tornam independentes a interface com o utilizador, a lógica de negócio, o acesso aos dados e os próprios dados (Figura 5.6). O diagrama de instalação da Figura 5.7 representa os diferentes componentes aplicacionais relacionados com aspectos da infraestrutura tecnológica.

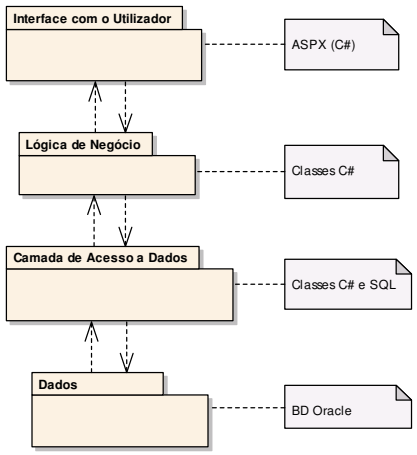


Figura 5.6 – Arquitectura de software de n camadas do gestBarragens

Na Figura 5.9 está representada a forma como os componentes da interface gráfica (e.g., páginas ASPX) interagem através das respectivas acções que podem ser realizadas dentro de cada componente. As relações de coerência entre os artefactos *classes C#* representados na figura (ela própria um artefacto) através dos componentes respectivos (que se comportam igualmente como artefactos é conseguida de forma genérica relacionando cada página aspx com um artefacto presente num modelo. Porém, neste caso em particular, alguns dos componentes podem existir em diferentes diagramas o que obriga a criar a especificar melhor a relação de coerência criando rastos específicos para estes componentes.

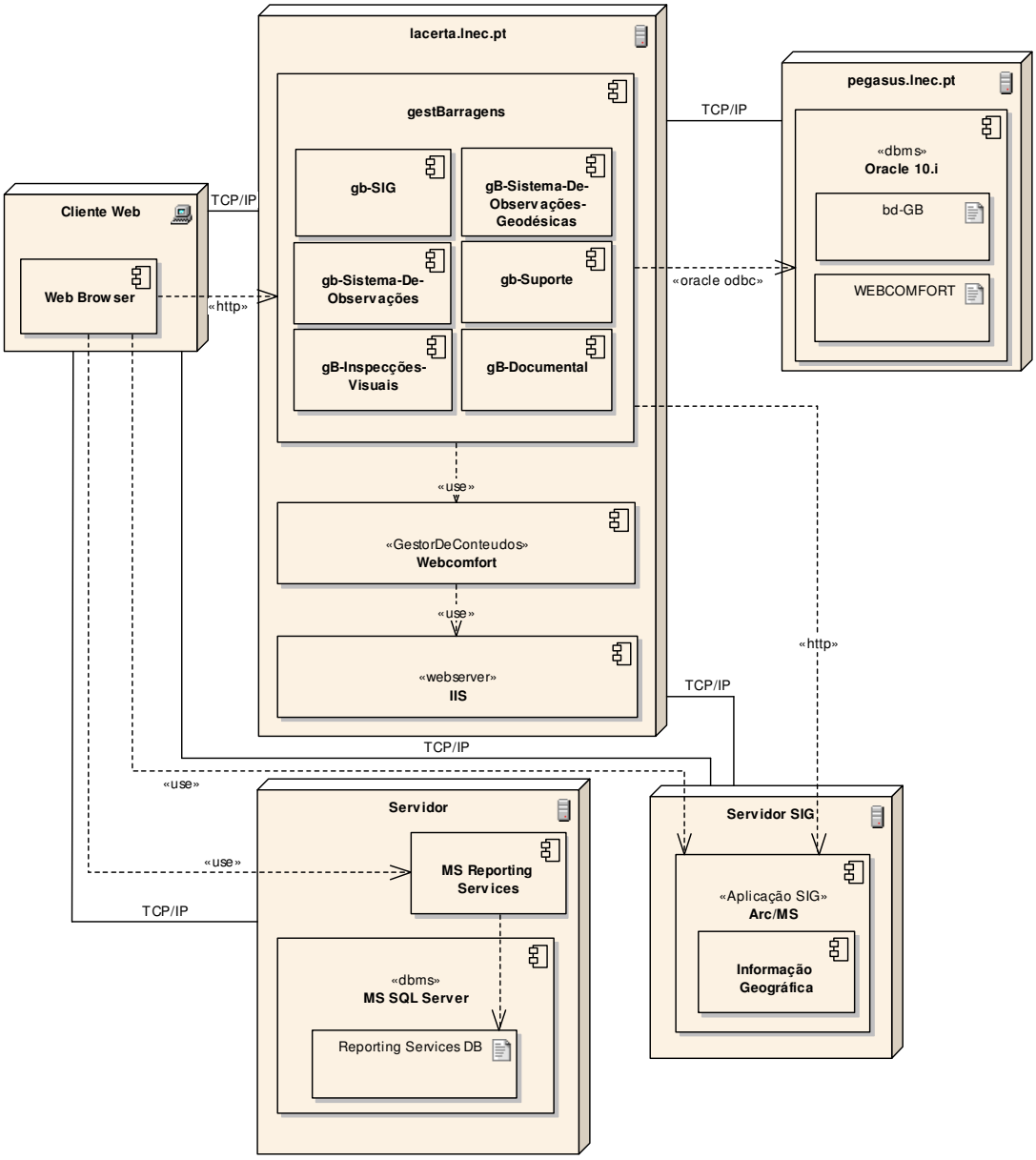


Figura 5.7 – Arquitectura aplicacional do gestBarragens (diagrama de instalação)

No âmbito deste trabalho foi elaborado um relatório referente à especificação técnica do sistema [Silva et al., 08] que complementa os diferentes documentos já existentes. Para além de um documento de "Especificação de Requisitos Técnicos" [Silva et al., 06] tinham já sido produzidos três manuais de utilização tratando diferentes áreas da aplicação [Silva et al., 07a, b, c]. A especificação técnica do sistema pretendeu dar uma imagem dos aspectos técnicos do projecto, entretanto identificados, complementando os relatórios anteriores de duas formas: (1) abordando aspectos anteriormente não tratados; (2) introduzindo alterações a partes dos documentos já existentes.

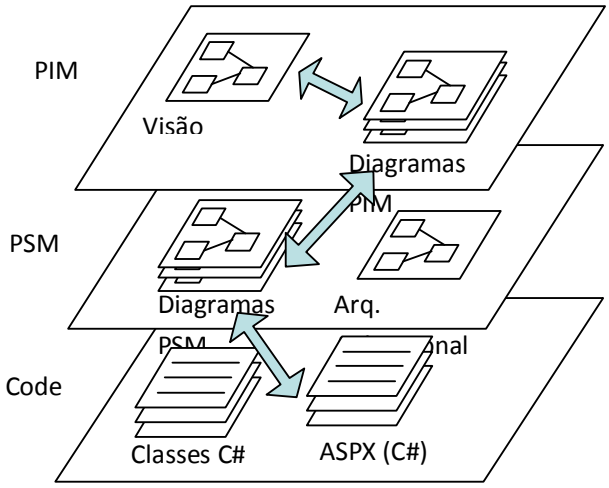


Figura 5.8 – Alguns artefactos do gestBarragens

Sendo esta uma aplicação que se encontrava já em fase de produção (i.e., já estava a ser utilizada com o fim para que foi criada) era importante entender se as alterações que a fase de desenvolvimento produziu, tornavam ou não ultrapassados os artefactos existentes.

Tipo de Elemento UML	Total (Vers. A)	Total (Vers. B)	Total (Vers. C)
Pacotes:	45	71	103
Diagramas:	53	80	119
Componentes:	7	65	350
Associações:	1368	1703	1656
Generalizações:	322	359	415
Notas:	28	12	14
Classes:	1119	1735	1454
Dependências:	93	138	625
Atributos:	7631	16616	11356
Operações:	9695	15414	11619
Agregações:	22	26	27

Tabela 5.1 - Número total de artefactos que representam elementos UML presentes em três versões do repositório (da ferramenta CASE) do *gestBarragens*

Cedo se verificou, após um trabalho de comparação entre os elementos presentes nos diagramas e os artefactos implementados, que existiam diferenças (aceitáveis quando é comparado um

documento de especificação de requisitos, produzido no início de um projecto, com o resultado final[21] do mesmo).

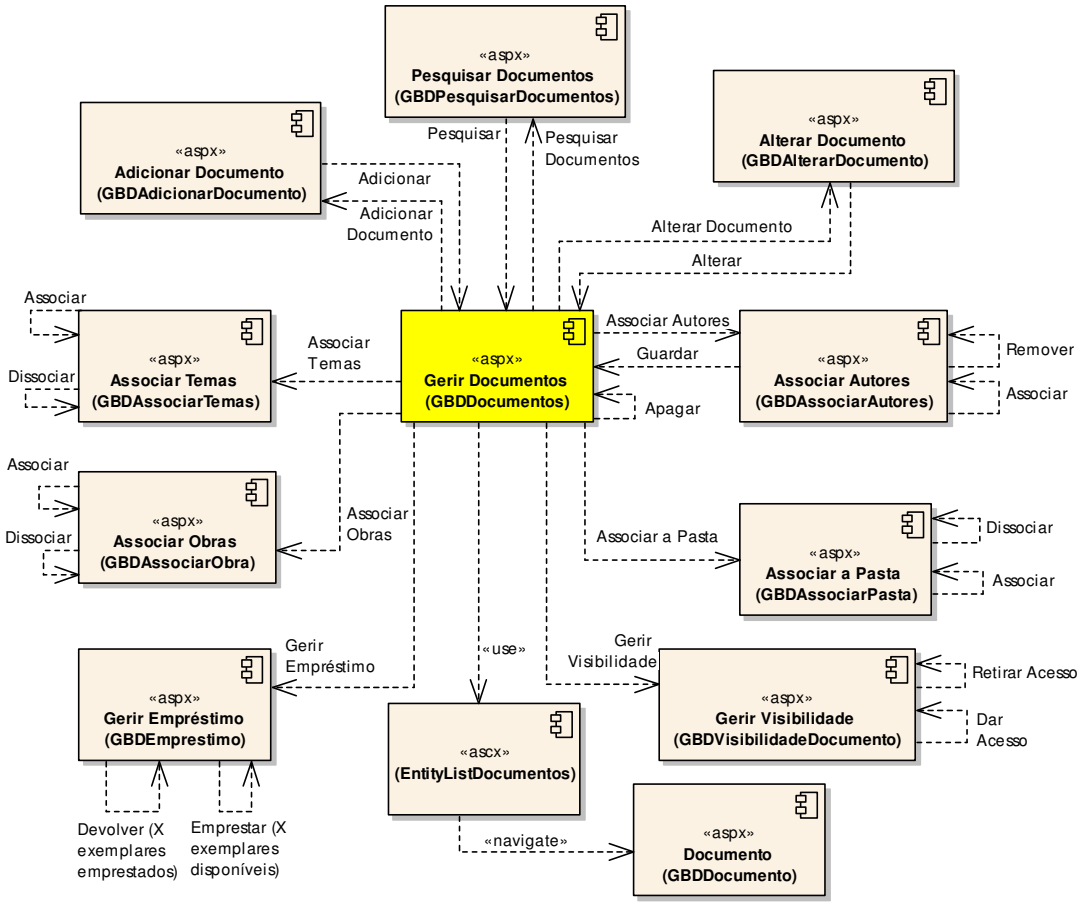


Figura 5.9 – Diagrama de componentes da interacção Gerir Documentos

A ferramenta CASE usada na documentação do projecto foi o Enterprise Architect [Sparx, 10] Na Tabela 5.1 estão presentes, para cada uma de três versões do repositório os respectivos totais de artefactos que representam elementos conceptuais UML. Embora existam outros elementos, estes são os mais importantes em número de ocorrências. Na versão A foram introduzidos os elementos constantes da documentação prévia (manualmente e por importação de artefactos existentes). Na versão B foram criados manualmente novos diagramas e foi feita a importação de um conjunto de artefactos (e.g., classes C#) que levaram à criação de um elevado número de elementos UML. Na versão C os modelos foram refinados e foram eliminados muitos elementos UML já existentes que serviram apenas como base para a construção de alguns diagramas (e.g., diagramas de componentes).

[21] Obviamente não existe propriamente um "resultado final" na medida em que o projecto é continuamente alterado ao longo do tempo. Considera-se aqui como a entrada em produção do projecto como ponto de comparação.

Artefactos	Total
Classes C#	1815
Ficheiros ASPX	426
Ficheiros ASCX	52
Tabelas na BD	397
Documentos de apoio (manuais, relatórios, etc.)	5
Diagramas UML	119

Tabela 5-2 - Total de artefactos presentes no *gestBarragens*, agregados por tipo

Como foi referido anteriormente, segundo a abordagem ReacT-MDD os diagramas podem ser considerados artefactos, inseridos num outro denominado *repositório*. Os diagramas de classes correspondentes à modelação de domínio (já existentes na documentação anterior) sofreram algumas alterações que visaram melhorar a sua qualidade no que diz respeito aos seguintes aspectos:

- Coerência entre vistas de diferentes níveis - O modelo da base de dados nem sempre reflectia o modelo de domínio. Alguns modelos de domínio foram alterados para que exista maior relação entre estes e os modelos da base de dados.
- Normalização de nomeação - A nomeação dos elementos foi normalizada.
- Refinamentos de concepção - Os diagramas do modelo de domínio foram produzidos por diferentes pessoas com estilos diferentes que tornam a leitura conjunta mais complexa. O estilo foi uniformizado facilitando-se assim a leitura dos referidos diagramas.

Neste âmbito, o ReacT-MDD foi usado para:

- Obter os totais de elementos existentes - Através do fornecedor externo EA os diferentes repositórios foram analisados no que diz respeito aos elementos relevantes.
- Criar algumas regras de coerência simples - Foram criadas regras de coerência de teste para se conhecer o tempo de resposta num projecto com um elevado número de artefactos.

No ambiente de teste criado, o tempo de resposta para uma alteração ao estado do sistema foi, em média, inferior a cinco segundos (realizando-se 40 testes envolvendo cinco regras diferentes).

6 Conclusões

A actividade de desenvolvimento de sistemas de informação tem progredido ao longo dos anos, principalmente naquilo que diz respeito às linguagens utilizadas e às metodologias adoptadas. No entanto as ferramentas de apoio à modelação continuam a realizar quase o mesmo tipo de actividades (embora mais rápida, fácil e correctamente) do que no passado. É necessário o aparecimento de uma nova classe de aplicações que consiga produzir, gerir e manter ao longo do tempo as relações entre os diversos artefactos produzidos por estas ferramentas e as restantes. Todas as fases do ciclo de vida de uma aplicação informática (e.g., definição de requisitos, análise, concepção, implementação, testes, operacionalização, manutenção) produzem artefactos relacionados com a estrutura da própria aplicação. Estes artefactos devem poder ser relacionados, através de uma especificação aberta, permitindo-se assim a sua actualização.

A coerência entre os artefactos pressupõe o conhecimento não só de cada um dos artefactos como das relações possíveis entre eles. Assim sendo, a criação de relações entre artefactos implica que se estabeleçam também relações ao nível dos metamodelos correspondentes. A metamodelação é usada neste contexto para permitir o estabelecimento dessa *estrutura de relações*.

Quando um artefacto (conceptual ou operacional) não se encontra num estado coerente com os demais é a qualidade dos artefactos no seu todo que fica comprometida. As decisões sobre a aplicação deixam de ser determinísticas, i.e. deixa de haver uma *certeza* sobre o próprio estado actual dos artefactos da aplicação. As decisões são tomadas com alguma probabilidade de se adequarem ao verdadeiro estado do sistema, estando eventualmente já ultrapassadas. Neste contexto, é necessário um mecanismo que permita impor regras que mantenham a coerência entre os artefactos, à semelhança de outras indústrias existentes.

A importância deste problema pode ser facilmente verificada se se analisar aquilo que se passa na Construção Civil, e.g., com a construção de um edifício. Se um arquitecto realizar um projecto e não existir alguém que verifique a sua execução, podem existir desvios significativos em relação àquilo que foi determinado inicialmente. Estes desvios podem levar inclusivamente a um aspecto diferente daquele que o arquitecto tinha em mente quando produziu o projecto, ou mesmo a erros estruturais que eventualmente podem ter consequências negativas para o trabalho posterior. Uma acção tão simples como acrescentar uma tomada numa parede já terminada necessita que os esquemas que localizam as redes (e.g., eléctrica, dados, água, saneamento) do edifício estejam actualizadas. Quando tal não acontece podem existir acidentes, eventualmente com risco para o executante da accção. Para que aquilo que é efectivamente construído corresponda ao projecto, deve existir um acompanhamento da obra por parte de um técnico especializado que emprega técnicas de Engenharia para realizar essa actividade. Neste contexto o acompanhamento totalmente automático

é, pelo menos por enquanto, impossível visto que tem de existir uma verificação de uma realidade física *in loco*.

O desenvolvimento de aplicações é um âmbito com particularidades que o diferenciam de outras áreas da Engenharia, nomeadamente da Engenharia Civil. Apenas naquilo que diz respeito à variabilidade das aplicações podem referir-se os seguintes aspectos:

- As aplicações após começarem a ser usadas podem ser actualizadas com grande periodicidade (e.g., diária). Um edifício pode ser reparado ou sofrer alterações pontuais ao longo da sua vida útil, de dezenas ou centenas de anos.

- Ocasionalmente os requisitos das aplicações podem mudar, podendo essa alteração envolver mudanças importantes nos conceitos ou nas tecnologias envolvidas. As alterações aos requisitos de um edifício podem ocorrer eventualmente, embora essas alterações sejam menos frequentes (e.g, criar mais uma faixa de rodagem numa ponte, transformar um edifício de escritórios num centro comercial).

- Os diagramas que servem para representar os constituintes da aplicação e a sua funcionalidade não têm uma linguagem completamente normalizada, intuitiva e objectiva. Existem diferentes processos, metodologias, notações, tipos de ferramentas e paradigmas de programação, evoluindo continuamente.

- A tecnologia envolvida na produção de aplicações muda mais rapidamente do que em qualquer outra indústria, sendo grande parte dela constituída por outras aplicações.

A variabilidade, traduzida em actualização, evolução ou manutenção, revela que as aplicações devem ser consideradas segundo uma perspectiva dinâmica. Uma aplicação não é um produto que depois de acabado é usado até ser substituído por outro. Pelo contrário, pode considerar-se que uma aplicação evolui (eventualmente através de versões das suas partes) ao longo do tempo, até deixar de ser mantida e utilizada.

Como as aplicações são materializadas em artefactos, são estes que têm de ser verificados e actualizados para que a aplicação evolua correctamente. As alterações de estado (relevantes neste contexto) são aquelas que dizem respeito à estrutura da aplicação. Assim sendo, cada alteração a um artefacto (conceptual ou operacional) deve ser verificada na óptica da coerência.

A capacidade de rastrear artefactos, ou seja de seguir relações entre um dado artefacto e os demais, só é possível se estes estiverem sincronizados. Por isso enquadrou-se a rastreabilidade numa componente reactiva, em que a coerência é verificada e são tomadas decisões quanto à sua manutenção.

Esta verificação, juntamente com a capacidade de reagir a alterações incorrectas, permitindo a rastreabilidade reactiva, constitui um novo tipo de aplicações, ou pelo menos uma nova funcionalidade. Vendo o problema de outra forma, pode afirmar-se que o que se pretende com esta nova categoria de aplicações é realizar de forma sistematizada e o mais automatizadamente possível as acções de acompanhamento do desenvolvimento e manutenção para que todas as decisões

tomadas sobre um nível conceptual não sejam comprometidas pela implementação desses conceitos num outro nível. Não é importante para esta abordagem se o utilizador desenvolve modelos a partir de código ou *vice versa*. Os artefactos existem (podendo ter um carácter permanente, e.g. num sistema legado), são criados pelos utilizadores ou são gerados automaticamente, e a propagação de alterações deve ser feita em todos os níveis conceptuais.

A *framework* proposta pode operar em conjunto com as ferramentas já utilizadas para a modelação e desenvolvimento de sistemas de informação sendo ortogonal às metodologias e linguagens utilizadas. Embora a aplicação produzida segundo a framework constitua, no estado actual, um protótipo, permitiu verificar a exequibilidade dos conceitos apresentados. A criação de relações de coerência entre artefactos produzidos por diferentes aplicações, permitiu aumentar o nível de confiança nos artefactos documentais. Apesar de existir uma verificação da coerência, o estádio de desenvolvimento do protótipo não permite tirar conclusões no que diz respeito a uma quantificação de eventuais ganhos de produtividade resultantes da sua utilização. Porém, foi conseguida uma uniformização no tratamento dos fornecedores externos, para além dos conceitos usados, que facilitam e melhoram a qualidade do tratamento da rastreabilidade reactiva.

Com esta abordagem permite-se que sejam utilizadas técnicas mais próximas da Engenharia no desenvolvimento e manutenção de sistemas de informação. Não se trata de limitar a criatividade ou a liberdade de desenvolver soluções engenhosas para os problemas que vão surgindo. Trata-se, isso sim, de uma forma de criar e manter mais facilmente, com melhor qualidade e com menores custos, os projectos de sistemas de informação.

6.1 Trabalho Futuro

O trabalho futuro centrar-se-à nas seguintes áreas:

- Criação de fornecedores externos para a ferramenta ReacT-Workbench – Implementação de ligações a outras ferramentas como SGBD, ferramentas CASE e processadores de texto.
- Definição e implementação de metamodelos mais abrangentes – Criação e aperfeiçoamento dos metamodelos necessários à execução da ferramenta ReacT-Workbench tendo em vista o tratamento de um maior número de artefactos bem como de um tratamento mais aprofundado.
- Criação de *parsers* para diferentes linguagens de programação – Inclusão, através de fornecedores externos, de *parsers* para diferentes linguagens de programação (ou descrição).
- Criação de interfaces gráficas que optimizem a criação de rastos entre artefactos, quer através de editores de texto optimizados, quer através de ferramentas gráficas.

Publicações Científicas e Congressos Nacionais e Internacionais

Congressos Nacionais

Aspectos de Sincronização em Modelos UML, Marco Costa, Alberto Rodrigues da Silva*,* Actas da 6ª Conferência da Associação Portuguesa de Sistemas de Informação (CAPSI'2005), Bragança, 2005

Congressos Internacionais

XIS Generative Programming Techniques, Alberto Rodrigues da Silva, Gonçalo Lemos, Tiago Matias, Marco Costa, Generative Programming and Component Engineering (GPCE'03), Erfurt, Germany, 2003

RT-MDD Framework – A Practical Approach, Marco Costa, Alberto Rodrigues da Silva, 3rd European Conference on Model Driven Architecture (ECMDA), Traceability Workshop, Haifa, Israel, 2007

Synchronization Issues in UML Models, Marco Costa, Alberto Rodrigues da Silva, 9th International Conference on Enterprise Information Systems (ICEIS), Funchal, Portugal, 2007

ReacT-MDD – Reactive Traceability in Model-Driven Development, Marco Costa, Alberto Rodrigues da Silva, 12th International Conference on Enterprise Information Systems (ICEIS), Funchal, Portugal, 2010 (em fase de aprovação)

Publicações Científicas

Arquitecturas de Sistemas de Informação e a Iniciativa MDA, Marco Costa, Alberto Rodrigues da Silva, Anais Científicos da Universidade Independente, Lisboa, 2003

ReacT-MDD – Rastreabilidade Reactiva de Artefactos no Desenvolvimento de Sistemas de Informação, Marco Costa, Alberto Rodrigues da Silva, Revista Lusíada: Tecnologias de Informação, 1/2010, Universidade Lusíada Editora, 2010

Relatórios Técnicos

gestBarragens - Sistema Integrado de Gestão da Informação para o Controlo de Segurança de Barragens, LNEC - Especificação Técnica do Sistema, Alberto Rodrigues da Silva, Marco Costa, Luis de Sousa, SIQuant - 01/2008

Índice de Figuras

Bibliografia

[Agrawal, 03] *Metamodel Based Model Transformation Language*, Aditya Agrawal. OOPSLA Companion, 2003

[Aho, 06] *Compilers: Principles, Techniques and Tools,* A. Aho, M. Lam, R. Sethi, J. Ullman, 2nd Edition, Addison Wesley, 2006

[Ahtee, 96] *Evaluation of CASE Tools By a Test Specifications Model (TSM),* Tero Ahtee, 3rd Doctoral Consortium on Advanced Information Systems Engineering, Heraklion, Crete, 20 a 21 de Maio de 1996

[Akehurst, 04] *Proposal for a Model Driven Approach to Creating a Tool to Support the RM-ODP,* Dave Akehurst, The 8th International IEEE Enterprise Distributed Object Computing Conference, Monterey, California, USA, 20-24 de Setembro de 2004

[Akehurst et al., 04] *The Kent modelling framework user guide.* Dave Akehurst, Octavian Patrascoui, Rob Smith, http://www.cs.kent.ac.uk/projects/kmf

[Albizuri-Romero, 00] *A Retrospective View of CASE Tools Adoption*, Miren Begoña Albizuri-Romero, SIGSOFT Softw. Eng. Notes 25, 2, Págs.46-50, Março de 2000

[AndroMDA] *AndroMDA*, www.andromda.org

[Baik, 00] *The Effects Of CASE Tools On Software Development Effort*, Jongmoon Baik, Dissertação de PhD, University of Southern California

[Baik et al., 00] *Empirical Analysis of CASE Tool Effects on Software Development Effort,* Jongmoon Baik e Barry Boehm, Center for Software Engineering, http://sunset.usc.edu/publications/ TECHRPTS/2000/usccse2000-504/usccse2000-504.pdf, 2000

[Becker et al., 97] *Supporting the Evaluation and Selection of CASE Tools,* Karin Becker e Dilnei Venturini, CiteSeer Scientific Literature Digital Library, http://citeseer.ist.psu.edu/ 145126.html, 1997

[Beckworth, 93] *Selection Criteria for CASE Tools,* Geoff Beckworth, Dep. Of Computing and Mathematics, Deakin University, Australia, http://citeseer.ist.psu.edu/182672.html, 1993

[Bell, 08] *Software Development Amidst the Whiz of Silver Bullets*, Alex E. Bell, Communications of the ACM, Vol. 51, Nr. 8, Agosto de 2008

[Berard, i] *Information Technology Management,* Edward V. Berard, The Object Technology, http://www.itmweb.com/essay553.htm

[Bettin, J. et al., 03] *Generative Model Transformer: An Open Source MDA Tool Initiative*, OOPSLA, http://www.softmetaware.com/oopsla2003/pos10-bettin.pdf, 2003

[Bézivin, 98a] *Who's Afraid of Ontologies?,* Jean Bézivin, OOPSLA'98 Workshop on Model Engineering, Methods and Tools Integration with CDIF, Vancouver, 1998

[Bézivin, 98b] *The sBrowser: A Prototype Meta-Browser for Model Engineering,* OOPSLA'98 Workshop on Model Engineering, Methods and Tools Integration with CDIF, Vancouver, 1998

[Blumhardt, 09] *Hosting the .NET Composition Primitives*, Nicholas Blumhardt, Microsoft Corporation, http://mef.codeplex.com/Project/Download/FileDownload.aspx?DownloadId =62133, 2009

[Boive et al., 01] *UML CASE Tool Review*, Tom Boive e Peter Ordén, Aland's Institute of Technology - ATL (Mariehamn, Finland), May 2001, http://www.ie.inf.uc3m.es/ggenova/pfc-OrdenBoive2001.html

[Booch, 94] *Object-Oriented Analysis And Design - With Applications Second Edition,*Grady Booch, The Benjamin/Cummings Publishing Company, Inc., 1994

[Bopearachchi, 02] *Structuration Des Connaissances Par La Méthode HBDS "HyperGraph Based Data Structure",* Ashinsa Bopearachchi, http://membres.lycos.fr/coursderecreation/HBDS_BASE/

[Bouillé, i] *Fuzziness Structuring and Processing in an Object-Oriented GIS,* François Bouillé http://www.sbg.ac.at/geo/agit/papers94/bouille.htm

[Burback, 98] *The Booch Methodology,* Ronald LeRoi Burback, http://www-db.stanford.edu/~burback/watersluice/node55.html, 1998

[Chen, 75] *The Entity-Relationship Model: Toward a Unified View of Data,* Peter P. Chen, Proc. of the Int. Conf. on Very Large Data Bases, Sept. 22-24, 1975, Framingham, Massachusetts, USA

[Chervany et al., 98] *CASE tools: understanding the reasons for non-use*, Norman Chervany, Diane Lending, ACM SIG Computer Personnel, Volume 19, Issue 2, Abril de 1998

[Chiorean, 05] *OCLE 2.0 User Manual*, Dan Chiorean, http://lci.cs.ubbcluj.ro/ocle/, 2005

[CodePlex, 10] *Managed Extensibility Framework Documentation*, CodePlex, http://mef.codeplex.com/wikipage?title=Overview&referringTitle=Documentation , 2010

[CommentCaMarche, 08] *MERISE - Initiation à la conception de systèmes d'information,* CommentCaMarche.net, http://www.commentcamarche.net/merise/concintro.php3

[Costa, 95] Dissertação de Licenciatura em Matemáticas Aplicadas – Ramo de Informática, Marco Costa, Universidade Lusíada de Lisboa, 1995

[Costa et al., 03a] *Arquitecturas de Sistemas de Informação e a Iniciativa MDA*, Marco Costa, Alberto Rodrigues da Silva, Anais Científicos da Universidade Independente, Lisboa, 2003

[Costa et al., 05] *Aspectos de Sincronização em Modelos UML,* Marco Costa, Alberto Rodrigues da Silva*,* Actas da 6ª Conferência da Associação Portuguesa de Sistemas de Informação (CAPSI'2005), Bragança, 2005

[Costa et al., 07] *RT-MDD Framework – A Practical Approach,* Marco Costa, Alberto Rodrigues da Silva, 3rd ECMDA Traceability Workshop, Haifa, Israel, 2007

[Costa et al., 07] *Synchronization Issues in UML Models,* Marco Costa, Alberto Rodrigues da Silva, 9th International Conference on Enterprise Information Systems, Funchal, Portugal, 2007

[Cusumano, 2010] *Technology Strategy and Management – The Evolution of Platform Thinking,* Michael Cusumano, Communications of the ACM, Vol. 53, Nr.1, Janeiro de 2010

[Czarnecki et al., 03] *Classification of Model Transformation Approaches,* Krsysztof Czarnecki, Simon Helsen, OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003

[Dahl et al, 66] *SIMULA: an ALGOL-Based Simulation Language,*Ole-Johan Dahl, Kristen Nygaard, Communications of the ACM, Vol.9-9, ISSN:0001-0782, Setembro de 1966

[Dahl et al., i] *How Object-Oriented Programming Started,* Ole-Johan Dahl, Kristen Nygaard http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html

[Davis, 98] *Mapping between CDIF and EXPRESS for a Study Period on Mapping Modelling Languages for Analysis & Design,* Hugh Davis, OOPSLA'98 Workshop on Model Engineering, Methods and Tools Integration with CDIF, Vancouver, 1998

[DeMarco et al., 79] *Structured Analysis and System Specification,* Tom DeMarco, P. J. Plauger Prentice Hall, ISBN 0138543801, 1979

[Dershem et al, 95] *Programming Languages, Structures and Models,* Herbert Dershem, Michael Jipping, PWS Publishing Co., 1995

[Dobing et al., 06] *How UML Is Used,* Brian Dobing, Jeffrey Parsons, Communications of the ACM, Vol. 49 - Nr. 5, Maio de 2006

[Dollard, 04] *Code Generation in Microsoft .NET,*Kathleen Dollard, Apress, 2004

[Forte et al., 91] *CASE Outlook: Guide to Products and Services*, G. Forte and K. McCully, CASE Consulting Group, 1991

[Gamma et al., 94] *Design Patterns: Elements of Reusable Object-Oriented Software,* Erich Gamma, Richar Helm, Ralph Johnson, John Vlissides, Addison-Wesley Professional Computing Series, 1994

[Gehani, 89] *Ada: An Advanced Introduction,* Narain Gehani, Prentice Hall; 2nd edition, ISBN: 0130043346, 1989

[Giesenow, 08] *How do I Create And Use T4* Templates?, Hilton Giesenow, Microsoft Corporation, http://msdn.microsoft.com/en-us/vsx/cc308634.aspx, Março de 2008

[Gogolla et al., 07] USE: A UML-based specification environment for validating UML and OCL, Martin Gogolla, Fabian Büttner e Mark Richters, Science of Computer Programming, 69, Dezembro de 2007

[Harel, 87] *Statecharts: A visual formalism for complex systems,*David Harel, Science of Computer Programming, 8(3):231--274, Junho de 1987

[Herrington, 2003] *Code Generation In Action,* Jack Herrington, Manning Pub. Co, 2003

[Hilliard, 07] All About IEEE Std 1471, David Hilliard, http://www.iso-architecture.org/ieee-1471/docs/all-about-ieee-1471.pdf, Junho de 2007

[Höchmuller et al., 01] *CASE Tools – A Framework for Assessment and Evaluation,* E. Höchmuller, R.T. Mittermeir, E. Kofler, H. Steinberger, http://citeseer.ist.psu.edu/472173.html, 2001

[Hnatkowska et al., 04] *On understanding of refinement relationship,* Bogumila Hnatkowska, Zbigniew Huzar e Lech Tuzinkiewicz, UML2004 Seventh International Conference on UML Modeling Languages and Applications, Lisbon, Portugal, Outubro de 2004

[Hutchings, i] *Introduction to Methodologies and SSADM,* Tim Hutchings, http:// www.comp.glam.ac.uk/pages/staff/tdhutchings/chapter4.html

[Hussman et al., 00] *Modular Architecture for a Toolset Supporting OCL*, Hussman, Demuth e Finger, Proceedings of UML 2000, Springer Verlag, 2000

[IEEE, 92] IEEE Std 1209-1992 IEEE Recommended Practice for the Evaluation and Selection of CASE Tools, IEEE, http://standards.ieee.org, 1992

[Iivari, 96] *Why Are CASE Tools Not Used,* Juhani Iivari, Communications of the ACM, Vol.39, N.10, Págs. 94-103, Outubro de 1996

[ISO/IEC, 95] *Information technology – Guideline for the evaluation and selection of CASE tools* (ISO/IEC 14102:1995), ISO/IEC Technical Committee JTC 1/SC 7, ISO Standards, 1995

[Jackson, 75] *Principles of Program Design,* Michael Jackson, Academic Press, 1975

[Jacobson, 92] *Object-Oriented Software Engineering,* Ivar Jacobson, Magnus Christerson, Patrik Johnsson e Gunnar Övergaard, Addison Wesley, ISBN 0201544350, 1992

[Jensen, 04] *Borland Delphi 2005 Reviewer's Guide,* Cary Jensen, Jensen Data Systems, 2004

[Lenvedovszky et al., 04] *A Systematic Approach To Metamodeling Environments and Model Trasnformation Systems in VMTS,* Tihámer Lenvedovszky, László Lengyel, Gergely Mezei, Hassau Charaf, International Workshop on Graph-Based Tools (GraBaTs'04) Roma, Itália, October 2 , 2004

[KBSI, i] *IDEF Family of Methods*, Knowledge Based Systems, Inc., http://www.idef.com/default.html

[KCSL, i] *KCSL Jackson Workbench,* http://www.jacksonworkbench.co.uk/

[Kim et al., 02] *The complementary use of IDEF and UML modelling approaches,* Cheol-Han Kim, R. H. Weston, A. Hodgsonb and Kyung-Huy Lee, Elsevier Science, 2002

[King, 88] Creating Effective Software: Computer Program Design Using the Jackson Methodology, David King, Yourdon, 1988

[Kline et al., 02] *Empirical Study of Software Developers' Experiences,* R. Kline e A. Seffah, http://citeseer.ist.psu.edu/544831.html, 2002

[Laddad, 03] AspectJ in Action: Practical Aspect-Oriented Programming, Ramnivas Laddad, Manning Publications, 2003

[Le Blanc et al., 92] *A Structured Approach to the Evaluation and Selection of CASE Tools,* Louis A. Le Blanc e Willard M. Korn, Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: technological challenges of the 1990's, Kansas City, EUA, Págs. 1064-1069, 1992

[Lemesle, 98] *Meta-modeling and Modularity: Comparison Between MOF, CDIF & sNets Formalisms,* Richard Lemesle, OOPSLA'98 Workshop on Model Engineering, Methods and Tools Integration with CDIF, Vancouver, 1998

[Lundell et al., 02] *Comments on ISO 14102: the standard for CASE-tool evaluation,* Bjorn Lundell e Brian Lings, Computer Standards & Interfaces, 24, Págs 381-388, Ed. Elsevier Science, 28 de Maio de 2002

[Martin, 89a] *Information Engineering: Introduction,* James Martin, Prentice Hall

[Martin, 89b] Information Engineering Book II: Planning and Analysis, James Martin, Prentice Hall, 1989

[McNaughton, 02] *We Compare Microsoft Visio with Rational XDE for Modeling Your .NET Applications,* Allan McNaughton, http://www.devx.com/enterprise/Article/9749, 2002

[Microsoft, 08] *Microsoft Visual Studio SDK 1.1,* Microsoft Corporation, http://www.microsoft.com/ downloads/details.aspx?FamilyId=59EC6EC3-4273-48A3-BA25-DC925A45584D&displaylang= en, 2008

[Microsoft, 10] *Compiling to MSIL*, in .NET Framework Developer's Guide, Microsoft Corporation, http://msdn.microsoft.com/en-us/library/c5tkafs1(VS.71).aspx, 2010

[Model Systems, 02] *Model Systems and Structured Systems Analysis and Design Method,* Model Systems, http://www.modelsys.com/msssadm.htm, 2002

[Monteiro, 89] *Álgebra Linear e Geometria Analítica,* António Antunes Monteiro, Ed. Assoc. dos Estudantes da Fac. de Ciências de Lisboa, 1989

[Mubin, i] *Constructing MDA-based Application Using Rational XDE for .NET,* Ashirul Mubin, http://cs.ua.edu/630/Notes/1

[Nosek, et al., 92] *Ease of learning and using a CASE software tool: an empirical evaluation*. SIGCPR Comput. Pers., Vol.14, Issue 1-2, ACM, Págs. 60-65, Novembro de 1992

[O'Regan, 04] *Introduction to Aspect-Oriented Programming*, Graham O'Regan, O'Reilly OnJava.com, http://onjava.com/pub/a/onjava/2004/01/14/aop.html, 2004

[OMG] Object Management Group, http://www.omg.org

[OMG, 02] *Meta Object Facility(MOF) Specification,* Object Management Group, http://www.omg.org/docs/formal/02-04-03.pdf, Vers. 1.4, Abril de 2002

[OMG, 02a] OMG-XML Metadata Interchange (XMI) Specification, V. 1.2, http://www.omg.org/cgi-bin/doc?formal/2002-01-01, 1 de Janiro de 2002

[OMG, 03] *MDA Guide Version 1.0.1,* Object Management Group, http://www.omg.org/docs/omg/03-06-01.pdf, Vers.1.0.1, 2003

[OMG, 03a] *OMG-Unified Modeling Language – Semantics,* Object Management Group, http://www.omg.org/docs/formal/03-03-09.pdf, Vers. 1.5, Março de 2003

[OMG, 03b] *UML 2.0 OCL Final Adopted specification*, Object Management Group, http://www.omg.org/docs/ptc/03-10-14.pdf, 14 de Outubro de 2003

[OMG, 05] *Unified Modeling Language: Diagram Interchange – Version 2.0,* Object Management Group, http://www.omg.org/docs/ptc/05-06-04.pdf, 4 de Junho de 2005

[OMG, 05a] *MOF QVT Final Adopted Specification,* Object Management Group, http://www.omg.org/cgi-bin/apps/doc?ptc/05-11-01.pdf, 1 de Novembro de 2005

[OMG, 06] *Meta Object Facility (MOF) Core Specification, version 2.1.1*, Object Management Group, http://www.omg.org/cgi-bin/doc?formal/06-01-01, 1 de Janeiro de 2006

[OMG, 07] *Unified Modeling Language (UML), version 2.1.1*, Object Management Group, http://www.omg.org/cgi-bin/doc?formal/07-02-05, 5 de Fevereiro de 2007

[Paige, 07] *Metamodel-Based Model Conformance and Multiview Consistency Checking,* Richard F. Paige, ACM Transactions on Software Engineering and Methodology, Vol. 16, Nº 3, Article 11, Julho de 2007

[Patterson, 01] *Investigating XSLT: The XML transformation language,* Linda May Patterson, IBM Developer Toolbox Technical Magazine, 1 de Agosto de 2001

[Perini et al., 05] *Automating Model Transformations in Agent-Oriented Modelling,* Anna Perini, Angelo Susi, Proceedings of 6th International Workshop AOSE 2005, Utrecht, NL, 25-26 de Julho de 2005

[Porto, 08] *Dicionário da Língua Portuguesa,* Porto Editora, ISBN-13: 978-972-0-01343-9, 2008

[QVTP, 03] *Revised submission for MOF 2.0 Query / Views / Transformations RFP,* QVT-Partners, Vers. 1.1, http://qvtp.org, 18 de Agosto de 2003

[Ramsin et al., 08] *Process-Centered Review of Object Oriented Software Development Methodologies,* Raman Ramsin, Richard F. Paige, ACM Computing Surveys 40, 1, Article 3, Fevereiro de 2008

[Richters et al., 00] *Validating UML models and OCL constraints,* Mark Richters e Martin Gogolla, Proceedings of Unified Modelling Language (UML'00), Springer Verlag, 2000

[Robinson et al, 94] *Object-oriented SSADM,* Keith Robinson, Graham Berrisford, Prentice Hall, 1994

[Rose et al, 08] *The Epsilon Generation Language,* Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos and Fiona A.C. Polack, ECMDA'08, Berlin, 2008

[Rumbaugh et al., 90] *Object-Oriented Modeling and Design,* James Rumbaugh, Michael Blaha, William Lorensen, Frederick Eddy, W. Premerlani, Prentice Hall, ISBN 0136298419, 1990

[SCE, i] *James Martin,* Smart Computing Encyclopedia, http://www.smartcomputing.com

[Schmidt, 06] *Model-Driven Engineering*, Douglas Schmidt, Computer, IEEE, Fevereiro de 2006

[SEI, 95] *The Capability Maturity Model: Guidelines for Improving the Software Process* Software Engineering Inst. Carnegie Mellon Univ., Addison-Wesley Professional, ISBN: 0201546647, 1995

[SEI, 05] *What is a CASE Environment,* Carnegie Mellon Software Engineering Institute, http://www.sei.cmu.edu/legacy/case/case_whatis.html, 27 de Julho de 2005

[Silva et al., 03] *XIS Generative Programming Techniques,* Alberto Rodrigues da Silva, Gonçalo Lemos, Tiago Matias, Marco Costa, Generative Programming and Component Engineering (GPCE'03), Erfurt, Germany, 2003

[Silva et al., 06] *gestBarragens: Especificação de Requisitos Técnicos, v2.1*, Coordenadores: Alberto Silva, Helena Galhardas, Eliane Portela; Autores: Alberto Silva, Helena Galhardas, José Barateiro, Hugo Matos, Jorge Gonçalves, Tiago Martins, Helder Soares, Marco Custódio, Novembro de 2006

[Silva et al., 07a] *Manual de Utilizador do "GESTBARRAGENS" Volume 1 – Aspectos Gerais e de Suporte, v3.0*, Coordenadores: Alberto Silva, Helena Galhardas, Eliane Portela; Colaboradores: José Barateiro, Hugo Matos, Jorge Gonçalves, Fevereiro de 2007

[Silva et al., 07b] *Manual de Utilizador do "GESTBARRAGENS" Volume 2 – Observações, v3.0*, Coordenadores: Alberto Silva, Helena Galhardas, Eliane Portela; Colaboradores: José Barateiro, Hugo Matos, Jorge Gonçalves, Fevereiro de 2007

[Silva et al., 07c] *Manual de Utilizador do "GESTBARRAGENS" Volume 3 – Outros Módulos e Aplicações, v3.0*, Coordenadores: Alberto Silva, Helena Galhardas, Eliane Portela; Colaboradores: José Barateiro, Hugo Matos, Jorge Gonçalves, Tiago Martins, Helder Soares, Marco Custódio, Fevereiro de 2007

[Silva et al., 08] gestBarragens - Sistema Integrado de Gestão da Informação para o Controlo de Segurança de Barragens, LNEC - Especificação Técnica do Sistema, Alberto Rodrigues da Silva, Marco Costa, Luis de Sousa, SIQuant - 01/2008

[Schenck et al, 94] *Information Modeling: The Express Way*, Douglas Schenck, Peter Wilson, Oxford University Press, ISBN: 0195087143, 1994

[Sharp et al, 03] *Microsoft Visual C# .NET Step By Step,* John Sharp, Jon Jagger, Microsoft Press, 2003

[Sparx, 10] *Enterprise Architect*, Sparx Systems, http://www.sparxsystems.com.au, 2010

[Stento et al, 03] *White Paper: Persistent Data Development Tools Validate the Model Driven Architecture Approach*, Ruth Stento, ObjectStore, http://www.omg.org/registration/ registration-ruth-stento-wp.htm, 2003

[Tardieu et al., 83] *La méthode Merise - Tome 1 Principes et outils*, Hubert Tardieu, Arnold Rochfeld, René Colletti, Editions d'organisation, Paris, 1983

[Texto, 07] *Dicionário da Língua Portuguesa*, Texto Editores, 2007

[TI, 87] *Texas Instruments Announces Commercial Availability of Integrated "CASE" Tools to Automate Information Systems Life Cycle,* Texas Instruments, http://www.ti.com/corp/docs/ company/history/case.shtml, 1987

[Titan, i] *SRC & Information Enginnering,* Titan Systems Corporation, http://www.srcorp.com/ factsheets/info_eng.pdf

[Tratt, 03] *QVT: The High Level Scope*, Lawrence Tratt, http://qvtp.org/, 30 de Maio de 2003

[UMT-QVT, 05] *UML Model Transformation Tool*, http://umt-qvt.sourceforge.net/, 2005

[Vanhoof et al, 05] *Supporting Modular Transformation Units with Precise Transformation Traceability Metadata*, Bert Vanhoof, Yolande Berbers, ECMDA Traceability Workshop, 7-10 November 2005, Nurenberg, Germany, SINTEF, pp. 15-27, 2005

[Varro, 02] *Automated Program Generation for and by Model Transformation Systems,* Daniel Varro, Proc. AGT 2002: Workshop on Applied Graph Transformation, 2002

[Voelter, 03] *A Catalog of Patterns for Program Generation*, Markus Voelter, 2003-04, V.1.6 http://www.voelter.de/publications/patterns.html

[Yourdon et al, 78] *Structured Design,* Ed Yourdon, Larry Constantine, Prentice Hall, 1978

[Yourdon, 10] *Just Enough Structured Analysis and System Specification,* Ed Yourdon http://yourdon.com/strucanalysis/wiki/index.php?title=Introduction, 2010

[W3C, 04] *Extensible Markup Language (XML) 1.0 (Third Edition),* World Wide Web Consortium Recommendation, http://www.w3.org/TR/2004/REC-xml-20040204/, 4 de Fevereiro de 2004

[Welsh, 03] *How Software Modeling Tools Are Being Used,* Tom Welsh, Cutter Consortium, Enterprise Architecture Advisory Service Executive Update Vol. 6 , N. 9, http://www.cutter.com/research/2003/edge031230.html, Dezembro de 2003

[Wikipedia, 08] *Traceability,* Wikipedia, http://en.wikipedia.org/wiki/Traceability

[Woolridge, 04] *An Introduction to Use Case Analysis,* Richard Woolridge, http://www.cbd-hq.com/articles/1999/991115rw_caseanalysis.asp, 2004

[Zettel, 05] *Methodology Support in CASE Tools and Its Impact on Individual Acceptance and Use: A Controlled Experiment*, Jörg Zettel, Empirical Software Engineering, Kluwer Pub., Vol 10, Issue 3, ISSN:1382-3256, Julho de 2005

Anexo A – Ferramentas de apoio ao desenvolvimento de sistemas de informação

Sendo o âmbito do trabalho a rastreabilidade entre artefactos produzidos por ferramentas de desenvolvimento e análise de sistemas de informação, listam-se em seguida diversas ferramentas CASE que utilizam o UML e as suas características mais relevantes para este trabalho:

- ArgoUML, Tigris.org: Aplicação *open source* codificada na linguagem Java. Gera código Java bem como noutras linguagens através de módulos adicionados. A geração é apenas no sentido modelo para código. Permite desenhar diagramas de casos de utilização, diagramas de classes, diagramas de sequência, diagramas de colaboração, diagramas de máquinas de estados, diagramas de instalação e diagramas de actividades. Inclui a capacidade de verificar os modelos criados através de um conjunto de regras pré-definidas (denominadas *críticas*) que podem ser activadas ou desactivadas.

- Artiso Visual Case, AMD: Ferramenta CASE centrada no desenho de diagramas UML 1.x. Permite a geração bidireccional de esqueletos de classes em Java e de tabelas SQL. Possui também um editor de SQL que permite preencher o comportamento dos métodos com *queries* SQL e *stored procedures* específicas a alguns SGBDs (e.g., Microsoft Access, Microsoft SQL Server, Oracle, MySQL, Sybase, PostgreSQL, Pervasive 8, Interbase).

- BOUML, Free Software Foundation: Ferramenta CASE não comercial que tem por objectivo a especificação e a geração de código em C++, Java e Idl. Funciona não só na plataforma Microsoft Windows como em Unix/Linux/Solaris. A geração de código é feita recorrendo a ficheiros *templates* externos à aplicação que podem ser editados. Suporta os principais diagramas do UML 2.0 e devido à sua simplicidade, por enquanto, é uma aplicação com um óptimo desempenho quando comparada com outras ferramentas comerciais.

- Enterprise Architect, Sparx Systems: Ferramenta CASE que permite o desenho completo dos treze diagramas UML 2.0. Inclui uma maneira de realizar diversas transformações de modelos (designadas *transformadas*) através de *templates* criados numa linguagem própria. A ferramenta não elimina ou modifica elementos que não tenham sido originalmente gerados permitindo acrescentar e manter

com segurança código não gerado. A ferramenta presume que uma alteração ao modelo é sempre feita no nível conceptual mais elevado sendo feita a sua propagação ao(s) nível(is) mais baixo(s). Existem diversas transformações pré-definidas (e.g., DDL, EJB Entity, EJB Session, Java, XSD e C#) que limitam-se a transformar alguns aspectos estruturais dos modelos criados. Permite também a geração e a edição de documentação em formato RTF recorrendo a uma editor gráfico.

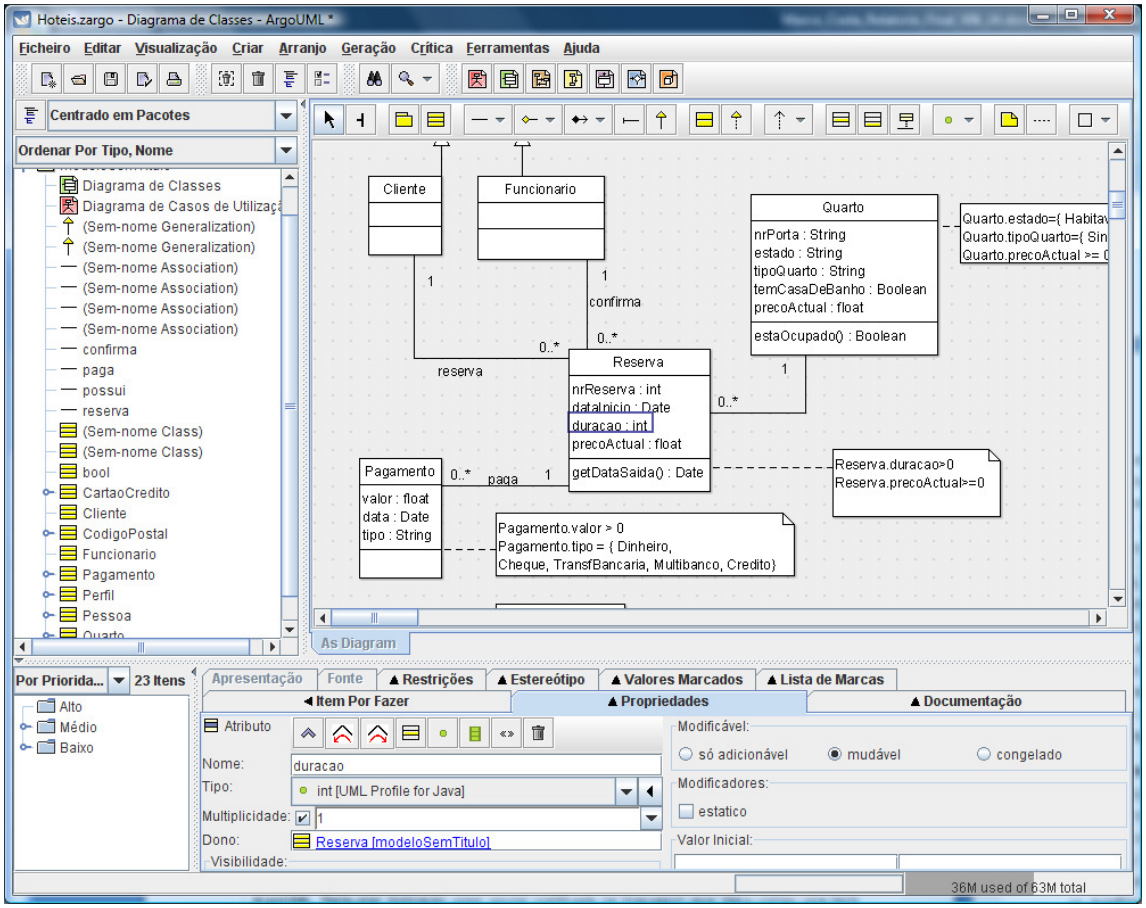


Figura A.1 – Ferramenta CASE Argo UML

- **OptimalJ, Compuware:** Como o nome indica é uma ferramenta dirigida para a criação de código Java através de modelos, neste caso UML. Os criados pelo utilizador estão descritos em três metamodelos: Modelo de Domínio (PIM), Modelo da Aplicação (PSM) e Modelo do Código, sendo o primeiro o de maior nível de abstração e o último o de menor. O Modelo de Domínio é transformado no Modelo da Aplicação que define a arquitectura da aplicação baseada numa certa tecnologia. Posteriormente o Modelo Aplicacional é transformado no Modelo do Código. Os metamodelos podem ser criados, modificados ou estendidos usando um editor gráfico. Esta ferramenta é uma das implementações mais completas do MDA existentes no mercado.

- **Nucleus BridgePoint, Accelerated Technology:** Ferramenta CASE específica para sistemas de tempo real. Começa por se definir um PIM que em seguida é marcado com detalhes de implementação. Este modelo serve então para a geração de código C ou C++. Utiliza um perfil do UML denominado xtUML. Inclui uma *action language* que permite uma geração de 100% do código do programa assim como o *debugging* das aplicações geradas. Como o contexto é reduzido (aplicações de tempo real), os

modelos, juntamente com a *action language,* traduzem todos os aspectos do problema assim como os detalhes de implementação. A aplicação utiliza apenas diagramas de classes UML, diagramas de actividades e o código da referida linguagem ignorando todos os outros diagramas UML.

- Power Designer, Sybase: É um conjunto de ferramentas que combina modelação de aplicações por UML, modelação de processos de negócio e modelação de bases de dados. O Power Designer é produzido pelo fabricante de um dos mais bem difundidos sistemas de gestão de bases de dados relacionais (SGBDR). Este facto condicionou de certa forma o desenvolvimento das ferramentas de modelação tornando-as centradas numa escolha tecnológica que posteriormente se alargou a outros 44 SGBDR. Permite a geração de esquemas SQL e esqueletos de classes em diversas linguagens de programação. Além de outras capacidades permite a geração de relatórios altamente parametrizáveis recorrendo ao repositório central de informação.

- Rational Rose Suite, IBM: Conjunto de ferramentas CASE dirigidas para diferentes tipos de utilização que incluem modelação específica para plataformas (e.g., Developer for UNIX, XDE Developer for Java) e modelação de PIMs com o Rational Software Architect. O facto de ser um conjunto de 12 ferramentas, em vez de uma única, aumenta a complexidade da sua utilização. Permite desenhar todos os diagramas UML, embora apenas utilize os diagramas de classes para a geração de código. Suporta geração *round trip* para diferentes linguagens sendo esta sempre provocada pelo utilizador.

- Together, Borland: É uma família de ferramentas CASE com capacidades específicas para diferentes plataformas e perfis de utilização. O Together Architect está dirigido para o arquitecto de sistemas de informação, com uma visão geral sobre as diferentes arquitecturas, ambientes e linguagens presentes na organização. O Together Designer é a ferramenta utilizada para a construção dos diagramas UML necessários à análise do sistema de informação. O Together Developer é uma ferramenta de modelação centrada no código. Esta aplicação possui a característica única de existir uma sincronização simultânea entre o código e as classes UML definidas. A geração de código Java dá-se a partir dos Diagramas de Sequência e dos Diagramas de Classes. Está disponível a versão do Together Developer para o IDE Eclipse e foi anunciada a existência das versões para Borland JBuilder e para Microsoft Visual Studio .NET (no último caso gerando código C#). Os domínios do Together Architect e do Together Designer sobrepõe-se cabendo ao utilizador entender qual a sua motivação e escolher a ferramenta correcta para o seu caso.

- UModel 2005, Altova: Ferramenta CASE com capacidade de geração bidireccional de código Java a partir de modelos UML 2.0. A geração é realizada nos dois sentidos de forma explícita sendo que as opções de sincronização têm de ser tomadas novamente, cada vez que se faz uma geração, directa ou inversa.

- Visual Paradigm for UML, Visual Paradigm: É uma ferramenta CASE dirigida para o desenho de diagramas UML e a geração de código Java. Suporta geração bidireccional de código podendo ser integrada com outras ferramentas (IDEs) que suportam a fase de implementação do processo de desenvolvimento (e.g., Eclipse/IBM WebSphere, Borland JBuilder, NetBeans IDE/Sun One Studio, Oracle JDeveloper, BEA WebLogic Workshop). Esta ferramenta consegue desenhar todos os diagramas do UML 2.0.

Durante o desenvolvimento de aplicações, muitos dos artefactos são produzidos por IDEs que se têm tornado ao longo dos anos, cada vez mais completos naquilo que diz respeito à relação entre os artefactos produzidos. E.g., é possível encontrar verificações de integridade do código, ainda que algo rudimentares, como no caso do Visual Studio, quando é verificada a data do ficheiro fonte de uma secção de código na fase de edição.

- Delphi, Borland: Em 1983 a Borland criou um IDE para a criação de programas na linguagem Pascal, denominado então de Turbo Pascal [Jensen, 2004]. Era composto de um editor em modo texto, com reduzidas capacidades gráficas, executado no sistema operativo MSDOS e de um compilador de Pascal. O Turbo Pascal teve uma evolução continuada nos anos seguintes tendo-se tornado no ambiente de programação mais utilizado, para esta linguagem. Foram sendo acrescentadas capacidades ao longo dos anos, hoje presentes na quase totalidade dos IDEs, como o texto colorido consoante a sintaxe, um modelo de programação orientada por objectos e um editor com capacidades gráficas acrescidas. Em Fevereiro de 1995 a Borland lançou o Delphi consistindo numa evolução suficientemente grande do produto anterior para poder ser considerado um novo produto. Seguindo com uma evolução da linguagem Pascal, este ambiente de programação permitia desenvolver aplicações baseadas em componentes, através de uma interface gráfica mais intuitiva levando a uma maior produtividade. A versão Delphi 2005 acrescenta a capacidade de se programar não só na linguagem Delphi (evolução do Pascal) como em C#, HTML e SQL, convergindo com outros IDEs no suporte a diferentes linguagens dentro do mesmo ambiente de programação. A ferramenta permite já o desenvolvimento cooperativo de aplicações através da componente de gestão de versões e perfis e um optimizador de código com criação de testes para a plataforma. NET.

- Eclipse/IBM WebSphere: O projecto não comercial Eclipse foi criado tendo como principal patrocinador a IBM. O objectivo deste projecto é criar um IDE genérico que possa utilizar diversos compiladores (da mesma linguagem ou de diferentes linguagens) e editores integrados. Através de módulos que são acrescentados ao ambiente base, o programador adiciona novas capacidades (e.g., compilador de C#, editor de classes UML, interpretador de OCL). Esta ferramenta tem uma grande quantidade de módulos disponíveis sendo a sua maior parte para a linguagem Java. Na verdade a quase totalidade dos programadores que utilizam este IDE programam em Java. A IBM utiliza muito do código desenvolvido no Eclipse na aplicação comercial IBM WebSphere que garante maior estabilidade na sua execução devido a uma verificação acrescida do código utilizado.

- Netbeans/Sun One Studio: O Netbeans é comparável ao Eclipse no sentido em que também é um projecto não comercial, suportado por uma empresa, neste caso a Sun Microsystems. No entanto, ao contrário do anterior, este IDE foi especificamente concebido para desenvolver código na linguagem Java. Além da versão não comercial existe também uma versão comercializada pela Sun Microsystems, denominada Sun One Studio, que permite uma utilização mais simples de características complexas (e.g., utilização de bases de dados, computação distribuída, ligação a servidores aplicacionais).

- Visual Studio .NET, Microsoft: Este IDE é uma ferramenta de apoio ao desenvolvimento de aplicações na plataforma .NET. A ferramenta pode ser utilizada em qualquer linguagem de programação que tenha um compilador .NET. Existem já mais de vinte linguagens com compiladores disponíveis para esta plataforma sendo as mais utilizadas o C#, o VB.NET e o C++. Todos os

compiladores desta plataforma geram código numa linguagem intermédia (IL, *Intermediate Language)* que é lida e compilada antes da execução em qualquer sistema operativo em que se tenha instalado a *framework* .NET. O Visual Studio .NET pode ser estendido tanto quanto às linguagens usadas, através da inclusão de outros compiladores, como de módulos que podem desempenhar outras funções no desenvolvimento. A partir da versão 2005 a ferramenta passou a integrar um diagrama de classes sincronizado com o código. Para além de *plugins* que podem ser acrescentados à ferramenta, existem ainda outros mecanismos de adicionar funcionalidades. Um destes mecanismos (DSL Designer) foi usado no desenvolvimento do protótipo ReacT-Workbench para permitir a construção de uma linguagem visual de definição de metamodelos. Foi ainda usado outro mecanismo associado à linguagem T4 para a criação de *templates* de geração automática de código.

São ainda relevantes para este trabalho outras aplicações que envolvem a geração de código sem no entanto se enquadrarem nas duas categorias de aplicações anteriores:

- AndroMDA: Este projecto não comercial teve por objectivo inicial criar uma ferramenta, utilizada por programadores da linguagem Java, que facilitasse algumas tarefas rotineiras de escrita de código na plataforma J2EE, a partir de modelos descritos em ficheiros XMI. Posteriormente a ferramenta evoluiu abarcando a geração de código para outras plataformas. No entanto, pelo menos por enquanto, ainda continua a ser utilizada quase exclusivamente para a geração de componentes J2EE. O AndroMDA não pretende ter qualquer espécie de sincronização entre os modelos e o código gerado, limita-se apenas a facilitar essa geração.

- Codagen, Codagen: Esta aplicação pode ser integrada num CASE (e.g., IBM Rational Rose) para permitir capacidades acrescidas de geração de código. O principal interesse da ferramenta é o de permitir a geração dos modelos PSM (modelos específicos à plataforma) a partir de modelos PIM (modelos independentes da plataforma). Segundo a abordagem Codagen todas as decisões tomadas ao nível do PIM são mantidas no PSM tal como estas são também mantidas na geração de código posterior. Esta abordagem pressupõe a utilização de um processo de desenvolvimento conhecido como *em cascata*, i.e., caso uma decisão patente no PSM seja contestada por um programador é necessário primeiro verificar o PIM, gerar o(s) PSM(s) e o código associado, quando necessário, sendo apenas nesse momento seguro alterar o código. Sem que este cuidado seja tomado corre-se o risco de as alterações serem eliminadas por uma geração de código posterior. Os modelos PSM incluem pontos de extensibilidade ao nível do código que permitem ao programador codificar a parte do problema que não foi coberta pelo PSM, sem correr o risco de perder essas alterações posteriormente. Esta ferramenta realiza os dois tipos de geração (PIM para PSM e PSM para código) acrescentando metainformação aos diagramas do nível mais elevado. Pode até acontecer que a informação que é acrescentada ao PIM possa pertencer ao PSM, o que contraria a própria definição dos dois modelos.

- Documentator, Rippen.de: Esta aplicação produz a documentação de um projecto de sistemas de informação, através de técnicas de geração automática, a partir de um modelo construído na aplicação IBM Rational Rose. Os elementos gráficos da documentação produzida (e.g., tipos de letra, formatação de tabelas, diagramas) podem ser alterados através de *templates*.

Produto	Fabricante	Tipo	Referência Web
		Ferramentas CASE	
ArgoUML	Tigris.org	Ferramenta CASE *open source* com geração de código Java (ou outras linguagens recorrendo a extensões) a partir de um subconjunto dos diagramas UML.	argouml.tigris.org
Artiso Visual Case	AMD	Ferramenta CASE com geração bidireccional de código Java centrada em aplicações com acesso a SGBDs relacionais.	www.visualcase.com
Enterprise Architect	Sparx Systems	Ferramenta CASE com geração para diversas linguagens de programação e transformação de modelos através de templates.	www.sparxsystems.com.au
Nucleus BridgePoint	Accelerated Technology	Ferramenta CASE com geração automática de programas a partir de perfil UML e action language	argouml.tigris.org
OptimalJ	Compuware	Ferramenta CASE com capacidade de transformação entre modelos tal como entre modelos e código utilizando um metamodelo.	www.compuware.com/products/optimalj/
Power Designer	Sybase	Conjunto de ferramentas CASE orientadas para o desenvolvimento de aplicações centradas em dados implementadas em diversas linguagens e SGBDRs.	www.sybase.com/products/development integration/powerdesigner
Rational Rose Suite	IBM	Conjunto de ferramentas CASE cada uma com um tipo diferente de utilização suportando geração round trip para diferentes linguagens.	www-306.ibm.com/software/rational/sw-bycategory/subcategory/SW710.html
Together	Borland	Conjunto de ferramentas CASE cada uma com um tipo diferente de utilização suportando sincronização simultânea entre o modelo e código C# e Java.	

UModel 2005	Altova	Ferramenta CASE com geração bidireccional de código Java	www.altova.com/products_umodel.html
Visual Paradigm for UML	Visual Paradigm	Ferramenta CASE com geração bidireccional de código Java.	www.visual-paradigm.com
		IDEs	
Delphi	Borland	IDE para Delphi (dialecto do Pascal) e C# permitindo a geração de código SQL e a sincronização do modelo da base de dados com o modelo de dados da aplicação.	
Eclipse / IBM WebSphere	Não comercial/ IBM	IDE genérico para diferentes linguagens com capacidade de inclusão de diferentes módulos, consoante as necessidades dos programadores.	www.eclipse.org
Netbeans / Sun One Studio - Microsystems	Não comercial/ Sun Microsystems	IDE específico para a linguagem Java, extensível através de módulos que podem ser acrescentados para realizarem diferentes tipos de acções.	www.netbeans.org
Visual Studio .NET	Microsoft	IDE que suporta diferentes linguagens, específico para o desenvolvimento de aplicações na plataforma .NET, extensível com a inclusão de outros módulos.	msdn.microsoft.com/vstudio/
		Outras Ferramentas	
Codagen	Codagen	Conjunto de módulos aplicacionais integrados numa ferramenta CASE que acrescentam capacidades de geração automática de código.	www.codagen.com
AndroMDA	Não comercial	Framework para a geração de código para a plataforma J2EE a partir de modelos especificados na linguagem XMI (ou convertidos para esta).	www.andromda.org
Documentator	Rippen.de	Gerador de documentação para a ferramenta Rational Rose.	www.rippen.de

Anexo B – Avaliação de ferramentas CASE

Existem algumas propostas quanto à comparação e avaliação de ferramentas CASE. Para Ordén e Boive [Ordén et al., 01] a comparação pode ser feita quanto à adequação da ferramenta ao desenho de todos os elementos da linguagem gráfica, neste caso o UML. Estes autores partiram de um conjunto de ferramentas conhecidas e consideraram um conjunto de diagramas. Tentaram então desenhá-los em cada uma das ferramentas, comparando o resultado gráfico com aquilo que seria esperado. Cada aspecto de um diagrama que não podia ser desenhado foi anotado contribuindo para uma nota negativa da ferramenta. Este método é menos objectivo do que aparenta visto que o critério para se considerar uma ferramenta *melhor* do que outra resulta de um conjunto de elementos gráficos não desenhados, independentemente da sua real importância. Um elemento gráfico que não esteja presente num produto (e.g., a *classe* UML como caso extremo) pode ser mais importante para a maioria dos utilizadores do que dois outros elementos (e.g., ligações n-árias, concorrência no diagrama de sequência de mensagens).

Uma via alternativa foi proposta por Geoff Beckworth [Beckworth, 93] onde a avaliação é feita tomando-se um conjunto de objectivos que devem ser cumpridos pela ferramenta ideal. Entram neste conjunto (a) poder suportar o ciclo de desenvolvimento em parte ou na sua totalidade, (b) o uso de um repositório de informação, (c) o desenho de diagramas, (d) a existência do produto para a plataforma de hardware/software necessária, (e) ter uma arquitectura de software aberta, (f) ser compatível com software já existente, (g) suportar metodologias estruturadas, (h) poder gerar protótipos, (i) usar um ciclo de desenvolvimento bem definido. Eram ainda requisitos importantes ter custos baixos, permitirem o desenvolvimento de aplicações de tempo real e serem utilizados por pessoas com diferentes níveis de conhecimento neste âmbito (eventualmente cada aplicação pode servir um nível de utilizadores). A avaliação conta ainda com um uma matriz em que cada ferramenta é cruzada com uma característica relevante (e.g., capacidade de trabalhar em rede). Num conjunto grande de produtos esta matriz revela os produtos mais completos, no que diz respeito às características consideradas. Segundo esta avaliação os melhores produtos devem ser escolhidos combinando os objectivos e requisitos com a análise da matriz.

Em [Le Blanc et al., 92] usam-se três fases para seleccionar a melhor ferramenta CASE para uma dada organização. Inicialmente listam-se as ferramentas existentes e reduz-se essa lista para um número reduzido (idealmente dois ou três produtos) a partir de um conjunto de critérios funcionais muito específicos à metodologia adoptada pela organização. Essa selecção prévia depende de

características que as ferramentas podem ou não possuir. Consoante o número e o tipo (característica impeditiva ou não) das respostas negativas o produto é excluído. Na segunda fase parte-se da lista reduzida e refina-se os critérios de avaliação, obtém-se informação mais detalhada da ferramenta CASE e finalmente avalia-se os finalistas escolhendo a melhor alternativa. Os critérios de avaliação são desenvolvidos dentro de quatro categorias: (1) requisitos técnicos do CASE; (2) requisitos funcionais do CASE; (3) documentação e aprendizagem; e (4) informação comercial. Na terceira fase, denominada de confirmação, a equipa produz alguns produtos e verifica a reacção dos utilizadores. Os comentários dos utilizadores podem levar a uma alteração dos requisitos de avaliação e mesmo à escolha de uma nova ferramenta, existindo a sua substituição pela actual ou a sua adição às ferramentas usadas, consoante seja necessário.

Existe também a indicação de um modelo de especificação de testes para ferramentas CASE [Ahtee, 96] que contribui para a sua avaliação. Este modelo inclui: (1) Questionário sobre a ferramenta; (2) Lista de avaliação das características da ferramenta; (3) Lista de avaliação da utilização da ferramenta em testes e (4) Teste de diagramas a partir de uma dada metodologia. No entanto este modelo não parece ter tido um desenvolvimento posterior ficando numa fase embrionária de definição.

Uma das propostas mais consistentes para a avaliação de ferramentas CASE é o standard ISO14102 ([ISO/IEC, 95], [Lundell et al., 02]). O standard fornece (p. 1): (1) Uma orientação no sentido de se identificarem os requisitos organizacionais para as ferramentas CASE; (2) Orientação na avaliação da comparação entre esses requisitos e as características das ferramentas e (3) Um processo para seleccionar a ferramenta mais apropriada, de entre várias, baseado em medidas das características definidas. Na verdade o standard não é um processo de avaliação mas sim uma via para os produzir. Um processo específico pode ser visto como uma instanciação do standard, desde que cumpra as suas normas. Conforme definido no standard (p. 6), a avaliação é um processo sequencial que consiste nos seguintes subprocessos: iniciação, estruturação, avaliação e selecção. Cada um destes subprocesos está dividido em actividades, que por sua vez estão divididos em tarefas, simples e complexas (por sua vez divididas em subtarefas). O processo de iniciação serve para identificar as necessidades da organização, os seus objectivos e as suas expectativas, naquilo que diz respeito a este tipo de ferramentas. Reconhece-se assim a especificidade das necessidades de cada organização, através de factores técnicos e não técnicos. O ISO 14102 define 125 características atómicas organizadas numa hierarquia, que podem ser consideradas e incluídas num processo de avaliação específico. Uma das críticas apontadas a este standard é a definição das características ser demasiado vaga. Este ponto é fundamental se for necessário comparar avaliações realizadas por diferentes entidades. Se o significado dado às características for diferente, mesmo que ligeiramente, os dados não podem ser comparados. Infelizmente não existe consenso na indústria quanto à terminologia a adoptar o que torna este problema muito comum. Esta situação é usual numa área do conhecimento nova, como é o caso desta. Assim diferentes empresas, mesmo utilizando o standard ISO 14102, ou outro qualquer, podem chegar a resultados absolutos diferentes, pesando diferentemente as características identificadas. O próprio standard releva um outro problema inerente à natureza da avaliação. Mesmo considerando que as características estão correctamente definidas, a avaliação das mesmas é por vezes subjectiva devido ao facto delas próprias dizerem respeito a uma realidade subjectiva. Como exemplo tome-se o critério *usabilidade*. Dois

utilizadores/avaliadores com diferentes níveis de experiência, conhecimento prévio da ferramenta em particular, interesse por áreas de conhecimento, etc., poderão avaliar com resultados distintos o mesmo produto. Por outro lado, estas características subjectivas são muitas vezes as mais importantes para a avaliação no seu conjunto (e.g., a usabilidade). Existem assim duas formas de se ultrapassar este problema: (1) a característica pode ser substituída por um conjunto de subcaracterísticas atómicas com maior objectividade ou (2) a característica pode ser acompanhada por uma métrica específica que além de concretizar a sua semântica define a avaliação de forma mais objectiva. A dimensão temporal evidencia a reduzida perenidade das listas de características. Estas podem evoluir ao longo do tempo incluindo novas características, eliminando algumas das existentes ou mesmo alterando o significado das existentes, à medida que a tecnologia evolui e vão sendo acrescentadas inovações às ferramentas existentes.

Porém as maiores dificuldades dizem respeito à própria estruturação das características [Lundell et al., 02] presentes no standard. Através de diversos estudos de campo os mesmos autores verificaram que algumas características consideradas importantes pelas organizações estudadas ou não estavam consideradas no standard (normalmente devido a uma diferença de nomenclatura), ou estavam distribuídas por diversas características do standard, ou ainda definidas a diferentes níveis de abstracção (e com fins diferentes).

Foi também produzido por Becker e Venturini [Becker et al., 97] um sistema automático de apoio à decisão sobre a avaliação e a selecção de ferramentas CASE. Este trabalho parte do standard IEEE 1209 [IEEE, 92] naquilo que diz respeito aos critérios de avaliação e ao processo de decisão. O processo de avaliação inicia-se na fase de preparação onde é identificada a finalidade da avaliação, bem como o âmbito e eventuais restrições existentes. Nesta fase são escolhidas as características de avaliação que podem ser as já definidas no standard (cerca de 180 características atómicas organizadas numa hierarquia) ou outros. As características que resultam em respostas numéricas, booleanas ou enumerações são mais facilmente enquadráveis numa avaliação quantitativa. As características que envolvem descrições textuais podem ser relevantes para a documentação da avaliação e para a contextualização de outras características. Posteriormente são determinadas as funcionalidades da ferramenta e a sua qualidade, naquilo que diz respeito às características enunciadas. Para tal inicia-se a procura de produtos candidatos à avaliação, à qual segue-se a avaliação propriamente dita. Os resultados obtidos durante esta fase poderão ser utilizados posteriormente na selecção da ferramenta.

O processo de selecção inicia-se, da mesma forma que a avaliação, com a a definição da finalidade da selecção, o âmbito e eventuais restrições existentes. Consideram-se então os valores para a ponderação dos critérios. Esta decisão é de grande importância pois condiciona os resultados obtidos. Ao longo do tempo a organização pode rever essa ponderação alterando também os resultados obtidos sem no entanto eliminar o trabalho efectuado anteriormente. Após se aplicar os valores da ponderação aos resultados obtidos no processo de avaliação chega-se a uma recomendação de selecção. Cada um dos processos pode ser feito independentemente e os dados da avaliação podem ser reutilizados posteriormente. No entanto, como em todos os processos de avaliação, caso os critérios de avaliação mudem será necessário refazer pelo menos uma parte da avaliação existente. A reutilização dos resultados está de qualquer forma condicionada ao à

estabilidade das versões dos próprios produtos. Por isso é aconselhável realizar novas avaliações quando existam versões com características novas.

Existem hoje as fundações para conseguir uma avaliação e uma selecção tanto quanto possível objectiva. Os standards IEEE 1209 e ISO 14102 estão suficientemente bem definidos para constituirem pelo menos uma base para a produção de uma avaliação. É porém necessário tomar a decisão certa quanto às características escolhidas para os dois processos. Como foi referido, não existe ainda consenso para a nomenclatura adoptada pelos standards, mas esse facto apenas afecta uma entidade que queira partilhar os resultados da avaliação com outra. Para uma avaliação dentro de uma organização o problema não se coloca, desde que a equipa envolvida na avaliação tenha chegado a acordo quanto à nomenclatura adoptada.

Para a recolha de informação sobre as ferramentas propõe-se em seguida uma metodologia que visa retirar uma parte da subjectividade inerente a qulquer processo de avaliação que envolva a intervenção humana.

- São escolhidos os avaliadores que devem ter uma experiência considerável na utilização deste tipo de ferramentas. Tanto quanto possível devem ser recursos com experiência em diversas ferramentas, não apenas especializados numa única;

- São escolhidos os especialistas em cada uma das ferramentas a avaliar. Estes terão um período suficientemente alargado para conhecerem as ferramentas através de testes que podem ser semelhantes ou aprofundados para cada ferramenta;

- As caracaterísticas a estudar são escolhidas e o seu significado é definido, tanto quanto possível, promovendo-se a homogeneização da abordagem dos avaliadores. Devem ser tidos em consideração os aspectos particulares da organização que vai utilizar a ferramenta. Para tal é importante que uma parte considerável da equipa de avaliação inclua elementos internos à organização, preferencialmente com grande experiência.

- Os avaliadores devem inquirir os especialistas sobre cada uma das ferramentas estudadas. Cada avaliador deve inquirir apenas um grupo de características, que lhe são atribuídas. Pretende-se assim diminuir o risco de uma diferença de perspectiva poder diminuir a qualidade dos dados obtidos. Esta actividade deve ser feita tanto quanto possível sem que existam grandes intervalos entre as inquirições, idealmente no mesmo dia, tarde ou manhã.

- Para o processo de selecção são definidos os valores para a ponderação das caracteríticas. Esta actividade reveste-se da maior importância devendo ser baseada, quando possível, em experiências anteriores.

- Após a ferramenta ser escolhida devem ser realizados alguns projectos e deve ser revista a avaliação anteriormente feita, à luz da experiência entretanto adquirida. As características e os seus valores de ponderação devem igualmente ser revistos para que uma avaliação posterior tenha ainda maior qualidade.

A diferenciação entre os especialistas e os avaliadores tem como objectivo reduzir a subjectividade da avaliação através de diferentes perspectivas sobre o mesmo produto. Por outro lado, o facto de cada avaliador centrar-se apenas num grupo (tanto quanto possível reduzido) de características, contribui para que o avaliador não se disperse em diferentes aspectos da ferramenta e a avaliação seja mais justa. O último item promove a revisão contínua do próprio processo de avaliação e selecção tendo em vista a melhoria da sua qualidade.

Anexo C – Sintaxe abstracta do QVT

O metamodelo da linguagem de relações está estruturado em três *packages:* `QVTBase`, `QVTTemplate` e `QVTRelation`.

O *package QVTBase* contém um conjunto básico de conceitos, muitos deles anteriormente definidos nas especificações EMOF e OCL que estruturam as tranformações, as suas regras e os seus modelos de partida e de fim. Introduz também a noção de *padrão* como um conjunto de predicados sobre *variáveis* através de expressões OCL. Estas classes são extendidas em packages específicos a cada linguagem de forma a fornecer uma semântica específica à linguagem. A sintaxe concreta da linguagem textual encontra-se no Anexo C (Pág. 162).

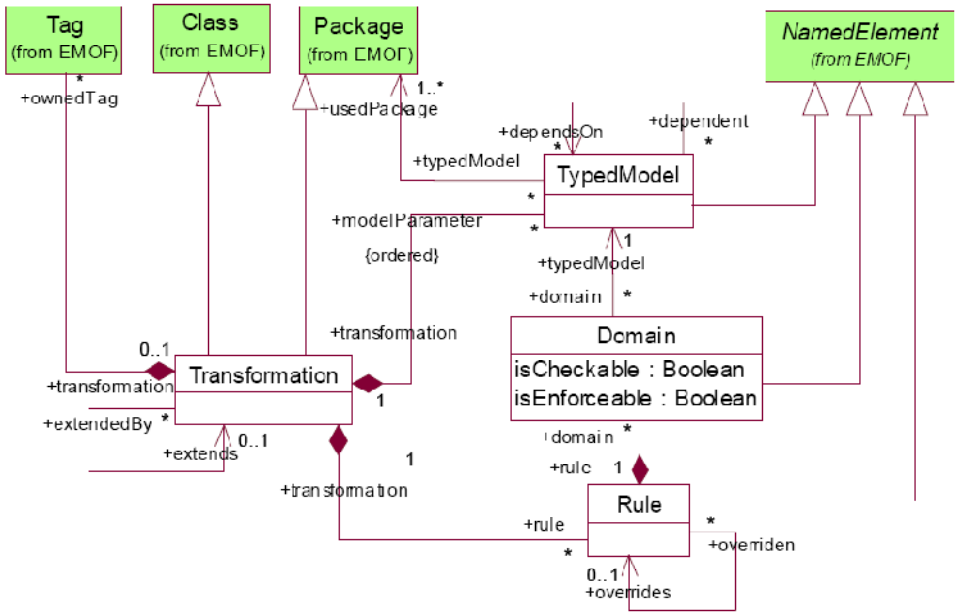


Figura C.1 – Metamodelo do *package QVTBase* (transformações e regras)

O *package* `QVTBase` inclui os seguintes conceitos essenciais:

- `Transformation`: Uma transformação define a forma como um conjunto de modelos pode ser transformado num outro conjunto. Contém um conjunto de *regras* que especificam o seu comportamento aquando da execução e é executada sobre um conjunto de modelos com tipos já definidos. Sintacticamente, uma `Transformation` é uma subclasse, tanto de `Package` como de `Class`. Como `Package` fornece um espaço de nomeação para as regras nela contidas. Como *Class* pode definir propriedades e operações – propriedades para especificar valores de configuração,

caso existam, necessários em tempo de execução e operações para implementar funcionalidades necessárias, quando necessárias, requeridas pela transformação. Uma transformação pode estender outra transformação, tendo a extensão a propriedade transitiva.

- `TypedModel`: Um modelo tipificado especifica um parâmetro tipificado de uma transformação. Em tempo de execução, um modelo que é passado para a transformação pelo seu nome só pode conter elementos cujos os tipos sejam especificados no conjunto de *packages* associados ao modelo. Em tempo de execução, uma transformação é sempre executada num sentido, i.e. escolhendo um dos modelos como alvo da transformação. Um modelo alvo pode ser produzido a partir de um ou mais modelos fonte, sendo que nestes casos a transformação pode necessitar que a selecção de elementos num modelo restrinja a selecção de elementos noutro modelo. Esta situação pode ser modelada através de um modelo tipificado que declara uma dependência noutro modelo.

- `Domain`: Um *domínio* especifica um conjunto de elementos de um modelo tipificado com interesse para uma *regra*. A classe `Domain` é abstracta, sendo as suas subclasses concretas responsáveis por especificar o mecanismo exacto pelo qual um conjunto de elementos do modelo podem ser especificados. Pode ser especificado como um padrão indicado através de um grafo, um conjunto de variáveis e restrições, ou por outro mecanismo ajustado para o efeito. Um domínio pode ser marcado como *checkable* e *enforceable,* conforme já referido.

- `Rule`: Uma *regra* especifica a forma como os elementos do modelo especificados pelos seus domínios estão relacionados uns com os outros, e a forma como os elementos de um domínio devem ser computados a partir dos elementos dos outros domínios. A class *Rule* é uma classe abstracta, cujas subclasses concretas são responsáveis por especificar a semântica exacta de como os domínios estão relacionados. Uma regra pode reimplementar uma outra regra. A reimplementação é executada quando as condições derivadas são verificadas. A semântica exacta da derivação é específica à subclasse.

- `Function`: Uma *função* é uma *operação* sem efeitos colaterais pertença de uma *transformação.* Uma função produz necessariamente o mesmo resultado cada vez que é invocada com determinados argumentos, podendo ser especificada através duma expressão OCL ou ter uma implementação de caixa-preta.

- `FunctionParameter`: Os parâmetros de função representam as variáveis que são usadas como argumentos para a função, podendo ser usadas pela expressão OCL que especifica a função.

- `Predicate`: Um *predicado* é uma expressão com valor lógico que existe num *padrão.* É especificada por uma *expressão OCL* que pode conter referências a variáveis do *padrão* que contém o *predicado*.

- `Pattern`: Um padrão é um conjunto de predicados e declarações de variáveis, os quais quando avaliados no contexto de um modelo, resultam num conjunto de ligações a variáveis.

Um outro *package* é o *QVTTemplate* que inclui os conceitos relativos à definição de *templates*.

Os conceitos presentes neste *package* são os seguintes:

- `TemplateExp`: Uma *expressão de template* especifica um padrão que é aplicado aos elementos de um modelo candidato de uma transformação. O elemento que verifica o padrão pode estar ligado a uma variável que, por sua vez, pode ser referida noutra parte da expressão. Uma *expressão de template* corresponde a uma parte de um modelo só quando a expressão *where* que lhe está associada é verdadeira. Uma *expressão de template* pode corresponder tanto a um único elemento do modelo como a uma colecção de elementos do modelo, dependendo de ser uma *expressão de template de objecto* ou uma *expressão de template de colecção.*

- `ObjectTemplateExp`: Uma *expressão de template de objecto (*ETO) especifica um padrão que apenas pode corresponder a um único elemento do modelo. Uma ETO tem um tipo correspondente à sua classe (*class*). Uma ETO é especificada por uma colecção de *itens de template de propriedade* que correspondem a diferentes atributos da classe referenciada.

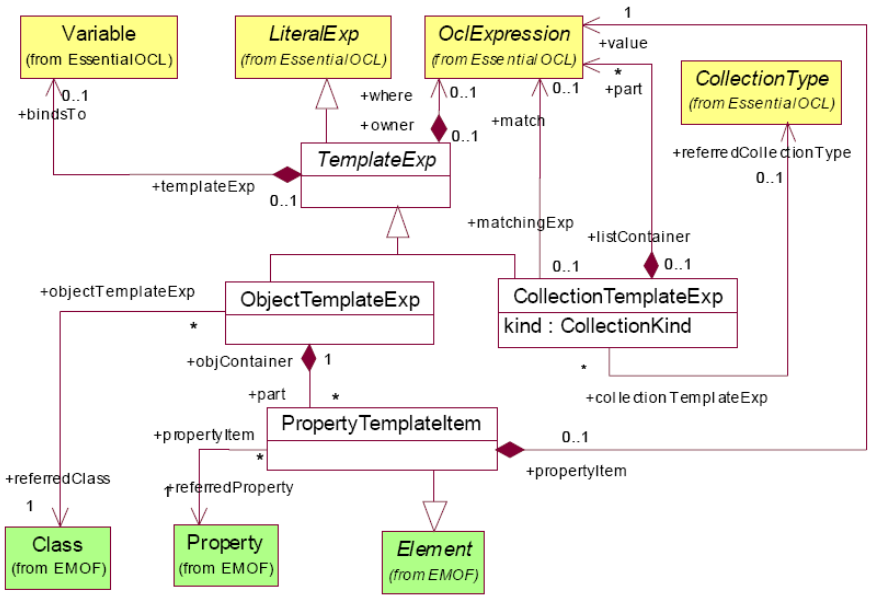


Figura C.2 – Metamodelo do *Package QVTTemplate*

- `CollectionTemplateExp`: Uma *expressão de template de colecção* (ETC) especifica um padrão que corresponde a uma colecção de elementos. O tipo da colecção resultante é dado pelo *tipo da colecção referenciada.* Uma ETC pode ser especificada de três formas diferentes, i.e., por enumeração, por compreensão ou por selecção dos membros. Sintacticamente estes três modos são indicados com as palavras reservadas `Enumeration`, `Comprehension` e `MemberSelection`, no atributo `kind:` . A interpretação do modelo nos três casos referidos é a seguinte:

- *Enumeração* – O conjunto de *expressões das partes* especifica exacatamente os elementos que devem pertencer à colecção.

- *Compreensão* – A expressão de comparação, que pode ser uma variável ou um template de objecto, liga-se a um elemento da colecção. Se um template de objecto é usado então todos os elementos da colecção devem corresponder ao padrão especificado pela *expressão de template de objecto.* Da mesma forma, todos os elementos da colecção devem satisfazer a *expressão das partes*.

- *Selecção de membros* – A expressão de comparação, neste caso uma variável, liga-se a um elemento da colecção. Se o tipo da colecção é uma sequência ou um conjunto ordenado, a expressão de comparação liga-se ao primeiro elemento da colecção. A *expressão das partes*, que só pode ser uma variável, liga-se ao resto da colecção.

- `PropertyTemplateItem`: Uma *item de template de propriedade* é usado para especificar restrições aos valores das propriedades do elemento do modelo que está a ser comparado.

Existe ainda o *package* `QVTRelation` que inclui os conceitos relativos à linguagem de relações do modelo QVT.

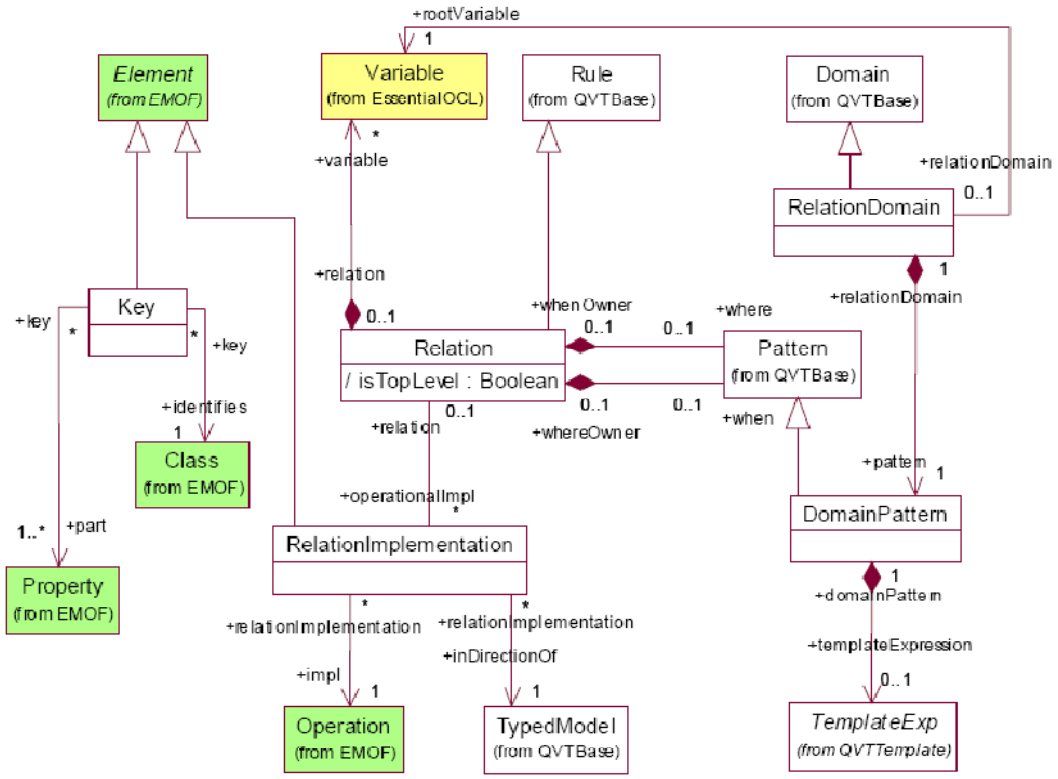


Figura C.3 – Modelo do Package QVTRelation

Os conceitos presentes no *package* `QVTRelation` são os seguintes:

- `Relation`: Uma relação é a unidade básica para a especificação do comportamento das transformações, no contexto da linguagem de relações. É uma subclasse concreta de `Rule` e especifica a relação que deve existir entre os elementos de um conjunto de modelos candidatos. Uma relação é definida por dois ou mais *domínios da relação* que especificam os elementos do

modelo que devem ser relacionados, um restrição *when* que especifica as condições em que a relação é válida, e por uma restrição *where* que especifica a condição que deve ser satisfeita pelos elementos do modelo que estão a ser relacionados.

- `RelationDomain`: Um *domínio da relação* especifica um conjunto de elementos do modelo relevantes através de um *padrão de domínio,* o qual pode ser visto como um grafo de elementos, com propriedades e relações, contendo um elemento inicial, ligado à *root variable* do domínio da relação.

- `DomainPattern`: Um padrão de domínio especifica um grafo através de expressões de template consistindo em expressões de template de objectos e itens de template de propriedade. Um padrão de domínio tem uma expressão de template inicial que tem de estar ligada à variável inicial do domínio de relação à qual ele pertence. Uma expressão de template de objecto pode incluir outras expressões de template podendo constituir ela própria um grafo em árvore.

- `Key`: Uma *chave* define um conjunto de propriedades de uma classe que identificam univocamente uma instância da classe no contexto de um modelo. Uma classe pode ter diversas chaves, da mesma forma que uma tabela no modelo relacional.

- `RelationImplementation`: Uma *implementação de relação* especifica uma implementação operacional em caixa-preta opcional para garantir o domínio da relação. A operação caixa-preta é invocada apenas quando a relação é executada na direcção do *modelo tipado* associado ao domínio que deve garantido e a relação é avaliada como falsa. A operação invocada é responsável por realizar as alterações necessárias ao modelo de forma a satisfazer a relação especificada. Existe uma excepção em tempo de execução se a relação é avaliada como falsa depois da operação ser executada. A assinatura da operação pode ser derivada da especificação de domínio da relação, i.e., um parâmetro de saída correspondente ao domínio garantido e um parâmetro de entrada correspondente a cada um dos outros domínios.

Anexo D – Linguagem de mapeamentos operacionais do QVT

A linguagem de mapeamentos operacionais do QVT permite definir transformações usando uma abordagem imperativa (através de transformações operacionais) bem como complementa as transformações relacionais com operações imperativas que implementam as relações (abordagem híbrida). Nesta secção foi feito um resumo da linguagem bem como uma descrição resumida da sua sintaxe e da sua semântica.

D.1 - Transformações Operacionais e Tipos de Modelos

Uma transformação operacional representa a definição de uma transformação unidireccional que é expressa de forma imperativa. Define uma assinatura indicando os modelos envolvidos na transformação e define uma operação de entrada para a sua execução (denominada *main*). Como uma classe, uma transformação operacional é uma entidade instanciável com propriedades e operações.

O exemplo seguinte mostra a assinatura e o ponto de entrada de uma transformação denominada *Uml2Rdbms* a qual transforma diagramas de classes UML em tabelas RDBMS:

```
transformation Uml2Rdbms(in uml:UML,out rdbms:RDBMS) {
    // the entry point for the execution of the transformation
    main() {
       uml.objectsOfType(Package)->map packageToSchema();
    }
  . . .
}
```

A assinatura do exemplo anterior declara que um modelo *rdbms* do tipo *RDBMS* será produzido a partir de um modelo *uml* do tipo *UML*. A operação de entrada *main* obtém os objectos do tipo *Package* e para cada um aplica a operação de mapeamento denominada *packageToSchema().*

Neste caso as expressões *UML* e *RDBMS* representam tipos de modelos. Um tipo de modelo é definido por um metamodelo (um conjunto de *packages* MOF), um género de conformidade (*strict* ou *effective*) e um conjunto opcional de condições. No exemplo dado, tanto *UML* como *RDBMS* referem-se a metamodelos implícitos. Quando a sintaxe textual é usada, aquele que escreve a transformação não é obrigado a indicar quais são os *packages* MOF que vão ser utilizados. Tal pode

ser feito em tempo de execução indicando-se referências aos ficheiros que contêm os metamodelos. Alternativamente, podem ser indicadas explicitamente as referências aos metamodelos:

```
modeltype UML uses SimpleUml("http://omg.qvt-examples.SimpleUml");
modeltype RDBMS "strict" uses SimpleRdbms;
```

A primeira declaração indica que o tipo de modelo UML está definido no ficheiro correspondente, através da URL dada e assume a conformidade *effective* (dada por omissão). Apesar de um modelo apontado pela variável *uml* ter que ser lido segundo esta definição, tal não significa que tenha de ser instância deste metamodelo em particular. Basta que os seus conceitos estejam decritos no metamodelo que define o tipo *UML* para que seja válido.

Na segunda declaração o tipo de modelo *RDBMS* tem a expressão *"strict"* associada. Neste caso um modelo tem de explicitamente ser definido através deste metamodelo para que possa ser considerado válido. Pode ainda ser acrescentado um bloco *where* que acrescenta restrições à aceitação do modelo como válido, segundo o metamodelo respectivo.

```
modeltype UML uses SimpleUml("http://omg.qvt-examples.SimpleUml")
where { self.objectsOfType(Class)->size()>=1};
```

No exemplo anterior foi indicada uma nova definição para o tipo de modelo *UML* que impõe a restrição de o modelo ter pelo menos uma classe.

D.2 - Bibliotecas

Uma biblioteca contém definições que podem ser reutilizadas pelas transformações. Pode definir tipos específicos e pode definir operações (e.g., operações de consulta ou construtores das metaclasses). Uma biblioteca é importada através de uma das duas formas disponíveis de importação: *access* ou *extension.* A biblioteca de base do QVT, denominada *Stdlib*, não necessita de ser explicitamente importada nas definições das transformações. A declaração seguinte define uma bilbioteca denominada *MyUmlFacilities*, a qual engloba uma lista de operações de pesquisa em modelos UML:

```
library MyUmlFacilities(UML);
query UML::Class::getAllBaseClasses() : Set(Class);
query UML::Element::getUsedStereotypeNames() : Set(String);
```

A declaração seguinte ilustra a utilização da mesma biblioteca no contexto de uma transformação *UmlCleaning:*

```
transformation UmlCleaning(inout umlmodel:UML14)
  extends MyUmlFacilities(UML14),
  access MathUtils;
  var allSuper : Set(Class); // a global variable
  main () {
     allSuper := umlmodel.objectsOfType(Class)
                    ->collect(i|i.getAllBaseClasses());
     // ...
  }
```

A transformação *UmlCleaning* extende a biblioteca *MyUmlFacilities,* o que significa que todas as operações definidas na biblioteca funcionam como se tivessem sido definidas localmente na transformação. O tipo de modelo *UML14* é uma variável local sendo ligado à variável *UML* da biblioteca *MyUmlFacilities.* No exemplo acima foi também importada uma outra bilbioteca *(MathUtils)* cuja implementação pode ter sido realizada noutra linguagem diferente do QVT.

D.3 - Operações de mapeamento

Uma operação de mapeamento é uma operação que implementa um mapeamento entre um ou mais modelos iniciais e um ou mais modelos finais. Uma opração de mapeamento é sintacticamente descrita por uma assinatura, uma condição de guarda *(when)*, um corpo do mapeamento e uma pós-condição *(where).* Uma operação de mapeamento é um refinamento do conceito de relação, anteriormente dado.

A operação de mapeamento *packageToSchema* definida em seguida mostra a forma como um *package* UML deve ser transformado num esquema relacional:

```
mapping Package::packageToSchema() : Schema
   when { self.name.startingWith() <> "_"}
{
   name := self.name;
   table := self.ownedElement->map class2table();
}
```

A relação implícita associada a esta operação de mapeamento tem a seguinte estrutura:

```
relation REL_PackageToSchema {
   checkonly domain:uml (self:Package)[]
   enforce domain:rdbms (result:Schema)[]
   when { self.name.startingWith() <> "_" }
}
```

A operação de mapeamento *packageToSchema* comporta-se como qualquer operação que seja definida na metaclasse *Package*. A variável *self* refere-se à instância da metaclasse *Package* que foi passada para a operação. A condição *when* restringe o âmbito da execução do corpo. No exemplo dado, se o nome de um *package* for iniciado com o carácter "_" a operação devolve apenas o valor *null.* Caso contrário o corpo da operação é executado.

A sintaxe genérica para o corpo da operação de mapeamento é a seguinte:

```
mapping <dirkind0> X::mappingname
   (<dirkind1> p1:P1, <dirkind2> p2:P2) : r1:R1, r2:R2
   when { ... }
   where { ... }
   {
      init { ... }
      population { ... }
      end { ... }
   }
```

Na sintaxe dada *<dirkindN>* refere-se às palavras reservadas *in, inout* ou *out* que indicam o sentido da operação de mapeamento*.* A secção *init* contém algum código que deve ser executado depois da instanciação dos valores de saída. A secção *population* contém código para preencherem os parâmetros de saída e a secção *end* contém código adicional para ser executado depois da saída da operação.

Entre as secções de *init* e *population*, existe uma secção implícita, denominada secção de instanciação, a qual cria todos os parâmetros de saída que têm um valor nulo após a secção de inicialização. Tal significa que, de forma a devolver o objecto existente em vez de criar um novo, é necessário apenas atribuir o parâmetro de saída dentro da secção de inicialização.

D.4 - Criação de objectos e modificação em operações de mapeamento

A linguagem de mapeamentos operacionais define uma forma de criar e modificar elementos do modelo. Trata-se de uma construção sintáctica, denominada *expressão de objecto,* usando a palavra reservada *object:*

```
object s:Schema {
   name := self.name;
   table := self.ownedElement->map class2table();
}
```

No exemplo anterior a expressão de objecto refere-se a uma variável existente denominada *s*, a qual necessariamente tem de ser do tipo *Schema.* A semântica desta expressão é a seguinte: se a variável *s* tem um valor nulo, é criada uma instância de Schema e atribui-se esta à variável. A lista de expressões do corpo da expressão de objecto é então executada em sequência. A expressão devolve o valor da variável *s.* Se o valor de *s* tem uma variável nula, não ocorre a instanciação, em vez disso, o corpo é usado para actualizar o objecto existente.

No âmbito de uma expressão de objecto, as propriedades da variável alvo (*s* no exemplo dado) podem ser acedidas directamente. Por isso a expressão de atribuição `name := self.name` é o mesmo que `s.name=self.name`.

As expressões de objecto podem ocorrer no contexto de uma operação de mapeamento. No exemplo seguinte é usada uma expressão de objecto para garantir a existência de um objecto da classe *Schema* quando exista um objecto da classe *Package* cujo o nome não seja iniciado com o carácter "_".

```
mapping Package::packageToSchema() : result:Schema
   when { self.name.startingWith() <> "_"}
{
   population {
      object result:Schema {
         name := self.name;
         table := self.ownedElement->map class2table();
      }
   }
}
```

Como a operação de mapeamento *packageToSchema* não tem uma secção de inicialização, a variável *result* é inicializada com uma nova instância de *Schema* depois do controlo da execução entrar na secção *population.* Assim quando a expressão de objecto é atingida, esta actua sobre um objecto já existente sendo realizada a sua modificação.

D.5 - Encadeamento e inclusão de operações de mapeamento

A definição seguinte descreve uma operação de mapeamento *class2table* que cria uma tabela do modelo relacional a partir de uma classe persistente UML. No interior do corpo desta operação são ainda usados duas outras operações definidas externamente. A operação *attr2Column* cria uma coluna a partir de um atributo e a operação *class2Key* cria uma instância de *Key* associada aos atributos primários da classe.

```
mapping Class::class2table() : Table when {self.isPersistent()}
{
   name := 't_' + self.name;
   column := self.attribute->map attr2Column());
   key := self.map class2key(result.column);
}
mapping Attribute::attr2Column() : Column {
   name:=self.name;
   type:=getSqlType(self.type);
}
mapping Class::class2Key(in cols:Sequence(Column)) : Key {
   name := 'k_'+ self.name;
   column := cols[kind='primary'];
}
```

Graças à existência das expressões de objectos é possível reescrever esta definição usando uma única operação de mapeamento:

```
mapping Class::class2table() : Table when {self.isPersistent()}
{
   name := 't_' + self.name;
   column := self.attribute -> object(a) Column{
      name=a.name;
      type=getSqlType(a.type);
   };
   key := object Key {
      name := 'k_'+ self.name;
      column := result.column[kind='primary'];
   };
}
```

A vantagem de resolver o encadeamento de operações de mapeamento através da inclusão de expressões de objectos é tornar o código mais conciso. Por outro lado, a separação do código em diferentes operações torna-o mais facilmente reutilizável.

Como foi referido, uma expressão de objecto não cria uma nova instância caso a variável alvo já esteja ligada a um valor não nulo. Na expressão "`object Key {...}`", como não existe referência a nenhuma variável, é sempre criada uma nova instância. Note-se ainda que a expressão "`self.attribute -> object(a) Column{...}`" é equivalente a "`self.attribute -> xcollect(a| object Column{...})`", onde *xcollect* é uma variante imperativa OCL da construção *collect.*

D.6 - Construtores

Existe ainda outra forma de expressar a criação de instâncias das metaclasses, definindo construtores[22]. Os construtores são operações especializadas que criam instâncias de um dado tipo. Por exemplo, para criar-se uma *UML::Operation*, pode ser definido um constructor que aceita uma lista de nomes de parâmetros e cria para cada um deles uma instância de *UML::Parameter.*

```
constructor UML::Operation::Operation(opname:String,Sequence(String));
```

Este constructor pode ser usado por qualquer transformação que trate dos modelos UML evitando a criação de diversas operações de mapeamento com o mesmo fim.

No caso do exemplo *Uml2RDBMS*, podem ser definidos construtores para *Column* e para *Key:*

```
constructor Column::Column (n:String,t: String) { name:=n; type:=t; }

constructor Key::Key(n:String,primarycols:Sequence(Column))

{name:=n;column:=primarycols;}
```

Com os dois construtores definidos pode ainda ser realizada outra variante da operação de mapeamento *class2table:*

```
mapping Class::class2table() : Table when {self.isPersistent()}
{
   name := 't_' + self.name;
   column := self.attribute->new(a) Column(a.name,getSqlType(a.type))};
   key := new Key('k_'+self.name,t.column[kind='primary'];
}
```

Embora tal não seja referido na norma, não há nada que impeça uma implementação da linguagem com construtores múltiplos para uma mesma classe. Neste caso a forma de desambiguar a chamada deverá ser pelo número e tipo de parâmetros, como é aliás usual nas linguagens de programação orientadas por objectos como C# ou Java.

---

[22] A expressão c*onstructor operations* da especificação original do QVT foi substituída por *construtores* na medida em que é evidente o paralelismo entre esta construção sintáctica e os construtores das linguagens de programação como C++, Java ou C#

D.7 - Funções

Uma função[23] realiza um certo conjunto de cálculos sobre um ou mais objectos e fornece um resultado. O corpo de uma função é uma lista ordenada de expressões que são executadas em sequência. Uma função pode alterar os próprios parâmetros, e.g., uma lista pode ser passada como parâmetro e os seus valores podem ser alterados dentro do corpo e o seu efeito ser visível depois do fim da chamada da operação. Uma função de selecção é uma função que não altera os seus parâmetros.

As funções permitem escrever facilmente operações de selecção na medida em que o utilizador não está limitado a escrever apenas uma expressão. A palavra reservada *return* é usada para sair após a função.

```
query Class::isPersistent() : Boolean = self.kind='persistent';
query Association::isPersistent() : Boolean =
(self.source.kind='persistent' and self.destination.kind='persistent');
```

O exemplo seguinte mostra uma função de pesquisa mais sofisticada definida usando um bloco de expressões:

```
query Class::checkConsistency(typename:String) : Boolean {
   if (not typename) return false;
   if (cl := self.namespace.lookForClass(typename) ) return false;
   return self.compareTypes(cl);
}
```

Demonstra-se em seguida a utilização de uma função com alteração dos parâmetros:

```
helper Package::computeCandidates(inout list:List) : List {
   if (self.nothingToAdd()) return list;
   list += self.retrieveCandidates();
   return list;
}
```

Pode ser definidas funções sobre tipos primitivos, como *strings:*

```
query String::addUnderscores() : String
                    = "_".concat(self).concat("_");
```

D.8 - Tipos intermédios

Uma transformação operacional pode usar *classes intermédias* ou *propriedades intermédias* para a sua definição. No exemplo seguinte a transformação *Uml2Rdbms* foi redefinida de forma a tratar mais correctamente atributos não primitivos. Em vez de criar uma coluna por cada atributo, pretende-se agora criar tantas colunas quantas existem num tipo de dados complexo. Uma aproximação possível para resolver este problema recursivo é usar dados intermédios.

---

[23] *Helpers* no original foi substituído por *funções* na medida em que a sua definição é intuitivamente próxima do conceito de função da maior parte das linguagens de programação

```
intermediate class LeafAttribute
                {name:String;kind:String;attr:Attribute;};
intermediate property Class::leafAttributes :
                Sequence(LeafAttribute);
```

Na declaração anterior, a classe *LeafAttribute* declara uma estrutura que representa os atributos primitivos aplanados. A propriedade intermédia *leafAttributes* permite à instância da classe guardar todos os objectos intermédios derivados da sua definição. Tendo a declaração anterior é possível produzir uma nova definição para o mapeamento *class2table* que será:

```
mapping Class::class2table() : Table
   when {self.isPersistent() ;} {
      init {
         self.leafAttributes := self.attribute->
                        map attr2LeafAttrs();
      }
      name := 't_' + self.name;
      column := self.leafAttributes->map leafAttr2OrdinaryColumn();
      key := object Key {
         name := 'k_'+ self.name;
         column := result.column[kind='primary'];
      };
   }
```

Os atributos finais na árvore, são criados na secção de inicialização através da invocação da operação de mapeamento recursiva denominada *attr2LeafAttrs* (não desenvolvida aqui). A iteração sobre esta lista é então usada para criar as colunas através da operação *leafAttr2OrdinaryColumn.*

É de salientar que as propriedades intermédias são manipuladas como propriedades normais da metaclasse que serve de contexto para a operação (neste caso *UML::Class*). As propriedades intermédias são extensões às propriedades que não existem fora do âmbito onde são definidas.

D.9 - Actualização de objectos e resolução de referências

Uma técnica comum na transformação de modelos é a utilização de diversas passagens para resolver referências cruzadas entre elementos do modelo. A linguagem fornece o mecanismo de *resolução* para permitir o acesso a objectos alvo criados anteriormente de objectos iniciais. Este mecanismo utiliza implicitamente os registos de rastreabilidade criados pela execução da operação de mapeamento.

A definição *asso2table* seguinte é responsável por adicionar uma chave estrangeira a uma tabela relacional anteriormente criada. Para tal é necessário obter-se uma tabela já existente.

```
mapping Association::asso2table() : Table
   when {self.isPersistent()}
{ -- result is the default name for the output parameter of the rule
   init { result := self.destination.resolveone(#Table); }
   foreignKey := self.map asso2ForeignKey();
   column := result.foreignKey.column;
}
```

A operação *resolveone* inpecciona os dados de rastreabilidade para verificar se existe uma instância de *Table* criada pela associação com a classe alvo e que satisfaça a condição booleana. No caso apresentado a condição é ser do tipo *Table* (a notação *#Table* é um atalho para *isKindOf(Table))*.

Existem três variantes para a *resolução.* O operador *invresolve* realiza o tratamento inverso, isto é, procura os objectos que foram responsáveis pela criação do objecto passado como argumento do contexto*.* O operador *resolveIn* procura os objectos alvo criados de um objecto fonte através de uma única operação de mapeamento. No exemplo seguinte ilustra-se esta utilização: a partir de uma lista de classes Java (instâncias de *Jclass*) que têm um campo denominado *packageName* indicando o nome do *package* a que pertencem, a transformação *Jclass2Jpackage* cria um package Java (*Jpackage*) para cada nome de *package* que é encontrado na lista de classes Java.

```
transformation JClass2JPackage(inout javamodel:JAVA);
   main () { javamodel->objectsOfType(JClass)->jclass2jpackage();}
   mapping Class::jclass2jpackage() : JPackage () {
      init {
         result := resolveIn(jclass2jpackage,true)
            ->select(p|self.package=p.name)->first();
         if result then return;
      }
      name := self.package;
   }
```

No exemplo anterior, usa-se *return* para evitar criar mais do que um *package* com o mesmo nome.

A variante final do operador de *resolução* é a capacidade de adiar a recepção dos objectos até ao fim da transformação (i.e., até ao fim da execução da operação de entrada). Resoluções diferidas podem ser úteis para evitar definir várias passagens de uma transformação. No exemplo seguinte tratam-se os ciclos que possam existir devido a dependências hierárquicas entre classes:

```
transformation Uml2Java(in uml:UML,out java:JAVA)
   main() : JAVA::Interface {
      uml->objectsOfType(Class)->map transformClass();
   }
   mapping UML::transformClass() : JAVA::Interface () {
      name := "Ifce".concat(self.name);
      base := self.superClass->late resolve(#JAVA::Interface);
   }
```

Na definição seguinte trata-se o mesmo problema, mas desta vez usando duas passagens:

```
transformation Uml2Java(in uml:UML, out java:JAVA)
   main() : JAVA::Interface {
      uml->objectsOfType(Class)->map transformClass();
      uml->objectsOfType(Class)->map transformClassInheritance();
   }
   mapping UML::transformClass() : JAVA::Interface {
      name := "Ifce".concat(self.name);
   }
   mapping UML::transformClassInheritance() : JAVA::Interface {
      base := self.superClass->
                     resolveIn(transformClass,#JAVA::Interface);
   }
```

Em termos de execução, o operador *late resolve* é sempre associada a uma atribuição. Conceptualmente devolve *null* guardando entretanto toda a informação que é necessário para reexecutar a inspecção e a atribuição.

D.10 - Composição de transformações

A composição de transformações é uma característica essencial para produzir transformações complexas e envolvendo um elevado número de metaelementos.

Como exemplo, considere-se que a transformação *Uml2Rdbms* necessita que o modelo inicial *uml* seja reduzido de forma a não conter nenhuma associação redundante. Será necessário estender a definição da transformação anterior invocando, antes desta, uma outra transformação que garanta a "limpeza" do modelo inicial. Tal pode ser conseguido através da seguinte definição:

```
transformation CompleteUml2Rdbms(in uml:UML,out rdbms:RDBMS)
   access transformation UmlCleaning(inout UML),
   extends transformation Uml2Rdbms(in UML,out RDBMS);
main() {
   var tmp: UML = uml.copy();
   var retcode := (new UmlCleaning(tmp))->transform();
                        //performs the cleaning
   if (not retcode.failed())
      uml.objectsOfType(Package)-> map packageToSchema()
   else raise "UmlModelTransformationFailed";
}
```

No exemplo anterior é demonstrado o uso dos mecanismos de reutilização *access* e *extension.* Uma importação através de *access* comporta-se como uma importação normal de um *package.* Se a importação for realizada com a palavra reservada *extension* combina-se a semântica da importação normal de um *package* com a herança de classes. O mesmo exemplo ilustra igualmente: (1) a capacidade de executar transformações localizadas, como *UmlCleaning;* (2) a capacidade de executar uma instanciação explícita da transformação (através do operador *new)* e (3) a capacidade de invocar operações de alto nível sobre modelos, como a operação *copy().*

D.11 - Reutilização em operações de mapeamento

A linguagem fornece dois mecanismos de reutilização ao nível das operações de mapeamento: herança de mapeamentos ou fusão de mapeamentos.

Uma operação de mapeamento pode herdar de uma outra operação de mapeamento. No que diz respeito à semântica de execução, o mapeamento herdado é executado depois da secção de inicialização do mapeamento que está a herdar. O exemplo seguinte ilustra a utilização da herança de mapeamento. O mapeamento que cria colunas RDBMS estrangeiras reutiliza o mapeamento definido para criar colunas RDBMS "normais".

```
mapping Attribute::attr2Column (in prefix:String) : Column {
   name := prefix+self.name;
   kind := self.kind;
   type :=
      if self.attr.type.name='int'
      then 'NUMBER'
      else 'VARCHAR'
      endif;
}

mapping Attribute::attr2ForeignColumn (in prefix:String) : Column
```

```
   inherits leafAttr2OrdinaryColumn {
      kind := "foreign";
}
```

Dentro de uma transformação, uma operação de mapeamento pode também declarar uma lista de operações de mapeamento que complementa a sua execução. A este processo chama-se *fusão* de mapeamentos. Em termos de execução, a lista ordenada de mapeamentos fundidos é executada em sequência após a secção *end.* As regras de compatibilidade entre o mapeamento que chama e aquele que é chamado restringem os parâmetros do mapemanto complementar a estarem conforme os parâmetros que servem de base à fusão.

O exemplo seguinte demonstra uma definição de transformação que usa a fusão de mapeamentos. Este estilo de escrita permite definir especificações modulares onde podem ser tidos em conta o acoplamento e a coesão entre os mapeamentos.

```
// Rule 1 (in english): A Foo should be transformed into an Atom
// and a Bar. The name of the Bar is upperized and the name of the
// Bar is lowerized
mapping Foo::foo2atombar () : atom:Atom, bar:Bar
merges foo2barPersistence, foo2atomFactory
{
   object atom:{name := "A_"+self.name.upper();}
   object bar:{ name := "B_"+self.name.lower();}
}
// Rule 2: Persistent attributes of Foo are treated as volatile
// Bar properties
mapping Foo::foo2barPersistence () : atom:Atom, bar:Bar
when {foo.isPersistent();} {
   object bar:{ property := self.attribute->map persistent2volatile(); }
}
// Rule 3: An Atom factory should be created for each Atom and
// have the name of the associated Bar.
mapping Foo::foo2atomFactory () : atom:Atom, bar:Bar {
   object bar:{ factory := object Factory {name := bar.name}};
}
```

Um mapeamento fundido não é invocado se a condição de guarda não é satisfeita. No exemplo dado a regra 3 tem a condição `foo.isPersistent()` que tem de ser satisfeita para que o mapeamento seja executado.

O código seguinte mostra uma invocação do mapeamento `foo2atombar`. Os valores resultantes são atribuídos às variáveis *atom* e *bar*.

```
var f := lookForAFooInstance();
var (atom: Atom, bar: Bar) := f.foo2atombar();
```

Devemos notar que conceptualmente o parâmetro resultante é tratado como *parâmetro opcional* da operação de mapeamento. Assim, a primeira chamada a `f.foo2atombar()` é equivalente a invocar `f.foo2atombar(null, null)`. Em contraste, `foo2barPersistence()` e `foo2atomFactory()` são invocados internamente com as instâncias de `atom` e `bar` criadas pelo mapeamento `foo2atombar()`. Este mecanismo permite aos mapeamentos fundidos alterar as instâncias criadas pelo mapeamento que serve de base à fusão.

D.12 - Disjunção de operações de mapeamento

Uma operação de mapeamento pode ser definida como a disjunção de uma lista ordenada de mapeamentos. Tal significa que a invocação da operação resulta na selecção do primeiro mapeamento cuja condição de guarda (tipo e condição *when*) é válida. O valor *null* é devolvido se nenhum dos mapeamentos é válido.

No exemplo seguinte usa-se a palavra reservada `disjuncts` para definir a operação de mapeamento `convertFeature()`:

```
mapping UML::Feature::convertFeature () : JAVA::Element
disjuncts convertAttribute, convertOperation, convertConstructor() {}
mapping UML::Attribute::convertAttribute : JAVA::Field {
   name := self.name;
}
mapping UML::Operation::convertConstructor : JAVA::Constructor {
   when {self.name = self.namespace.name;}
   name := self.name;
}
mapping UML::Operation::convertOperation : JAVA::Constructor {
   when {self.name <> self.namespace.name;}
   name := self.name;
}
```

D.13 - Extensões aos tipos

A linguagem operacional estende o sistema de tipos de base do OCL e do MOF com três tipos genéricos que podem ser descritos da seguinte forma: listas mutáveis, dicionários e tuplos anónimos.

Uma lista mutável *(List)* contém um lista de elementos ordenados[24] pela ordem de entrada na lista. Em contraste com uma colecção OCL usual, uma *List* pode ser alterada. Uma *List* é um tipo parametrizável, i.e., é definido sobre um tipo[25]. Quando não seja dado um tipo para a criação da lista, é assumido o tipo *Any* para a lista.

```
var mylist := List{1,2,3,4}; // a list literal
mylist.add(5);
```

Um dicionário *(Dict)* pode ser visto como uma estrutura de dados que guarda valores acessíveis por chaves. É também um tipo mutável, i.e., um objecto pode ser alterado depois de ser criado.

```
var mydict := Dict{"one"=1,"two"=2,"three"=3}; // a dictionary literal
mydict.put("four",1);
```

[24] Na língua Inglesa as palavras *sorted* e *ordered* diferenciam os dois tipos de ordem, podendo ser traduzidas, respectivamente, neste contexto por lista ordenada (i.e., com os elementos ordenados entre si) e lista ordeira (i.e., com os elementos em fila). Note-se ainda que já a linguagem Smalltalk tinha esta distinção através das colecções *SortedCollection* e *OrderedCollection.*

[25] Uma *List* é definida em QVT da mesma forma que uma colecção genérica em C#, i.e., indicando o tipo dos elementos que podem pertencer a essa *List*.

Um tuplo anónimo é um tuplo cujas posições não têm designação. Pode ser usado de uma forma abreviada quando é necessário colocar os valores do tuplo em variáveis individuais.

```
var mytuple := Tuple{1,2,3}; // an anonymous tuple literal
var (x,y,z) := mytuple; // unpacking the tuple into three variables
```

Existe um outro mecanismo que torna o sistema de tipos mais flexível. O *typedef* permite anexar restrições adicionais a um tipo existente, num determinado contexto. Permite também definir sinónimos para tipos complexos. Quando usado na assinatura de uma operação de mapeamento, as constantes *typedef* são acrescentadas à condição de guarda da operação. A condição é expressa entre parentesis rectos, após o tipo tomado como referência.

```
typedef TopLevelPackage = Package [_parentInstance()=null];
typedef AttributeOrOperation = Any [#Attribute or #Operation];
typedef Activity = ActionState[stereotypedBy("Activity")];
```

O tipo definido por um *typedef* é considerado estar no âmbito do tipo do modelo do tipo tomado como referência[26]. E.g. se o tipo `ActionState` existe no contexto de um tipo de modelo *UMLDiagram*, então `stereotypedBy("Activity")` é também considerado apenas nesse contexto.

Um *typedef* pode também ser usado para definir um sinónimo para um tipo complexo:

```
typedef PersonInfo = Tuple{name:String,phone:String};
```

D.14 - Expressões imperativas

Como já foi referido, a linguagem de mapeamentos operacionais é uma linguagem imperativa para definir transformações. Estende o OCL inluindo todas os elementos necessários para realizar transformações complexas de uma forma simples. As expressões imperativas em QVT realizam um compromisso entre algumas características funcionais encontradas no OCL e algumas construções sintácticas usualmente encontradas em linguagens imperativas como Java ou C#. O exemplo mais relevante da união destas duas aproximações é a possibilidade de utilizar expressões de bloco para realizar um determinado cálculo.

```
compute (v:T := initexp) { ... self.getSomething() ...};
```

No exemplo anterior é devolvida a variável *v* após a execução do corpo de instruções. Neste corpo de instruções podem existir referências a variáveis definidas num âmbito mais vasto podendo estas ser alteradas. Um bloco não é, portanto, uma função, apenas um conjunto de instruções.

Esta construção pode ser combinada com uma expressão *while:*

```
self.myprop := while(v:T = initexp; v<>null)
{ … self.getSomething() …}
```

[26] Considera-se que um tipo insere-se num modelo que por sua vez tem ele próprio um tipo (tipo do modelo)

```
// "while(v;cond) body" is a shorthand for
// "compute(v) while(cond) body"
```

A expressão *forEach* é um ciclo imperativo que pode também iterar sobre um bloco e opcionalmente pode realizar um filtro sobre os elementos da lista:

```
self.ownedElement->forEach(i|i.isKindOf(Actor)) { … }
range(2,8)->forEach(i) { … }
```

Dentro de ciclos imperativos é possível utilizar-se *break* e *continue.* A combinação de *compute* e de *forEach* permite definir operações imperativas *xcollect* e *xselect* assim como outras operações de manipulação de alto nível.

A linguagem define igualmente uma construção do tipo "if-then-else" que não está restringida como a construção sintáctica correspondente em OCL, na medida em que não é estritamente necessário existir "else". A notação é a seguinte:

```
var x:= if (self.name.startsWith("_")) 0
        elseif (self.type.isPrimitive()) while (res := 0; res<10) { … };
        else -1;
```

Em geral, não é obrigatório usar esta expressão de controlo apenas em expressões. Pode ser usadas como é usual nas linguagens imperativas.

```
if (x==0) {
   list->forEach(i) {
    if (...) continue;
    ...
   }
}
```

6.2 D.15 - Outras palavras reservadas

A palavra *this* representa a instância que está a ser definida pela transformação. Pode ser usada implicitamente para se referir à propriedades da classe de transformação ou às suas operações.

Numa operação contextual (uma função de selecção ou uma mapeamento operacional) a palavra *self* representa o parâmetro contextual.

No âmbito de uma operação contextual a palavra *result* representa o resultado, podendo este ser um valor único ou um tuplo contendo os parâmetros declarados como resultado.

A palavra *null* é um literal que pode ser atribuído a uma variável de qualquer tipo[27]. Pode ser explicitamente devolvida por uma operação ou pode ser implicitamente devolvida quando o resultado não é indicado.

[27] Esta parte da especificação do QVT implica que mesmo as variáveis dos tipos básicos possam receber o valor *null*, o que não é usual nas linguagens de programação como o Java ou C#. É, mais uma vez, notória a influência de linguagens não tipificadas como o Smalltalk.

6.3 D.16 - Características avançadas: definição dinâmica e paralelismo

Quando se trata com um processo MDA complexo, pode ser necessário definir transformações que usam definições de transformações elas próprias geradas automaticamente. É mesmo possível para uma definição de uma transformação ser o resultado de outra definição de transformação na medida em que um modelo QVT pode ser representado como um modelo. A linguagem providencia uma operação prédefinida *asTransformation* que permite considerar uma definição de transformação (um modelo tipificado através de um tipo de modelo que aceita definições de acordo com o QVT) como uma instância da respectiva classe de transformação. A implementação desta operação tipicamente requer compilar a definição da transformação em tempo de execução.

O exemplo seguinte ilustra uma possível utilização para este mecanismo. A transformação *Pim2Psm* transforma um modelo PIM num modelo PSM. Neste ponto, o modelo inicial PIM é primeiramente anotado, usando um conjunto ordenado de *packages* UML que definem as regras de transformação a serem inferidas das anotações, tendo por base um qualquer formalismo proprietário orientado por UML. Cada *package* UML definindo uma transformação é transformado na especificação de acordo com o QVT. Quando executada em sequência, cada uma das definições de transformação QVT acrescenta o seu próprio conjunto de anotações ao modelo PIM. No fim, o PIM anotado é convertido num modelo PSM usando uma transformação *AnnotadePim2Psm.*

```
transformation PimToPsm(inout pim:PIM, in transfSpec:UML, out psm:PSM)
access UmlGraphicToQvt(in uml:UML, out qvt:QVT)
access AnnotatedPimToPsm(in pim:PIM, out psm:PSM);

main() {
   transfSpec->objectsOfType(Package)->forEach(umlSpec:UML) {
      var qvtSpec : QVT;
      var retcode := new UmlGraphicToQvt(umlSpec,qvtSpec).transform();
      if (retcode.failed()) {
         log("Generation of the QVT definition has failed",umlSpec);
         return;};
      if (var transf := qvtSpec.asTransformation()) {
         log("Instanciation of the QVT definition has failed",umlSpec);
         return;}
      if (transf.transform(pimModel,psmModel).failed()) {
         log("failed transformation for package spec:",umlSpec);
         return;}
   }
}
```

O exemplo dado, embora correcto sintacticamente, tem a atribuição "`var transf := ...`" que é simultaneamente uma definição de variável, uma atribuição propriamente dita e a devolução de um valor booleano. Como o contexto da variável é a operação *main()* uma solução mais estruturada seria declará-la juntamente com as restantes variáveis da operação e inicializá-la com o valor *null.*

Uma outra característica avançada é a execução em concorrência das diversas transformações. Estas são úteis quando não existem restrições de sequência sobre um conjunto de transformações contidas numa única. Em termos de execução, invocar a transformação de alto nível funciona como criar um conjunto de subprocessos para realizam a tarefa. A sincronização é realizada esperando pelas variáveis que são devolvidas pela execução das subtransformações.

No exemplo seguinte transforma-se um modelo de requisitos num modelo PSM, decompondo-o em dois modelos intermédios PIM (um para a interface gráfica e outro para o comportamento) e fundindo-os depois no modelo PSM.

```
transformation Req2Psm (inout pim:REQ, out psm:PSM)
access Req2Pimgui(in req:REQ, out pimGui:PIM)
access Req2Pimbehavior(in req:REQ, out pimBehavior:PIM),
access Pim2Psm(in pimGui:PIM, in pimBehavior:PIM, out psm:PSM);

main() {
   var pimGui : PIM := PIM::createEmptyModel();
   var pimBehavior : PIM := PIM::createEmptyModel();
   var tr1 := new Req2Pimgui(req, pimGui);
   var tr2 := new Req2Pimbehavior(req, pimBehavior);
   var st1 := tr1.parallelTransform(); // forks the PIM GUI transformation
   var st2 := tr2.parallelTransform();
                  // forks the PIM Behavior transformation
   this.wait(Set{st1,st2});           // waits patiently
   if (st1.succeeded() and st2.succeeded())
       new Pim2Psm(pimGui,pimBehavior,psm).transform();
                  // creates the executable model

}
```

Anexo E - Sintaxe abstracta e semântica da linguagem de relações do QVT

Nesta secção será abordada a sintaxe abstracta da linguagem de mapeamentos operacionais. Os conceitos são apresentados de uma forma gráfica através de diagramas de classes e posteriormente é feita uma pequena descrição de cada uma das classes envolvidas. Quando necessário, são criadas novas subsecções que descrevem com maior detalhe a semântica de um dado conceito.

O formalismo operacional do QVT é descrito por dois *packages* EMOF: `QVTOperational` e `ImperativeOCL`. Os seguintes pacotes são importados: `EMOF`, `EssentialOCL`, `QVTBase`, `QVTTemplate` e `QVTRelation`.

O *package* `QVTOperational` define os conceitos que são necessários para especificar as definições das transformações escritas imperativamente. Este *package* define um conjunto de conceitos estruturais genéricos (e.g., *módulo* e *operação imperativa*) e de conceitos mais específicos (e.g., *transformações operacionais, operações de mapeamento*). Como já referido, usa o *package* `ImperativeOCL` que é uma extensão ao OCL com construções sintácticas comuns às linguagens de programação imperativas.

O *package QVTOperational* especifica os seguintes conceitos:

`OperationalTransformation`: Uma transformação operacional representa a definição de uma transformação unidireccional expressada imperativamente. Tem uma assinatura que indica qual ou quais os modelos envolvidos na transformação e define uma operação de entrada, denominada `main`, a qual representa o código inicial a ser executado para realizar a transformação. Uma transformação operacional necessita de uma assinatura, no entanto pode não ter uma implementação. Permite-se assim implementações em caixa-preta definidas fora do QVT. Uma transformação operacional pode *estender* ou *aceder* (através das palavras reservadas `extend` e `access`) uma transformação operacional já existente ou uma biblioteca existente (ver classes `ModuleImport` e `Library`).

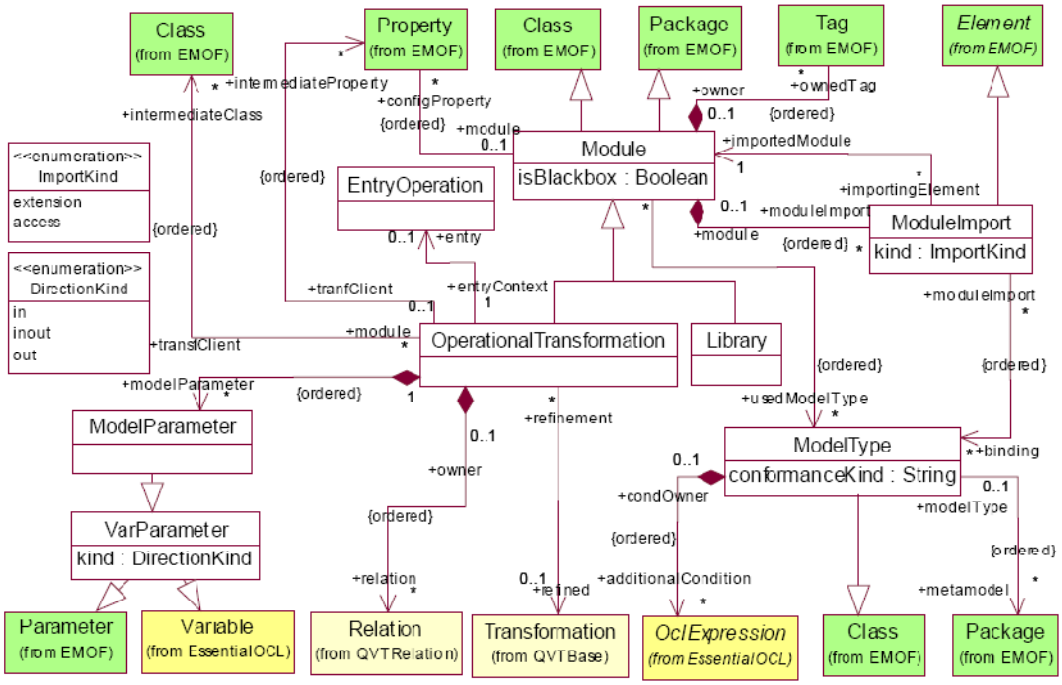


Figura E.1 – *QVT Operational Package* – Transformações Operacionais

Uma transformação operacional pode definir propriedades de configuração, i.e., propriedades cujo o valor só é dado em tempo de execução. Pode também definir propriedades imediatas e definir explicitamente novas classes para guardar dados intermédios. Quando uma transformação operacional refina uma transformação relacional (Secção 2.4.2), o sentido da transformação tem de ser indicado. Sintacticamente, uma `OperationalTransformation` é uma subclasse de `Module`, logo, por herança, também o é de `Class` e de `Package`. Como herda da metaclasse `Class` pode definir propriedades e operações (neste caso podem ser *funções de selecção, operações de mapeamento* e *construtores*) tendo que ser instanciada para ser executada. Como também herda da classe `Package` pode definir e conter tipos específicos para serem usados no âmbito da definição da transformação.

A semântica da execução de uma transformação operacional pode ser dada facilmente comparando esta com uma classe Java. Todas as propriedades e operações da transformação funcionam como atributos e métodos da classe Java. Adicionalmente, para cada parâmetro (modelo de inicial ou final) existe um atributo. Um construtor existe também para a transformação: a sua assinatura corresponde à lista de parâmetros `in` e `inout`. O código deste construtor implícito (na transformação) realiza as seguintes acções:

- Para cada parâmetro out é criado um domínio MOF com conteúdo vazio;
- Os parâmetros são colocados nos respectivos atributos da transformação e
- É feito o carregamento dos valores das propriedades de configuração, quando existentes.

A instanciação da transformação pode ser implícita (a instância pode estar a ser referida usando a variável `this`) ou explícita. No segundo caso é usada uma expressão `InstantiationExp`. O preenchimento de propriedades de configuração pode ser realizado usando um mecanismo externo,

e.g. usando um ficheiro de configuração. Tal como em Java, a instanciação e a execução da operação de entrada (`main`) são acções diferentes. Uma transformação é explicitamente invocada usando a operação pré-definida `transform`. Uma invocação desta operação provoca a execução da lista de expressões do corpo da operação de entrada. No fim da execução da operação `main`, as atribuições diferidas, caso existam, são executadas em sequência. Só então termina a execução da transformação.

A notação para definir a transformação usa a palavra reservada `transformation` no cabeçalho. É possível criar diversas transformações seguidas, num único ficheiro, da seguinte forma:

```
// defining multiple transformations in a single file
transformation Uml2Rdbms(in uml:UML, out rdbms:RDBMS) {
   // content of the transformation definition
}
transformation Logical2PhysicalRdbms(inout rdbms:RDBMS) {
   // content of the transformation definition
}
```

A declaração seguinte define uma transformação operacional designada `Uml2Rdbms` com uma assinatura formada por dois parâmetros (modelos) `uml` e `rdbms` com tipos `UML` e `RDBMS`, respectivamente. Define também uma marca de metainformação (`tag`) que indica o criador da transformação.

```
transformation Uml2Rdbms(in uml:UML, out rdbms:RDBMS);
tag "author" Uml2Rdbms = "Pepe";
```

Os módulos importados (transformações ou bibliotecas) são indicados na mesma instrução, após os parâmetros dos modelos, usando as palavras reservadas `extends` ou access. As palavras transformation ou library podem ser usadas para documentarem o tipo de módulo que está a ser importado.

```
transformation Uml2Rdbms(in uml:UML, out rdbms:RDBMS)
extends BasicUml2Rdbms,           // extending a transformation
extends library UMLUtilities(UML) // extending a library
access library MathLibrary;       // accessing a math library
```

Todas as declarações access podem estar em instruções diferentes (não fazendo parte necessariamente do cabeçalho). Uma transformação operacional indica um refinamento de uma outra transformação operacional através da palavra reservada `refines`:

```
transformation Uml2Rdbms(in uml:UML, out rdbms:RDBMS)

refines R_UML2RDBMS;
```

As classes intermédias e as propriedades intermédias de uma transformação podem ser representadas usando as palavras reservadas `property` e `class` prefixadas com a palavra `intermediate`:

```
intermediate class LeafAttribute {...}
intermediate property UML::Attribute::extravalue : String;
```

As propriedades que são propriedades de configuração são declaradas usando o qualificador `configuration`:

```
configuration property UML::Attribute::MAX_SIZE : String;
```

`Library`: Uma biblioteca é um agrupamento de operações e definições de tipos que são juntos para poderem ser reutilizados posteriormente. A *QVT Standard Library* é um exemplo de biblioteca. À excepção desta todas as outras bibliotecas têm de ser explicitamente importadas. Uma biblioteca pode ser declarada como caixa-preta (neste caso não é dada uma implementação às suas operações). Uma biblioteca pode declarar uma lista de tipos de modelos sobre os quais pode operar. Esta lista constitui a assinatura da biblioteca. Sintacticamente, uma `Library` é uma subclasse de Module, sendo por isso subclasse de `Class` e de `Package`. A notação para definir uma biblioteca é similar à notação usada para definir uma transformação, excepto que deve ser utilizada a palavra reservada `library`, em vez de transformation. A assinatura da biblioteca é, neste caso, uma lista de tipos de modelos, em contraste com as transformações, onde a assinatura é composta de parâmetros de modelos. A declaração seguinte define uma biblioteca chamada `UmlUtilities` que estende a biblioteca `BasicUmlUtilities`, tendo uma assinatura formada por um tipo de modelo `UML1_4`:

```
library UmlUtilities(UML1_4)
   extends BasicUmlUtilities(UML1_4)
   access MathLibrary ;
```

`Module`: Um módulo é uma unidade contendo um conjunto de operações e de tipos definidos para operar em modelos. Este conceito define atributos comuns partilhados por transformações operacionais e bibliotecas. Sintacticamente, um `Module` é uma subclasse de `Class` e `Package`. Sendo uma `Class` pode definir propriedades e operações. Sendo um `Package` pode definir e conter tipos específicos para serem usados na definição do módulo.

`ModuleImport`: uma *importação de módulo* representa a utilização de um módulo através de uma das duas semânticas de importação possíveis. Com a semântica `extension`, a importação do módulo é semelhante a definir as operações e os atributos do módulo importado no próprio módulo. Assim, uma operação ou uma propriedade do módulo importado são considerados como fazendo parte do módulo importador, sendo esta semântica semelhante à da extensão de classes. Com a semântica `access`, as definições do módulo importado não são herdadas pelo módulo importador. Neste caso tem de ser usada uma *instância* do módulo importado para se aceder a estas definições. No caso do acesso a uma biblioteca, uma instância do módulo importado fica implicitamente disponível. Quando o módulo não é uma biblioteca, a instância que está a ser acedida pelo módulo deve ser criada explicitamente. Dois nomes idênticos vindos de dois módulos diferentes podem ser distinguidos qualificando-os. No entanto, um símbolo local tem sempre precedência sobre os símbolos importados.

`ModelParameter`: Um *modelo parâmetro* é um parâmetro para uma transformação operacional. Desta forma um conjunto ordenado de modelos parâmetros formam a assinatura da transformação. Cada modelo parâmetro refere-se implicitamente a um modelo que participa numa selecção ou numa transformação. Cada modelo parâmetro contém uma indicação explicitando o efeito da execução do módulo sobre o modelo: `in` significa que as alterações não são permitidas, `inout` significa que o modelo pode ser alterado, `out` significa que o modelo deve ser criado. Estes parâmetros modelos são globalmente acessíveis dentro da transformação. Cada modelo parâmetro

tem um tipo (`ModelType`). Os modelos parâmetros são anotados como simples parâmetros na assinatura da transformação. Se o sentido da transformação não é fornecido, o valor `in` é assumido por omissão.

`ModelType`: Cada modelo parâmetro tem um tipo de modelo, o qual é definido ou referenciado por uma transformação ou por uma biblioteca. Um tipo de modelo é definido por um metamodelo, um tipo de conformidade e um conjunto opcional de restrições. O metamodelo define um conjunto de classes e de propriedades de elementos que são esperados pelas transformações.

Quando uma transformação operacional é instanciada, os parâmetros passados como argumentos devem estar em conformidade com os tipos dos *modelos parâmetros*. A conformidade é definida de duas formas: `strict` e `effective`. Quando a conformidade é `strict`, os objectos do domínio do modelo devem ser necessariamente instâncias das classes do metamodelo associado. Quando a conformidade é `effective`, qualquer objecto no domínio do modelo que tem um tipo que se refere a um tipo do metamodelo tem de conter as propriedades definidas no respectivo metamodelo com os tipos de dados compatíveis. A ligação entre os tipos de diferentes metamodelos é baseada na comparação de nomes, excepto para uma renomeação dada pela anotação `alias`. A conformidade efectiva pemite que sejam definidas transformações flexíveis aplicadas a metamodelos semelhantes. E.g., se uma transformação é definida em UML 1.4 mas não usa nenhum dos elementos específicos ao UML 1.4, pode ser também usada com modelos do UML 1.3. Em ambos os casos (`strict` e `effective`), a conformidade de modelos implica também a conformidade com a regras de correcta criação dos metamodelos associados. E.g., se o UML 1.4 é o metamodelo, as regras de correcta criação são regras que estão definidas no documento de especificação formal da OMG. Para restringir o conjunto dos modelos participantes, um tipo de modelo pode especificar uma lista outras condições (expressas como expressões OCL) que necessitam de serem válidas para os modelos participantes. E.g., uma transformação que espere modelos UML com casos de uso pode ser restringida de forma a não admitir modelos que não tenham este tipo de diagramas. Um tipo de modelo é definido como uma subclasse de `Class` tal que é possível definir operações e propriedades nele. As propriedades definidas nos tipos podem ser usadas para se observar o conteúdo dos objectos que pertençam ao modelo, durante o tempo de execução da transformação. Mais precisamente, existe um âmbito MOF correspondente a cada parâmetro. Qualquer criação de um objecto ocorre num âmbito associado ao modelo parâmetro. Quando uma transformação operacional é instanciada, os parâmetros do modelo passados como argumentos devem estar em conformidade com os tipos dos modelos da transformação instanciada.

Quando um elemento de um modelo é criado por uma transformação, é necessário conhecer qual o modelo em que o elemento é criado. Para este fim, é possível utilizar operações de inspecção em *modelos parâmetros* (e.g., `objects()` e `objectsOfType())` para devolver um objecto previamente criado. Em MOF existe o conceito de âmbito (`Extent`) o qual é apenas um conjunto de objectos. Relaciona-se aqui o conceito de modelo (representado por modelos parâmetros numa definição de transformação) com o conceito de âmbito MOF, assumindo que para cada modelo parâmetro existe um âmbito MOF.

Um tipo de modelo é referido por um nome na sua assinatura ou na declaração access ou extends de uma transformação ou biblioteca. No exemplo seguinte, os nomes dos símbolos UML e RDBMS são necessariamente tipos de modelos, não sendo por isso necessário estar a declará-los explicitamente:

```
transformation Uml2Rdbms(in uml:UML, out rdbms:RDBMS);
```

Quando um tipo de modelo é explicitamente declarado, a sintaxe é a seguinte:

```
modeltype <modeltypeid> "<conformance>"
uses <packageid>("<uri>") where {<expressions>…};
```

Outras condições podem usar a variável `self` que está implicitamente definida no bloco where e se refere conceptualmente a uma instância do tipo do modelo (i.e., um modelo). A declaração seguinte indica que um tipo de modelo usa para a sua definição um *package* existente chamado `SimpleUml`, fornecendo a sua URI. A segunda declaração apenas explicita uma URI e indica uma conformidade `strict`.

```
modeltype UML uses SimpleUml("http://omg.qvt-samples.SimpleUml);
modeltype RDBMS "strict" uses "http://omg.qvt-samples.SimpleRdbms";
transformation Uml2Rdbms(in uml:UML, out rdbms:RDBMS);
```

Um tipo de modelo pode ter o mesmo nome que uma definição de metamodelo. Se não existe uma definição explícita do tipo do modelo, tal é equivalente a declarar um tipo de modelo cujo o "metamodelo referido" é o dado metamodelo. Neste caso a conformidade efectiva do tipo de modelo é assumida.

```
metamodel SimpleUML { … };
metamodel SimpleRDBMS { … };
transformation Uml2Rdbms(in uml:SimpleUML, out rdbms:SimpleRDBMS);
```

`VarParameter`: Um parâmetro de variável é um conceito abstracto que é introduzido para permitir referir-se aos parâmetros da mesma forma que as variáveis são referidas, especificamente em expressões OCL. Sintacticamente, um VarParameter é um Parameter MOF que é também uma Variable.

`DirectionKind`: Um género de sentido é um tipo enumerado que contém os tipos de sentidos possíveis para os parâmetros (i.e., `in`, `out`, `inout`).

`ImportKind`: Um género de importação é um tipo enumerado que contém os valores possíveis para a semântica da importação de modelos (i.e., `access`, `extension`).

A Figura E.2 representa os conceitos que estão relacionados com a definição de operações imperativas e propriedades contextuais.

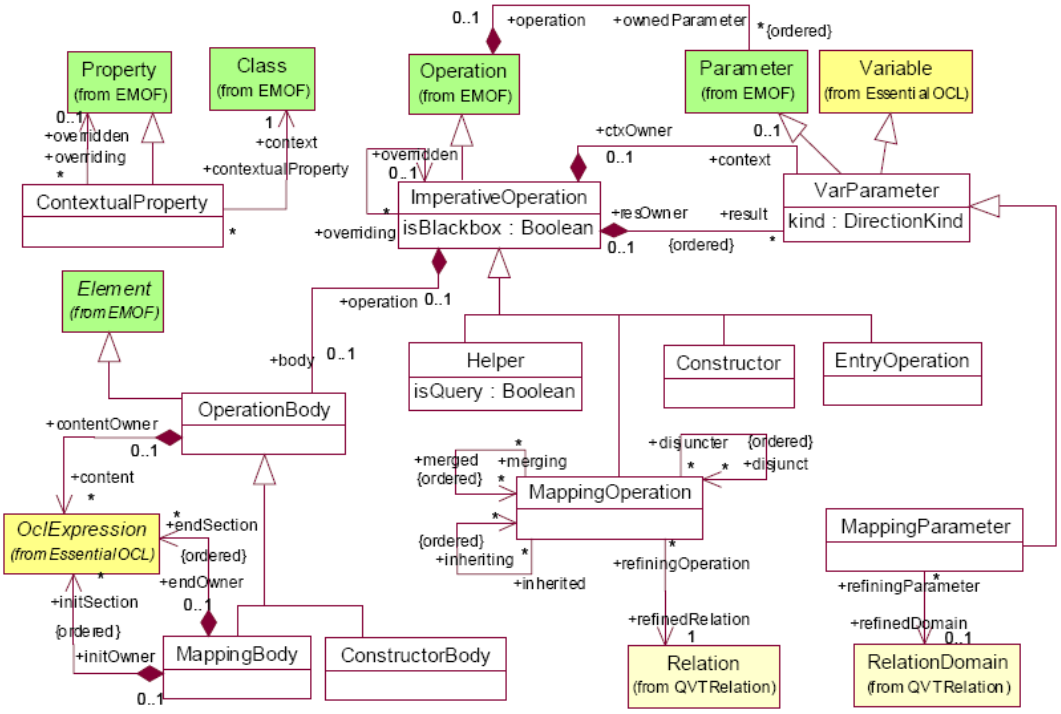


Figura E.2 – *QVT Operational Package* – Definição de características imperativas

Os conceitos mais importantes expressos na definição de características imperativas da linguagem de mapeamentos operacionais são os seguintes:

`ImperativeOperation`: Uma operação imperativa estende a noção geral de *operação MOF* com a capacidade de definir um corpo imperativo e uma assinatura mais enriquecida. A acrescentar aos parâmetros usuais de uma operação MOF, uma operação imperativa pode declarar um parâmetro de contexto e zero ou mais parâmetros de resultado. O Parâmetro de contexto, denominado self, tem um tipo (designado de tipo de contexto da operação). Estes parâmetros aplicam-se uniformemente a funções de selecção ou a operações de mapeamento. Uma operação imperativa pertence a uma transformação operacional ou a uma biblioteca. Uma operação imperativa pode sobrepor-se a uma outra definida num nível superior da árvore de herança. Neste caso, o nome da operação mais específica tem de ser o mesmo da operação genérica e a sua assinatura tem de ser compatível (i.e., têm de ter o mesmo número de parâmetros e cada um deles tem de estar comforme com o tipo do parâmetro correspondente na operação genérica). Uma operação imperativa que define um contexto na sua assinatura é denominada *operação contextual.* No âmbito de um módulo, duas operações contextuais apenas podem ter o mesmo nome se tiverem contextos diferentes. Conceptualmente uma operação contextual tem o comportamento de uma operação que estende um tipo contextual referenciado. E.g., numa definição de transformação que lida com modelos UML, pode ser necessário usar uma operação de selecção denominada `getAllAbstractBaseActors` sobre instâncias de Acto (Actor neste caso é o tipo contextual). De forma a não alterar a definição da metaclasse Actor inserindo uma nova operação (o que deve ser evitado quando se trata de metaclasses padrão), a referida operação de selecção faria parte da transformação operacional. O contexto é então usado para associar a operação de selecção à classe que está a ser estendida de uma forma lógica. I.e., realizar uma distinção explícita entre *posse* e *contexto* evita criar variantes de

metamodelos que servem apenas para determinadas transformações. Em termos de domínios de nomeação, a definição de uma operação contextual insere um novo símbolo no espaço de nomeação do módulo que tem a posse (seja transformação ou biblioteca) e, simultâneamente, insere um símbolo no espaço de nomeação da classe do contexto. Como consequência, uma operação contextual pode ser invocada ou como uma operação não contextual (`self` é o primeiro argumento da expressão chamada) ou como uma operação do contexto da classe (`self` é o objecto que recebe a chamada da operação).

`EntryOperation`: Uma *operação de entrada* é o ponto de entrada na execução da transformação. O seu corpo contém uma lista ordenada de expressões que devem ser executadas em sequência. A transformação apenas pode definir uma operação de entrada. Uma operação de entrada não tem parâmetros podendo apenas aceder a todas as propriedades ou parâmetros globais da transformação e às variáveis locais definidas por si (tal como todas as restantes operações). A notação para a operação de entrada é similar às outras operações tendo esta o nome `main`.

```
transformation UmlCleaning(inout uml:UML);
main() { uml->objectsOfType(Package)->map cleanPackage();}
```

`Helper`: Uma função é uma operação que realiza um cálculo sobre um ou mais elementos e fornece um resultado. O corpo de uma função é uma lista ordenada de expressões que são executadas em sequência. Quando mais do que um resultado é declarado na assinatura da função, a invocação da operação devolve um tuplo. A menos que a propriedade `isQuery` seja verdadeira, uma função pode ter efeitos colaterais nos parâmetros, e.g. uma lista pode ser passada e alterada no corpo da função e esse efeito pode ser visível após a execução da função. No entanto não é possível criar ou alterar objectos numa função, exceptuando para os tipos prédefinidos como conjuntos, tuplos e para propriedades intermédias. A notação é semelhante à de qualquer outra operação imperativa exceptuando que devem ser usadas as palavras `query` ou `helper` (sendo a última usada no caso em que a operação provoca efeitos colaterais nas variáveis). O corpo pode ser uma única expressão, sem usar chavetas, ou um conjunto de expressões entre chavetas. A declaração seguinte define uma função de selecção que devolve todas as classes derivadas de uma classe UML. Neste exemplo o corpo não está definido, o que significa que a função tem uma implementação caixa-preta. Esta função está definida na biblioteca UMLUtilities:

```
library UmlUtilities(UML);
query Class::getAllDerivedClasses() : Set(Class);
```

`Constructor`: Um *construtor* é uma operação que define a forma como se cria ou preenche as propriedades de uma instância de uma dada classe. Este conceito corresponde à noção de método construtor das linguagens orientadas por objectos como Java ou C#. Um construtor pode ser definido como uma operação sobre uma classe a ser construída ou pode ser pertença de um módulo que realiza o papel de *Factory*[28] da classe. Um construtor pode ser definido no âmbito de uma biblioteca de forma a poder ser reutilizado em diversas transformações. Um construtor é uma forma de

[28] Refere-se aqui o padrão *Factory* [Gamma et al, 94] onde existe uma classe que actua como construtora dos objectos de outra classe

factorizar o código necessário para criar e preencher um objecto, não sendo necessário declarar os parâmetros do resultado. O nome do construtor é usualmente o nome da classe que deve ser instanciada, não sendo, no entanto, obrigatório que assim seja. Podem existir construtores com diferentes nomes. Por outro lado, para criar um objecto não é necessário criar uma operação construtora. Para cada classe, quando não esteja definido explicitamente, existe um construtor definido por omissão, sem parâmetros e com o nome da classe. A notação para declarar construtores é similar à notação para declarar qualquer operação imperativa excepto que usa a palavra reservada `constructor` e não declara qualquer resultado. O nome do construtor é o nome do tipo do contexto e o seu corpo é colocado entre chavetas. A declaração seguinte define um construtor para a metaclasse `Column`.

```
constructor Column::Column (n:String,t: String) { name:=n; type:=t; }
```

`ContextualProperty`: Uma *propriedade contextual* é uma propriedade que é pertença de uma transformação ou de uma biblioteca mas é definida como uma extensão ao tipo referido como sendo o *contexto*. Estas propriedades são acedidas da mesma forma que quaisquer outras propriedades do contexto referido. A utilização deste tipo de propriedades é particularmente útil para definir *propriedades intermédias* como extensões a metaclasses envolvidas na transformação. Os dados intermédios são criados temporariamente por uma transformação para se realizar algum tipo de cálculo sobre eles mas não são parte do resultado esperado. A notação para as propriedades contextuais usa a palavra reservada `property`. Pode ainda ser complementada com o qualificador `intermediate` se a propriedade for definida como uma propriedade intermédia da transformação operacional.

```
intermediate property Class::leafAttributes : Sequence(LeafAttribute);
```

`MappingOperation`: Uma *operação de mapeamento* é uma operação que implementa o mapeamento entre um ou mais modelos iniciais e um ou mais modelos finais. Uma operação de mapeamento pode ser dada apenas com a sua assinatura ou também com a definição de um corpo imperativo. Quando não exista corpo para a operação, esta é denominada de *operação caixa-preta*. Este tipo de operações é útil para utilizar operações implementadas noutras linguagens de programação que de outra forma poderia ser dificil implementar em QVT (e.g. operações de análise lexical ou sintáctica). Uma operação de mapeamento define sempre uma relação onde cada domínio da relação corresponde a um parâmetro da transformação. A condição `when` actua como uma précondição ou guarda, dependendo do modo de invocação da operação de mapamento. A condição `where` actua como pós-condição para a operação de mapeamento. O corpo da operação é estruturado em três secções opcionais. A secção de inicialização é usada para os cálculos anteriores à própria instanciação dos resultados. A secção de preenchimento é usada para preencher os resultados e a secção de finalização é usada para definir cálculos finais que sejam realizados antes da saída do corpo. Existem igualmente três mecanismos de reutilização e composição associados às operações de mapeamento. Uma operação pode herdar de outra o que siginifica invocar a secção de inicialização da operação herdada depois de executar a sua própria secção de inicialização. Uma operação de mapeamento pode também fundir outras operações o que significa invocar as operações fundidas após a secção de finalização. Uma operação pode também ser definida como uma disjunção de outras operações, i.e. selecciona-se entre o conjunto de mapeamentos disjuntos o

primeiro que satisfaz a condição `when` e realiza-se a sua invocação. Em seguida define-se a semântica da execução das operações de mapeamento segundo um conjunto de casos:

Execução de uma operação de mapeamento: Uma operação de mapeamento pode declarar um parâmetro contextual, estendendo, nesse caso, o seu tipo. Realizar o mapeamento significa encontrar a operação para chamar com base no tipo do objecto que serve de fonte (variável `self`). Depois de se resolver a chamada da operação, é passado um tuplo com todos os parâmetros do mapeamento. O parâmetros incluem, nesta ordem: o parâmetro de contexto, caso exista, os parâmetros de entrada e os parâmetros de saída. Todos os parâmetros marcados com `out`, incluindo os parâmetros do resultado, têm o seu valor inicializado a `null`. Todos os valores não nulos `in` ou `inout`, excepto para os tipos primitivos, são passados por referência. No entanto não é possível alterar o valor de um objecto marcado como `in`. Depois de passar os parâmetros, o género de conformidade dos parâmetros é verificada, seguindo-se a verificação da condição `when`. Se uma destas verificações é inválida é devolvido o valor `null`. Se a condição de guarda é válida, é verificado o rasto da relação para se saber se a relação ainda se mantém. Se tal acontece, os parâmetros `out` são preenchidos usando os tuplos de rastreabilidade da relação e o valor associado aos parâmetros de resultado é devolvido. De outra forma o corpo da relação é executado em quatro fases: 1) É iniciada a execução da secção de inicialização sequencialmente (nesta secção usualmente são encontradas atribuições invocações de mapeamentos e selecções ou atribuições explícitas dos parâmetros de output); 2) No fim da secção de inicialização, é executada uma secção de instanciação implícita que provoca a instanciação e o preenchimento de todos os parâmetros `out` que são instâncias de objectos e cujo o valor ainda é `null` (o tuplo correspondente ao rasto da relação é preenchido, a relação é considerada como válida e a informação de rastreabilidade fica disponível para consulta posterior); 3) A secção de preenchimento é então executada em sequência; 4) A secção de finalização é executada em sequência, realizando-se os cálculos que devem acontecer após a devolução dos valores de saída, por parte da operação.

Execução de um mapemanto que herda de um outro mapeamento: Neste caso invoca-se primeiro a secção de inicialização, incluindo a secção de instanciação implícita, invocando-se então os mapeamentos herdados. A invocação dos mapeamentos herdados segue a semântica usual, execpto para os parâmetros out que podem iniciar as invocações com valores diferentes de null (e.g. quando um parâmetro out foi alterado pela secção de inicialização do mapeamento que herda).

Execução de um mapeamento com fusão de outros mapeamentos: Os mapeamentos fundidos são executados no fim da execução do mapeamento que os chama. Os parâmetros do mapeamento de maior nível são passados para os mapeamentos fundidos, incluindo os valores actuais para os parâmetros `out`. A conformidade dos parâmetros segue as regras de conformidade genéricas.

Execução de um mapeamento definido como disjunção de outros mapeamentos: Uma invocação de uma operação de mapeamento definida como disjunção de outra operação de mapeamento é dada em dois passos: primeiro, as condições de guarda dos mapeamentos disjuntos são executadas até que uma das condições seja válida. Se nenhuma das condições de guarda é válida, o valor null é imediatamente devolvido. Caso contrário, o corpo do mapeamento cuja condição de guarda é válida é executado. A assinatura do mapeamento com a disjunção deve estar conforme a assinatura dos

mapeamentos disjuntos. Especificamente, o resultado da disjunção tem de ser um super tipo do tipo do resultado dos mapemanetos compostos.

A forma genérica da assinatura de um mapeamento é a seguinte:

```
mapping inout <contexttype>::<mappingname> (<parameters>,) : <result-parameters>
inherits <rulerefs>, merges <rulerefs>, disjuncts <rulerefs>,
refines <rulerefs> when {<exprs>} where { <exprs>}
```

A variável <`contexttype`> só existe quando o mapeamento declara um parâmetro contextual. Para os parâmetros resultado o sentido é necessáriamente `out` e não é explicitado. Para todos os outros parâmetros, o valor dado por omissão é `in`. A declaração seguinte é um exemplo de operação de mapeamento que define um parâmetro contextual (do tipo `Package`), e uma condição de guarda. Neste caso o mapeamento é com uma implementação em caixa-preta na medida em que o corpo não é visível.

```
mapping Package::packageToSchema() : Schema
when { self.name.startingWith() <> "_"};
```

`MappingParameter`: É um parâmetro de uma operação de mapeamento, tendo um tipo de sentido que restringe as operações que podem ser realizadas sobre ele quando a operação é invocada. Os possíveis valores para os tipos de sentido são: `in`, `inout` e `out`.

`OperationBody`: Um corpo de operação contém a implementação da operação imperativa, a qual é feita de uma lista ordenada de expressões que são executadas em sequência. No corpo da operação está implícito um âmbito que é contido no âmbito da definição da operação. As variáveis e parâmetros definidos nos âmbitos mais gerais estão acessíveis neste corpo. No âmbito do corpo da operação a variável `self` representa o parâmetro contextual, caso exista, e a variável `result` representa o parâmetro a ser devolvido, o qual é um tuplo quando diversos resultados são simultâneamente devolvidos.

`ConstructorBody`: Em contraste com a notação geral para as operações, no corpo do contrutor a variável que representa o objecto instanciado pode ser omitida ao referir-se às propriedades. Na declaração seguinte é explicitado um construtor para a classe Message que define dois atributos `name` e `type`:

```
constructor Message::Message(messName:String,messType:String) {
name := messageName; // same as result.name := messageName
type := messType:String; // same as result.type := messType
}
```

MappingBody: Define a estrutura do corpo de uma operação de mapeamento. É uma classe que especifica a classe OperationsBody. A notação genérica é a seguinte:

```
mapping <mapping_signature> // see MappingOperation description
{
   init { … } // init section
   population { … } // population section
   end { … } // end section
}
```

Em muitos casos algumas destas secções podem não existir. A regra para interpretar um corpo que não tenha a secção *population* é a seguinte: 1) Se a operação de mapeamento define um resultado

único, a lista de expressões no corpo é a lista de expressões na expressão do objecto implícita; 2) Se a operação de mapeamento define mais do que um resultado, a lista de expressões no corpo é a lista de expressões da secção de preenchimento (que não existe explicitamente). Esta convenção facilita a escrita de especificações concisas na medida em que a situação em que ocorre apenas um valor de saída é muito comum. Por vezes, pode ser necessário usar explicitamente a palavra *population*, e.g. para alterar parâmetros inout. De acordo com as regras anteriores a declaração:

```
mapping A::AtoB() : B {
   init { … }
   myprop1 := … ;
   myprop2 := …;
}
```

é equivalente a:

```
mapping A::AtoB() : B {
   init { … }
   population {
      object result:B {
         myprop1 := … ;
         myprop2 := …;
      };
   }
}
```

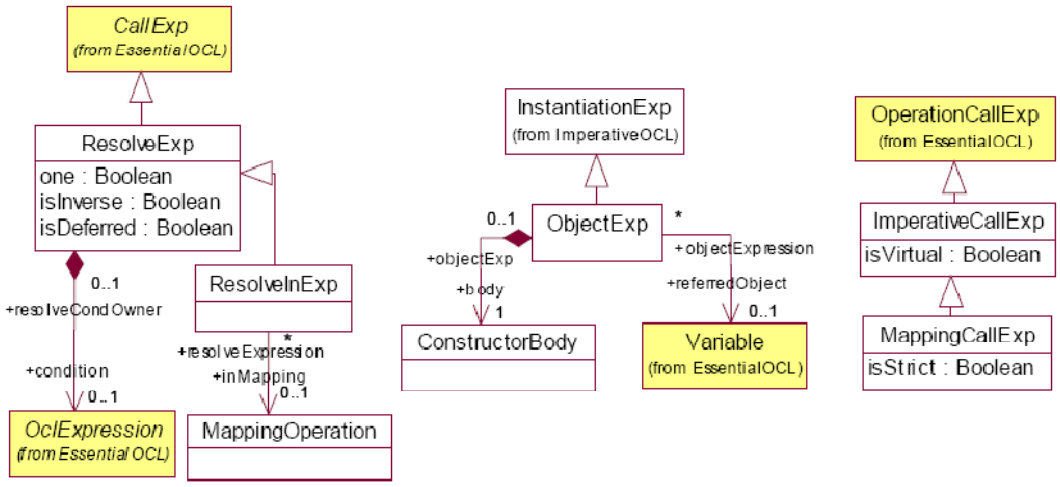


Figura E.3 – QVT Operational Package – Utilização de operações imperativas

Os conceitos mais relevantes na utilização de operações de mapeamento são os seguintes:

`ImperativeCallExp`: Uma *expressão de chamada imperativa* representa a invocação de uma qualquer operação imperativa. A menos que o atributo `isScoped` seja verdadeiro, esta invocação é virtual, i.e. a chamada da operação depende do tipo do parâmetro contextual (da mesma forma que nas linguagens Java, C++ ou C#). Uma chamada imperativa é qualquer chamada a uma operação onde o objecto receptor pode não ser dado explicitamente. A sintaxe genérica é:

```
<operationreference> ( <arg1>, <arg2>, ..., <argN> ) or
<source>.<operationreference> ( <arg1>, <arg2>, ..., <argN> ) or
<source>-><operationreference> ( <arg1>, <arg2>, ..., <argN> )
```

A terceira forma é usada para operações sobre colecções. Quando exista ambiguidade é necessário qualificar os nomes das operações com o contexto a que pertencem (e.g. quando dois módulos `accessed` definem operações com o mesmo nome).

`MappingCallExp`: Uma *expressão de chamada de mapeamento* representa a invocação de uma operação de mapeamento. Uma operação de mapeamento pode ser invocada tanto em modo *strict* como em modo *standard*, dependendo do valor da propriedade booleana `strict`. Em modo *strict* a condição `when` é avaliada como uma pré-condição, i.e., provoca o lançamento de uma excepção se é avaliada como falsa. Em contraste, quando o mapeamento é invocado em modo *standard*, sendo a condição `when` avaliada como falsa, a execução do corpo do mapeamento é ignorada e é devolvido o valor `null` ao contexto de chamada da operação. Uma operação de mapeamento é explicitada tal como qualquer outra operação imperativa, utilizando-se neste caso as palavras reservadas `map` ou `xmap`. A segunda palavra chave é usada quando a propriedade `strict` é verdadeira. Se o mapemaneto invocado define um parâmetro contextual a notação de chamada necessita de um objecto receptor:

```
// for a mapping defined with a contextual signature:
// Class::class2table() : Table
myumlclass.map class2table(); // invocation with non strict semantics
myumlclass.xmap class2table(); // invocation with strict semantics

// for a mapping defined with a non-contextual signature:
// attr2Column(Attribute) : Table
map attr2column(myattr); // invocation with non strict semantics
xmap attr2column(myattr); // invocation with strict semantics
```

As palavras reservadas map e xmap podem ser chamadas sobre uma lista e terem, se necessário, a variável iteradora entre parentesis. O iterador pode estar definido explicita ou implicitamente.

```
self.ownedElement->map class2table();
// shorthand of self.ownedElement->xcollect(i) i.map class2table();
// the iterator is implicit

self.ownedElement[#Class]->xmap(i) i.class2table();
// the iterator variable is explicitly passed in
// parentheses of xmap keyword
```

É sempre possível invocar uma operação de mapeamento usando uma referência para uma instância de uma transformação como receptor da chamada.

```
// for a mapping defined with a non-contextual signature:
// attr2Column(Attribute) : Table
this.map attr2column(myattr);
// the this keyword refers to the current transformation instance
```

Quando a operação de mapeamento tem um tipo de contexto, o argumento do contexto é passado como primeiro argumento. Se o mapeamento declarar parâmetros adicionais, os argumentos correspondentes são passados após o argumento do contexto.

```
// for a mapping defined with a contextual signature:
// Class::class2table() : Table
this.map class2table(myumlclass);
// equivalent to myumlclass.map class2table()
```

Quando a operação de mapeamento chamada não é invocada no âmbito da transformação actual (a instância de transformação dada pela variável `this`) a única forma de realizar essa chamada é passar uma referência à instância da transformação como receptor da chamada. O exemplo seguinte demonstra esta situação: `cleaningTransf` é uma instância da transformação que foi importada. Uma chamada explícita ao mapeamento `removeDup` é feita por intermédio desta instância.

```
transformation Uml2Java(in uml:UML,out java:JAVA)
access transformation UmlCleaning(UML);
mapping UmlCleaning::Class::removeDups(); // declaring the signature of
                                          // an imported mapping
main () {
cleaningTransf = UmlCleaning(uml);        // instantiating the imported
                                          // transformation
                            // first pass: cleaning the UML classes
uml->objectsOfType(Class)   // invoking the imported transformation
     ->forEach (cl) cleaningTransf.map removeDups(cl);
                            // second pass: transforming all UML classes
uml->objectsOfType(Class)->forEach (cl)
cl.map umlclass2javaclass ();
                       // equivalent to: this.map umlclass2javaclass(cl)
}
mapping UML::Class::umlclass2javaclass(): JAVA::Class { … }
```

`ResolveExp`: Uma *expressão de resolução* é uma expressão que inspecciona os objectos de rastreabilidade da transformação de forma a devolver os objectos alvo criados ou alterados pelas invocações da operação de mapeamento executadas previamente nos objectos fonte. Conceptualmente, para cada invocação de um mapeamento, a transformação regista a correspondência entre os objectos iniciais e finais que participam na invocação do mapeamento. Uma expressão de resolução tem uma expressão condicional que é usada para filtrar os objectos alvo. Existem diversas variantes para esta sintaxe: 1) Em vez de se seleccionar todos os objectos alvo que satisfazem a condição, é possível devolver apenas o primeiro elemento em que tal acontece; 2) Em vez de se seleccionar objectos criados ou alterados, é pedida a operação inversa, i.e., seleccionar todos os objectos iniciais responsáveis pela alteração ou criação de um determinado objecto alvo; 3) É possível invocar a expressão em modo diferido, i.e. a selecção dos alvos é feita apenas no fim da execução da transformação. A informação de rastreabilidade para uma invocação de uma operação de mapeamento é criada depois da execução da secção de inicialização. Esta informação contém um tuplo que guarda uma referência à operação de mapeamento (ou, o que é equivalente, uma referência à relação correspondente) e o valor de cada parâmetro, incluindo variáveis de contexto e variáveis de resultado. A especificação do QVT garante que a informação de rastreabilidade tem de conhecer o tipo de sentido de cada parâmetro (`in`, `out` e `inout`), bem como as posições das variáveis e a sua relação com os valores envolvidos. Contudo, não existe na especificação uma referência ao facto de a própria definição das operações de mapeamento poder mudar (e.g. com a inclusão de um novo parâmetro). Neste caso seria necessário guardar em tempo de execução uma referência à versão da assinatura da operação que desencadeou a alteração ou criação dos objectos.

Anexo F – Sintaxe Textual Concreta do QVT

A linguagem textual de relações do QVT tem a seguinte gramática expressa através de uma BNF:

```
<topLevel> ::= ('import' <filename> ';' )* <transformation>*
<filename> ::= <identifier>
<transformation> ::= 'transformation' <identifier> '('
                     <modelDecl> (; <modelDecl>)* ')'
                     ['extends' <identifier> (',' <identifier>)* ]
                     '{'
                        <keyDecl>* ( <relation> | <query> )*
                     '}'
<modelDecl> ::= <modelId> ':' <metaModelId> (, <metaModelId>)*
<modelId> ::= <identifier>
<metaModelId> ::= <identifier>
<keyDecl> ::= 'key' <classId> '{' <propertyId> (, <propertyId>)* '}' ';'
<classId> ::= <identifier>
<propertyId> ::= <identifier>
<relation> ::= ['top'] 'relation' <identifier> ['overrides' <identifier>]
                '{'
                <varDeclaration>*
                (<domain> | <primitiveTypeDomain>)+
                <when>? <where>?
                '}'
<varDeclaration> ::= <identifier> (, <identifier>)* ':' <typeCS> ';'
<domain> ::= [<checkEnforceQualifier>]
             'domain' <modelId> [ <identifier> ] ':' <typeCS>
             '{' <propertyTemplate>* '}' [ '{' <oclExpressionCS> '}' ]
             ['implementedby' <OperationCallExpCS>]
             ';'

<primitiveTypeDomain> ::= 'primitive' 'domain' <identifier> ':' <typeCS> ';'
<checkEnforceQualifier> ::= 'checkonly' | 'enforce'
<when> ::= 'when' '{' <oclExpressionCS> '}'
<where> ::= 'where' '{' <oclExpressionCS> '}'
<query> ::= 'query' <pathNameCS> '(' [<paramDeclaration> (','
            <paramDeclaration>)*] ')' ':' [<paramDeclaration> (','
            <paramDeclaration>)*] ( ';' | '{' <oclExpressionCS> '}' )
```

A sintaxe das expressões QVT tem as seguintes extensões ao OCL:

```
<oclExpressionCS> ::= <propertyCallExpCS>
                    | <variableExpCS>
```

```
                    | <literalExpCS>
                    | <letExpCS>
                    | <ifExpCS>
                    | <template>
                    | '(' <oclExpressionCS> ')'
                    | (<oclExpressionCS> ';')*

<template> ::= <objectTemplate>  | <collectionTemplate>
objectTemplate ::= [<identifier>] ':' <typeCS> '{' <propertyTemplate>* '}'
<propertyTemplate> ::= <identifier> '=' <oclExpressionCS>
<collectionTemplate> ::= <identifier> ':' <collectionTypeIdentifierCS>
                         '(' <typeCS> ')'
                         '{'
                             <setComprehensionExpression>
                             | <memberSelectionExprCS>
                             | <oclExpressionCSList>
                             (',' <oclExpressionCSList>)*
                         '}'
<setComprehensionExpression> ::= (<identifier> | <objectTemplate>)
                                    '|' <oclExpressionCS>
<memberSelectionExprCS> ::= (<identifier> | <objectTemplate> | '_')
                            '++' (<identifier> | '_')
```

7 Anexo G – Notação Gráfica do QVT

No passado, a notação gráfica foi um dos elementos mais importantes para a aceitação do UML, permitindo aos utilizadores representar abstracções dos sistemas subjacentes de uma forma intuitiva e natural. A sintaxe da linguagem gráfica pode ser usada de duas formas: como forma de representar transformações nos diagramas UML usuais ou para representar transformações, domínios e padrões num novo tipo de diagramas (*diagrama de transformação*).

Uma relação relaciona dois ou mais padrões, sendo cada um deles um conjunto de objectos, ligações entre eles, e valores. A estrutura de um padrão pode ser indicada através de um diagrama de objectos. Assim sendo, o QVT parte deste tipo de diagramas e acrescenta algumas extensões à notação gráfica de forma a definir o diagrama de transformação. A Figura G.1 representa a relação *UML2Rel*, entre classes e atributos UML e tabelas e colunas do modelo relacional. Foi introduzido o símbolo  para representar a transformação.

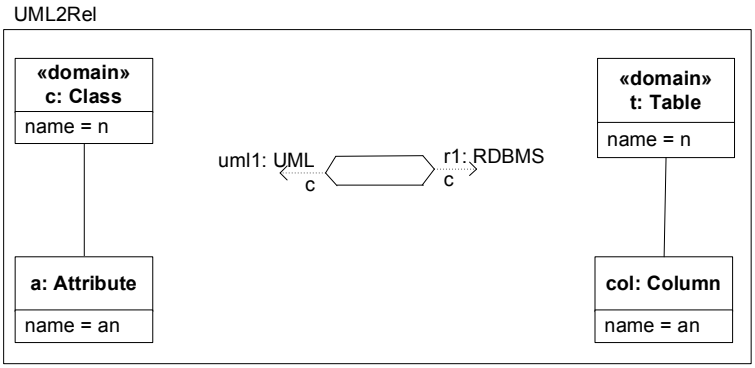


Figura G.1 – Relação QVT entre uma Classe UML e uma Tabela Relacional

As expressões "uml1: UML" e "r1: RDBMS" indicam que esta relação ocorre entre dois modelos candidatos "uml1" e "r1" cujos os *packages* "UML" e "RDBMS" representam os seus meta modelos respectivos. A letra "C" em cada extremidade da relação indicam que ambos os domínios são `checkonly`. Caso um dos domínios deva ser marcado como `enforceable` deve utilizar-se a letra maiúscula "E".

A Figura G.1 corresponde à seguinte descrição textual:

```
relation UML2Rel {
  checkonly domain uml1 c:Class {name = n, attribute = a:Attribute{name = an}}
  checkonly domain r1 t:Table {name = n, column = col:Column{name = an}}
}
```

A condição *where* de uma relação pode ser mostrada usando um rectângulo (Figura G.2) sendo o conteúdo da condição escrito no interior desse rectângulo. A mesma figura representa a relação *PackageToSchema* que estende a relação anterior (*UML2Rel*) especificando que esta deve ser aplicada em todas as classes dentro de um *package*. A condição *when*, quando necessária, pode ser indicada de uma forma similar.

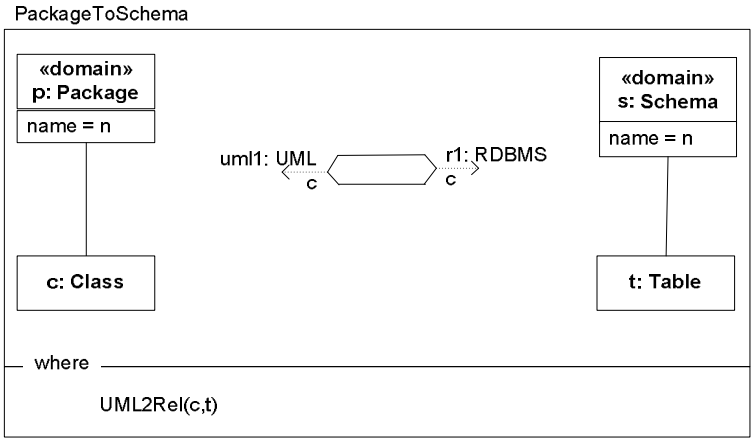


Figura G.2 – Diagrama de transformação com condição *where*

Uma restrição pode ser criada sobre um objecto ou sobre um padrão. Tal pode ser conseguido associando uma nota UML ao elemento à qual a restrição diz respeito.

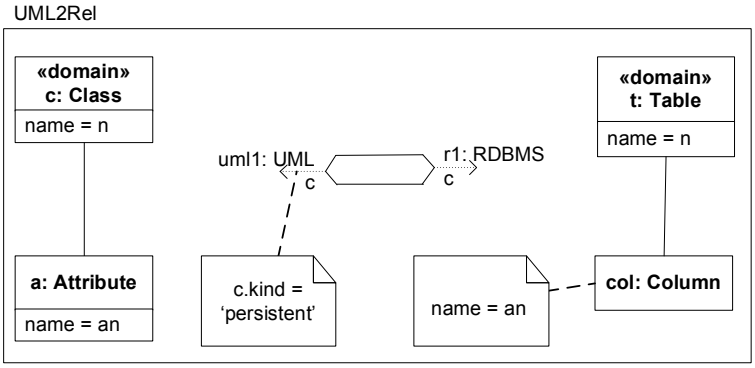


Figura G.3 – Diagrama de transformação QVT com restrições

Na Figura G.3 a restrição "name=an" que poderia estar como atributo do objecto "col: Column" foi substituída por uma nota com o mesmo valor. Ambas as formas têm o mesmo efeito prático. Neste caso em particular, a nota aumenta desnecessariamente o número de elementos gráficos, dificultando a leitura do diagrama no seu conjunto.

Nos exemplos anteriores todos os padrões são compostos de objectos individuais e associações entre eles. A notação suporta igualmente especificações envolvendo conjuntos de objectos. No diagrama de transformação da Figura G.4 os atributos da classe UML ("Class") são representados por

uma colecção. A tabela relacional ("Table") tem uma coluna que corresponde ao número de atributos da classe correspondente[29].

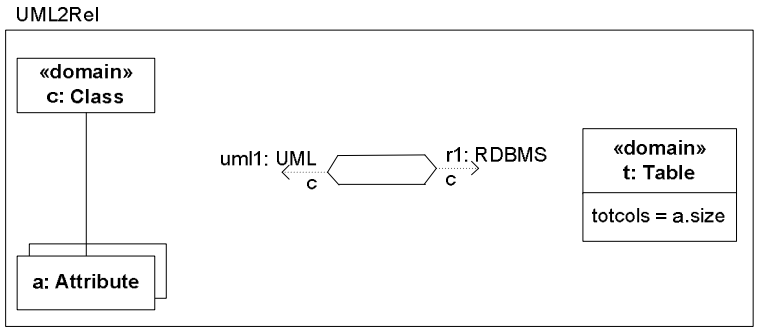
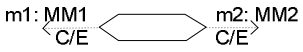
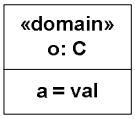
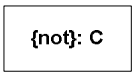
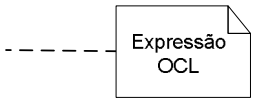


Figura G.4 – Utilização de um conjunto de elementos num diagrama de transformação QVT

O QVT inclui as seguintes convenções gráficas a serem utilizadas em diagramas de transformação:

Notação	Descrição
	Uma relação entre os modelos "m1" e "m2", tendo "MM1" e "MM2" como metamodelos, respectivamente. Pode ser escrita uma letra "C" ou "E" referente às palavras reservadas Checkeable e Enforced.
	Um template de objecto tendo o tipo "C" e sendo referido pela variável livre "o". Este objecto pertence ao domínio "domain".
	Um template de objecto tendo o tipo "C" e uma restrição que a propriedade "a" tenha o valor "val". Neste caso "val" pode ser uma qualquer expressão OCL.
	"oSet" é um template de objecto que corresponde a um conjunto de objectos do tipo "C".
	Um template not apenas é válido quando não existe um objecto do tipo "C" que satisfaça a restrição associada a ele.

[29] Tendo em vista a simplificação da explicação não foi indicada a restrição que faz com que as classes e tabelas estejam relacionadas.



Uma restrição que pode ser associada tanto a um domínio como a um template de objecto.

Anexo H – *Parsers*

Uma das acções indicadas denomina-se de análise gramatical e é levada a cabo por um módulo denominado *Parser.* Este módulo pode ser facilmente enquadrado recorrendo ao âmbito das linguagens de programação.

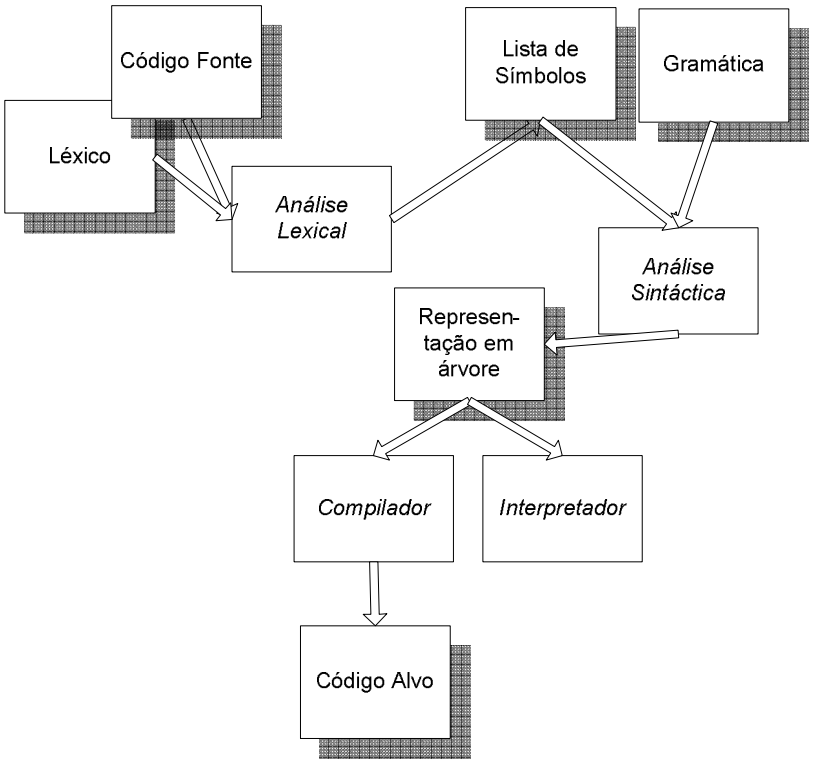


Figura H.1 – Processo de análise gramatical e síntese de um programa

Um programa é usualmente escrito através de uma linguagem de programação textual (e.g., BNF [Aho, 06]). Mesmo quando se recorre a um editor gráfico para desenhar algumas partes da aplicação, como a interface gráfica, na verdade todas as acções correspondem a alguma manipulação de texto no código do programa produzido. Para que o programa possa ser compilado ou emulado é necessário criar-se uma representação do código fonte segundo as regras da linguagem de programação. Esta representação é usualmente conseguida através de um conjunto de estruturas de dados que produzem um grafo em árvore do programa. Esta fase da compilação corresponde à análise do código fonte, sucedendo-se então a sua síntese para a linguagem alvo, eventualmente a linguagem máquina.

Nalguns casos esta representação em árvore pode servir inclusivé para o programa ser colocado a funcionar. Os programas escritos em linguagens interpretadas funcionam através de uma aplicação, o *interpretador*, que pode criar uma representação em árvore, directamente do código fonte, servindo para que sejam desencadeadas as acções pretendidas.

Existem diversos elementos neste componente que são relevantes para o caso de estudo. O primeiro elemento realiza a análise lexical do código através de uma gramática da linguagem. Nesta fase, a cadeia de caracteres que serve de input é partida em pedaços ("tokens") que são as palavras reservadas, constantes, identificadores e símbolos que estão definidos na linguagem. Após esta fase estar completa utiliza-se o segundo componente, o analisador sintáctico, para produzir uma representação em árvore do programa.

A representação em árvore é necessária para que se consiga produzir o mapeamento entre o código propriamente dito e o modelo que lhe está subjacente. É assim possível produzir as alterações necessárias aos modelos, ou gerar o código correspondente.

A implementação dos parsers de linguagens de programação foi feita recorrendo a uma representação orientada por objectos dos elementos sintácticos identificados na sintaxe (BNF) bem como à programação do tratamento dos dados e dos comportamentos respectivos.

Anexo I – Linguagem de Templates

Uma das linguagens de programação usadas durante a implementação do projecto é a Text Templating Transformation Toolkit (T4) [Gienenow, 08]. Esta linguagem utiliza uma linguagem auxiliar (C# ou VB.NET) para gerar código noutra linguagem qualquer. O código da linguagem auxiliar é inserido para realizar operações que criam valores no ficheiro resultado. O ficheiro de texto criado pode ter código noutra linguagem qualquer, visto este ser um resultado da geração automática directa, através do *template* e do modelo criado no módulo ReacT-Designer.

Nesta linguagem pode ser usada uma linguagem de programação como o C# ou o VB.NET para criar código noutra linguagem qualquer. Na verdade é criado um ficheiro de texto cuja sintaxe é definida no *template.* O template é dividido em parcelas de código executável e de texto literal. O código executável é incluido no interior de *tags* <# #>, <#= #>, <#@ #> e <#+ #>.

Dá-se em seguida um exemplo de um template criado para a geração de metamodelos a partir de um diagrama. No exemplo indicado na Figura 4.6 é definido o metamodelo de um fornecedor externo "Database".

```
<#@ template
inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation
" language = "C#" #>
<#@ output extension=".xml" #>
<#@ MetamodelLanguage processor="MetamodelLanguageDirectiveProcessor"
requires="fileName='Database.mmt'"  #>
<#@ import namespace = "System.Collections.Generic" #>
<?xml version="1.0" encoding="utf-8" ?>
<Model name="<#= this.ModelRoot.Name #>">
<#
  foreach (ModelType type in this.ModelRoot.Types)
  {
    ModelClass modelClass = type as ModelClass;

    if ((modelClass != null) && (modelClass.IsAbstract !=
MC.MetamodelLanguage.InheritanceModifier.Abstract))
    {
#>
       <ModelClass name="<#= type.Name #>"<# if (modelClass.Kind=="EntryPoint")
#> kind="EntryPoint"<# ; #>>
          <Attributes>
<#
       foreach( string s in getAttributes(modelClass))
       {#>
              <Attribute name="<#= s #>" />
<#
       }
#>
```

```
          </Attributes>
          <Operations>
<#
       foreach( string s in getOperations(modelClass))
       {#>
              <Attribute name="<#= s #>" />
<#
       }
#>
          </Operations>
<#    if (modelClass.Targets.Count > 0)
      {
#>
          <AssociationTo>
<#
           // Get links to other metaclasses
           foreach (Association association in
Association.GetLinksToTargets(modelClass))
           {
                 if ( (association.TargetMultiplicity.ToString() == "ZeroMany") |
(association.TargetMultiplicity.ToString() == "OneMany"))
                 {#>
                       <Association name="<#= association.Target.Name #>s" />
<#          }
                 else
                 {
#>
                       <Association name="<#= association.Target.Name #>" />
<#          }
           }#>
          </AssociationTo>
<#
      } #>
      </ModelClass>
<#
    }
  }
#>
</Model>

<#+

    private static System.Collections.Generic.List<String>
getAttributes(ModelClass modelClass)
    {
          List<String> list = new List<String>();
          foreach(ModelAttribute att in modelClass.Attributes)
          {
                list.Add( att.Name );
          }
          if (modelClass.Superclass != null)
                list.AddRange(getAttributes(modelClass.Superclass));
          return list;
    }

    private static System.Collections.Generic.List<String>
getOperations(ModelClass modelClass)
    {
          List<String> list = new List<String>();
          foreach(Operation att in modelClass.Operations)
          {
                list.Add( att.Name );
          }
          if (modelClass.Superclass != null)
                list.AddRange(getOperations(modelClass.Superclass));
          return list;
    }
```

```
#>
```

Após o *template* ser executado, é gerado automaticamente um ficheiro XML com os dados que definem o modelo do fornecedor externo. As relações de herança são eliminadas e todos os atributos e métodos herdados passam a fazer parte das classes respectivas. No mesmo exemplo é gerado o seguinte código:

```
<?xml version="1.0" encoding="utf-8" ?>
<Model name="Database">
       <ModelClass name="Table">
          <Attributes>
              <Attribute name="Name" />
          </Attributes>
          <Operations>
              <Attribute name="AddColumn(String columnName, String dataType)" />
              <Attribute name="RemoveColumn(String columnName)" />
              <Attribute name="SetName(string name)" />
              <Attribute name="Fill(String pathName)" />
          </Operations>
          <AssociationTo>
                       <Association name="Columns" />
          </AssociationTo>
      </ModelClass>
       <ModelClass name="Column">
          <Attributes>
              <Attribute name="Name" />
          </Attributes>
          <Operations>
              <Attribute name="SetName(string name)" />
              <Attribute name="Fill(String pathName)" />
          </Operations>
          <AssociationTo>
                       <Association name="DataType" />
          </AssociationTo>
      </ModelClass>
       <ModelClass name="Database" kind="EntryPoint">
          <Attributes>
              <Attribute name="Name" />
          </Attributes>
          <Operations>
              <Attribute name="AddTable(String tableName)" />
              <Attribute name="RemoveTable(String tableName)" />
              <Attribute name="SetName(string name)" />
              <Attribute name="Fill(String pathName)" />
          </Operations>
          <AssociationTo>
                       <Association name="Tables" />
          </AssociationTo>
      </ModelClass>
       <ModelClass name="DataType">
          <Attributes>
              <Attribute name="Name" />
          </Attributes>
          <Operations>
              <Attribute name="SetName(string name)" />
              <Attribute name="Fill(String pathName)" />
          </Operations>
      </ModelClass>
</Model>
```

Ao mesmo tempo que o código anterior é gerado, é igualmente criado um conjunto de classes C# que funcionam como estrutura básica para a implementação das funcionalidades do fornecedor externo. Assim, enquanto o código XML gerado permite documentar as estruturas de dados que são

usadas constitui igualmente uma fonte para os outros componentes (e.g., o motor de rastreabilidade) conhecerem a estrutura dos fornecedores externos criados. Sendo igualmente um resultado desta geração, os esqueletos das classes criadas são artefactos operacionais ao servirem de base para a implementação dos fornecedores externos. Reproduz-se, em seguida, o código gerado por um outro *template* de geração. Note-se que neste caso a classe representada *Database* é visível do exterior do forncedor externo tendo o atributo de compilação respectivo.

```
[Export(typeof(ReacTContracts.ProviderContract))]
public class Database : ReacTContracts.ProviderContract{

      List<String> operations = new List<String>();
      List<String> attributes = new List<String>();

      public void InvokeOperation(string name, params string[] values)
      {
          switch (name)
          {
             case "AddTable" : this.AddTable(values[0]); break;
             case "RemoveTable" : this.RemoveTable(values[0]); break;
             case "AddColumn" : this.AddColumn(values[0]); break;
             case "RemoveColumn" : this.RemoveColumn(values[0]); break;
             case "SetName" : this.SetName(values[0]); break;
             case "Fill" : this.Fill(values[0]); break;
             default: Console.WriteLine("Error"); break;
          }
      }
      public String Name { get; set; }
      void AddTable(String tableName) {
      }

      void RemoveTable(String tableName) {
      }

      void AddColumn(String tableName, String columnName, string dataType) {
      }

      void RemoveColumn(String tableName, String columnName) {
      }

      void SetName(string name) {
      }

      void Fill(String pathName) {
      }

      public List<Table> Tables { get; set; }

}
```

Índice Remissivo