JiStar

Rastreabilidade Entre Código Java e Modelos de Metas i*

Ana Maria da Mota Moura[1] e Julio Cesar Sampaio do Prado Leite[1]

[1] Departamento de Informática da PUC Rio, Rio de Janeiro, Brasil
[ammmoura,julio]@inf.puc-rio.br

Resumo. Nos últimos anos, a engenharia de requisitos orientada a metas tem sido extensivamente pesquisada e os resultados indicam diversas vantagens sobre outras abordagens para a modelagem de requisitos. Uma vez que o software foi construído sob outras perspectivas e paradigmas, como compreender suas metas para fins de evolução? Na literatura, existem heurísticas de rastreabilidade que podem auxiliar no resgate das metas do software quando aplicadas no sentido código-requisitos. Entretanto, a rastreabilidade para trás, aplicada a sistemas construídos sob outros paradigmas pode gerar modelos extensos e com um nível de abstração muito baixo. Esta pesquisa visa definir um modelo de rastreabilidade para trás entre o código da aplicação e o modelo de metas através do uso de metadados de código. Em particular entre sistemas desenvolvidos em Java e modelos i*. Embora existam pesquisas sobre o mapeamento de modelos i* para representações OO, a rastreabilidade para trás é menos frequente na literatura. Desenvolvemos o framework JiStar que é composto por: (A) um conjunto de anotações de código Java para elementos do framework i* com vistas a viabilizar uma rastreabilidade entre os artefatos de código Java e modelos i*, e; (B) um exportador de modelos i* com vistas a permitir a geração automática de modelos de metas nos formatos HTML e PiStar. Avaliamos a viabilidade da nossa abordagem usando o código fonte de um sistema real chamado RioBus visando reconciliá-lo com suas metas. Nossos resultados indicam a adequação dessa rastreabilidade para fins de evolução do software.

Palavras Chave: JiStar, Rastreabilidade, iStar, Java.

1 Introdução

A Engenharia de Requisitos Orientada a Metas (GORE - *Goal Oriented Requirements Engineering*) tem sido utilizada nas iterações iniciais do desenvolvimento de software [10, 23, 24]. Entretanto, algumas pesquisas têm demonstrado a utilidade de modelos de metas durante o projeto de arquitetura, codificação, teste, monitoramento, adaptação e evolução [4, 12]. GORE auxilia o engenheiro de requisitos a lidar com os interessados para descobrir suposições e fundamentos ocultos, assim como, desenvolver e explorar potenciais alternativas [23] pois as metas capturam, em diferentes níveis de abstração, os vários objetivos que o software deve alcançar [24]. Algumas pesquisas têm feito o uso de GORE para além do desenvolvimento baseado em agentes, como integração

com métodos ágeis [13] e outros tipos de desenvolvimento [3]. Deste modo, há indícios de que softwares construídos com o uso de GORE podem estar mais alinhados com as metas da organização e serem mais amenos para evolução. Neste contexto, como evoluir softwares construídos sob outras perspectivas e/ou paradigmas mantendo-os alinhados com suas metas?

Quando o software não foi desenvolvido com GORE, boa parte do conhecimento sobre as metas (i.e.: razões da existência do software) fica disperso em e-mails, atas de reunião e sob a forma de conhecimento tácito dos envolvidos no projeto. Neste contexto, após algum tempo, não é trivial identificar se todas as metas já estão implementadas, ou reutilizar algumas operacionalizações de metas flexíveis, ou identificar o propósito de algumas partes do código com relação aos objetivos da organização, entre outros problemas de engenharia de software.

Este trabalho é parte de uma pesquisa para a reengenharia de sistemas autoadaptativos guiada pelo requisito não funcional de consciência [15], onde propomos um conjunto de heurísticas para extrair um modelo de metas a partir de sistemas orientados a objetos (OO) a fim de evoluir tais sistemas com base em suas metas flexíveis de consciência ainda que não tenham sido construídos sob a perspectiva GORE. Alguns trabalhos na literatura sugerem abordagens para o mapeamento do modelo de metas até a implementação do sistema [1, 4, 20], ou seja, visam garantir que o software construído corresponda ao desejado [21], gerando vínculos os quais chamamos de rastreabilidade [2, 19, 21]. A rastreabilidade pode ser avante (dos requisitos para o código) e/ou para trás (do código para os requisitos). Esta última poderia ser utilizada para gerar um modelo intencional a partir do código, porém softwares desenvolvidos com outros paradigmas não orientados a agentes (ex.: Programação Orientado a Objetos – POO) podem gerar modelos de meta extensos e pouco abstratos, conforme observamos ao aplicarmos as nossas heurísticas em estudos de caso com sistemas reais para diferentes finalidades de reengenharia [16, 17]. Apesar das nossas heurísticas possibilitarem uma forma semiautomatizada de gerar modelos de metas i*, nestes estudos, nos deparamos com modelos de metas extensos que foram utilizados com alguma dificuldade pelos pesquisadores que não eram familiarizados com o código, devido à nomenclatura e quantidade dos elementos que vinham diretamente do código fonte.

Diante deste contexto, a principal questão de pesquisa neste trabalho é: Como obter um modelo de metas a partir de um código que foi construído sem a perspectiva GORE, com um nível de abstração mais próximo dos objetivos da aplicação, sem perder a rastreabilidade com o código?

Delimitamos o escopo da nossa pesquisa ao uso do framework i* [24], para representar o modelo de metas, e ao paradigma OO, pois o nosso conjunto de heurísticas para a rastreabilidade para trás [15] considera características específicas de programação orientada a objetos (POO). O i* foi escolhido por ser um framework bem difundido na literatura [9] e com inúmeras extensões i* (vide catálogo [11]).

Em resposta à nossa questão de pesquisa, definimos um modelo de rastreabilidade para trás, do código fonte para o modelo de metas da aplicação, através do uso de metadados de código. Esta rastreabilidade pode prover a geração automática de modelos de metas a partir do código. Em particular, entre modelos em i* e sistemas desenvolvidos em Java. A linguagem Java foi escolhida por sua ampla utilização na indústria e na

literatura. A rastreabilidade é proposta através de um framework denominado JiStar[1] composto por um conjunto de anotações de código Java, para representar as informações dos modelos i*, e um gerador de modelo de metas em HTML e PiStar 2.0 [18].

Avaliamos a viabilidade da nossa abordagem usando o código fonte de um aplicativo chamado RioBus visando reconciliá-lo com as metas as quais seus usuários conseguem alcançar em sua versão atual.

A partir desta introdução, o restante do artigo está organizado da seguinte forma: Na Seção 2, apresentamos a fundamentação teórica. Na Seção 3, relacionamos alguns trabalhos correlatos. Na Seção 4, apresentamos o conjunto de anotações presente no framework JiStar, enquanto na Seção 5 mostramos a exportação de modelos de metas para HTML e PiStar 2.0 utilizando também o JiStar. Na Seção 6, apresentamos nosso estudo de viabilidade com base no aplicativo móvel real denominado RioBus. Finalmente, na Seção 7, apresentamos nossas conclusões ressaltando as contribuições, desafios e trabalhos futuros.

2 Fundamentação Teórica

2.1 Framework i*

A versão do framework i* escolhida para esta pesquisa é a 1.0 [10], porque a sua intencionalidade distribuída auxilia a modularização, e a relação meios-fim é o ponto chave para mapear a variabilidade. Não utilizamos a versão 2.0 pois, entre as mudanças realizadas, estão a junção dos relacionamentos meios-fim (*means-end*) e decomposição-tarefa (*task-decomposition*) em um único relacionamento denominado refinamento (*refinement*) e a junção dos relacionamentos entre atores e-parte-de (*is-part-of*), desempenha (*plays*), ocupa (*occcupies*), e cobre (*covers*) em um relacionamento denominado participa-em (*participates-in*). A riqueza presente nos tipos de relacionamentos da versão 1.0 nos ajuda a qualificar melhor as relações existentes entre os elementos.

O framework i* define meta como uma condição ou estado de algo no mundo que o ator gostaria de alcançar [24]. O framework i* consiste de dois modelos: O modelo de Dependência Estratégica (SD - *Strategic Dependency*) que descreve o processo em termos de relacionamentos de dependência intencional entre agentes que dependem um do outro para que metas sejam alcançadas, tarefas sejam executadas, recursos sejam fornecidos ou metas-flexíveis sejam satisfeitas a contento; e o modelo de Raciocínio Estratégico (SR - *Strategic Rationale*) que descreve as questões e preocupações que os agentes têm acerca de processos existentes e alternativas propostas, e como eles podem ser endereçados em termos de uma rede de relacionamentos *means-end*.

Os elementos do modelo SD do framework i* são: o Ator (*Actor*) que, em geral, é uma unidade para a qual dependências intencionais podem ser descritas. Este pode ser especializado em três tipos a saber: Agentes (*Agents*), Papéis (*Roles*) e Posições (*Positions*). Um Agente é um ator com manifestações físicas e concretas, como um indivíduo, mas pode ser usado para referir-se a humanos assim como agentes artificiais (hardware/software). Um Papel é uma caracterização abstrata do comportamento de um

[1] https://github.com/RE-Projects/JiStar

ator social dentro de um contexto especializado ou domínio. Uma Posição é uma abstração intermediária entre um papel e um agente. Os atores possuem relacionamento específicos entre si: Um agente ocupa uma posição (*occupies*); mas também pode desempenhar um papel (*plays*); enquanto uma posição cobre (*covers*) vários papéis, e; todos eles são um tipo de (*is-a*) ator. Todos os tipos de ator podem ter subpartes (*is-part-of*). Todos os tipos de atores ainda podem ser uma instanciação de um ator mais genérico (ins). O relacionamento de Dependência (*Dependency*) estabelece uma relação entre dois atores onde um depende (*depender*) do outro (*dependee*) para alcançar alguma meta ou objetivo (*dependum*). Um *dependum* pode ser uma meta, uma meta-flexível, uma tarefa ou um recurso. Cada um será explicado no modelo SR.

Os elementos do modelo SR do framework i* são: a Meta (*Goal*) uma condição ou estado de algo no mundo que o ator gostaria de alcançar, enquanto Meta-Flexível (*Softgoal*) é uma condição no mundo que o ator gostaria de alcançar, porém os critérios para que esta condição seja alcançada não são claramente definidos a princípio e estão sujeitos à interpretação. Geralmente é uma meta de qualidade que guia ou restringe os outros elementos. A Tarefa (*Task*) especifica um modo de fazer algo. Quando uma tarefa é especificada como um subcomponente de uma tarefa maior, isto restringe a tarefa maior àquele curso de ação. O Recurso (*Resource*) é uma entidade (física ou informacional). Finalmente, uma fronteira de ator (ator genérico, agente, papel ou posição) é usada como uma estrutura modular para representar o raciocínio interno de um ator.

O relacionamento Meios-Fim (*Means-End*) indica uma forma para alcançar um propósito (i.e.: meta, meta-flexível, tarefa ou recurso). É possível indicar, no modelo, diferentes meios para alcançar um mesmo fim, sabendo-se que são alternativas e deverá ser seguida ou uma ou outra. Geralmente, uma tarefa é o meio para alcançar algum destes propósitos: uma meta a ser alcançada; uma tarefa a ser executada; um recurso a ser produzido; e uma meta-flexível a ser alcançada a contento. Um link de Contribuição (*Contribution*) é usado no contexto de metas-flexíveis para informar se o meio leva a uma contribuição positiva ou negativa. O relacionamento de Decomposição (*Decomposition*) é usado a partir de uma tarefa para indicar a decomposição da tarefa em sub-meta, subtarefa, recurso para, e meta-flexível para.

2.2 Mapeamento de Modelo i* para Sistemas OO

Na literatura, encontramos trabalhos com propostas para integrar modelos de metas (onde são especificados requisitos organizacionais) e modelos orientados a objetos (OO). Em [6], os autores criaram um conjunto de regras de transformação de modelos i* para modelos OO usando pUML (precise UML) em conjunto com OCL (*Object Constraint Language*). O conjunto é formado por seis regras gerais de transformação e esta transformação pode ser realizada com o apoio da extensão GOOD (*Goals into Object Oriented Development*) para a ferramenta Rational Rose. Esta extensão faz parte da ferramenta OME (*Organization Modeling Environment*).

Em 2003, este conjunto de regras foi estendido [1] e para suportá-lo foi desenvolvida uma versão estendida da ferramenta GOOD denominada XGOOD. Este conjunto de regras estendido (ver Tabela 1) foi publicado em livro no ano de 2011 [7] e em 2015 foi a base para o desenvolvimento de um novo ferramental de apoio. Neste novo

ferramental de apoio, o modelo i* é desenvolvido na ferramenta iStarTool [14] e, posteriormente, exportado no formato XMI para que seja importado na ferramenta Eclipse (com um plugin da linguagem ATL). Em seguida, as regras de transformação descritas agora em ATL são aplicadas e o modelo OO de saída é gerado também no formato XMI, o que possibilita sua importação em uma ferramenta CASE para visualização do modelo resultante.

Se utilizássemos as heurísticas de mapeamento do modelo de metas para representações OO (ver Tabela 1), tal como são descritas, no sentido para trás, então poderíamos gerar modelos extensos e menos abstratos. Além disso, alguns elementos da POO como interfaces e classes abstratas que, muitas vezes são utilizadas para tratar variabilidade, não teriam correspondência para mapeamento e seria difícil definir se uma classe deveria ser mapeada para um agente, papel ou posição (ver regra G1.1 na Tabela 1). Nosso trabalho difere deste por fazer o mapeamento no sentido do código OO para o modelo de metas i* através do uso de metadados de código, ou seja, nós identificamos para elementos da POO quais os elementos i* eles podem representar.

3 Trabalhos Relacionados

O trabalho de Yu et. al. [25] propõe uma estratégia de engenharia reversa de sistemas legados com vistas a obter um modelo de metas. O referido trabalho recupera modelos de meta a partir de código legado estruturado (i.e.: OO) e não estruturado. De um modo geral, a técnica é composta por quatro passos principais: 1) refatorar o Código fonte através da extração de métodos considerando seus comentários de código; 2) converter o código refatorado em um programa estruturado e abstrato através de refatoração com statechart e da construção de gráficos de hammock; 3) extrair um modelo de metas a partir da árvore de sintaxe abstrata do programa estruturado; e 4) identificar NFR e derivar metas-flexíveis baseado na rastreabilidade entre o código e o modelo de metas e no uso de testes de função. Ou seja, os RNF são derivados a partir da observação dos efeitos nas métricas de qualidade ao se ligar/desligar os RNF identificados. A estratégia proposta neste trabalho se diferencia pelo uso de anotações de código para auxiliar na rastreabilidade dos elementos do POO (Programação Orientada a Objetos) para os elementos do modelo i*. Além disso, fazemos uso de fronteiras de atores (ver Seção 2.1) como uma estrutura modular, possibilitando representar elementos modulares do POO.

O trabalho de Serrano [20] propõe o desenvolvimento de sistemas multiagentes centrado no conceito de intencionalidade. Os autores descrevem um conjunto de heurísticas para o mapeamento de modelos i* para código baseado em BDI (*Belief – Desire - Intention*) usando JADEX. Em sua Tese, Serrano [20] relata que o conjunto de heurísticas de mapeamento proposto poderia ser utilizado no sentido inverso para obter um modelo de metas i* a partir de código JADEX baseado em BDI, ou seja, uma rastreabilidade bidirecional. O nosso trabalho difere deste por possuir um conjunto de heurísticas que mapeiam sistemas OO para modelo i*, possibilitando a rastreabilidade da intencionalidade de sistemas que não sejam multiagentes.

Tabela 1. Regras de transformação [1, 7].

Regra	i*	pUML
G1.1	Agentes, papéis ou posições	Classes
G1.2	Relacionamento i*s-part-of*	Agregação
G1.3	Relacionamento *is-a*	Generalização / especialização
G1.4	Relacionamento *occupies*	Associação nomeada occupies
G1.5	Relacionamento *covers*	Associação nomeada covers
G1.6	Relacionamento *plays*	Associação nomeada plays
G2.1	Tarefas no modelo SD	Métodos com visibilidade pública na classe fornecedora
G2.2	Tarefas no modelo SR	Métodos com visibilidade privada na classe fornecedora
G3.1	Recursos no modelo SD	Classe, se a dependência tem característica de um objeto, ou, caso contrário, atributo com visibilidade privada na classe fornecedora
G3.2	Recurso (sub-recurso) no modelo SR	Atributo com visibilidade privada na classe, se não puder ser entendido como um objeto, ou, caso contrário, uma classe
G4.1	Meta-Flexível no modelo SD	Atributo com visibilidade pública na classe fornecedora
G4.2	Meta-Flexível no modelo SR	Atributo com visibilidade privada na classe à qual a meta-flexível pertence
G5	Relacionamento *task-decomposition*	Pré e pós condições em OCL na operação correspondente
G6.1	Relacionamento *means-end* de Meta-flexível para Meta-Flexível	A disjunção dos valores dos meios implica no valor do fim
G6.2	Relacionamento *means-end* de tarefa para meta-flexível e tarefa para recurso	A pós-condição das tarefas meio implicam no valor do fim
G6.3	Relacionamento *means-end* de tarefa para tarefa	A disjunção da pós-condição dos meios implica as pós-condições do fim

4 JiStar – Conjunto de Anotações de Código

O framework JiStar possibilita a rastreabilidade entre o código fonte Java e as metas do sistema, permitindo a geração automática de um modelo de metas em i*. O framework possui um conjunto de anotações de código relacionadas aos elementos do framework i* para a modelagem de metas. Com o uso deste framework é possível verificar quais metas estão sendo atendidas, reutilizar operacionalizações de metas flexíveis, realizar análise de impacto no caso de mudanças nas metas, entre outras vantagens advindas da

rastreabilidade entre o código e o modelo de requisitos. A rastreabilidade é dada pelo uso dos nomes dos elementos no modelo que é conservado nas anotações de código

Escrevemos um conjunto de anotações para manter a rastreabilidade entre os modelos de metas em i* para o código Java, onde a aplicação de cada tipo de anotação foi inspirada nas nossas heurísticas iniciais [15]. A vantagem de usar anotações ao invés as heurísticas é que é possível anotar somente os elementos essenciais para a compreensão das metas organizacionais que são alcançadas por meio da aplicação, omitindo detalhes técnicos que poderiam deixar o modelo pouco abstrato e com um grande volume de informações.

É importante destacar que todas as anotações foram criadas com política de retenção *runtime* para estarem disponíveis em tempo de execução e poderem ser acessadas por reflexão. Todas as anotações também foram especificadas para serem apresentadas com o JavaDoc do código. Para auxiliar no uso das anotações, disponibilizamos um tutorial junto ao Framework Futuramente, é possível desenvolver algum ferramental de apoio para apoiar o uso das anotações.

Apresentaremos o conjunto de anotações e a forma de utilizá-las. Criamos anotações para representar elementos (@Actor, @Goal, @Softgoal, @Resource e @Task) e relacionamentos entre os elementos (@Contribution, @TaskDecomposition, @MeansEnd). Os atributos obrigatórios de cada anotação serão indicados com *.

Para ilustrar o uso das anotações, utilizamos um sistema didático para cadastro de receitas cuja a estrutura do projeto encontra-se na Fig 1. A classe ReceitaGui da Fig 1 é a interface gráfica para cadastro das receitas. É a partir dela que os usuários começam a ter suas metas satisfeitas. A classe Receita da Fig 1. É a classe de entidade que será persistida com os dados das receitas, enquanto a classe ReceitaControle trata as solicitações vindas da interface gráfica. Para implementar a persistência das receitas na base de dados, a classe FabricaConexao instancia a conexão com o banco de dados e a classe ReceitaDAO (que implementa a interface IDAO) implementa os métodos com comandos de DML (*Data Modification Language*).

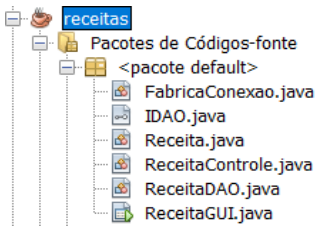

Fig. 1. Estrutura do projeto de exemplo “receitas”.

Como o modelo de metas é um modelo mais abstrato que o nível de código, o engenheiro de software deve anotar somente as classes e elementos que contribuem para a compreensão das razões da existência aplicação e suas qualidades. Classes e outros elementos cuja razão de ser seja puramente técnica, como FabricaConexao e IDAO, não devem ser anotados. Assim, o modelo ficará mais abstrato e com menos elementos do que o código. Sugerimos que os engenheiros comecem a anotar pelas classes que

fazem interface com os usuários e utilize o seu conhecimento tácito sobre as razões da aplicação.

- **@Actor(** *name**=”nome do ator” e *type**=[*general* | *agente* | *role* | *position*]**) –** Esta anotação pode ser utilizada em classe, interface (incluindo anotações), ou enumeração. Ela é utilizada para indicar cada ator que está sendo representado por determinado código. Ao exportar um modelo i* em qualquer formato, somente os elementos associados a algum ator aparecerão no modelo.

```
@Actor(name = "Cozinheiro", type = ActorType.AGENT)
public class ReceitaGUI extends javax.swing.JFrame {...}
```

- **@Goal(** *name**=”nome da meta”, *description*=”descrição da meta”**)** – Esta anotação pode ser utilizada em classe, interface (incluindo anotações) e enumeração para indicar as metas que os atores conseguem e/ou devem alcançar naquela unidade de código. É possível utilizar mais de uma @Goal por unidade de código.

```
@Actor(name = "Cozinheiro", type = ActorType.AGENT)
@Goal(name = "Que receitas sejam criadas", description =
"Criar novas receitas")
public class ReceitaGUI extends javax.swing.JFrame {…}
```

- **@Softgoal(** *name**=”nome da meta flexível [tópico]”, *description*=”descrição da meta flexível”**)** – Esta anotação pode ser utilizada em classe, interface (incluindo anotações), enumeração e/ou atributo para relacionar as metas flexíveis (metas de qualidade) que os atores conseguem e/ou devem alcançar naquela unidade de código. É possível utilizar mais de uma @Softgoal por unidade de código. As operacionalizações destas metas flexíveis, quando disponíveis, serão indicadas pela anotação @Contribution.

```
@Actor(name="Cozinheiro", type=ActorType.AGENT)
...
@Softgoal(name = "Persistir[2] [receita]", description =
"Persistir as receitas")
public class ReceitaGUI extends javax.swing.JFrame {…}
```

- **@Task(** *name**=”nome da tarefa” e *description**=” descrição da tarefa”**)** – Esta anotação pode ser utilizada em métodos para fornecer informações acerca da tarefa realizada pelo método em questão.

```
@Task(name="Gravar Receita", description="Grava a receita
na base de dados")
private void jButton2ActionPerformed(java.awt.event.Ac-
tionEvent evt) {…}
```

[2] A Persistência é um requisito não funcional que pode ser operacionalizado de diferentes formas como persistência em base de dados ou persistência em arquivo [22].

- **@Resource(** *name**=”nome do recurso”**)** – Esta anotação pode ser utilizada em tipos (i.e.: classe, interface (incluindo anotações), enumeração), atributo e/ou variável local para indicar recursos. É importante ressaltar que uma unidade de código como uma classe pode ser entendida como um recurso, como classes de entidade, por exemplo. No caso de uso em tipos, estes serão apresentados no modelo quando forem utilizados por algum ator e não poderão acumular a anotação @Actor.

```
@Resource(name="Receita")
public class Receita {
    private int id;
    private String nome;
    private String ingredientes;
    private String preparo;    ...}
```

- **@Contribution(** *type**=[*make* | *help* | *hurt* | *break*), *softgoal**=”meta flexível”**)** – Esta anotação pode ser utilizada em métodos para indicar um relacionamento de contribuição do método para uma meta flexível. Se o método possuir a anotação @Task, esta será relacionada à meta flexível em questão. Caso contrário, será criada uma tarefa com o nome do método. É possível utilizar mais de uma @Contribution por unidade de código.

```
@Contribution(type=ContributionType.MAKE, softgoal="Per-
sistir [receita]")
...
 private void jButton2ActionPerformed(java.awt.event.Ac-
tionEvent evt) {
        try{
        ReceitaControle controle = new ReceitaControle();
        Receita r = new Receita(txtNome.getText()); ...}
```

- **@TaskDecomposition(**type***=**[resource_for | softgoal_for | goal | sub_task], elemento*****=”nome do elemento”**)** – Esta anotação pode ser utilizada em métodos, uso de algum tipo ou declaração de parâmetro. Ela indica um relacionamento de decomposição de uma tarefa para uma subtarefa, um recurso para a tarefa, uma submeta ou uma meta flexível para a tarefa. É possível utilizar mais de uma @TaskDecomposition por unidade de código.

```
@TaskDecomposition(type=TaskDecomposition-
Type.RESOURCE_FOR, element="Receita")
...
private void jButton2ActionPerformed(java.awt.event.Ac-
tionEvent evt) {  ...
        try{
        ReceitaControle controle = new ReceitaControle();
        @Resource(name = "Receita") Receita r = new Re-
ceita(txtNome.getText()); ...}
```

- **@MeansEnd(** *endType**=[resource | goal], *end**=”nome da meta ou recurso”**)** – Esta anotação pode ser utilizada em métodos para indicar que aquele método é uma tarefa (meio) para alcançar determinada meta, meta flexível ou recurso. É possível utilizar mais de uma @MeansEnd por unidade de código.

```
@MeansEnd(endType=MeansEndType.GOAL,end="Criar receitas")
...
private void jButton2ActionPerformed(java.awt.event.Ac-
tionEvent evt) { ...
   try{ ReceitaControle controle = new ReceitaControle();
        @Resource(name = "Receita")
         Receita r = new Receita(txtNome.getText()); ...}
```

Futuramente, desenvolveremos anotações para fornecer meta informações referente aos relacionamentos entre os atores, incluindo informações de dependência. Além disso, estamos averiguando os benefícios de usar a APT (*Annotation Processing Tool*) para customizar a forma como o compilador Java processa as nossas anotações para o compilador sinalizar erros de uso das anotações em tempo de compilação.

5 JiStar – Exportador de Modelo i*

O framework JiStar permite a exportação das metainformações referente a intencionalidade do código para um modelo i* SR no formato HTML e/ou PiStar[3] 2.0. Quando criarmos anotações para o relacionamento de dependência, poderemos exportar modelos i* SD também. Um modelo de metas facilita a leitura das metas por parte dos engenheiros de requisitos, clientes e todas as partes interessadas no negócio suportado pelo sistema. Com um modelo de metas atualizado em mãos, todas as partes interessadas podem contribuir para a evolução do software.

Para gerar o modelo de metas a partir do código fonte anotado, após engenheiro de software anotar o código (vide exemplos da Seção 4), deve executar o gerador de modelo de metas passando o caminho dos arquivos compilados do projeto (*.class) e o modelo desejado (-html ou -pistar), conforme trecho de comando abaixo (exemplo para Windows). Neste exemplo, o JiStar gera um arquivo chamado goal_model que pode ser carregado na ferramenta PiStar 2.0. O modelo resultante do exemplo é apresentado na Fig 2. O agente Cozinheiro possui a meta “Que receitas sejam criadas” e a meta flexível “Persistir as receitas” para utilizá-las futuramente. A tarefa “Gravar Receita” é um meio para alcançar a meta “Que receitas sejam criadas” e contribui para a meta flexível “Persistir as receitas”, para isso, ela consome o recurso “Receita”.

```
java -jar JiStar-1.0.jar "\\caminho\*.class" -pistar
```

Conseguimos exportar para PiStar[3] 2.0, que suporta a versão 2.0 do framework i*, por ainda não implementarmos as anotações referentes a alguns relacionamentos. Além

[3] https://www.cin.ufpe.br/~jhcp/pistar/tool/index.html

disso, futuramente, adicionaremos a possibilidade de exportação para iStarML que suporta a versão 1.0 do framework i*.

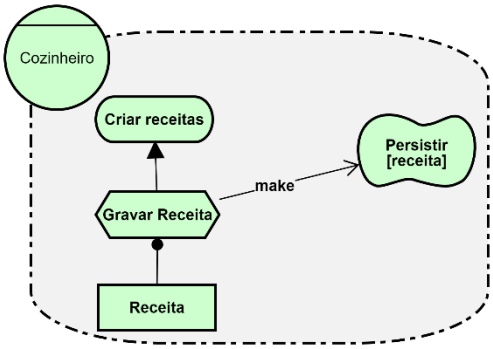

Fig. 2. Modelo de metas i* do exemplo didático sistema de receitas.

6 Estudo de Viabilidade

Para avaliar a viabilidade do framework JiStar, optamos por resgatar as metas do aplicativo RioBus[4]. O RioBus é app desenvolvido em Java para a plataforma Android que mostra a localização geográfica dos ônibus da cidade do Rio de Janeiro. Os dados são oferecidos publicamente pela prefeitura do Rio de Janeiro, em parceria com a FETRANSPOR e Iplanrio. As posições dos ônibus são recuperadas pelo GPS embarcado neles, enviados para a FETRANSPOR e, por fim, a Iplanrio a disponibiliza na página do projeto de dados abertos Data Rio.

Estudo

Os detalhes do estudo serão apresentados em um relatório técnico, resumidamente:

Preparação do estudo: fizemos o download do código fonte no GitHub, estudamos o código fonte com seus comentários e utilizamos o app.

Execução: realizamos a anotação do código fonte e para nos auxiliar na identificação de metas flexíveis (qualidades) de consciência utilizamos o catálogo de consciência de software [8]. Em seguida fizemos a extração do modelo i* (ver Fig. 3).

Análise do Resultado: o modelo resultante apresenta três atores, onde o ator "Usuário" foi obtido a partir da classe MainActivity, o ator "Marcador de Mapa" foi obtido a partir da classe MapMarker e o ator "Localizador de ônibus" foi obtido a partir da classe BusSearchTask. Os principais elementos internos de cada agente, que são relacionados diretamente ao negócio, foram anotados e o modelo resultante é apresentado na Fig 3. As demais classes não foram consideradas essenciais para o entendimento do negócio ou foram anotadas como recurso (estas aparecem somente dentro do raciocínio estratégico de cada agente quando anotadas também como um recurso do agente).

Avaliação do Resultado: para avaliar o modelo i* exportado pelo framework JiStar, convidamos três engenheiros de software da indústria com mais de 14 anos de

[4] https://github.com/RioBus/android-app

experiência para avaliar o modelo resultante quanto a sua compreensibilidade e usabilidade. A Tabela 2 apresenta as perguntas da avaliação e as respostas dos engenheiros de software.

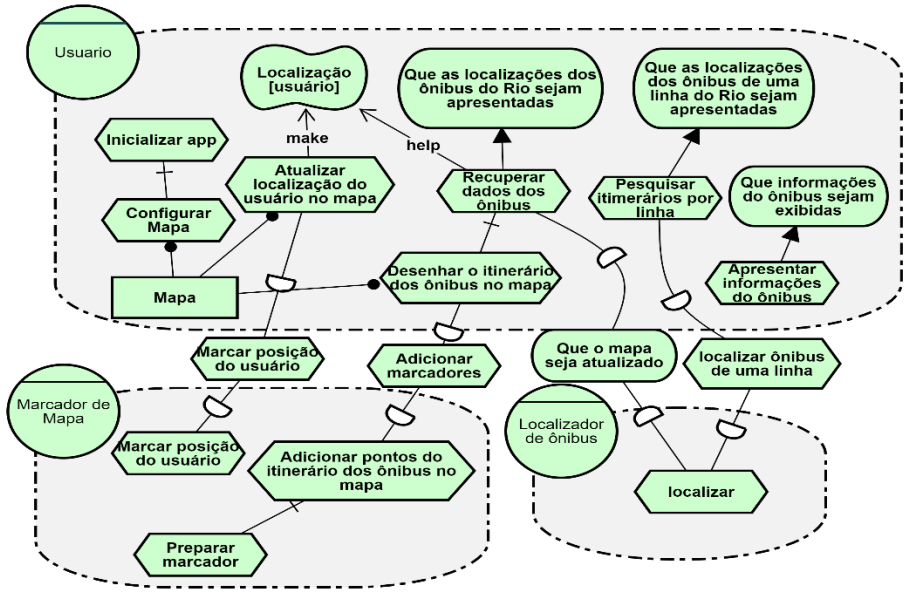

Fig. 3. Modelo i* extraído após anotação do código fonte.

Tabela 2. Perguntas e respostas de avaliação do estudo com engenheiros de software

Perguntas	Engenheiro SW 1	Engenheiro SW 2	Engenheiro SW 3
Você utilizaria o modelo em reuniões com o cliente para elicitação de requisitos em uma evolução do sistema?	Sim	Não, pois os meus clientes não conhecem a linguagem.	Sim
Você considera o modelo mais abstrato que o código?	Sim	Sim	Sim
Você conseguiria explicar o propósito do sistema após ler o modelo?	Sim	Sim	Sim
Você considera importante poder manter a rastreabilidade entre o código e as metas do sistema?	Sim	Sim	Sim

Ameaças à validade do estudo: o estudo ainda é preliminar e envolveu apenas 3 engenheiros de softwares da indústria. Além disso, as anotações foram aplicadas apenas pelos pesquisadores e os engenheiros avaliaram o resultado, o qual pode variar de

acordo com a pessoa que anota o código. O uso demasiado das anotações pode gerar um modelo com muitos elementos e pouco abstrato, mesmo não sendo esta a nossa recomendação.

Apesar das ameaças à validade do estudo, de acordo com o resultado da avaliação, temos um primeiro indício de que o uso do framework JiStar pode auxiliar na reconciliação de sistemas OO com suas metas, possibilitando a exportação de um modelo mais abstrato e com linguagem mais apropriada para engenheiros de requisitos e clientes.

7 Conclusão

Neste trabalho, propomos o framework JiStar para auxiliar na rastreabilidade para trás entre código fonte OO e modelos de metas i*. Nosso framework contribui para: (A) Fornecer um modelo de rastreabilidade do código para as metas pois com o uso das anotações é possível adicionar meta dados de intencionalidade (i.e.: metas) ao código, fazendo a manutenção do vínculo diretamente e somente no código. Quando desejar ver no formato de modelo, basta exportar o modelo; (B) Automatizar a exportação de um modelo de metas pois a parir do código anotado com o framework JiStar podemos exportar modelos i* nos formatos HTML e PiStar 2.0; (C) Prover modelos i* mais abstratos que o código pois seguindo as dicas de uso, os engenheiros de software anotarão somente os elementos do código que possuem relação com a intencionalidade dos usuários (i.e.: suas metas). Isto gera modelos mais abstratos e menos extensos. O uso demasiado das anotações pode levar a um modelo menos abstrato. E; (D) Prover modelos com informações compreensíveis por engenheiros de requisitos e clientes pois à medida que as anotações permitem o uso de uma linguagem humanizada, o modelo resultante permite que os engenheiros de requisitos e os clientes se comuniquem e utilizem o modelo para a evolução do software. O engenheiro de software, por sua vez, consegue identificar os pontos de evolução no código devido ao vínculo entre o código e as metas.

Nosso objetivo era conseguir um modelo de metas a partir de sistemas construídos sem a perspectiva GORE, de tal modo que este modelo possua uma linguagem mais apropriada para engenheiros de requisitos e clientes sem perder a rastreabilidade para o código foi alcançado. O modelo resultante pode ser utilizado para a evolução do software. Nossa pesquisa utiliza este modelo na reengenharia de sistemas autoadaptativos.

Futuramente, iremos ampliar nosso conjunto de anotações para contemplar relacionamentos entre atores, incluindo o relacionamento de dependência. Isto permitirá a geração de modelo i* SD. Além disso, pretendemos exportar o modelo também para o formato iStarML [5] que suporta o framework i* na versão 1,0 e nos possibilitará gerar diagramas com todas as metainformações que estão disponíveis no JiStar. Realizaremos novos estudos de caso com vistas a avaliar o esforço e a complexidade de uso do framework pois neste trabalho avaliamos o modelo resultante.

Agradecimentos

Os autores agradecem a Petrobras e a PUC Rio por todo o apoio fornecido para a realização desta pesquisa.

Referências

1. Alencar, F.M. et al.: New Mechanism for the Integration of Organizational Requirements and Object Oriented Modeling. In: WER. pp. 109–123 (2003).
2. Almentero, E.K.: Dos Requisitos ao Código: Um Processo para Desenvolvimento de Software mais Transparente. PUC-Rio (2013).
3. Bolchini, D., Paolini, P.: Capturing Web Application Requirements through Goal-Oriented Analysis. In: WER. pp. 16–28 (2002).
4. Bresciani, P. et al.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems. 8, 3, 203–236 (2004).
5. Cares, C. et al.: iStarML: The i* mark-up language. (2007).
6. Castro, J. et al.: Closing the gap between organizational requirements and object oriented modeling. Journal of the Brazilian Computer Society. 7, 1, 05–16 (2000).
7. Castro, J. et al.: Integration of i* and object-oriented models. Social modeling for requirements engineering. 457–484 (2011).
8. Cunha, H. de S.: Desenvolvimento de Software Consciente com Base em Requisitos. PUC-Rio (2014).
9. Dalpiaz, F. et al.: iStar tutorial online, https://www.dropbox.com/s/4l2k4tbywb8wekk/iStar-tutorial-online.pdf?dl=0, last accessed 2020/03/27.
10. Darwish, N.R., Zohdy, B.S.: Goal Modeling Techniques for Requirements Engineering. International Journal of Computer Science and Information Security. 14, 7, 739 (2016).
11. Gonçalves, E. et al.: A Catalogue of iStar Extensions. In: WER. (2018).
12. Horkoff, J. et al.: Using goal models downstream: a systematic roadmap and literature review. International Journal of Information System Modeling and Design (IJISMD). 6, 2, 1–42 (2015).
13. Jaqueira, A. et al.: Using i* Models to Enrich User Stories. iStar. 13, 55–60 (2013).
14. Malta, Á. et al.: iStarTool: Modeling Requirements using the i* Framework. In: IStar. pp. 163–165 (2011).
15. Moura, A.M. da M.: Awareness Driven Software Reengineering. In: Requirements Engineering Conference (RE), 2017 IEEE 25th International. pp. 550–555 IEEE (2017).
16. Moura, A.M. da M. et al.: Improving Urban Mobility for the Visually Impaired using the Awareness Quality. In: Proceedings of the XVIII Brazilian Symposium on Software Quality. pp. 59–68 (2019).
17. de Oliveira, R.F. et al.: Reengineering for Accessibility: A Strategy Based on Software Awareness. In: Proceedings of the 17th Brazilian Symposium on Software Quality. pp. 180–189 ACM (2018).
18. Pimentel, J., Castro, J.: pistar tool–a pluggable online tool for goal modeling. In: 2018 IEEE 26th International Requirements Engineering Conference (RE). pp. 498–499 IEEE (2018).
19. Sayão, M., do Prado Leite, J.C.S.: Rastreabilidade de requisitos. RITA. 13, 1, 57–86 (2006).
20. Serrano, M.: Desenvolvimento Intencional de Software Transparente Baseado em Argumentação. PUC-Rio (2011).
21. da Silva, L.F. et al.: C & L: uma ferramenta de apoio à engenharia de requisitos. PUC (2004).
22. da Silva, L.F.: Uma Estratégia Orientada a Aspectos para a Modelagem de Requisitos. Tese de Doutorado, Computer Science Department, PUC-Rio (2006).
23. Yu, E.: Agent orientation as a modelling paradigm. Wirtschaftsinformatik. 43, 2, 123–132 (2001).
24. Yu, E.: Modelling strategic relationships for process reengineering. (1995).
25. Yu, Y. et al.: Reverse engineering goal models from legacy code. In: Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on. pp. 363–372 IEEE (2005).