Capítulo 4

Model-driven Engineering in the Development of Ubiquitous Applications: Technologies, Tools and Languages

Marcos Alves Vieira[1] e Sergio T. Carvalho[2]

[1]Instituto Federal de Educação, Ciência e Tecnologia Goiano (IF Goiano)
Iporá – GO – Brasil

[2]Instituto de Informática – Universidade Federal de Goiás (UFG)
Goiânia – GO – Brasil

marcos.vieira@ifgoiano.edu.br, sergio@inf.ufg.br

Abstract

Model-driven Engineering (MDE) is an approach that considers models as the main artifacts in software development. Models are generally built using domain specific languages, such as UML and XML. These languages, in turn, are defined by their own metamodel. In this context, this proposal aims to present a short course, with theoretical and practical approach, covering the basics of MDE as well as key frameworks and languages available for its support, aiming at the construction of ubiquitous applications. By the end of the short course, each participant will have the necessary background to be able to build a graphical modeling tool to design models in accordance with a particular metamodel. These models can then be used to document and maintain systems from different domains.

Resumo

A Engenharia Dirigida por Modelos (Model-driven Engineering - MDE) é uma abordagem que considera os modelos como os principais artefatos no desenvolvimento de um software. Modelos são geralmente construídos usando linguagens específicas de dominio, como a UML e a XML. Essas linguagens, por sua vez, são definidas por metamodelos próprios. Nesse contexto, essa proposta tem como objetivo apresentar um minicurso, com enfoque teórico e prático, abordando os fundamentos de MDE, assim como os principais frameworks e linguagens disponíveis para o seu suporte, com foco na construção de aplicações ubíquas. Ao final do minicurso, cada participante terá os conhecimentos necessários para construir uma ferramenta de modelagem gráfica que possibilite construir modelos em conformidade com um metamodelo em particular. Esses modelos podem então ser usados para documentar e manter sistemas de diferentes domínios.

4.1. Introdução

Esse capítulo tem como objetivo capacitar os seus leitores na construção de uma ferramenta de modelagem gráfica para concepção de modelos em conformidade com um determinado metamodelo. Seu público-alvo são estudantes de graduação e pós-graduação, além de profissionais da área de desenvolvimento de sistemas.

Os conceitos são apresentados diretamente por meio das ferramentas e de uma boa parcela de codificação e testes. Os leitores terão, portanto, a oportunidade de ter contato, não só com as definições e os aspectos conceituais da área, como também com o desenvolvimento passo a passo considerando um cenário-exemplo de computação ubíqua.

O capítulo está dividido em três partes:

- A primeira parte (Seção 4.2) apresenta alguns dos fundamentos teóricos necessários para a construção de um metamodelo para modelagem de espaços inteligentes:
  - Subseção 4.2.1: Computação Ubíqua
  - Subseção 4.2.2: Computação Sensível ao Contexto
  - Subseção 4.2.3: Espaços Inteligentes Fixos
  - Subseção 4.2.4: Espaços Inteligentes Pessoais
  - Subseção 4.2.5: Objetos Inteligentes
  - Subseção 4.2.6: Engenharia Dirigida por Modelos (*Model-driven Engineering* - MDE)
- A segunda parte (Seção 4.3) consiste nos fundamentos tecnológicos envolvidos:
  - Subseção 4.3.1: Eclipse Modeling Framework (EMF)
  - Subseção 4.3.2: Eclipse Graphical Modeling Framework (GMF)
  - Subseção 4.3.3: Epsilon, uma família de linguagens e ferramentas relacionadas à MDE
- Por fim, a terceira parte, apresentada na Seção 4.4, traz um tutorial alicerçado nos conceitos teóricos e tecnológicos previamente apresentados, com foco no desenvolvimento de uma ferramenta de modelagem gráfica que possibilite a construção de modelos com base em um metamodelo próprio, além de trazer um exemplo de ferramenta de modelagem construída com os conhecimentos apresentados nesse minicurso.

As considerações finais do capítulo são apresentadas na Seção 4.5.

4.2. Fundamentação Teórica

Nesta seção são apresentados os fundamentos teóricos e tecnológicos necessários para o desenvolvimento da ferramenta de modelagem gráfica.

4.2.1. Computação Ubíqua

A utilização de computadores pode ser dividida em três fases ou eras [Weiser and Brown 1997]. A primeira diz respeito à época em que o computador era um recurso escasso e atendia a diversos usuários, caracterizada como era dos *mainframes*. A segunda era corresponde aos computadores pessoais (PCs), onde cada pessoa tem acesso exclusivo a um computador. A terceira, por sua vez, representa a computação ubíqua, conforme previsto por Mark Weiser [Weiser 1991].

Weiser vislumbrou a possibilidade de tornar a utilização da computação invisível ao usuário, fundindo-a com elementos do dia-a-dia, ou seja, fazendo com que o usuário não precisasse perceber a tecnologia para aproveitar seus benefícios. Segundo este conceito, a computação estaria permeada nos objetos do ambiente físico do usuário, não requerendo dispositivos computacionais tradicionais para a interação, tais como teclado e mouse. Na computação ubíqua, o foco do usuário sai do dispositivo computacional que ele manipula e passa para a tarefa ou para a ação a ser realizada.

É de suma importância que um ambiente ubíquo ofereça suporte à mobilidade, isto é, manter disponíveis os recursos computacionais do usuário enquanto este se move de um lugar para outro [Lupiana et al. 2009]. Contudo, essa mobilidade requer uma infraestrutura de rede bem definida e uma série de outros recursos, tais como meios de obter a localização exata do usuário e também de possibilitar a descoberta e o uso dos serviços disponíveis.

Pesquisadores por todo o globo têm se sentido atraídos pela computação ubíqua. O Comitê de Pesquisa em Computação do Reino Unido (*UK Computing Research Committee* - UKCRC), por exemplo, identificou na área alguns dos grandes desafios da computação para as próximas décadas [Kavanagh and Hall 2008].

Sistemas de computação ubíqua são encontrados em diversos domínios. Um sistema ubíquo para assistência domiciliar à saúde (também conhecida como *homecare*), por exemplo, pode ser utilizado para identificar as atividades físicas cotidianas de um indivíduo ou auxiliar no tratamento de pessoas, indicando os horários para se tomar as medicações (*e.g.*, [Carvalho et al. 2011, Sztajnberg et al. 2009]), entre outras funções relacionadas ao cuidado com a saúde. Sistemas ubíquos para aprendizagem podem possibilitar a formação de grupos de alunos para trabalhar em conjunto na solução de um determinado problema proposto pelo professor, cada um utilizando seu próprio dispositivo móvel pessoal (*e.g.*, [Yau et al. 2003]), ou facilitar a participação em salas de aula virtuais por meio de reconhecimento de voz e gestos (*e.g.*, [Shi et al. 2003]).

A computação ubíqua necessita se apoiar em outros conceitos para que possa ser implementada em sua plenitude, como a computação sensível ao contexto, espaços inteligentes e objetos inteligentes. Estes conceitos são apresentados nas subseções seguintes.

4.2.2. Computação Sensível ao Contexto

Uma aplicação ubíqua deve ser minimamente intrusiva, o que exige um certo conhecimento de seu contexto de execução, isto é, deve ser possível para a aplicação obter informações sobre o estado dos usuários e do ambiente de execução, possibilitando que ela modifique seu comportamento com base nessas informações [Erthal 2014].

Neste trabalho o conceito de aplicação ubíqua utiliza uma adaptação daquele utilizado por [Roriz Junior 2013].

Uma aplicação ubíqua é um conjunto de instâncias de uma mesma aplicação em um cenário de computação ubíqua.

O termo "sensibilidade ao contexto" foi mencionado pela primeira vez por Schilit e Theimer [Schilit and Theimer 1994], como um software que "se adapta de acordo com a sua localização de uso, o conjunto de pessoas e os objetos próximos, assim como as mudanças que esses objetos sofrem no decorrer do tempo". Chagas *et al.* [Chagas et al. 2010] afirmam que a sensibilidade ao contexto permite "utilizar informações relevantes sobre entidades do ambiente para facilitar a interação entre usuários e aplicações".

As mudanças ocorridas no modelo de um software em execução podem ser desencadeadas por modificações no ambiente físico onde este software está em execução. Como exemplo, pode-se citar a elevação da temperatura de uma sala, que faz com que o software responsável por monitorar aquele ambiente ative o aparelho de ar-condicionado. O sistema em execução, nesse caso, reage a um contexto capturado (aumento de temperatura) e modifica seu comportamento (ativação do ar-condicionado).

A definição de contexto, no escopo deste trabalho, está relacionada àquela proposta por Abowd *et al.* [Abowd et al. 1999] e Dey [Dey 2001]:

Qualquer informação que possa ser usada para caracterizar a situação de uma pessoa, lugar ou objeto relevante para a interação entre um usuário e uma aplicação, incluindo estes dois últimos.

Nesse sentido, um sistema sensível ao contexto é capaz de obter informações do seu ambiente de execução, avaliar esta informação e mudar seu comportamento de acordo com a situação [Cetina et al. 2009].

4.2.3. Espaços Inteligentes

A convergência de tecnologias móveis e da Internet, propiciada principalmente pela popularização dos dispositivos e pela facilidade de acesso por meio de conexões móveis de terceira e quarta gerações (3G e 4G), está tornando o mundo mais conectado e com acesso à Internet mais disponível. Isso, combinado com Web das Coisas (*Web of Things* - WoT) [Guinard et al. 2009] e Internet das Coisas (*Internet of Things* - IoT) [Atzori et al. 2010], traz como potencial a capacidade de interligar, monitorar e controlar remotamente diversos dispositivos cotidianos (TVs, fechaduras, interruptores de lâmpadas, alarmes residenciais, etc.), fortalecendo o conceito de computação ubíqua.

A diminuição do escopo do ambiente de computação ubíqua mitiga os problemas de integração, em particular, devido à possibilidade de prever alguns comportamentos do usuário no local. Os espaços inteligentes (*smart spaces*) utilizam essa premissa para instrumentar e projetar a infraestrutura do ambiente como meio de estabelecer serviços para habilitar a computação ubíqua em um determinado ambiente.

A computação sensível ao contexto é propiciada pelos espaços inteligentes, pois estes permitem a aquisição de conhecimento a respeito do ambiente e a adaptação dos seus participantes no sentido de se tirar melhor proveito deste ambiente [Cook and Das 2007].

Para tal, os sensores monitoram e coletam informações do ambiente físico e determinadas ações são realizadas baseadas em decisões tomadas por um mecanismo de raciocínio.

Os mecanismos de raciocínio filtram e gerenciam as grandes quantidades de informações que trafegam nos espaços inteligentes diariamente. O papel de um mecanismo de raciocínio se divide em duas frentes: *(i)* modelar as informações coletadas em conhecimento abstrato útil; e *(ii)* raciocinar sobre esse conhecimento para apoiar de maneira eficaz as atividades diárias dos usuários [Cook and Das 2007].

Os espaços inteligentes estendem a computação para os ambientes físicos e permitem que diferentes dispositivos forneçam suporte coordenado aos usuários baseado em suas preferências e na atual situação do ambiente físico (contexto) [Smirnov et al. 2013]. Em outras palavras, em um nível de abstração bastante elevado, os espaços inteligentes podem ser tidos como ambientes de computação ubíqua que entendem e reagem às necessidades humanas [Lupiana et al. 2009].

Lupiana *et al.* [Lupiana et al. 2009] analisaram as diversas definições de espaços inteligentes, dadas por diferentes autores, para propor uma definição mais abrangente:

Um ambiente computacional e sensorial altamente integrado que efetivamente raciocine sobre os contextos físico e de usuário do espaço para agir transparentemente com base em desejos humanos.

Os autores, em seguida, fornecem mais detalhes sobre esta definição, argumentando que um ambiente é *altamente integrado* quando este é saturado com dispositivos de computação ubíqua e sensores completamente integrados com redes sem fio; o *raciocínio efetivo* pode ser atingido por um mecanismo pseudo-inteligente para o ambiente como um todo, e não somente para dispositivos ou componentes individuais; *contexto de usuário* refere-se a perfis individuais, políticas, localização atual e *status* de mobilidade; e *transparência* está relacionada com as ações humanas e com o suporte à mobilidade sem que, para isso, seja necessária a interação direta do usuário.

Dada a vasta quantidade de informação que pode trafegar em um espaço inteligente, os numerosos objetos inteligentes que o compõem, e a possibilidade da entrada e saída de usuários, que carregam consigo seus dispositivos móveis, a segurança é um dos aspectos críticos desses ambientes. Em [Al-Muhtadi et al. 2003], os autores elencam uma série de requisitos de segurança com os quais os espaços inteligentes devem se preocupar, incluindo:

- Acesso multinível, ou seja, disponibilizar diferentes níveis de acesso de acordo com políticas pré-definidas, com a situação atual do espaço inteligente e com os recursos disponíveis.
- Uma política de acesso descritiva, bem definida, flexível e de fácil configuração.
- A autenticação dos usuários humanos, bem como das aplicações e dos dispositivos móveis que entram e saem do espaço inteligente.

Outro aspecto importante a ser considerado em um espaço inteligente é a mobilidade do usuário. Os espaços inteligentes devem oferecer suporte à mobilidade e integrar

técnicas e dispositivos computacionais para possibilitar a interação do usuário com o ambiente de maneira transparente e intuitiva [Lupiana et al. 2009].

A pesquisa na área de espaços inteligentes é bastante pujante. Há mais de uma década os pesquisadores vêm propondo soluções para a área, como é o caso do trabalho de Johanson e Fox [Johanson and Fox 2002], que propõem o Event Heap, um espaço de trabalho colaborativo que estende o TSpaces, espaço de tuplas apresentado pela IBM Research [Wyckoff et al. 1998], no qual diferentes aplicações podem publicar e receber eventos de seu interesse.

O trabalho de Coen *et al.* [Coen et al. 2000] apresenta a Metaglue, uma linguagem de programação baseada em Java, desenvolvida nos laboratórios de inteligência artificial do Instituto de Tecnologia de Massachusetts (MIT). O objetivo da Metaglue é permitir a construção de aplicações para espaços inteligentes, oferecendo suporte a uma série de necessidades específicas deste tipo de aplicação, como, por exemplo, interconectar e gerenciar hardware e software heterogêneos, adicionar, remover, modificar e atualizar componentes em um sistema em tempo de execução, bem como controlar a alocação de recursos.

Mais recentemente, os trabalhos na área de espaços inteligentes que têm chamado atenção são as iniciativas de código-fonte aberto: openHAB[1] e Smart-M3[2].

O projeto openHAB (*open Home Automation Bus*) visa fornecer uma plataforma de integração universal para automação residencial. O openHAB é uma solução Java, completamente baseada em OSGi[3], o que facilita seu objetivo de possibilitar a interconexão de hardware de diversos fornecedores, mesmo que estes utilizem diferentes protocolos de comunicação. Smart-M3 é uma plataforma que oferece às aplicações distribuídas uma visão compartilhada das informações e serviços presentes em ambientes ubíquos, baseada no modelo arquitetural de quadro-negro (*blackboard*). Os espaços inteligentes são tidos como provedores de informações e as aplicações que compõem os espaços inteligentes, por sua vez, produzem e consomem informações.

Os espaços inteligentes tradicionais discutidos nesta subseção são referenciados neste trabalho como *espaços inteligentes fixos*. Em distinção a eles, é apresentado a seguir o conceito de *espaços inteligentes pessoais*.

4.2.4. Espaços Inteligentes Pessoais

Existem diversas iniciativas com objetivo de projetar os tradicionais espaços inteligentes, ou espaços inteligentes fixos, como os pioneiros Gaia [Román et al. 2002], Aura [Sousa and Garlan 2002], Olympus [Ranganathan et al. 2005] e a casa inteligente Gator Tech [Helal et al. 2005]. Além desses, existe também uma série de outros trabalhos mais recentes (*e.g.*, [Corredor et al. 2012, Freitas et al. 2014, Honkola et al. 2010]). Entretanto, essas propostas focam em fornecer serviços em espaços confinados e geograficamente limitados, e têm uma perspectiva centrada no sistema, na qual os usuários são atores externos que não estão contidos no espaço inteligente, ou seja, os usuários estão meramente localizados nos espaços inteligentes, não fazendo parte deles [Taylor 2011]. Esta visão

[1] `http://www.openhab.org`
[2] `http://sourceforge.net/projects/smart-m3/`
[3] `http://www.osgi.org/Main/HomePage`

contrasta com o conceito original de computação calma introduzido por Mark Weiser [Weiser and Brown 1997], que tem o usuário como o núcleo da computação.

Além disso, programar apenas espaços inteligentes fixos pode levar a ilhas de ubiquidade separadas por espaços vazios, onde o suporte à computação ubíqua é limitado, pois estes não possibilitam o compartilhamento de dispositivos e serviços com outros espaços inteligentes [Crotty et al. 2009].

Em contraste com os espaços inteligentes tradicionais, que são fixos e limitados a uma determinada área lógica ou física, um espaço inteligente pessoal (*Personal Smart Space* - PSS) é formado com base nos conceitos de computação ubíqua aliados a uma rede corporal. Os diversos sensores, atuadores e dispositivos que um usuário carrega formam uma rede corporal [Chen et al. 2011, Latré et al. 2011] e esta, por sua vez, apoiada em uma infraestrutura de software projetada para esta finalidade, compõe o seu espaço inteligente pessoal [Dolinar et al. 2008].

Em seu trabalho, Korbinian *et al.* [Roussaki et al. 2008] listam as principais características de um PSS:

1. **O PSS é móvel**: ao contrário das abordagens de espaços inteligentes tradicionais, os limites físicos de um PSS se movem com o usuário. Essa funcionalidade possibilita a sua sobreposição com outros espaços inteligentes (fixos ou pessoais).

2. **O PSS tem um "dono"**: o dono de um PSS é a pessoa sobre a qual o PSS opera. Isso possibilita que o PSS seja personalizável. As preferências de um PSS podem ser levadas em consideração na resolução de conflitos, por exemplo, para definir a melhor temperatura do ar-condicionado de uma sala com base na média das preferências dos usuários presentes.

3. **O PSS deve oferecer suporte a um ambiente *ad-hoc***: um PSS deve ser capaz de operar em ambientes de redes estruturadas e também *ad-hoc*, promovendo o uso do PSS como integrador de dispositivos.

4. **O PSS deve ser capaz de se adaptar**: as aplicações dentro um PSS devem ser capazes de se adaptar à situação corrente. Além disso, o PSS deve facilitar a interação de suas aplicações com o ambiente.

5. **O PSS pode aprender a partir de interações anteriores**: ao minerar as informações armazenadas, o PSS pode detectar tendências e inferir as condições em que as mudanças de comportamento ou preferências do usuário são manifestadas. Isso permite que recomendações sejam feitas quando o PSS interage com outros PSS ou até mesmo para agir proativamente com base nas intenções do usuário.

As duas primeiras características possibilitam que o PSS acompanhe seu *dono*, estando sempre disponível e permitindo a interação com outros espaços inteligentes, sejam fixos ou mesmo pessoais [Taylor 2008].

As características quatro e cinco habilitam o autoaperfeiçoamento, um dos principais objetivos de um PSS. O autoaperfeiçoamento é o aprendizado de tendências no comportamento do usuário, possibilitando recomendações e previsões para que adaptações ao

ambiente sejam realizadas automaticamente para o usuário [Gallacher et al. 2010]. Dessa forma, a sensibilidade ao contexto é um aspecto crucial para a funcionalidade de um PSS.

Uma infinidade de fontes de contexto fornece dados que precisam ser coletados, disseminados e gerenciados de maneira eficiente, como, por exemplo, informações relacionadas à localização e atividades do usuário, padrões de movimento, temperatura, nível de ruído do ambiente, dentre outros [Roussaki et al. 2015]. Em [Roussaki et al. 2012], os autores apresentam as características de um componente de gerenciamento de contexto para espaços inteligentes pessoais, construído com base em um conjunto de requisitos específicos para PSS, tais como: *(i)* gerenciamento das fontes de contexto; *(ii)* modelagem, gerenciamento, armazenamento e processamento de informações de contexto histórico; e *(iii)* mecanismos de gerenciamento de eventos.

Uma recente proposta de modelagem de espaços inteligentes foi apresentada em [Vieira 2016] e considera o conceito de espaços inteligentes pessoais explorado pelo projeto PERSIST (*Personal Self-Improving Smart Spaces*) [Dolinar et al. 2008], o qual apresenta a visão de que os espaços inteligentes pessoais fornecem uma interface entre o usuário e os vários serviços e dispositivos disponíveis. Dessa forma, o espaço inteligente pessoal de um usuário, formado pelos objetos inteligentes (sensores, atuadores e demais dispositivos) que ele carrega consigo, pode interagir com outros espaços inteligentes, sejam estes pessoais ou fixos.

O conceito de espaços inteligentes pessoais é considerado como uma alternativa para que os serviços computacionais estejam sempre disponíveis ao usuário, independentemente de sua movimentação pelos ambientes, minimizando o problema das ilhas de ubiquidade.

4.2.5. Objetos Inteligentes

A constante miniaturização dos dispositivos computacionais e o surgimento de tecnologias como o RFID (*Radio-Frequency IDentification*) possibilitaram o rastreamento de objetos do dia-a-dia em ambientes confinados, como lojas ou depósitos.

Os objetos inteligentes (*smart objects*), são uma evolução desses "objetos rastreáveis". Nesse conceito, os objetos inteligentes são entidades físico/digitais, aumentados com capacidades de sensoriamento, processamento e possibilidade de conexão em rede [Kortuem et al. 2010]. Um objeto inteligente pode perceber seu ambiente por meio de sensores e se comunicar com outros objetos próximos. Essas capacidades permitem que os objetos inteligentes trabalhem colaborativamente para determinar o contexto e adaptar seu comportamento [Siegemund 2004].

Em contraste com as *tags* RFID, os objetos inteligentes podem "sentir", registrar e interpretar o que está acontecendo com eles mesmos e com o ambiente físico, agir por conta própria, comunicar-se com outros objetos inteligentes e trocar informações com as pessoas [Kortuem et al. 2010].

Entretanto, os objetos inteligentes não se referem apenas aos objetos do dia-a-dia não digitais. Dispositivos computacionais tradicionais, como *smartphones*, PDAs, *players* de música, etc. podem ter ou ser aumentados com tecnologias sensitivas, tais como sensores, atuadores e algoritmos de percepção [Kawsar 2009].

Kawsar [Kawsar 2009] argumenta que um objeto inteligente deve possuir certas propriedades para garantir seu pleno funcionamento e interação com demais objetos inteligentes, tais como:

- **ID única**: é essencial poder identificar unicamente os objetos inteligentes no mundo digital. A identificação pode ser o endereço da interface de rede ou o endereço no nível de aplicação, considerando o serviço de resolução de nomes apropriado.
- **Autoconsciência**: espera-se que um objeto inteligente seja capaz de saber seu estado operacional e situacional, além de ser capaz de se descrever.
- **Sociabilidade**: um objeto inteligente deve ser capaz de se comunicar com outros objetos inteligentes e entidades computacionais (*e.g.*, uma aplicação sensível ao contexto) para compartilhar sua autoconsciência.
- **Autonomia**: um objeto inteligente deve ser capaz de tomar certas ações. Essas ações podem ser tão simples como mudar seu estado operacional (*e.g.*, mudar-se do estado desligado para ligado) ou tão complexas como adaptar seu comportamento por meio de tomadas de decisão autônomas.

4.2.6. Engenharia Dirigida por Modelos

O conceito de Engenharia Dirigida por Modelos (*Model-Driven Engineering* - MDE) considera que os modelos são os principais artefatos no desenvolvimento de um sistema. Segundo esta abordagem, os modelos não servem apenas para descrever ou documentar um software, mas também para atuar no seu desenvolvimento, manutenção e operação [Schmidt 2006, Seidewitz 2003]. Um modelo é uma representação gráfica ou textual de alto nível de um sistema, onde cada um de seus elementos é uma representação virtual de um componente presente no sistema real. Os relacionamentos e as abstrações utilizadas em um modelo são descritos por um metamodelo [Völter et al. 2013].

As técnicas de MDE, tais como Desenvolvimento Dirigido por Modelos (*Model-Driven Development* - MDD) e Arquitetura Dirigida por Modelos (*Model-Driven Architecture* - MDA) propõem o uso de abstrações mais próximas do domínio do problema como uma forma de mitigar a distância semântica existente entre o problema a ser solucionado e o ferramental (software) utilizado para tal.

A ênfase na integração entre a parte tecnológica e o conhecimento específico de um determinado domínio é um importante aspecto da MDE [Favre and Nguyen 2005], e a utilização de seus princípios aumenta a qualidade dos sistemas de software, o grau de reuso e, como resultado implícito, a eficiência em seu desenvolvimento [Bézivin et al. 2005].

Dada a popularidade do uso de modelos, surgiu a necessidade de uma padronização para a construção de metamodelos e modelos. Dessa forma, o *Object Management Group* (OMG)[4] apresentou uma arquitetura de metamodelagem de quatro camadas, denominada *Meta-Object Facility* (MOF). Na MOF, cada elemento de uma camada inferior é uma instância de um elemento de uma camada superior, conforme ilustrado na Figura 4.1.

[4] `http://www.omg.org`

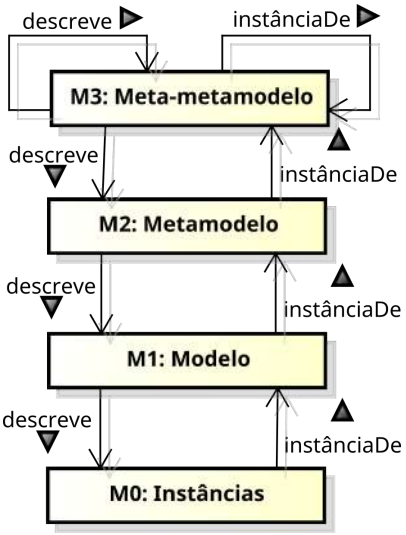

Figura 4.1. Camadas da arquitetura de metamodelagem MOF. Adaptado de [Völter et al. 2013].

As camadas MOF podem ser descritas da seguinte forma [Seidewitz 2003]:

- **Camada M3**: representa o meta-metamodelo da MOF, também chamado de Modelo MOF, utilizado para construção dos metamodelos. O modelo MOF formaliza suas próprias abstrações, eliminando a necessidade de um nível superior. Outro exemplo de membro desta camada é o Ecore, que é baseado no MOF.
- **Camada M2**: contém os metamodelos que podem ser utilizados para modelar sistemas de domínio específico. A *Unified Modeling Language* (UML) é um exemplo de membro desta camada.
- **Camada M1**: composta por modelos que descrevem sistemas utilizando as definições constantes em seus respectivos metamodelos presentes em M2.
- **Camada M0**: contém as entidades ou objetos que formam o sistema em execução, que são criadas a partir das definições presentes em M1.

Os metamodelos geralmente são construídos para permitir a criação de modelos que expressem conceitos específicos de um domínio [López-Fernández et al. 2015], como por exemplo, sistemas de distribuição de energia elétrica (*microgrids*) [Sampaio Junior 2014], *CrowdSensing* [Melo 2014], ou Linhas de Produto de Software [Carvalho 2013, Ferreira Filho 2014]. Sendo assim, um metamodelo pode ser considerado uma Linguagem de Modelagem Específica de Domínio (*Domain-Specific Modeling Language* - DSML). Uma DSML é "uma linguagem textual ou gráfica que oferece, por meio das notações e abstrações apropriadas, poder de expressividade com foco em um domínio de problema particular, para visualizar, especificar, construir e documentar artefatos de um sistema software" [Chiprianov et al. 2014, Van Deursen et al. 2000].

Como qualquer linguagem, as DSMLs possuem dois componentes principais [Ferreira Filho 2014]: sintaxe e semântica. A sintaxe de uma DSML pode ser dividida em sin-

taxe abstrata e sintaxe concreta. A sintaxe abstrata define seus conceitos e relacionamentos entre eles, enquanto a sintaxe concreta mapeia esses conceitos em elementos visuais que são usados nos modelos. A semântica de uma DSML é o significado das representações da sintaxe. A sintaxe abstrata é o componente mais importante de uma DSML. "É comum encontrar DSMLs sem definições formais de sua semântica ou sem uma representação concreta, mas sua sintaxe abstrata é imperativa" [Ferreira Filho 2014].

Em [Kolovos et al. 2006], os autores especificam uma série de requisitos gerais (de 1 a 7) e requisitos adicionais (8 e 9) para construção de linguagens específicas de domínio, a saber:

1. **Conformidade**: as construções devem corresponder a importantes conceitos do domínio.

2. **Ortogonalidade**: cada construção da linguagem deve ser usada para representar exatamente um conceito distinto do domínio.

3. **Suporte**: é interessante oferecer suporte à linguagem por meio de ferramentas para modelagem e gerenciamento de programação, *e.g.*, criação de código, edição e transformação de modelos.

4. **Integração**: a linguagem, e suas ferramentas podem ser utilizadas em conjunto com outras linguagens e ferramentas com o mínimo de esforço.

5. **Longevidade**: assume-se com esse requisito que o domínio em consideração persista por um período de tempo suficiente para justificar a construção da linguagem e suas ferramentas.

6. **Simplicidade**: uma linguagem deve ser o mais simples possível.

7. **Qualidade**: a linguagem deve fornecer mecanismos para construção de sistemas de qualidade.

8. **Escalabilidade**: a linguagem deve fornecer construções para ajudar a gerenciar descrições de larga escala, além de permitir a construção de sistemas menores.

9. **Usabilidade**: esse requisito diz respeito a conceitos como economia, acessibilidade e facilidade de compreensão. Essas características podem ser parcialmente cobertas pelos requisitos gerais, *e.g.*, simplicidade pode ajudar a promover a facilidade de compreensão.

4.3. Fundamentação Tecnológica

As subseções seguintes descrevem os conceitos tecnológicos envolvidos na construção dos metamodelos propostos e na implementação da ferramenta de modelagem gráfica.

4.3.1. *Eclipse Modeling Framework* (EMF)

O *Eclipse Modeling Framework* (EMF) [Steinberg et al. 2008] é um *framework* de modelagem construído sobre o ambiente de desenvolvimento integrado (*Integrated Development Environment* - IDE) *Eclipse*. O EMF fornece mecanismos para a criação, edição e validação de modelos e metamodelos, além de permitir a geração de código a partir dos modelos. Para tal, o EMF possibilita a geração de uma implementação em linguagem Java, de maneira que cada uma das classes do metamodelo (chamadas de metaclasses) corresponde a uma classe em Java. Dessa forma, estas classes podem ser instanciadas para criar modelos em conformidade com o metamodelo. O EMF também possibilita criar editores para modelos em conformidade com os seus metamodelos.

Os metamodelos construídos no EMF são instâncias do meta-metamodelo Ecore que, por sua vez, é baseado no meta-metamodelo MOF. Sendo assim, o Ecore é a linguagem central do EMF [Steinberg et al. 2008]. Um metamodelo com base no Ecore é definido por meio de instâncias de classes do tipo: `EClass`, `EAttribute`, `EReference`, `ESuperType` e `EDataType`. Uma `EClass` representa uma classe composta por atributos e referências. Um `EAttribute` é um atributo que possui um nome e um tipo. Uma `EReference` define uma associação entre classes e, no caso da definição de uma superclasse, para se valer dos conceitos de herança, essa associação é do tipo `ESuperType`. Por fim, um `EDataType` é o tipo de um atributo, cujo valor pode ser um tipo primitivo (número inteiro ou real, *string*, booleano, etc...), uma enumeração `EEnum` ou uma referência a uma `EClass`.

Uma visão geral dos componentes do meta-metamodelo Ecore [Merks and Sugrue 2009], com seus atributos, operações e relacionamentos, pode ser vista na Figura 4.2. A seguir é apresentado um breve detalhamento dos elementos que o compõem.

- `EClass`: modela classes. Classes são identificadas por um nome e podem conter um número de características, *i.e.*, atributos e referências. Para permitir o suporte a herança, uma classe pode referenciar outra classe como seu supertipo. A herança múltipla também é permitida e, nesse caso, diferentes classes são referenciadas como supertipos. Uma classe pode ser abstrata e, dessa forma, uma instância sua não pode ser criada. Caso uma classe seja definida como uma interface, sua implementação não é criada durante a geração de código.

- `EAttribute`: modela atributos e são identificados por um nome e possuem um tipo. Limites inferiores e superiores são especificadas para a multiplicidade do atributo.

- `EDataType`: modela tipos simples cuja estrutura não é modelada. Eles agem como empacotadores que denotam um tipo primitivo ou um tipo de objeto definido em Java. São identificados por um nome e são mais frequentemente utilizados como tipos de atributos.

- `EReference`: modela uma extremidade de uma associação entre duas classes. Elas são identificadas por um nome e um tipo, onde o tipo representa a classe na outra extremidade da associação. A bidirecionalidade é suportada ao parear uma

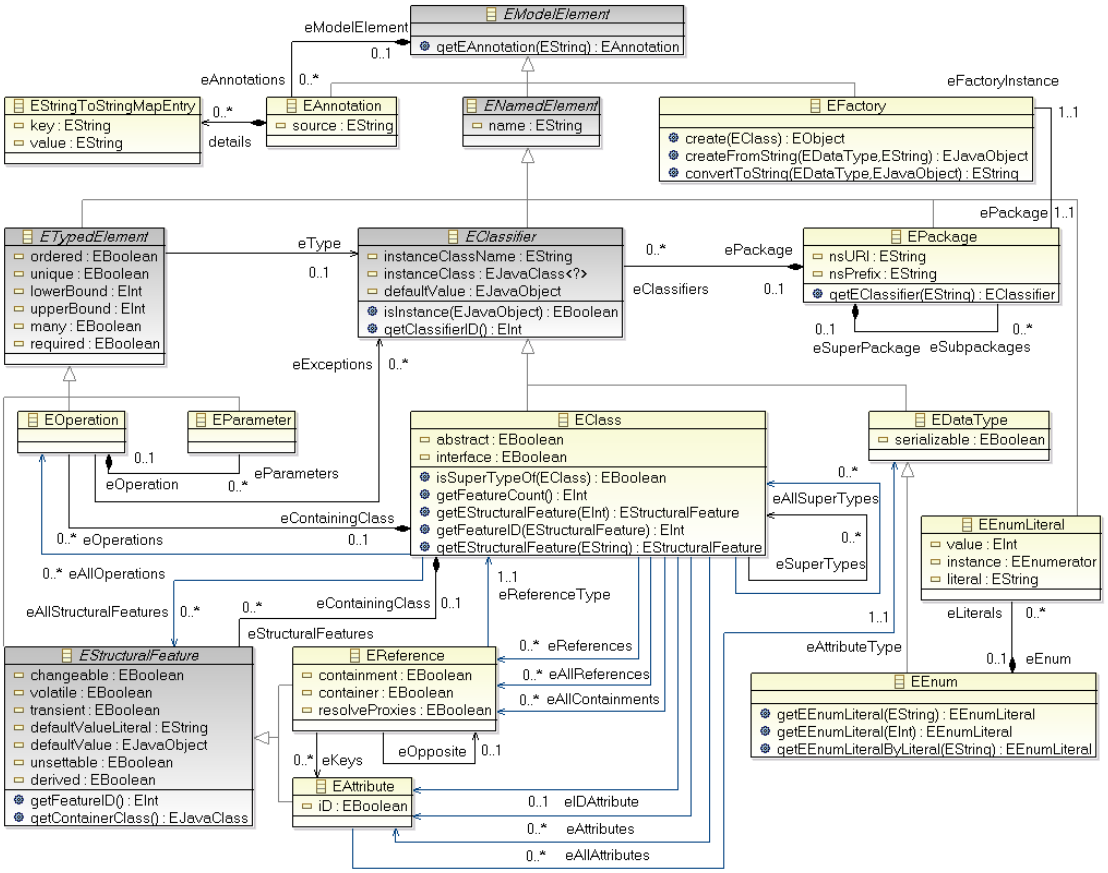

Figura 4.2. Meta-metamodelo Ecore [Merks and Sugrue 2009].

referência com seu oposto, *i.e.*, uma referência na classe representando a outra extremidade da associação. Limites inferiores e superiores são especificados na referência para denotar sua multiplicidade. Uma referência pode oferecer suporte a um tipo forte de associação, chamado *containment*, semelhante à associação do tipo "composição", da UML.

- `EModelElement`: modela os elementos de um modelo Ecore. É a raiz abstrata de uma hierarquia de classes Ecore.

- `EPackage`: modela pacotes, contêineres para classificadores, *i.e.*, classes e tipos de dados. O nome de um pacote não precisa ser único, pois seu *namespace URI* é utilizado para identificá-lo. Esse URI é usado na serialização de documentos de instâncias, juntamente com o prefixo do *namespace*, para identificar o pacote da instância.

- `EFactory`: modela fábricas para criação de instâncias de objetos. As fábricas permitem a criação de operações para instanciar classes e para converter valores de dados para *strings* e vice-versa.

- `EAnnotation`: modela anotações para associar informações adicionais a um elemento do modelo.
- `EClassifier`: modela os tipos dos valores. É a classe-base comum dos tipos de dados e para classes que servem como tipos de um elemento tipado, sendo portanto a base comum para os atributos, referências, operações e parâmetros.
- `ENamedElement`: modela elementos que são nomeados. A maior parte dos elementos no Ecore são modelos que são identificados por um nome e, portanto, estendem essa classe.
- `ETypedElement`: modela elementos que são tipados, *e.g.*, atributos, referências, parâmetros e operações. Todos os elementos tipados possuem uma multiplicidade associada especificada por seus limites inferiores (`lowerBound`) e limites superiores (`upperBound`). Um limite inferior indefinido é especificado pelo valor `-1` ou pela constante `ETypedElement.UNBOUNDED_MULTIPLICITY`.
- `EStructuralFeature`: modela as características que possuem valores em uma classe. É a classe-base comum para atributos e referências. Os seguintes atributos booleanos são usados para caracterizar atributos e referências.
  - `Changed`: indica se o valor da característica pode ser modificado.
  - `Derived`: indica se o valor da característica deve ser computado por meio de outras características relacionadas.
  - `Transient`: indica se o valor da característica é omitido da serialização persistente do objeto.
  - `Unsettable`: indica se o valor da característica tem um estado não definido, em distinção do estado poder ser definido por qualquer valor específico.
  - `Volatile`: indica se a característica não possui campo de armazenamento gerado em sua classe de implementação.
- `EOperation`: modela operações que podem ser invocadas em uma classe específica. Uma operação é definida por um nome e uma lista de zero ou mais parâmetros tipados, representando sua assinatura. Assim como todos os elementos tipados, uma operação especifica um tipo, que representa o seu tipo de retorno, que pode ser `null` para representar nenhum valor de retorno. Uma operação também pode especificar zero ou mais exceções especificadas como classificadores que podem representar os tipos de exceções que podem ser lançadas.
- `EParameter`: modela um parâmetro de entrada de uma operação. Um parâmetro é identificado por um nome e, assim como todos os elementos tipados, especifica um tipo, que representa o tipo de valor que pode ser passado como argumento correspondendo àquele parâmetro.
- `EEnumLiteral`: modela os membros do conjunto de enumeração de valores literais. Uma enumeração literal é identificada por um nome e possui um valor inteiro associado, assim como um valor literal usado durante a serialização, que é o seu próprio nome caso este valor seja *null*.

- `EEnum`: modela tipos de enumeração, os quais especificam conjuntos de enumeração de valores literais.

4.3.2. *Eclipse Graphical Modeling Framework* (GMF)

O *Eclipse Graphical Modeling Framework* (GMF)[5] é um *framework* com base no IDE *Eclipse*, que permite a construção de editores gráficos para criação de modelos que estejam em conformidade com um metamodelo específico.

O GMF requer que alguns modelos específicos sejam criados para possibilitar a geração de um editor gráfico: `GMFGraph`, `GMFTool` e `GMFMap`.

- O **modelo gráfico** (`GMFGraph`) especifica os elementos gráficos (formas, conexões, rótulos, decorações, etc.) usados no editor.
- O **modelo da ferramenta** (`GMFTool`) especifica as ferramentas para criação de elementos que estarão disponíveis na paleta do editor.
- O **modelo de mapeamento** (`GMFMap`) mapeia os elementos gráficos no modelo gráfico e as ferramentas de criação no modelo da ferramenta com a sintaxe abstrata dos elementos do metamodelo Ecore (classes, atributos, referências, etc.).

Assim que os três modelos referidos anteriormente estão criados, o GMF oferece funcionalidades para transformações M2M (modelo-para-modelo ou *model-to-model*) para criação do modelo gerador (`GMFGen`), que oferece customizações adicionais para o editor. Então, por último, transformações M2T (modelo-para-texto ou *model-to-text*) produzem um novo projeto (`.diagram`), que contém código em linguagem Java para instanciação do editor. O GMF oferece um assistente, chamado *GMF dashboard* e apresentado na Figura 4.3, para geração semi-automática das versões iniciais dos modelos necessários. A Figura 4.4 oferece uma visão geral do processo de criação destes modelos, na forma de um Diagrama de Atividades da UML.

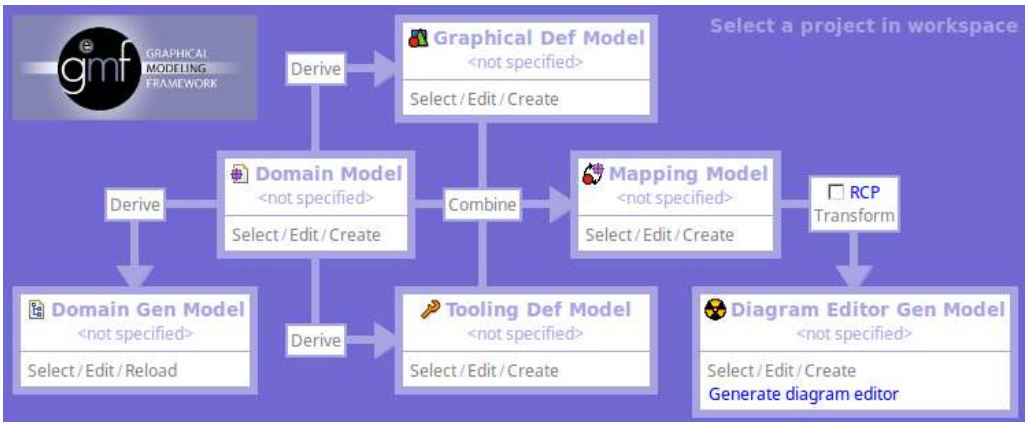

Figura 4.3. *GMF dashboard*: assistente para criação dos modelos GMF.

[5] `https://www.eclipse.org/gmf-tooling/`

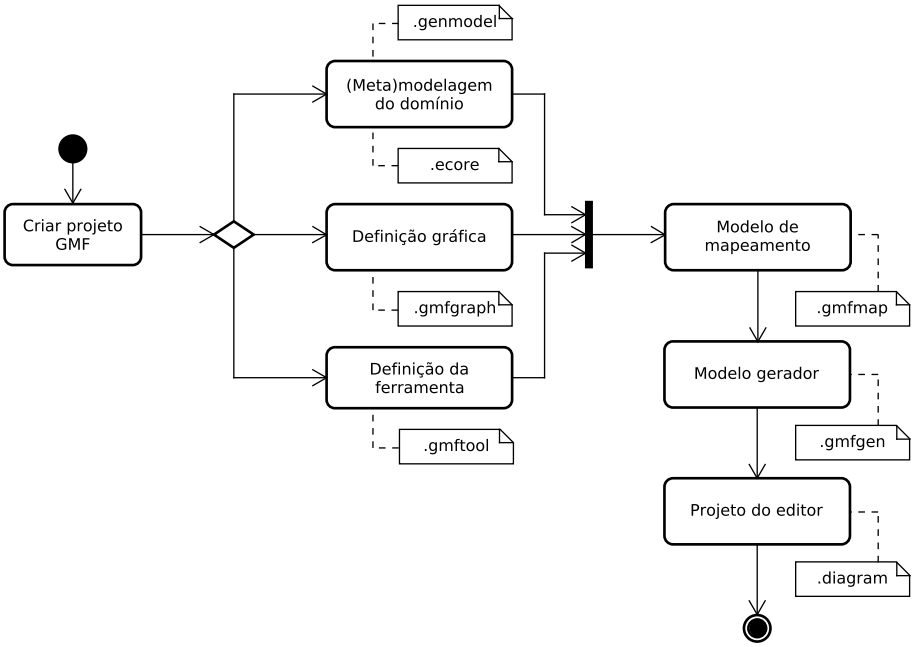

Figura 4.4. Processo de criação dos modelos GMF. Adaptado de [Eclipse Foundation, The 2015e].

Os modelos gerados podem requerer pequenos ajustes. Tais customizações são feitas editando manualmente os modelos com auxílio de um editor de textos ou do editor em árvore, ambos nativos do EMF. A Figura 4.5 apresenta a edição do modelo `GMFMap` utilizando o editor em árvore, enquanto a Figura 4.6 apresenta a edição do modelo usando o editor de texto. Além disso, a cada alteração no metamodelo Ecore, todos os modelos GMF devem ser gerados novamente, pois o GMF não oferece mecanismos para atualizar seus modelos automaticamente, diferentemente do EMF [Kolovos et al. 2015a].

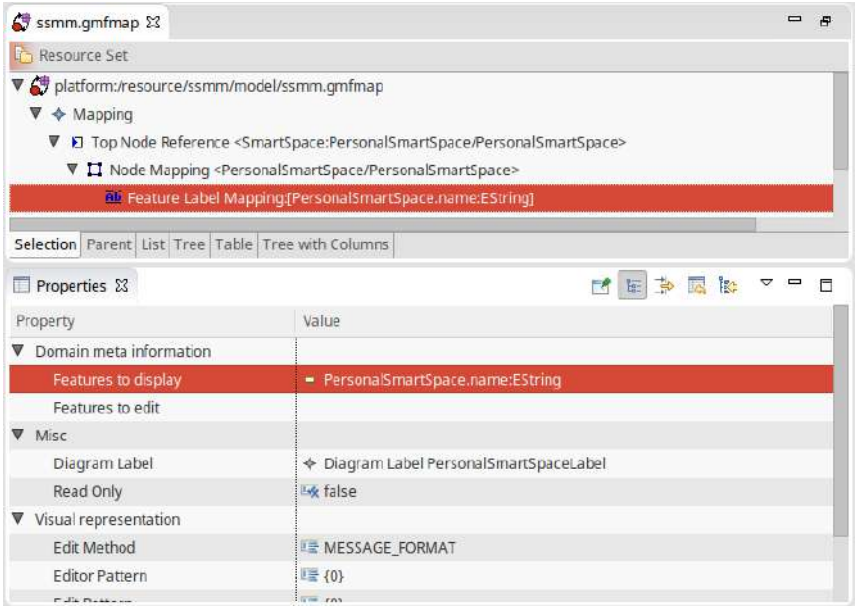

Figura 4.5. Edição do `GMFMap` usando editor em árvore.

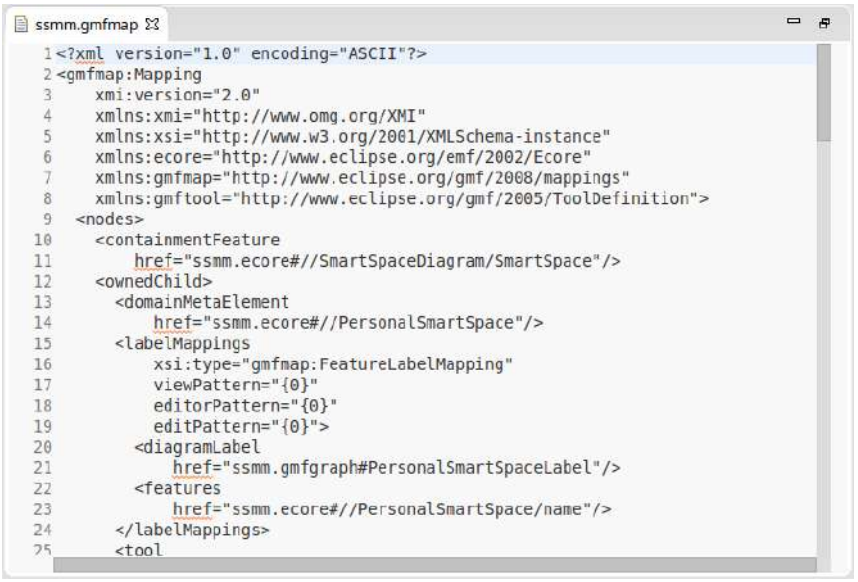

Figura 4.6. Edição do `GMFMap` usando editor de texto.

4.3.3. *Epsilon*

Epsilon[6] é o acrônimo para Plataforma Extensível de Linguagens Integradas para Gerenciamento de Modelos (em inglês, *Extensible Platform of Integrated Languages for mOdel maNagement*) [Kolovos et al. 2015b]. Trata-se de uma família de linguagens e ferramentas de suporte para atividades de manipulação de modelos, tais como, geração de código, transformação, comparação, união, refatoramento e validação de modelos.

Atualmente, as seguintes linguagens compõem a *Epsilon*:

- *Epsilon Object Language* (EOL)
- *Epsilon Validation Language* (EVL)
- *Epsilon Transformation Language* (ETL)
- *Epsilon Comparison Language* (ECL)
- *Epsilon Merging Language* (EML)
- *Epsilon Wizard Language* (EWL)
- *Epsilon Generation Language* (EGL)

Para cada uma de suas linguagens, existem ferramentas de desenvolvimento com base no IDE *Eclipse*, que oferecem funcionalidades como destaque de sintaxe e manipulação de erros, além de um interpretador para a linguagem. Os interpretadores de linguagens *Epsilon* podem inclusive ser executados de maneira independente em aplicações Java ou Android [Kolovos et al. 2015b], por meio de suas Interfaces de Programação

[6] `https://www.eclipse.org/epsilon/`

de Aplicações (*Application Programming Interface* - API)[7], possibilitando seu uso em aplicações que não foram construídas com uso do ambiente de desenvolvimento *Epsilon*.

Nas subseções seguintes, encontra-se uma breve descrição das duas primeiras linguagens (EOL e EVL), além da ferramenta *Eugenia*, que também compõe a família *Epsilon*. Essas tecnologias são utilizadas na construção da ferramenta de modelagem gráfica que é objetivo deste minicurso. Informações detalhadas sobre as linguagens *Epsilon* podem ser obtidas em [Kolovos et al. 2015b].

4.3.4. *Epsilon Object Language* - EOL

A linguagem EOL[8] é o núcleo das linguagens da família *Epsilon*. Todas as demais linguagens da família estendem a EOL, tanto em sintaxe quanto em semântica. Sendo assim, a EOL oferece um conjunto de funcionalidades sobre as quais as demais linguagens são implementadas. Além disso, a EOL pode ser usada independentemente, como uma linguagem de propósito geral para gerenciamento de modelos, automatizando tarefas que não são específicas das demais linguagens da família, tais como criar, consultar e modificar modelos EMF [Kolovos et al. 2015b].

Dentre as principais características da EOL, destacam-se [Eclipse Foundation, The 2015b]:

- Todas as construções usuais em programação, como laços `while` e `for`, variáveis, etc.
- Possibilidade de criar e realizar chamadas a métodos de objetos Java.
- Suporte para a adição dinâmica de operações em metaclasses em tempo de execução.
- Suporte a interação com o usuário.
- Possibilidade de criação de bibliotecas de operações para serem importadas e utilizadas em outras linguagens *Epsilon*.

Programas escritos em EOL são organizados em módulos (*modules*), conforme ilustrado pela Figura 4.7. Cada módulo define um corpo (*body*) e um número de operações (*operations*). O corpo é um bloco de declarações que são avaliadas quando o módulo é executado. Cada operação define o tipo de objeto sobre o qual ela é aplicável (*contexto*), um nome (*name*), um conjunto de parâmetros (*parameters*) e um tipo de retorno (*return type*), que é opcional. Módulos também podem importar (*import*) outros módulos.

Além de possibilitar a construção de operações, a linguagem EOL também oferece uma série de operações previamente definidas para cada um de seus tipos de dados, apresentados na Figura 4.8, além de permitir a utilização de tipos nativos (*native*), definidos pelo usuário em sua linguagem-base, por exemplo, uma classe Java. O Código 4.1 apresenta um exemplo de utilização da linguagem EOL.

[7] `https://www.eclipse.org/epsilon/download/`
[8] `https://www.eclipse.org/epsilon/doc/eol/`

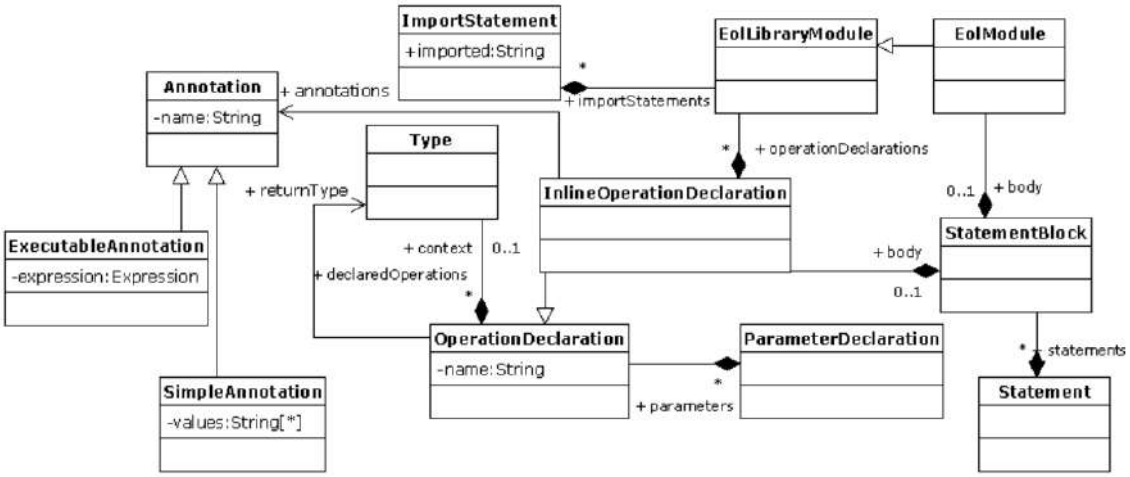

Figura 4.7. Estrutura de módulos da linguagem EOL [Kolovos et al. 2015b].

Código 4.1. Exemplo de código em linguagem EOL para calcular e imprimir a profundidade de cada árvore (`Tree`) [Kolovos et al. 2015b].

```
1  var depths = new Map;
2
3  for (n in Tree.allInstances.select(t|not t.parent.isDefined()))
       {
4      n.setDepth(0);
5  }
6
7  for (n in Tree.allInstances) {
8      (n.name + " " + depths.get(n)).println();
9  }
10
11 operation Tree setDepth(depth : Integer) {
12     depths.put(self,depth);
13     for (c in self.children) {
14         c.setDepth(depth + 1);
15     }
16 }
```

4.3.5. *Epsilon Validation Language* - EVL

O objetivo da linguagem EVL[9] é oferecer funcionalidades de validação à família *Epsilon*. Dessa forma, a linguagem EVL pode ser utilizada para especificar e avaliar restrições – também chamadas de invariantes – em modelos de um metamodelo específico. As restrições escritas em EVL são similares às restrições OCL (*Object Constraint Language*) [Object Management Group 2015], contudo, a linguagem EVL também oferece suporte a dependências entre as restrições (*e.g.*, se a restrição A falhar, então não avalie a restrição B), mensagens de erro customizáveis e a especificação de consertos (*quick fixes*) que podem ser acionados para reparar inconsistências [Eclipse Foundation, The 2015c].

[9] `https://www.eclipse.org/epsilon/doc/evl/`

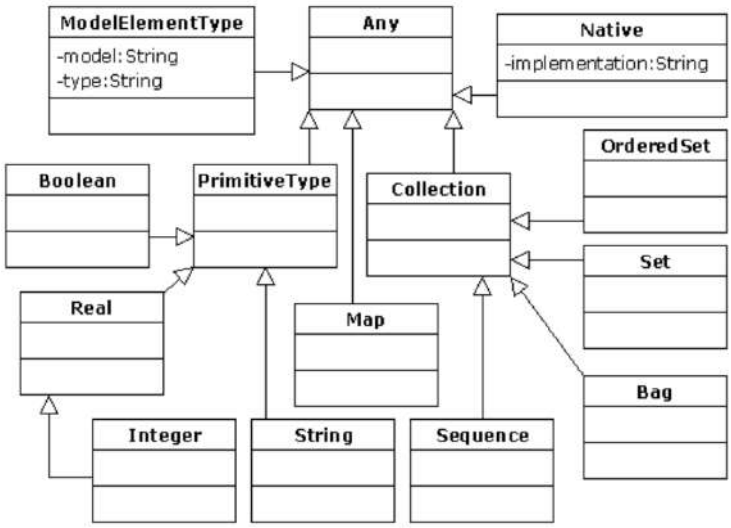

Figura 4.8. Visão geral do sistema de tipos de dados da linguagem EOL [Kolovos et al. 2015b].

As principais características da linguagem EVL são, dentre outras [Eclipse Foundation, The 2015c]:

- Distinção entre erros (*error*) e avisos (*warning*) durante a validação.
- Especificação de consertos (*quick fix*) para erros encontrados pelas restrições.
- Todas as funcionalidades oferecidas pela linguagem EOL.
- Suporte a operações lógicas de primeira ordem oriundas da OCL (`select`, `reject`, `collect`, etc.).
- Suporte a interação com usuário.

Em EVL, as especificações das validações são organizadas em módulos (`EvlModule`). Além de operações, um módulo EVL também pode conter um conjunto de invariantes agrupadas pelo contexto no qual serão aplicadas, juntamente com um número de blocos anteriores (`pre`) e posteriores (`post`). A Figura 4.9 ilustra os elementos da sintaxe abstrata da linguagem EVL, descritos a seguir.

- `Context`: um contexto especifica o tipo de instâncias sobre as quais as invariantes (`invariant`) serão avaliadas. Cada contexto pode opcionalmente definir uma guarda (`guard`) que limita sua aplicabilidade para um subconjunto menor de instâncias de um tipo especificado. Dessa forma, se uma guarda falha para uma instância de um tipo específico, nenhuma de suas invariantes é avaliada.
- `Invariant`: como ocorre em OCL, cada invariante EVL define um nome (`name`) e um corpo (`check`). Entretando, ela pode opcionalmente definir uma guarda (`guard`). Como forma de oferecer *feedback* para o usuário, cada invariante pode definir uma mensagem (`message`) que contém uma descrição da razão pela qual

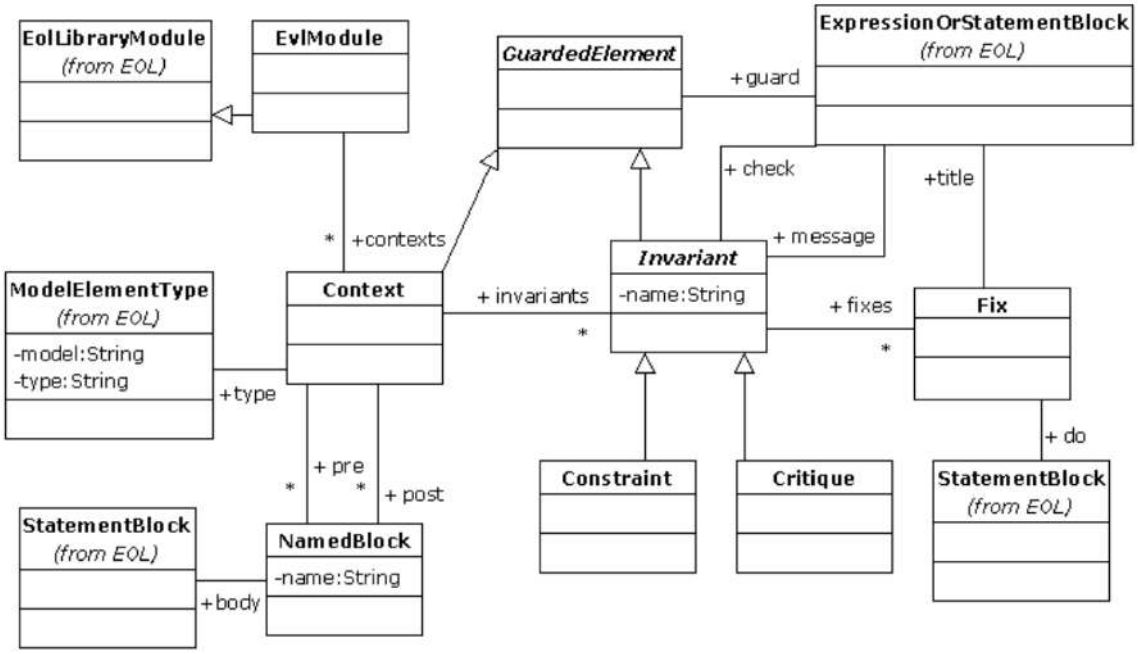

Figura 4.9. Sintaxe abstrata da linguagem EVL [Kolovos et al. 2015b].

cada restrição falhou em um elemento em particular. Para permitir conserto semi-automático de erros, uma invariante pode opcionalmente definir um ou mais consertos (`fixes`), tratados no *Eclipse* como *"quick fix"*. Cada invariante pode ser uma restrição (`constraint`) ou crítica (`critique`), permitindo lançar mensagens de erro (*error*) ou aviso (*warning*), respectivamente.

- `Guard`: guardas são usadas para limitar a aplicabilidade das invariantes.
- `Fix`: permite a correção semi-automática de erros encontrados durante a validação. É possível definir um título (`title`) e um bloco `do`, onde a funcionalidade de conserto será definida usando a linguagem EOL.
- `Constraint`: restrições em EVL são usadas para capturar erros críticos que invalidam o modelo. Restrições (`constraint`) são uma subclasse das invariantes (`invariant`), herdando, portanto, todas as suas características.
- `Critique`: em distinção às restrições, críticas (`critiques`) são usadas para capturar situações que não invalidam o modelo, mas devem ser consideradas para melhorar sua qualidade. Assim como as restrições, é uma subclasse das invariantes (`invariant`).
- `Pre` e `Post`: contém declarações EOL que podem ser executadas antes (`pre`) ou após (`post`) a avaliação das invariantes.

O Código 4.2 apresenta um exemplo de regra EVL que impede que uma metaclasse em particular possua uma referência para si mesma, sugerindo como correção (*quick fix*) apagar o autorreferenciamento encontrado, conforme ilustrado na Figura 4.12.

Código 4.2. Exemplo de código em linguagem EVL para impedir auto-referenciamento em uma determinada metaclasse.

```
1  context HasSubFSS {
2    constraint CannotSelfReference {
3      check : self.source.name <> self.target.at(0).name
4      message : 'Cannot make a self reference'
5        fix {
6          title : "Delete self reference"
7          do {
8            delete self;
9          }
10       }
11   }
12 }
```

Figura 4.10. Erro encontrado no modelo por uma regra escrita em linguagem EVL.

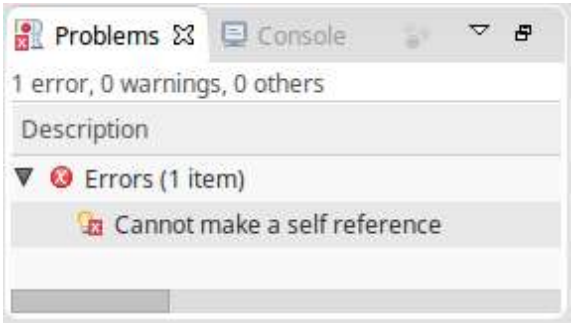

Figura 4.11. Descrição do erro.

4.3.6. *Eugenia*

Eugenia[10] é uma ferramenta da família *Epsilon* que gera automaticamente os modelos `GMFGraph`, `GMFTool` e `GMFMap`, necessários para a implementação de um editor GMF, utilizando como base em um metamodelo Ecore anotado, escrito em linguagem *Emfatic*[11]. A linguagem *Emfatic* permite a representação de metamodelos Ecore de forma textual e possui sintaxe similar à da linguagem Java [Daly and Eclipse Foundation, The 2015a, Kolovos et al. 2015a]. A IDE *Epsilon* possui funcionalidades que permitem a transformação de modelos Ecore em código *Emfatic* e vice-versa.

O principal objetivo da *Eugenia* é diminuir a complexidade na criação de ferramentas de modelagem gráficas com base no GMF, por meio de anotações de alto nível,

[10] `https://www.eclipse.org/epsilon/doc/eugenia/`
[11] `https://www.eclipse.org/epsilon/doc/articles/emfatic/`

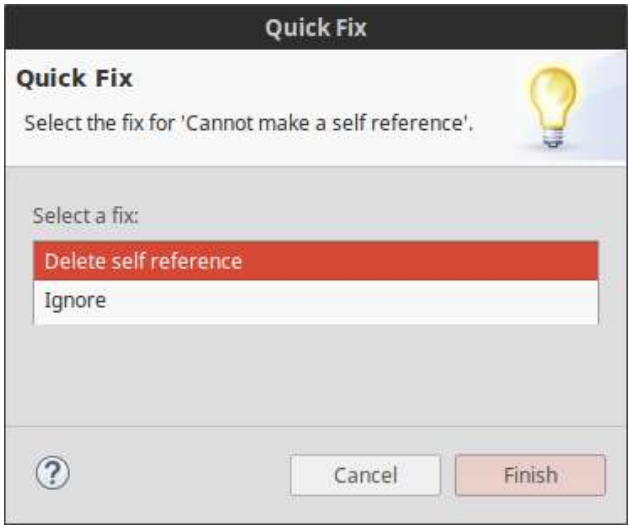

Figura 4.12. Sugestão de correção (*quick fix*).

feitas diretamente no código *Emfatic* [Daly and Eclipse Foundation, The 2015b]. Wienands e Golm [Wienands and Golm 2009] realizaram um experimento industrial e chegaram à conclusão de que implementar um editor gráfico utilizando puramente EMF e GMF é um processo propenso a erros, principalmente por requerer que os desenvolvedores escrevam e mantenham, em baixo nível, uma série de complexas relações entre os modelos GMF. Além disso, "mais desafiador do que construir um editor GMF, é sua manutenibilidade, uma vez que o GMF, ao contrário do EMF, não oferece mecanismos para atualizar seus modelos automaticamente quando o metamodelo é modificado" [Kolovos et al. 2015a].

A *Eugenia* permite transformações automatizadas entre o código *Emfatic* anotado, que representa o metamodelo, e os modelos de baixo nível necessários para geração da ferramenta GMF, aumentando o nível de abstração no qual os desenvolvedores devem trabalhar para construir suas ferramentas gráficas para edição de modelos com base em um metamodelo [Kolovos et al. 2015a]. Em suma, a *Eugenia* oferece seis categorias de anotações [Kolovos et al. 2015a], que são utilizadas nos códigos *Emfatic* para guiar a criação da ferramenta gráfica de modelagem. Os atributos suportados em cada tipo de anotação e sua respectiva lista de valores não serão aqui abordados, podendo, entretanto, ser obtidos em [Eclipse Foundation, The 2015d]. A seguir está a lista de anotações suportadas pela *Eugenia*:

- `@gmf.diagram`: usada para especificar configurações do diagrama, tais como o tipo do elemento raiz do modelo e a extensão do arquivo do editor gráfico.
- `@gmf.node`: usada para indicar quais elementos da sintaxe abstrata irão representar nós (vértices) na sintaxe gráfica, além da sua forma, cor, tamanho, rótulo, etc.
- `@gmf.link`: usada para indicar quais elementos da sintaxe abstrata irão representar arestas na sintaxe gráfica, além de especificar sua espessura, cor, estilo, tipo de pontas das setas, rótulos, etc.

- `@gmf.compartment`: são usadas para indicar quais nós podem ser aninhados dentro de outros nós na sintaxe gráfica (*e.g.*, atributos são aninhados dentro das classes em um Diagrama de Classes da UML).
- `@gmf.affixed`: são usadas para indicar quais nós devem estar anexos à borda de outros nós na sintaxe gráfica.
- `@gmf.label`: usada para especificar rótulos adicionais para um nó na sintaxe gráfica.

O Código 4.3 traz o código *Emfatic* de um metamodelo para representar sistemas de arquivos [Eclipse Foundation, The 2015d]. Por meio deste código e suas anotações, é possível gerar um editor GMF completamente funcional utilizando a ferramenta *Eugenia*, o qual é apresentado na Figura 4.13.

Código 4.3. Código em linguagem Emfatic de um metamodelo de sistemas de arquivos [Eclipse Foundation, The 2015d].

```
1  @namespace(uri="filesystem", prefix="filesystem")
2  @gmf
3  package filesystem;
4
5  @gmf.diagram
6  class Filesystem {
7      val Drive[*] drives;
8      val Sync[*] syncs;
9  }
10
11 class Drive extends Folder {
12
13 }
14
15 class Folder extends File {
16     @gmf.compartment
17     val File[*] contents;
18 }
19
20 class Shortcut extends File {
21     @gmf.link(target.decoration="arrow", style="dash")
22     ref File target;
23 }
24
25 @gmf.link(source="source", target="target", style="dot", width=
       "2")
26 class Sync {
27     ref File source;
28     ref File target;
29 }
30
31 @gmf.node(label = "name")
32 class File {
33     attr String name;
34 }
```

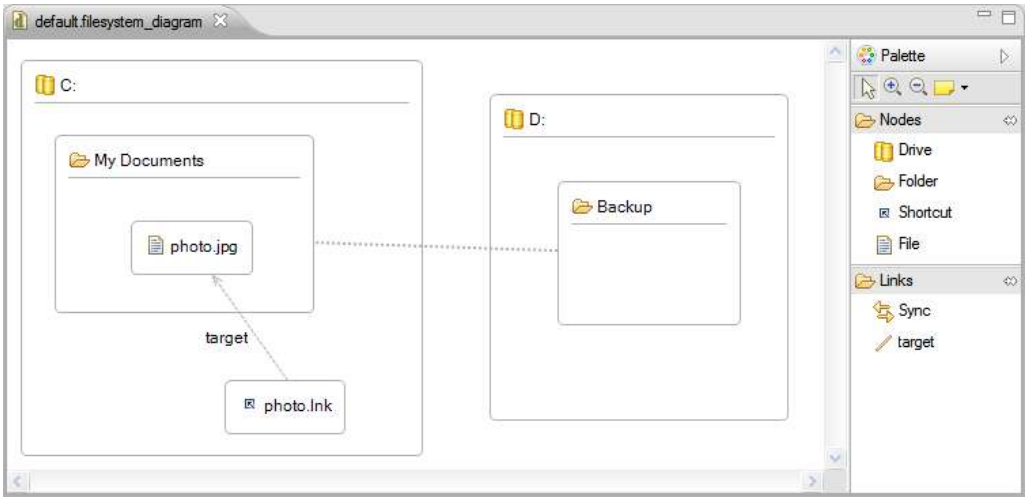

Figura 4.13. Editor GMF gerado com apoio da ferramenta Eugenia [Eclipse Foundation, The 2015d].

Por fim, a *Eugenia* oferece meios de realizar ajustes finos na aparência da ferramenta de modelagem, por meio da definição de regras de transformação em modelos independentes (*e.g.*, `ECore2GMF.eol`), utilizando a linguagem EOL. As regras desses modelos são executadas logo após a derivação dos modelos GMF pela *Eugenia*. Sendo assim, ao contrário do que ocorre quando se trabalha diretamente com GMF, não é necessário realizar os ajustes na aparência da ferramenta sempre que houver novas mudanças no metamodelo. O Código 4.4 apresenta o conteúdo do arquivo `ECore2GMF.eol`, cujo objetivo é remover as bordas das figuras nos modelos gerados em um editor GMF, o qual foi implementado com apoio da ferramenta *Eugenia*.

Código 4.4. Código em linguagem EOL (`ECore2GMF.eol`) para remover as bordas das figuras dos modelos, em um editor GMF criado com auxílio da Eugenia [Eclipse Foundation, The 2015a].

```
1 -- Find the attribute figure
2 var attributeFigure = GmfGraph!Rectangle.all.
3     selectOne(r|r.name = 'AttributeFigure');
4
5 -- ... delete its border
6 delete attributeFigure.border;
```

4.4. Tutorial para Construção de Ferramenta de Modelagem com base em um Metamodelo Ecore

Este tutorial tem objetivo de sintetizar os conhecimentos adquiridos nas seções anteriores na forma de um produto de software que consiste em uma ferramenta de modelagem para produzir modelos em conformidade com um determinado metamodelo Ecore.

Dessa forma, considera-se que o leitor tenha conhecimento prévio das tecnologias EMF, GMF e da família de linguagens *Epsilon*, apresentadas nas Subseções 4.3.1, 4.3.2 e 4.3.3, respectivamente. Será utilizado o IDE *Eclipse Neon* (versão 4.6), juntamente com

a *Epsilon* versão 1.4[12].

O tutorial está organizado da seguinte forma. Na Subseção 4.4.1 é apresentado o passo-a-passo para construção da ferramenta de modelagem gráfica. A Subseção 4.4.2 traz informações sobre como fazer refinamentos estéticos na ferramenta, a saber: *(i)* adicionar ícones para os nós, *(ii)* modificar os ícones da paleta (barra de ferramentas) e *(iii)* mudar o estilo da fonte dos rótulos. A Subseção 4.4.3 apresenta como adicionar regras de validação em linguagem EVL. A Subseção 4.4.4 traz algumas lições aprendidas para contornar comportamentos errôneos e inesperados do IDE. Por último, a Subseção 4.4.5, apresenta brevemente um exemplo de ferramenta de modelagem gráfica construída com os conhecimentos adquidiros neste minicurso.

4.4.1. Construção da Ferramenta Gráfica

Esta subseção apresenta o passo-a-passo para construção da ferramenta de modelagem gráfica.

Construção do metamodelo Ecore

1. Criar o projeto para o metamodelo Ecore
   (a) Acessar o menu "*File*" > "*New*" > "*Other...*"
   (b) Selecionar "*Ecore Modeling Project*"
   (c) Clicar em "*Next*"
   (d) Digitar um nome para o projeto e clicar em "*Finish*"
   (e) Aceitar a mudança da perspectiva, caso lhe seja sugerido
2. Construir um metamodelo válido
   (a) Construir o metamodelo
      i. Utilizar as ferramentas da paleta para construir as metaclasses e as suas relações
      ii. Dica: adicione um atributo "*name*", do tipo `EString`, para cada metaclasse que represente um nó.
   (b) Validar o metamodelo
      i. Assim que terminado, certifique-se de que seu metamodelo seja válido, acessando o menu "*Diagram*" > "*Validate*"

Conversão do arquivo Ecore (`.ecore`) para Emfatic (`.emf`)

1. Criar um novo projeto para a ferramenta

[12] `https://www.eclipse.org/epsilon/download/`

(a) Acessar o menu "*File*" > "*New*" > "*Project*"

(b) Selecionar a opção "*General*" > "*Project*"

(c) Definir nome do projeto. Exemplo: "*minhaFerramenta*"

(d) Clicar em "*Finish*"

2. Criar pasta para armazenar os (meta)modelos

   (a) Selecionar pasta raiz do projeto recém criado

   (b) Acessar o menu "*File*" > "*New*" > "*Folder*"

   (c) Nomear a pasta como "*model*"

   (d) Clicar em "*Finish*"

3. Copiar o arquivo `.ecore` do projeto do metamodelo para o projeto da ferramenta

   (a) Navegar até a pasta "*model*" no projeto do metamodelo já criado

   (b) Selecionar o arquivo `.ecore`

   (c) Acessar o menu "*Edit*" > "*Copy*"

   (d) Navegar até a pasta "*model*" no projeto da ferramenta

   (e) Acessar o menu "*Edit*" > "*Paste*"

4. Gerar arquivo Emfatic (`.emf`) a partir do Ecore (`.ecore`)

   (a) Clicar com o botão direito do mouse sobre o arquivo *.ecore*

   (b) Selecionar "*Generate Emfatic Source*"

5. Adicionar as anotações necessárias ao arquivo `.emf`

   (a) Clicar duas vezes sobre o arquivo `.emf` e adicionar as anotações necessárias

   (b) A Subseção 4.3.6 contém mais detalhes sobre as anotações disponíveis

Geração da ferramenta gráfica

1. Gerar a ferramenta de modelagem

   (a) Clicar com o botão direito do mouse sobre o arquivo `.emf`

   (b) Selecionar "*Eugenia*" > "*Generate GMF Editor*"

   (c) Aguardar a mensagem de conclusão: "*Code generation completed successfully*"

   (d) Clicar em "*OK*"

   - Ao final, o Eugenia irá criar quatro novos projetos, com terminação `.diagram`, `.edit`, `.editor` e `.tests`, além dos modelos `GenModel`, `GMFGen`, `GMFGraph`, `GMFTool` e `GMFMap`, necessários para o GMF instanciar a ferramenta de modelagem, conforme detalhado na Subseção 4.3.6.

Acessando a ferramenta de modelagem gráfica

1. Iniciar a ferramenta de modelagem
   (a) Clicar com o botão direito na pasta raiz do projeto da ferramenta
   (b) Selecionar "*Run As*" > "*Eclipse Application*"
2. Criar projeto na ferramenta de modelagem para armazenar o(s) modelo(s) em conformidade com o metamodelo
   (a) Acessar o menu "*File*" > "*New*" > "*Project*"
   (b) Selecionar a opção "*General*" > "*Project*"
   (c) Definir nome do projeto. Exemplo: "*meuModelo*"
   (d) Clicar em "*Finish*"
3. Criar modelo na ferramenta de modelagem
   (a) Acessar o menu "*File*" > "*New*" > "*Example...*"
   (b) Selecionar a opção que representa o seu metamodelo (1^a opção) > "*Next*"
   (c) Definir o nome do modelo ou aceitar a sugestão
   (d) Clicar em "*Finish*"

4.4.2. Customizando a Ferramenta de Modelagem

Esta subseção apresenta como realizar refinamentos estéticos na ferramenta de modelagem. A Subseção 4.4.2.1 detalha como adicionar figuras em formato vetorial para representar os nós; a Subseção 4.4.2.2 apresenta como modificar os ícones da paleta (barra de ferramentas); e a Subseção 4.4.2.3 como usar regras de transformação escritas em linguagem EOL para modificar o estilo da fonte dos rótulos dos nós e vértices dos modelos construídos na ferramenta de modelagem.

4.4.2.1. Adicionar Ícones SVG para Representar as `EClasses`

Esta subseção demonstra como utilizar figuras vetoriais em formato SVG[13] para representar os nós, como alternativa às formas padrão (*e.g.*, retângulo (*rectangle*), elipse (*ellipse*) e retângulo com cantos arredondados (*rounded*)).

Considera-se que os ícones SVG já foram construídos pelo leitor. Existem vários *websites*[14] que disponibilizam ícones SVG gratuitos. A ferramenta *Inkscape*[15] pode ser usada para construir ou editar os ícones.

[13] `https://pt.wikipedia.org/wiki/SVG`
[14] *e.g.*, `http://www.flaticon.com/`, `http://www.freepik.com/free-icons`
[15] `https://inkscape.org/pt/`

1. Criar pasta para armazenar os ícones SVG
   (a) Selecionar pasta raiz do projeto da ferramenta
   (b) Acessar o menu "*File*" > "*New*" > "*Folder*"
   (c) Nomear a pasta como "*icons*"
   (d) Clicar em "*Finish*"
2. Adicionar os ícones em formato SVG à pasta recém criada
3. Adicionar a pasta de ícones ao *build path* do projeto
   (a) Abrir o arquivo "*build.properties*"
   (b) Selecionar a pasta "*icons*" em "*Binary Build*"
   (c) Salvar o arquivo "*build.properties*": Acessar o menu "*File*" > "*Save*"
4. Atualizar as anotações dos nós no arquivo `.emf`
   (a) Exemplo:
   ```
   @gmf.node(figure="svg", label.icon="false", label="name",
   svg.uri="platform:/plugin/NOMEDOSEUMETAMODELO/icons/NOMEDOICONE.svg")
   ```
5. Gerar novamente a ferramenta de modelagem
6. Iniciar a ferramenta de modelagem

4.4.2.2. Modificar os Ícones da Paleta

É possível modificar os ícones padrões da paleta (barra de ferramentas) da ferramenta de modelagem para melhor representar os nós (*Objects*) e vértices (*Connections*). Os ícones devem estar em formato GIF[16]. Os aplicativos Inkscape ou Gimp[17] podem ser usados para criar ou editar figuras nesse formato.

1. Adicionar os ícones em formato .GIF à pasta
   `/NOMEDOSEUMETAMODELO.edit/icons/full/obj16`
   (a) Cada arquivo de ícone deve ter exatamente o mesmo nome da metaclasse que irá representar, acompanhado da extensão `.gif`. Exemplo: "*Person.gif*"
2. Gerar novamente a ferramenta de modelagem

[16] `https://pt.wikipedia.org/wiki/Graphics_Interchange_Format`
[17] `https://www.gimp.org/`

4.4.2.3. Alterar o Estilo da Fonte dos Rótulos

Esta subseção apresenta como modificar o estilo dos rótulos dos nós e vértices dos modelos construídos na ferramenta de modelagem. Para tal, faz-se necessário escrever regras de transformação utilizando a linguagem EOL, detalhada na Subseção 4.3.4.

1. Criar um arquivo chamado "*ECore2GMF.eol*" na mesma pasta onde se encontra o arquivo `.emf`
2. Editar o arquivo "*ECore2GMF.eol*", de acordo com o desejado. Exemplos:
   - Mudar o texto do rótulo para itálico

```
1  var label = GmfGraph!Label.all.selectOne(l|l.name = '
       NOMEDOROTULO');
2  label.font = new GmfGraph!BasicFont;
3  label.font.style = GmfGraph!FontStyle#ITALIC;
```

   - Mudar o texto do rótulo para negrito

```
1  var label = GmfGraph!Label.all.selectOne(l|l.name = '
       NOMEDOROTULO');
2  label.font = new GmfGraph!BasicFont;
3  label.font.style = GmfGraph!FontStyle#ITALIC;
```

3. Gerar novamente a ferramenta de modelagem

Obter o nome do rótulo que deseja alterar

1. Abrir o modelo `.gmfgraph` com Exeed
   (a) Clicar com o botão direito sobre o arquivo `.gmfgraph`
   (b) Acessar "*Open With*" > "*Other...*"
   (c) Selecionar "*Exeed Editor*" e clicar em "*OK*"
2. Acessar menu "*Exeed*" > "*Show Structural Info*"
3. Navegar pelos nós "*Canvas*" > "*Figure Gallery Default*"
4. Expandir o rótulo que deseja alterar. Deve ser algo como "*Figure Descriptor NOMEDAMETACLASSELabelFigure*"
5. Acessar as propriedades do "filho" do tipo `Label` e clicar com o botão direito do mouse e acessar "*Show Properties View*"
6. Copiar o valor da propriedade "*name*", que corresponde ao nome do rótulo. Por exemplo "*NOMEDAMETACLASSELabelFigure*"

4.4.3. Regras de Validação em Linguagem EVL

Esta subseção descreve como adicionar regras de validação adicionais aos modelos gerados com a ferramenta de modelagem gráfica, utilizando a linguagem EVL, descrita na Subseção 4.3.5.

1. Crie um novo *plugin* para guardar o arquivo `.evl`
   - Acesse o menu "*File*" > "*New*" > "*Other...*"
   - Escolha a opção "*Plug-in Project*" e clique em "*Next*"
   - Escolha um nome para o *plugin*. Exemplo: "*NOMEDOSEUMETAMODELO.validation*"
   - Clique em "*Next*" e, em seguida, em "*Finish*"
2. Configure as dependências
   (a) Abra o arquivo "*MANIFEST.MF*", que se encontra na pasta "*META-INF*"
   (b) Acesse a aba "*Dependencies*"
   (c) Adicione à lista de dependências:
      i. `org.eclipse.ui.ide`
      ii. `org.eclipse.epsilon.evl.emf.validation`
3. Crie uma pasta chamada "*validation*" na raiz do projeto do *plugin*
4. Crie, dentro dessa pasta, um arquivo `.evl` para escrever suas regras de validação.
   (a) Clique com o botão direito sobre a pasta "*validation*"
   (b) Acesse o menu "*New*" > "*File*"
   (c) Escreva o nome do arquivo. Exemplo: `NOMEDOSEUMETAMODELO.evl`
   (d) Adicione nesse arquivo as regras de validação utilizando a linguagem EVL. Exemplo:

```
1  context MetaClasseX {
2    constraint CannotSelfReference {
3      check : self.source.name <> self.target.at(0).name
4      message : 'Cannot make a self reference'
5        fix {
6          title : "Delete self reference"
7          do {
8            delete self;
9          }
10       }
11   }
12 }
```

5. Vincule as regras de validação à ferramenta de modelagem

(a) Abra o arquivo "*MANIFEST.MF*", que se encontra na pasta "*META-INF*"

(b) Acesse a aba "*Extensions*"

(c) Adicione a extensão: `org.eclipse.epsilon.evl.emf.validation`

i. Clique com o botão direito do mouse na extensão adicionada

ii. Acesse "*New*" > "*constraintsBinding*"

iii. Selecione o "*constraintsBinding*" adicionado e edite seus valores:

A. "*namespaceURI*": deve ser o mesmo do projeto da ferramenta de modelagem.
Exemplo: `http://www.example.org/NOMEDOSEUMETAMODELO`

B. "*constraints*": localização do arquivo que contém as regras EVL, criado no passo 4.
Exemplo: `validation/NOMEDOSEUMETAMODELO.evl`

(d) Adicione a extensão: `org.eclipse.ui.ide.markerResolution`

i. Clique com o botão direito do mouse na extensão adicionada

ii. Acesse "*New*" > "*markerResolutionGenerator*". Faça isso duas vezes.

iii. Selecione o primeiro "*markerResolutionGenerator*" e edite seus valores:

A. "*class*": `org.eclipse.epsilon.evl.emf.validation.EvlMarkerResolutionGenerator`

B. "*markerType*": `NOMEDOSEUMETAMODELO.diagram.diagnostic`

iv. Selecione o segundo "*markerResolutionGenerator*" e edite seus valores:

A. "*class*": `org.eclipse.epsilon.evl.emf.validation.EvlMarkerResolutionGenerator`

B. "*markerType*": `org.eclipse.emf.ecore.diagnostic`

6. Teste as regras de validação

(a) Inicie a ferramenta de modelagem

(b) Crie um modelo que inflija alguma regra de validação definida

(c) Valide o modelo acessando o menu "*Diagram*" > "*Validate*"

- Caso sejam encontrados erros, eles estarão visíveis na *view Problems*. Para visualizá-la acesse o menu "*Window*" > "*Show View*" > "*Problems*"

4.4.4. Lições Aprendidas

Nesta subseção estão descritas possíveis soluções para problemas comuns que podem ser encontrados no processo de desenvolvimento da ferramenta de modelagem gráfica.

1. Gerar a ferramenta de modelagem a cada nova mudança nos arquivos do projeto da ferramenta

2. Funcionamento anormal da ferramenta de modelagem (1)

(a) Apagar todos os projetos gerados (`.diagram`, `.edit`, `.editor` e `.tests`)

(b) Apagar todos os arquivos da pasta model, exceto o `.ecore` e o `.emf`

(c) Gerar a ferramenta de modelagem novamente

3. Funcionamento anormal da ferramenta de modelagem (2)
    (a) Fechar a ferramenta de modelagem
    (b) Fechar o IDE Eclipse Eugenia
    (c) Iniciar o IDE Eclipse Eugenia
    (d) Iniciar a ferramenta de modelagem
4. Objetos ou Conexões (*links*, referências ou associações) não aparecem na paleta da ferramenta de modelagem
    (a) Certifique-se de que todos os nós e links estão relacionados na classe raiz do arquivo `.emf`
        - A classe raiz é a que começa com a anotação `@gmf.diagram`
5. Erro ao gerar ferramenta de modelagem
    (a) Verificar se há erros de compilação e resolvê-los
        i. Acessar menu "*Window*" > "*Show View*" > "*Problems*"
    (b) Caso não haja erro de compilação, tentar os passos 2 ou 3
6. Modelo gráfico não aparece na ferramenta
    (a) Certifique-se de que todos os projetos gerados pelo Eugenia (*i.e.*, projeto da ferramenta, `.diagram`, `.edit`, `.editor`, `.tests` e `.validation` (se for o caso)) estão abertos
    (b) Se for preciso, gere novamente a ferramenta de modelagem

4.4.5. Uma Ferramenta de Modelagem Gráfica de Cenários de Computação Ubíqua

Essa subseção apresenta uma ferramenta de modelagem gráfica para cenários de computação ubíqua, para permitir a criação de modelos em conformidade com o metamodelo para modelagem de cenários de computação ubíqua, apresentado em [Vieira 2016], apresentado na Figura 4.14.

O editor gráfico foi implementado utilizando o *Eclipse Graphical Modeling Framework* (GMF). O editor gráfico foi incrementado com o uso de linguagens da família *Epsilon* e sua construção foi apoiada pela ferramenta *Eugenia*, que também integra a família *Epsilon*. As tecnologias utilizadas nesta subseção foram apresentadas na Subseções 4.3.2 e 4.3.3. Os códigos-fonte aqui apresentados são apenas trechos.

A Figura 4.15 apresenta a ferramenta de modelagem sendo utilizada para a configuração de um cenário de computação ubíqua. A janela da ferramenta é dividida em duas partes: *(i)* e *(ii)*. Em *(i)*, há a área reservada para a construção do modelo. Em *(ii)*, estão os ícones que representam os conceitos do metamodelo, agrupados em uma barra de ferramentas (ou paleta). A barra de ferramentas é dividida em duas partes: *(a)* ferramentas para criação de elementos, ou seja, instâncias dos tipos definidos no metamodelo; e *(b)* ferramentas para criação de associações entre os elementos.

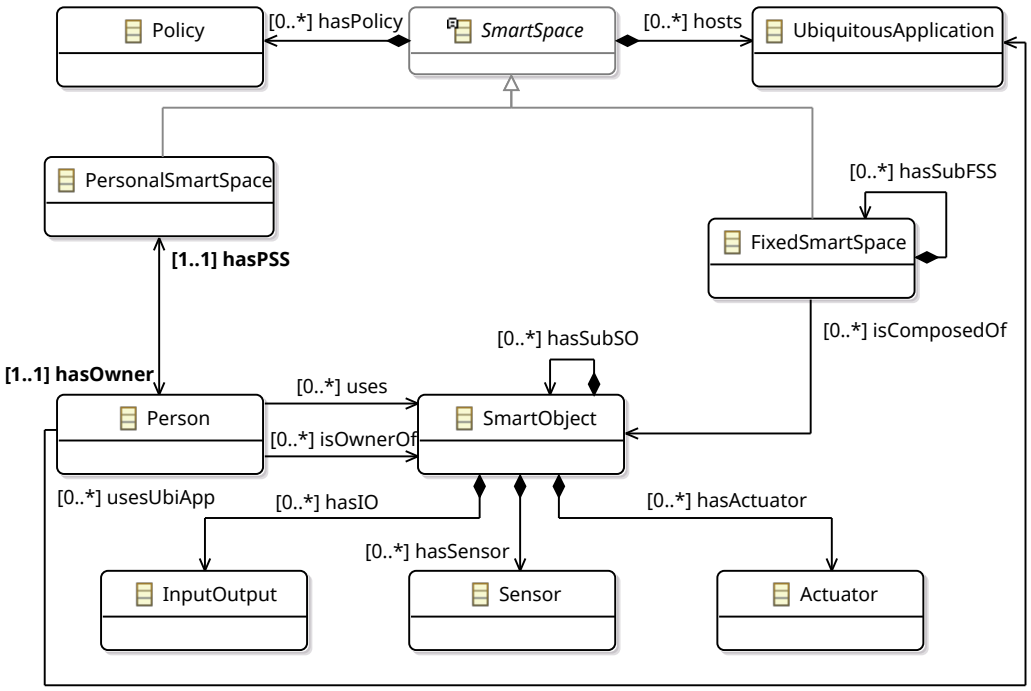

Figura 4.14. Metamodelo proposto para modelagem de cenários compostos por espaços inteligentes pessoais e fixos [Vieira 2016].

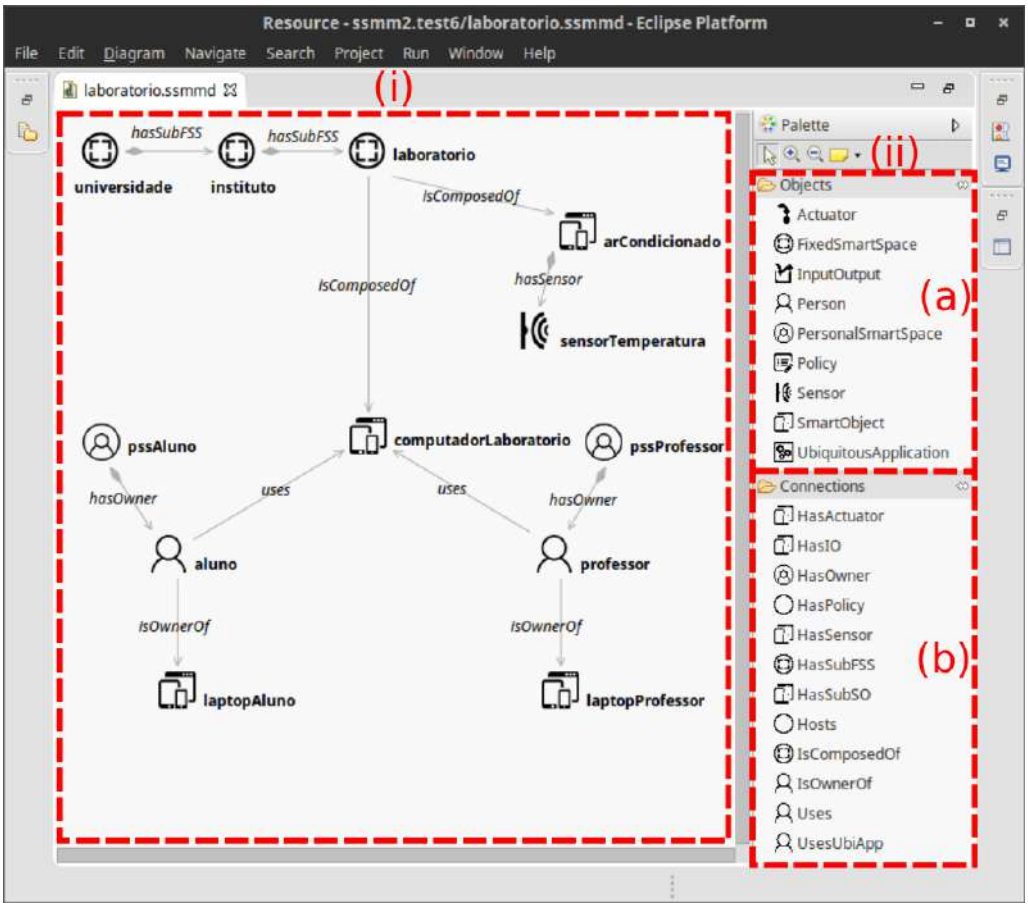

Figura 4.15. Ferramenta de modelagem sendo utilizada para construção de um cenário de computação ubíqua.

Os modelos construídos pela ferramenta de modelagem podem ser salvos em dois arquivos XML com extensões distintas: `.ssmm` e `.ssmmd`. O arquivo com extensão `.ssmm` representa a instância do metamodelo e contém os elementos, seus atributos e as associações entre os elementos; o arquivo com extensão `.ssmmd`, por sua vez, serve para indicar a posição de cada elemento e associação no diagrama gráfico, de maneira que suas posições sejam preservadas ao fechar e abrir novamente o modelo.

A ferramenta de modelagem permite validar o modelo construído para garantir sua conformidade com o seu metamodelo. Caso alguma inconsistência seja encontrada, ela é apontada, permitindo sua correção. Para estender a validação dos modelos, foram definidas regras por meio da linguagem EVL (*Epsilon Validation Language*), permitindo incrementar as regras sintáticas definidas pelo metamodelo com restrições adicionais. A Tabela 4.1 apresenta as regras EVL que foram implementadas.

Tabela 4.1. Regras EVL implementadas na ferramenta gráfica de modelagem.

Regra	**Tipo**	**Descrição**	**Elemento ao qual se aplica**	***Quick fix(es)***
RV1	*Error*	O elemento deve possuir nome definido	Todos	Atribuir nome ao elemento
RV2	*Warning*	O nome do elemento deve iniciar com letra minúscula	Todos	Sugere o nome com inicial minúscula Opção para renomear o elemento
RV3	*Error*	Não pode existir outro elemento do mesmo tipo com mesmo nome	Todos	Opção para renomear o elemento
RV4	*Error*	Não pode haver autorreferenciamento para o elemento	`FixedSmartSpace`, `SmartObject`	Remover a autorreferência

O Código 4.5 apresenta as regras RV1, RV3 e parte da regra RV4: a regra RV1 exige que o elemento tenha um nome; a regra RV3 impede que dois elementos do mesmo tipo tenham nomes iguais; o trecho da regra RV4 apresentado impede o autorreferenciamento a elementos do tipo `FixedSmartSpace`, ou seja, impede que um FSS esteja contido nele próprio. Foram definidos *quick fixes* (indicados no código pelo bloco *fix*) que, quando ativados, realizam a correção do erro encontrado de maneira simplificada. O *quick fix* da regra RV3 solicita a entrada de um novo nome para o elemento duplicado e, então, aplica o novo nome ao elemento. O *quick fix* da regra RV4 apaga o autorreferenciamento para o nó, conforme ilustrado na Figura 4.16.

Código 4.5. Regras EVL: RV1, RV3 e parte da regra RV4.

```
1 //regra RV1
2 constraint HasName {
3     check : self.name.isDefined()
4     message : 'Unnamed ' + self.eClass().name + ' not allowed'
5     fix {
6         title : 'Set a name...'
7         do {
```

```
8               self.name := System.user.prompt('Please enter a name
                    ');
9           }
10      }
11  }
12  //regra RV3
13  constraint CheckUniqueName {
14      guard : not self.~checked.isDefined()
15
16      check {
17          var others := Actuator.all.select(c|c.name = self.name
                and c <> self);
18          if (others.size() > 0) {
19              for (other in others) {
20                  other.~checked := true;
21              }
22              return false;
23          } else {
24              return true;
25          }
26      }
27
28      message : 'Duplicated ' + self.eClass().name + ' name'
29      fix {
30          title : 'Rename...'
31          do {
32              self.name := System.user.prompt('Please enter a new
                    name', self.name);
33          }
34      }
35  }
36  //regra RV4 (parcial)
37  context HasSubFSS {
38      constraint CannotSelfReference {
39          check : self.source.name <> self.target.at(0).name
40          message : 'Cannot make a self reference'
41          fix {
42              title : "Delete self reference"
43              do {
44                  delete self;
45              }
46          }
47      }
48  }
```

Por último, foram feitos ajustes finos na ferramenta de modelagem para melhorar a aparência dos rótulos dos elementos e das associações entre os elementos dos modelos construídos, com o objetivo de facilitar a leitura dos modelos e evitar confusões entre rótulos de associações e rótulos de elementos. Assim sendo, duas regras foram escritas em linguagem EOL (*Epsilon Object Language*) para *(i)* mudar os rótulos das associações para itálico e *(ii)* mudar os rótulos dos elementos para negrito. O Código 4.6 apresenta a regra que muda para itálico os rótulos das associações.

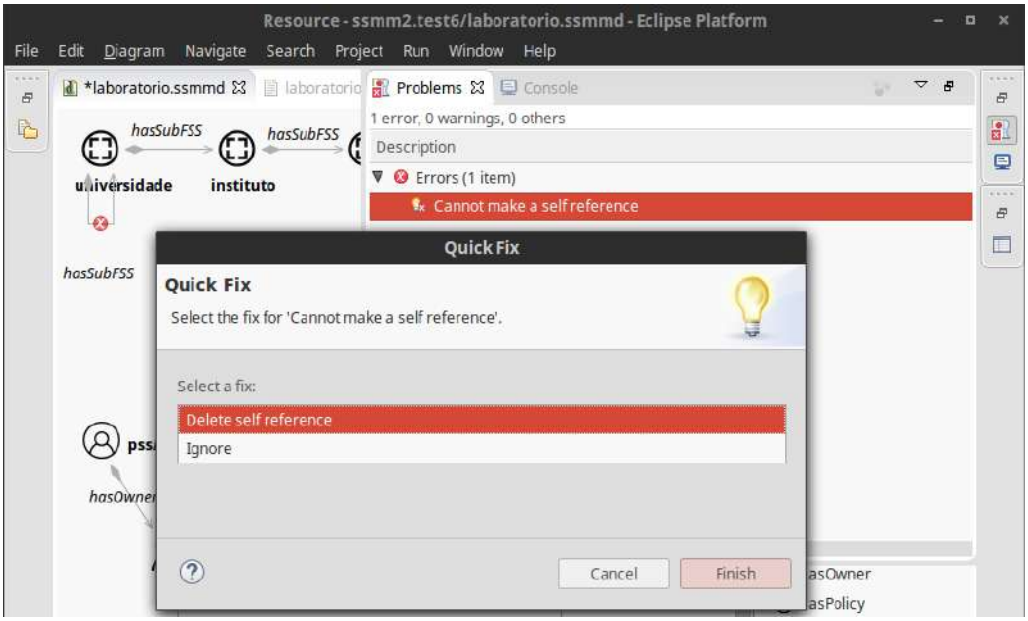

Figura 4.16. Sugestão de correção (*quick fix*) para autorreferenciamento não permitido por meio de regras EVL.

Código 4.6. Regras de transformação escritas em linguagem EOL para estilização da ferramenta de modelagem.

```
1  for (c in GmfGraph!Label.all) {
2    if(c.name = 'HasActuatorLabelLabel' or
3       c.name = 'HasIOLabelLabel' or
4       c.name = 'PersonalSmartSpaceHasOwnerExternalLabel' or
5       c.name = 'HasPolicyLabelLabel' or
6       c.name = 'HasSensorLabelLabel' or
7       c.name = 'HasSubFSSLabelLabel' or
8       c.name = 'HasSubSOLabelLabel' or
9       c.name = 'HostsLabelLabel' or
10      c.name = 'FixedSmartSpaceIsComposedOfExternalLabel' or
11      c.name = 'PersonIsOwnerOfExternalLabel' or
12      c.name = 'PersonUsesExternalLabel' or
13      c.name = 'PersonUsesUbiAppExternalLabel'){
14      c.font = new GmfGraph!BasicFont;
15      c.font.style = GmfGraph!FontStyle#ITALIC;
16   }
17 }
```

4.5. Considerações Finais

Este minicurso introdutório visa, não só introduzir os participantes à área da engenharia dirigida a modelos, como também capacitá-los quanto à construção de um modelo completo, envolvendo desde a concepção do metamodelo até a construção de uma ferramenta de modelagem gráfica com a qual é possível construir modelos derivados de seu metamodelo. Esses modelos poderão, então, ser usados para documentar e manter sistemas de diferentes domínios.

Referências

Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M., and Steggles, P. (1999). Towards a better understanding of context and context-awareness. In *Handheld and ubiquitous computing*, pages 304–307. Springer.

Al-Muhtadi, J., Ranganathan, A., Campbell, R., and Mickunas, M. D. (2003). Cerberus: a context-aware security scheme for smart spaces. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, 2003.(PerCom 2003).*, pages 489–496. IEEE.

Atzori, L., Iera, A., and Morabito, G. (2010). The Internet of Things: A survey . *Computer Networks*, 54(15):2787 – 2805.

Bézivin, J., Jouault, F., and Touzet, D. (2005). Principles, standards and tools for model engineering. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 28–29. IEEE.

Carvalho, S. T. (2013). *Modelagem de Linha de Produto de Software Dinâmica para Aplicações Ubíquas*. PhD thesis, Universidade Federal Fluminense, Niterói, RJ, Brasil.

Carvalho, S. T., Copetti, A., and Loques Filho, O. G. (2011). Sistema de computação ubíqua na assistência domiciliar à saúde. *Journal Of Health Informatics*, 3(2).

Cetina, C., Giner, P., Fons, J., and Pelechano, V. (2009). Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42(10):37–43.

Chagas, J., Ferraz, C., Alves, A. P., and Carvalho, G. (2010). Sensibilidade a contexto na gestão eficiente de energia elétrica. *IN: Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos. Anais do XXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos. Gramado: UFRGS*, pages 145–158.

Chen, M., Gonzalez, S., Vasilakos, A., Cao, H., and Leung, V. (2011). Body Area Networks: A Survey. *Mobile Networks and Applications*, 16(2):171–193.

Chiprianov, V., Kermarrec, Y., Rouvrais, S., and Simonin, J. (2014). Extending enterprise architecture modeling languages for domain specificity and collaboration: application to telecommunication service design. *Software & Systems Modeling*, 13(3):963–974.

Coen, M. H., Phillips, B., Warshawsky, N., Weisman, L., Peters, S., and Finin, P. (2000). Meeting the computational needs of intelligent environments: The metaglue system. In *Managing Interactions in Smart Environments*, pages 201–212. Springer.

Cook, D. J. and Das, S. K. (2007). How smart are our environments? An updated look at the state of the art. *Pervasive and mobile computing*, 3(2):53–73.

Corredor, I., Bernardos, A. M., Iglesias, J., and Casar, J. R. (2012). Model-driven methodology for rapid deployment of smart spaces based on resource-oriented architectures. *Sensors*, 12(7):9286–9335.

Crotty, M., Taylor, N., Williams, H., Frank, K., Roussaki, I., and Roddy, M. (2009). A Pervasive Environment Based on Personal Self-improving Smart Spaces. In Gerhäuser, H., Hupp, J., Efstratiou, C., and Heppner, J., editors, *Constructing Ambient Intelligence*, volume 32 of *Communications in Computer and Information Science*, pages 58–62. Springer Berlin Heidelberg.

Daly, C. and Eclipse Foundation, The (2015a). Emfatic Language Reference. "https://www.eclipse.org/epsilon/doc/articles/emfatic/".

Daly, C. and Eclipse Foundation, The (2015b). EuGENia. "https://www.eclipse.org/epsilon/doc/eugenia/".

Dey, A. K. (2001). Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7.

Dolinar, K., Porekar, J., McKitterick, D., Roussaki, I., Kalatzis, N., Liampotis, N., Papaioannou, I., Papadopoulou, E., Burney, S. M., Frank, K., Hayden, P., and Walsh, A. (2008). PERSIST Deliverable D3.1: Detailed Design for Personal Smart Spaces. http://www.ict-persist.eu/?q=content/persist-deliverables-and-publications. [Online; acessado em Abril-2015].

Eclipse Foundation, The (2015a). Customizing a GMF editor generated by EuGENia. "https://www.eclipse.org/epsilon/doc/articles/eugenia-polishing/".

Eclipse Foundation, The (2015b). Epsilon Object Language. "https://www.eclipse.org/epsilon/doc/eol/".

Eclipse Foundation, The (2015c). Epsilon Validation Language. "https://www.eclipse.org/epsilon/doc/evl/".

Eclipse Foundation, The (2015d). EuGENia GMF Tutorial. "https://www.eclipse.org/epsilon/doc/articles/eugenia-gmf-tutorial/".

Eclipse Foundation, The (2015e). Graphical Modeling Framework Documentation. "https://wiki.eclipse.org/Graphical_Modeling_Framework/Documentation/Index".

Erthal, M. S. (2014). Uma proposta para interpretação de contexto em ambientes inteligentes. Master's thesis, Universidade Federal Fluminense, Niterói, RJ, Brasil.

Favre, J.-M. and Nguyen, T. (2005). Towards a megamodel to model software evolution through transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):59–74.

Ferreira Filho, J. B. (2014). *Leveraging model-based product lines for systems engineering*. PhD thesis, Université Rennes 1, Paris, France.

Freitas, L. A., Costa, F. M., Rocha, R. C., and Allen, A. (2014). An architecture for a smart spaces virtual machine. In *Proceedings of the 9th Workshop on Middleware for Next Generation Internet Computing*, page 7. ACM.

Gallacher, S. M., Papadopoulou, E., Taylor, N. K., and Williams, M. H. (2010). Putting the 'Personal' into Personal Smart Spaces. In *Proceedings of Pervasive Personalisation Workshop*, volume 2010, pages 10–17.

Guinard, D., Trifa, V., Pham, T., and Liechti, O. (2009). Towards Physical Mashups in the Web of Things. In *Proceedings of INSS 2009 (IEEE Sixth International Conference on Networked Sensing Systems)*, pages 196–199, Pittsburgh, USA.

Helal, S., Mann, W., El-Zabadani, H., King, J., Kaddoura, Y., and Jansen, E. (2005). The gator tech smart house: A programmable pervasive space. *Computer*, 38(3):50–60.

Honkola, J., Laine, H., Brown, R., and Tyrkko, O. (2010). Smart-M3 information sharing platform. In *ISCC*, pages 1041–1046.

Johanson, B. and Fox, A. (2002). The Event Heap: a coordination infrastructure for interactive workspaces. In *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on*, pages 83–93.

Kavanagh, J. and Hall, W. (2008). Grand challenges in computing research 2008. `http://www.ukcrc.org.uk/grand-challenge/`. [Online; acessado em Abril-2015].

Kawsar, F. (2009). A document-based framework for user centric smart object systems. *PhD in Computer Science, Waseda University, Japan*.

Kolovos, D. S., GarcÃa-DomÃnguez, A., Rose, L. M., and Paige, R. F. (2015a). Eugenia: towards disciplined and automated development of gmf-based graphical model editors. *Software & Systems Modeling*, pages 1–27.

Kolovos, D. S., Paige, R. F., Kelly, T., and Polack, F. A. (2006). Requirements for domain-specific languages. In *Proceedings of the First ECOOP Workshop on Domain-Specific Program Development*.

Kolovos, D. S., Rose, L. M., GarcÃa-DomÃnguez, A., and Paige, R. F. (2015b). The Epsilon Book. "`https://www.eclipse.org/epsilon/doc/book/`".

Kortuem, G., Kawsar, F., Fitton, D., and Sundramoorthy, V. (2010). Smart objects as building blocks for the internet of things. *Internet Computing, IEEE*, 14(1):44–51.

Latré, B., Braem, B., Moerman, I., Blondia, C., and Demeester, P. (2011). A survey on wireless body area networks. *Wireless Networks*, 17(1):1–18.

López-Fernández, J. J., Cuadrado, J. S., Guerra, E., and de Lara, J. (2015). Example-driven meta-model development. *Software & Systems Modeling*, 14(4):1323–1347.

Lupiana, D., O'Driscoll, C., and Mtenzi, F. (2009). Taxonomy for ubiquitous computing environments. In *Networked Digital Technologies, 2009. NDT '09. First International Conference on*, pages 469–475.

Melo, P. C. F. (2014). CSVM: Uma Plataforma para CrowdSensing Móvel Dirigida por Modelos em Tempo de Execução. Master's thesis, Instituto de Informática - Universidade Federal de Goiás, Goiânia-GO, Brasil.

Merks, E. and Sugrue, J. (2009). Essential EMF. "`https://dzone.com/refcardz/essential-emf`". "[Online; acessado em Junho-2015]".

Object Management Group (2015). Object Constraint Language (OCL). "`http://www.omg.org/spec/OCL/`".

Ranganathan, A., Chetan, S., Al-Muhtadi, J., Campbell, R. H., and Mickunas, M. D. (2005). Olympus: A high-level programming model for pervasive computing environments. In *Third IEEE International Conference on Pervasive Computing and Communications, 2005. PerCom 2005.*, pages 7–16. IEEE.

Román, M., Hess, C., Cerqueira, R., Campbell, R. H., and Nahrstedt, K. (2002). Gaia: A Middleware Infrastructure to Enable Active Spaces. *IEEE Pervasive Computing*, 1:74–83.

Roriz Junior, M. P. (2013). C3S: Uma Plataforma de Middleware de Compartilhamento de Conteúdo para Espaços Inteligentes. Master's thesis, Instituto de Informática - Universidade Federal de Goiás, Goiânia-GO, Brasil.

Roussaki, I., Kalatzis, N., Liampotis, N., Frank, K., Sykas, E. D., and Anagnostou, M. (2012). Developing context-aware personal smart spaces. In Alencar, P. and Cowan, D., editors, *Handbook of Research on Mobile Software Engineering: Design, Implementation, and Emergent Applications*, chapter 35, pages 659–676. IGI Global, Hershey, PA, USA.

Roussaki, I., Kalatzis, N., Liampotis, N., Kosmides, P., Anagnostou, M., and Sykas, E. (2015). Putting Personal Smart Spaces into Context. In Díaz, V. G., Lovelle, J. M. C., and García-Bustelo, B. C. P., editors, *Handbook of Research on Innovations in Systems and Software Engineering*, chapter 27, pages 710–730. IGI Global, Hershey, PA, USA.

Roussaki, I., Kalatzis, N., Liampotis, N., Papaioannou, I., Pils, C., Crotty, M., AlanWalsh, Frank, K., Whitmore, J., McKitterick, D., Taylor, N., McBurney, S., Papadopoulou, E., Williams, H., Dolinar, K., Porekar, J., Venezia, C., and Bucchiarone, A. (2008). PERSIST Deliverable D2.1: Scenario Description and Requirements Specification. `http://www.ict-persist.eu/?q=content/persist-deliverables-and-publications`. [Online; acessado em Abril-2015].

Sampaio Junior, A. R. (2014). Controle de Microgrids Dirigido por Modelos. Master's thesis, Instituto de Informática - Universidade Federal de Goiás, Goiânia-GO, Brasil.

Schilit, B. N. and Theimer, M. M. (1994). Disseminating active map information to mobile hosts. *Network, IEEE*, 8(5):22–32.

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31.

Seidewitz, E. (2003). What models mean. *IEEE software*, 20(5):26–32.

Shi, Y., Xie, W., Xu, G., Shi, R., Chen, E., Mao, Y., and Liu, F. (2003). The smart classroom: merging technologies for seamless tele-education. *IEEE Pervasive Computing*, 2(2):47–55.

Siegemund, F. (2004). A context-aware communication platform for smart objects. In *Pervasive Computing*, pages 69–86. Springer.

Smirnov, A., Kashevnik, A., Shilov, N., and Teslya, N. (2013). Context-based access control model for smart space. In *Cyber Conflict (CyCon), 2013 5th International Conference on*, pages 1–15. IEEE.

Sousa, J. P. and Garlan, D. (2002). Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In Bosch, J., Gentleman, M., Hofmeister, C., and Kuusela, J., editors, *Software Architecture*, volume 97 of *IFIP - The International Federation for Information Processing*, pages 29–43. Springer US.

Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Pearson Education.

Sztajnberg, A., Rodrigues, A. L. B., Bezerra, L. N., Loques, O. G., Copetti, A., and Carvalho, S. T. (2009). Applying context-aware techniques to design remote assisted living applications. *International Journal of Functional Informatics and Personalised Medicine*, 2(4):358–378.

Taylor, N. (2008). Personal eSpace and Personal Smart Spaces. In *Self-Adaptive and Self-Organizing Systems Workshops, 2008. SASOW 2008. Second IEEE International Conference on*, pages 156–161.

Taylor, N. (2011). Personal Smart Spaces. In Ferscha, A., editor, *Pervasive Adaptation: The Next Generation Pervasive Computing Research Agenda*, pages 79–80. Institute for Pervasive Computing, Johannes Kepler University Linz, Linz, AUS.

Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-Specific Languages: An Annotated Bibliography. *Sigplan Notices*, 35(6):26–36.

Vieira, M. A. (2016). Modelagem de espaços inteligentes pessoais e espaços inteligentes fixos no contexto de cenários de computação ubíqua. Master's thesis, Universidade Federal de Goiás, Goiânia, Goiás, Brazil.

Völter, M., Stahl, T., Bettin, J., Haase, A., and Helsen, S. (2013). *Model-driven software development: technology, engineering, management*. John Wiley & Sons.

Weiser, M. (1991). The computer for the 21st century. *Scientific american*, 265(3):94–104.

Weiser, M. and Brown, J. S. (1997). The coming age of calm technology. In *Beyond calculation*, pages 75–85. Springer.

Wienands, C. and Golm, M. (2009). Anatomy of a visual domain-specific language project in an industrial context. In *Model Driven Engineering Languages and Systems*, pages 453–467. Springer.

Wyckoff, P., McLaughry, S. W., Lehman, T. J., and Ford, D. A. (1998). T spaces. *IBM Systems Journal*, 37(3):454–474.

Yau, S. S., Gupta, S. K., Karim, F., Ahamed, S. I., Wang, Y., and Wang, B. (2003). Smart classroom: Enhancing collaborative learning using pervasive computing technology. In *ASEE 2003 Annual Conference and Exposition*, pages 13633–13642. sn.

Biografia Resumida dos Autores

Prof. Me. Marcos Alves Vieira
Professor do Instituto Federal de Educação, Ciência e Tecnologia Goiano - Campus Iporá, onde atualmente é coordenador do curso superior de Tecnologia em Análise e Desenvolvimento de Sistemas. É Bacharel em Informática pelo Instituto Federal de Educação, Ciência e Tecnologia de Goiás (IFG) e Mestre em Ciência da Computação pelo Instituto de Informática (INF) da Universidade Federal de Goiás (UFG). Durante seu mestrado, desenvolveu pesquisa com foco em Espaços Inteligentes para Computação Ubíqua e Engenharia Baseada em Modelos. Participante em comunidades de software livre e colaborador em eventos de software livre regionais.

Prof. Dr. Sérgio Teixeira de Carvalho
Professor do Instituto de Informática (INF) da Universidade Federal de Goiás (UFG) desde 2006, atuando no mestrado, na pós-graduação lato sensu e na graduação. Por muitos anos atuou principalmente como coordenador de projetos de software, ocupando posições como Diretor de Sistemas de Informação e Diretor de Suporte Tecnológico. Por mais de dezesseis anos atuou como profissional de T.I. de diversas empresas públicas e privadas. Possui graduação em Ciência da Computação pela Universidade Federal de Goiás (UFG), Mestrado e Doutorado em Computação pela Universidade Federal Fluminense (UFF). Seus principais campos de atuação são Computação Ubíqua/Pervasiva, em especial voltada para aplicações na área de Sistemas de Informação em Saúde, Informática em Saúde, Arquitetura de Software, Linha de Produto de Software Dinâmica e Arquiteturas Adaptativas.