
César Henrique Bernabé

# Evolução da Arquitetura do Zanshin – um framework para análise de requisitos em tempo de execução

Vitória, ES

2017

César Henrique Bernabé

# Evolução da Arquitetura do Zanshin – um framework para análise de requisitos em tempo de execução

Monografia apresentada ao Curso de Ciência da Computação do Departamento de Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Universidade Federal do Espírito Santo – UFES

Centro Tecnológico

Departamento de Informática

Orientador: Prof. Dr. Vítor E. Silva Souza

Vitória, ES

2017

César Henrique Bernabé
Evolução da Arquitetura do Zanshin – um framework para análise de requisitos em tempo de execução/ César Henrique Bernabé. – Vitória, ES, 2017-
46 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Vítor E. Silva Souza

Monografia (PG) – Universidade Federal do Espírito Santo – UFES
Centro Tecnológico
Departamento de Informática, 2017.

1. Engenharia de Requisitos Orientada a Objetivos. 2. Zanshin. I. Souza, Vítor Estêvão Silva. II. Universidade Federal do Espírito Santo. IV. Evolução da Arquitetura do Zanshin – um framework para análise de requisitos em tempo de execução

CDU 02:141:005.7

César Henrique Bernabé

# Evolução da Arquitetura do Zanshin – um framework para análise de requisitos em tempo de execução

Monografia apresentada ao Curso de Ciência da Computação do Departamento de Informática da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Vitória, ES
2017

# Agradecimentos

AGRADECIMENTOS

*"Sou contra a violência, pois quando parece fazer o bem,*
*o bem é apenas temporário.*
*O mal que causa é permanente.*
*(Gandhi)*
*Alguns de nós pensam que aguentar nos faz fortes.*
*Mas às vezes, é desistir.*
*(Herman Hesse)*

# Resumo

Com o avanço da tecnologia, sistemas de software são executados em ambientes cada vez mais dinâmicos, sendo assim obrigados a lidar com requisitos cada vez mais complexos. Nesse universo tecnológico, onde sistemas precisam reagir a diversos tipos de condições destaca-se o uso de sistemas que modificam seu comportamento e performance em tempo de execução para satisfazer requisitos mesmo em caso de falha, podendo se replanejar e reconfigurar a medida que novas exigências são encontradas e precisam ser atendidas. *Zanshin* é uma abordagem baseada em Engenharia de Requisitos Orientada a Objetivos (Goal-Oriented Requirements Engineering ou simplesmente GORE) criada para apoiar o desenvolvimento de sistemas adaptativos. Baseado no fato de que todo sistema possui, ainda que implícito, um ciclo de retroalimentação, o framework utiliza dessa premissa para monitorar e avaliar o comportamento do sistema alvo, enviando assim instruções de adaptação para o mesmo. Para auxiliar o processo de modelagem dos sistemas adaptativos são definidas duas novas classes de Requisitos: - Requisitos de Percepção (*Awareness Requirements* ou s), que especificam os indicadores de convergência que o sistema deve atingir para que determinado objetivo seja cumprido; - Requisitos de Evolução (*Evolution Requirements* ou *EvoReqs*), que definem as medidas a serem seguidas pelo processo de adaptação, ou seja, as instruções que serão enviadas ao sistema alvo mediante falha de um objetivo; Este trabalho discute uma nova proposta para o metamodelo operacional do *Zanshin*, que foi reformulado buscando compreender as restrições de *GORE* de uma forma mais precisa do que na adotada anteriormente. Além disso, para apoiar o processo de modelagem dos requisitos dentro desta abordagem, é apresentada a ferramenta CASE Unagi, que implementa um editor gráfico para modelos de sistemas adaptativos e oferece um sistema de integração com o *Zanshin*, permitindo assim que modelos criados sejam importados diretamente para a plataforma de apoio a sistemas adaptativos.

**Palavras-chaves**: Engenharia de Requisitos, GORE, Zanshin, Unagi...

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

| | |
|---|---|
| ABNT | Associação Brasileira de Normas Técnicas |
| HTML | HyperText Markup Language |
| UML | Unified Modeling Language |
| URL | Uniform Resource Locator |
| XML | eXtensible Markup Language |
| GORE | Goal Oriented Requirements Engineering |
| EMF | Eclipse Modeling Framework |
| API | Application Programming Interface |
| MDD | Model Driven Development |

# Sumário

# 1 Introdução

O avanço da tecnologia nas ultimas décadas permitiu que a complexidade das atividades realizadas por computadores se tornasse cada vez maior, demandando que projetos de sistemas de software passassem a abranger ainda mais os detalhes de domínio do ambiente em que os programas computacionais seriam executados (ANDERSSON et al., 2009; BRUN et al., 2009). Esse fator motivou estudos na área de modelagem e projeto de sistemas, fazendo com que novas pesquisas buscassem abranger os processos de projeto, construção e teste. Entretanto, para garantir a estabilidade dos sistemas, fazia-se uso majoritariamente da intervenção humana, o que rapidamente tornou-se inviável à medida que os sistemas cresciam para atender o aumento na demanda de usuários (ANDERSSON et al., 2009). Assim, a utilização de sistemas adaptativos vem tornando-se a solução mais viável e prática para a atual conjuntura do desenvolvimento de softwares. Além disso, o aumento do número de diferentes dispositivos e situações em que os softwares podem ser executados faz com que esses passem a enfrentar uma grande diversidade de contextos (muitas vezes imprevisíveis) de execução, fundamentando ainda mais a pesquisa na área de softwares adaptativos (KEPHART; CHESS, 2003).

Sistemas adaptativos são dotados da capacidade de tomar decisões para se ajustar e se reconfigurar mediante mudanças de contexto, permitindo assim que os requisitos elicitados continuem a ser atendidos de forma satisfatória (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). Entretanto, poucas soluções desse tipo consideram a modelagem das características adaptativas no sistema desde a fase de levantamento de requisitos de software, como por exemplo o *Zanshin* (SOUZA, 2012), uma abordagem que baseia-se em modelos de requisitos para projetar características adaptativas em sistemas através de novos requisitos chamados Requisitos de Adaptação.

## 1.1 Objetivos

Este trabalho possui dois objetivos principais: a reconstrução do metamodelo operacional do *Zanshin* e a criação de uma ferramenta gráfica usada para modelar sistemas adaptativos através da Engenharia de Requisitos Orientada a Objetivo (*Goal Oriented Requirements Engineering* ou *GORE*). O primeiro objetivo refere-se ao metamodelo de objetivos baseado em $i^\star$, usado pelo framework para casar os requisitos do modelo de domínio específico com as instâncias dos elementos do metamodelo de *GORE*, e assim garantir que os objetivos do sistema estão sendo atendidos satisfatoriamente. Para ambas as atividades, foram utilizados os conceitos aprendidos ao longo do curso de Ciência da Computação, dessa forma são objetivos específicos deste projeto:

- Levantar as deficiências dos metamodelo atual do Zanshin, identificando os pontos em que as relações entre os elementos deveriam ser modificadas para representarem mais fielmente as formalidades da Engenharia de Requisitos Orientada a Objetivos, bem como pontos onde as decomposições entre elementos deveria tornar-se restrita, por exemplo.
- Elaborar um metamodelo atualizado que, além de representar mais rigorosamente a hierarquia dos elementos *GORE*, também reflita as necessidades da arquitetura do *Zanshin*, como por exemplo as relações de entre elementos.
- Modificar o código fonte do *framework* para que o mesmo possa utilizar o novo metamodelo desenvolvido e executar o mecanismo de adaptações considerando esse novo metamodelo.
- Desenvolver uma ferramenta que permita ao usuário criar uma representação gráfica do modelo do sistema alvo e implementar, dentro dessa ferramenta, um módulo que permita converter o modelo gráfico representado para arquivos XML que podem ser importados diretamente para o *Zanshin*.
- Apresentar os trabalhos desenvolvidos, juntamente com as perspectivas futuras de otimização de ambos os sistemas apresentados nesse trabalho.

## 1.2 Metodologia

O trabalho realizado compreendeu as seguintes atividades:

1. *Revisão Bibliográfica*: Estudo sobre Engenharia de Requisitos Orientada a Objetivos, Desenvolvimento Orientado a Modelos e suas ferramentas, de publicações acadêmicas sobre *Zanshin* e sobre desenvolvimento de sistemas adaptativos.
2. *Estudo das Tecnologias:* Levantamento das tecnologias disponíveis para *Eclipse Modeling Framework* (EMF) que permitam o desenvolvimento de editores gráficos dentro da plataforma Eclipse™, de tecnologias que podem ser utilizadas para o desenvolvimento de editores gráficos e do código fonte do *Zanshin*.
3. *Elaboração do novo Metamodelo*: Nessa etapa o novo metamodelo a ser usado foi elaborado gradativamente a partir dos de informações obtidas dos documentos estudados e das discussões realizadas em reuniões com grupo de estudos de Engenharia de Requisitos Orientada a Objetivos na Ufes.
4. *Adequação do Zanshin ao novo Metamodelo*: Após finalização do metamodelo, inicou-se processo de adequação do *Framework* para que o mesmo pudesse operar adequadamente de acordo com a nova proposta, consistindo da modificação do código fonte

do *Zanshin*, bem como realização de testes de validação para garantir a consistência do novo metamodelo.

5. *Implementação da Ferramenta de Modelagem*: Uma primeira versão da ferramenta foi desenvolvida no contexto de trabalho de Iniciação Científica, entretanto a mesma usava o metamodelo antigo do *Zanshin*. Essa etapa consiste, portanto, no refatoramento da ferramenta gráfica que permite a modelagem de sistemas adaptativos seguindo as formalidades do novo metamodelo proposto para o sistema *Zanshin*, bem como a criação de módulo java que permite a exportação do modelo desenvolvido nessa ferramenta para arquivo XML adequado aos padrões do *Framework*.

6. *Redação da Monografia:* Escrita da monografia, etapa obrigatória do processo de elaboração do Projeto de Graduação. Para a escrita desta, foi utilizada a linguagem *LaTeX*[1] utilizando o template *abnTeX*[2] que atende os requisitos das normas da ABNT (Associação Brasileira de Normas Técnicas) para elaboração de documentos técnicos e científicos brasileiros. Para apoiar este processo, foi utilizado o aplicativo *TeXstudio* [3].

## 1.3 Organização do Texto

Este texto está dividido em quatro partes principais além desta introdução, que seguem:

- **Capítulo 2** – Referencial Teórico: apresenta discussão acerca de *GORE* e *Model Driven Development*, focando na relação desses tópicos com sistemas adaptativos e com o processo de desenvolvimento da ferramenta *Unagi*. Ademais, é discutida a arquitetura do sistema *Zanshin* e suas características.

- **Capítulo 4** – Proposta: nesse capitulo são apresentados os processos e decisões que levaram a elaboração do novo metamodelo do *Zanshin*, bem como as modificações decorrentes dessas modificações na arquitetura da plataforma. Além disso, na seção seguinte é abordado o processo de desenvolvimento da ferramenta *Unagi* e os pormenores da implementação de todos os modulos da mesma.

- **Capítulo ??** – Revisão: é feita validação do metamodelo evoluído do *Zanshin* e da implementação da ferramenta *Unagi*.

- **Capítulo 6** – Considerações Finais: apresenta as conclusões obtidas ao final deste trabalho, tal como as dificuldades encontradas e as perspectivas de trabalhos futuros

[1] LaTeX – http://www.latex-project.org/
[2] abnTeX – http://www.abntex.net.br
[3] www.texstudio.org

para esse contexto.

# 2 Referencial Teórico

Este capítulo apresenta os principais conceitos teóricos que alicerçaram a evolução do metamodelo de requisitos do *Zanshin* e de conceitos que fundamentaram o desenvolvimento da ferramenta *Unagi*. A seção 2.1 fala sobre Engenharia de Requisitos Orientada a Objetivos, destacando os principais conceitos dessa área que foram utilizados ao longo deste trabalho. A seção 2.2 apresenta o sistema *Zanshin* e os detalhes do metamodelo original do *Framework*. A seção 2.3 apresenta um breve resumo sobre Desenvolvimento Orientado a Modelos (*Model Driven Development* ou MDD) assim como as principais ferramentas que foram utilizadas durante o desenvolvimento do *Unagi*, como as funcionalidades EMF de modelagem do Eclipse™, o plugin Sirius™, dentre outros.

## 2.1 Engenharia de Requisitos Orientada a Objetivos

A Egenharia de Software é uma área da Ciência da Computação voltada ao estudo dos processos, métodos, técnicas, ferramentas e ambientes de suporte ao desenvolvimento de software, apoiando-se principalmente nas práticas e aplicações da área de Gerência de Projetos com o objetivo de promover melhor organização, produtividade e qualidade em todo o processo de desenvolvimento de um software (FALBO, 2014).

Dentro da área de Engenharia de Software, destaca-se uma importante subárea, a área de Egenharia de Requisitos de Software, focada no processo de elicitação de requisitos, considerados fatores determinantes no sucesso do desenvolvimento de um software (FALBO, 2017). Requisitos podem ser entendidos como a definição do que o sistema pode prover, ou também entendidos como o que o sistema é capaz de fazer para atingir um determinado objetivo (PFLEEGER, 2004).

Devido ao fato de requisitos estarem diretamente ligados aos objetivos do sistema, destaca-se também a Engenharia de Requisitos Orientada a Objetivos, uma subárea da Engenharia de Requisitos. Objetivos são parte importante do processo de elicitação de requisitos, seu propósito é indicar as principais necessidades que justificam a criação de um determinado sistema, demonstrando os casos em que as funcionalidades do mesmo satisfarão as necessidades elicitadas, além de dizer como o sistema deve ser construído para satisfazê-las (ROSS; SCHOMAN, 1977).

Em uma descrição geral e resumida do processo de identificação de objetivos, pode-se dizer que o potencial software é analisado nos ambientes organizacional, operacional e técnico, onde são assim identificados os problemas de contexto e as oportunidades de solução desses problemas. Então, os objetivos são criados com foco na resolução dos

problemas e das oportunidades identificadas. Tendo em mãos os objetivos do sistema devidamente refinados, os requisitos do sistema são então elaborados para que esses objetivos sejam devidamente atendidos. Além de apoiar no processo de modelagem de requisitos, objetivos são usados para apoiar outros propósitos como gerenciamento de conflitos e o processo de verificação (LAPOUCHNIAN, 2005). De acordo com (LAMSWEERDE, 2001), objetivos podem ser reformulados em diferentes níveis de abstração dependendo do tipo de necessidade que o sistema alvo deve atender, abrangendo desde interesses referentes a estratégias de negócios até conceitos técnicos de atividades, assim referindo-se a requisitos funcionais e não-funcionais.

A necessidade de uso de objetivos no processo de modelagem de sistemas de software vem se tornando cada vez mais clara a medida que analistas percebem que:

- Objetivos provêm critérios claros de completude dos requisitos do sistema, permitindo também que requisitos desnecessários sejam descartados (LAMSWEERDE, 2001).
- Objetivos facilitam o processo de entendimento dos requisitos pelas partes interessadas (LAMSWEERDE, 2001).
- O uso de objetivos melhora a legibilidade de documentos de especificação de requisitos, pois permite que engenheiros possam enxergar com mais clareza as alternativas de desenvolvimento dos requisitos do sistema. Além de facilitar o processo de gerenciamento de conflitos (LAMSWEERDE, 2001).
- Objetivos dirigem parte do processo de elicitação de requisitos, facilitando a identificação de boa parte deles (LAPOUCHNIAN, 2005).

Diferentemente dos requisitos, objetivos podem precisar da cooperação entre diferentes tipos de refinamentos para que sejam atendidos de forma suficiente (DARDENNE; LAMSWEERDE; FICKAS, 1993). Em outras palavras, um objetivo diretamente relacionado ao sistema a ser criado torna-se um Requisito, enquanto um objetivo sob responsabilidade de um agente do ambiente em que o software será executado torna-se uma Pressuposição de Domínio (ou *Domain Assumptions*) e, nesse caso, são satisfeitos devido a uma regra de negócio (LAMSWEERDE, 2001; LAMSWEERDE; DARIMONT; LETIER, 1998). Objetivos funcionais podem ser classificados como objetivos rígidos (*Hard Goals*), cujo critério de satisfação pode ser atendido de forma técnica (DARDENNE; LAMSWEERDE; FICKAS, 1993) e objetivos fracos (*Soft Goals*), que não possuem critérios claros de satisfação, entretanto são úteis quando deseja-se comparar os melhores refinamentos ao objetivo estudado. Para que *Soft Goals* tenham um parâmetro claro de satisfabilidade, são adicionados a eles os Critérios de Qualidade (*Quality Constraints*). Por exemplo, um *Soft Goal* “Baixo Custo” pode ser refinado no critério de qualidade “Custo deve ser menor que mil reais”. Por fim, (JURETA; MYLOPOULOS; FAULKNER, 2008) define outro tipo de

refinamento para especificar a atendibilidade de um objetivo: as tarefas (*Taks*), que são os passos a serem tomados para que um determinado objetivo seja cumprido. Em outras palavras, tarefas são definidas por funcionalidades do sistemas que, se executadas com sucesso, são consideradas satisfeitas (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012).

Objetivos relacionam-se um com o outro através de refinamentos. Segundo (DARDENNE; FICKAS; LAMSWEERDE, 1991; DARDENNE; LAMSWEERDE; FICKAS, 1993), objetivos podem ser refinados usando grafos E/OU (*AND/OR*). O critério de satisfabilidade de objetivos refinados em "E" ou "OU" segue os conceitos da lógica booleana: refinamentos do tipo "E" implicam que para que um objetivo seja considerado satisfeito, todos os sub-objetivos refinados a partir dele devem estar bem sucedidos, enquanto refinamentos do tipo "OU" relacionam o objetivo principal com um conjunto de alternativas, ou seja, basta que um de seus refinamentos seja atendido para que ele também seja considerado alcançado. Objetivos são refinados até atingirem um nível de granularidade em que são decompostos apenas por tarefas que podem ser completado com sucesso por um ator (humano ou outro sistema) (SOUZA et al., 2013). Refinamentos podem acontecer entre Objetivos e outros Objetivos, *Softgoals*, Tarefas e Pressuposições de Domínio.

Em questões de representação gráfica, os modelos de objetivos discutidos nesse texto são grafos ordenados que exibem as exigências das partes interessadas no topo do modelo e abaixo, objetivos (e tarefas) mais refinados, adotando uma topologia do tipo árvore. A simbologia utilizada é baseada na sintaxe de $i^\star$ (YU et al., 2011). Um exemplo de modelo de objetivos representando um sistema de despacho de ambulâncias é mostrado na Figura 1.

### 2.1.1 Modelos de Objetivos em Tempo de Execução

Muitas vezes os requisitos de um *software* precisam ser modificados durante o ciclo de execução do mesmo. Além disso, ao longo do processo de especificação as partes interessadas no sistema podem apresentar requisitos condicionais, ou seja, que assumem diferentes configurações dependendo da ocorrência de determinada situação (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). Em outras palavras, há a necessidade de sistemas que possam se automonitorar e, caso necessário, se adaptarem para que seus objetivos continuem sendo satisfeitos (DALPIAZ et al., 2013). Esse tipo de sistema geralmente é composto por duas partes principais: a primeira sendo o sistema em si, que executa uma tarefa para cumprir um objetivo desejado e a segunda sendo um sistema de monitoramento do primeiro, que envia a ele instruções de modificação de suas configurações para que seus objetivos continuem sendo atendidos (SOUZA et al., 2013).

O sistema de monitoramento é construído fundamentado na premissa de que todo *software* possui um ciclo de retroalimentação (*feedback loop*) (BRUN et al., 2009), e assim realizam o processo de adaptação com base nesse ciclo, aplicando controladores de resposta

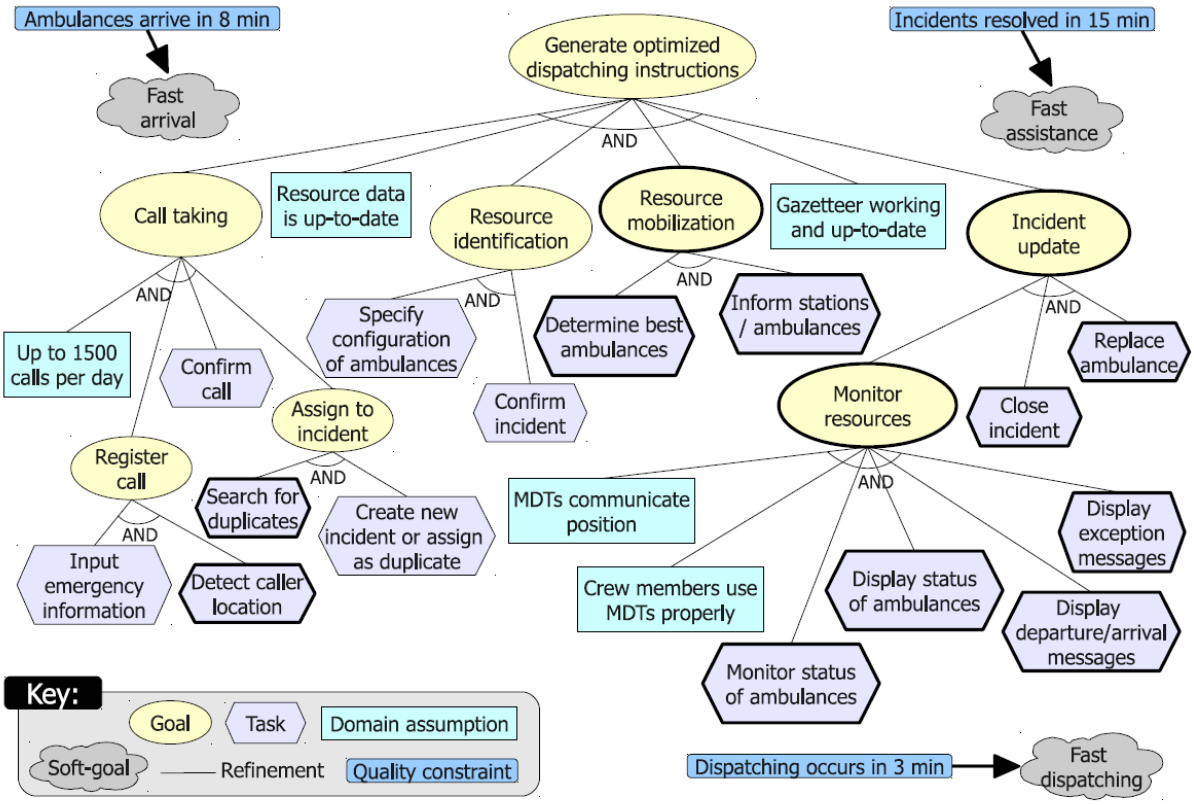


Figura 1 – Exemplo de modelos de objetivos (SOUZA, 2012)

que monitoram o comportamento do sistema e injetam estratégias de adaptação (SOUZA et al., 2013). O módulo adaptador verifica, de acordo com as saídas do sistema alvo, se os objetivos internos a esse estão sendo atendidos e, para isso, necessita importar o modelo de objetivos (SOUZA et al., 2013) enriquecido de elementos que indicam os requisitos a serem observados e as estratégias de adaptação relativas.

Modelos de sistemas adaptativos incluem requisitos autoconscientes, ou seja, definidos em relação ao sucesso, falha ou qualidade de serviço de outros requisitos (SOUZA et al., 2013). Assim, esses são considerados "requisitos especiais", já que sua operacionalização está relacionada a mudança de outros requisitos (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). Ademais, o comportamento do sistema é caracterizado por eventos que ocorrem em tempo de execução e que estão diretamente ligados a instâncias de objetivos (DALPIAZ et al., 2013). Além disso, é importante observar que essa abordagem é considerada orientada a objetivos já que os requisitos mencionados são derivados do refinamento de objetivos elicitados para o sistema.

Requisitos autoconscientes são divididos em dois tipos principais: Requisitos de Percepção (*Awareness Requirements* ou *AwReqs*) (SOUZA et al., 2013) e Requisitos de Evolução (*Evolution Requirements* ou *EvoReqs*) (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). *AwReqs* são requisitos que referem-se ao estado de outros requisitos em tempo de execução, representando situações onde as partes interessadas desejam que o

sistema se adapte (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012), e podem se referir a qualquer tipo de elemento, sejam objetivos, *Softgoals*, tarefas e pressuposições de domínio. Ainda mais, *AwReqs* indicam o quão critico um requisito pode ser ao descrever o grau de tolerância a falhas do mesmo (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). Antes da execução de um sistema, os requisitos estão em estado "Não Decidido" (*Undecided*), e então pode assumir os estados "Sucesso" (*Succeeded*), "Falha" (*Failed*), e no caso de objetivos e tarefas, "Cancelado" (*Canceled*) (SOUZA et al., 2013). É facilmente notável que o processo de elicitação de requisitos de percepção só acontece depois que o modelo de objetivos é levantado, e assim como o processo de construção de objetivos, *AwReqs* devem ser sistematicamente criados.

*EvoReqs* são requisitos que modificam o espaço de comportamento do sistema, permitindo que novas alternativas de requisitos sejam usadas, baseando-se em um conjunto pré-definido de etapas de evolução para os requisitos monitorados (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012). Isto é, *EvoReqs* são requisitos que especificam uma série de operações primárias em relação a outros requisitos diante de determinadas situações, dizendo ao sistema como adaptar-se (SOUZA; LAPOUCHNIAN; MYLOPOULOS, 2012), como por exemplo, instruções para adicionar, remover ou modificar o estado de um objetivo (em nível de instância), desfazer as ações de uma execução que resultou em falha, entre outras (SOUZA et al., 2013).

Em suma, *AwReqs* especificam quando um determinado objetivo precisa de mudanças para continuar a ser atendido, enquanto *EvoReqs* especificam como executar tais mudanças. A seguir, o modelo de exemplo apresentado na seção 2.1 é novamente apresentado, porém com novos requisitos de adaptação que são devidamente discutidos na próxima sessão.

### 2.1.2 Exemplo de Modelagem de Caso de Uso

Na Figura 2 é apresentado o modelo visual completo do sistema de despacho de ambulâncias (*Adaptive Computer-aided Ambulance Dispatch* ou *A-CAD*), nele observa-se o objetivo principal "Gerar Instruções de Despacho Otimizadas", representado por uma oval, que é imediatamente refinado em outros objetivos e em uma pressuposição de domínio (retângulo). O refinamento entre o objetivo raiz e seus filhos imediatos é do tipo "E" e portanto, para que o objetivo principal seja considerado satisfeito, todos as suas decomposições de primeiro grau precisam ser satisfeitas. Então, verifica-se que o primeiro nível de refinamento do objetivo principal é composto dos objetivos "Gerenciar Chamadas", "Identificação de Recursos", "Mobilização de Recursos", "Obtenção de Mapas"e da pressuposição de domínio "Dados sobre recursos está sempre atualizado".

O processo de refinamento do modelo então segue até que todos os objetivos sejam completamente decompostos em tarefas ou pressuposições de domínio. *Softgoals*

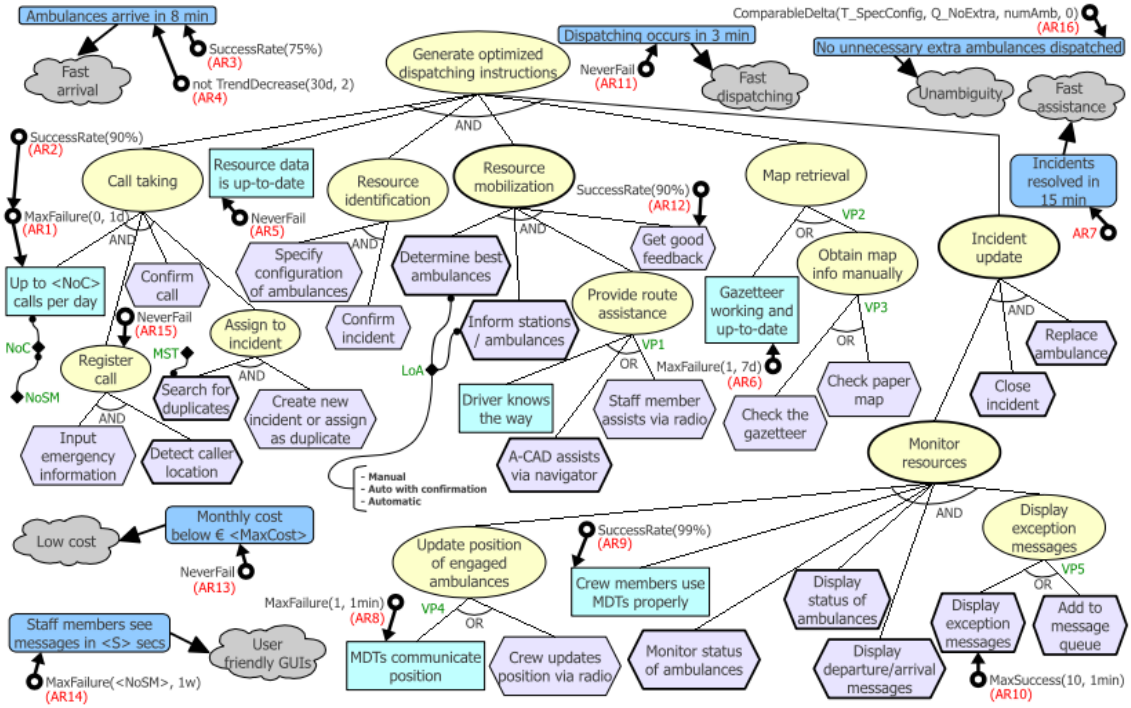


Figura 2 – Exemplo de modelos de objetivos de um sistema de despacho de ambulâncias (SOUZA, 2012)

são simbolizados por nuvens e refinados em critérios de operacionalização representados por retângulos com cantos arredondados. Exemplificando, o *Softgoal* "Chegada Rápida" é operacionalizado por "Ambulâncias chegam em oito minutos", assim, tem-se um critério claro de satisfação para um objetivo que antes possuía diversos tipos de interpretação, porém agora, sabe-se que uma ambulância chega rapidamente se consegue estar no local do acidente em menos de oito minutos a partir da chamada.

Os requisitos de percepção são representados por um círculo oco. Por exemplo, o *AwReq* identificado por "AR15", indica que o objetivo "Registrar Chamados" deve "Nunca falhar" (`NeverFail`). Os *EvoReqs* referentes a cada um dos *AwReqs* não são representados nesse modelo, e devem ser especificados em forma de sequencia de operações sobre os elementos do modelo de objetivos (essa escolha visa aprimorar a legibilidade do modelo). Além dos *AwReqs*, são especificados também os parâmetros de controle (*Control Parameters*), representados por losangos, que indicam parâmetros do sistema que podem ser reconfigurados durante a adaptação. Todas as formas de representação estão resumidas na Figura 3.

Sobre os *EvoReqs* do "AR15", podemos definir seus respectivos Requisitos de Evolução. Primeiramente, se o objetivo vir a falhar, define-se a primeira estratégia de adaptação como "Tentar Novamente em 5 segundos no máximo uma vez" (`RetryStrategy(5000)`), caso falhe mais do que uma vez, aplica-se outra estratégia "Diminuir Condições ao Desabilitar Filho" (`RelaxDisableChild(TDetectCaller)`), que desativa o requisito "Detectar

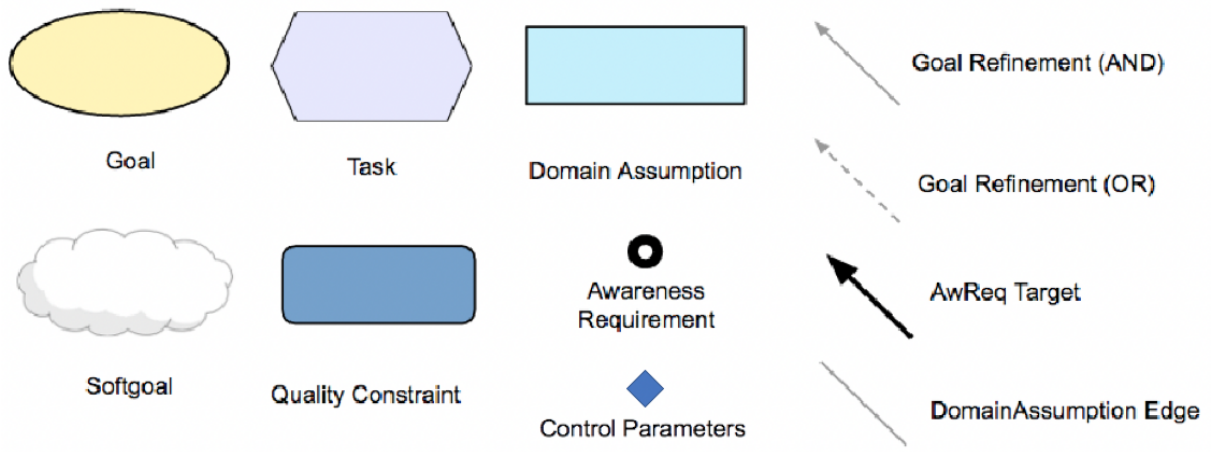


Figura 3 – Representação gráfica de elementos de *GORE*

Tabela 1 – Tabela de especificação das estratégias de adaptação de AR15.

| | | |
|---|---|---|
| AR15 | NeverFail(G_RegCall) | 1. Retry(500)<br>2. RelaxDisableChild(T_DetectCaller) |

Localização da Ligação", também aplicado no máximo uma vez. Essas decisões são sumarizadas na tabela 1.

O processo de especificação de como as estratégias de evolução são realizadas pelo sistema é discutido na próxima seção.

## 2.2 Zanshin

Apresentado por (SOUZA, 2012), *Zanshin* é um *framework* que utiliza de ciclos de retro-alimentação para monitorar sistemas e enviar estratégias de adaptação com base em informações de modelos de requisitos orientados a objetivos enriquecidos com elementos como os *AwReqs* e os *EvoReqs*. O nome *Zanshin* vem de um termo usado em artes marciais japonesas e significa estado de completa consciência.

Em tempo de execução, os elementos do modelo de objetivos são representados por classes e instanciados cada vez que o usuário (ou sistema) busca atingir um objetivo, e então o sistema passa a enviar mensagens a essas instâncias quando detectar falhas. Percebe-se assim, que objetivos e pressuposições de domínio não são tratadas como invariantes que devem sempre ser atingidas, já que o sistema pode falhar ao tentar atingir seus objetivos iniciais, e então o componente de adaptação lidará com essas falhas e tomará medidas para que os objetivos voltem a ser satisfeitos (SOUZA et al., 2013).

Assim como o monitoramento de objetivos através do *Feddback Loop*, o *Zanshin* também monitora Parâmetros (*Parameters*) que podem ser definidos em dois tipos. Primeiro, os Pontos de Variação (ou *Variation Points*), que representam refinamentos do tipo

"OU" (*OR*). Por exemplo, o VP4 do modelo de objetivos do sistema A-CAD (Figura 2) que refere-se ao objetivo "Atualizar posição de ambulâncias em uso" especifica que atualizações de posição das ambulâncias podem ser obtidas automaticamente ou via rádio. Segundo, as Variáveis de Controle (ou *Control Variables*), que são abstrações referentes a pontos de variação repetitivos ou usados em grande escala, como por exemplo a variável MST ("Tempo de Busca Mínimo" ou *Minimum Search Time*), que refere-se a tarefa "Procurar por (incidentes) duplicados". A especificação do comportamento dessas estratégias é discutido em seções posteriores.

O metamodelo dos modelos de objetivos usados no *Zanshin* é representado na Figura 4, e especificado no código do *framework* como um modelo *Ecore* e carregado em memória como objetos Java™ usando um *Framework* do Eclipse™ conhecida como *Eclipse Modeling Framework*. Nessa caso, o modelo *Ecore* representa os tipos de requisitos em nível de classe (SOUZA et al., 2013).

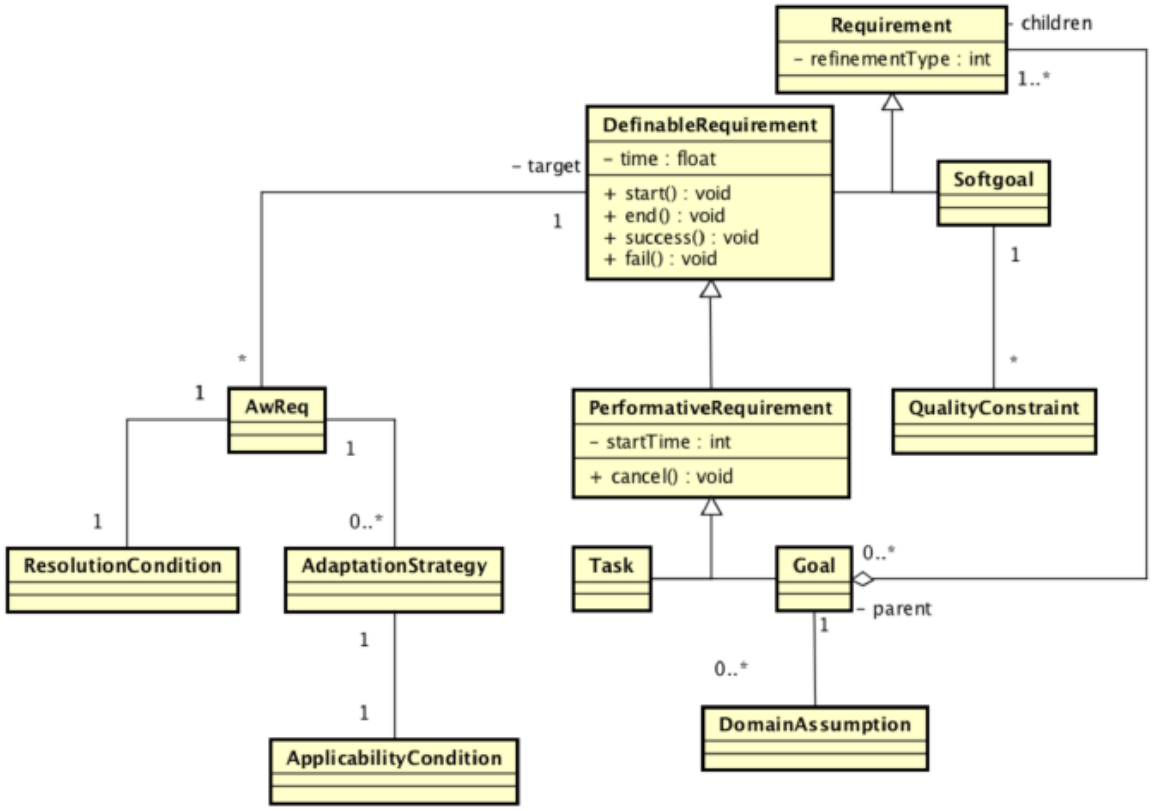


Figura 4 – Metamodelo que define a sintaxe abstrata para o *Zanshin*

O *Zanshin* é ser divido em quatro componentes principais, entretanto essa seção discute desses: o componente de monitoramento e o componente de adaptação.

### 2.2.1 Monitoramento

O módulo de monitoramento necessita que o sistema alvo implemente funcionalidades de registro (*logs*). Através do registro, o sistema pode detectar mudanças nos estados das instâncias de um *AwReq* e assim notificar o serviço de adaptação sobre essas ocorrências.

Para identificar o modelo de objetivos do sistema alvo, o componente de monitoramento necessita da especificação do modelo do domínio também em *Ecore*. Através dele, o *Zanshin* criará instancias desses objetivos estendendo as classes carregadas do metamodelo de *GORE* referentes ao tipo de cada um deles (Figura 4). Por exemplo, ao criar tarefa "Especificar configuração de ambulâncias", o sistema criará uma instancia dessa tarefa específica, que fará referência a classe `Task` criada no momento em que o metamodelo foi importado. A classe de tarefa especializa a classe `PerformativeRequirement` que é uma especialização `DefinableRequirement` (ver Figura 5).

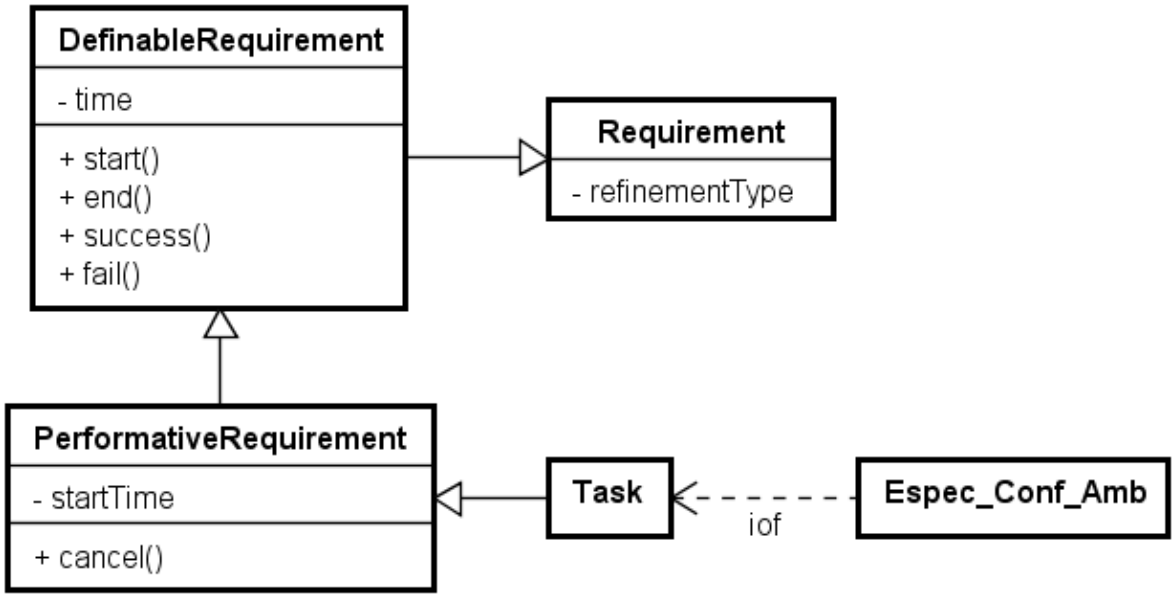


Figura 5 – Exemplo de um elemento do modelo de domínio específico instanciado em *Zanshin*.

*Performative Requirements* definem os tipos de requisitos que são ou podem ser refinados em tarefas, e assim possuem ações que são executadas pelo sistema ou seus usuários. *Definable Requirements* são os requisitos que deve possuir um estado definido em algum momento da execução, como por exemplo os objetivos e pressuposições de domínio.

Assim que o *Zanshin* cria as classes dos requisitos do sistema alvo, o processo de monitoramento apoia nos métodos definidos nas metaclasses `DefinableRequirement` e `PerformativeRequirement` (SOUZA, 2012) para realizar a monitoração, os métodos disponíveis são:

- `start()`: esse método é chamado nas classes de critérios de qualidade e pressuposições de domínio imediatamente antes da análise de seu estado de satisfação. Para o caso de tarefas, esse método é chamado assim que um usuário inicia uma tarefa, e então todos os ancestrais a esse elemento também tem esse método invocado.
- `sucess()`: chamado quando um requisito é satisfeito e propagado para todos os acentrais a esse objetivo para "avisar" que o filho chegou ao estado de sucesso.
- `fail()`: segue a mesma lógica dos item anteriores, propagando a "falha" de um requisito aos seus ancestrais.
- `cancel()`: para o caso de requisitos como objetivos e tarefas, esse método é usado quando um procedimento é cancelado pelo usuário, e usa da mesma lógica de propagação dos métodos anteriores para informar o ocorrido aos pais.
- `end()`: chamado assim que um requisito muda para os estados de sucesso, falha ou cancelamento.

Através da descrição dos métodos acima mencionados, é possível entender como o processo de monitoramento funciona: basicamente os requisitos monitorados possuem métodos que podem ser chamados em cada caso que assumem, seja de sucesso, falha o ou cancelamento. Assim que chamados, os métodos invocam os mesmos procedimentos nos elementos pais, permitindo que a atualização de um requisito se propague a todos os elementos que interessem, atualizando assim todo o modelo.

Logo que o componente monitor detecta uma mudança de estado em um requisito, ou seja, assim que qualquer um dos métodos acima é invocado, ele imediatamente envia uma notificação sobre essa mudança ao módulo de adaptação, que então inicia o processo definido para aquele determinado requisito de percepção (SOUZA, 2012).

### 2.2.2 Adaptação

Em termos de implementação, o algoritmo do módulo de adaptação cria uma sessão para cada *AwReq* que está sendo monitorado e então gera uma fila de eventos que se referem a estratégias adaptativas. Assim, essa linha de eventos pode ser usada para verificar se uma estratégia é ou não aplicável a determinada situação. Primeiramente, deve-se verificar o estado de um objetivo apontado por um *AwReq*, para isso é checada a condição de resolução daquele. Então, caso uma avaliação considere que o objetivo está em estado de falha, o sistema segue (seguindo uma ordem pré-definida) a lista de estratégias de adaptação disponíveis, procurando alguma que tenha sua condição de aplicabilidade verdadeira. Caso encontre, aplica a estratégia definida por aquela condição de aplicabilidade e então retorna ao estado inicial, onde a checagem da satisfabilidade do objetivo é realizada novamente. Caso o sistema retorne a fase de checagem do estado do objetivo e esse ainda

não tenha sido satisfeito, o processo começa novamente, obedecendo restrições como o número de tentativas de aplicação de uma estratégia, definido individualmente. Caso não encontre uma condição para aplicar uma estratégia, o sistema ativa o método "abortar" (`abort()`) (SOUZA et al., 2013).

Ao final, quando uma sessão de adaptação é considerada resolvida, a mesma deve ser terminada e se for necessário que o processo seja aplicado novamente, uma nova sessão é criada. Entretanto, se uma sessão termina sem ter resolvido o problema, o *framework* continuará trabalhando nela do ponto em que parou assim que receber uma nova requisição para adaptação daquele mesmo *AwReq*. Porém, algumas estratégias também podem forçar que a sessão seja reiniciada quando executada (SOUZA et al., 2013). Essa processo é conhecido como Evento de Ação-Condição (*Event Condition Action* ou *ECA*) (MORIN et al., 2009).

#### 2.2.2.1 ECA

O código mostrado em 2.1 resume o processo *ECA* para realizar a seleção de estratégias de adaptação:

Listagem 2.1 – Código do processo ECA

```
1  processEvent(ar : AwReq) {
2    session = findOrCreateSession(ar.class);
3    session.addEvent(ar);
4    solved = ar.condition.evaluate(session);
5    if(solved) break;
6
7    ar.selectedStrategy = null;
8    for each s in ar.strategies {
9      appl = s.condition.evaluate(session);
10     if (appl) {
11       ar.selectedStrategy = s;
12       break ;
13     }
14   }
15
16   if (ar.selectedStrategy == null)
17     ar.selectedStrategy = ABORT;
18
19   ar.selectedStrategy.execute(session);
20   ar.condition.evaluate(session);
21 }
```

O algoritmo de 2.1 incia obtendo a sessão de adaptação referente a classe do *AwReq* requisitado (caso não haja uma sessão, uma nova é criada). Obtida a sessão de adaptação, o algoritmo tem então acesso a lista de eventos de aplicabilidade referentes, então, adiciona o *AwReq* a sessão e imediatamente verifica o estado da mesma (verificando a condição de resolução), parando caso tenha retornado estado de sucesso. Caso contrário, o processo continua procurando por uma estratégia que seja aplicável, verificando se a condição de

aplicabilidade é verdadeira; se sim, interrompe o processo e dá a sessão de adaptação uma nova chance de verificar o estado do *AwReq*. Caso todas as condições sejam falsas e nenhuma estratégia seja selecionada, seleciona a estratégia padrão (`abort()`) e termina o processo (SOUZA, 2012).

Para exemplificar esse processo, tomemos novamente o *AwReq* "AR15", que garante que o requisito "Registrar chamada" deve nunca falhar. Caso seja detectado pelo processo de monitoramento que esse requisito apresenta estado de falha, o módulo monitor imediatamente ativa o componente de adaptação, que segue o processo ECA para aplicar estratégias ao "AR15". Pela Listagem 2.2, ve-se que a condição de resolução desse requisito é do tipo `SimpleResolutionCondition` (solucionado se os filhos, dependendo do refinamento, estão solucionados), e que a primeira estratégia a ser selecionada é `RetryStrategy`, ou seja, tentar novamente (em 5s), entretanto essa estratégia possui a condição de aplicabilidade `MaxExecutionsPerSessionApplicabilityCondition`, o que significa que ela só pode ser aplicada um determinado número de vezes naquela sessão (nesse caso apenas uma vez, de acordo com o atributo `maxExecutions`). Caso essa condição seja falsa, a próxima estratégia refere-se a desativar um dos filhos desse objetivo (`RelaxDisableChildStrategy`), no caso a tarefa "Detectar localização da chamada", e possui a mesma condição de aplicabilidade. Se nenhuma dessas condições puder ser satisfeitas, então o sistema aborta, caso contrário, seleciona a primeira estratégia aplicável e verifica novamente o estado do objetivo. A condição `SimpleResolutionCondition` refere-se ao fato de que um objetivo é dito satisfeito apenas se seus filhos estiverem em estado de sucesso (respeitando a regra booleana do refinamento).

Listagem 2.2 – Estratégias de adaptação de AR15

```
1 <awreqs xsi:type="acad:AR15">
2   <condition xsi:type="eca:SimpleResolutionCondition"/>
3     <strategies xsi:type="eca:RetryStrategy" time="5000">
4       <condition xsi:type="eca:MaxExecutionsPerSessionApplicabilityCondition"
            maxExecutions="1"/>
5     </strategies>
6     <strategies xsi:type="eca:RelaxDisableChildStrategy" child="//@rootGoal/
          @refinements.0/@refinements.0/@refinements.1">
7       <condition xsi:type="eca:MaxExecutionsPerSessionApplicabilityCondition"
            maxExecutions="1"/>
8     </strategies>
9 </awreqs>
```

É importante salientar que as classes referentes a estratégias de adaptação, assim como as condições de aplicabilidade e resolução podem ser estendidas para casos mais complexos, envolvendo inclusive a interação humana (SOUZA, 2012). Uma lista completa das estratégias de adaptação disponíveis por padrão no *Zanshin* é mostrada na Tabela 2.

Tabela 2 – Tabela de Requisitos de Adaptação

| Regra | Efeito |
|---|---|
| Abort() | Sistema deve falhar de forma prevista, exibindo mensagem de |
| Delegate(a) | Delegar tarefa a um outro ator do sistema. |
| RelaxDisableChild(r, l, child) | Para de considerar o estado de um requisito em determinada |
| Replace(r, copy, l, newReq) | Substitui requisito. |
| Retry(copy, time) | Tenta novamente em determinado período de tempo. |
| StrengthenEnableChild(r, l, child) | Volta a considerar estado de um requisito. |
| Warning(a) | Avisa um ator sobre o estado atual do sistema. |

## 2.3 Desenvolvimento Orientado a Modelos

Pesquisadores vem tentando ao longo dos anos criar abstrações que ajudem programadores a focar no conteúdo do desenvolvimento ao invés das especifidades da tecnologia de criação adotada (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). O Desenvolvimento Orientado a Modelos pode ser visto como a forma de programação de mais alto nível de abstração existente atualmente (ATKINSON; KUHNE, 2003), promovendo o uso de artefatos do processo de desenvolvimento de software para lidar com complexidade através de abstração (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). Em outras palavras, MDD parte da premissa que um sistema é um modelo consistente com seu metamodelo (VUJOVIĆ; MAKSIMOVIĆ; PERIŠIĆ, 2014) e assim, em vez de exigir que programadores escrevam cada simples detalhe da implementação de um sistema, permite que uma funcionalidade necessária para um software pode ser visualmente modelada (ATKINSON; KUHNE, 2003). Portanto, essa técnica viabiliza que muitas atividades complexas (porém rotineiras) sejam automatizadas na área de programação de software, como por exemplo o suporte a persistência, interoperabilidade e distribuição (ATKINSON; KUHNE, 2003).

A modelagem usa da percepção visual humana para melhorar o processo de compreensão sobre o domínio de um software, já que modelos nos auxiliam a entender problema complexos e suas possíveis soluções através da abstração. Assim, MDD baseia-se na premissa de que o desenvolvimento de software deve focar principalmente na produção de modelos e não na criação de código (SELIC, 2003). A primeira vantagem dessa abordagem é que podemos usar conceitos mais ligados ao domínio do problema que software vai resolver do que conceitos técnicos ligados a linguagem de programação, em consequência essa vantagem acarreta em alguns outros benefícios: modelos são mais compreensíveis do que códigos e portanto, tornam-se também mais fáceis de especificar e manter (SELIC, 2003). Além disso, modelos são menos sensíveis a alterações de tecnologias, ou seja, são independentes de plataforma (SELIC, 2003). Esse trabalho foca em MDD como ferramente de produção automática de código através da interpretação de modelos visuais (SELIC, 2003; VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014).

### 2.3.1 Eclipse Modeling Framework

O Eclipse™ é um projeto de código aberto com o objetivo de prover uma plataforma de desenvolvimento altamente integrada. O processo de criação de sistemas no Eclipse™ pode ser divido em alguns projetos, entre eles o Projeto de Modelagem (*Modeling Project*), que foca em tecnologias baseadas no desenvolvimento orientado a modelos (STEINBERG et al., 2008). Esse ambiente é chamado de *Eclipse Modeling Framework* ou *EMF*, e provê funcionalidades como transformação de modelos, integração de bases de dados e geração de editores gráficos (STEINBERG et al., 2008). Modelos especificados através de EMF relacionam abstrações de modelagem diretamente a seus conceitos de implementação, sendo a união das tecnologias UML, XML e Java™, permitindo a conversão automática entre todas essas ferramentas (STEINBERG et al., 2008).

A tecnologia EMF pode ser melhor explicada com um exemplo: um sistema de gerenciamento de ordem de compras de uma loja, que necessite incluir casos como "cobrar" e "entregar" em um endereço, e uma coleção de itens (nesse caso compras). A figura 7 mostra o diagrama em UML do sistema. Esse modelo pode também ser descrito dentro do EMF usando modelos *Ecore*, como na Figura 8. Modelos *Ecore* são basedos no metamodelo para especificação exibido na Figura 6, onde classes são representadas por `EClass`, atributos por `EAttribute`, relações por `EReference` e tipos de dados por `EDataType`. Assim, a "conversão" do modelo UML para *Ecore* é dada ao se instanciar classes de *Ecore* de acordo com a especificidade do domínio do problema, como pode ser visto na Figura 9 (STEINBERG et al., 2008).

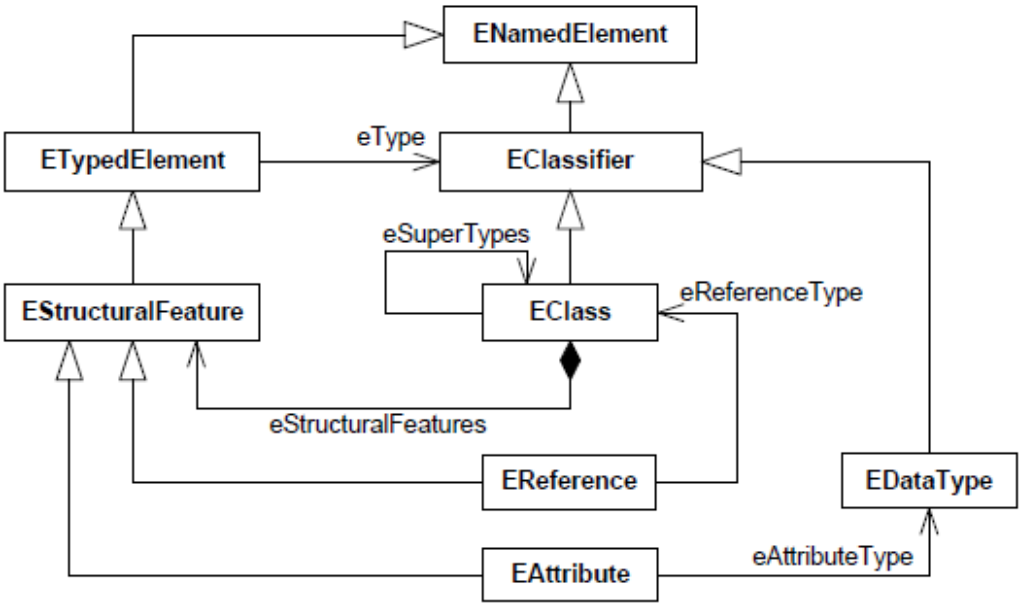


Figura 6 – Metamodelo de *Ecore* (KERN, 2008)

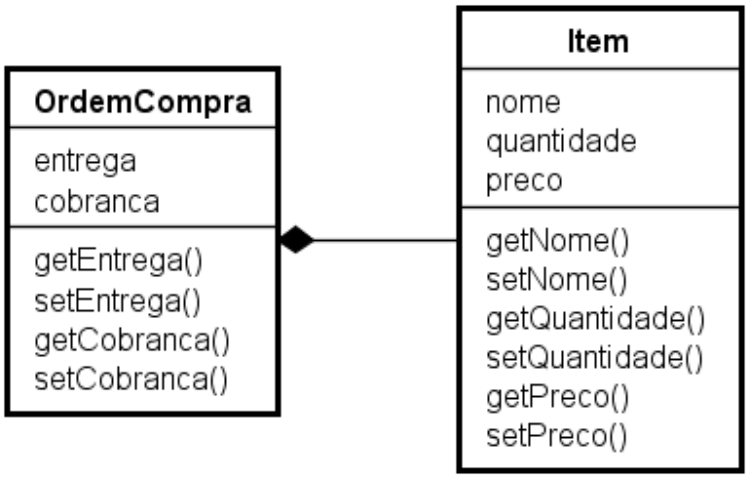


Figura 7 – Classes do diagrama do exemplo em UML

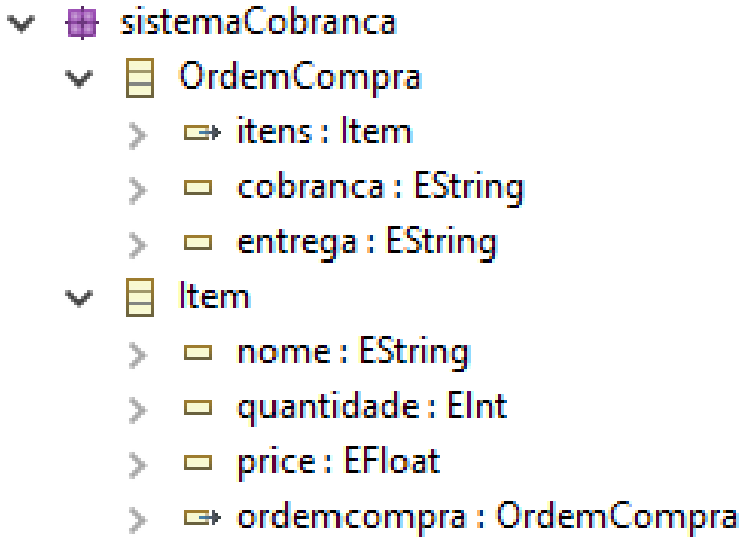


Figura 8 – Classes do diagrama do exemplo em *Ecore*

Assim que o modelo é detalhado em *Ecore*, o EMF está pronto para gerar código automaticamente, seguindo os seguintes passos:

- Para cada tipo `EClass` são criadas uma interface e a classe de implementação correspondente. Então para o exemplo de classe `OrdemCompra` serão criadas a interface `OrdemCompra` e a classe `OrdemCompraImpl`. Essa especificação permite que sejam implementadas funções de persistência e distribuição, porém não serão discutidas aqui por fugirem do escopo desse trabalho.
- As classes do tipo `EAtributte` são transformadas em atributos nas classes correspondentes
- As classes tipo `EReference` são transformadas em referências nas respectivas classes

as quais referenciam.

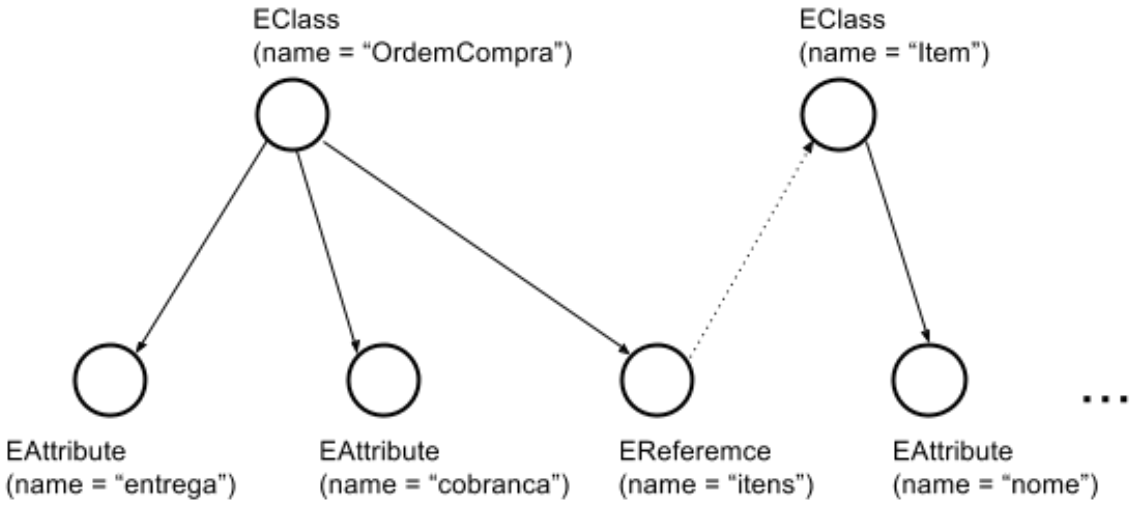


Figura 9 – Conversão do diagrama de exemplo de UML para *Ecore*

Em suma, o *Framework* de modelagem do Eclipse™ permite que usuários criem modelos apoiados em metamodelos, e baseando-se no modelo criado, gerem partes de código de sistema (VUJOVIĆ; MAKSIMOVIĆ; PERIŠIĆ, 2014).

### 2.3.2 Sirius

O Sirius™ é um plugin que simplifica o *Framework* de Modelagem Gráfica (*Graphical Modeling Framework* ou *GMF*) do Eclipse™, reduzindo a complexidade de uso do mesmo e permitindo a produção de editores gráficos de modelos personalizados (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). O Sirius™ é construído em cima do fato de que o Eclipse™ provê utilidades para (des)serialização de modelos, checagem de condições e geração de editores baseados em *Ecore* (BUDINSKY, 2004). Representações criadas com Sirius™ podem ser apresentadas em diagramas, tabelas e árvores (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). Em síntese, Sirius™ provê ferramentas que permitem a especificação de um modelo de um domínio qualquer em diferentes perspectivas gráficas (VUJOVIĆ; MAKSIMOVIĆ; PERIŠIĆ, 2014).

Através de modelos de especificação (*Viewpoint Specification Model* ou VSM), o *plugin* permite especificar a estrutura, aparência e comportamento do metamodelo do editor a ser criado (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014). Os VSMs são especificados através de arquivos *.odesign* (VIYOVIĆ; MAKSIMOVIĆ; PERISIĆ, 2014), e baseam-se metamodelo de domínio descrito em *Ecore*. Um exemplo de caracterização de representação de modelos é mostrado na figura 10.

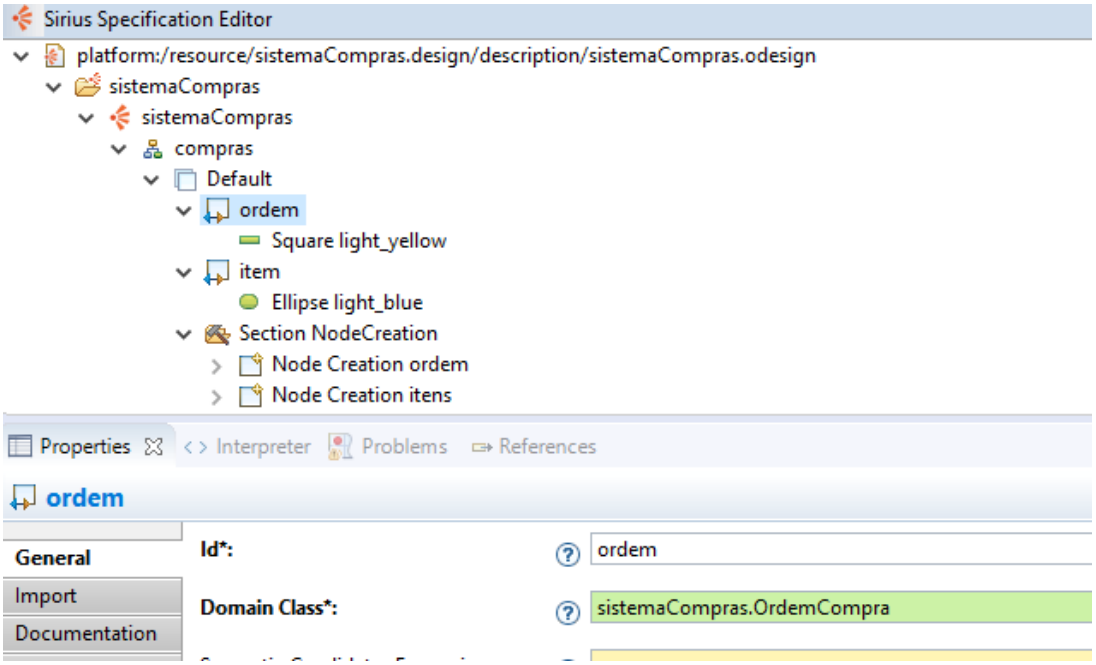


Figura 10 – Exemplo de *Viewpoint Specification Model*

Sirius™ provê vários mecanismos que auxiliam no gerenciamento da complexidade de modelos (MADIOT; PAGANELLI, 2015):

- **Camadas**: é possível separar elementos em camadas diferentes, e ativar ou desativar essas.
- **Filtros**: exibir ou esconder elementos dependendo de determinada condição.
- **Fersonalização de Estilos**: permite modificar propriedades gráficas de elementos do diagrama.
- **Regras de Validação**: permite que a qualidade do modelo seja avaliada.

# 3 Zanshin

Esta seção trata do processo de reavaliação do metamodelo operacional do sistema *Zanshin*, anteriormente já apresentado na Figura 4.

## 3.1 Motivação

Durante estudos realizados sobre a implementação do *Framework Zanshin*, detectaram-se algumas oportunidades de melhoria relacionadas principalmente ao metamodelo de objetivos utilizado pelo sistema. A maioria das oportunidades de melhoria identificadas referem-se a dois motivos: ou o metamodelo não era restrito o suficiente, deixando que algumas situações indesejadas pudessem ser modeladas; ou o metamodelo não refletia mais alguns conceitos de *GORE* que foram melhores elaborados em discussões recentes sobre o tema.

## 3.2 Revisão do Metamodelo

Uma análise de cada elemento do metamodelo antigo é feita a seguir:

1. **Requirement**: o fato desse elemento estar no topo da hierarquia e todos os outros elementos serem especializações dele faz com que suas características sejam herdadas por todos os outros componentes do modelo. Porém, nota-se que algumas características na verdade são inerentes a apenas alguns elementos. Por exemplo: *Requirement* possui a relação de agregação *Parent* para *Children*, permitindo que cada elemento tenha zero ou um "Pai" e um ou vários "Filhos". Entretanto, a agregação pai/filho é indesejada nos elementos *AwReq*, *DomainAssumption* e *QualityConstraint*, visto que esses não são refinados, apenas possuem elementos alvo.

2. **DeinableRequirement** e **Softgoal**: *Softgoal* possui o mesmo tipo de comportamento de um *DefinableRequirement* e portanto, seria esperado que aquele também fosse uma especialização deste. Assim, com essa modificação faz-se-ia desnecessário a utilização de duas classes (*Requirement* e *DefinableRequirement*) na composição do modelo, que poderiam perfeitamente ser reduzidas a apenas uma.

3. **Softgoal** e **QualityConstraint**: baseando-se no fato de que *Softgoals* são compostos de *Quallity Constraints* pode-se identificar que a relação mais apropriada a esses dois elementos seria a relação de composição. Além disso, o modelo permite que um *Softgoal* contenha nenhuma operacionalização, o que não faria sentido pois assim

não seria possível identificar quando esse elemento foi bem sucedido ou não.

4. **DefinableRequirement** e **AwReq**: de acordo com as propostas apresentadas na literatura atual para os *AwReqs*, fica claro que esses elementos, ao se referirem a qualquer outro tipo de elemento e tratarem especificamente do elemento ao qual se referem, deveriam ter na verdade uma relação de composição com o elemento mais alto da hierarquia de especializações, permitindo então que todos os outros elementos do modelo contessem *AwReqs*.

5. **Goal** e **Task**: *Tasks* são refinamentos de *Goal* e por isso seria importante que houvesse algum tipo de relacão direta entre os dois elementos, onde fosse possível identificar que quando um objetivo fosse operacionalizado por tarefas e quais seriam essas tarefas. Além disso, identificou-se uma oportunidade de melhoria da nomenclatura de "*Goal*" para "*HardGoal*", refletindo assim uma melhor impressão sobre o papel do elemento e a diferença entre ele e *Softgoal*.

6. **DomainAssumption**: não é claro no modelo a forma como *DomainAssumptions* se relacionam com os outros elementos.

## 3.3 Proposta

Após levantamento das melhorias possíveis, foi elaborado o metamodelo apresentado na Figura 11.

Primeiramente, pode-se observar que o as especialiações e relações de agregação e composição entre os elementos foram refinadas, assim, identifica-se que:

- A partir de agora, todos os elementos são especializações de *GOREElement*, que passa a englobar características das antigas classes *Requirement* e *DefinableRequirement*, resolvendo então parte do problema [2].

- A agregação pai/filho agora ocorre em dois tipos e acontece em um elemento mais específico: um requisito pode ser decomposto (*decompose*) em outros, referindo-se assim a refinamentos do tipo "E", ou refinado através da agregação denominada "alternative", que trata de alternativas a um requisito, ou seja, refinamentos do tipo "OU". Dessa forma o metamodelo pode retratar melhor os tipos de refinamentos admitidos e quais são, exatamente, os componentes que os aceitam. Além do mais, a partir desta melhoria os refinamentos acontecem somente nos elementos que especializam *GoalOrientedRequirement*, ou seja, somente *Hardgoals*, *Softgoal* e *Tasks*. Essa modificação soluciona o problema [1].

- Nessa nova proposta, há uma relação de decomposição entre o elemento raiz (*GOREElement*) e *AwReq*, portanto, explicita-se a noção de que qualquer requisito pode

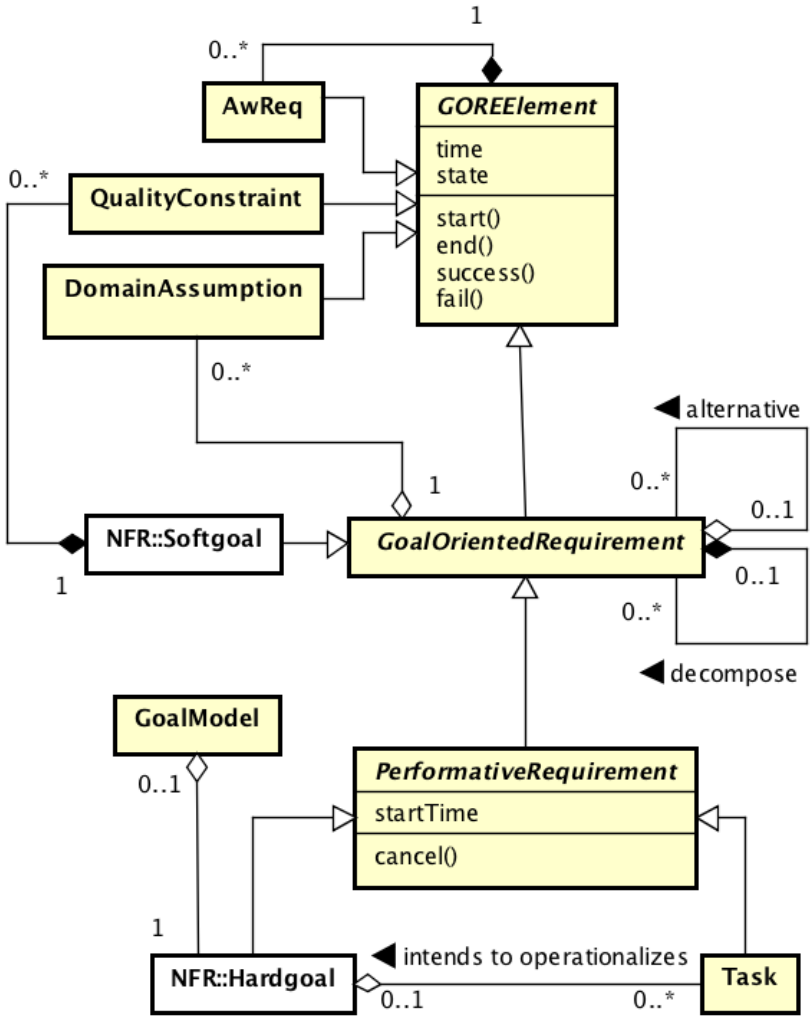


Figura 11 – Proposta de evolução do metamodelo do *Framework Zanshin*

conter requisitos adaptativos, eliminando a outra parte do problema [2]. Isso também tem um efeito positivo em relação a especificação do XML usado para criar modelos específicos de domínio, que fica mais claro e conciso.

- *Softgoal* e *QualityConstraint* passam a se relacionar por meio de relação de decomposição, e portanto também fica mais evidente que todo *Softgoal* deve ser operacionalizado por pelo menos um *QualityConstraint*, dando fim assim ao problema [3].

- Nessa proposta, *AwReqs* tem uma relação direta de composição com o *GOREElement*, e portanto, define-se que qualquer tipo de requisito no modelo pode conter elementos de adaptação, assim resolve-se o problema [4].

- No novo metamodelo, foi criada uma relação direta entre *Tasks* e *Hardgoals* (previamente apenas *Goals*), assim fica explícito o relacionamento entre esses dois elementos, sinalizando que objetivos são refinados em tarefas, que podem ser refinadas apenas nelas mesmas. Soluciona-se então o problema [5].

- Propõe-se, por fim, uma relação de agregação entre *DomainAssumption* e *GoalOrientedRequirement*, assim fica claro que *Domain Assumptions* são elementos que interagem apenas com Objetivos e Tarefas (*Hardgoals*, *Softgoal* e *Tasks*), findando o problema [6].

Uma observação importante que deve ser feita refere-se ao fato de que um elemento só pode conter refinamentos exclusivamente do tipo "E" ("AND") ou exclusivamente do tipo "OR". Essa decisão tem por objetivo facilitar a leitura e interpretação do diagrama, e não compromete a representatividade do modelo pois assume-se que se necessário, pode-se agrupar refinamentos do mesmo tipo em um objetivo, como exemplificado na Figura 12.

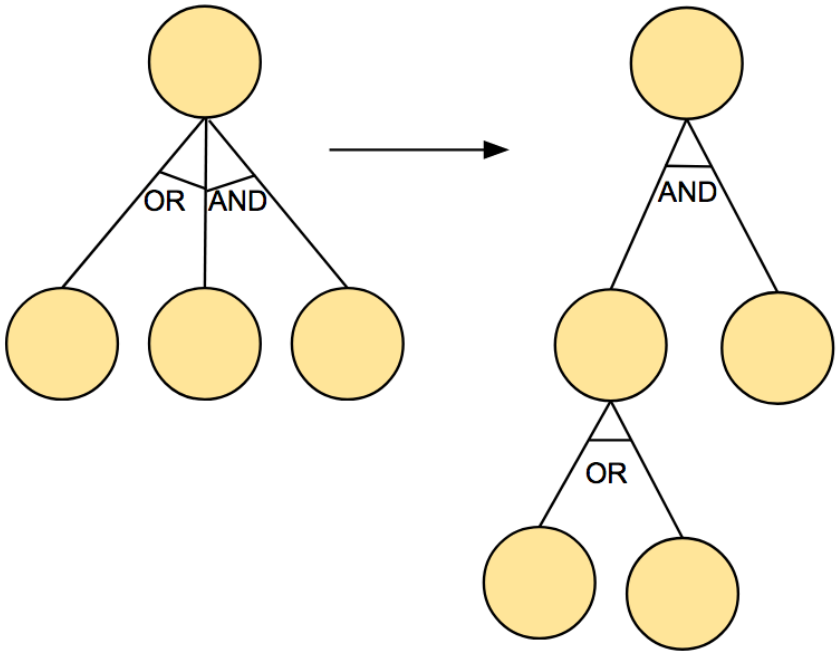


Figura 12 – Exemplificação da representação de refinamentos no novo metamodelo.

# 4 Unagi

O editor gráfico *Unagi* foi desenvolvido com o principal objetivo de facilitar a criação de arquivos de especificação do modelo de dominio para uso no *Zanshin*, considerando que até então não havia nenhuma ferramenta que automatizasse esse processo, assim o usuário precisaria escrever o arquivo XML e *Ecore* manualmente.

Usando um ferramental disponível na plataforma Eclipse™ para criação de editores gráficos usando EMF, além da utilização de aspectos de desenvolvimento orientado a modelos, o processo de desenvolvimento do *Unagi* pode ser dividio em duas fases: a criação do editor gráfico usando Sirius™ e o desenvolvimento do conversor de diagramas apoiado em código gerado automaticamente por *Model Driven Development*.

## 4.1 Criação do Editor Gráfico

A primeira parte do desenvolvimento da ferramenta consitiu na criação do editor gráfico para modelagem de diagramas de objetivos para sistemas de domínios específicos. Devido ao fato de *Zanshin* ser um sistema que também se baseia nas ferramentas da plataforma Eclipse™, aliado ao fato dessa plataforma também prover todos os utensílios necessários para o desenvolvimento de editores de diagramas, como *Ecore*, EMF e o plugin Sirius™, foi fácil decidir que para o desenvolvimento do *Unagi* também seriam usados os mesmos recursos, com objetivo assim de permitir maior compatibilidade com o *Zanshin* em trabalhos futuros.

Inicialmente, o mesmo modelo *Ecore* usado para operacionalização dos requisitos no *Zanshin* foi usado para geração automática de código dentro do Eclipse™ EMF. É importante salientar que junto com as classes específicas de domínio, o EMF também gera código automático para criação de editores e para criação de processos de teste e validação (Figura 13). Assim, após completa especificação do *Ecore*, foi gerado código automático para implantação do editor. O código gerado pode então ser executado como uma aplicação Eclipse™ e a partir daí pode-se criar um editor de modelos personalizado usando o Sirius™.

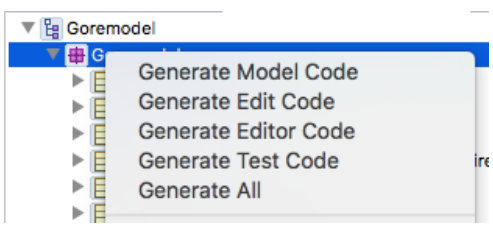


Figura 13 – Geração automática de código no Eclipse™

Rodando na instância do Eclipse™ a partir do código de editor gerado automaticamente, o plugin Sirius™ permite que sejam definidos pontos de perspectiva com diferentes especificações de representação gráfica para os objetos do modelo *Ecore*. Essa especificação é realizada a partir da criação de Projetos de Especificação de Perspectiva (*Viewpoint Specification Project* ou *VSP*), onde são definidos como os elementos serão representados, como suas instâncias serão armazenadas nas classes do metamodelo *Ecore*, qual o comportamento do modelo quando relações são criadas, dentre outros.

A princípio, é criado dentro do modelo de VSP uma camada (*layer*) que representa a perspectiva a ser representada, e nelas são especificados os elementos do modelo *Ecore* que serão representados, aqui chamados de *Nodes*, que pode ter um estilo padrão ou um estilo condicionado a determinada situação, além disso, pode-se usar formas geométricas já existentes ou importar imagens externas.

Como visto na Figura 14, o elemento *Goal* tem como representação padrão uma elipse amarela. Essa representação pode ser melhor configurada através da paleta de propriedades da mesma (Figura 15), onde podem ser especificadas características como cor, cor da linha e cor do nome.

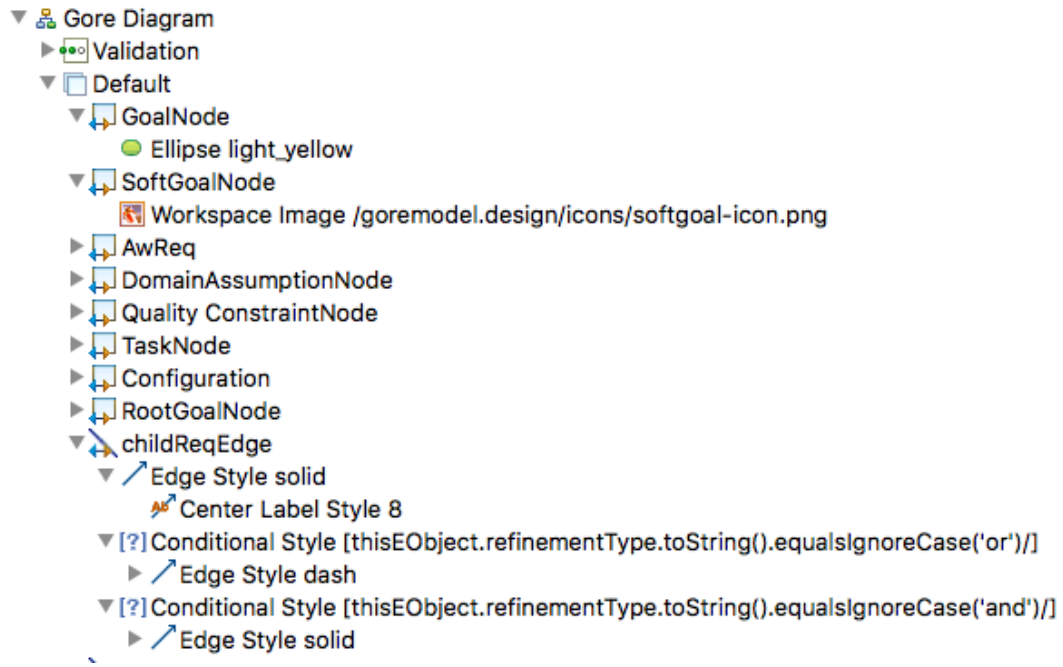


Figura 14 – Sirius Viewpoint Specification Project

Na Figura 14 também pode-se observar além da representação dos *Nodes*, é necessário configurar a criação das linhas de ligação entre os elementos, pois através dessas são representados os refinamentos entre os componentes do modelo. Em `childReqEdge` é especificado que se um objeto possui tipo de refinamento "OU" ("OR"), deve ser usado o tipo de linha tracejada, entretanto, se for do tipo "E" ("AND"), usa-se linha sólida. A linguagem usada para escrita de termos condicionais é conhecida como Acceleo Query

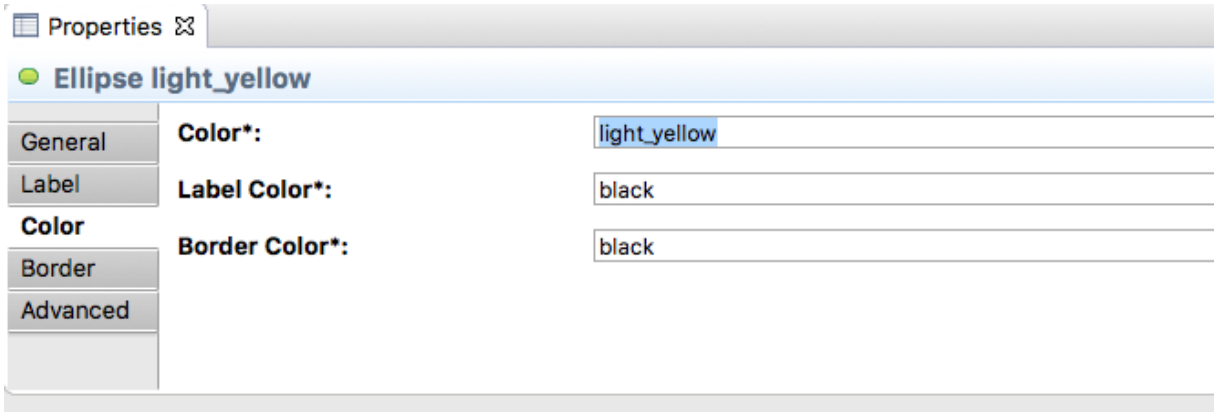


Figura 15 – Propriedades de Representação de Elemento em Sirius™

Language (MUSSET et al., 2006).

Além da caracterização dos elementos que serão representados naquela perspectiva, Sirius™ também necessita que sejam configuradas propriedades de instanciação dos elementos, assim, deve criar além dos *Nodes* de represntação, os *Nodes* de criação, mostrados na Figura 16, onde podem ser observados, por exemplo, a criação do *Node Goal*. Nessa etapa é necessario que primeiramente seja especificado em que tipo de instância do metamodelo o novo objeto será armazenado, para o *Node* de exemplo pode-se observar que há duas ocasiões:

- Caso o elemento for o elemento pai, ou seja, a raiz da árvore de representação, deve ser armazenado em uma variável especial da classe *GoalModel*, de nome *rootGoal*. Esse caso é verificado ao verificar se a variável ainda não foi defindа (`Case[oclIsUndefined(container.rootGoal)]`).
- Caso o elemento for refinamento de qualquer nível do elemento pai, então é guardado na variável *children* da classe *GoalModel*.

Decididos os casos, é necessário criar uma instancia da classe relativa ao novo elemento, esse processo é definido através das propriedades da opção *Create Instance*(Figura 16) mostrada na Figura 17. O campo *referenceName* deve se referir a variável da classe de contenimento que será usada para armazenar o novo elemento, enquanto *Type Name* refere-se a classe do novo componente do modelo.

Por fim, foi modelado uma diferente perspectiva para especificação das estratégias de adaptação dos *AwReqs*, permitindo que cada um tenha sua propria representação em um diagrama separado, com o objetivo de evitar que o modelo principal ficasse "poluído" por excesso de informação.

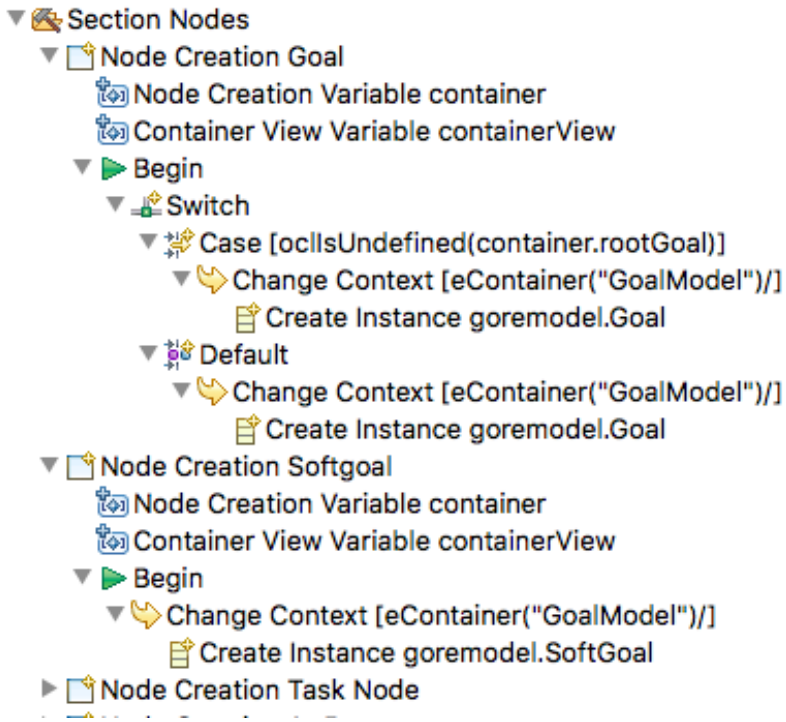


Figura 16 – Criação de Nodes no Sirius$^{TM}$- Exemplo de criação de nó

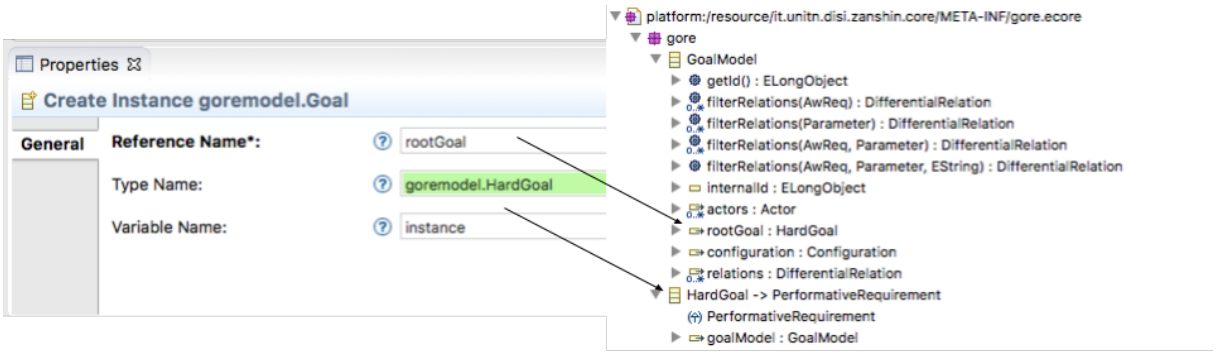


Figura 17 – Criação de Nodes no Sirius$^{TM}$- Exemplo de criação de instancia de nó

### 4.1.1 O Editor Gráfico

Após todo o processo de detalhamento da representação dos elementos do editor, é possivel então criar um model de especificação de perspectiva (*Viewpoint Specification Model*), que permite a criação do diagrama usando recursos de arrastar e soltar, intuitivos por estarem presentes na maioria dos editores gráficos atuais. A Figura 18 mostra o editor em execução, onde pode ser viso a área de modelagem, bem como a paleta de elementos que podem ser criados apenas arrastando-os para a área de desenho, assim como pode-se notar a presença dos tipos de refinamentos disponíveis para modelagem, que podem ser selecionados e então desenhados apenas clicando no elemento de origem e depois clicando no elemento destino. As propriedades relativas a cada elemento do modelo podem ser modificadas por meio da paleta de opções que aparece na área inferior do editor (Figura 19).

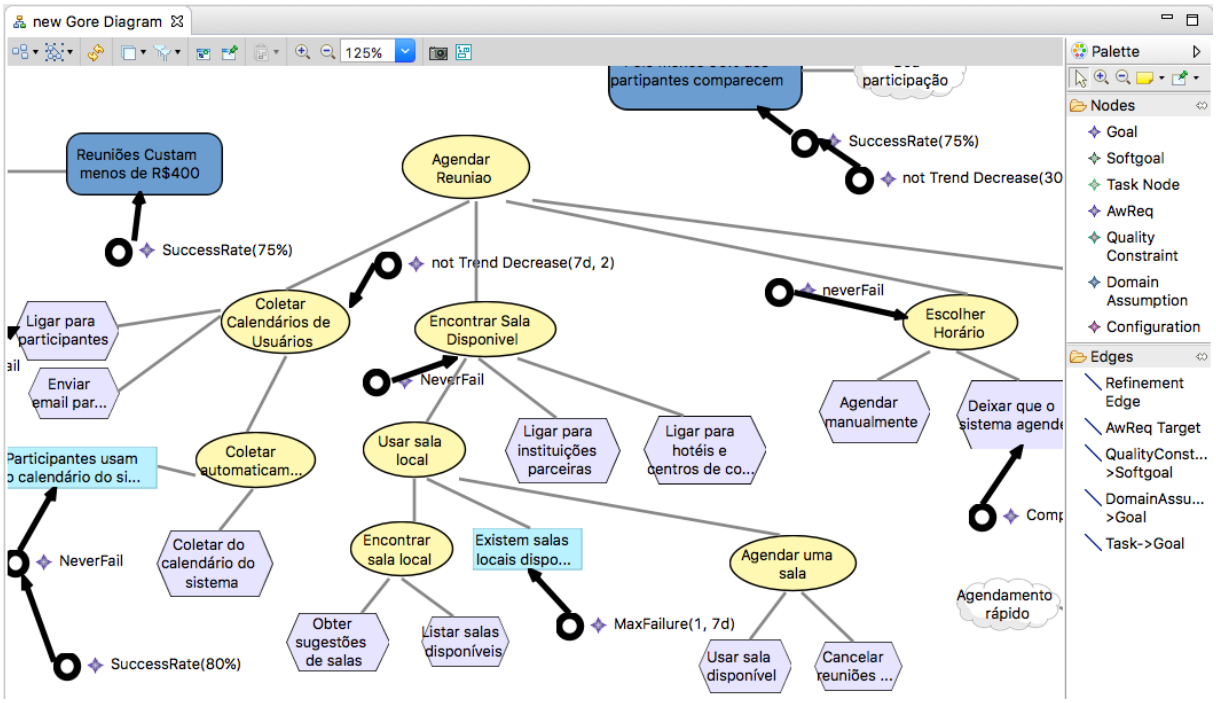


Figura 18 – Editor Unagi

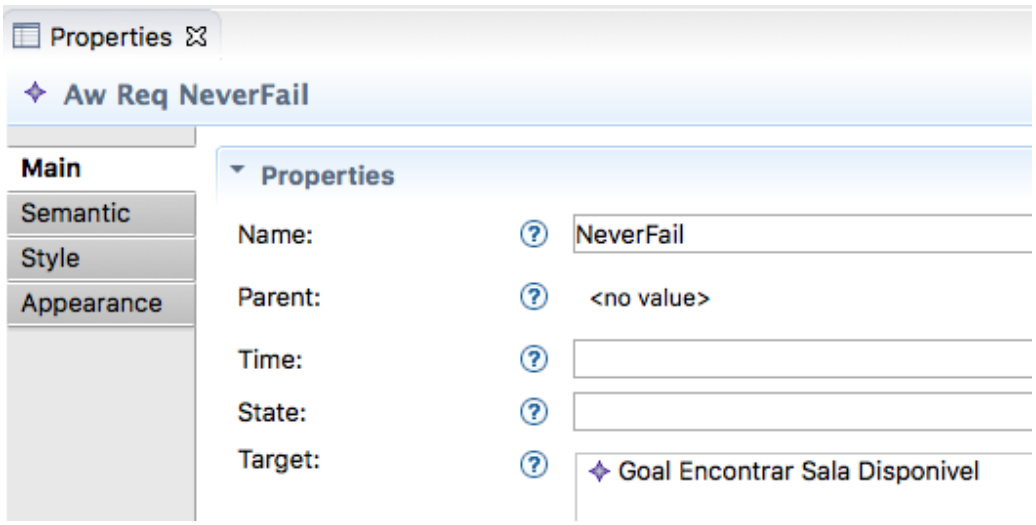


Figura 19 – Editor Unagi - Paleta de Opções

Para acessar a perspectiva que permite o detalhamento dos *EvoReqs* referentes a cada *AwReq* é necessário simplesmente um clique duplo sobre o Requisito de Percepção deseja, o editor entao oferece a opção de criação de um subdiagrama de detalhamento dos elementos, mostrado na Figura 20. Assim, o usuário pode determinar os as Condições de Resolução, as Condições de Aplicabilidade e as Estratégias de Adaptação usando a mesma lógica de arrastar e soltar, selecionando os elementos na parte direita da tela. É importante dizer que o modelo só aceita a criação de um elemento específico após seu elemento "pai"ter sido criado, por exemplo, só é possível criar uma Estratégia de Adaptação apoós ter definido uma Condição de Aplicabilidade.

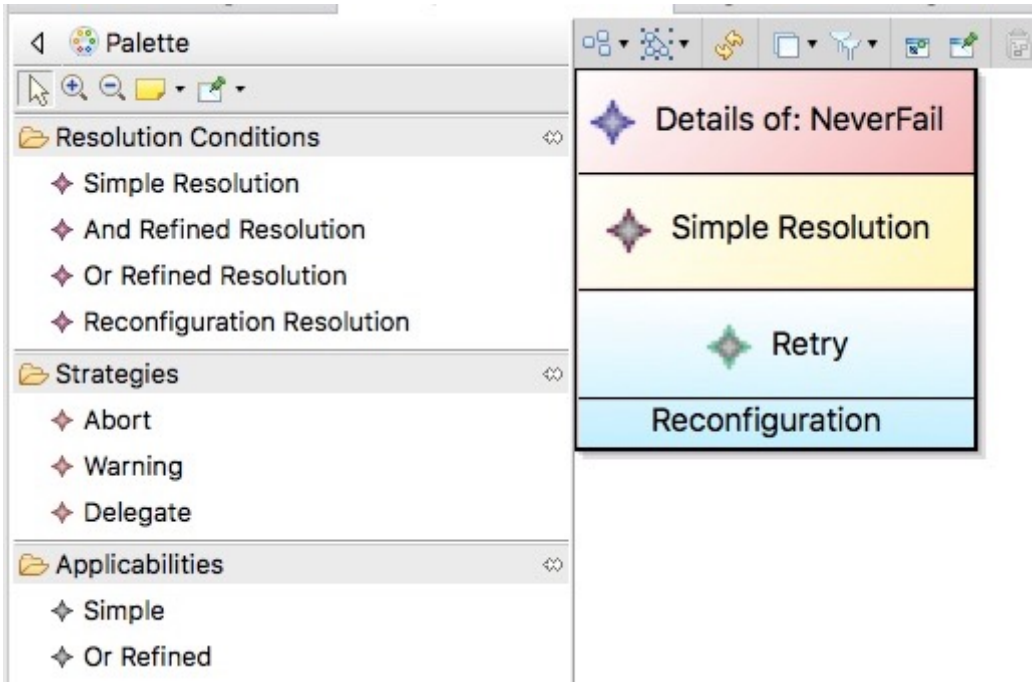


Figura 20 – Editor Unagi - Subdiagrama de Especificação de *EvoReqs*

# 5 Validação

# 6 Considerações Finais

Este capítulo apresenta as conclusões do trabalho realizado, mostrando suas contribuições. Por fim, são apresentadas suas limitações e perspectivas de trabalhos futuros.

## 6.1 Conclusões

Com a necessidade de acompanhamento dos alunos egressos do DI/Ufes, objetivando estimular alunos do ensino médio pela área da informática, viu-se a oportunidade de desenvolver um sistema web para atender esta necessidade. Além disso, como muitos estudantes do DI/Ufes desenvolvem ferramentas como parte de seu projeto final de graduação, viu-se a necessidade de integrar futuras ferramentas de forma a serem realmente utilizadas.

Os objetivos elencados no Capítulo 1 foram alcançados, de forma que toda a documentação indicada pela Engenharia de Software foi feita. Primeiramente os requisitos foram levantados e analisados, gerando os Documento de Especificação de Requisitos contendo os requisitos funcionais e não funcionais, a descrição do propósito do sistema e do minimundo, a definição dos atores, casos de uso, diagrama de estado e diagrama de classe. Após esta fase deu início ao desenvolvimento do Documento de Projeto contendo os Atributos de Qualidade e Táticas, os modelos propostos pelo FrameWeb e a arquitetura de software para o SAE.

Dentre as dificuldades encontradas para o desenvolvimento desse trabalho podemos destacar o estudo e entendimento das tecnologias Java EE tais como JAAS, CDI, JPA. Assim, observou-se a necessidade de realizar pesquisas de exemplos e tutoriais e a leitura da documentação destes frameworks. Outra dificuldades encontradas foram: assimilar os conceitos do FrameWeb tendo em vista o curto período de tempo para o desenvolvimento do projeto e escrita da monografia; Implementar o SAE como um módulo de um sistema que visa a integração de outros sistemas a serem desenvolvidos nos projetos de graduação, visto que a base desse sistema integrador (Marvin) estava ainda em construção.

Durante a fase de desenvolvimento do projeto e da implementação do mesmo, foi possível praticar e avaliar o método FrameWeb, verificando que ele auxiliou no desenvolvimento com os modelos de projeto e do perfil UML propostos pelo método por eles aproximarem o modelo de projeto arquitetural da implementação do sistema, reduzindo assim o tempo gasto com o desenvolvimento. Por outro lado, sentiu-se a falta de um forma de especificar um modelo de segurança, mostrando quais classes seriam protegidas e quais usuários teriam acesso a elas. Uma sugestão para especificar esse modelo de segurança

encontra-se na Figura, que utiliza o modelo de aplicação do FrameWeb visto que as classes desse modelo que serão protegidas, foi utilizado cores para especificar quais usuários terão acesso as classes.

Por fim, cabe destacar o grande desafio que foi integrar as diferentes disciplinas realizadas durante o curso de Ciência da Computação, pois elas foram vistas muitas vezes de forma teórica e separadamente uma da outra, mas a experiência adquirida com o desenvolvimento desse trabalho foi enorme e proveitosa, pois foi possível colocar na prática os conceitos aprendidos em sala de aula superando as dificuldades encontradas e, além disso, foi possível adquirir conhecimentos de novas tecnologias que servem para resolver os problemas que podemos encontrar no dia-a-dia.

## 6.2 Limitações e Perspectivas Futuras

No final do desenvolvimento de um software, tipicamente novas necessidades são identificadas. A manutenção e a evolução de software devem ser um trabalho constante, de forma que o ciclo de vida não finalize na homologação, mas permaneça ao longo de toda a vida do software.

A partir dos resultados alcançados, algumas limitações podem ser observadas, o que dá margem para a realização de trabalhos futuros, sendo assim alguns trabalhos surgirão a partir deste. Essas limitações são apresentadas nos itens abaixo.

- Adicionar ao método FrameWeb um modelo onde seja possível modelar os controle de segurança do sistema.
- Ampliar o escopo do sistema do DI/Ufes para todos os departamentos da Ufes, assim todos os cursos poderiam ser incluídos.

# Referências

ANDERSSON, J. et al. Modeling dimensions of self-adaptive software systems. In: *Software engineering for self-adaptive systems*. [S.l.]: Springer, 2009. p. 27–47. Citado na página 11.

ATKINSON, C.; KUHNE, T. Model-driven development: a metamodeling foundation. *IEEE software*, IEEE, v. 20, n. 5, p. 36–41, 2003. Citado na página 27.

BRUN, Y. et al. Engineering self-adaptive systems through feedback loops. In: *Software engineering for self-adaptive systems*. [S.l.]: Springer, 2009. p. 48–70. Citado 2 vezes nas páginas 11 e 17.

BUDINSKY, F. *Eclipse modeling framework: a developer's guide*. [S.l.]: Addison-Wesley Professional, 2004. Citado na página 30.

DALPIAZ, F. et al. Runtime goal models: Keynote. In: IEEE. *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*. [S.l.], 2013. p. 1–11. Citado 2 vezes nas páginas 17 e 18.

DARDENNE, A.; FICKAS, S.; LAMSWEERDE, A. van. Goal-directed concept acquisition in requirements elicitation. In: IEEE COMPUTER SOCIETY PRESS. *Proceedings of the 6th international workshop on Software specification and design*. [S.l.], 1991. p. 14–21. Citado na página 17.

DARDENNE, A.; LAMSWEERDE, A. V.; FICKAS, S. Goal-directed requirements acquisition. *Science of computer programming*, Elsevier, v. 20, n. 1-2, p. 3–50, 1993. Citado 2 vezes nas páginas 16 e 17.

FALBO, R. A. *Engenharia de Software*. [s.n.], 2014. 141 p. Disponível em: <http://www.inf.ufes.br/~falbo/files/ES/Notas_Aula_Engenharia_Software.pdf>. Citado na página 15.

FALBO, R. A. *Engenharia de Software*. [s.n.], 2017. 71 p. Disponível em: <http://www.inf.ufes.br/~falbo/files/ER/Notas_Aula_Engenharia_Requisitos.pdf>. Citado na página 15.

JURETA, I.; MYLOPOULOS, J.; FAULKNER, S. Revisiting the core ontology and problem in requirements engineering. In: IEEE. *International Requirements Engineering, 2008. RE'08. 16th IEEE*. [S.l.], 2008. p. 71–80. Citado na página 16.

KEPHART, J. O.; CHESS, D. M. The vision of autonomic computing. *Computer*, IEEE, v. 36, n. 1, p. 41–50, 2003. Citado na página 11.

KERN, H. The interchange of (meta) models between metaedit+ and eclipse emf using m3-level-based bridges. In: *8th OOPSLA Workshop on Domain-Specific Modeling at OOPSLA*. [S.l.: s.n.], 2008. v. 2008. Citado 2 vezes nas páginas 7 e 28.

LAMSWEERDE, A. V. Goal-oriented requirements engineering: A guided tour. In: IEEE. *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*. [S.l.], 2001. p. 249–262. Citado na página 16.

LAMSWEERDE, A. V.; DARIMONT, R.; LETIER, E. Managing conflicts in goal-driven requirements engineering. *IEEE transactions on Software engineering*, IEEE, v. 24, n. 11, p. 908–926, 1998. Citado na página 16.

LAPOUCHNIAN, A. Goal-oriented requirements engineering: An overview of the current research. *University of Toronto*, p. 32, 2005. Citado na página 16.

MADIOT, F.; PAGANELLI, M. Eclipse sirius demonstration. In: *P&D@ MoDELS*. [S.l.: s.n.], 2015. p. 9–11. Citado na página 31.

MORIN, B. et al. Models@ run. time to support dynamic adaptation. *Computer*, IEEE, v. 42, n. 10, 2009. Citado na página 25.

MUSSET, J. et al. Acceleo user guide. *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf*, v. 2, 2006. Citado na página 38.

PFLEEGER, S. L. *Engenharia de software: teoria e prática*. [S.l.]: Prentice Hall, 2004. Citado na página 15.

ROSS, D. T.; SCHOMAN, K. E. Structured analysis for requirements definition. *IEEE transactions on Software Engineering*, IEEE, n. 1, p. 6–15, 1977. Citado na página 15.

SELIC, B. The pragmatics of model-driven development. *IEEE software*, IEEE, v. 20, n. 5, p. 19–25, 2003. Citado na página 27.

SOUZA, V. E. S. *Requirements-based software system adaptation*. Tese (Doutorado) — University of Trento, 2012. Citado 8 vezes nas páginas 7, 11, 18, 20, 21, 23, 24 e 26.

SOUZA, V. E. S. et al. Requirements-driven software evolution. *Computer Science-Research and Development*, Springer, v. 28, n. 4, p. 311–329, 2013. Citado 4 vezes nas páginas 19, 21, 22 e 25.

SOUZA, V. E. S.; LAPOUCHNIAN, A.; MYLOPOULOS, J. (requirement) evolution requirements for adaptive systems. In: IEEE PRESS. *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. [S.l.], 2012. p. 155–164. Citado 4 vezes nas páginas 11, 17, 18 e 19.

SOUZA, V. E. S. et al. Awareness requirements. In: *Software Engineering for Self-Adaptive Systems II*. [S.l.]: Springer, 2013. p. 133–161. Citado 3 vezes nas páginas 17, 18 e 19.

STEINBERG, D. et al. *EMF: eclipse modeling framework*. [S.l.]: Pearson Education, 2008. Citado na página 28.

VIYOVIĆ, V.; MAKSIMOVIĆ, M.; PERISIĆ, B. Sirius: A rapid development of dsm graphical editor. In: IEEE. *Intelligent Engineering Systems (INES), 2014 18th International Conference on*. [S.l.], 2014. p. 233–238. Citado 2 vezes nas páginas 27 e 30.

VUJOVIĆ, V.; MAKSIMOVIĆ, M.; PERIŠIĆ, B. Comparative analysis of dsm graphical editor frameworks: Graphiti vs. sirius. In: *23nd International Electrotechnical and Computer Science Conference ERK, Portorož, B*. [S.l.: s.n.], 2014. p. 7–10. Citado 2 vezes nas páginas 27 e 30.

YU, E. et al. 1 social modeling for requirements engineering: An introduction. *Social Modeling for Requirements Engineering*, MIT Press, p. 3–10, 2011. Citado na página 17.