3.
Anexação de Requisitos ao Código

Esse capítulo descreve o primeiro grande desafio para se obter a transparência de software: anexar os requisitos ao código. Para isso, é necessário que se possa rastrear os requisitos até o código e rastrear trechos de código de volta aos requisitos. Apresentamos uma abordagem para a rastreabilidade que mantém o paradigma intencional dos requisitos ao código e utiliza heurísticas transformacionais como rastro.

Segundo o professor John Mylopoulos (Mylopoulos 2011), a transparência de software é um princípio fundamental, pois obriga os desenvolvedores a anexarem os requisitos ao código. Isso beneficiaria não só a engenharia de requisitos, mas também outras disciplinas, como teste de software.

O SIG de transparência de software (Leite e Cappelli 2010; Aló 2009) – vide Capítulo 2: Seção 2.2 – define auditabilidade como um dos requisitos não-funcionais, ou meta flexíveis, que contribuem positivamente para a transparência de software. Ainda considerando o SIG de transparência de software, a rastreabilidade é a meta flexível mais importante para a auditabilidade, uma vez que não sendo possível rastrear um artefato ou parte dele para outro artefato torna-se muito difícil auditá-lo.

Iniciamos nossos estudos sobre como anexar requisitos ao código do software através do desenvolvimento de um estudo de caso. Uma das maiores dificuldades era superar a grande diferença de nível de abstração entre os dois artefatos, requisitos e código, que muitas vezes envolvia inclusive uma mudança de paradigma. O estado da arte da Engenharia de Requisitos, ou seja, a Engenharia de Requisitos Orientada a Metas (*Goal-Oriented Requirements Engineering* - GORE) (Van Lamsweerde 2001; Mylopoulos 2008), focava na intencionalidade de atores sociais e no contexto organizacional. Entretanto, os requisitos no paradigma intencional eram utilizados para desenvolver sistemas orientados a objetos, o que causava uma ruptura de paradigma durante o desenvolvimento do software.

Durante nossos estudos de intencionalidade, com o NFR Framework (Chung et al. 2000) e o *framework* i* (Yu 1997), desenvolvemos um primeiro estudo de caso envolvendo um SMA intencional. Esse SMA intencional implementa uma aplicação ubíqua no domínio de comércio eletrônico, modelada de forma intencional usando o *framework* i*. O SMA foi implementado para o *framework* JADEX, um *add-on* para a plataforma JADE. Durante o desenvolvimento desse estudo de caso, observamos as primeiras associações entre as abstrações dos modelos do *framework* i*, as abstrações do modelo BDI e o código em XML/Java para o *framework* JADEX.

Com base nas associações observadas, construímos um primeiro conjunto de heurísticas para facilitar o desenvolvimento de SMAs intencionais a partir de modelos i*. Essas heurísticas transformacionais anexavam os requisitos da aplicação ubíqua ao código do SMA, uma vez que elas próprias podiam ser entendidas como o rastro. Um segundo estudo de caso foi desenvolvido (Serrano et al. 2008c). Esse estudo de caso focava no desenvolvimento de *frameworks* intencionais para apoiar o desenvolvimento sistemático de aplicações ubíquas no domínio odontológico. Ao longo desse estudo de caso, refinamos as heurísticas transformacionais e incluímos heurísticas mais específicas, como as que transformam as dependências entre atores em protocolos de comunicação entre agentes.

Todos os estudos de caso posteriores (Serrano et al. 2009; Serrano e Leite 2011c) utilizaram as heurísticas transformacionais desenvolvidas nos dois primeiros estudos de caso e possibilitaram pequenas evoluções em uma ou mais heurísticas. Os estudos de caso posteriores foram: (i) um *framework* para ciência de contexto em aplicações ubíquas; (ii) um sistema de agendamento de reuniões intencional; (iii) um simulador para situações de raciocínio meio-fim; (iv) um sistema de iluminação inteligente, e (v) uma versão em SMA intencional do Lattes-Scholar (Lattes-Scholar 2011a; Lattes-Scholar 2011b).

Esse capítulo apresenta heurísticas transformacionais para guiar o desenvolvimento de SMAs a partir dos requisitos até o código de SMAs intencionais. Além disso, também propomos: (i) usar modelos i* como modelos de requisitos e ainda como modelos arquiteturais; e (ii) implementar o sistema no modelo BDI usando JADEX (Pokahr et al. 2005), que também é centrado na abstração intencional. A Seção 3.1 apresenta as associações observadas entre as

abstrações do *framework* i* e as abstrações do modelo BDI e as associações observadas entre as abstrações do modelo BDI e o código do SMA intencional para o *framework* JADEX. As heurísticas transformacionais de desenho e de implementação são apresentadas na Seção 3.2. O uso das heurísticas transformacionais como elos de rastreabilidade, ou rastros, é exposto na Seção 3.3. Os trabalhos relacionados são discutidos na Seção 3.4. Finalmente, a Seção 3.5 apresenta as considerações finais do capítulo.

3.1. Associações entre Abstrações dos Modelos i* e BDI e Código JADEX

Os requisitos descritos em modelos intencionais são desenhados em agentes inteligentes que simulam o raciocínio humano através do modelo BDI. Optou-se pelo modelo BDI como a base para a arquitetura mental dos agentes, pois o modelo i* utiliza o mesmo paradigma de orientação, ou seja, a orientação a metas. Assim, evitam-se transições abruptas de nível de abstração entre os modelos de requisitos e de desenho. O objetivo desse processo é baixar o nível de abstração das especificações do software em desenvolvimento, obtendo-se o desenho do software.

Existem diferenças significativas entre um modelo i* e uma especificação BDI. As duas principais diferenças são: (i) os modelos i* representam uma rede de atores sociais, enquanto o modelo BDI visa representar a arquitetura mental interna de um único agente, e (ii) o modelo i* representa metas flexíveis como tarefas, e metas e metas flexíveis contribuem para as metas flexíveis de forma positiva ou negativa através de elos de contribuição.

Embora existam diferenças significativas entre a semântica dos dois modelos, os modelos compartilham muitas semelhanças, como os conceitos de ator/agente, metas/desejos, tarefas/intenções para atingir metas, crenças/crenças, recursos/crenças, entre outros. Assim, foi possível relacionar as abstrações dos dois modelos, como apresentado na Figura 3.1.

A abstração de ator do *framework* i* é representada no modelo BDI como um agente inteligente, pois um ator do *framework* i* possui as atitudes mentais informativas (visão do mundo e de seu estado interno), motivacionais (metas a serem atingidas) e deliberativas (ações para alterar o ambiente e atingir as metas).

As abstrações de agente, posição e papel do *framework* i* são representadas no modelo BDI também como agentes, pois todas essas abstrações são especializações da abstração de ator e, portanto, também possuem as mesmas atitudes mentais.

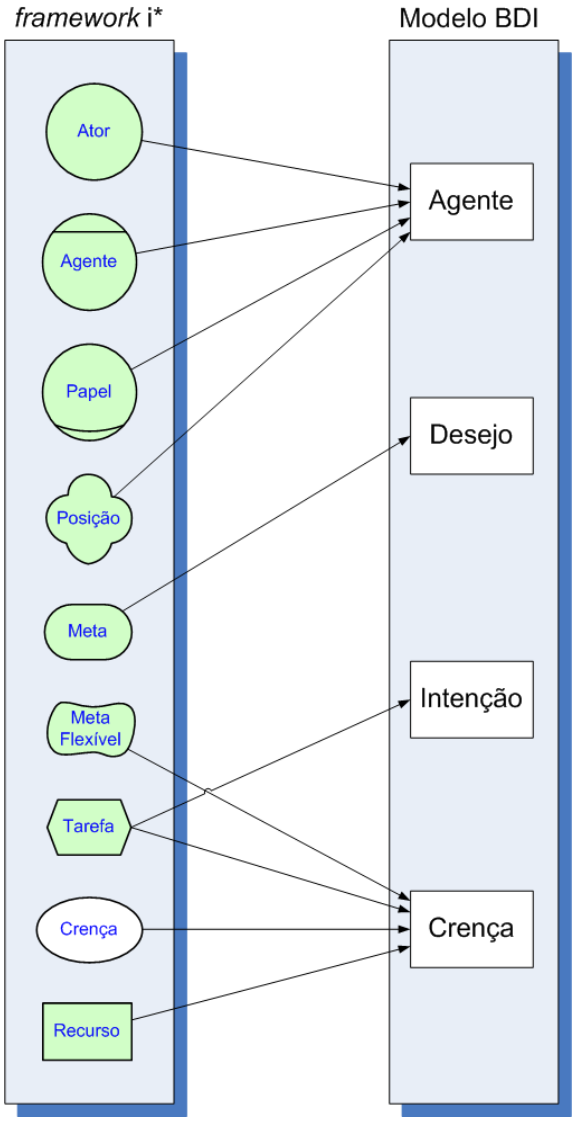

Figura 3.1 - Relações entre as abstrações do *framework* i* e do modelo BDI

A abstração de meta do *framework* i* está diretamente relacionada à abstração de desejo do modelo BDI, pois ambas representam estados desejados para o ambiente que envolve o ator/agente. A abstração de meta flexível do *framework* i* não possui um relacionamento direto com alguma abstração do modelo BDI. Porém, podemos relacioná-la com a abstração de crença, já que uma meta flexível representa como um ator enxerga um critério de qualidade do mundo real.

A abstração de tarefa do *framework* i* está diretamente relacionada à abstração de intenção do modelo BDI, uma vez que ambas descrevem uma ação ou sequencia de ações para tentar atingir uma meta. Entretanto os impactos que uma tarefa exerce sobre metas flexíveis não fazem parte da abstração de intenção do modelo BDI e devem ser representados como crenças do agente.

Originalmente, o *framework* i* não incluía a abstração de crença, o que já foi corrigido. A abstração de crença do *framework* i* é a mesma abstração de crença utilizada no modelo BDI. Recursos, entretanto, exigem uma maior atenção. Recursos que representam informações podem ser representados integralmente como uma crença no modelo BDI. Já os recursos que representam objetos do mundo real precisam antes ser abstraídos, selecionando apenas as características relevantes ao agente.

A Figura 3.2 mostra as relações entre as abstrações da especificação BDI e do código de SMAs em JADEX. Agentes executáveis são implementados como agentes JADEX. Agentes não executáveis (originados a partir de papéis e posições) tornam-se capacidades que devem ser assimiladas por agentes executáveis. Desejos são traduzidos como metas atingíveis, mantidas ou realizadas de acordo com a *tag* "type" da especificação BDI. As intenções são traduzidas como planos do agente – classes Java que estendem a classe "Plan" do JADEX. As crenças com cardinalidade 0..1 ou 1..1 são traduzidas como crenças do agente, enquanto as crenças com cardinalidade 0..n ou 1..n são traduzidas como um conjunto de crenças.

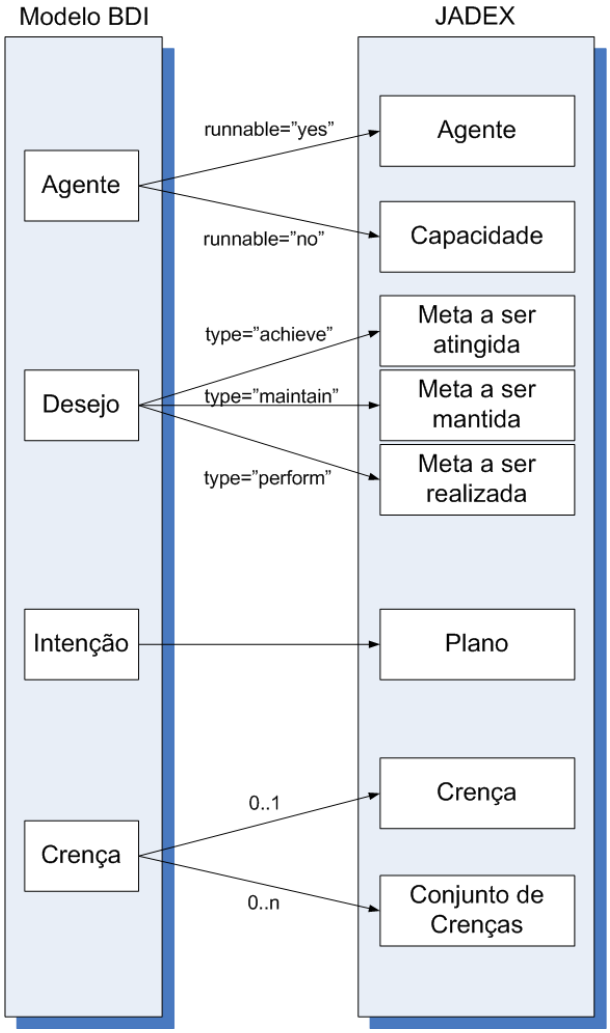

Figura 3.2 - Relações entre as abstrações do modelo BDI e código JADEX

3.2. Heurísticas Transformacionais de Desenho e de Implementação

Com base nos relacionamentos entre as abstrações dos modelos i* e BDI, apresentados na Figura 3.1, foi possível produzir heurísticas que facilitam a obtenção da especificação BDI para o software (Serrano e Leite 2011c; Serrano et al. 2009). Essa especificação é um documento XML que possui uma estrutura definida através de um documento XML Schema. A Tabela 3.1 mostra as principais heurísticas transformacionais de desenho que produzem a especificação BDI a partir dos modelos Dependência Estratégica e *Rationale* Estratégico do *framework* i*.

Tabela 3.1 - Heurísticas transformacionais de desenho: do *framework* i* para a especificação BDI

#	Quando aplicar	Ação	Saída (trechos em XML)
01	Para cada ator (ou agente, papel ou posição) pertinente à aplicação.	Criar um agente na especificação BDI. Atores e agentes possuem a *tag* runnable="yes". Papéis e posições possuem a *tag* runnable="no".	<agent name="[actor_name]" runnable="[yes|no]"> <beliefs/> <desires/> <intentions/> </agent>
02	Para cada meta de um ator.	Criar um desejo na especificação do agente.	<desire name="[goal_name]" type=""/>
03	Para cada meta flexível de um ator.	Criar uma crença com o tipo "Softgoal".	<belief name="[softgoal_name]" type="Softgoal" />
04	Para cada tarefa que visa atingir uma meta de um ator.	Criar uma intenção e associá-la ao desejo correspondente à meta.	<intention name="[task_name]"> <desire>[goal_name] </desire> <script lang="" /> </intention>
05	Para cada tarefa que visa atingir uma meta de um ator.	Criar uma crença com o tipo "Task".	<belief name="[task_name]" type="Task" />
06	Para cada tarefa que visa atingir uma meta de um ator.	Criar um cenário para descrever a tarefa e incluí-lo na *tag* "script" da intenção.	<script lang="scenarios"> [scenario] </script>
07	Para cada tarefa decomposta em subtarefas.	Criar um cenário que utiliza subcenários para descrever a tarefa e incluí-los na *tag* "script" da intenção.	<script lang="scenarios"> [scenario] [sub-scenario1] ... </script>
08	Para cada crença do ator com cardinalidade 0..1 ou 1..1.	Criar uma crença na especificação do agente.	<belief name="[belief_name]" type="[belief_type]" />
09	Para cada crença do ator com cardinalidade 0..n ou 1..n.	Criar um conjunto de crenças na especificação do agente.	<beliefset name="[beliefset_name]" type="[beliefset_type]" />
10	Para cada recurso que representa uma informação.	Criar uma crença na especificação do agente.	<belief name="[belief_name]" type="[belief_type]" />

#	Quando aplicar	Ação	Saída (trechos em XML)
11	Para cada recurso que representa um conjunto de informações	Criar um conjunto de crenças na especificação do agente.	<beliefset name="[beliefset_name]" type="[beliefset_type]" />
12	Para cada recurso que representa um recurso físico.	Abstrair do recurso físico as informações relevantes para o agente e criar uma crença.	<belief name="[belief_name]" type="[belief_type]" />
13	Para cada dependência por meta entre dois agentes.	Criar um desejo nas especificações dos dois agentes envolvidos e iniciar um protocolo de requisição (episódio de um cenário).	Heurística 02 <script lang="scenarios"> [scenario] [request episode] </script>
14	Para cada dependência por meta flexível entre dois agentes.	Acessar o grau de satisfação da meta flexível. Não é necessário estabelecer uma comunicação entre os agentes, pois impactos são sempre divulgados para todos os agentes.	<script lang="scenarios"> [scenario] [access episode] </script>
15	Para cada dependência por tarefa entre dois agentes.	Criar a tarefa na especificação de ambos os agentes e iniciar um protocolo de requisição.	Heurística 04 <script lang="scenarios"> [scenario] [request episode] </script>
16	Para cada dependência por recurso entre dois agentes.	Criar a crença em ambos os agentes e iniciar um protocolo de requisição.	Heurística 09, 10 ou 11 <script lang="scenarios"> [scenario] [request episode] </script>
17	Para cada dependência por meta entre vários agentes.	Criar a meta na especificação de todos os agentes envolvidos. Requisitar a todos os agentes ou responder a todos.	Heurística 02 <script lang="scenarios"> [scenario] [request_all episode] [inform_all episode] </script>

#	Quando aplicar	Ação	Saída (trechos em XML)
18	Para cada dependência por meta flexível entre vários agentes.	Acessar o grau de satisfação da meta flexível. Não é necessário estabelecer uma comunicação entre os agentes, pois impactos são sempre divulgados para todos os agentes.	<script Lang="scenarios"> [scenario] [access episode] </script>
19	Para cada dependência por tarefa entre vários agentes.	Criar a tarefa na especificação de todos os agentes envolvidos. Requisitar a todos ou responder a todos os agentes.	Idem à heurística 05 <script lang="scenarios"> [scenario] [request_all episode] [inform_all episode] </script>
20	Para cada dependência por recurso entre vários agentes.	Criar a crença na especificação de todos os agentes envolvidos. Requisitar a todos ou responder a todos os agentes.	Heurística 09, 10 ou 11 <script lang="scenarios"> [scenario] [request_all episode] [inform_all episode] </script>

A Figura 3.3 mostra o modelo arquitetural parcial no *framework* i* do estudo de caso Lattes-Scholar (mais detalhes no Capítulo 7). Para ilustrar o processo de aplicação das heurísticas, foi incluída uma numeração ao lado das abstrações do modelo. Cada número indica a heurística transformacional de desenho a ser aplicada para se obter a especificação do software no modelo BDI.

Os agentes "Página Web do Lattes-Scholar", "Google" e "Lattes" são agentes externos ao software e, portanto, não são desenhados ou implementados. O agente "Gerente do SMA do Lattes-Scholar" é um agente interno desenhado como um agente executável do SMA. Os papéis "Pesquisador de Currículos" e "Repositório de Currículos" são desenhados como agentes não executáveis e serão, posteriormente, implementados como capacidades a serem utilizadas por agentes executáveis. As metas "URLs dos Currículos dos Pesquisadores sejam Recuperadas" e "Fotos dos Pesquisadores sejam Recuperadas" são desenhadas

como desejos de ambos os agentes envolvidos em cada uma das dependências. As metas flexíveis “Tempo de Resposta Adequado [Busca pelo Pesquisador]” e “Disponibilidade [Lattes]” são desenhadas como crenças do tipo “Softgoal” na especificação BDI dos agentes. Os recursos “Nome do Pesquisador”, “URL do Currículo Escolhido”, “URL do Currículo do Pesquisador” e “Currículo do Pesquisador” representam informações e são desenhados como crenças dos agentes. Os recursos “Lista de Pesquisadores” e “URLs dos Currículos dos Pesquisadores” representam conjuntos de informações e são desenhados como um conjunto de crenças no modelo BDI. O recurso “Fotos dos Pesquisadores” representa um recurso físico que deve ser abstraído do mundo real. No caso, as informações relevantes não são as fotos em si, e sim a URL de cada uma das fotos.

As dependências por meta ou recurso são desenhadas como protocolos de requisição para que o agente *dependee* (Yu 1995) faça o que deve ser feito. Os protocolos de requisição são descritos em episódios de cenários de intenções. Esses protocolos de requisição serão implementados como trocas de mensagens entre os agentes através do uso da capacidade “Protocols” do JADEX. As dependências por metas flexíveis não envolvem protocolos de interação, uma vez que cada agente do SMA pode verificar por si mesmo o grau de satisfação das metas flexíveis. Os impactos nas metas flexíveis, causados pelas ações do agente *dependee*, são comunicados a todos os agentes por *broadcast* logo que ocorrem.

A especificação BDI produzida, que é um documento XML, possui diversas lacunas que devem ser preenchidas manualmente pelo engenheiro de software.

O primeiro tipo de lacuna a ser preenchida é o tipo das crenças do agente. Esse tipo pode ser uma classe de objetos ou texto, número, *booleano*, data, alfanumérico, entre outros tipos atômicos. Caso seja necessário, o engenheiro de software pode construir diagramas de classe para detalhar os atributos, os métodos e as associações entre as classes. Evita-se o uso de tipos específicos de uma linguagem, uma vez que a especificação BDI não pressupõe uma linguagem de implementação.

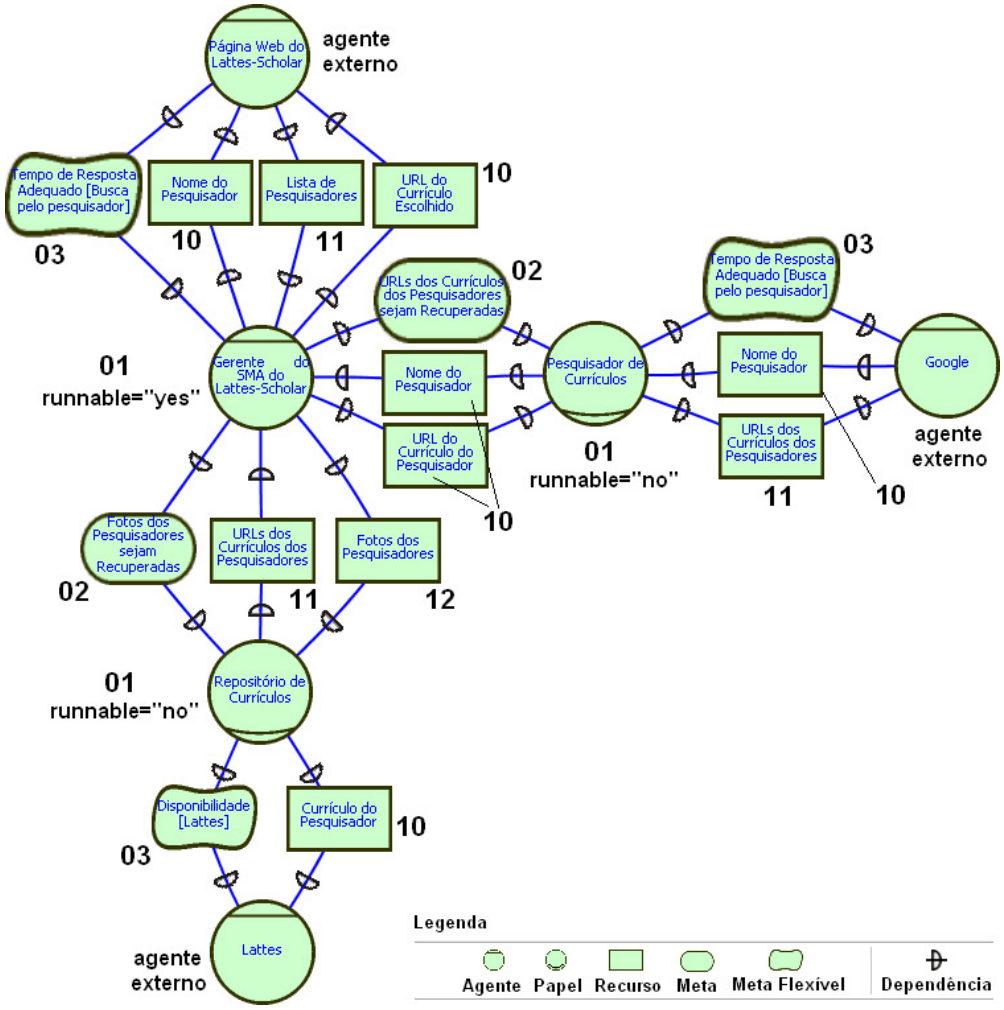

Figura 3.3 - Aplicação das heurísticas de desenho no modelo arquitetural parcial do Lattes-Scholar

O segundo tipo de lacuna a ser preenchido é o tipo de cada desejo dos agentes. Existem três tipos muito bem definidos de desejos: atingir, manter e realizar. Um desejo do tipo atingir é um desejo que representa um estado do ambiente que se deseja atingir. Uma vez atingido esse estado, o desejo deixa de estar ativo. Um desejo do tipo manter representa um estado do ambiente que deve ser mantido. O agente deve corrigir o ambiente toda vez que um evento faça o estado do ambiente se alterar. Um desejo do tipo realizar coloca o agente em um modo semelhante a um servidor, aguardando eventos e realizando ações quando solicitadas.

O terceiro tipo de lacuna a ser preenchido é o *script* das intenções. Esse *script* é uma lista de ações que explicam de uma forma mais detalhada as intenções dos agentes. As ações são descritas em uma linguagem especificada no

atributo “lang”. Assim, obtém-se um sequenciamento temporal de ações que não pode ser obtido a partir dos modelos do *framework* i*, atemporais por definição. As interações entre os agentes também são detalhadas nesses *scripts*, explicitando as trocas de mensagens, o conteúdo dessas mensagens e os protocolos de comunicação utilizados.

Optamos por utilizar a linguagem de cenários (Leite et al. 1997) para descrever as intenções dos agentes, por se tratar de uma linguagem semi-estruturada baseada em linguagem natural. Cada ação é um episódio do cenário. As tarefas decompostas em sub-tarefas dos modelos do *framework* i* são descritas por cenários que referenciam sub-cenários. Os cenários são inseridos na *tag* “script” das intenções e o atributo “lang” é definido como “Scenario”.

Com base nos relacionamentos entre as abstrações do modelo BDI e do código JADEX, expostos na Figura 3.2, foi possível produzir heurísticas que facilitam a obtenção de *templates* de código JADEX, tanto dos ADF em XML quanto dos planos especificados em classes Java. A Tabela 3.2 mostra algumas dessas heurísticas.

Tabela 3.2 - Heurísticas transformacionais de implementação: da especificação BDI para código JADEX

#	Quando aplicar	Ação	Saída (XML ou Java)
01	Para cada agente executável.	Criar o ADF em branco de um agente com o nome “[nome_agente].agent.xml” em um pacote Java com nome [nome_agente].	<agent name="[agent_name]" package="[agent_name]"> <imports/> <capabilities/> <beliefs/> <goals/> <plans/> <events/> <expressions/> <properties/> <configurations/> </agent>

#	Quando aplicar	Ação	Saída (XML ou Java)
02	Para cada agente não executável.	Criar o ADF em branco de uma capacidade com o nome "[nome_agente] .capability.xml" em um pacote Java com nome [nome_agente].	<capability name="[agent_name]" package="[agent_name]"> <imports/> <capabilities/> <beliefs/> <goals/> <plans/> <events/> <expressions/> <properties/> <configurations/> </capability>
03	Para cada desejo do tipo atingir de um agente.	Criar uma meta do tipo atingir no ADF do agente ou capacidade.	<achievegoal name="desire_name"/>
04	Para cada desejo do tipo manter de um agente.	Criar uma meta do tipo manter no ADF do agente ou capacidade.	<maintaingoal name="desire_name"/>
05	Para cada desejo do tipo realizar de um agente.	Criar uma meta do tipo realizar no ADF do agente ou capacidade.	<performgoal name="desire_name"/>
06	Para cada crença com tipo "Softgoal" de um agente.	Criar uma crença com o tipo "Softgoal" no conjunto de crenças "softgoals" do ADF.	<belief name="[softgoal_name]" type="Softgoal" />
07	Para cada intenção de um agente.	Criar um plano no ADF.	<plan name="[intention_name]"> <bodyclass= "[intention_name]"/> <trigger> <goalref="[desire_name]"/> </trigger> </plan>

#	Quando aplicar	Ação	Saída (XML ou Java)

08	Para cada intenção de um agente.	Criar uma classe Java com o nome [nome_da_intenção] que estende a classe “Plan” do JADEX no mesmo pacote do ADF.	//imports
			public class [intention_name]
			extends Plan {
			public [intention_name]() {
			}
			public void body() {
			}
			}

09	Para cada intenção de um agente.	Copiar os cenários do *script* da intenção como comentários para os métodos do plano do agente. Subcenários são criados como métodos.	public void [sub_scenario] {
			//[sub_scenario]
			}
			public void body() {
			//[scenario]
			this.[sub_scenario]();
			}

10	Para cada agente que possui protocolos de interação em seus episódios dos cenários das intenções	Incluir a capacidade “Protocols” do *framework* JADEX	<capabilities> <capability name="procap" file="jadex.planlib.Protocols"/> </capabilities>
11	Para cada sincronização entre intenções nos episódios dos cenários das intenções	Criar um evento interno de sincronização no ADF do agente	<events> <internalevent name=[event_name]/> </events>
12	Para cada crença com tipo “Softgoal” de um agente.	Criar uma crença com o tipo “Softgoal” no conjunto de crenças “softgoals” do ADF.	<belief name="[softgoal_name]" type="Softgoal" />
13	Para cada crença com tipo “Task” de um agente.	Criar uma crença com o tipo “Task” no conjunto de crenças “tasks” do ADF.	<belief name="[task_name]" type="Task" />
14	Para cada crença com cardinalidade 0..1 ou 1..1 de um agente.	Criar uma crença no ADF com tipo convertido para tipos Java ou classe Java.	<belief name="[belief_name]" type="[belief_type]" />

#	Quando aplicar	Ação	Saída (XML ou Java)
15	Para cada desejo do agente	Criar um meta-*goal* no ADF	<me*tag*oal name=”me*tag*oal[n]”> [parametersets] <trigger> <goal ref=”[desire_name]”/>
16	Para cada desejo do agente	Criar um meta-plano no ADF	<plan name=”metaplan[n]”> <body class=”Metaplan”/> <trigger> <goal ref=”me*tag*oal[n]”/>

Após a aplicação das heurísticas, existem várias lacunas no código dos agentes que precisam ser preenchidas, manualmente, pelo engenheiro de software. Algumas dessas lacunas são preenchidas a partir dos cenários produzidos. A Figura 3.4 mostra alguns trechos de código que podem ser obtidos a partir dos cenários, como os comentários para os planos em Java.

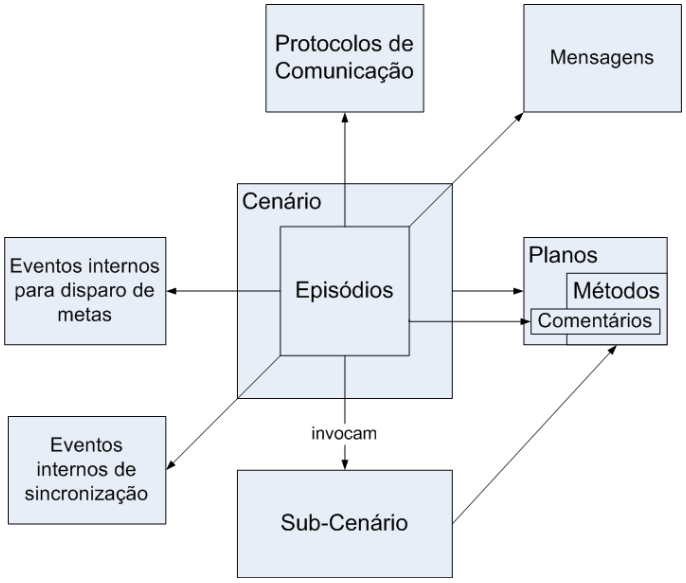

Figura 3.4 - Trechos de código que podem ser obtidos a partir dos cenários

Após as mudanças de nível de abstração desde os requisitos especificados em modelos intencionais do *framework* i* até o código do sistema multi-agentes em JADEX, observamos que os atores intencionais do i* foram desenhados como agentes inteligentes reativos/deliberativos. Esses agentes reagem a mudanças no meio ambiente que os envolvem e também visam atingir objetivos agindo sobre o meio. Porém, nota-se que esses agentes não estão prontos para raciocinar sobre critérios de qualidade e os impactos que suas ações exercem sobre eles. Esses critérios de qualidade, ou metas flexíveis, e os impactos das ações sobre eles já

haviam sido elicitados, modelados e analisados no modelo i*. Portanto, houve uma perda de capacidade de raciocínio do ator/agente ao baixarmos o nível de abstração até o código. Para evitar essa perda de capacidade de raciocínio, o próximo capítulo, o Capítulo 4, descreve uma máquina de raciocínio qualitativa para agentes intencionais avaliarem metas flexíveis em tempo de execução.

3.3. Geração dos Rastros a partir das Heurísticas Transformacionais

Segundo o SIG de Transparência (Leite e Cappelli 2010), a rastreabilidade entre os artefatos ajuda a transparência de software, pois contribui positivamente para a auditabilidade. Os rastros entre os artefatos são fundamentais para afirmar que foi possível anexar os requisitos ao código.

Usualmente, matrizes de rastreabilidade são utilizadas para descrever os elos entre os artefatos produzidos durante o desenvolvimento do software. Entretanto, os elos de rastreabilidade, normalmente, são anotados na matriz de rastreabilidade após a criação dos artefatos. Ainda é necessário um considerável esforço para se manter essa matriz, atualizando-a sempre que um artefato é criado, destruído ou modificado.

Na nossa abordagem para a anexação de requisitos ao código, todos os demais artefatos são criados por heurísticas transformacionais aplicadas nos requisitos do software ou na especificação BDI. Assim, é possível utilizar as heurísticas aplicadas como os rastros.

Para os nossos propósitos, definimos um rastro, ou elo de rastreabilidade, como um par **origem-destino**, no caso de rastreabilidade do tipo *forward* (sentido requisitos-código). Esse mesmo par pode ser invertido para um par **destino-origem**, no caso de rastreabilidade do tipo *backward* (sentido código- requisitos).

A origem de um determinado par pode ser obtida a partir da condição da heurística transformacional aplicada. O destino de um determinado par pode ser obtido a partir da ação da heurística. Se a ação da heurística cria ou modifica dois ou mais artefatos, é possível também inferir um par origem-destino entre esses artefatos.

A heurística transformacional de desenho 06 (vide Tabela 3.2), por exemplo, cria um cenário e o insere como o conteúdo de uma *tag* “script” de uma

intenção na especificação BDI para cada tarefa que visa atingir uma meta do agente. Dessa forma, a partir dessa heurística transformacional de desenho podemos inferir três pares origem-destino, que serão os rastros entre os artefatos: (i) um rastro do tipo tarefa-cenário, onde uma tarefa de um ator no modelo i* é rastreada para um cenário no desenho do software. O arquivo do cenário possui o nome "[nome do modelo]\[nome do ator]\[nome da tarefa]-[versão].cen"; (ii) um rastro do tipo tarefa-*tag* "script", onde a tarefa é rastreada para o conteúdo de uma *tag* na especificação BDI. Essa *tag* pode ser encontrada, na especificação BDI da mesma versão, na *tag* "script" de uma *tag* "intention" de nome [nome da tarefa], dentro de uma *tag* "agent" de nome [nome do ator]; e (iii) um rastro do tipo cenário-*tag* "script", onde o cenário de nome "[nome do modelo]\[nome do ator]\[nome da tarefa]-[versão].cen" é rastreado para a mesma *tag* descrita em (ii). O fato de que toda tarefa pode ser rastreada para um cenário, e vice-versa, facilita a compreensão dos elos de rastreabilidade.

O mesmo processo de geração de rastros pode ser aplicado a todas as heurísticas transformacionais de desenho e de implementação. Assim, não existe a necessidade de se manter uma matriz de rastreabilidade, embora seja necessária uma maior disciplina do engenheiro de software ao nomear os artefatos.

3.4. Trabalhos Relacionados

Para guiar o desenvolvimento de sistemas multi-agentes, Tropos (Castro et al. 2002; Giunchiglia et al. 2003; Bresciani et al. 2004; Bertolini et al. 2006) oferece uma metodologia orientada a agentes. Esse método está centrado no *framework* i* (Yu 1997), cujas abstrações (como as metas, metas flexíveis, tarefas, recursos, crenças e outras) são usadas para modelar os requisitos e os detalhes de projeto do SMA em desenvolvimento. Tropos sugere cinco fases, desde *early requirements* até a implementação, denominadas: *Early Requirements*, *Late Requirements*, Desenho Arquitetural, Desenho Detalhado e Implementação.

Existem diferentes abordagens (Penserini et al. 2007; Silva et al. 2011) centradas na metodologia Tropos que propõem soluções para: (i) desenvolver SMAs aplicando heurísticas para mapear modelos i* para código de SMAs no

modelo BDI; e (ii) desenvolver SMAs a partir de arquiteturas organizacionais em i* para arquiteturas baseadas em agentes aplicando diagramas da Agent UML (Bauer et al. 2001) para capturar a intencionalidade dos agentes.

Em (Perini e Susi 2006), os autores sugerem algumas heurísticas que não cobrem todas as abstrações do *framework* i*. Por exemplo, eles não tratam as abstrações papel e posição. Eles também não discutem como traduzir dependências entre atores para protocolos de interações entre agentes. Além disso, eles propõem uma máquina de raciocínio simples que considera apenas as prioridades das metas flexíveis. Em nossa proposta (Serrano e Leite 2011b), oferecemos heurísticas para todas as abstrações do *framework* i*. A máquina de raciocínio dos agentes é enriquecida com um conjunto de regras nebulosas – vide Capítulo 4 – para aprimorar a capacidade cognitiva dos agentes ao lidar com as incertezas intrínsecas envolvidas nas metas flexíveis.

Em (Silva et al. 2011), os autores utilizam uma extensão da UML (UML 2011) para modelar os agentes e a intencionalidade. Afirmamos que modelos i* capturam essas informações e que não existe a necessidade de se introduzir outras notações ou diagramas. Obviamente, vale ressaltar que alguns diagramas da UML são apropriados para lidar com situações ou aspectos que o *framework* i* não trata, como, por exemplo, a temporalidade.

3.5. Considerações Finais

Nesse capítulo, apresentamos uma visão geral do nosso trabalho para anexar os requisitos ao código. Para isso, centramos o desenvolvimento do SMA nos conceitos de intencionalidade e de rastreabilidade. O apoio tecnológico proposto oferece heurísticas transformacionais de desenho e de implementação para conduzir o desenvolvimento desde modelos i* até o código de SMAs intencionais em JADEX. Essas heurísticas transformacionais formam os rastros entre os artefatos, provendo uma rastreabilidade *forward* (dos requisitos para o código) e *backward* (do código para os requisitos).