PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Ana Maria da Mota Moura

Reengenharia de Sistemas Autoadaptativos Guiada pelo Requisito Não Funcional de Consciência de *Software*

Tese de Doutorado

Tese apresentada como requisito parcial para obtenção do grau de Doutor pelo Programa de Pós–graduação em Informática do Departamento de Informática da PUC-Rio.

Orientador: Prof. Julio Cesar Sampaio do Prado Leite

Rio de Janeiro

Setembro de 2020

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Ana Maria da Mota Moura

Reengenharia de Sistemas Autoadaptativos Guiada pelo Requisito Não Funcional de Consciência de *Software*

Tese apresentada como requisito parcial para obtenção do grau de Doutor pelo Programa de Pós–graduação em Informática do Departamento de Informática da PUC-Rio. Aprovada pela Comissão Examinadora abaixo.

Prof. Julio Cesar Sampaio do Prado Leite
Orientador
Departamento de Informática - PUC-Rio

Prof. Alberto Barbosa Raposo
Departamento de Informática - PUC-Rio

Prof. Edward Hermann Hæusler
Departamento de Informática - PUC-Rio

Prof. Eduardo Kinder Almentero
Departamento de Informática - UFRRJ

Prof. Jaelson Freire Brelaz de Castro
Departamento de Informática - UFPE

Rio de Janeiro, 24 de setembro de 2020

Ana Maria da Mota Moura

Graduou-se em Tecnologia em Informática pelo CEFET Campos em 2004. Gradou-se em Ciência da Computação pela Universidade Candido Mendes em 2005. Recebeu o título de Mestre em Ciências em Engenharia de Sistemas na COPPE/UFRJ em 2009. É engenheira de *software* na Petrobras e professora de informática na FAETEC RJ.

Ficha Catalográfica

Moura, Ana Maria da Mota

Reengenharia de Sistemas Autoadaptativos Guiada pelo Requisito Não Funcional de Consciência de *Software* / Ana Maria da Mota Moura; orientador: Julio Cesar Sampaio do Prado Leite – Rio de Janeiro: PUC, Departamento de Informática, 2020.

189 f. : il. (color.) ; 29,7 cm.
1. Tese (doutorado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2020.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Reengenharia guiada por requisitos. 3. Sistemas Autoadaptativos. 4. Consciência de *software.* 5. Requisito Não Funcional (RNF). 6. i* (i-estrela). 7. Engenharia de requisitos orientada a metas. 8. JiStar. I. Julio Cesar Sampaio do Prado Leite. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Dedicatória

Dedico esta pesquisa aos meus pais, esposo e filhos pelo apoio incondicional em diversos momentos da minha trajetória acadêmica. Sem eles, nada seria possível.

Agradecimentos

A Deus, pelas bênçãos recebidas!

Em especial aos meus pais, Antonio Carlos de Moura e Claudia Maria R. da M. Moura, simplesmente por tudo.

Um agradecimento especial ao meu esposo, Rafael, à minha filha Maria Antonia e aos meus gatos Romeu e Julieta pelo companheirismo, carinho, paciência, otimismo e amor.

Ao meu irmão Luís Carlos, minha irmã de coração Fernanda e meus sobrinhos e afilhados pelas palavras de carinho, amor e amizade inigualáveis.

Ao professor Julio C. S. do P. Leite, meu orientador, por ter me aceito como sua orientanda e membro deste excelente grupo de pesquisa, além da paciência, confiança e compreensão dos momentos difíceis pelos quais passei. Muito obrigado por compartilhar parte de seu conhecimento, me mostrando o caminho da pesquisa e da superação de desafios.

Aos membros da minha banca pelos ensinamentos e paciência.

Ao grupo de Engenharia de Requisitos, cada um de vocês, pela amizade, carinho e auxílio.

Aos meus amigos, pelo apoio, carinho, ajuda, companheirismo, paciência, pelos os momentos que cada um contribuiu com um toque especial e uma palavra de motivação.

À Petrobras e a PUC-Rio, por me dar a oportunidade de estudar em uma universidade esplêndida.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Resumo

Ana Maria da Mota Moura; Sampaio do Prado Leite, Julio Cesar. **Reengenharia de Sistemas Autoadaptativos Guiada pelo Requisito Não Funcional de Consciência de *Software***. Rio de Janeiro, 2020. 154p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Nos últimos anos, foi desenvolvido um número significativo de sistemas autoadaptativos (i.e.: sistemas capazes de saber o que está acontecendo sobre si mesmo e que, consequentemente, implementam parcialmente a qualidade de consciência). A literatura tem pesquisado extensivamente o uso da engenharia de requisitos orientada a metas e o uso da arquitetura de referência MAPE (*Monitor-Analyze-Plan-Execute*) para o desenvolvimento de sistemas autoadaptativos. Entretanto, construir tais sistemas com base em estratégias de referência não é trivial, podendo resultar em problemas estruturais que impactam negativamente alguns atributos de qualidade do produto final (e.g.: reusabilidade, modularidade, modificabilidade e entendibilidade). Neste contexto, estratégias de reengenharia para a reorganização de tais sistemas são pouco exploradas, limitando-se a recuperar e a reestruturar a lógica da adaptação em modelos de baixo nível. Esta prática mantém a dificuldade do tratamento da qualidade de consciência como um requisito não funcional (RNF) de primeira classe, impactando diretamente na seleção da arquitetura e implementação do sistema. Nossa pesquisa visa mitigar esse problema através de uma estratégia de reengenharia de sistemas autoadaptativos, centrada no RNF de consciência de *software*, com vistas a auxiliar na remoção de alguns problemas recorrentes na implementação do MAPE conforme a literatura. A estratégia de reengenharia está organizada em quatro subprocessos: (A) recuperar a intencionalidade do sistema com ênfase em suas metas de consciência, gerando um modelo de metas *AS-IS*; (B) especificar o modelo de metas *TO-BE* reutilizando um conjunto de *SRconstructs* para operacionalizar o RNF de consciência de *software* conforme o padrão MAPE; (C) redesenhar o sistema revisando as operacionalizações de consciência e selecionando as tecnologias para implementar o MAPE, e; (D) finalmente, reimplementar o sistema conforme nova estrutura, adicionando metainformações de código para manter a rastreabilidade para o mecanismo de autoadaptação visando facilitar novas evoluções. O escopo da nossa pesquisa são sistemas autoadaptativos orientados a objetos (OO), utilizando o *framework* i* como linguagem para os modelos orientados a metas. Nossos resultados de avaliações em sistemas autoadaptativos OO desenvolvidos em Java para dispositivos móveis com Android demonstram que a estratégia auxilia no realinhamento do sistema com as boas práticas recomendadas pela literatura facilitando futuras evoluções.

Palavras–chave

Sistemas Autoadaptativos; Reengenharia de *Software*; Consciência de *Software*; Engenharia Reversa para Modelo de Metas; MAPE.

Abstract

Ana Maria da Mota Moura; Sampaio do Prado Leite, Julio Cesar. **Self-Adaptive Systems Reengineering Driven by the *Software* Awareness Non-Functional Requirement**. Rio de Janeiro, 2020. 154p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

In recent years, a significant number of self-adaptive systems (i.e.: systems capable of knowing what is happening about themselves, and consequently partially implementing the quality of awareness) have been developed. The literature has extensively researched the use of *goal* oriented requirements engineering and the use of the MAPE (Monitor-Analyze-Plan-Execute) reference architecture for the development of self-adaptive systems. However, building such systems based on reference strategies is not trivial, it can result in structural problems that negatively impact some quality attributes of the final product (e.g.: reusability, modularity, modifiability and understandability). In this context, reengineering strategies for the reorganization of such systems are poor explored, and they are limited to recovering and restructuring the logic of adaptation in low-level models. This approach keeps the difficulty of treating the awareness quality as a first-class non-functional requirement (NFR) directly affecting architecture selection and implementation of the system. Our research aims to mitigate this problem through a strategy of reengineering self-adaptive systems, centered on *software* awareness as an NFR. This strategy will assist in the removal of some recurring problems in the implementation of MAPE according to the literature. The reengineering strategy is organized into four sub-processes: (A) recover the intentionality of the system with an emphasis on its awareness *goals*, generating an *AS-IS goal* model; (B) specify the *TO-BE goal* model by reusing a set of *SRconstructs* to operationalize the *software* awareness NFR according to the MAPE standard; (C) redesign the system by reviewing the operationalizations of awareness and selecting the technologies to implement the MAPE, and; (D) finally, reimplement the system according to a new structure, adding code metadata to maintain traceability for the self-adaptation mechanism in order to facilitate new evolutions. The scope of our research is object-oriented (OO) self-adaptive systems using the i* *framework* as a language for *goal*-oriented models. Our results of evaluations, for OO self-adaptive systems developed in Java for mobile devices with Android, show that the strategy helps in realigning the system with the best practices recommended by the, facilitating future developments.

Keywords

Self-adaptive systems; *Software* Reengineering; *Software* Awareness; Reverse Engineering for *Goals* Model; MAPE.

Sumário

Lista de Figuras

Lista de Tabelas

Listas

Lista de Abreviaturas e Siglas

ABSA	Architecture-Based-Self-Adaptation
API	Application Program Interface
BDI	Belief – Desire - Intention
FIPA	The Foundation for Intelligent Physical Agents
GOOD	Goals into Object Oriented Development
GORE	Goal Oriented Requirements Engineering
GQO	Goal Question Operationalization
GSP	Goals, Skills and Preferences
HIME	Hierarchical i-star Modeling Editor
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IoT	Internet of Things
JADE	Java Agent DEvelopment Framework
MAPE	Monitor–Analyze–Plan–Execute
NFR	Non-Functional Requirement
OCL	Object Constraint Language
OME	Organization Modeling Environment
pUML	precise UML
RNF	Requisito Não Funcional
SEAMS	International Symposium on Software Engineering for Adaptive and Self-Managing Systems
SADT	Structured analysis and design technique
SD	Strategic Dependency
SIG	Softgoal Interdependency Graph
SMA	Sistemas Multiagentes
SR	Strategic Rationale
SRP	Single Responsibility Principle
UML	Unified Modeling Language

"Educa a criança no caminho em que deve andar; e até quando envelhecer não se desviará dele.

Provérbios 22:6"

1
Introdução

Neste capítulo, contextualizamos o desenvolvimento de sistemas autoadaptativos, corroborando com a motivação da nossa pesquisa. Apresentamos também os desafios ao abordar técnicas de reorganização de sistemas autoadaptativos. Em seguida, caracterizamos a nossa pesquisa e a solução proposta. Por fim, especificamos a organização do trabalho.

1.1.
Contexto

Nos últimos anos, tem-se desenvolvido um número significativo de softwares autoadaptativos[1] e autogerenciáveis[2] (i.e.: com capacidade de autonomamente se adaptar a novas circunstâncias e se recuperar de falhas (DALPIAZ; GIORGINI; MYLOPOULOS, 2009)). Por causa da capacidade de autoadaptar-se, acreditamos que softwares autoadaptativos e autogerenciáveis possuem a qualidade de ser consciente. Deste modo, adotamos a seguinte definição para consciência de *software*, a saber (CUNHA, 2014): "a capacidade do *software* de adquirir conhecimento sobre o que acontece no ambiente no qual está inserido e sobre seu próprio comportamento, através de suas próprias percepções ou por meio de informações externas". De modo geral, ao longo desta tese utilizaremos o termo *software* autoadaptativo para referir-se tanto à autoadaptação quanto ao autogerenciamento.

O desenvolvimento de tais sistemas não é trivial e a literatura de desenvolvimento de sistemas autoadaptativos recomenda a engenharia de requisitos orientada a metas como estratégia de requisitos (ALI; DALPIAZ; GIORGINI, 2010; BARESI; PASQUALE; SPOLETINI,

[1] Sistemas autoadaptativos são sistemas capazes de modificar seu comportamento e/ou estrutura em resposta à sua percepção do ambiente e do próprio sistema, e seus requisitos. (DE LEMOS et al., 2013)
[2] Sistemas auto gerenciáveis são sistemas que são capazes de autoconfiguração, auto adaptação e autocura, auto monitoramento e autoajuste, e assim por diante, muitas vezes sob a bandeira de sistemas auto- * ou autônomos. (KRAMER; MAGEE, 2007)

2010; CUNHA, 2014; FUXMAN et al., 2004; GIORGINI; MYLOPOULOS; SEBASTIANI, 2005; SOUZA; MYLOPOULOS, 2012; SUPRIANA; SURENDRO, 2018) e laços de controle (*feedback loop*) como solução arquitetural e de implementação, na qual o modelo arquitetural MAPE[3] (*Monitor–Analyze–Plan–Execute*), proposto pela IBM (IBM, 2006), tem se destacado. Entretanto, pouco se discute sobre estratégias de evolução e renovação de softwares autoadaptativos que foram construídos aquém das recomendações da literatura ou aqueles que resultaram do uso inadequado de tais recomendações. Estes sistemas podem ter sofrido impactos negativos na qualidade do produto final, como os impactos descritos por Serikawa et al. (SERIKAWA et al., 2016) e San Martín et al. (SAN MARTÍN et al., 2020), e que são apresentados na Seção 2.3.2.

Uma vez que o sistema autoadaptativo está construído, se o desenvolvimento dos requisitos e a gestão dos requisitos (SAYÃO; DO PRADO LEITE, 2006) não foram eficazes, há uma grande quantidade de conhecimentos acerca do mecanismo de autoadaptação que: ou só está embutida no código fonte (e.g.: operacionalizações de consciência, valores de referência), ou permanece sob a forma de conhecimento tácito dos profissionais que lidam com o *software*. Nessa situação, esses conhecimentos, que são de grande valia para a evolução do *software*, não estão refletidos nos artefatos de definição do produto (e.g.: requisitos e arquitetura), por não terem sido criados propriamente ou por estarem defasados.

A situação é agravada se requisitos não funcionais (RNFs) não forem propriamente identificados e, devido a isso, a arquitetura pode não refletir a abstração apropriada, tornando as operacionalizações dos RNFs difíceis de serem reutilizadas e evoluídas. As decisões arquiteturais embutidas no código fonte podem incluir as abstrações que foram selecionadas para separar, combinar e encapsular conceitos de vários tipos nas descrições arquiteturais (KANDÉ, 2003), como o próprio requisito de consciência. Estes conceitos atravessam as fronteiras dos módulos arquiteturais e geralmente derivam a partir de RNFs como propriedades de qualidade (MAHMOUD, 2015). Rozanski e Woods (ROZANSKI; WOODS, 2011) definem propriedades de qualidade como algo visível externamente, ou propriedade não funcional de um sistema

[3] O modelo proposto pela IBM é denominado MAPE-K, mas é comumente conhecido por MAPE. O K representa um repositório de conhecimento ("***K****nowledge*") que é utilizado pelas quatro funções do MAPE (*Monitor–Analyze–Plan–Execute*). Deste modo, subentende-se que o K é inerente ao MAPE.

como performance, segurança, ou escalabilidade. Complementando esta visão de requisitos não funcionais e sua influência na arquitetura de *software*, Chung e Leite (CHUNG; DO PRADO LEITE, 2009) estimam que os RNFs sirvam como um fator crítico durante o desenvolvimento do sistema, servindo como critério de seleção para escolher entre alternativas de projetos e soluções de implementação.

Diante do contexto exposto, como reestruturar sistemas autoadaptativos que foram construídos aquém das recomendações da literatura ou aqueles que se tornaram obsoletos devido a mudanças das necessidades das partes interessadas?

Chikofsky & Cross (CHIKOFSKY; CROSS, 1990) definem a reengenharia como "o exame e alteração de um Sistema alvo para reconstituí-lo em uma nova forma e a sequente implementação desta nova forma". Complementando esta visão, Leite (LEITE, 1996) acredita que a reengenharia é uma prática apropriada para estas situações em que a evolução é independente do processo de construção do *software*.

1.2. Motivação

O RNF de consciência é um tipo específico de requisito relacionado a ter conhecimento sobre algo (CUNHA, 2014) e este tipo de requisito é a base de softwares autoadaptativos. Uma vez que RNFs de consciência sejam identificados, é importante escolher as abstrações adequadas para a arquitetura do *software* autoadaptativo. Como padrão de arquitetura e implementação para sistemas autoadaptativos, a literatura recomenda soluções com base em laços de controle (*feedback loop*). Na literatura, destaca-se o desenho arquitetural MAPE proposto pela IBM (IBM, 2006), no qual as responsabilidades para o monitoramento e adaptação estão bem divididas e organizadas. Porém, este não é um padrão trivial de ser utilizado (ABUSETA; SWESI, 2015) e acreditamos que o seu uso pode ser guiado desde os requisitos, se o RNF de consciência de *software* for tratado como um requisito de primeira classe.

No contexto de requisitos, a engenharia de requisitos orientada a metas ou GORE (*Goal Oriented Requirements Engineering*) tem sido extensivamente pesquisada e os resultados indicam diversas vantagens sobre outras perspectivas para a modelagem de requisitos em sistemas

autoadaptativos (ALI; DALPIAZ; GIORGINI, 2010; CUNHA, 2014; FUXMAN et al., 2004; SOUZA; MYLOPOULOS, 2012). Entretanto, apesar das recomendações encontradas na literatura, é possível identificar vários sistemas, em repositórios de código, como o GitHub, ou na indústria, que foram construídos de maneira *ad-hoc* ou com problemas na implementação do próprio MAPE.

Serikawa et al (SERIKAWA et al., 2016) identificaram dois padrões de problemas na implementação da função monitor do MAPE:

— **Monitores Oprimidos**: este problema é caracterizado pela existência de pelo menos um monitor sendo responsável pela execução de dois ou mais monitores, fazendo com que os monitores tenham a mesma frequência e uma ordem específica de execução e.
— **Monitor Obscuro**: este problema é caracterizado pelo fato de a lógica de monitoramento, transmissão e pré-processamento estarem dispersas no código sem que o monitor esteja implementado como uma entidade de primeira classe.

San Martín et al. (SAN MARTÍN et al., 2020) identificaram mais três diferentes problemas na implementação do MAPE:

— **Entradas de Referência Dispersas**: este problema é caracterizado quando os valores de referência utilizados para saber quais os estados do sistema que devem ser alcançados e/ou mantidos estão dispersos pelo sistema sem uma abstração apropriada para eles.
— **Executores e Atuadores Mistos**: este problema é caracterizado pela falta de uma clara distinção entre executores e atuadores.
— **Alternativas Obscuras**: este problema é caracterizado pela ausência de abstração apropriada para armazenar as alternativas de adaptação, tornando-as não evidentes no código.

O uso da engenharia de requisitos orientada a metas é normalmente empregado em atividades de engenharia avante, ou seja, na definição de novos componentes num sistema de *software* (BARESI; PASQUALE; SPOLETINI, 2010; CUNHA, 2014; SOUZA; MYLOPOULOS, 2012). Uma vez que o sistema autoadaptativo tenha sido construído com base em outras perspectivas, como resgatar sua intencionalidade e seus RNFs de consciência operacionalizados para guiar a sua evolução a partir de seus requisitos?

Identificamos, na literatura, o trabalho de Yu et al. (YU et al., 2005) em que os pesquisadores propõem a engenharia reversa de sistemas legados que oferecem algum tipo de serviço (e.g.: "envio de e-mail") para modelos de metas em GSP (*Goals Skills and Preferences*) para fins de reengenharia com vistas a promover o reuso deste mesmo serviço sob diferentes formas (e.g.: web *service* e componentes). Apesar de este trabalho fazer a engenharia reversa de código para modelos de metas, ele não disponibiliza suporte para a identificação de RNFs de consciência, não oferece auxílio para reconstituir o novo sistema no formato MAPE e a rastreabilidade entre o modelo de metas obtido e o código não é explícita.

Ainda que se possa resgatar a intencionalidade do sistema autoadaptativo e seus RNFs de consciência em uma forma mais abstrata, é preciso guiar a reimplementação deste RNF de modo que o produto final tenha uma arquitetura aderente ao padrão MAPE (IBM, 2006), o qual tem sido recomendado fortemente na literatura. Através do uso deste padrão, as responsabilidades e abstrações inerentes a autoadaptação estarão bem definidas e organizadas.

Neste contexto, identificamos o framework *Rainbow* (GARLAN et al., 2004; GARLAN; SCHMERL; CHENG, 2009) que oferece um mecanismo externo de suporte geral para a autoadaptação baseado na arquitetura do sistema gerenciado e possibilita a reengenharia de sistemas não autoadaptativos em sistemas autoadaptáveis. Encontramos também o trabalho de Serikawa et al. (SERIKAWA et al., 2016), que propõe uma estratégia de refatoração de sistemas autoadaptativos com ênfase na reestruturação da função monitor do MAPE, e o trabalho de San Martín et al. (SAN MARTÍN et al., 2020), que recomenda três estratégias de refatoração com ênfase em reestruturar os valores de referência indicando a necessidade de adaptação

Após a reengenharia do sistema autoadaptativo, uma vez adquirida a consistência entre suas metas, sua arquitetura conforme o padrão MAPE e seu novo código, é importante manter esta consistência. A pesquisadora Liliana Pasquale, em sua apresentação (PASQUALE, 2020) sobre o artigo "*Fuzzy goals for requirements-driven adaptation*" (BARESI; PASQUALE; SPOLETINI, 2010) vencedor do prêmio "*Most Influential Paper Award*" no RE'20, ressalta que a manutenção da consistência entre as metas, a arquitetura e a implementação é ainda um problema em aberto.

Por estas razões, acreditamos ser importante e justificável a busca por estratégias que possibilitem a reengenharia de sistemas autoadaptativos, que visem realinhá-los com as práticas recomendadas pela literatura (i.e.: GORE e MAPE) e auxiliem na remoção dos problemas recorrentes de implementação do MAPE. Acreditamos que esta estratégia deve buscar reutilizar o máximo possível da implementação de autoadaptação existente internamente, mas deve abranger a revisão dos RNFs de consciência além das operacionalizações adotadas. Aliado a isto, deve ser possível manter a rastreabilidade entre as metas, a arquitetura e o código com vistas a auxiliar na manutenção da consistência entre estes níveis de abstração.

1.3. Caracterização da Pesquisa

Conforme contextualizamos, há alguns anos o *software* tornou-se onipresente e vários softwares autoadaptáveis e autogerenciáveis foram desenvolvidos. Segundo Cunha (CUNHA, 2014), os sistemas autoadaptativos foram muito impulsionados pelo aumento no número de aplicativos para dispositivos móveis, computação vestível e computação ubíqua, principalmente no que se refere a softwares sensíveis ao contexto. Enquanto os sistemas autogerenciados têm a capacidade de satisfazer seus objetivos na mudança de ambientes, sem necessitar de supervisão humana, e continuar a operação em diferentes condições.

Considerando as exposições realizadas neste capítulo, grandes esforços foram alocados no desenvolvimento de métodos, técnicas, processos, frameworks, entre outras soluções para a criação de tais softwares (ARCAINI; RICCOBENE; SCANDURRA, 2015; BRABERMAN et al., 2015; GARLAN; SCHMERL; CHENG, 2009; GIORGINI; MYLOPOULOS; SEBASTIANI, 2005; IBM, 2006; MISELDINE; TALEB-BENDIAB; RANDLES, 2005; SOUZA; MYLOPOULOS, 2012). Entretanto, pouco se explorou sobre a reestruturação daqueles sistemas autoadaptativos que não seguiram tais boas práticas e que precisam ser reestruturados devido a problemas de implementação do MAPE recorrentes na literatura (SAN MARTÍN et al., 2020; SERIKAWA et al., 2016). Diante do que foi descrito, vislumbramos os seguintes desafios e/ou questões de pesquisas:

— *Como Resgatar a intencionalidade dos sistemas autoadaptativos revelando seus RNFs de consciência já operacionalizados, em um modelo de metas de forma a promover a modularidade dos monitores e sensores do padrão arquitetural MAPE?*
— *Como especificar o RNF de consciência em modelos de metas reutilizando a meta arquitetura MAPE, de tal modo que promova a remoção dos problemas recorrentes de implementação do MAPE encontrados na literatura?*

A partir dos desafios identificados, e utilizando uma pesquisa exploratória[4] (OIVO et al., 2004), elaboramos as seguintes hipóteses:

Hipótese 1*: reutilizar meta conhecimento de consciência de software pode auxiliar na identificação de metas flexíveis e operacionalizações de consciência em sistemas autoadaptativos.*

Para avaliar a hipótese 1, recuperamos um modelo de metas i* do sistema autoadaptativo real, chamado *PhoneAdapter*[5], com as metas flexíveis de consciência operacionalizadas no código fonte e comparamos o nosso resultado com as operacionalizações identificadas em outros trabalhos correlatos sobre este mesmo sistema.

Hipótese 2*: reutilizar a meta arquitetura MAPE de sistemas autonômicos em modelos de metas pode ajudar na reorganização da modularidade de sistemas autoadaptativos, diminuindo os problemas arquiteturais recorrentes na literatura.*

Para avaliar a hipótese 2, reespecificamos o sistema *PhoneAdapter*[5] através do reuso da meta arquitetura MAPE em *SRconstructs* [6]. Comparamos o modelo resultante com a estrutura proposta pela IBM (IBM, 2006) e com os resultados das refatorações correlatas que identificamos na literatura (SAN MARTÍN et al., 2020; SERIKAWA et al., 2016).

[4] Métodos exploratórios são usados principalmente para gerar ideias e hipóteses e menos para controlar as suposições. (OIVO et al., 2004)
[5] Escolhemos o sistema *PhoneAdapter* por ele possuir os problemas de implementação do MAPE, segundo a análise de Serikawa et al. (SERIKAWA et al., 2016) e San Martín et al. (SAN MARTÍN et al., 2020). Ele está disponível em https://github.com/Advanse-Lab/*PhoneAdapter*/releases/tag/v1.5.
[6] Um *SRconstruct* é uma parte da estrutura do "*rationale*" de um ator, e modela os elementos necessários para o alcance da meta alvo do próprio construto.

Hipótese 3*: futuras evoluções de sistemas autoadaptativos podem ser facilitadas após a reengenharia ser aplicada.*

Para avaliar a hipótese 3, fizemos um estudo exploratório a fim de analisar o impacto de algumas tarefas de manutenção na versão original do *PhoneAdapter*[5] e na versão pós-reengenharia. Comparamos os nossos resultados com os obtidos por trabalhos correlatos (SAN MARTÍN et al., 2020; SERIKAWA et al., 2016).

1.4. Solução Proposta

Com vistas a responder às questões de pesquisa e/ou desafios apresentados anteriormente e testar nossas hipóteses, propomos uma estratégia de reengenharia de sistemas autoadaptativos guiada pelo requisito não funcional de consciência de *software*. Restringimos o escopo da nossa pesquisa a sistemas autoadaptativos orientados a objetos (OO) para que pudéssemos adotar características específicas do paradigma OO a fim de resgatar a intencionalidade do sistema, como, por exemplo, o mapeamento de classes e interfaces para tipos de atores (i.e.: agentes, posições, papéis ou atores genéricos).

Escolhemos o *framework* i* na versão 1.0 (YU, 1995) como linguagem para o modelo de metas, porque a sua intencionalidade distribuída auxilia a modularização. Além disso, a relação meios-fim é o ponto chave para mapear a variabilidade com a semântica praticada em nossa pesquisa. Este *framework* também é bastante difundido na academia (DALPIAZ; FRANCH; HORKOFF, 2016b) e possui várias extensões catalogadas[7] ((GONÇALVES et al., 2018)) com diferentes finalidades. Não utilizamos a versão 2.0, pois entre as mudanças realizadas, estão a junção dos relacionamentos meios-fim e decomposição de tarefa em um único relacionamento denominado refinamento, cuja semântica difere parcialmente dos relacionamentos substituídos. Também levamos em conta a impossibilidade de representar o meio para alcançar um recurso no "*rationale*" interno do ator exceto quando este é um elemento de dependência entre dois atores. Estas mudanças limitariam a abrangência de nossas heurísticas de mapeamento de código OO para modelo de metas.

[7] http://istarextensions.cin.ufpe.br/catalogue/

Similarmente aos subprocessos de reengenharia sugeridos por Leite (LEITE, 1996), a nossa estratégia está organizada em quatro subprocessos principais, a saber:

— **Recuperar**: no primeiro subprocesso da estratégia, o nosso objetivo é resgatar a intencionalidade do sistema. Assim, obtém-se um modelo de metas i* *AS-IS* a partir do código fonte da aplicação alvo da reengenharia com os RNFs de consciência de *software* que já estão operacionalizados (i.e.: implementados) no código. Com o intuito de alcançar este objetivo, utilizamos um conjunto de heurísticas de mapeamento dos elementos estáticos presentes na programação orientada a objetos (POO) para elementos do *framework* i* (YU, 1995). A fim de auxiliar a localizar operacionalizações de consciência no código, utilizamos o catálogo de consciência de *software* (CUNHA, 2014) com possíveis operacionalizações em um formato compreensível por humanos e máquinas. Além disso, criamos um conjunto de meta-informações de código que pode ser utilizado para indicar operacionalizações de consciência que não foram resgatadas com o auxílio do catálogo assim como elementos (e.g.: tarefas, recursos, metas e outros) não identificados através das heurísticas.
— **Especificar**: no segundo subprocesso da estratégia, o nosso objetivo é evoluir o modelo de metas i* *AS-IS* para um modelo i* *TO-BE* em direção ao padrão arquitetural MAPE. Para isto, especificamos o RNF de consciência a partir do modelo de metas i* *AS-IS* através do reuso de construtos de razão estratégia (i.e.: *SRconstructs*). Nossos *SRconstructs* (MOURA et al., 2019) estendem o *SRconstruct* proposto por Cunha (CUNHA, 2014) para contemplar os elementos do padrão arquitetural MAPE. Além disso, neste subprocesso é possível adicionar novos requisitos de consciência e já especificá-los com o uso dos *SRconstructs* propostos.
— **Redesenhar**: no terceiro subprocesso da estratégia, os nossos objetivos são: identificar as tecnologias para implementar as características de cada elemento do MAPE de acordo com a linguagem de programação que será utilizada; revisar as operacionalizações dos RNFs de consciência pré-existentes e selecionar as operacionalizações daqueles que tenham sido adicionados.
— **Reimplementar**: o nosso objetivo, no quarto e último subprocesso da estratégia, é reimplementar a aplicação conforme a reorganização da estrutura especificada no modelo i* *TO-BE*. Assim sendo, deverão ser utilizadas as técnicas e as operacionalizações selecionadas

no subprocesso anterior e utilizar as *guidelines* e *templates* sugeridos. Ao longo da reimplementação, também devem ser adicionadas metainformações de código no ensejo de apontar as tarefas referentes aos elementos do padrão MAPE e para indicar as operacionalizações dos RNFs de consciência.

1.5. Organização da Tese

Partindo desta Introdução, esta tese está organizada em mais 5 capítulos, da seguinte forma:

No **Capítulo 2**, é feita uma revisão da literatura acerca dos conceitos necessários para compreender a estratégia apresentada nesta tese, e que serviram como fonte de inspiração e alicerces para esta pesquisa. São abordados temas essenciais como o *framework* i*, o RNF de consciência de *software*, o padrão arquitetural MAPE e reengenharia de *software*.

No **Capítulo 3**, são apresentadas as técnicas criadas ou adaptadas pela autora.

No **Capítulo 4**, é apresentada a nossa estratégia de reengenharia de sistemas Autoadaptativos guiada pelo RNF de consciência de *software*, na qual descrevemos os seus quatro subprocessos, a saber: recuperar, especificar, redesenhar e reimplementar.

No **Capítulo 5,** são descritos os estudos conduzidos para avaliar as hipóteses desta pesquisa.

Finalmente, no **Capítulo 6**, são descritas as contribuições alcançadas nesta tese em função das questões de pesquisa estabelecidas, limitações identificadas em cada subprocesso da nossa estratégia, além de perspectivas para trabalhos futuros.

2
Conceitos

Neste capítulo, apresentamos uma revisão dos conceitos e abordagens envolvidos na estratégia de reengenharia desenvolvida nesta pesquisa. De modo geral, apresentamos uma visão do *framework* i* 1.0 (YU, 1995) e do *framework* iStar 2.0 (DALPIAZ; FRANCH; HORKOFF, 2016a) e justificamos a nossa escolha pelo *framework* i* 1.0 como linguagem de modelagem orientada a metas para esta pesquisa. Discorremos brevemente sobre as estruturas canônicas do *framework* i*, propostas por Oliveira (OLIVEIRA; LEITE; CYSNEIROS, 2008), as quais são utilizadas nesta pesquisa para particionar os modelos i* e torná-los mais amenos à análise e evolução. Abordamos também o RNF de consciência de *software* e o catálogo proposto por Cunha (CUNHA, 2014), pois o seu conhecimento é reutilizado ao longo dos subprocessos da nossa estratégia. Mostramos uma revisão do padrão arquitetural MAPE, o qual é utilizado para especificar o RNF de consciência de *software*, assim como os problemas na implementação do MAPE que são recorrentes na literatura. Por fim, apresentamos os fundamentos da reengenharia de sistemas de um modo geral e os trabalhos correlatos.

2.1.
Framework i*

Em nossa estratégia, propomos o realinhamento de sistemas autoadaptativos com a engenharia de requisitos orientada a metas ou GORE (*Goal Oriented Requirements Engineering*) pois esta é fortemente recomendada pela literatura para tais sistemas, como contextualizado anteriormente. Ao longo dos anos, foram criados alguns frameworks ou linguagens para representar modelos de metas, dos quais destacamos o framework i*(YU, 1995), KAOS (DARDENNE; VAN LAMSWEERDE; FICKAS, 1993), Tropos (BRESCIANI et al., 2004) por estes terem sido utilizados para a modelagem de sistemas autoadaptativos em alguns trabalhos citados nesta tese e por terem sido analisados quanto a frequência de uso no

estudo feito por Horkoff et al. (HORKOFF et al., 2016). Dentre estes, escolhemos o *framework* i* (YU, 1995) para a modelagem dos requisitos.

Conforme estudo publicado em 2016 (HORKOFF et al., 2016), o *framework* i* é bem difundido na literatura quando comparado a outros frameworks (ver Figura 1), e tem sido utilizado para a modelagem de sistemas autoadaptativos ou conscientes (ALI; DALPIAZ; GIORGINI, 2010; BRESCIANI et al., 2004; CUNHA, 2014; SOUZA; MYLOPOULOS, 2012). Em i*, é possível representar tanto requisitos funcionais (i.e.: metas) quanto requisitos não funcionais (i.e.: metas flexíveis) como elementos de primeira ordem. A modelagem em i* mostra as alternativas para a satisfação das metas e possibilita a satisfação a contento das metas flexíveis pois as utiliza na seleção das alternativas. Estas são as razões pelas quais escolhemos o i*. Entretanto, o i* possui as versões 1.0 e 2.0 e, a seguir, apresentaremos as duas versões e justificaremos a nossa escolha pela versão 1.0 nesta pesquisa.

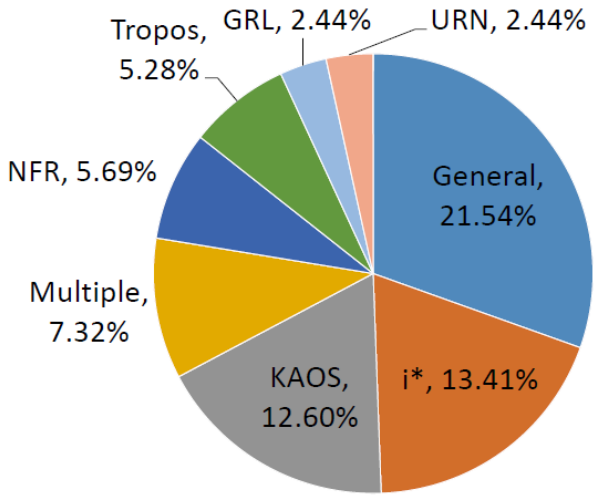


Figura 1 Frameworks utilizados em 246 publicações (HORKOFF et al., 2016).

2.1.1. *Framework* i* 1.0

O *framework* i* 1.0 (YU, 1995) 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 meios-fim.

Modelo SD

O modelo SD provê uma descrição intencional de um processo em termos de uma rede de relacionamentos de dependência entre atores.

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 ator social dentro de um contexto especializado ou domínio. Uma Posição é uma abstração intermediária entre um papel e um agente.

Os atores possuem relacionamentos específicos entre si (ver Figura 2): um agente ocupa uma posição (*occupies*); mas também pode desempenhar um papel (*plays*); enquanto uma posição cobre (*covers*) vários papéis, e; 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*), conforme Figura 2, 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 e cada um será explicado no modelo SR.

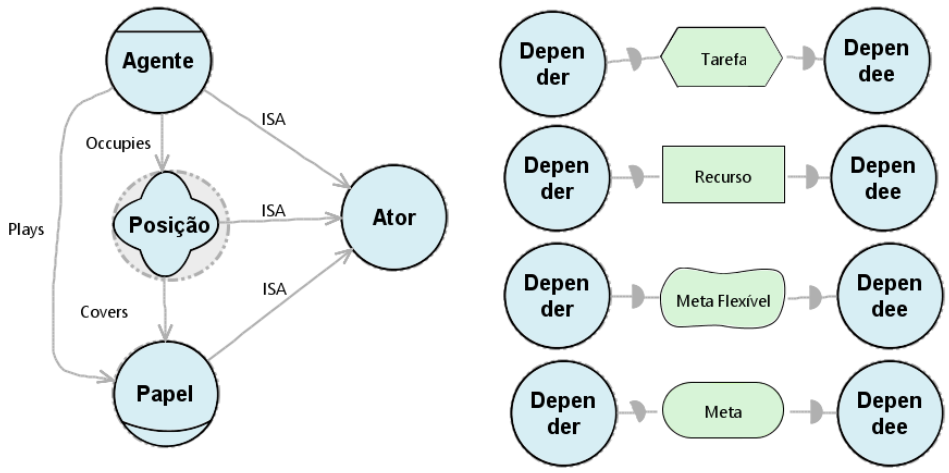


Figura 2 Elementos do modelo SD.

Em uma dependência por meta, o *depender* depende do *dependee* para que certo estado do mundo seja alcançado. Ao *dependee* é dada à liberdade de escolher como fazê-lo. Com uma dependência por meta, o *depender* ganha a habilidade de assumir que a condição ou estado do mundo será alcançado, mas torna-se vulnerável pois o *dependee* pode falhar em realizar tal condição.

Em uma dependência por tarefa, o *depender* depende do *dependee* para executar uma atividade. Uma dependência por tarefa especifica como a tarefa deverá ser realizada, mas não o porquê. O *depender* é vulnerável pois o *dependee* pode falhar em executar a tarefa.

Em uma dependência por recurso, um ator (o *depender*) depende do outro (o *dependee*) para disponibilizar uma entidade (física ou informacional). Ao estabelecer esta dependência, o *depender* ganha a habilidade de usar esta entidade como um recurso ao mesmo tempo que se torna vulnerável se a entidade estiver indisponível.

Em uma dependência por meta flexível, um *depender* depende do *dependee* para executar alguma tarefa que atenda uma meta flexível. O significado de meta flexível é especificado em termos de métodos que são escolhidos no curso de perseguir uma meta. Como em uma dependência por meta, um *depender* ganha a habilidade de assumir que a condição ou estado do mundo será alcançado, mas torna-se vulnerável, pois o *dependee* pode falhar em realizar tal condição. A diferença aqui é que as condições a serem atingidas são elaboradas à medida que a tarefa é desempenhada.

O modelo também faz distinção entre vários graus de dependência. No lado do *depender*, uma dependência forte significa que o *depender* é mais vulnerável e provavelmente tomará fortes medidas para mitigar a vulnerabilidade. No lado do *dependee*, uma forte dependência implica que o *dependee* fará um grande esforço para tentar entregar o *dependum*. O modelo provê três graus de força a saber: *Open* (*uncommitted*), *Committed* e *Critical*. Estes se aplicam independentemente em cada lado da dependência. Graficamente, usa-se O para *open*, sem marcação para *committed* e X para *critical*.

Modelo SR

O modelo SR provê uma descrição intencional dos processos em termos dos elementos do processo e do raciocínio por trás deles.

Os elementos do modelo SR do *framework* i* são (ver Figura 3): a Meta (*Goal*) que é uma condição ou estado de algo no mundo que o ator gostaria de alcançar; a Meta-Flexível (*Softgoal*) que 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*) que 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*) que 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 modelo SR também possui relacionamentos entre os elementos intencionais a saber (ver Figura 3): relacionamento Meios-Fim (*Means-End*) que 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 estes são alternativas e deverá ser seguida ou uma ou outra alternativa. 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.

Quando o relacionamento meios-fim envolve uma meta flexível, é indicado o tipo de contribuição (*Contribution*) para informar se o meio leva a uma contribuição dos seguintes tipos: *MAKE* representa a crença de que o meio em particular provê evidências suficientes para a satisfação do fim; *BREAK* é utilizado quando se acredita que o meio irá impedir ("quebrar") a satisfação do fim; HELP ("+") nos casos positivos; HURT ("-") para casos negativos; SOME+ para alguma contribuição positiva; enquanto SOME- representa uma contribuição negativa.

O relacionamento de Decomposição de tarefa (*Task-Decomposition*) é usado a partir de uma tarefa para indicar a decomposição da tarefa em submeta, subtarefa, recurso para, e meta flexível para.

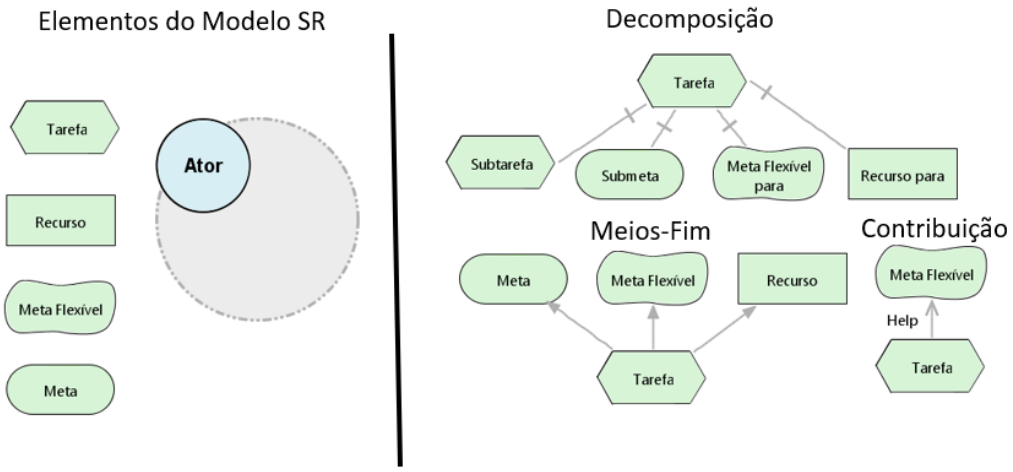


Figura 3 Elementos do modelo SR.

2.1.2. *Framework* iStar 2.0

Conforme o guia da linguagem iStar 2.0, (DALPIAZ; FRANCH; HORKOFF, 2016a, 2016b), o iStar 2.0 nasceu como resposta à necessidade de equilibrar a natureza aberta do *framework* i* 1.0 e uma possível solução para os problemas de adoção para além da comunidade de especialistas. A natureza intencionalmente aberta do *framework* i* 1.0 levou a criação de múltiplas propostas de extensões do i*.

Podemos conhecer várias delas no catálogo de extensões do i* (GONÇALVES et al., 2018). O uso flexível de i* foi proveitosamente empregado por pesquisadores, no entanto este uso também levou à algumas desvantagens e a mais crítica

delas é a dificuldade de difundir o *framework* para além da academia devido a: os novos praticantes encontram dificuldade de aprendizado; os educadores não têm um corpo de conhecimento compartilhado para ensinar; os profissionais não recebem uma referência estabelecida para o uso de i* no seus projetos, e; os fornecedores de tecnologia não podem determinar facilmente quais são as principais construções e as técnicas a serem aplicadas sobre essas construções.

Deste modo, a comunidade de pesquisadores em i* começou uma iniciativa para identificar um conjunto de conceitos amplamente aceitos na linguagem i*. O principal objetivo é manter aberta a capacidade de adaptar o *framework* ao mesmo tempo em que definem as construções fundamentais. Para claramente distinguir do seu predecessor, *framework* i* 1.0 (YU, 1995), este novo núcleo da linguagem recebeu o nome iStar 2.0 (DALPIAZ; FRANCH; HORKOFF, 2016a).

Atores continuam sendo o centro da natureza de modelagem social da linguagem. Atores são ativos, entidades autônomas que desejam alcançar suas metas através do exercício do seu conhecimento, em colaboração com outros atores. Na linguagem iStar 2.0, dois tipos de atores são distinguidos: Papel (*Role*) é uma caracterização abstrata do comportamento de um ator social dentro de um contexto específico ou domínio. Agente (Agent) é um ator com manifestações físicas e concretas, como um indivíduo humano, uma organização, ou um departamento. A representação gráfica de tais atores segue o *framework* i* 1.0, conforme mostra a Figura 4. A intencionalidade de cada ator também continua sendo explícita através da fronteira de ator (ver Figura 5), onde são localizados seus elementos intencionais junto a seus inter-relacionamentos.

No iStar 2.0, os atores podem ser interrelacionados por dois tipos diferentes de relacionamentos: é-um (*is-a*) e participa-em (*participates-in*). O relacionamento é-um (*is-a*) representa o conceito de generalização/especialização. Somente papéis podem ser especializados em papéis, ou atores genéricos em atores genéricos.

Agentes não podem ser especializados por serem definidos como elementos concretos. O relacionamento participa-em (*participates-in*) representa qualquer tipo de associação, que não seja generalização/especialização entre dois atores. Não existe restrições quanto ao tipo de ator que participa neste relacionamento, mas,

dependendo dos atores relacionados, o relacionamento pode ter diferentes significados. Existem duas situações típicas a saber: quando se relaciona um agente a um papel com este tipo de relacionamento, indica que um agente desempenha um papel; enquanto ao relacionar atores do mesmo tipo indica um relacionamento parte-de.

Figura 4 Exemplos de ator, papel e agente no *framework* iStar 2.0. (DALPIAZ; FRANCH; HORKOFF, 2016a).

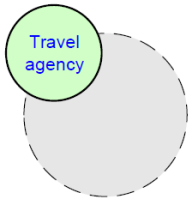


Figura 5 Exemplo de fronteira de ator no *framework* iStar 2.0 (DALPIAZ; FRANCH; HORKOFF, 2016a).

Os elementos intencionais do iStar 2.0 são algo que os atores desejam. A linguagem possui os elementos descritos a seguir. A Figura 6 mostra a representação gráfica destes elementos.

— Meta é uma condição ou estados de desejos no mundo que o ator deseja alcançar e que tenha uma visão clara dos critérios de realização.
— Qualidade é um atributo para o qual um ator deseja algum nível de realização. A qualidade pode guiar a seleção de alternativas para o alcance das metas.
— Tarefa representa ações que um ator deseja que sejam executadas, geralmente com o propósito de alcançar uma meta.
— Recurso é uma entidade física ou informacional que o ator requer para executar uma tarefa.

Figura 6 Exemplos de meta, qualidade, tarefa e recurso (DALPIAZ; FRANCH; HORKOFF, 2016a).

As dependências representam relacionamentos sociais e são definidas com cinco argumentos a saber: o ator (*depender*) que depende de algo (*dependum*) a ser

provido; o elemento intencional (*dependerElmt*) dentro da fronteira do ator de onde a dependência começa, que explica a razão pela qual a dependência existe; o elemento intencional (*dependum*) que é o objeto da dependência; o ator (*dependee*) que deverá prover o *dependum*, e; o elemento intencional (*dependeeElmt*) que explica como o *dependee* pretende prover o *dependum*. A apresenta um exemplo de dependência, onde o papel Estudante depende do ator Agência de Viagem para que a meta "pacote de viagem reservado" seja alcançada. O ator "Agência de Viagem" tem a intenção de alcançar esta meta através da execução da tarefa "reservar pacote via expedia".

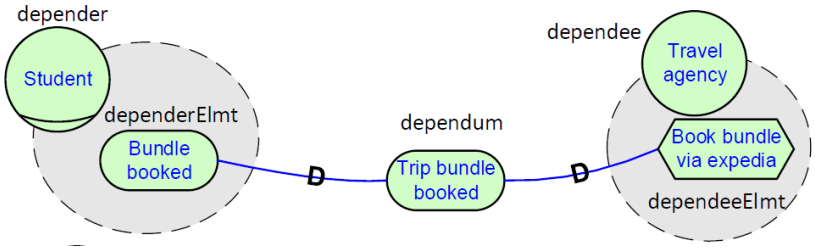


Figura 7 Exemplo de dependência (DALPIAZ; FRANCH; HORKOFF, 2016a).

O tipo do *dependum* configura a semântica da dependência, onde:

— Meta é esperado que o *dependee* alcance a meta, e ele é livre para escolher como;
— Qualidade é esperado que o *dependee* satisfaça a contento a qualidade, e ele é livre para escolher como;
— Tarefa é esperado que o *dependee* execute a tarefa do modo prescrito;
— Recurso é esperado que o *dependee* torne o recurso disponível para o *depender*.

Existem quatro tipos de relacionamentos entre elementos intencionais: refinamento (*refinement*), necessário-para (*needed-by*), contribuição (*contribution*) e qualificação (*qualification*).

O relacionamento refinamento é genérico e pode ligar metas e tarefas hierarquicamente. Este relacionamento pode ser de dois tipos: E ou OU inclusivo, mas não podem ser usados ambos a partir de uma mesma meta ou tarefa. No relacionamento de refinamento E, filhos devem ser alcançados (meta) ou executados (tarefa) para que o pai seja alcançado. Enquanto no relacionamento de refinamento OU inclusivo, pelo menos um dos filhos deve ser alcançado (meta) ou executado (tarefa)

para que o pai seja alcançado. Como este é um relacionamento genérico, é preciso compreender a sua semântica que varia de acordo com os elementos relacionados.

Se o pai é uma meta:

— No refinamento do tipo E, uma meta filha é um sub estado do mundo que é parte da meta pai. Enquanto uma tarefa filha é uma subtarefa que deve ser completada.
— No refinamento do tipo OU inclusivo, uma tarefa filha é um modo particular (um "meio") para alcançar a meta pai (o "fim"). Enquanto uma meta filha é uma submeta que pode ser alcançada para completar a meta pai.

Se o pai é uma tarefa:

— No refinamento do tipo E, uma tarefa filha é uma subtarefa que é identificada como parte da tarefa pai, enquanto uma meta filha é uma meta descoberta analisando a tarefa pai.
— No refinamento do tipo OU inclusivo, uma meta filha é uma meta cuja existência é descoberta analisando a tarefa pai que pode substituir a tarefa original. Enquanto uma tarefa filha é um meio de executar a tarefa pai.

A Figura 8 mostra exemplos dos elos de refinamento, onde o relacionamento E é representado por um T e o relacionamento OU é representado por uma seta. No primeiro exemplo, A meta "autorização obtida" é alcançada quando a as metas "Requisição preparada" e "autorização assinada" são alcançadas. No segundo exemplo, a meta "Viagem reservada" é alcançada através do alcance da meta "Partes da viagem reservada" ou da execução da tarefa "Reservar pacote". No terceiro exemplo, a meta "Tickets reservados" é alcançada através da execução da tarefa "Agência compra tickets". Por fim, no quarto exemplo, a tarefa "processar formulário" é completada quando a meta "Detalhes validados" é alcançada e as tarefas "Requisitar autorização" e "notificar requisitante" são executadas.

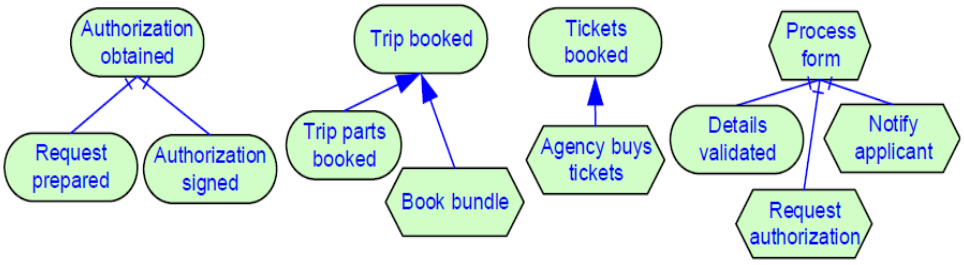


Figura 8 Exemplos do relacionamento refinamento (DALPIAZ; FRANCH; HORKOFF, 2016a).

O relacionamento necessário-para liga uma tarefa a um recurso, indicando que o recurso é necessário para a execução da tarefa. No exemplo da é mostrado que o recuso "cartão de crédito" é necessário para executar a tarefa "Pagar pelos tickets".

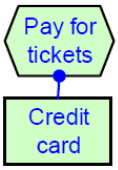


Figura 9 Exemplo de relacionamento necessário-para (DALPIAZ; FRANCH; HORKOFF, 2016a).

O relacionamento contribuição representa os efeitos dos elementos intencionais sobre uma qualidade. Ele auxilia o processo de tomada de decisão entre alternativas de metas ou tarefas. A contribuição é definida a partir de um elemento intencional para uma qualidade e pode dos seguintes tipos:

— *Make*: quando o elemento intencional provê evidência suficiente para a satisfação da qualidade.
— *Help*: quando o elemento intencional provê pouca evidência para a satisfação da qualidade.
— *Hurt*: quando o elemento intencional provê pouca evidência para a não satisfação da qualidade.
— *Break*: quando o elemento intencional provê evidência suficiente para a não satisfação da qualidade.

O relacionamento de qualificação relaciona uma qualidade ao seu assunto (i.e.: tarefa, meta ou recurso). Isto é, o relacionamento de qualificação expressa uma qualidade desejada sobre a execução de uma tarefa, ou sobre o alcance de uma meta, ou sobre um recurso. Por exemplo, dizer que a meta flexível "sem erro" qualifica a

meta “Requisição preparada” significa dizer que o preparo da requisição deve ser alcançado de modo que não haja erro.

No iStar 2.0 existe o conceito de visão de modelos e não tipos de modelos. Deste modo, o modelo é único, mas pode ser visto de diferentes maneiras. A linguagem propõe inicialmente três visões a parir do modelo: a visão SR (*Strategic Rationale*) onde os elementos são vistos em detalhe como no *framework* i* 1.0; a visão SD (*Strategic Dependency*) onde somente os atores e suas dependências são visualizadas também como no *framework* i* 1.0, e; a visão híbrida onde as visões anteriores são combinadas e é possível mesclar a visualização detalhada de alguns atores com ênfase no raciocínio estratégico de um conjunto particular de atores.

2.1.3. Comparação Entre As Versões 1.0 E 2.0

No sitio iStar *Language*[8], é apresentada uma breve comparação entre os dois frameworks (ver Tabela 1). De modo geral, foram realizadas as seguintes modificações: a especialização de ator “Posição” foi excluída; houve a junção dos relacionamentos é-parte-de, desempenha, ocupa e cobre em um único relacionamento denominado participa-em; a meta flexível passou a ser denominada qualidade; houve também a junção dos relacionamentos meios-fim (*means-end*) e decomposição-tarefa (*task-decomposition*) em um único relacionamento genérico denominado refinamento (*refinement*), e; por fim, foram criados dois novos relacionamentos entre elementos intencionais, qualificação e necessário-para.

É importante ressaltar que a simplificação dos relacionamentos em número levou a uma variação na semântica dos relacionamentos substitutos que é determinada de acordo com os elementos relacionados.

O relacionamento necessário-para indica que um recurso é necessário para a execução de uma tarefa, substituindo a decomposição de tarefa em recurso ainda que não tenha sido descrito como tal no guia iStar 2.0. Devido à extinção do relacionamento meios-fim, não é possível indicar meios para disponibilizar um recurso no “*rationale*” do ator. Somente é possível indicar o provimento de um recurso (*dependum*) através da dependência de recurso entre atores, onde o *dependeeElmt* explica como o *dependee* pretende prover o *dependum*.

[8] https://sites.google.com/site/istarlanguage/diff?authuser=0

Tabela 1 Sumário das diferenças entre i* 1.0 e iStar 2.0 (iStar *Language*[8]).

	i* 1.0	iStar 2.0
Atores	Atores genéricos	Atores genéricos
	Papéis, posições, agentes	Papéis, agentes
Relacionamentos entre atores	é-um	é-um
	é-parte-de, desempenha, ocupa, cobre	participa-em
	Instancia	-
Elementos intencionais	meta, tarefa, recurso	meta, tarefa, recurso
	Meta-flexível	qualidade
Relacionamentos entre elementos intencionais	Meios-fim, decomposição-tarefa	refinamento
	Contribuição	contribuição
		qualificação, necessário-para

Como mencionado anteriormente, em nossa pesquisa, escolhemos utilizar o *framework* i* 1.0 (YU, 1995). A sua intencionalidade distribuída auxilia a modularização, e a relação meios-fim é o ponto chave para mapear a variabilidade com a semântica que adotamos nesta pesquisa. Não utilizamos a versão 2.0 pelos seguintes motivos:

— Apesar de ser possível refinar tarefas em metas no framework iStar 2.0, nossa estratégia utiliza a decomposição de tarefa em meta do *framework* i* 1.0, a qual possui uma semântica diferente do relacionamento refinamento do iStar 2.0. Na decomposição de tarefa em meta, a meta é uma submeta da tarefa que não é executada pela tarefa propriamente, mostrando um ponto de variabilidade que pode ser alcançado através de diferentes meios (i.e.: alternativas). Como no exemplo da Figura 10, o método (“tarefa”) updateUserLocation chama o método markUserPosition da classe (“Agente”) MapMarker, pois é preciso alcançar a meta “Que a posição do usuário seja marcada” para que a tarefa “atualizar a posição do usuário seja realizada”. Se utilizássemos o relacionamento refinamento do iStar 2.0, significaria que a meta, refinada a partir da tarefa, não é coberta pela tarefa segundo a definição do guia iStar (DALPIAZ; FRANCH; HORKOFF, 2016a);

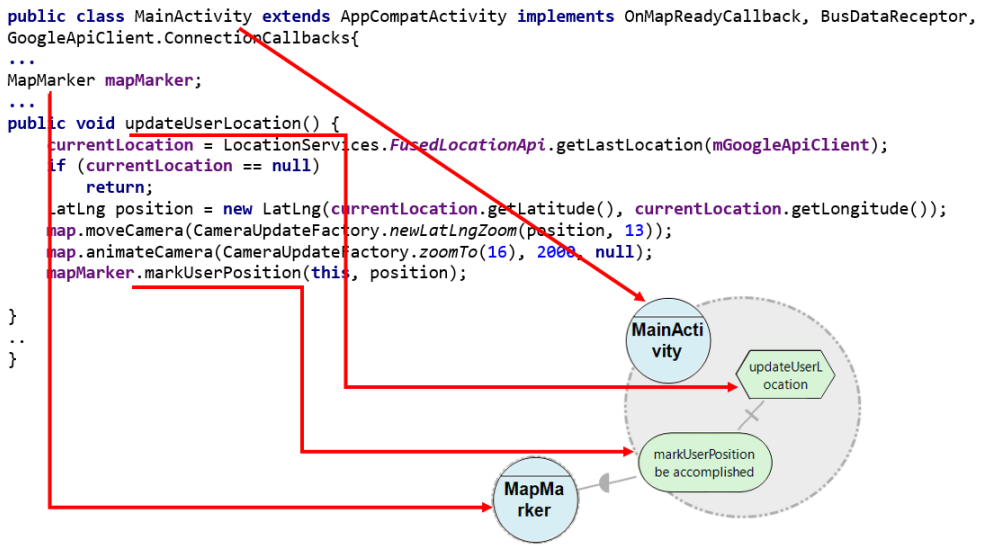


Figura 10 Exemplo de mapeamento de decomposição de tarefa em submeta.

— Devido à extinção do relacionamento meios-fim e a consequente impossibilidade de representar meios para disponibilizar um recurso no interior do ator, como no exemplo de mapeamento da Figura 11, onde o método *onHandleIntent* é o método (“meio”) para alcançar o atributo (“recurso”) *mTime* na classe (“Agente”) *ContextManager*, a nossa estratégia utiliza o relacionamento meios-fim do *framework* i* 1.0. Se utilizássemos a versão iStar 2.0, teria que existir uma dependência deste recurso entre atores para conseguirmos especificar como o recurso é alcançado;

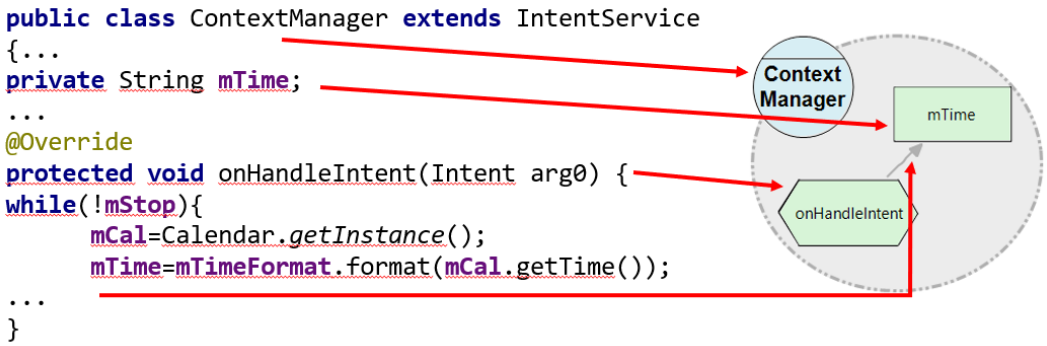


Figura 11 Exemplo de mapeamento de relacionamento meios-fim do tipo tarefa-recurso.

— Em nossa pesquisa, vemos a meta como um estado único e não divisível, em submetas ou subtarefas, a ser alcançado e que sempre é alcançado através de um meio (“tarefa”), então optamos pelo relacionamento meios-fim do *framework* i* 1.0 para mostrar as alternativas (“meios”) para alcançar uma meta.

Em nossa estratégia, resgatamos os meios concretos já implementadas para alcançar uma meta enquanto realinhamos o sistema autoadaptativo a um modelo de metas.

2.1.4. Estruturas Canônicas Do *Framework* i*

Analisar modelos em i* pode ser uma tarefa árdua a *depender* da extensão do modelo (e.g.: número de elementos e elos). Em sua tese, Oliveira (OLIVEIRA, 2008) descreve as estruturas canônicas do *framework* i* presentes na Figura 12, são elas: a situação de dependência estratégica (*SDsituation* (PADUA; OLIVEIRA; CYSNEIROS, 2006)); e o constructo de razão estratégica (*SRconstruct* (OLIVEIRA et al., 2008)) respectivamente. Estas estruturas foram idealizadas a partir da ideia central do *framework* i* que é "representar, através de modelos, os atores e as dependências que os atores têm uns com os outros para que metas próprias sejam atingidas".

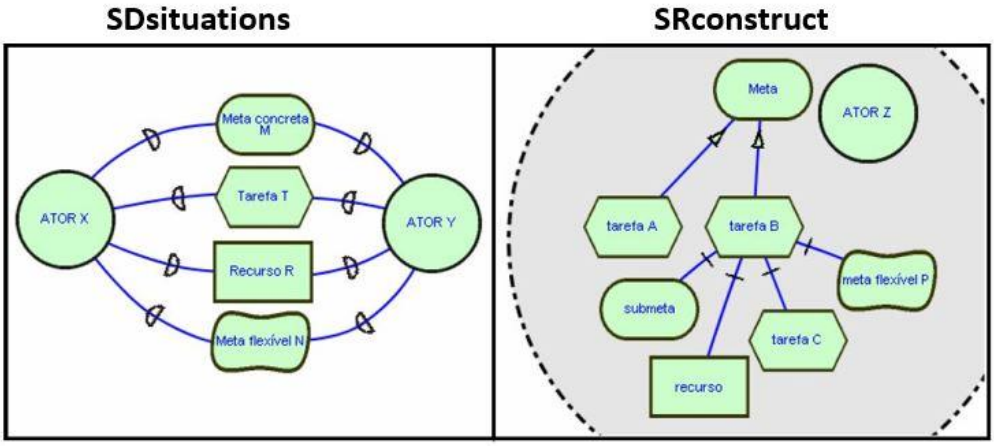


Figura 12 Ilustração das estruturas canônicas do i* (OLIVEIRA, 2008).

Vimos na Seção 2.1.1, que o *framework* i* possui dois modelos: SD e SR. Tais modelos costumam ser grandes e complexos, pois o modelo SD representa os atores e interdependências e o modelo SR detalha o "*rationale*" de cada ator do modelo anterior. Como a nossa estratégia recupera modelos i* a partir de código fonte, o baixo nível de abstração tende a recuperar modelos bem extensos. Para tornar estes modelos mais amenos de serem analisados e evoluídos, optamos por utilizar as estruturas canônicas propostas por Oliveira (OLIVEIRA; LEITE; CYSNEIROS, 2008) para particionar/subdividir os modelos i* em estruturas menores que mantenham algum significado.

Conforme descrito por Oliveira (OLIVEIRA; LEITE; CYSNEIROS, 2008), as estruturas canônicas podem ser compreendidas da seguinte forma:

— ***SDsituation*** é uma estrutura de dependências estratégicas mínimas entre atores de um contexto organizacional, Uma *SDsituation* define um bloco de elementos de dependência com intencionalidade situacional compartilhada. No exemplo da Figura 12, é apresentada uma *SDsituation* entre os atores "Ator x" e "Ator y", onde o Ator x depende do Ator y para alcançar a "meta concreta m" e o "recurso r" e o Ator y depende do Ator x para executar a "tarefa t" e operacionalizar a "meta flexível n". Deste modo, a invés de analisar todo o modelo i*, analisamos partes dele que compõem um contexto organizacional;

— ***SRconstruct*** é uma parte da estrutura do "*rationale*" de um ator que estabelece uma estratégia para a satisfação de uma meta, modelando os elementos necessários para o alcance da meta alvo do próprio construto. No exemplo da Figura 12, é apresentado o "*rationale*" do Ator z para alcançar a "meta" que é o centro deste *SRconstruct*. Esta meta pode ser alcançada através das tarefas "tarefa a" ou "tarefa b". A tarefa b, por sua vez, é decomposta em "submeta", "recurso", "meta flexível p" e "tarefa c".

Para fins de particionamento dos modelos i*, é importante ressaltar que uma *SDsituation* reflete somente uma "Situação de Dependência Estratégica", a qual pode ser formada por uma ou mais dependências (i.e.: de meta, de meta flexível, de tarefa ou de recurso). Uma *SDsituation* pode ser identificada separadamente das outras *SDsituations* formando uma cadeia de interdependência entre estas situações. As interdependências podem ser do tipo física, lógica ou temporal.

— Física: quando um recurso é preparado por uma *SDsituation* e requisitado por outra;

— Lógica: quando uma ou mais *SDsituations* necessitam da conclusão de outra *SDsituation* para a iniciação, ou quando uma ou mais *SDsituations* necessitam da conclusão de outra *SDsituation* para a conclusão;

— Temporal: quando uma ou mais *SDsituations* necessitam esperar algum tempo após o início de outra *SDsituation*, ou quando uma ou mais *SDsituatons* necessitam esperar algum tempo após a conclusão de outra *SDsituation*

Como modelos em i* podem ser complexos para serem analisados, utilizar *SDsituations* pode trazer alguns benefícios (OLIVEIRA et al., 2008):

— Elicitar *SDsituations* antes de modelar é o melhor modo de gerenciar a complexidade ao invés de lidar com muitas dependências ao mesmo tempo. Isto se confirma porque cada *SDsituation* deve ser identificada separadamente, embora o engenheiro de requisitos deva elicitar interdependências estratégicas entre as *SDsituations*.

— Validar requisitos utilizando uma representação possível de ser lida é útil porque os interessados se sentem mais confortáveis com modelos centrados em linguagem natural. A validação pode ser customizada por meio das *SDsituations* aplicando mais de um ponto de vista (pontos de vista dos *dependers* ou *dependees*).

— *SDsituations* podem ajudar a gerenciar os requisitos mantendo a rastreabilidade (para frente e para trás) durante o processo de elicitação e mantendo uma baseline para registrar a evolução dos requisitos.

Estas estruturas canônicas nos permitiram ainda criar padrões para especificar o RNF de consciência de *software* conforme o padrão arquitetural MAPE (IBM, 2006) que será apresentado mais adiante.

2.1.5. Mapeamento De Modelo i* Para POO

Como delimitamos o escopo da nossa pesquisa a sistemas autoadaptativos orientados a objetos (OO), buscamos, na literatura, trabalhos com propostas para integrar modelos de metas (onde são especificados requisitos organizacionais) e modelos OO.

Em um dos trabalhos (CASTRO; ALENCAR; CYSNEIROS, 2000), os autores criaram um conjunto de regras de transformação de modelos i* para modelos OO utilizando pUML (*precise* UML) em conjunto com OCL (*Object Constraint Language*). O conjunto é formado por seis regras gerais de transformação que são listadas na Tabela 2 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*).

Tabela 2 Regras de transformação (CASTRO; ALENCAR; CYSNEIROS, 2000).

Regra	Descrição
G1	i* podem ser mapeados para classes em pUML.
G1.1	Composição de atores corresponde a agregação de classes.
G2	Tarefas são mapeadas para operações de classe.
G2.1	Um relacionamento de dependência de tarefa corresponde a uma operação pública na classe fornecedora.
G2.2	Uma tarefa no modelo SR é mapeada em uma operação local na classe correspondente em pUML.
G3	Recursos em i* são mapeados para classes em pUML e um atributo público em do tipo booleano indica a disponibilidade do recurso.
G4	Metas e metas-flexíveis estratégicas são mapeadas para atributos do tipo booleano ou tipos enumerados respectivamente em classes pUML.
G4.1	Dependências de metas ou metas-flexíveis são mapeadas para atributos públicos do tipo booleano e tipo enumerado respectivamente na classe fornecedora em pUML.
G4.2	Metas e metas-flexíveis são mapeadas respectivamente para atributos locais (privados) do tipo booleano e enumerado na classe pUML.
G5	A decomposição de tarefa é representada por pré e pós condições (expressada em OCL) para a operação correspondente em pUML.
G6	O resultado da análise dos relacionamentos meios-fim é representado por disjunção OCL dos possíveis meios para alcançar o mesmo fim.

Em 2003, este mesmo conjunto de regras foi estendido conforme apresentado na Tabela 3. Para suportá-lo, foi desenvolvida uma versão estendida da ferramenta GOOD, denominada XGOOD. Este conjunto de regras estendido foi publicado em livro no ano de 2011 (CASTRO et al., 2011) 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 (MALTA et al., 2011) 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.

Outro trabalho que gera código OO a partir de modelos i*, é o trabalho de Serrano e Leite (SERRANO; LEITE, 2011a). Neste trabalho, eles propuseram uma estratégia de desenvolvimento de sistemas multiagentes (SMA) a partir de modelos arquiteturais em i* (SERRANO; LEITE, 2011a). Nesta abordagem, os autores propõem heurísticas de projeto e implementação para guiar o desenvolvimento de SMA dos requisitos ao código centrado em agentes intencionais. Os autores usam

o *framework* i* para modelar os requisitos e o projeto e, uma vez que o sistema esteja modelado, a implementação deve ser feita com base no modelo BDI (*Belief – Desire - Intention*) do JADEX (BRAUBACH; LAMERSDORF; POKAHR, 2003), um add-on para a plataforma JADE (BELLIFEMINE et al., 2002) de SMA.

Tabela 3 Regras de transformação estendidas (ALENCAR et al., 2003).

Regra	i*	pUML
G1.1	Agentes, Papéis ou posições	Classes
G1.2	Relacionamento e-parte-de	Agregação
G1.3	Relacionamento é-um	Generalização / especialização
G1.4	Relacionamento ocupa	Associação nomeada ocupa
G1.5	Relacionamento cobre	Associação nomeada cobre
G1.6	Relacionamento desempenha	Associação nomeada desempenha
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
G4.2	Meta-Flexível no modelo SR	Atributo com visibilidade privada na classe à qual a meta flexível pertence
G5	Relacionamento decomposição de tarefa	Pré e pós condições em OCL na operação
G6.1	Relacionamento Meios-fim de Meta-flexível para Meta-Flexível	A disjunção dos valores dos meios implica no valor do fim
G6.2	Relacionamento meios-fim de tarefa para meta flexível e tarefa para recurso	A pós-condição das tarefas meio implicam no valor do fim
G6.3	Relacionamento meios-fim de tarefa para tarefa	A disjunção da pós-condições dos meios implica nas pós-condições do fim

JADE (Java *Agent DEvelopment Framework*) é um *framework* implementado na linguagem Java. Este *framework* possibilita a implementação de SMA através de um middleware, que é conforme com as especificações FIPA (*The Foundation for Intelligent Physical Agents*), e através de um conjunto de ferramentas gráficas para suporte à depuração de código e implantação

Apesar da existência de diferentes semânticas entre modelos i* e BDI, esses modelos compartilham alguns conceitos comuns. A Figura 13 mostra as associações entre o i* e o BDI que foram utilizadas para criar um conjunto de heurísticas para a transformação de modelos i* SD e SR em especificações BDI.

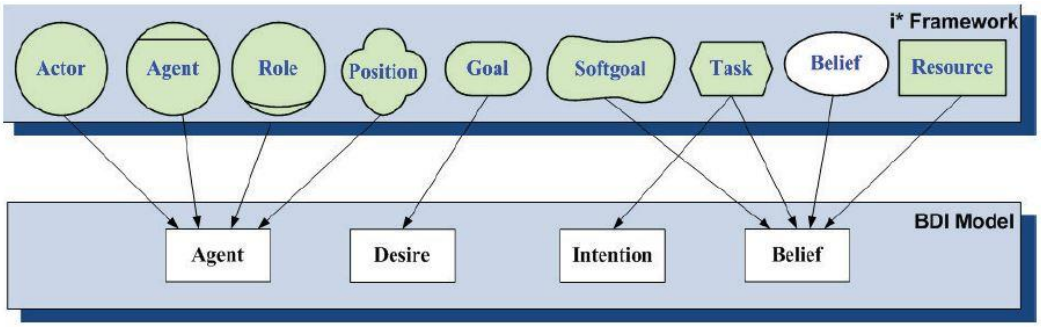


Figura 13 Associação entre abstrações do i* e abstrações BDI (SERRANO; LEITE, 2011a).

O *framework* JADEX implementa uma arquitetura BDI para agentes da plataforma JADE. Por exemplo, intenções definidas na especificação BDI são implementadas como classes Java que estendem a classe "*Plan*" de JADEX. A Figura 14 ilustra as associações entre as abstrações da especificação BDI e o código em JADEX. Estas associações foram utilizadas para criar um conjunto de heurísticas de transformação de uma especificação BDI para códigos de agentes em JADEX.

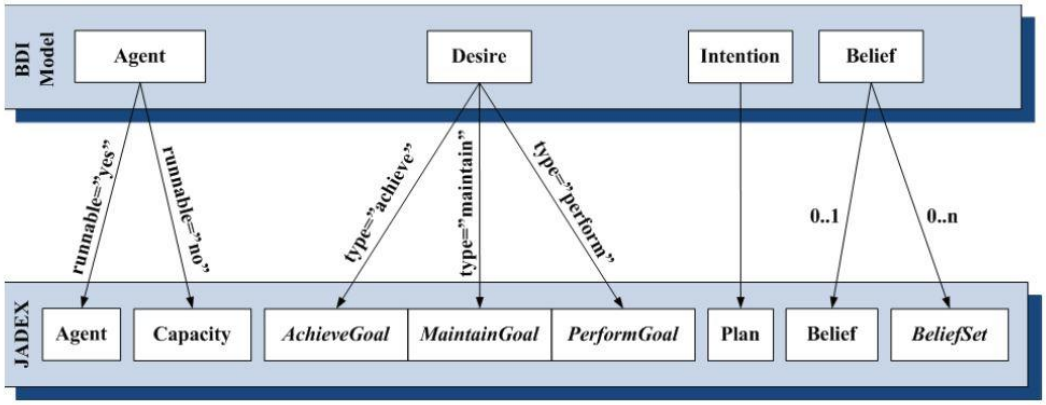


Figura 14 Associação entre abstrações BDI e abstrações Jadex (SERRANO; LEITE, 2011a).

Em nossa estratégia, restringimos a estratégia a sistemas autoadaptativos orientados a objetos (OO), para que pudéssemos adotar características específicas do paradigma OO ao realinhar os sistemas autoadaptativos OO com sua intencionalidade representada em um modelo de metas i*.

Apesar da existência de pesquisas que fazem o mapeamento de modelos i* para representações OO, se utilizássemos as heurísticas, tal como foram descritas na literatura, porém no sentido para trás, poderíamos gerar modelos extensos e menos abstratos, ou ter uma limitação de realinhar somente sistemas autoadaptativos multiagentes. Além disso, alguns elementos da POO como interfaces e classes abs-

tratas 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 3).

As nossas heurísticas de mapeamento de código OO para modelo i*, as quais são descritas mais adiante no Capítulo 3, podem ser utilizadas para obter um modelo de metas a partir de sistemas autoadaptativos OO, ainda que estes não sejam SMA, e também podem ser utilizadas para SMA implementados com linguagens OO, como o *framework* JADEX (BRAUBACH; POKAHR; LAMERSDORF, 2004) que estende a plataforma JADE (BELLIFEMINE et al., 2002) e possibilita a implementação de agentes intencionais na linguagem Java com base em especificações BDI (*Belief – Desire - Intention*) (SERRANO; LEITE, 2011a).

2.2. Consciência De *Software*

Nesta pesquisa, adotamos o conceito estendido de consciência de *software* conforme proposto por Cunha (CUNHA, 2014): "A capacidade do *software* obter conhecimento, através de sua própria percepção ou por dados externos, acerca de seu ambiente e sobre suas próprias metas". Cunha (CUNHA, 2014) identificou e organizou informações sobre a consciência de *software* sob a forma de um catálogo para requisitos não funcionais. Este catálogo foi modelado através do *Softgoal Interdependency Graph* (SIG) proposto no NFR *framework* (CHUNG et al., 2000).

O NFR *framework* (CHUNG et al., 1999, 2000) possibilita modelar requisitos não funcionais (RNF) e as interdependências entre eles em um *Softgoal Interdependency Graph* (SIG). No SIG, o requisito não funcional (RNF) é representado por *softgoals* que podem ser refinados por tipo, método ou correlação. O refinamento por tipo permite encontrar subtipos e a contribuição destes subtipos na satisfação a contento do tipo pai. O refinamento por método permite ao desenvolvedor catalogar técnicas de desenvolvimento para operacionalização do *softgoals*. A operacionalização do tipo superior é alcançada à medida que os subtipos são operacionalizados. O refinamento por correlação permite a criação de catálogos com a inferência de possíveis interações, positivas ou negativas, entre *softgoals*.

No catálogo de consciência de *software*, o requisito não funcional de consciência é instanciado para o tópico *software* e é refinado através do método de de-

composição por tipo. Como complemento ao catálogo, o autor descreveu cada requisito não funcional do SIG com base nos padrões de descrição de RNF propostos por Supakkul et al (SUPAKKUL et al., 2010). Utilizamos este tipo de catálogo (SIG) para facilitar a identificação de operacionalizações que estão implementadas no código-fonte do sistema ao realinhar o sistema autoadaptativo a um modelo de metas i*. No catálogo apresentado na Figura 15, a consciência de *software* foi refinada em quatro tipos:

— **Consciência de Contexto** – É a consciência do ambiente onde o *software* opera. Este conhecimento, por sua vez, é especializado em **ambiente computacional** (i.e.: recursos disponíveis, capacidade de rede, conexões e outros), **ambiente físico** (i.e.: valor de temperatura, presença de movimento ou fumaça, nível de ruído e assim por diante) e **localização** (i.e.: informação de localização em tempo real baseado em escopo global);

— **Consciência de Tempo** – Esta consciência é caracterizada pela percepção de mudança de tempo. Ainda que o tempo seja parte do contexto, Cunha (CUNHA, 2014) acredita no tempo como uma dimensão ortogonal ao contexto e às dimensões de consciência. Nesta visão, o tempo é importante para a consciência como de modo geral.

— **Consciência do Autocomportamento** – Estar ciente do seu próprio comportamento está relacionado a saber sobre suas metas, alternativas para satisfazer suas metas e a eficiência da sua própria operação.

— **Consciência de Relações Sociais** – Em um ambiente aberto, um *software* pode precisar conhecer sua vizinhança. A vizinhança é composta por atores como usuários ou outros softwares. Além de ter conhecimento sobre usuários, pode ser requerido estar ciente das normas que podem ser vistas como a definição de padrões de comportamento que visem regular o funcionamento do ambiente em geral e também pode ser preciso conhecer as diferentes relações em que podem estar envolvidos.

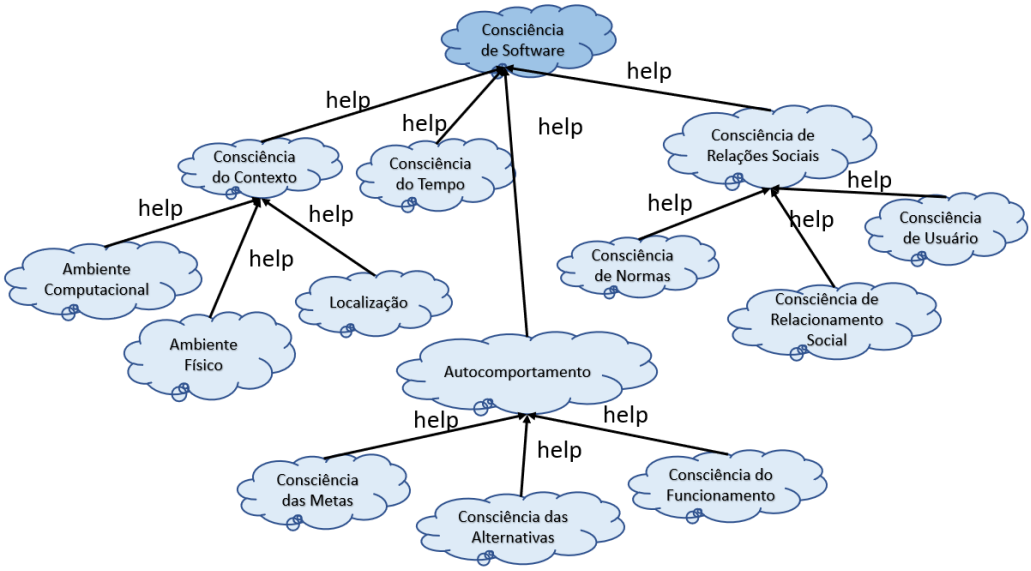


Figura 15 SIG de consciência de *software* adaptado de (CUNHA, 2014).

A fim de identificar operacionalizações que são relevantes para consciência de *software*, Cunha (CUNHA, 2014) utilizou o método *Goal Question Operationalization* (GQO) proposto em (SERRANO; LEITE, 2011b). As perguntas são apresentadas no catálogo através de padrões de perguntas para requisitos não funcionais, enquanto as operacionalizações são apresentadas através de padrões de operacionalização de requisitos não funcionais. Seguindo o método GQO, algumas perguntas gerais a consciência *software* são: "Quais dados devem ser adquiridos?", "Como obter os dados desejados?" e "Como os dados adquiridos devem ser interpretados?". Como a primeira questão depende do domínio do problema, o trabalho de Cunha concentrou-se em mecanismos para responder às duas últimas perguntas.

A Figura 16 mostra um exemplo de questões para cada tipo de consciência de contexto. Por exemplo, para identificar operacionalizações para a consciência de localização é relevante questionar “Onde está o usuário/dispositivo/*software*?”, “Que cobertura é necessária, local ou global?” e “O *software* irá operar em ambiente interno ou local aberto?”. Com base nestas perguntas, Cunha identificou algumas sugestões de operacionalizações apresentadas na Figura 17 para a consciência de localização.

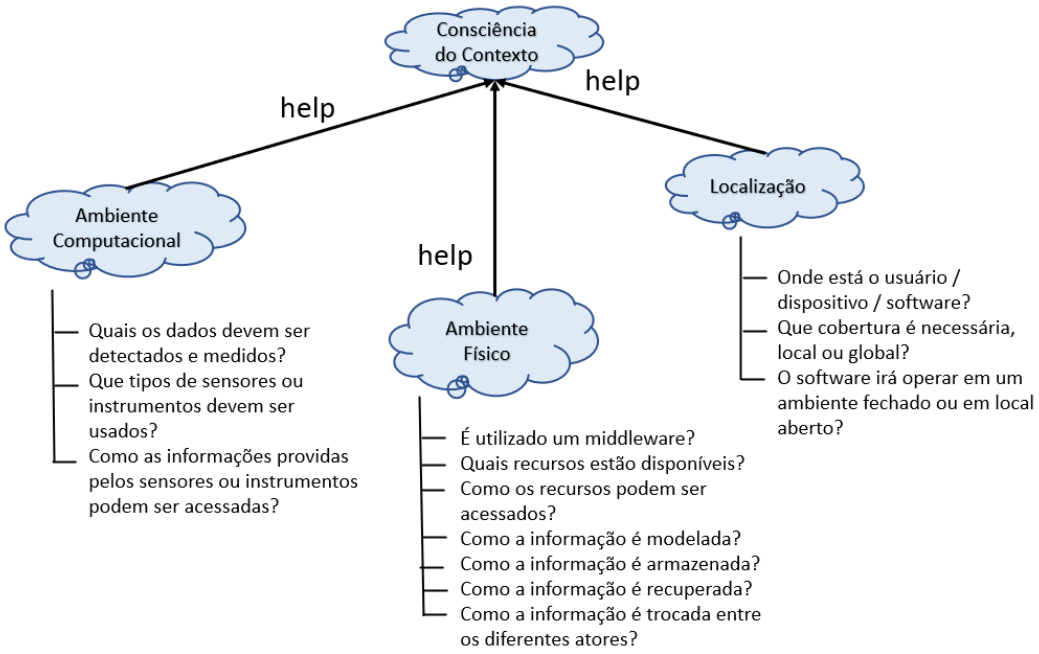


Figura 16 Padrão Questão (*Question Pattern*) para consciência de contexto.

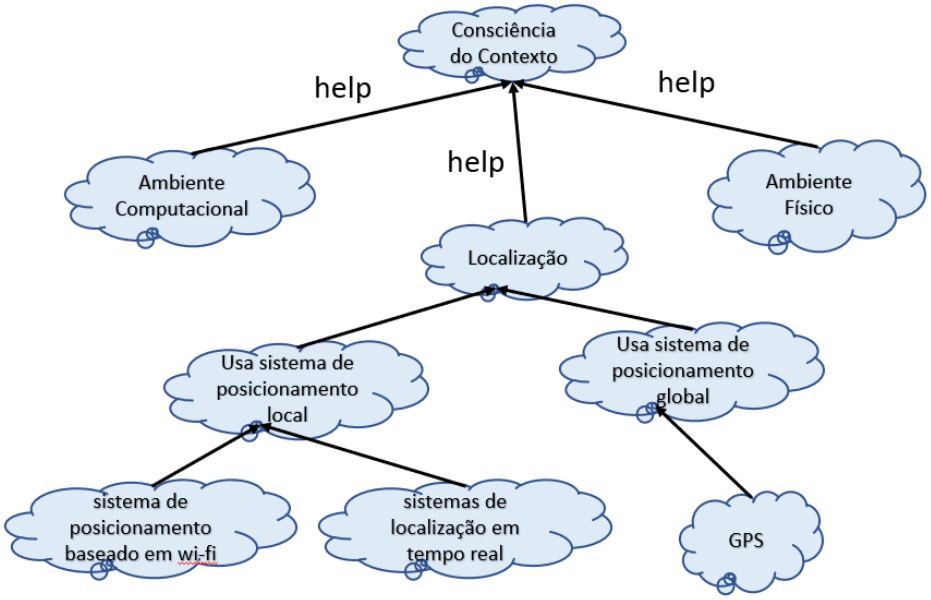


Figura 17 Operacionalizações para a consciência de localização.

A principal contribuição do catálogo de consciência de *software* é proporcionar a possibilidade de reuso de conhecimento do RNF de consciência de *software*, expresso através do refinamento desse requisito em subtipos de consciência de *software* e alternativas para operacionalização destes subtipos. Além do reuso do conhecimento embutido no catálogo propriamente dito, é possível reutilizar as perguntas elaboradas com o método GQO para identificar novas operacionalizações dentro do domínio e tecnologia envolvidos no contexto da reengenharia.

2.3.
Desenvolvimento De Sistemas Autoadaptativos

Um sistema autoadaptativo é um sistema de *software* intensivo aumentado com a habilidade de responder a uma variedade de mudanças que podem ocorrer em seu ambiente, metas, ou no próprio sistema por adaptar sua estrutura e comportamento em tempo de execução autonomamente (ABUSETA; SWESI, 2015). Pesquisas em desenvolvimento de sistemas auto adaptativos (ABUSETA; SWESI, 2015; ARCAINI; RICCOBENE; SCANDURRA, 2015; DE LEMOS et al., 2013; IGLESIA; WEYNS, 2015) identificaram que técnicas como os laços de controle de *feedback* são vitais para o desenvolvimento de tais sistemas. Neste tipo de técnica, o modelo **MAPE** ou **MAPE-K** proposto pela IBM (IBM, 2006) tem se destacado.

2.3.1.
MAPE

Segundo o padrão arquitetural da IBM (IBM, 2006), para um sistema ou parte dele se autogerir é preciso que ele tenha meios automáticos de coletar dados sobre o próprio sistema, possa analisar estes dados e determinar se mudanças são necessárias, possa criar um plano ou uma sequência de ações que especifiquem as mudanças necessárias e possa executar tais ações.

A Figura 18 mostra brevemente o funcionamento da gestão autonômica do modelo MAPE-K, também conhecido como MAPE pois subentende-se que o K ("*Knowledge*") é inerente ao MAPE, conforme descrito pelo projeto arquitetural da IBM (IBM, 2006).

No sistema de gestão autonômica, o **sensor** é responsável por coletar dados sobre o ambiente e sobre o sistema propriamente dito. Esses dados são utilizados pelo **monitor** para identificar sintomas (e.g.: métricas geradas a partir da agregação e filtro de dados coletados). Os sintomas são avaliados pelo **analisador** que correlaciona e modela situações complexas, identificando a necessidade de mudança, ou seja, quando isto ocorre, ele gera uma requisição de mudança. Esta requisição, por sua vez, é tratada pelo **planejador** que provê mecanismos para criar um plano de ação ou estratégia para alcançar metas e objetivos. O planejador utiliza as informações de política para guiar o seu trabalho. A estratégia, por sua vez, é utilizada pelo **executor** que controla a execução da estratégia, identificando o **atuador** necessário

para executar as atualizações dinamicamente. Todos os dados coletados, políticas e informações geradas (i.e.: sintomas, requisições de mudança, planos de mudança e políticas) são armazenados na **base de conhecimento** (“*Knowledge*”) da Figura 18.

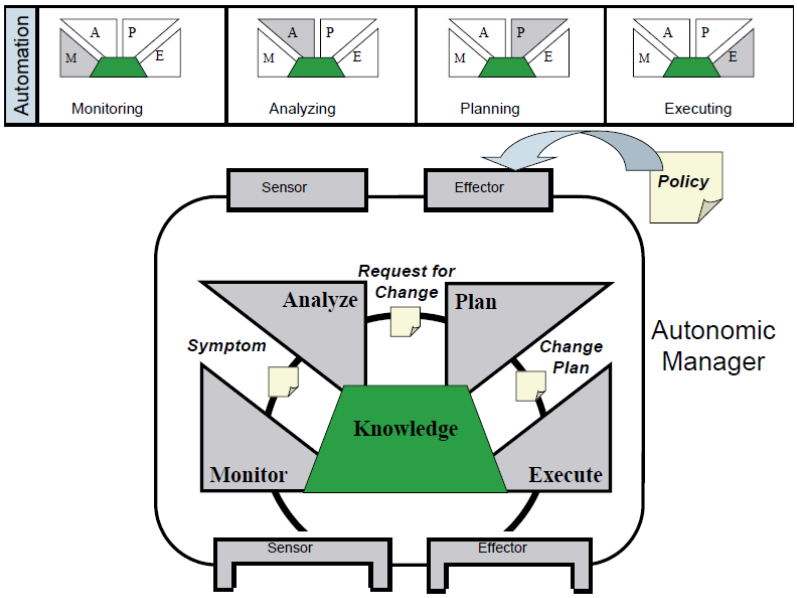


Figura 18 Modelo de referência MAPE-K proposto pela IBM (IBM, 2006).

Apesar deste padrão arquitetural ser bem difundido na literatura de sistemas auto adaptativos, o projeto de interação entre os diferentes componentes não é trivial. Abuseta e Swesi (ABUSETA; SWESI, 2015) ressaltam que na prática há várias possibilidades de interação entre os componentes do MAPE-K. Em algumas situações, um monitor pode ser notificado por vários sensores ou pode ter um sensor dedicado. Analogamente, um analisador pode ser notificado por um conjunto de monitores ou apenas um. Em outras situações, pode ser necessário que haja interação entre componentes do mesmo tipo (e.g.: monitor-monitor e analisador-analisador). Adicionalmente, os sensores podem coletar dados periodicamente ou a partir de algum evento, entre outros exemplos de variações que deverão ser avaliadas pelos engenheiros de *software* sistema a sistema.

O modelo de referência (ver Figura 18) da IBM prevê que cada gestor autonômico provê interfaces de seus sensores e atuadores para outros gestores autonômicos e/ou componentes na infraestrutura distribuída, o que ressalta as diferentes possibilidades de inter-relacionamentos citadas anteriormente. Neste contexto, surgiram algumas soluções que visam auxiliar a construção de sistemas auto adaptativos como frameworks (e.g.: *Rainbow* (GARLAN; SCHMERL; CHENG, 2009), Zanshin (SOUZA; MYLOPOULOS, 2012), CAA *framework* (BIEGEL; CAHILL,

2004) e SOCAM (GU; PUNG; ZHANG, 2004)), frameworks de referência (e.g.: MORPH (BRABERMAN et al., 2015)) e *design patterns* (e.g.: Abuseta e Swesi (ABUSETA; SWESI, 2015)).

O MAPE ou MAPE-k pode ser implementado interna ou externamente ao sistema e esta decisão deve ser realizada com base na existência de requisitos de consciência a serem atendidos pelo sistema.

Quando estes requisitos não são propriamente identificados como um RNF de primeira classe, os engenheiros de *software* podem não identificar a importância de se adotar uma arquitetura apropriada para a autoadaptação como o MAPE ou MAPE-K. Isto pode gerar soluções que mesclam as responsabilidades inerentes a autoadaptação com as funcionalidades do negócio e futuramente pode haver a necessidade de realizar a reengenharia do sistema.

2.3.2. Problemas De Implementação Do MAPE

Como o padrão MAPE (IBM, 2006) é um padrão de referência, conforme discutido aqui e em outros trabalhos da literatura, a sua implementação não é trivial e vários detalhes de baixo nível ficam a cargo do engenheiro de *software* e sua experiência. Como resultado, alguns sistemas autoadaptativos e até mesmo frameworks para desenvolvimento de tais sistemas possuem problemas de implementação do MAPE que são recorrentes na literatura.

Em (SERIKAWA et al., 2016), os autores identificaram dois problemas arquiteturais a saber: o problema **Monitores Oprimidos** que é caracterizado pela existência de pelo menos um monitor sendo responsável pela execução de dois ou mais monitores, isto força os monitores a terem a mesma frequência e uma ordem específica de execução mesmo que não seja necessário, e; o problema **Monitor Obscuro** que é caracterizado pelo fato da lógica de monitoramento, transmissão e pré-processamento estarem dispersas no código sem que o monitor esteja implementado como uma entidade de primeira classe.

Em (SAN MARTÍN et al., 2020), os desvios identificados e discutidos são: o problema **Entradas de Referência Dispersas** que ocorre quando os valores de referência utilizados pelo subsistema gerenciador para saber quais os estados do sistema gerenciado que devem ser alcançados e/ou mantidos estão dispersos pelo sistema sem uma abstração apropriada para eles; o problema **Executores e Atuadores Mistos** que é caracterizado pela falta de uma clara distinção entre executores

e atuadores, e; o problema **Alternativas Obscuras** que é caracterizado pela ausência de abstração apropriada para armazenar as alternativas de adaptação, tornando-as não evidentes no código.

Estes problemas recorrentes reduzem a qualidade do produto final e dificultam sua evolução. A Tabela 4 sumariza os impactos que descreveremos a seguir.

Quando há monitores oprimidos, existe um **alto acoplamento** entre os monitores que impõe uma ordem de execução e uso da mesma frequência de monitoramento/sensoriamento gerando **gasto desnecessário de recursos**. Enquanto, a existência de monitores obscuros leva a: baixa **reusabilidade**[9], pois não há uma modularidade dos monitores; dificuldade de compreensão do código, pois a junção da lógica de aquisição de diferentes dados promove uma baixa **compreensibilidade**[10]; e **manutenibilidade**[11], pois qualquer mudança em um monitor/sensor pode afetar os outros que estão juntos devido a falta de modularidade.

Quando as entradas de referência estão dispersas no código, essa dispersão compromete a **modularidade**[12] e **reusabilidade** desta informação. A **analisabilidade**[13] também é afetada, aumentando as chances de efeitos colaterais ao se adicionar ou remover novos valores de referência. A **modificabilidade**[14] e a **testabilidade**[15] também são impactadas uma vez que é gasto mais tempo para localizar tais valores dispersos e para testar esta abstração pode ser necessário criar casos de teste diferentes e independentes, respectivamente.

[9] Reusabilidade é o grau em que um ativo pode ser usado em mais de um sistema ou na construção de outros ativos. (“ISO/IEC 25010:2011”, [s.d.])

[10] Compreensibilidade é o grau em que sua finalidade é clara para o inspetor. Isso implica que nomes de variáveis ou símbolos são usados de forma consistente, módulos de código são auto descritivos e a estrutura de controle é simples ou de acordo com um padrão prescrito. (BOEHM; BROWN; LIPOW, 1976)

[11] Manutenibilidade é o grau de eficácia e eficiência com que um produto ou sistema pode ser modificado pelos mantenedores pretendidos. (“ISO/IEC 25010:2011”, [s.d.])

[12] Modularidade é o grau em que um sistema ou programa de computador é composto de componentes discretos, de modo que uma mudança em um componente tenha impacto mínimo em outros componentes. (“ISO/IEC 25010:2011”, [s.d.])

[13] Analisabilidade é o grau de eficácia e eficiência com o qual é possível avaliar o impacto em um produto ou sistema de uma alteração pretendida em uma ou mais de suas peças, ou diagnosticar um produto quanto a deficiências ou causas de falhas, ou identificar peças a serem modificadas. (“ISO/IEC 25010:2011”, [s.d.])

[14] Modificabilidade é grau em que um produto ou sistema pode ser modificado de forma eficaz e eficiente sem introduzir defeitos ou degradar a qualidade do produto existente. (“ISO/IEC 25010:2011”, [s.d.])

[15] Testabilidade é o grau de eficácia e eficiência com o qual os critérios de teste podem ser estabelecidos para um sistema, produto ou componente e os testes podem ser realizados para determinar se esses critérios foram atendidos

A existência do problema de executores e atuadores mistos indica que não há uma separação destas abstrações e isto reduz a **modularidade** e impacta a **reusabilidade**. Uma mudança em um destes elementos pode afetar o outro, uma vez que este problema afeta a **modificabilidade**. A **testabilidade** também é impactada devido ao acoplamento entre as abstrações.

Quando as alternativas de adaptação estão obscuras no código é porque elas estão implementadas junto a outras abstrações, o que indica uma **modularidade** não apropriada de tais alternativas que prejudica novas evoluções. A **analisabilidade** também é afetada e os engenheiros de *software* tem dificuldade de identificar efeitos colaterais ao modificar uma alternativa, por exemplo. Isto significa também que a qualidade de **modificabilidade** também é impactada negativamente.

Tabela 4 Atributos de qualidade impactados pelos problemas de implementação.

Problema	Impactos negativos na qualidade
Monitores Oprimidos	Alto acoplamento e gasto desnecessário de recursos
Monitor Obscuro	Reusabilidade, compreensibilidade e manutenibilidade
Entradas de Referência Dispersas	Modularidade, reusabilidade, analisabilidade, modificabilidade e testabilidade
Executores e Atuadores Mistos	Modularidade, reusabilidade, modificabilidade e testabilidade
Alternativas obscuras	Modularidade, analisabilidade e modificabilidade

2.4. Reengenharia

> "Reengenharia geralmente inclui alguma forma de engenharia reversa (para alcançar uma descrição mais abstrata) seguida de alguma forma de engenharia avante ou reestruturação. Isto pode incluir modificações no que diz respeito a novos requisitos não encontrados pelo sistema original." (CHIKOFSKY; CROSS, 1990).

A reengenharia pode ser utilizada nas circunstâncias em que a evolução do *software* é realizada independente do processo que deu origem ao *software* e, segundo Leite (LEITE, 1996), ela é composta por quatro principais subprocessos: recuperar, especificar, redesenhar e reimplementar. Com vistas a alcançar nosso propósito de sugerir uma reorganização arquitetural para acomodar melhor a qualidade

de consciência embutida no código legado, nós adotaremos uma estratégia de reengenharia onde cada subprocesso será centrado no uso da consciência como um RNF de primeira classe.

O modelo SADT da Figura 19, extraído e traduzido a partir de (LEITE, 1996), apresenta a forma como os subprocessos da reengenharia se comunicam a fim de evoluir o *software*. Isto possibilita tratar a manutenção a nível de design e cria oportunidades de reuso.

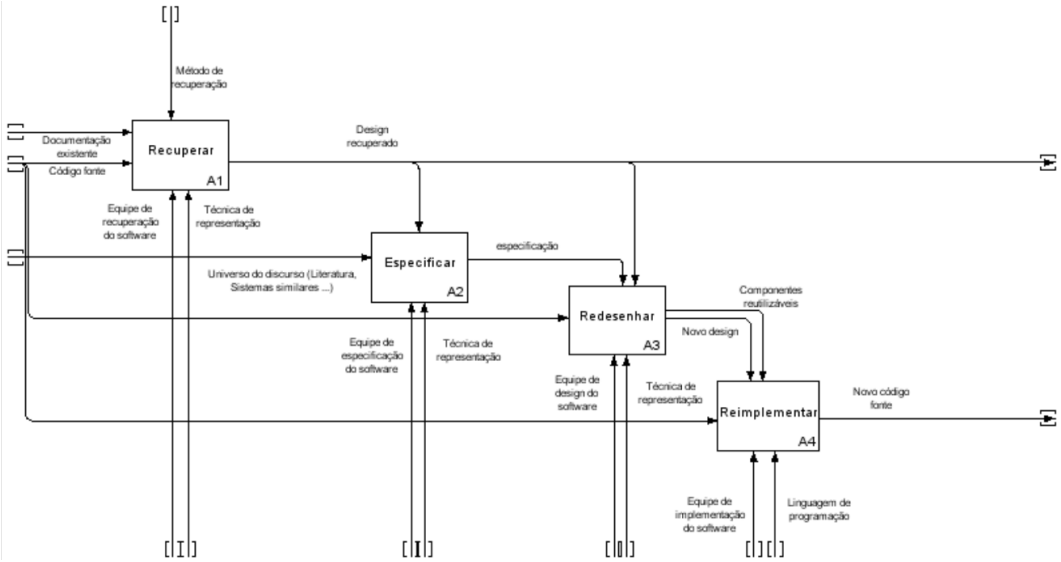


Figura 19 Método de reengenharia (LEITE, 1996).

Na Figura 19, o subprocesso recuperar é executado pela equipe de recuperação do *software* com base na documentação existente e no código fonte da aplicação. A equipe utiliza o método de recuperação para se guiar e recupera um design mais abstrato no formato da técnica de representação utilizada. O design recuperado é utilizado pela equipe de especificação do *software*, junto ao universo do discurso (e.g.: literatura e sistemas similares), para gerar a especificação do sistema ainda na linguagem da técnica de representação. A nova especificação, o design recuperado e o código fonte, por sua vez, são utilizados pela equipe de design do *software* para redesenhar a aplicação. Ao final do subprocesso de redesenhar, a equipe terá produzido um novo design em conformidade com a técnica de representação e um conjunto de componentes reutilizáveis para auxiliar a reimplementação da aplicação. Por fim, o novo design, o conjunto de componentes e o código fonte serão utilizados pela equipe de implementação no subprocesso reimplementar para gerar um novo código fonte.

2.5. Trabalhos Relacionados

Na literatura, encontramos trabalhos de reengenharia de softwares autoadaptativos que, de um modo geral, não oferecem a oportunidade de reorganização do mecanismo interno de autoadaptação desde os requisitos. Em nossa pesquisa, identificamos frameworks que oferecem um mecanismo de autoadaptação externo ao sistema, porém limitam a solução a uma tecnologia específica ou até mesmo possuem problemas arquiteturais, conforme demonstrado pelos trabalhos de Serikawa et al. (SERIKAWA et al., 2016) e San Martín et al. (SAN MARTÍN et al., 2020).

Também encontramos estratégias de refatoração, que, por sua natureza, são executadas no nível de código ou no nível de arquitetura e perdem a oportunidade de realinhar a aplicação com os seus requisitos e suas metas que são a base para a definição da arquitetura (CHUNG; DO PRADO LEITE, 2009). Além disso, identificamos o trabalho de Yu et al. (YU et al., 2005) que auxilia no resgate da intencionalidade do sistema para fins de reengenharia, porém não oferece suporte para compreender os RNFs de consciência já operacionalizados no código e sua ênfase é na geração de novos produtos com foco em reuso como web *services* e componentes.

O *framework Rainbow* (GARLAN et al., 2004; GARLAN; SCHMERL; CHENG, 2009) provê um mecanismo de suporte geral para a autoadaptação, que pode ser adaptado para diferentes classes de sistemas, e define uma linguagem, chamada Stitch, que se conecta ao *framework* e permite que a expertise de adaptação seja especificada e racionalizada, podendo ser utilizada para automatizar e coordenar adaptações para satisfazer múltiplos objetivos.

Conforme a Figura 20, o framework *Rainbow* possui elementos que atendem as funções do MAPE, porém o seu propósito é realizar a autoadaptação com base no modelo arquitetural do sistema a ser gerenciado. Seu propósito é prover uma forma dos engenheiros de *software* tornarem um sistema que não se adapta em autoadaptativo sem precisar reescrevê-lo inteiramente, possibilitando inclusive dimensionar fatores de qualidade para auxiliar na escolha da estratégia a ser utilizada na autoadaptação. No trabalho de Garlan et al. (GARLAN; SCHMERL; CHENG,

2009), a autoadaptação é feita a nível arquitetural e adiciona um mecanismo de autoadaptação externo ao sistema a ser adaptado.

Cámara et al (CÁMARA et al., 2016) relatam a adoção de uma arquitetura baseada em autoadaptação (ABSA - *Architecture-Based-Self-Adaptation*) para aprimorar sistemas legados autoadaptativos. Em seu relatório, eles relatam a reengenharia dos mecanismos de adaptação internos do sistema alvo e utilizam o *framework* Rainbow (GARLAN et al., 2004) para implementar o novo sistema. Entre as atividades realizadas por eles para compreender os mecanismos de adaptação atuais do sistema alvo estão: uso de textos, diagramas de componentes e fluxogramas.

Apesar do conhecimento adquirido acerca dos requisitos de adaptação sem o auxílio de uma base de conhecimento como um SIG, a representação destes é feita em nível arquitetural devido a própria proposta do framework *Rainbow*. Apesar de possuir elementos que suprem as funções de monitorar, analisar, planejar e executar do MAPE, este framework oferece a adoção de um tipo de arquitetura próprio que se baseia na arquitetura do sistema gerenciado e não é a ênfase do nosso trabalho.

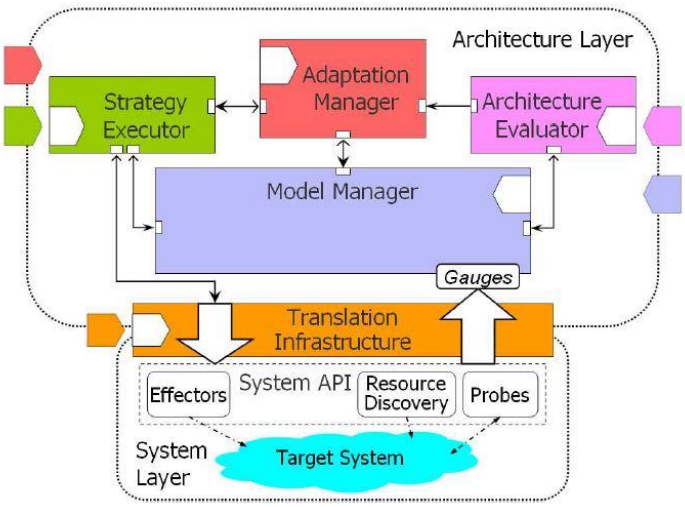


Figura 20 Framework *Rainbow*. (GARLAN; SCHMERL; CHENG, 2009)

Yijun et al. (YU et al., 2005) propõem uma estratégia de engenharia reversa que visa recuperar modelos de metas em GSP (*Goals, Skills and Preferences*) (HUI; LIASKOS; MYLOPOULOS, 2003) a partir de softwares legados para fins de reengenharia. A estratégia de reengenharia mencionada no trabalho de Yijun et al. (YU et al., 2005) é adaptada a partir do modelo "*Horseshoe*" (KAZMAN; WOODS;

CARRIÈRE, 1998) e tem o objetivo de obter softwares de alta variabilidade e genéricos. Para isto, os autores aplicam sua técnica de engenharia reversa sobre o código fonte de softwares legados que oferecem algum tipo de serviço (e.g.: envio de e-mail).

O modelo de metas resultante pode ser revisado, refinado e ampliado para gerar uma versão estendida do *software* que suporte o mesmo serviço, porém sob formas diferentes como Web *service* e/ou componentes. De modo geral, o método é composto por quatro grandes passos: "1) refatorar o código fonte através da extração de métodos com base em seus comentários; 2) Converter o código refatorado em um programa estruturado abstrato através de refatoração em diagramas de estado e construção de grafos de *hammock*; 3) extrair um modelo de metas a partir da árvore de sintaxe abstrata do programa estruturado; e 4) identificar requisitos não funcionais e derivar *softgoals* baseado na rastreabilidade entre o código e o modelo de metas".

Apesar do trabalho de Yijun et al. (YU et al., 2005) ser uma estratégia de engenharia reversa que tenta resgatar a intencionalidade do *software* legado para fins de reengenharia, a estratégia utilizada para identificar os RNFs não suporta a identificação dos tipos de consciência. Além disso, não são propostas estratégias para a especificação dos RNFs de consciência em direção a arquiteturas de *feedback loop* como o MAPE (IBM, 2006). Conforme mostra a Figura 21, a ênfase do trabalho de Yijun et al. (YU et al., 2005) é oferecer o reuso dos serviços disponíveis no *software* legado através de arquiteturas baseadas em Web *services* ou componentes. Para este fim, eles adaptaram o modelo *Horseshoe* (KAZMAN; WOODS; CARRIÈRE, 1998) para descobrir metas a partir do comportamento do sistema, ao longo da engenharia reversa do sistema, ao invés da recuperação da sua arquitetura estática.

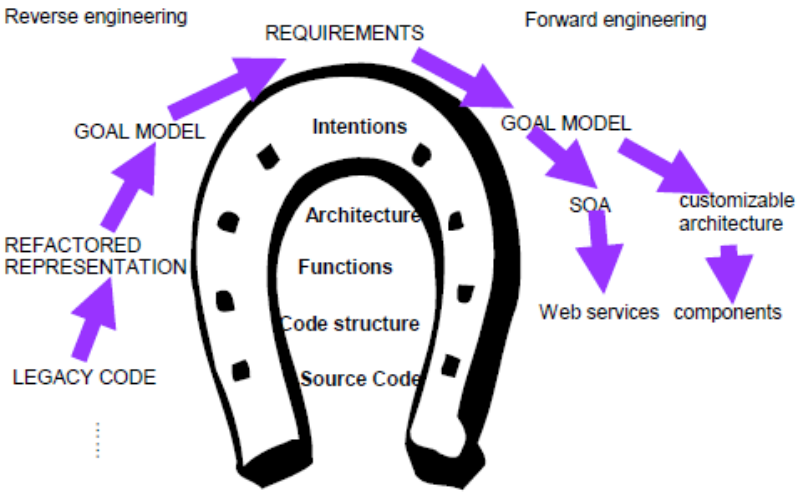


Figura 21 Modelo *Horseshoe* (KAZMAN; WOODS; CARRIÈRE, 1998) Usado em Yijun et al. (YU et al., 2005)

Serikawa et al. (SERIKAWA et al., 2016) propõem uma estratégia de refatoração para remover dois problemas arquiteturais na implementação da função Monitor do padrão MAPE em sistemas autoadaptativos. Os autores identificaram dois problemas arquiteturais a saber: o problema Monitores Oprimidos que é caracterizado pela existência de pelo menos um monitor sendo responsável pela execução de dois ou mais monitores, isto força os monitores a terem a mesma frequência e uma ordem específica de execução mesmo que não seja necessário, e; o problema Monitor Obscuro que é caracterizado pelo fato da lógica de monitoramento, transmissão e pré-processamento estarem dispersas no código sem que o monitor esteja implementado como uma entidade de primeira classe. Eles identificaram estes problemas até mesmo nos frameworks Zanshin[16] (SOUZA; MYLOPOULOS, 2012), CAA *Framework* (BIEGEL; CAHILL, 2004), SOCAM (GU; PUNG; ZHANG, 2004) entre outros. Os autores propõem uma estratégia de refatoração baseada em duas tarefas principais: criar uma classe para cada monitor existente no sistema, e; eliminar o laço único que gerencia os monitores visando remover a centralização da frequência de monitoramento e a sequência única de execução. A refatoração sugerida tem por objetivo garantir que os monitores estejam evidentes no código como entidades de primeira classe, independentes em termo de dados e ter sua própria frequência de execução. Como a refatoração, por definição, é uma estratégia

[16] https://github.com/sefms-disi-unitn/Zanshin

realizada no mesmo nível de abstração que, neste caso, é à nível de código, os autores criaram heurísticas para identificar monitores no código e em suas heurísticas cada sensor é equivalente a um monitor. Porém, outros trabalhos (ABUSETA; SWESI, 2015; ARCAINI; RICCOBENE; SCANDURRA, 2015; IBM, 2006) já mostraram que um monitor pode utilizar um ou mais sensores de acordo com o sintoma que ele objetiva monitorar, ou seja, de acordo com a sua meta. Deste modo, resgatar as consciências operacionalizadas em uma modelo de metas poderia auxiliar a dirimir entre o que são sensores e o que são efetivamente os monitores, além de revisar a frequência de cada sensor e o tipo de operacionalização da consciência.

San Martín et al. (SAN MARTÍN et al., 2020) analisaram sete sistemas autoadaptativos representativos, onde quatro deles são da conferência SEAMS (*International Symposium on Software Engineering for Adaptive and Self-Managing Systems*), e também identificaram desvios arquiteturais na implementação do modelo MAPE (IBM, 2006). Os desvios encontrados são: o problema Entradas de Referência Dispersas que ocorre quando os valores de referência utilizados pelo subsistema gerenciador para saber quais os estados do sistema gerenciado que devem ser alcançados e/ou mantidos estão dispersos pelo sistema sem uma abstração apropriada para eles; o problema Executores e Atuadores Mistos que é caracterizado pela falta de uma clara distinção entre executores e atuadores, e; o problema Alternativas Obscuras que é caracterizado pela ausência de abstração apropriada para armazenar as alternativas de adaptação, tornando-as não evidentes no código. Todos os problemas identificados pelos autores levam a impactos na qualidade do produto final: 1) maior acoplamento entre as entidades devidos as informações do mecanismo de adaptação estarem dispersas; 2) baixa coesão com informações e funcionalidades em locais não apropriados, o que também fere o padrão *Single Responsibility Principle* (SRP); 3) dificuldade de reutilização de partes do mecanismo de adaptação ; 4) dificuldade de compreensão da lógica de adaptação; 5) dificuldade de evoluir; entre outros impactos na qualidade.

San Martín et al.(SAN MARTÍN et al., 2020) mostram estratégias para identificar os problemas arquiteturais reportados e sugerem formas de refatoração para remover tais problemas: (A) Em caso de entradas de referência dispersas, é possível identificar o problema localizando-se o analisador e checando as regras que disparam adaptações pois estas regras possuem valores de referência em sua composição.

Em seguida, basta verificar se estes valores estão dispersos em vários módulos. Em caso de dispersão, sugere-se criar uma classe ou abstração que reúna os valores de referência; (B) Em caso de alternativas obscuras, é possível identificá-las localizando-se o analisador e as alternativas de adaptação as quais deveriam estar em uma abstração própria. Para remover este problema, as alternativas de adaptação devem estar declaradas isoladas de outros conceitos em uma abstração identificável para facilitar futuras evoluções; (C) Em caso de executores e atuadores mistos, é possível identificar o problema localizando os pontos responsáveis por executar as adaptações e discernir entre a lógica do executor e do atuador. Caso haja um misto das duas abstrações, os autores sugerem uma possível separação criando-se uma abstração para executor e outra para o atuador, onde os executores refere-se aos atuadores através de uma interface.

Em relação as estratégias de San Martín et al., descritas acima, temos as seguintes observações: : a estratégia (A) tenta identificar os valores de referência para reuni-los à nível de código e perde a oportunidade de leva-los à modelos de requisitos onde estes poderão ser revisados e utilizados na identificação de sintomas e das condições que os determinam; além disso, a estratégia (B), cuja ênfase é isolar as alternativas de adaptação de outros conceitos, poderia auxiliar na reconstrução das políticas que ligam os sintomas ao conjunto de ações para ajustar o sistema em determinada condição. Estas alternativas poderiam ser levadas a um modelo de requisitos para ser analisado junto a usuários e engenheiros de requisitos; Quanto a estratégia (C), o discernimento entre a lógica do executor e do atuador pode ser beneficiado pelo uso de um modelo de metas, onde as metas destes agentes possam ser elucidadas com vistas a dirimir os meios (“*rationale*”) para alcançar tais metas.

Por fim, não identificamos na literatura uma estratégia para a reengenharia de sistemas autoadaptativos que possibilite realinhá-los às práticas recomendadas pela literatura desde a engenharia de requisitos orientada a metas até o padrão arquitetural MAPE (IBM, 2006). Além disso, vimos nos trabalhos de Serikawa et al. (SERIKAWA et al., 2016) e San Martín et al. (SAN MARTÍN et al., 2020) que até mesmo frameworks que visam auxiliar na implementação do MAPE possuem problemas arquiteturais e impactam os sistemas que o reutilizam.

3 Técnicas Adaptadas ou Criadas

Neste capítulo, dissertamos acerca das técnicas adaptadas ou criadas no decorrer desta pesquisa para apoiar a estratégia de reengenharia. Elaboramos um conjunto de **heurísticas de mapeamento do paradigma OO para i***. Criamos um conjunto de ***SRconstructs*** **para especificar o RNF de consciência de *software*** em direção ao padrão arquitetural MAPE (IBM, 2006). Criamos também o ***framewok*** **JiStar** (MOURA; LEITE, 2020) que permite a rastreabilidade entre código Java e modelos de metas em i*, nos auxiliando nos subprocessos "recuperar" e "reimplementar". Propomos o **NFRJson** que é um formato para a representação de SIGs em json que nos auxilia na identificação de operacionalizações de consciência no código ao longo do subprocesso "recuperar". Escolhemos o json por ser leve e compreensível para engenheiros de *software* e máquinas.

3.1. Heurísticas De Mapeamento Do Paradigma OO Para i*

Assim como os atores do *framework* i* possuem dependências intencionais entre si que configuram uma colaboração para o alcance de suas metas, no paradigma OO (WALCZAK, 1998), múltiplos objetos colaboram entre si para a resolução de um problema através de troca de mensagens. Em (MOURA, 2017), propomos a versão inicial do nosso conjunto de heurísticas de mapeamento de elementos do paradigma OO para elementos do *framework* i*. Nesta tese, apresentamos uma extensão destas heurísticas. A Tabela 5 sumariza as heurísticas de mapeamento descritas a seguir.

Heurística 1. O termo classe pode ser compreendido como a definição de um conjunto de objetos similares (DIAZ, 1996; WALCZAK, 1998). A classe concreta é mapeada para agente, uma vez que, no *framework* i* (YU, 1995), um agente é um ator com manifestações físicas e concretas, podendo ser usado para referir-se a humanos assim como agentes artificiais como hardware e/ou *software*.

Tabela 5 Heurísticas de mapeamento do paradigma OO para o *framework* i*.

Id	Paradigma OO	i*
1	Classe concreta	Agente
2	Classe abstrata	Ator
3	Interface	Papel
4	Relacionamento de especialização	Relacionamento é-um (is-a) entre as classes envolvidas
5	Relacionamento de implementação	Relacionamento atua (plays) entre a classe e a interface
6	Método	Tarefa
7	Método construtor	Meta com a descrição dos parâmetros se houver e uma tarefa com um relacionamento de meios-fim para a meta
8	Atributo	Recurso
9	Parâmetro	Recurso e um relacionamento de decomposição de tarefa do tipo "Recurso para"
10	Invocação de métodos da própria classe	Relacionamento decomposição de tarefa em subtarefa
11	Configurar valor em atributo	Relacionamento do tipo meios-fim de tarefa para recurso
12	Invocação de um método pertencente a outra classe	Relacionamento de dependência de meta entre a tarefa e o outro ator. Se houver passagem de parâmetro, é criado também um relacionamento de dependência de recurso entre o ator e a tarefa
13	Invocação de um método com retorno pertencente a outra classe	Relacionamento de dependência de recurso entre a tarefa e o outro ator. Se houver passagem de parâmetro, é criado também um relacionamento de dependência de recurso entre o ator e a tarefa

Heurística 2. Uma classe abstrata é uma classe mais genérica que permite a definição de novas classes a partir de sua especialização (WALCZAK, 1998). A classe abstrata é mapeada para ator, pois, no *framework* i* (YU, 1995), o termo ator é usado para se referir genericamente a qualquer unidade para a qual se possa atribuir dependências intencionais e que também pode ser especializado.

Heurística 3. Segundo o paradigma OO (LEWIS; PÉREZ-QUIÑONES; ROSSON, 2004), uma interface é um contrato sob a forma de uma coleção de declarações de métodos e constantes. Quando uma classe implementa uma interface, ela promete implementar os métodos declarados naquela interface. Em nossas regras de mapeamento, a interface é mapeada para um papel, pois, no *framework* i* (YU, 1995), o termo papel é uma caracterização abstrata do comportamento de um ator social dentro de algum contexto e suas características são facilmente transferíveis para outros atores.

Heurística 4. O relacionamento de especialização é mapeado para um relacionamento é-um (*is-a*) entre as classes envolvidas que, por sua vez, serão mapeadas para algum tipo de ator.

Heurística 5. O relacionamento de implementação ou realização entre uma classe e uma interface é mapeado para um relacionamento atua (*plays*) entre a classe e a interface que, por sua vez, serão mapeadas para um agente ou ator e um papel respectivaemte.

Heurística 6. O método é mapeado para uma tarefa.

Heurística 7. O método construtor é mapeado para uma meta com a descrição dos parâmetros se houver e uma tarefa com um relacionamento de meios-fim para a meta.

Heurística 8. O atributo é mapeado para recurso.

Heurística 9. O parâmetro é mapeado para um recurso com um relacionamento de decomposição do tipo "Recurso para" entre o método ("tarefa") e o parâmetro ("recurso").

Heurística 10. A invocação de métodos da própria classe é mapeada para um relacionamento de decomposição de tarefa do tipo "subtarefa" entre os métodos envolvidos que, por sua vez, serão mapeados para tarefas.

Heurística 11. A configuração de valor em atributo dentro de um método é mapeada para um relacionamento meios-fim do método para o atributo que, por sua vez, serão mapeados para uma tarefa e um recurso respectivamente.

Heurística 12. A invocação de um método pertencente a outra classe é mapeada para um relacionamento de dependência de meta do método para a outra classe que, por sua vez, são mapeados para tarefa e ator respectivamente. Caso haja passagem de parâmetro, deve ser criado também um relacionamento de dependência de recurso do ator para a tarefa em questão.

Heurística 13. A invocação de um método com retorno pertencente a outra classe é mapeada para um relacionamento de dependência de recurso do método para a outra classe que, por sua vez, são mapeados para tarefa e ator respectivamente. Se houver passagem de parâmetro, é criado também um relacionamento de dependência de recurso do ator para a tarefa em questão.

3.2. Especificação Do RNF De Consciência Em Modelos I*

O RNF de consciência de *software* é crucial em sistemas autoadaptativos e precisa ser tratado como um requisito de primeira classe, pois os requisitos não funcionais auxiliam na seleção entre alternativas no design arquitetural (CHUNG; NIXON; YU, 1995). Vimos, no Capítulo 2, que o padrão arquitetural MAPE (IBM, 2006) é umas das abstrações de laços de controle mais recomendadas para sistemas auto adaptativos. Entretanto, vimos também que, por ser um padrão de referência, existem lacunas entre o padrão e a implementação propriamente. Lacunas estas que vários pesquisadores têm tentado preencher ao longo dos anos e, apesar disto, ainda existem problemas recorrentes na implementação do MAPE (SAN MARTÍN et al., 2020; SERIKAWA et al., 2016).

Entendendo a importância de modelar o conceito de consciência desde os requisitos para guiar a implementação futura deste RNF, Cunha e Leite (CUNHA; DO PRADO LEITE, 2014) propuseram uma extensão ao modelo i* e também um padrão para a modelagem do RNF de consciência em modelos i*. A extensão do modelo i* teve dois objetivos a saber: representar consciência através de contexto, com suas respectivas situações, e permitir a operacionalização (refinamento) do requisito de consciência guiando a implementação do mesmo. Na extensão do modelo i*, definida por Cunha e Leite (CUNHA; DO PRADO LEITE, 2014), foi incluso um tipo de elemento intencional para contexto e um relacionamento de argumentação entre o elemento de contexto e os meios alternativos para se alcançar a meta relacionada ao contexto. através de elos de argumentação é possível representar também as situações de um contexto tanto quanto o impacto que estas situações têm na escolha de alternativas para satisfação das metas as quais o contexto está relacionado.

Inspirado no padrão MAPE da IBM (IBM, 2006), Cunha (CUNHA, 2014) propôs uma abordagem para operacionalizar o requisito de consciência dividida em duas etapas integradas: a aquisição de dados, equivalente à função de monitoração no MAPE, e; a interpretação dos dados adquiridos, equivalente à função de análise no MAPE. A etapa de interpretação dos dados, por sua vez, é dividida em: identificação da situação a partir dos dados adquiridos; e Avaliação da ação alternativa. O conjunto com a meta flexível de consciência (fim) juntamente com o elemento de

consciência de contexto (meio) e as subtarefas para aquisição de dados e interpretação de dados, formam o *SRconstruct* da Figura 22.

Conforme mostra a estrutura *SRconstruct* da Figura 22, A tarefa "aquisição de consciência de contexto" é o meio para operacionalizar a consciência do tópico em questão. Esta mesma tarefa é decomposta em duas subtarefas, a saber: "aquisição de dados"; e "interpretação dos dado". Conforme a argumentação, "! situação A" ou "! situação B", são disparadas as tarefas "alternativa de ação 1" ou "alternativa de ação 2", respectivamente. Estas tarefas, por conseguinte, são meios para alcançar a meta de adaptação do sistema.

Como nossa estratégia de reengenharia visa realinhar os sistemas autoadaptativos com as boas práticas de arquitetura recomendadas pela literatura para tais sistemas (i.e.: MAPE), optamos por adaptar e expandir o *SRconstruct* proposto por Cunha e Leite (CUNHA; DO PRADO LEITE, 2014) para contemplar as funções do MAPE. Então, propomos inicialmente um conjunto de *SRconstructs* da (MOURA et al., 2019) apresentado na Figura 23. Note que criamos um *SRconstruct* para cada elemento do MAPE pois cada um deles tem sua própria meta no contexto da autoadaptação e "*rationale*" para alcançar tal meta.

Apesar da nossa proposta inicial contemplar as funções do MAPE, ainda não havia uma abstração clara para sensores e atuadores. Identificamos na literatura que a ausência destas abstrações traz impactos negativos à evolução dos sistemas autoadaptativos. Deste modo, nossos *SRconstructs* foram revisados a partir da sua versão inicial (MOURA et al., 2019) com o propósito de mitigar os problemas de implementação do MAPE recorrentes na literatura: monitor obscuro e monitores oprimidos de Serikawa et al. (SERIKAWA et al., 2016), e; Entradas de Referência Dispersas, Executores e Atuadores Mistos e Alternativas Obscuras de San Martín (SAN MARTÍN et al., 2020). Para alcançar nosso propósito, reutilizamos a meta arquitetura MAPE e o conjunto de design patterns para MAPE proposto por Abuseta e Swesi (ABUSETA; SWESI, 2015), obtendo como resultado os *SRconstructs* da Figura 24.

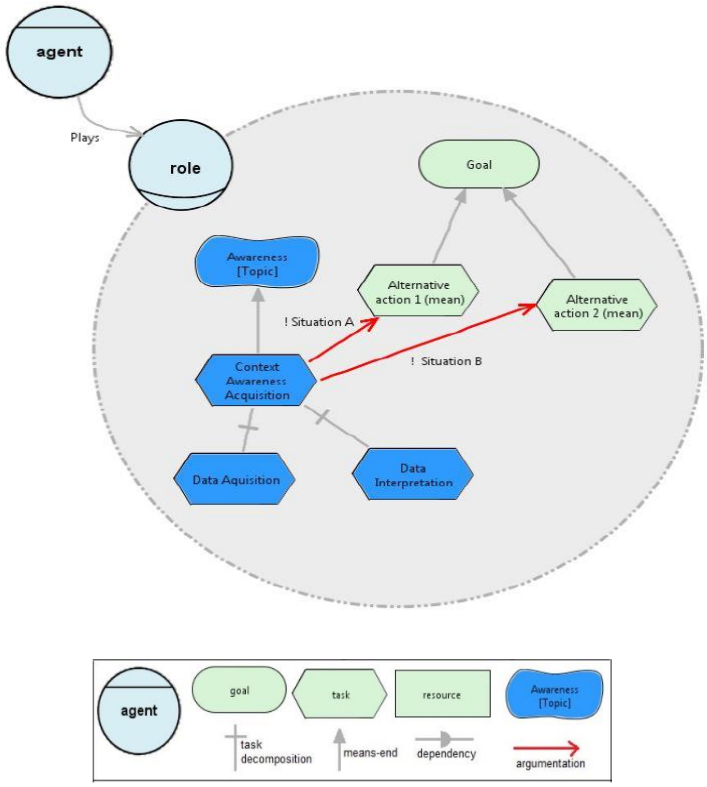


Figura 22 *SRconstruct* para o RNF de consciência de Cunha e Leite (CUNHA; DO PRADO LEITE, 2014).

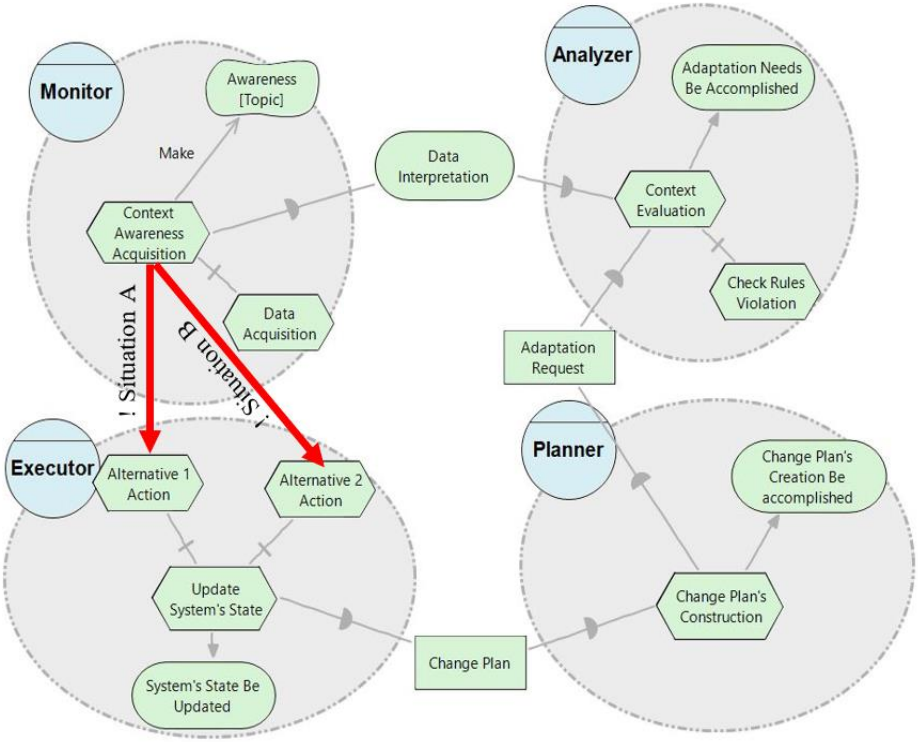


Figura 23 *SRconstruct* para o RNF de consciência (MOURA et al., 2019).

Conforme a Figura 24, para cada operacionalização de consciência, é preciso que seis agentes interajam conforme o modelo MAPE. Cada agente possui sua meta particular em conformidade com sua função no padrão MAPE e os elementos necessários para alcançar esta meta compõem cada *SRconstruct* junto com a meta. Os nomes entre colchetes deverão substituídos conforme o sistema alvo da reengenharia.

O agente Sensor possui a meta "Que o [dado] seja adquirido", ou seja, ele é responsável por operacionalizar a consciência sobre determinado dado. A tarefa "adquirir [dado]" é o meio para alcançar a meta e disponibilizar o dado para o agente Monitor. A tarefa "adquirir [dado]" necessita do recurso gatilho para disparar que pode ser uma frequência de tempo fixa, uma resposta a evento ou sob demanda.

O agente Monitor possui a meta "Que o [estado do sistema] seja monitorado". Para alcançar esta meta, o meio utilizado é a tarefa "Monitorar [estado do sistema]" e esta mesma tarefa operacionaliza a consciência de determinado estado do sistema. A tarefa "Monitorar [estado do sistema]" também é o meio para gerar o log do estado do sistema que é um recurso utilizado pelo analisador.

O agente Analisador, por sua vez, tem a meta "Que o [sintoma] seja verificado". O meio para alcançar esta meta é a tarefa "Verificar [sintoma]", na qual é verificada em uma lista de sintomas se há algum desvio dos valores de referência para as propriedades do sistema. Caso seja identificado algum sintoma, o Analisador enviará uma requisição de adaptação ao planejador contendo a lista de sintomas (i.e.: eventos e condições) detectados.

O agente Planejador possui a meta "Que uma estratégia para tratar [sintoma] seja elaborada", cujo meio para alcançá-la é a tarefa "Elaborar estratégia para tratar [sintoma]". Esta tarefa, consome o recurso "Requisição de adaptação" com os sintomas identificados e verifica na lista de políticas representada pelo recurso "Política [evento + condição -> ação]" quais ações ele deve adicionar à estratégia de modo a reestabelecer o sistema ao seu estado desejado. Uma vez elaborada a estratégia com as ações necessárias e na ordem de execução, o Planejador aciona o executor para desempenhar a estratégia.

O agente Executor possui a meta “Que a [estratégia] seja executada” que visa executar as ações definidas na estratégia. Estas ações são alternativas para alcançar alguma meta do sistema que não estava mais sendo alcançada conforme detectado pelo Analisador. A tarefa “Executar [estratégia]” é o meio para alcançar a meta deste agente e, para executar cada ação prevista na estratégia, ele depende do agente Atuador,

O agente Atuador possui a meta “Que o [estado do sistema] seja atualizado” e ele possui diferentes meios para ajustar o estado do sistema. A seleção do meio é feita pelo agente Executor pois este seleciona as ações conforme a estratégia.

Conforme Figura 24, cada elemento do padrão MAPE possui uma meta a ser alcançada e essa meta é o centro do seu *SRconstruct*. Além dos *SRconstructs* propriamente, é importante ressaltar que esses *SRconstructs* possuem dependências entre si, conforme o modelo da Figura 25. Deste modo, as dependências entre os agentes podem compor *SDsituations*[17]. No modelo da Figura 24, nós mostramos de modo mais explícito como os elementos de dependência são gerados (“meios-fim”) e como são consumidos (“decomposição de tarefa”). Enquanto o modelo da Figura 25 mostra os relacionamentos de dependência propriamente.

[17] Uma SDsituation define um bloco de elementos de dependência com intencionalidade situacional compartilhada.

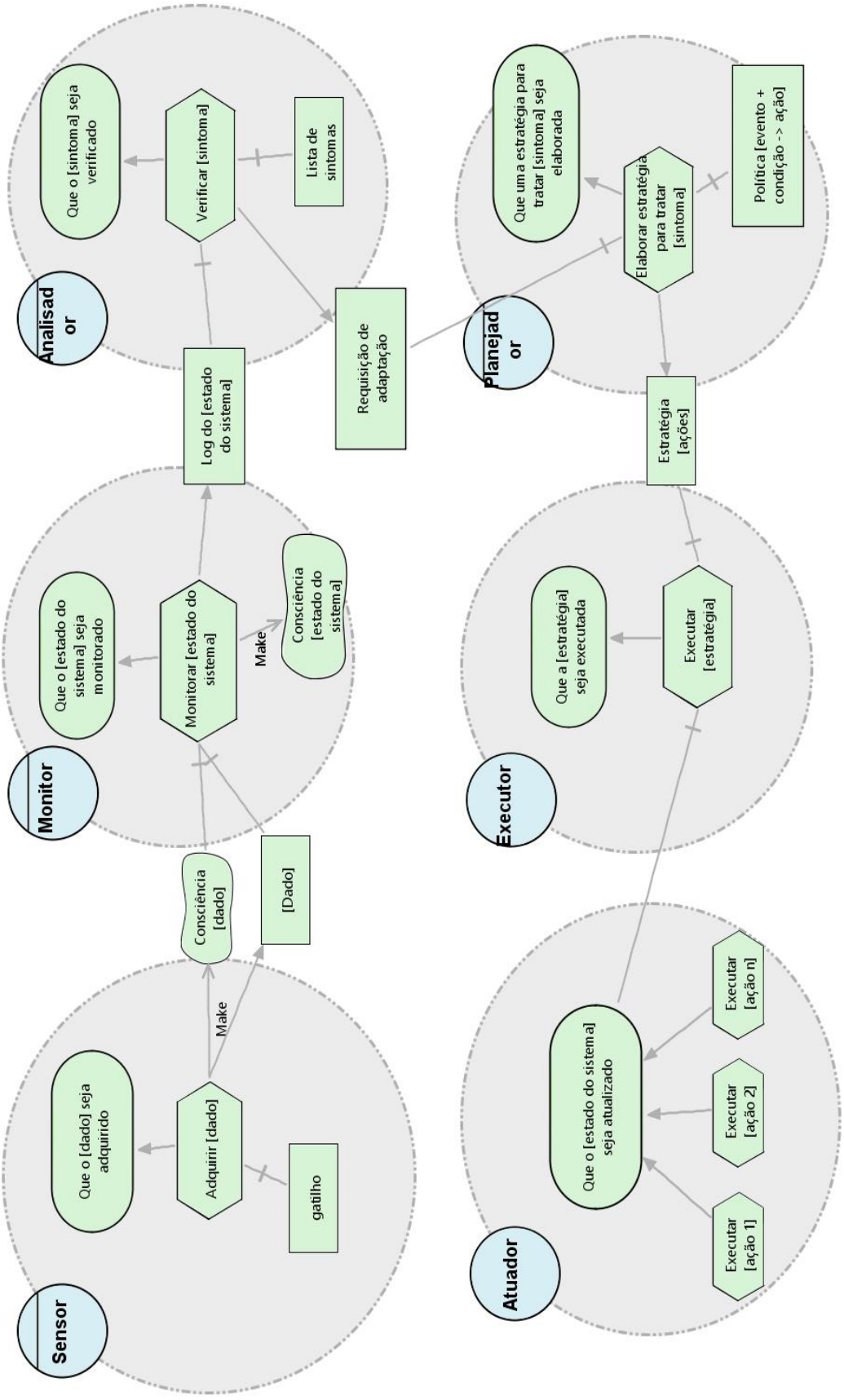


Figura 24 Modelo SR com *SRconstructs* para a modelagem do RNF de consciência de *software*.

No modelo MAPE, existem as seguintes situações de dependência estratégica entre os seus elementos:

— Monitoramento do estado do sistema: esta situação envolve o monitor e os sensores que ele utiliza para monitorar o estado do sistema em execução. Após o monitoramento, ele gera o log do estado do sistema que é utilizado pelo analisador. Caso haja dependências sensor-sensor e monitor-monitor estas podem ser especificadas aqui.

— Análise de sintomas: esta situação envolve o analisador e os monitores dos quais o primeiro espera receber o log do sistema. Como ele espera receber um ou mais logs do sistema e não importa como o recurso é gerado, os monitores que já tiveram o seu "*rationale*" especificado na situação de dependência estratégica "Monitoramento do estado do sistema" podem ser representados com o ator propriamente e sem a sua fronteira. Caso haja dependências entre analisadores, estas devem ser representadas aqui.

— Planejamento da estratégia de adaptação: esta situação relaciona o planejador e os analisadores dos quais ele espera receber alguma requisição de adaptação. Como ele espera receber uma ou mais requisições de adaptação e não importa como o recurso é gerado, os analisadores que já tiveram o seu "*rationale*" especificado na situação de dependência estratégica "Análise de sintomas" podem ser representados com o ator propriamente e sem a sua fronteira. Caso haja dependências entre planejadores, estas devem ser representadas aqui.

— Execução da adaptação do sistema: esta situação envolve o executor, o planejador dos quais ele espera receber a estratégia e os atuadores que ele utilizará para executar as ações. Como ele espera receber uma ou mais estratégias e não importa como o recurso é gerado, os planejadores que já tiveram o seu "*rationale*" especificado na situação de dependência estratégica "Planejamento da estratégia de adaptação" podem ser representados com o ator propriamente e sem a sua fronteira. Caso haja dependências entre executores, estas devem ser representadas aqui.

Com base no modelo SR da Figura 25, descrevemos estas *SDsituations* com o uso de *frames* (OLIVEIRA, 2008).

***SDsituation* Definition**				
SDsituation	Monitoramento do estado do sistema			
Goal	Que o estado do sistema seja monitorado			
Definition	Representa a situação de monitoramento do sistema com base nos dados coletados pelos sensores com vistas a atualizar o log de estado do sistema.			
ACTOR (*depender*)	ELEMENTS OF DEPENDENCY	TYPE	DEGREE	ACTOR (DEPENDEE)
Monitor	Consciência [Dado]	*Softgoal*	*Critical*	Sensor
Monitor	Dado	*Resource*	*Critical*	Sensor

***SDsituation* Definition**				
SDsituation	Análise de sintomas			
Goal	Que sintomas sejam verificados			
Definition	Representa a situação de analise o log do estado do sistema para verificar a existência de sintomas, o que indica a necessidade de adaptação			
ACTOR (*depender*)	ELEMENTS OF DEPENDENCY	TYPE	DEGREE	ACTOR (DEPENDEE)
Analisador	Log do estado do sistema	*Resource*	*Critical*	Monitor

***SDsituation* Definition**				
SDsituation	Planejamento da estratégia de adaptação			
Goal	Que uma estratégia seja elaborada			
Definition	Representa a situação de planejamento de uma estratégia para tratar uma requisição de adaptação			
ACTOR (*depender*)	ELEMENTS OF DEPENDENCY	TYPE	DEGREE	ACTOR (DEPENDEE)
Planejador	Requisição de adaptação	*Resource*	*Critical*	Analisador

***SDsituation* Definition**				
SDsituation	Execução da adaptação do sistema			
Goal	Que uma estratégia seja executada			
Definition	Representa a situação de execução de uma estratégia com o auxílio dos atuadores.			
ACTOR (*depender*)	ELEMENTS OF DEPENDENCY	TYPE	DEGREE	ACTOR (DEPENDEE)
Executor	Estratégia	*Resource*	*Critical*	Planejador
Executor	Que o estado do sistema seja atualizado	*Goal*	*Critical*	Atuador

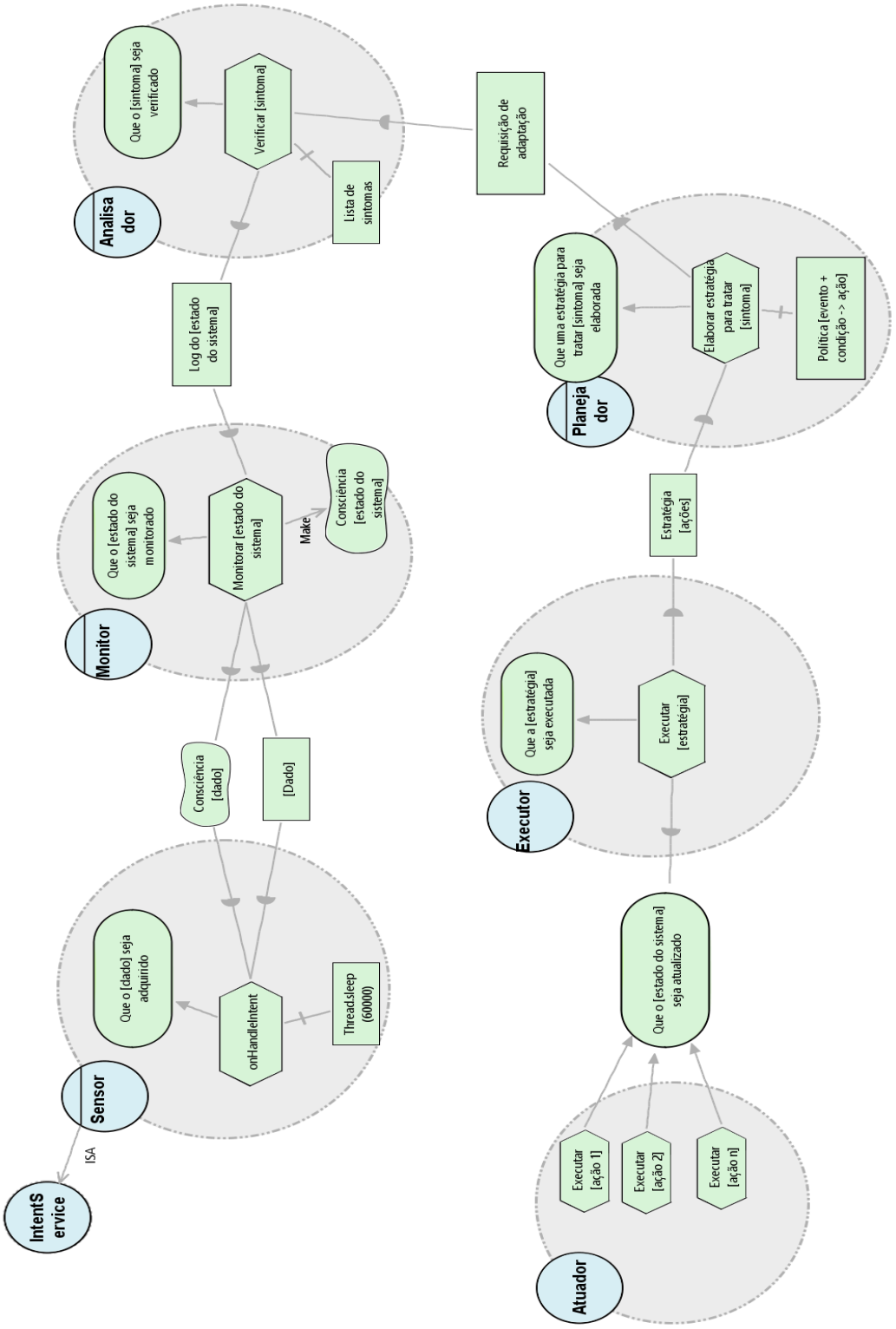


Figura 25 Dependências estratégicas entre os *SRconstructs* dos elementos do MAPE.

3.3. JiStar

O *framework* JiStar[18] oferece um conjunto de anotações de código Java para descrever a intencionalidade do *software* e uma exportador de modelos de metas no formato iStarML (CARES et al., 2011), PiStar (PIMENTEL; CASTRO, 2018) e HTML.

Este *framework* nos auxilia em dois subprocessos da estratégia de reengenharia: recuperar e reimplementar. No subprocesso recuperar, ele nos auxilia a obter um modelo de metas *AS-IS* mantendo a rastreabilidade com o código fonte. No subprocesso reimplementar, ele nos auxilia a manter a rastreabilidade para os elementos do mecanismo de autoadaptação que foram reimplementados com vistas a facilitar futuras evoluções.

A seguir apresentamos o conjunto de anotações de código Java para mapear elementos do *framework* i* (YU, 1995), onde os "atributos" com * indicam que é de preenchimento obrigatório.

@Actor(name*="nome do ator", 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 determinada unidade de código. É possível utilizar mais de uma @Actor por unidade de código. Ao exportar um modelo i* em qualquer formato, os elementos associados a algum ator aparecerão em seu interior ("fronteira") no modelo.

***@Goal*(name*="nome da meta", description="descrição da meta", actor*= "nome do ator")**

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.

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

***@Softgoal*(name*=”nome da meta flexível”, topic=”tópico”, description=”descrição da meta flexível”, elementType*=[NFR_SOFTGOAL | OPERATIONALIZING_SOFTGOAL | CLAIM_SOFTGOAL), label=[SATISFICED | DENIED | CONFLICTING | UNDEFINED], priority=[CRITICAL | DOMINANT | UNDEFINED], actor*=”nome do ator”)**

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 desejam alcançar naquela unidade de código. Os atributos que compõem esta anotação são baseados no *framework* i* e também no NFR *framework*. É possível utilizar mais de uma *@Softgoal* por unidade de código. As operacionalizações destas metas flexíveis, quando disponíveis, serão indicadas pela anotação @Contribution.

@Task(name*=”nome da tarefa”, description*=”descrição da tarefa” , actor*= “nome do ator”)

Esta anotação pode ser utilizada em métodos para fornecer informações acerca da tarefa realizada pelo método em questão.

***@Resource*(name*=”nome do recurso” , actor*= “nome do ator”)**

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.

***@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, será gerado um elemento tarefa e esta será relacionada a 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.

@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 em subtarefa, recurso, submeta ou meta flexível. É possível utilizar mais de uma @TaskDecomposition por unidade de código.

@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.

Como mencionado anteriormente, o *framework* JiStar permite a exportação de modelos de metas em i* nos formatos HTML, iStarML(CARES et al., 2011) e PiStar[19] (PIMENTEL; CASTRO, 2018). Os arquivos gerados são descritos na Tabela 6. Um modelo de metas facilita a leitura das metas por parte dos engenheiros de requisitos, clientes e as partes interessadas no negócio suportado pelo sistema. Com o uso do *framework* JiStar é possível manter uma rastreabilidade entre o código e as metas do sistema, possibilitando ter um modelo de metas sincronizado em mãos, onde as partes interessadas podem contribuir para a evolução do *software*.

Tabela 6 Formatos de exportação de modelos.

Formatos de exportação	Descrição
-istarml	Exportação para o formato iStarML (CARES et al., 2011). Neste formato, é gerado um arquivo *goal*_model.istarml que pode ser importado em ferramentas compatíveis com o formato (e.g.: OpenOME[20], Hime[21]).
-pistar	Exportação para a ferramenta PiStar 2.[19] (PIMENTEL; CASTRO, 2018). Neste formato, o *framework* gera um arquivo chamado *goal*_model.txt que pode ser importado na ferramenta PiStar.
-html	Exportação para o formato HTML. Neste formato, cada ator gera uma página HTML contendo os elementos do seu “*rationale*” e relacionamentos entre atores.

Ainda que tenhamos conseguido fazer a exportação de modelos de metas i* para a ferramenta PiStar, que permite a modelagem em iStar 2.0 (DALPIAZ;

[19] https://www.cin.ufpe.br/~jhcp/pistar/#
[20] https://se.cs.toronto.edu/trac/ome/
[21] upc.edu/gessi/istar/tools/hime/index.html

FRANCH; HORKOFF, 2016a), alguns dos elementos anotados não são exportados devido às diferenças estruturais e semânticas entre as duas versões do *framework* i*, conforme explicitado na Seção 2.1.3.

Quanto a estratégia, o *framework* JiStar pode gerar modelos de metas i* utilizando as nossas heurísticas de mapeamento de código OO para modelos de metas i* (Tabela 5 no Capítulo 4) e/ou as anotações de código. As opções e o parâmetro utilizado para a seleção da estratégia são apresentados na Tabela 7.

Tabela 7 Estratégias de geração de modelos.

Formatos de exportação	Descrição
-auto	Gera o modelo de metas utilizando nosso conjunto de heurísticas de código OO para modelos de metas i* (Tabela 5 no Capítulo 4).
-semiauto	Gera o modelo de metas utilizando nosso conjunto de heurísticas de código OO para modelos de metas i* (Tabela 5 no Capítulo 4) em conjunto com as anotações presentes no código fonte. Nesta estratégia a anotação tem prioridade sobre as heurísticas.
-manual	Gera o modelo de metas considerando somente as anotações. Esta estratégia é utilizada como padrão quando nenhuma estratégia é informada.

O *framework* JiStar também permite informar um SIG como parâmetro para auxiliar na identificação de metas flexíveis operacionalizadas no código fonte. O SIG deve ser descrito no formato NFRJson (ver Seção 3.4). As operacionalizações identificadas geram um relacionamento de contribuição do tipo "help" entre o método ("meio") e a meta flexível relacionada a operacionalização identificada, conforme o SIG. O uso do SIG combinado com o uso de anotações *@Contribution* pode gerar sobreposição de relacionamentos de contribuição, ou seja, uma mesma tarefa contribuir mais de uma vez para a mesma meta flexível com diferentes tipos de contribuição.

Para gerar o modelo de metas a partir do código de fonte, deve-se executar o próprio jar do *framework* passando o caminho dos arquivos compilados do projeto (*.class), o formato de saída desejado (i.e.: -istarml, -html, ou -pistar), a estratégia de geração de modelo e o caminho para os arquivos de SIG "*.json", No exemplo da Listagem 1, em Windows 10, executamos o jar do JiStar passando o caminho dos arquivos *.class, o formato de saída ("-pistar") e o *framework* gera um arquivo chamado "*goal*_model.txt" que pode ser carregado na ferramenta PiStar.

```
1.  java -jar JiStar-1.0.jar "caminho dos arquivos .class" -pistar
```

Listagem 1 Exportação de modelo de metas i* para PiStar.

3.4. NFRJson

Visando estruturar o conhecimento acerca de métodos que operacionalizam os RNFs de consciência presentes no catálogo de consciência de *software,* em um formato leve e compreensível para engenheiros de *software* e máquinas, optamos pelo uso do formato json. Neste contexto, inspirados no iStarJson (FRANCO BEDOYA et al., 2016), que é um formato de dados leve para modelos i*, criamos o NFRJson para a representação de SIGs (CHUNG et al., 2000).

Para descrever o formato NFRJson, criamos um json *schema*22. Isto facilita a criação e validação dos SIGs. A Listagem 2 apresenta uma visão parcial do NFRJson *schema* para ilustrar os tipos de nós e elos que o *schema* permite descrever. Na prática, o engenheiro de *software* pode utilizar o *schema* em um validador de arquivos json online, como o https://www.jsonschemavalidator.net/, para validar a descrição de um SIG seguindo o nosso formato NFRJson.

```
2.  {
3.    "$schema": "http://json-schema.org/draft-07/schema#",
4.    "properties": {"diagram": {"type": "string", "name": "diagram"},
5.      "nodes": { "items": {"properties": {
6.          "id": {"name": "id"},
7.          "name": {"name": "name"},
8.          "elementType": {"enum": [ "nfr-softgoal", "operationalizing-softgoal",
9.  "claim-softgoal" ], "name": "elementType"},
10.         "topic": {"name": "topic"},
11.          "label": {"enum": [ "satisficed", "denied", "conflicting", "undetermined"],
12.           "name": "label"},
13.          "priority": {"enum": [ "critical", "dominant"],
14.          "name": "priority" },
15.          "author": {"name": "author"},
16.          "creationTime": { "name": "creationTime" }}}},
17.     "edges": {"items": {
18.       "name": "link",
19.       "properties": {
20.         "id": {"name": "id"},
21.       "source": {"name": "source"},
22.       "target": {"name": "target"},
23.       "linkType": {"enum": [ "make", "help", "hurt", "brake", "and", "or"],
24.         "name": "linkType"}}}}}
25. }
```

Listagem 2 Visão parcial NFRJson *schema*[22].

[22] https://github.com/RE-Projects/NFRJson/blob/master/NFRJsonSchema.json

Conforme o *schema* da Listagem 2, o NFRJon permite descrever os três tipos de elementos propostos no NFR framework (CHUNG et al., 2000) para compor um SIG: nfr-*softgoal operationalizing-softgoal* e *claim-softgoal*. Os elementos do tipo *NFR-softgoals* representam requisitos não funcionais. Após os desenvolvedores refinarem o conjunto inicial de metas flexíveis ("RNFs"), eles precisam encontrar soluções que satisfaçam a contento tais metas flexíveis. Estas soluções proveem operações, processos, representações de dados, estruturas, restrições e agentes que operacionalizam as metas flexíveis e, por isso, são denominadas *operationalizing-softgoal*. O "*rationale*" do projeto é representado por elementos do tipo *claim-softgoal* que permitem justificar o modo como as metas flexíveis foram priorizadas e/ou refinadas e como os componentes, que as operacionalizam, foram selecionados.

Para ilustrar o uso do NFRJson, criamos a descrição do catálogo (SIG) de consciência de *software* (CUNHA, 2014) com o uso deste formato. O resultado é apresentado na Figura 26. Representar um SIG textualmente pode ser um trabalho árduo, ainda que seja um SIG pequeno com poucas operacionalizações. Neste contexto, vislumbramos, futuramente, poder especificar o SIG graficamente em uma ferramenta como o *RE-Tools*, que é um toolkit para modelagem de requisitos em diferentes notações incluindo o NFR *framework*, e exportar o modelo resultante para o formato NRFJson.

Na Figura 26, os nós representam os *softgoals* do tipo *nfr-softgoal* que compõem o catálogo de consciência de *software* (CUNHA, 2014). Estes possuem identificadores numerados de 01 a 14. Os elos estão identificados pelos números de 15 a 27 e são do tipo "help" conforme o catálogo.

```
{
  "diagram": "software awareness",
    "nodes": [{"id":"01","name":"awareness","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"02","name":"context awareness","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"03","name":"computing environment","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"04","name":"physical environment","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"05","name":"location","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"06","name":"time awareness","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"07","name":"self-behavior","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"08","name":"goals awareness","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"09","name":"alternatives awareness","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"10","name":"functioning awareness","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"11","name":"social-context awareness","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"12","name":"norms awareness","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"13","name":"social relationships awareness","topic":"software","elementtype":"nfr-softgoal"},
    {"id":"14","name":"user awareness","topic":"software","elementtype":"nfr-softgoal"}],
  "edges": [{"id":"15","linktype":"help", "source": "02", "target": "01"},
            {"id":"16","linktype":"help", "source": "06", "target": "01"},
            {"id":"17","linktype":"help", "source": "07", "target": "01"},
            {"id":"18","linktype":"help", "source": "11", "target": "01"},
            {"id":"19","linktype":"help", "source": "03", "target": "02"},
            {"id":"20","linktype":"help", "source": "04", "target": "02"},
            {"id":"21","linktype":"help", "source": "05", "target": "02"},
            {"id":"22","linktype":"help", "source": "08", "target": "07"},
            {"id":"23","linktype":"help", "source": "09", "target": "07"},
            {"id":"24","linktype":"help", "source": "10", "target": "07"},
            {"id":"25","linktype":"help", "source": "12", "target": "11"},
            {"id":"26","linktype":"help", "source": "13", "target": "11"},
            {"id":"27","linktype":"help", "source": "14", "target": "11"}]
}
```


Figura 26 Catálogo de consciência de *software* descrito com NFRJson.

4
Reengenharia de Sistemas Autoadaptativos Guiada pelo RNF de Consciência de *Software*

Neste capítulo, detalhamos a estratégia de reengenharia de sistemas autoadaptativos guiada pelo RNF de consciência de *software*, através da qual realinhamos sistemas autoadaptativos com as práticas recomendadas pela literatura (i.e.: engenharia de requisitos orientada a metas e o padrão arquitetural MAPE), auxiliando na remoção dos problemas recorrentes de implementação do MAPE apresentados na Seção 2.3.2.

4.1.
Visão Geral

A nossa estratégia de reengenharia é uma instância do meta processo de reengenharia proposto por Leite (LEITE, 1996), sendo composta também por quatro subprocessos principais a saber: **Recuperar**, **Especificar**, **Redesenhar** e **Reimplementar (ver** Figura 19**)**. A Figura 27 apresenta o SADT da estratégia de reengenharia guiada pelo RNF de consciência de *software*. Fizemos uso do modelo SADT (ROSS; SCHOMAN, 1977) para descrever os subprocessos presentes em nossa estratégia, assim como suas entradas e saídas, pois este permite uma decomposição hierárquica dos subprocessos. Em resumo, em nosso modelo SADT, as caixas representam nossos subprocessos, as setas à esquerda as entradas requeridas, as setas para baixo os controles, as setas para cima os mecanismos e as setas à direita as saídas da geradas.

Conforme a Figura 27, a estratégia, que será detalhada mais adiante, tem início no subprocesso **recuperar** (A1), onde a equipe de recuperação do *software* realiza a análise do código fonte do sistema alvo da reengenharia com o objetivo de gerar um modelo i* *AS-IS* do sistema com seus RNFs de consciência operacionalizados. A recuperação é guiada pelo método de recuperação e a identificação dos RNFs de consciência é realizada com o apoio do SIG de consciência de *software* (CUNHA, 2014) instanciado para a linguagem de programação do sistema alvo no

formato NFRJson. O modelo gerado pode ser enriquecido, com base em documentos existentes (e.g.: arquitetura e requisitos) ou conhecimento tácito vindo de especialistas do sistema, através da adição de metainformações de código (i.e.: framework JiStar) para aproximar ainda mais o modelo i* *AS-IS* da intencionalidade do sistema. O modelo de metas recuperado é utilizado pela equipe de especificação do *software* no subprocesso **especificar** (A2) como base para especificar os RNFs de consciência de *software* através do uso do nosso conjunto de *SRconstructs* para a especificação do RNF de consciência em modelos i* e em conformidade com o método de especificação. Neste subprocesso, também podem ser adicionados novos requisitos de consciência seguindo o mesmo método de especificação. Este novo modelo de metas é o modelo i* *TO-BE* que será utilizado pela equipe de *design* do *software* no subprocesso **redesenhar** (A3) para fazer o novo desenho da solução. Ao redesenhar, a equipe identifica as tecnologias necessárias para futuramente reimplementar os elementos do padrão MAPE, conforme as caraterísticas dos elementos (i.e.: sensor, monitor, analisador, planejador, executor e atuador) e a linguagem de programação escolhida. Guiada pelo método de redesenho, a equipe identifica componentes que possam ser reutilizados para operacionalizar novos RNFs de consciência ou melhores soluções para os RNFs de consciência pré-existentes no código fonte. O conhecimento presente no SIG de consciência de *software* (CUNHA, 2014) é reutilizado para apoiar a seleção de tais operacionalizações. Este novo modelo *TO-BE* redesenhado irá auxiliar a equipe de implementação do *software* no subprocesso **reimplementar** (A4) para reimplementar o sistema, onde o código fonte será reorganizado conforme nossos *guidelines* e *templates* de classe sugeridos além da linguagem de programação escolhida. Ao longo da reimplementação recomendamos o uso das anotações de código (i.e.: JiStar) para manter a rastreabilidade entre as metas e suas operacionalizações, assim como, entre os elementos de código criados em função do modelo MAPE. Ao final deste último subprocesso, um novo código fonte é gerado com a manutenção da rastreabilidade para suas metas e RNFs de consciência.

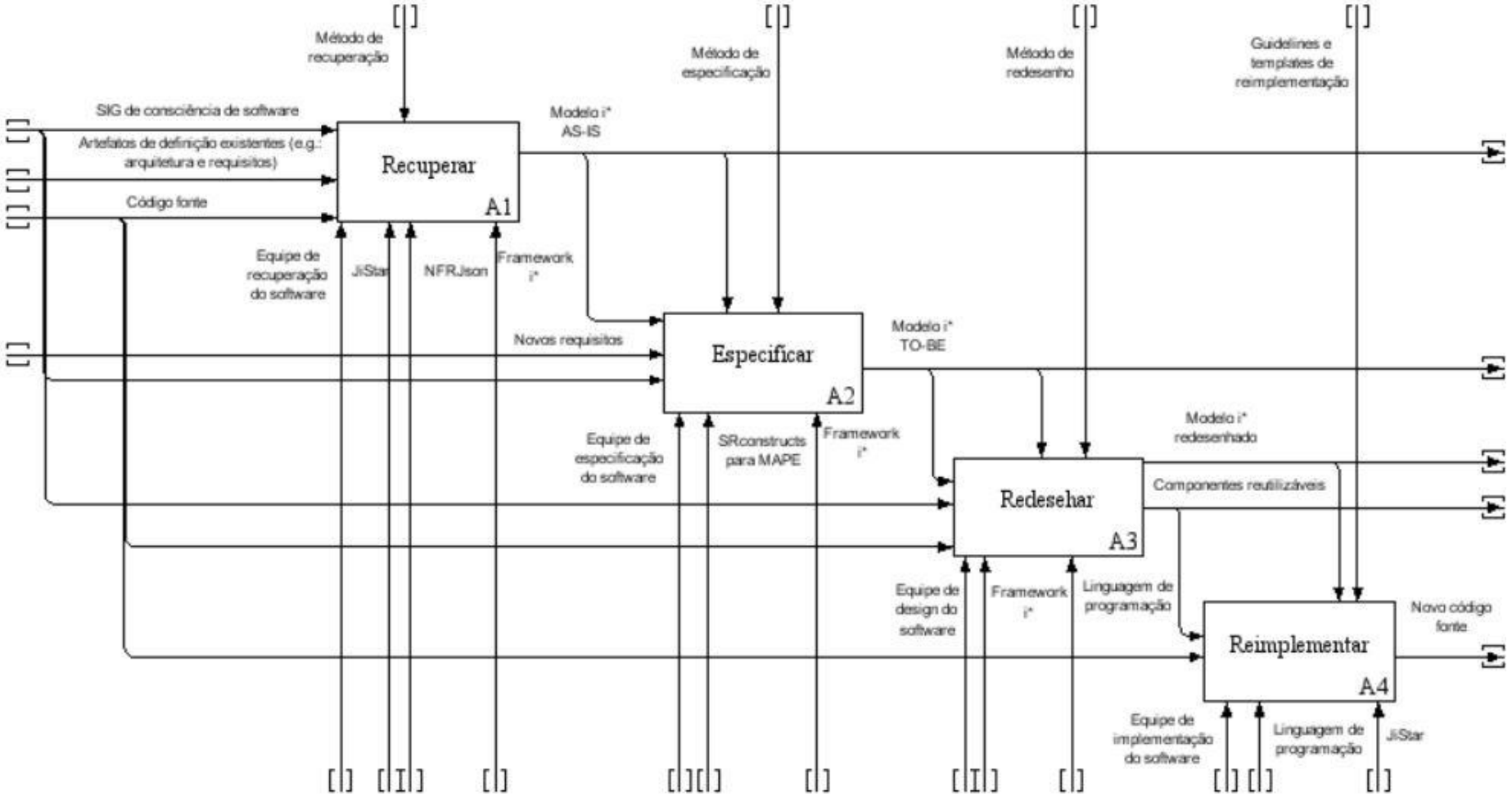


Figura 27 SADT da estratégia de reengenharia guiada pelo RNF de consciência.

A **equipe de recuperação** do *software* deve ser composta por profissionais com perfil de engenheiros ou arquitetos de *software* e estes devem conhecer a linguagem de programação do sistema alvo da reengenharia, o nosso método de recuperação, o framework i* 1.0, o framework JiStar, o formato NFRJson e o SIG de consciência de *software*. A **equipe de especificação** do *software* deve possuir profissionais com os perfis de engenheiros ou arquitetos de *software* e engenheiros de requisitos. Estes, por sua vez, devem possuir conhecimento sobre o método de especificação, framework i*, o padrão MAPE e o SIG de consciência de *software*. A **equipe de redesenho** deve ser composta por profissionais com perfil de engenheiros ou arquitetos de *software* e estes devem conhecer a linguagem de programação, o framework i*, o SIG de consciência de *software* e o método de redesenho. Por fim, a **equipe de reimplementação** do *software* deve ser composta por desenvolvedores que saibam ler modelos i*, conheçam os *guidelines* e *templates* da nossa estratégia, conheçam a linguagem de programação e o framework JiStar.

A partir desta visão geral, detalharemos cada subprocesso da nossa estratégia de reengenharia presentes na Figura 27.

4.2. Recuperar

O subprocesso Recuperar tem por objetivo obter um modelo i* *AS-IS* do sistema alvo da reengenharia.

Quatro etapas compõem o subprocesso:

(1) mapear os elementos do sistema autoadaptativo OO para um modelo i*, considerando seus elementos exceto os RNFs (i.e.: metas flexíveis ou qualidades) que são mapeados no próximo passo;

(2) mapear RNFs de consciência para o modelo de metas i*, de acordo com os métodos que estejam operacionalizando os RNFs de consciência e seu mapeamento para metas flexíveis no modelo de metas. Para auxiliar na identificação de operacionalizações, reutilizamos o catálogo de consciência de *software* elaborado por Cunha (CUNHA, 2014). O catálogo já pos-

sui sugestões de operacionalizações, mas podemos adicionar novas operacionalizações como comandos ou palavras-chave. Por exemplo, as palavras *"sensor", "accelerometer", "location", "GPS", "Bluetooth", "getTimeInstance", "getTime"* podem ser utilizadas para identificar operacionalizações de consciência;

(3) revisar o modelo de metas baseado em artefatos de definição (e.g.: documento de arquitetura, especificação de requisitos e manual do usuário), caso existam, ou com base no conhecimento tácito de especialistas do sistema;

(4) em caso de divergência de expectativas, atualize o código fonte ou reveja a lista de operacionalizações de RNF no SIG e retorne ao passo 1; Caso contrário, se o modelo de metas já está satisfatório, ele já pode ser utilizado no subprocesso especificar.

As etapas delineadas acima, são detalhadas a seguir.

(1) Mapear Sistema OO para modelo de metas i*

A primeira etapa da nossa estratégia é mapear os elementos do código de um sistema OO para um modelo de metas i*, utilizando nossas heurísticas de mapeamento a partir do paradigma OO para os elementos do *framework* i* explicitadas na Seção 3.1. Ao final desta etapa, nós temos um modelo de metas inicial onde é possível compreender o raciocínio estratégico dos atores internos do sistema e suas dependências.

(2) Mapear RNFs de consciência para o modelo de metas i*

A segunda etapa para recuperar um modelo de metas *AS-IS* é mapear os RNFs de consciência como *softgoals* no modelo i*. Para isto, realizamos uma análise do código fonte pesquisando por operacionalizações do RNF de consciência de *software*. Nesta etapa, utilizamos o catálogo de consciência de *software* (CUNHA, 2014) para auxiliar na identificação do tipo de consciência e das operacionalizações. Como este catálogo é um SIG refinado por tipo, é possível identificar os sub-*softgoals* que contribuem para a *softgoal* "consciência de *software"*. Além disso, o catálogo sugere algumas operacionalizações para implementar tais sub-*softgoals* e estas

são consideradas em nossa busca por operacionalizações no código. Entretanto, como as operacionalizações podem variar de acordo com a tecnologia a ser utilizada (e.g.: API, bibliotecas e componentes), reutilizamos as questões padrão sugeridas para cada tipo de consciência do catálogo na intenção de identificar operacionalizações mais específicas de acordo com a linguagem de programação do sistema autoadaptativo.

Para organizar as informações do catálogo e as operacionalizações em um formato compreensível para engenheiros de *software* e máquinas, optamos pelo uso do formato json, conforme descrito na Seção 3.4 do Capítulo 3.

No exemplo da Listagem 3, organizamos algumas APIs que podem operacionalizar a consciência de tempo na linguagem de programação Java. O sub-*softgoal "time awareness"* listado é proveniente do catálogo de consciência de *software* (CUNHA, 2014) e cada palavra ou comando foi escolhido com base nas perguntas relevantes para a identificação de operacionalizações no referido catálogo.

Após o preparo deste arquivo, no formato NFRJson, com as palavras e comandos que sugerem as operacionalizações, fazemos a análise do código buscando por estas palavras-chave e/ou comandos. Assim que identificamos algum dos termos, criamos o tipo de consciência como uma meta flexível no modelo de metas e ligamos à tarefa onde a operacionalização foi identificada através de um relacionamento de contribuição do tipo "*Make*". A referida tarefa já foi adicionada ao modelo na etapa anterior, conforme nossas heurísticas (ver Tabela 5). Fazemos esta busca para as operacionalizações do arquivo json, indicadas com o tipo "*operationalizing-softgoal*". Esta análise, assim como o uso das heurísticas, pode ser realizada com o uso do framework JiStar.

```
{
  "diagram": "software awareness",
  "nodes": [
{"id":"01","name":"software awareness","topic":"software","elementtype":"nfr-
softgoal"},
{"id":"02","name":"time awareness","topic":"software","elementtype":"nfr-softgoal"},
{"id":"03","name":"Calendar.getInstance","elementtype":"operationalizing-softgoal"},
{"id":"04","name":"Calendar.DAY_OF_WEEK","elementtype":"operationalizing-
softgoal"},
{"id":"05","name":"System.currentTimeMillis","elementtype":"operationalizing-
softgoal"}],
"edges": [
{"id":"06","linktype":"help", "source": "02", "target": "01"},
{"id":"07","linktype":"help", "source": "03", "target": "02"},
{"id":"08","linktype":"help", "source": "04", "target": "02"},
{"id":"09","linktype":"help", "source": "05", "target": "02"}]
}
```

Listagem 3 Exemplo de SIG para a operacionalização da consciência de tempo no formato NFR-Json.

(3) Revisar o modelo de metas i*

O modelo de metas recuperado através das heurísticas de mapeamento pode ser extenso proporcionalmente ao tamanho do sistema e de nomenclatura complexa, pois a nomenclatura dos elementos vem diretamente do código fonte. Deste modo, criamos uma terceira etapa, onde a equipe de recuperação revisa o modelo de metas *AS-IS* recuperado com base em artefatos pré-existentes sobre requisitos, arquitetura e/ou conhecimento tácito de especialistas do sistema ou arquitetos para validar o modelo resultante.

Para auxiliar nesta revisão, recomendamos a seguinte lista de perguntas para a verificação do modelo:

— Existe alguma operacionalização de consciência reconhecida pela equipe de recuperação, através de documentos de arquitetura ou por conhecimento tácito, que não tenha sido recuperada?

— Alguns elementos parecem ter nomes pouco compreensíveis?

— Precisa de um modelo mais compreensível sob o ponto de vista dos interessados?

— Existe conhecimento sobre a lógica de adaptação (e.g.: interpretação dos dados dos sensores, alternativas de adaptação, outros) que poderia estar mapeado de uma forma mais explícita?

— Existem elementos sem ligação no modelo?

Quando o modelo de metas não precisar mais de ajustes, o time de recuperação já pode passá-lo para o próximo subprocesso denominado “Especificar”. Entretanto, se a equipe de recuperação identifica que o modelo ainda não está adequado, eles devem realizar a próxima etapa para ajustar o código de modo que um novo mapeamento reflita as metas e qualidades de consciência do sistema conforme esperado pela equipe. É importante lembrar que não se deve ajustar diretamente o modelo, pois as mudanças seriam sobrepostas ao fazer uma nova análise do código.

(4) Resolver inconsistências de mapeamento

A quarta etapa será necessária somente se forem encontradas quaisquer inconsistências, no modelo i* recuperado, durante a revisão do modelo realizada na etapa (3). A ação a ser realizada para resolver as inconsistências deve ser escolhida de acordo com o tipo de inconsistência encontrada.

Operacionalização de consciência não mapeada: para ajustar a identificação de operacionalizações de consciência não identificadas com base no SIG, possuímos dois recursos: (A) o catálogo de consciência estruturado com o uso do NFR-Json pode ser atualizado com as operacionalizações não identificadas; ou (B) as anotações @Softgoal e @Contribution do framework JiStar podem ser utilizadas para indicar a meta flexível e a operacionalização que não foi mapeada na etapa (1). No caso da solução (A), adicione uma linha correspondente a operacionalização reconhecida pelo time de recuperação e crie também uma linha correspondente ao relacionamento entre esta operacionalização e o tipo de consciência relacionado. No caso da solução (B), adicione uma anotação *@Softgoal* a classe onde a operacionalização está localizada e, em seguida, adicione uma anotação *@Contribution* ao método que operacionaliza a meta flexível de consciência fazendo referência a anotação @Softgoal criada anteriormente.

As anotações *@Softgoal* e *@Contribution* possuem os seguintes atributos:

@Softgoal*(name*=”nome da meta flexível”, topic=”tópico”, description=”descrição da meta flexível”, elementType*=[NFR_SOFTGOAL | OPERA-*

TIONALIZING_SOFTGOAL | CLAIM_SOFTGOAL), label=[SATISFICED | DENIED | CONFLICTING | UNDEFINED], priority=[CRITICAL | DOMINANT | UNDEFINED], actor=”nome do ator”)*

@Contribution*(type*=[make | help | hurt | break), softgoal*=”meta flexível”)*

Ao utilizar a anotação *@Softgoal* é possível indicar e fornecer detalhes sobre uma meta flexível. Enquanto a anotação *@Contribution* é utilizada em métodos para indicar um relacionamento de contribuição do método (“meio”) para uma meta flexível. Se o método também possuir a anotação *@Task*, será criada uma tarefa conforme os dados da anotação *@Task* e esta tarefa será relacionada a meta flexível em questão conforme os dados da anotação *@Contribution*. Caso contrário, será criada uma tarefa com o nome do método e também o relacionamento para a meta flexível em questão conforme os dados da anotação *@Contribution*.

Ajustar nomenclatura e tipos de elementos mapeados: para ajustar a nomenclatura e/ou tipos dos elementos recuperados no modelo i* resultante da aplicação das heurísticas de mapeamento, deve-se adicionar as anotações de código disponíveis em nosso *framework* JiStar (MOURA; LEITE, 2020) ao código fonte, ou seja, metainformações a respeito da intencionalidade do código. Estas metainformações podem ser consideradas sempre que o código for analisado, ou seja, haverá uma rastreabilidade entre o código e a intencionalidade do sistema. Justamente para manter esta rastreabilidade, não recomendamos atualizar diretamente o modelo de metas i* *AS-IS*, pois ele poderá ficar defasado em relação ao código fonte. A anotação *@Resource*, por exemplo, pode ser utilizada para informar um nome mais amigável para um recurso.

@Resource*(name*=”nome do recurso” , actor*= “nome do ator”)*

Tornar o modelo mais compreensível sob o ponto de vista dos interessados: para tornar o modelo mais compreensível, pode-se optar por mapear somente elementos de código que estejam anotados. Neste caso, recomendamos mapear, inicialmente, os elementos das classes de interface gráfica utilizados pelos atores da organização e demais elementos que sejam identificáveis pelos interessados. Em seguida, mapear somente aqueles elementos que sejam considerados necessários

para a compreensão da intencionalidade do sistema conforme o ponto de vista dos usuários. Vários detalhes de implementação podem ser omitidos caso não sejam estritamente necessários.

Existência de conhecimento sobre a lógica de adaptação: caso exista conhecimento sobre a lógica de adaptação (e.g.: valores de referência, sintomas, alternativas de adaptação e outros) que não tenha sido claramente mapeado através das heurísticas, pode-se utilizar as anotações para evidenciar este conhecimento n modelo *AS-IS*. Isto facilitará a especificação, o redesenho e a reimplementação. Por exemplo, pode-se utilizar a anotação @Actor para criar um papel denominado "analisador" e associá-lo aos agentes que desempenham tal papel.

De modo geral, o uso das anotações auxilia a manter a rastreabilidade entre o modelo e o código, facilitando a reimplementação e permitindo expressar os conhecimentos que não são recuperados através das heurísticas. O conjunto de anotações do nosso *framework* JiStar foi apresentado na Seção 3.3.

Existência de elementos sem ligação no modelo: o mapeamento pode gerar elementos sem relacionamento no *rationale* de algum Agente do modelo de metas i* *AS-IS*. Isto pode ocorrer, por exemplo, porque o método só possui a assinatura, ou o atributo foi declarado e não é utilizado, ou existe uma anotação para meta flexível sem uma anotação de contribuição referente a ela, ou existe uma anotação para meta sem uma anotação de meios-fim referente a ela. Quando ocorrem estas situações, a equipe de recuperação deve analisar cada elemento visando complementar as anotações que faltam ou eliminar os elementos que não possuem ligação de acordo com o código.

4.3. Especificar

O subprocesso especificar tem por objetivo produzir um modelo i* *TO-BE* a partir do modelo *AS-IS* através da especificação dos RNFs de consciência de *software* em direção ao modelo MAPE. Para isto, a equipe de especificação irá detalhar o modelo de metas i* *AS-IS* com a lógica de autoadaptação, baseando-se nos RNFs

de consciência identificados. Este detalhamento deverá utilizar como guia as estruturas *SRconstructs* da Figura 24 (ver Seção 3.2 no Capítulo 3) e seguir as etapas a seguir.

(1) Especificar Sensores

Cada RNF de consciência identificado no modelo i* *AS-IS* é um sensor e precisa ter uma abstração própria, exceto nos casos em que as consciências são obtidas pelo mesmo sensor para evitar consumo demasiado de recursos. Por exemplo, o uso do sensor de GPS consome uma certa quantia de energia da bateria e deve ser utilizado com cuidado. Se há mais de uma consciência obtida a partir do sensor de GPS, deve priorizar a concentração destas em um único sensor.

Quando o agente operacionaliza mais de uma consciência a partir de diferentes sensores:

— Criar um novo agente para cada consciência adicional seguindo o *SRconstruct* da Figura 24;

— Mover os demais elementos relacionados a esta operacionalização, replicando os elementos compartilhados para posterior reorganização.

Quando o agente operacionaliza uma consciência ou mais de uma a partir do mesmo sensor, porém possui outras metas em seu "*rationale*":

— Criar um novo agente para a(s) consciência(s) seguindo o *SRconstruct* da Figura 28;

— Mover os demais elementos relacionados a esta operacionalização, replicando os elementos compartilhados para posterior reorganização.

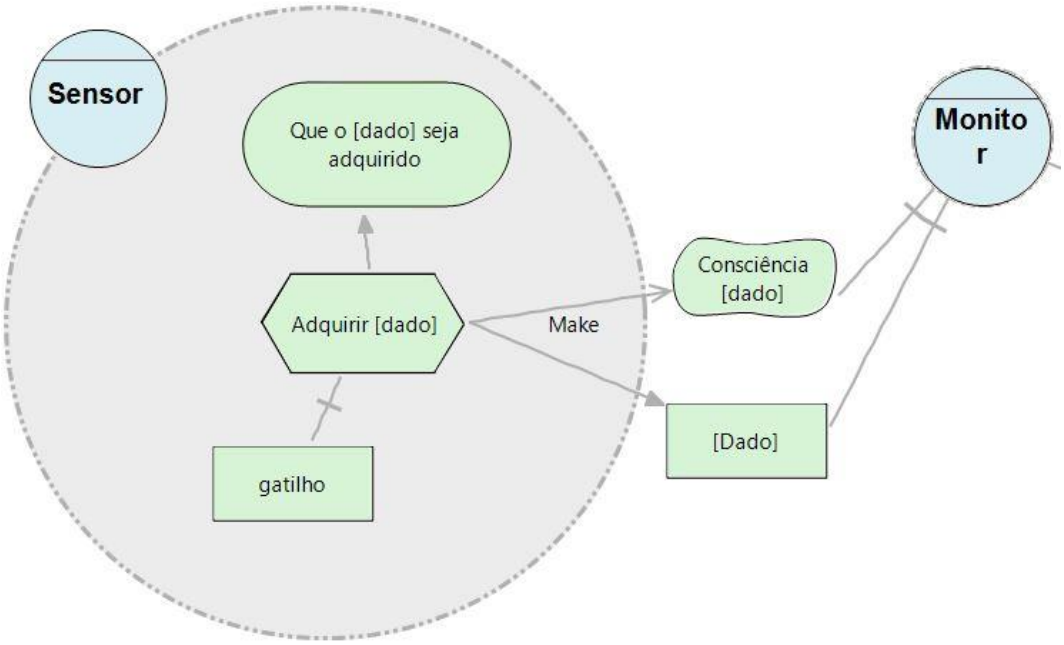


Figura 28 *SRconstruct* do agente Sensor.

(2) Especificar Monitores

Analise os agentes que recebem os dados lidos pelos sensores e realizam algum processamento sobre o dado antes de verificar os sintomas que levam a adaptação. Em seguida:

— Certifique-se de que haja um agente para monitorar cada informação que represente um log do estado do sistema. Estes agentes podem ser criados ou mantidos do modelo *AS-IS*, caso existam. É importante ressaltar que os agentes mantidos devem possuir somente a meta de "Que o [estado do sistema] seja monitorado" para que sejam reutilizados como monitores, conforme o *SRconstruct* da Figura 29;

— Mover os demais elementos relacionados à lógica de processamento dos dados dos sensores e difusão e/ou armazenamento do log do estado do sistema;

— Criar as dependências para a meta flexível de consciência e para o dado adquiridos de cada sensor referenciado pelo monitor.

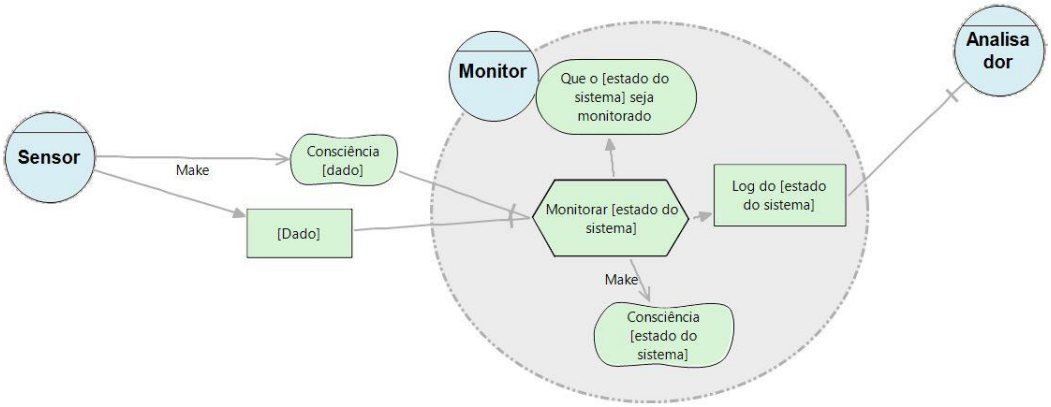


Figura 29 *SRconstruct* do agente Monitor.

(3) **Especificar Analisadores**

Analise os agentes que recebem o log do estado do sistema e realizam alguma verificação dos sintomas que levam a adaptação. Em seguida:

— Certifique-se de que haja um agente para analisar um ou mais logs do estado do sistema. Estes agentes podem ser criados ou mantidos do modelo *AS-IS*, caso existam. É importante ressaltar que os agentes mantidos devem possuir somente a meta de "Que o [sintoma] seja verificado" para que sejam reutilizados como analisadores, conforme o *SRconstruct* da Figura 30;

— Mover os demais elementos relacionados à lógica de análise dos sintomas e difusão e/ou armazenamento das requisições de adaptação;

— Criar dependência para o log do estado do sistema gerado pelo monitor.

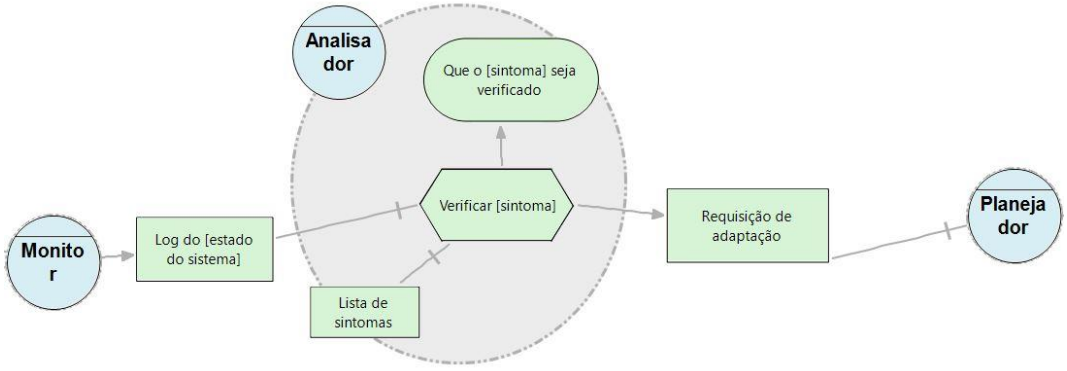


Figura 30 *SRconstruct* do agente Analisador.

(4) **Especificar Planejadores**

Analise os agentes que avaliam as alternativas de adaptação, de acordo com os sintomas identificados. Em seguida:

— Certifique-se de que haja um agente para planejar as ações de acordo com os sintomas identificados. Estes agentes podem ser criados ou mantidos do modelo *AS-IS*, caso existam. É importante ressaltar que os agentes mantidos devem possuir somente a meta de "Que uma estratégia para tratar [sintoma] seja elaborada" para que sejam reutilizados como planejadores, conforme o *SRconstruct* da Figura 31;

— Mover os demais elementos relacionados à lógica de seleção de alternativas de adaptação e difusão e/ou armazenamento da estratégia de adaptação do sistema;

— Crie dependência para a requisição de adaptação vinda do analisador.

(5) **Especificar Executores**

Analise os agentes que disparam as ações de adaptação, de acordo com a estratégia elaborada. Em seguida:

— Certifique-se de que haja um agente para controlar a execução das ações de acordo com a estratégia elaborada. Estes agentes podem ser criados ou mantidos do modelo *AS-IS*, caso existam. É importante ressaltar que os agentes mantidos devem possuir somente a meta de "Que a [estratégia] seja executada" para que sejam reutilizados como executores, conforme o *SRconstruct* da Figura 32;

— Mover os demais elementos relacionados à lógica de ordenação e disparo das ações de adaptação do sistema;

— Crie as dependências para a estratégia vinda do planejador e para meta "Que o [estado do sistema] seja atualizado" para cada atuador que será acionado.

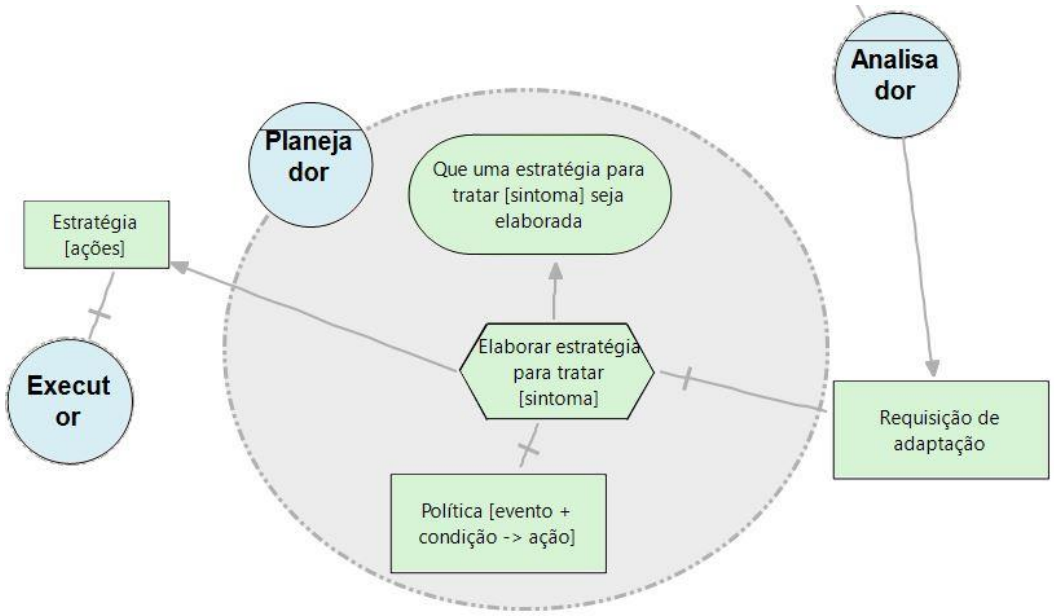


Figura 31 *SRconstruct* do agente Planejador.

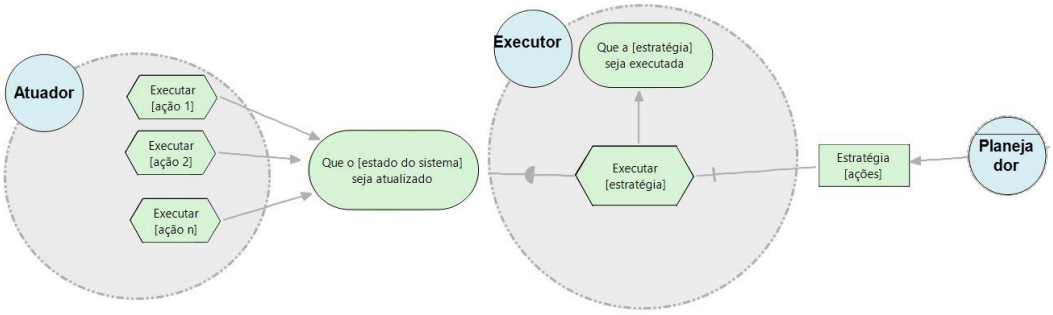


Figura 32 *SRconstructs* dos agentes Executor e Atuador.

(6) **Especificar Atuadores**

Analise os agentes que executam as ações de adaptação, de acordo com a estratégia em execução. Em seguida:

— Certifique-se de que haja um agente para executar as ações de acordo com o tipo da ação. Estes agentes podem ser criados ou mantidos do modelo *AS-IS*, caso existam. É importante ressaltar que os agentes mantidos devem possuir somente a meta de “Que o [estado do sistema] seja atualizado” para que sejam reutilizados como atuadores, conforme o *SRconstruct* da Figura 32;

— Mover os demais elementos relacionados à lógica de execução da ação de adaptação do sistema;

Ao final deste subprocesso, a equipe de especificação terá um modelo *TO-BE* especificado em direção ao padrão MAPE com as abstrações e reorganização necessárias para remover os problemas recorrentes de implementação do MAPE (ver Seção 2.3.2). Neste subprocesso, é possível também adicionar novos RNFs de consciência de *software* e estes devem ser especificados seguindo os *SRconstructs* apresentados. É importante ressaltar que, devido ao fato de que modelos em i* podem ser complexos de serem analisados, a equipe de especificação pode optar por particionar modelo seguindo as *SDsituations* que apresentamos na Seção 2.1.5.

4.4. Redesenhar

No subprocesso redesenhar, a equipe de redesenho deve: verificar, conforme a linguagem de programação a ser adotada, quais as tecnologias que serão utilizadas para implementar cada tipo de agente dos nossos *SRconstructs* especificados no subprocesso anterior.

Redesenhar agente Sensor

Uma característica importante do agente Sensor é que ele precisa executar em paralelo à aplicação (SERIKAWA et al., 2016). Deste modo, precisamos selecionar como iremos implementar esse paralelismo e adicionar esta informação ao modelo. Na linguagem Java, por exemplo, pode estender a classe *Thread* (*java.lang.Thread*) ou implementar a interface *Runnable* (*java.lang.Runnable*). Porém, quando utilizamos Java para desenvolver aplicativos para Android, podemos utilizar serviços de segundo plano estendendo a classe I*ntentService*. Segundo o guia para desenvolvedores Android[23], o serviço em segundo plano realiza uma operação que não é perceptível ao usuário.

De acordo com o tipo de solução que será utilizada para o agente Sensor, existem métodos específicos que precisam ser implementados para o seu funcionamento adequado. Por exemplo, se você está utilizando Java, independente de escolher entre estender a classe *Thread* ou implementar a interface *Runnable*, a aquisi-

[23] https://developer.android.com/guide/

ção de dados ocorrerá no método *run*. Porém, se você está desenvolvendo em Android e irá estender a class *IntetService*, você deverá realizar a aquisição de dados no método *onHandleIntent*.

Revisar operacionalizações

Como os agentes Sensores operacionalizam a consciência de dados que precisam ser adquiridos para a autoadaptação do sistema, ao redesenhá-los, deve se verificar, com o auxílio do catálogo de consciência de *software* (CUNHA, 2014), quais operacionalizações de consciência serão mantidas e quais serão adicionadas. Essas operacionalizações são utilizadas nas tarefas "Adquirir [dado]" dos agentes Sensores (ver Figura 24). Esta seleção será a lista de componentes reutilizáveis;

Redesenhar agente Monitor, Analisador, Planejador, Executor e Atuador

Os agentes Monitor, Analisador, Planejador, Executor e Atuador podem ser executados como um serviço em segundo plano paralelo à aplicação ou podem ser executados sob demanda usando o padrão de projetos *Observer* (GAMMA et al., 1993) como sugerido no design pattern para implementação do MAPE proposto por Abuseta e Swesi (ABUSETA; SWESI, 2015).

O modelo redesenhado possui detalhes técnicos, além do que já estava especificado no modelo *TO-BE*. Estes detalhes guiarão a reimplementação do sistema *TO-BE* reorganizado conforme o modelo MAPE.

4.5. Reimplementar

No subprocesso reimplementar, o sistema *TO-BE* é reorganizado com base no modelo redesenhado, no subprocesso anterior, onde existe a nova especificação e os detalhes técnicos para reimplementar seguindo o modelo MAPE. Para guiar a reimplementação, sugerimos algumas *guidelines* e *templates* de classes como proposto por Serikawa et al. (SERIKAWA et al., 2016). Caso a linguagem de programação seja a mesma grande parte do código será reaproveitado pois o nosso objetivo é reorganizar a autoadaptação com a possibilidade de adição ou remoção de RNF de consciência conforme as metas do sistema.

Reimplementação do agente Sensor

Separamos os conceitos de sensor e monitor pois um monitor pode solicitar a leitura de dados de mais de um sensor e o mesmo sensor pode fornecer dados para mais de um monitor (ABUSETA; SWESI, 2015; IBM, 2006). Uma vez que os sensores possuem a meta "Que o [dado] seja adquirido" (ver Figura 24 e Figura 28), utilizamos, parra os sensores, uma adaptação do *template* proposto em Serikawa et al. (SERIKAWA et al., 2016) para monitores.

Guidelines para implementar cada agente Sensor do modelo i* *TO-BE*:

— Verifique no modelo i* *AS-IS* qual a tarefa do agente que operacionaliza a consciência do sensor em questão;

— Identifique o código relacionado a aquisição do dado;

— Separe a lógica de aquisição, pré-processamento e difusão do dado;

— Crie uma classe conforme a classe *template* da Listagem 4. para mover o código selecionado conforme a tecnologia no modelo *TO-BE*.

```
43.  public class nomeSensor [ extends / implements ] [ Runnable /
44.  Thread / IntentServic e / . . . ] {
45.  // Atributos do sensor, frequência de execução
46.  public void onCreate ( ) {
47.  / / inicialização dos atributos do sensor
48.  }
49.  protected void [ run / onHandleIntent / . . . ] ( < parametros >) {
50.  while ( naoPara ) {
51.  / / Lógica de aquisição do dado, pré-processamento e distribuição do dado
52.
53.  try {
54.          Thread.sleep(//taxa de frequência);
55.      } catch (Exception e) {
56.      }
57.  }}
```

Listagem 4 *Template* para implementar o agente Sensor adaptado de (SERIKAWA et al., 2016).

Os agentes Monitor, Analisador, Planejador, Executor e Atuador podem ser executados sob demanda através do uso do padrão de projeto Observer (GAMMA et al., 1993; GAMMA, 1995), como sugerido por Abuseta e Swesi (ABUSETA; SWESI, 2015), ou podem ficar em execução em segundo plano. Deste modo, seus *templates* são bem similares exceto pelas tarefas "meios" específicos para alcançar a meta particular de cada agente. A seguir apresentaremos *guidelines* e *templates* de classe conforme o *SRconstruct* de cada um destes agentes.

Reimplementação do agente Monitor

O agente Monitor deseja alcançar a meta “Que o [estado do sistema] seja monitorado” (ver Figura 24 e Figura 29) e, para isso, solicita dados aos agentes Sensores. Após processar os dados recebidos, o Monitor cria um log do estado do sistema que será enviado ao Analisador.

Guidelines para implementar cada agente Monitor do modelo i* *TO-BE*:

— Verifique no modelo i* *AS-IS* qual a tarefa do agente que operacionaliza a consciência do estado do sistema em questão;

— Identifique o código relacionado a aquisição do estado do sistema;

— Separe a lógica de solicitação de dados aos sensores, processamento e difusão ou armazenamento do log de estado do sistema;

— Crie uma classe conforme classe *template* da Listagem 5 para mover o código selecionado conforme a tecnologia no modelo *TO-BE*. Na Listagem 5, o *template* é para desenvolvimento em Android onde podemos estender a classe *BroadcastReceiver* para receber informações difundidas com o uso de um objeto do tipo *Intent*.

```
58.  public class nomeMonitor [ extends / implements ] [
59.  IntentService / . . . ] {
60.  // Atributos
61.  private MyBroadcastReceiver mReceiver;
62.  ...
63.  public void onCreate ( ) {
64.  / / inicialização dos atributos do monitor
65.  mReceiver = new MyBroadcastReceiver();
66.  }
67.  protected void [onHandleIntent / . . . ] ( < parâmetros >) {
68.  while ( naoPara ) {}
69.  }
70.  public void onDestroy() {
71.       try {
72.          unregisterReceiver(mReceiver);
73.       } catch (Exception e) {... }
74.  }
75.  private class MyBroadcastReceiver extends BroadcastReceiver {
76.       @Override
77.       public void onReceive(Context c, Intent i) {
78.                 if(i.getAction().equals("dado desejado")){
79.  [Tipo] data = i.get[Tipo]Extra(nome do dado, valor padrão);
80.
81.  //Lógica de Monitoramento e [distribuição/armazenamento] do log do estado do sistema
82.       }
83.    }
84.  }
```

Listagem 5 *Template* para implementar o agente Monitor.

Reimplementação do agente Analisador

Conforme o modelo *TO-BE*, cada agente Analisador deseja alcançar a meta "Que o [sintoma] seja verificado" (ver Figura 24 e Figura 30). Para isto, o agente Analisador precisa receber a atualização do estado do sistema e verificar se existente algum sintoma que indique a necessidade de adaptação do sistema.

Guidelines para implementar cada agente Analisador do modelo i* *TO-BE*:

— Verifique no modelo i* *AS-IS* qual a tarefa do agente que operacionaliza a análise do estado do sistema em questão;

— Identifique o código relacionado a análise do estado do sistema;

— Separe a lógica de análise do estado, acesso à lista de sintomas e difusão ou armazenamento do log de estado do sistema;

— Crie uma classe conforme classe *template* da Listagem 6 para mover o código selecionado conforme a tecnologia no modelo *TO-BE*. Na Listagem 6, assim como na implementação do Monitor, o *template* é para desenvolvimento em

Android, onde podemos estender a classe *BroadcastReceiver* para receber informações enviadas através do objeto do tipo *Intent*. Após receber o log do estado do sistema, é preciso alocar a lógica de análise do estado e a consulta à lista de sintomas.

```
85.  public class nomeAnalisador [ extends / implements ] [
86.  IntentService / . . . ] {
87.  // Atributos
88.  private MyBroadcastReceiver mReceiver;
89.  ...
90.  public void onCreate ( ) {
91.  / / inicialização dos atributos do analisador
92.  mReceiver = new MyBroadcastReceiver();
93.  }
94.  protected void [onHandleIntent / . . . ] ( < parâmetros >) {
95.  while ( naoPara ) {}
96.  }
97.  public void onDestroy() {
98.      try {
99.         unregisterReceiver(mReceiver);
100.     } catch (Exception e) { ... }
101. }
102. private class MyBroadcastReceiver extends BroadcastReceiver {
103.     @Override
104.     public void onReceive(Context c, Intent i) {
105.                 if(i.getAction().equals("log do estado do sistema")){
106. [Tipo] log = i.get[Tipo]Extra(nome do log, valor padrão);
107. ...
108.                   verifySymptoms(log);
109.     }
110.   }}
111. private void verifySymptoms(log){
112. //Lógica de análise de existência de sintoma e difusão de requisição de adaptação
113. }}
```

Listagem 6 *Template* para implementar o agente Analisador.

Reimplementação do agente Planejador

O agente Planejador deseja alcançar a meta "Que uma estratégia para tratar [sintoma] seja elaborada" (ver Figura 24 e Figura 31). Para isto, o agente Planejador precisa receber o sintoma identificado pelo agente Analisador e criar uma estratégia formada por uma ou mais ações para adaptar o sistema.

Guidelines para implementar cada agente Planejador do modelo i* *TO-BE*:

— Verifique no modelo i* *AS-IS* qual a tarefa do agente que operacionaliza a definição do conjunto de ações para adaptar o sistema;

— Identifique o código relacionado a definição das ações de adaptação;

— Separe a lógica de criação e difusão da estratégia;

— Crie uma classe conforme classe *template* da Listagem 7 para mover o código selecionado conforme a tecnologia no modelo *TO-BE*. Na Listagem 7, o *template* é para desenvolvimento em Android onde podemos estender a classe *BroadcastReceiver* para receber informações difundidas com o uso de um objeto do tipo *Intent*.

```
114.  public class nomePlanejador [ extends / implements ] [
115.  IntentService / . . . ] {
116.  // Atributos
117.  private MyBroadcastReceiver mReceiver;
118.  ...
119.  public void onCreate ( ) {
120.  / / inicialização dos atributos do planejador
121.  mReceiver = new MyBroadcastReceiver();
122.  }
123.  protected void [onHandleIntent / . . . ] ( < parâmetros >) {
124.  while ( naoPara ) {}
125.  }
126.  public void onDestroy() {
127.        try {
128.           unregisterReceiver(mReceiver);
129.        } catch (Exception e) {... }
130.  }
131.  private class MyBroadcastReceiver extends BroadcastReceiver {
132.        @Override
133.        public void onReceive(Context c, Intent i) {
134.                  if(i.getAction().equals("requisição de adaptação")){
135.  [Tipo] symptom = i.get[Tipo]Extra(nome do sintoma, valor padrão);
136.  ...
137.                    createStrategy(symptoms);
138.        }
139.     }}
140.  private void createStrategy(symptoms){
141.  //Lógica de criação e difusão da estratégia (ações) para adaptação do sistema
142.  }}
```

Listagem 7 *Template* para implementar o agente Planejador.

Reimplementação do agente Executor

O agente Executor deseja alcançar a meta "Que a [estratégia] seja executada" (ver Figura 24 e Figura 32). Para isto, o agente Executor precisa receber a estratégia criada pelo agente Planejador e solicitar a execução de cada ação da estratégia aos atuadores específicos.

Guidelines para implementar cada agente Executor do modelo i* *TO-BE*:

— Verifique no modelo i* *AS-IS* qual a tarefa do agente que operacionaliza as ações para adaptar o sistema;

— Identifique o código relacionado a execução das ações de adaptação;

— Separe a lógica da adaptação propriamente, para usar no agente Atuador, da orquestração das ações;

— Crie uma classe conforme classe *template* da Listagem 8 para mover o código selecionado conforme a tecnologia no modelo *TO-BE*. Na Listagem 8, o *template* é para desenvolvimento em Android onde podemos estender a classe *BroadcastReceiver* para receber informações difundidas com o uso de um objeto do tipo *Intent*. No contexto do agente Executor, ele deve receber a estratégia enviada pelo agente Planejador.

```
143.  public class nomeExecutor [ extends / implements ] [
144.  IntentService / . . . ] {
145.  // Atributos
146.  private MyBroadcastReceiver mReceiver;
147.  ...
148.  public void onCreate ( ) {
149.  / / inicialização dos atributos do executor
150.  mReceiver = new MyBroadcastReceiver();
151.  }
152.  protected void [onHandleIntent / . . . ] ( < parâmetros >) {
153.  while ( naoPara ) {}
154.  }
155.  public void onDestroy() {
156.        try {
157.           unregisterReceiver(mReceiver);
158.        } catch (Exception e) {... }
159.  }
160.  private class MyBroadcastReceiver extends BroadcastReceiver {
161.        @Override
162.        public void onReceive(Context c, Intent i) {
163.                  if(i.getAction().equals("estrategia")){
164.  [Tipo] strategy = i.get[Tipo]Extra(nome da estrategia, valor padrão);
165.  ...
166.                    executeStrategy(estrategia);
167.        }
168.     }}
169.  private void executeStrategy(strategy){
170.  //Lógica de orquestração das ações e acionamento dos atuadores para fazer a adaptação
do sistema
171.  }}
```

Listagem 8 *Template* para implementar o agente Executor.

Reimplementação do agente Atuador

Conforme modelo *TO-BE*, o agente Atuador deseja alcançar a meta "Que o [estado do sistema] seja atualizado" (ver Figura 24 e Figura 32). Para isto, o agente

Atuador possui diferentes meios de ajustar o estado do sistema. Porém, para determinar qual meio (“ação”) deve ser executado, ele precisa receber do agente Executor qual ação deve ser executada.

Guidelines para implementar cada agente Atuador do modelo i* *TO-BE*:

— Verifique no modelo i* *AS-IS* qual a tarefa do agente que executa as ações para adaptar o sistema;

— Identifique o código relacionado a execução das ações de adaptação;

— Separe a lógica da ação de adaptação propriamente;

— Crie uma classe conforme classe *template* da Listagem 9 para mover o código selecionado conforme a tecnologia no modelo *TO-BE*. Na Listagem 9, o *template* é para desenvolvimento em Android onde podemos estender a classe *BroadcastReceiver* para receber informações difundidas com o uso de um objeto do tipo *Intent*. No contexto do agente Executor, ele deve receber a estratégia enviada pelo agente Planejador.

```
172.  public class nomeAtuador [ extends / implements ] [IntentService / . . . ] {
173.  // Atributos...
174.  private MyBroadcastReceiver mReceiver;
175.  public void onCreate ( ) {
176.  / / inicialização dos atributos do Atuador
177.  mReceiver = new MyBroadcastReceiver();}
178.  protected void [onHandleIntent / . . . ] ( < parâmetros >) {
179.  while ( naoPara ) {}}
180.  public void onDestroy() {
181.        try {
182.           unregisterReceiver(mReceiver);
183.        } catch (Exception e) {... }
184.  }
185.  private class MyBroadcastReceiver extends BroadcastReceiver {
186.        @Override
187.        public void onReceive(Context c, Intent i) {
188.                  if(i.getAction().equals(“ação”)){
189.  String action = i.getStringExtra(“ação”, valor padrão);
190.  Switch(action){
191.  Case “ação 1”:{ action1();break;}
192.  ...
193.  default:{}... }}}
194.  private void action1(){
195.  //Lógica de execução da ação para fazer a adaptação do sistema}}
```

Listagem 9 *Template* para implementar o agente Atuador.

Ao final desta etapa, o novo sistema deve estar implementado conforme o padrão MAPE e sem os problemas de implementação recorrentes na literatura. É

importante ressaltar que os *templates* aqui listados são sugestões e variam de acordo com a linguagem de programação a ser utilizada.

4.6. Comparação entre Abordagens

A nossa estratégia visa realinhar sistemas autoadaptativos as boas práticas recomendadas pela literatura, GORE no contexto da engenharia de requisitos e MAPE no contexto de arquitetura. A Tabela 8 resume brevemente um comparativo entre a nossa estratégia e cada trabalho relacionado na Seção 2.5.

Tabela 8 Comparativo entre abordagens.

Trabalho	Engenharia reversa para modelos de metas	Arquitetura do produto final	Suporte para identificação dos RNFs de consciência de *software*	Abrangência de funções do MAPE
(YU et al., 2005)	Sim (framework GSP).	Web *service* e/ou componentes.	Não.	Não usa MAPE.
(GARLAN; SCHMERL; CHENG, 2009)	Não.	*Rainbow.*	Não.	Não usa MAPE.
(SERIKAWA et al., 2016)	Não.	MAPE.	Sim, através de *guidelines*.	Monitor.
(SAN MARTÍN et al., 2020)	Não.	MAPE.	Não.	Base de conhecimento, Executores e Atuadores.
Nossa Estratégia	Sim (framework i*).	MAPE.	Sim, através do uso de SIG e metainformações de código.	Sensor, Monitor, Analisador, Planejador, Executor, Atuador e Base de conhecimento.

Para comparar nossa estratégia com as abordagens de reengenharia correlatas (Seção 2.5), consideramos relevante abordar os seguintes aspectos: a **engenharia reversa para modelos de metas,** uma vez que a reengenharia geralmente inclui alguma forma de engenharia reversa (para alcançar uma descrição mais abstrata); a **arquitetura do produto final**, uma vez que a etapa de engenharia reversa é seguida de alguma forma de engenharia avante ou reestruturação; o **suporte para identificação dos RNFs de consciência de *software*** operacionalizados no código, pois este tipo de RNF está diretamente relacionado a função de monitoramento do

MAPE, e; quanto a Abrangência de funções do MAPE que são reorganizadas com o uso da abordagem.

Comparando a nossa estratégia as abordagens de reengenharia relacionadas na Seção 2.5, quanto à engenharia reversa para modelos de metas, além do nosso trabalho, somente o trabalho de (YU et al., 2005) possibilita a obtenção de um modelo de metas a partir de sistemas legados usando o framework GSP (*Goals, Skills and Preferences*) (HUI; LIASKOS; MYLOPOULOS, 2003) como linguagem para o modelo metas. Neste aspecto, nosso trabalho difere por realinhar sistemas legados com modelos de metas em i* que, por sua intencionalidade distribuída, auxilia a representação da modularidade existente e o redesenho da modularidade conforme o MAPE.

Quanto a arquitetura do produto final, o trabalho de Yu et al. tem por objetivo gerar serviços para serem reutilizados sob diferentes formas como web *services* e componentes.

O trabalho de Garlan et al. (GARLAN; SCHMERL; CHENG, 2009) oferece um framework próprio para a adoção de características de autoadaptação denominado *Rainbow*, apesar deste framework possuir elementos que contemplam as funções do MAPE, este framework realiza a autoadaptação com base na arquitetura do sistema que está sendo gerenciado.

O trabalho de Serikawa et al. (SERIKAWA et al., 2016) e San Martín et al. (SAN MARTÍN et al., 2020), assim como o nosso trabalho, têm como objetivo que o produto final seja aderente ao padrão arquitetural MAPE.

Quanto ao suporte a identificação dos RNFs de consciência de *software*, além do nosso trabalho, encontramos algum suporte no trabalho de Serikawa et al. (SERIKAWA et al., 2016), o qual é feito com base em *guidelines* para buscar sensores disponíveis na API da linguagem de programação em específico. A nossa estratégia para identificar RNFs de consciência de *software* se difere por ser baseada em SIG, o que permite identificar o tipo do RNF de consciência além da sua operacionalização. Além disso, possibilitamos que o engenheiro de *software* indique operacionalizações próprias que não façam parte de uma API específica através do uso de metainformações de código.

Quanto a abrangência de funções do MAPE que são reorganizadas, os trabalhos de Yu et al. (YU et al., 2005) e Garlan et al. (GARLAN; SCHMERL; CHENG, 2009) não geram produtos aderentes ao padrão MAPE. O Trabalho de Serikawa et al. (SERIKAWA et al., 2016) reorganiza os sistemas autoadaptativos no que se refere a função monitor do MAPE por tratar a remoção dos problemas as arquiteturais “Monitores Oprimidos” e “Monitor Obscuro”. O trabalho de San Martín et al. (SAN MARTÍN et al., 2020) reorganiza duas informações da base de conhecimento (i.e.: entradas de referência e alternativas de adaptação) e os executores e atuadores por tratar da remoção dos problemas arquiteturais “Entradas de referência dispersas”, “Executores e Atuadores Mistos” e “Alternativas Obscuras”. Nossa estratégia reorganiza as funções sensor, monitor, analisador, planejador, executor, atuador e a base de conhecimento no que se refere a log de estados do sistema, sintomas e políticas.

5
Aplicação da Estratégia de Reengenharia

Neste capítulo, apresentamos a aplicação da nossa estratégia de reengenharia de sistemas autoadaptativos guiada pelo requisito não funcional de consciência de *software*.

A estratégia é aplicada parcialmente em um primeiro aplicativo autoadaptativo *RioBus* visando recuperar um modelo de metas i* *AS-IS* do sistema.

Em um segundo momento, nós conduzimos a reengenharia do aplicativo *PhoneAdapter*. Escolhemos este segundo aplicativo por ele ter sido analisado nos trabalhos de Serikawa et al. (SERIKAWA et al., 2016) e San Martín et al. (SAN MARTÍN et al., 2020) e possuir os problemas de implementação do MAPE mencionados anteriormente. Deste modo, fizemos um benchmarking[24] comparando a reorganização do sistema *PhoneAdapter* feita sob a nossa estratégia e sob as estratégias de Serikawa et al.(SERIKAWA et al., 2016) e San Martín et al. (SAN MARTÍN et al., 2020) à luz dos atributos de qualidade afetados pelos problemas de implementação. Com relação ao *PhoneAdapter*, analisamos ainda o impacto na realização de algumas tarefas de manutenção no sistema original e no sistema reorganizado conforme a nossa estratégia de reengenharia. Ao longo da avaliação dos resultados obtidos, demonstramos que nossas hipóteses são válidas.

5.1.*RioBus*

O ***RioBus***[25] é um serviço online de monitoramento de ônibus das linhas municipais da cidade do Rio de Janeiro, RJ, Brasil. Os dados são oferecidos publicamente pela prefeitura do Rio de Janeiro, em parceria com a FETRANSPOR e Iplanrio. As posições dos ônibus são recuperadas pelos dispositivos de GPS embarcados,

[24] Benchmarking é o processo de execução de dois ou mais sistemas com a finalidade de comparar as características desses sistemas, principalmente, o desempenho. (TRAVASSOS; GUROV; AMARAL, 2002)

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

enviadas para a FETRANSPOR e, por fim, a Iplanrio as disponibiliza na página do projeto de dados abertos Data.Rio.

Este serviço foi desenvolvido para diferentes clientes nas tecnologias Android, Ionic, JavaScript, IOS, entre outras. Nesta aplicação do subprocesso recuperar, nós escolhemos a versão do RioBus em Android.

A fim de ilustrar o uso do aplicativo *RioBus*, a Figura 33 mostra exemplos de uso do aplicativo. O primeiro exemplo é a tela de seleção de linha de ônibus específica, onde é possível visualizar uma lista com as linhas de ônibus da cidade do Rio de Janeiro. No segundo exemplo, após selecionar a linha 483, os ônibus desta linha são apresentados no mapa com um marcador para cada ônibus. No terceiro exemplo, selecionamos um dos marcadores e as informações (e.g.: velocidade, direção e última atualização) do ônibus representado por este marcador são exibidas. Por fim, no quarto exemplo, é apresentada a localização do usuário no mapa e os ônibus próximos aos usuários.

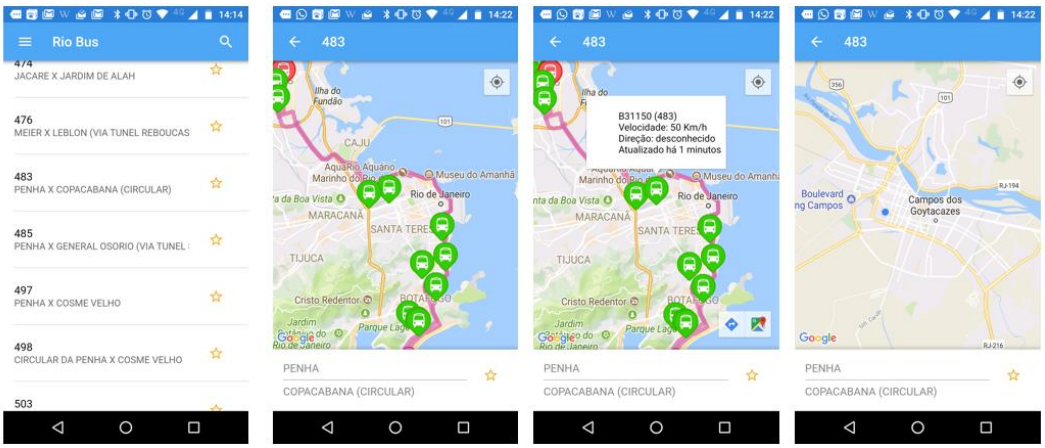


Figura 33 Telas do aplicativo RioBus em Android.

5.1.1.Recuperação de modelo de metas i* *AS-IS*

O código do aplicativo *RioBus* está organizado em seis pacotes e mais duas classes externas aos pacotes, conforme Figura 34.

No pacote *adapter*, existe a classe *BusInfoWindowAdapter* que é responsável por gerar o conteúdo da caixa de diálogo de informação de um ônibus escolhido pelo usuário no mapa. No pacote *asyncTasks*, existe a classe *BusSearchTask* que utiliza um objeto do tipo *HttpService*, presente no pacote *service*, para recuperar os

dados de itinerário e geolocalização dos ônibus e passa-la para classes que do tipo *BusDataReceptor*. No pacote *exception*, existem duas classes de exceção. No pacote *common*, existem duas classes de utilidade (i.e.: *checkInternetConnection* e *Util*) e a interface *BusDataReceptor*. No pacote model, temos quatro classes que representam entidades (i.e.: *Bus, BusData, Itinerary* e *Spot*) além da classe *MapMarker* que possui a responsabilidade de marcar o itinerário dos ônibus e a localização do usuário no mapa.

Por fim, o projeto possui a classe *MainActivity*, que é responsável por controlar a interação do usuário com a aplicação e exibir o mapa com as localizações de itinerários e usuário, e a classe *EnvironmentConfig* que é responsável por manter a informação do endpoint do serviço rest de onde são obtidos os dados de geolocalização dos ônibus.

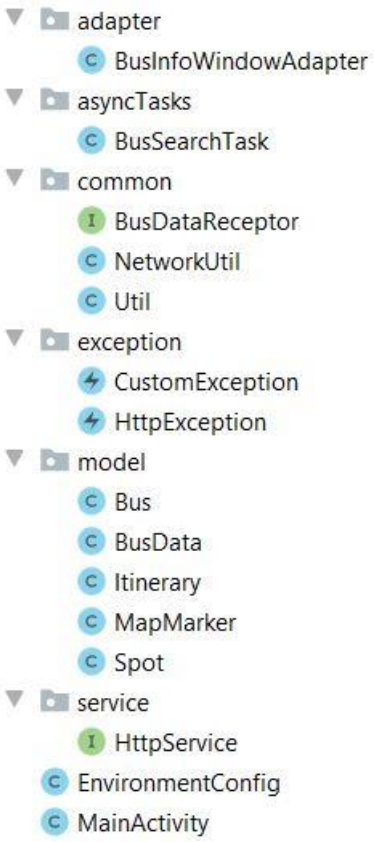


Figura 34 Classes e interfaces do aplicativo RioBus em Android.

Nesta aplicação do subprocesso recuperar, que é o primeiro subprocesso da nossa estratégia de reengenharia, fizemos a recuperação do modelo de metas i* *AS-IS* com vistas a realinhar o aplicativo com um modelo de metas i*, incluindo as metas flexíveis de consciência operacionalizadas.

Ao aplicar as etapas do subprocesso “recuperar”, obtivemos o modelo da Figura 35. É importante ressaltar que fizemos uso das heurísticas combinadas com anotações de código do framework JiStar em algumas partes, para que eventos de tela, assim como algumas nomenclaturas, ficassem mais fáceis de serem compreendidos no modelo. As anotações utilizadas foras *@Actor, @Goal, @Softgoal, @Task, @Contribution e @MeansEnd*.

5.1.2.Avaliação

Dado que realizamos a recuperação do modelo de metas i* *AS-IS* do aplicativo *RioBus,* a partir deste momento relatamos a verificação a nossa hipótese 1 acerca do reuso do catálogo de consciência de *software* para identificar metas flexíveis de consciência a partir das operacionalizações no código.

Hipótese 1: reutilizar meta conhecimento de consciência de *software* pode auxiliar na identificação de metas flexíveis e operacionalizações de consciência em sistemas autoadaptativos.

Com vistas a avaliar a nossa estratégia sob a perspectiva da hipótese 1, recuperamos um modelo de metas i* *AS-IS* do sistema autoadaptativo real chamado *RioBus* com as metas flexíveis de consciência operacionalizadas no código.

Conforme mostra a Figura 35, demonstramos que identificamos as seguintes consciências, a saber: localização para tópico “usuário”; localização para o tópico “ônibus”, e; consciência de ambiente computacional para o tópico “conexão de internet”. As consciências de localização do usuário e de conexão com a internet puderam ser identificadas mais facilmente por serem operacionalizadas por APIs que adicionamos ao catálogo de consciência no formato NFRJson, mas consciência de localização dos ônibus precisou ser explicitada via anotações e o conhecimento do catálogo foi reutilizado para identificar o tipo de meta flexível que usamos na anotação.

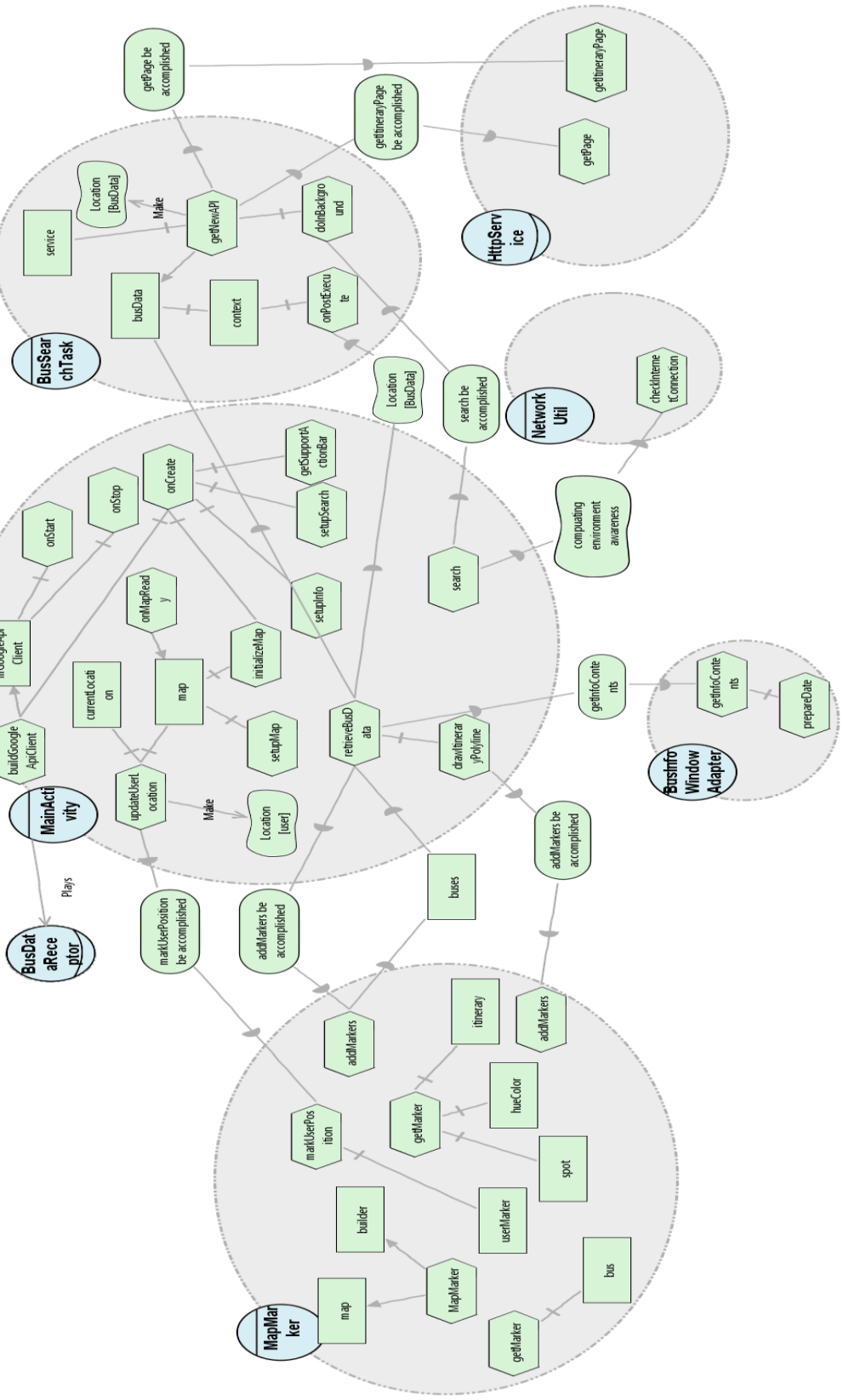
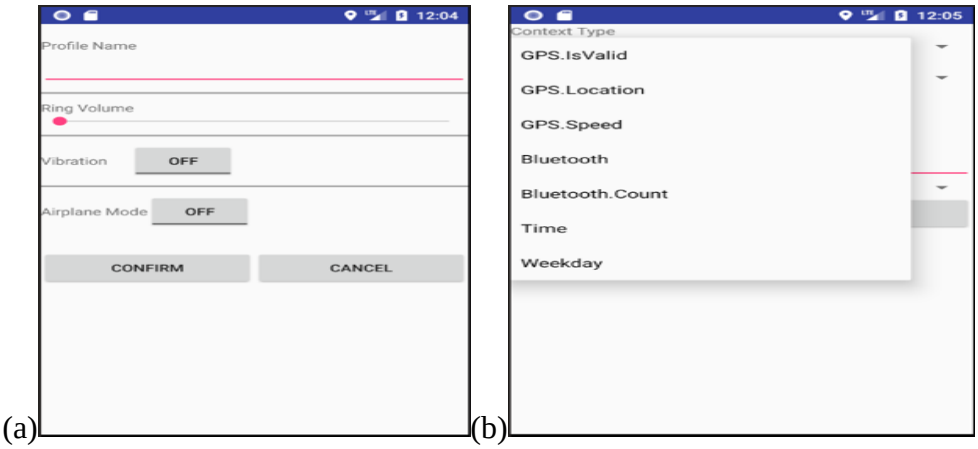


Figura 35 Modelo de metas *AS-IS* do aplicativo *RioBus* em Android.

5.2. *PhoneAdapter*

O ***PhoneAdapter***[26] é um aplicativo Android capaz de adaptar o perfil do smartphone de acordo com condições de contexto a serem selecionadas pelo usuário. As condições de contexto podem ser monitoradas pelos diferentes sensores existentes nos smartphones. Como por exemplo, o smartphone pode auxiliar um usuário a alternar para o modo silencioso quando ele chega ao trabalho e retorno ao modo campainha quando ele retornar para casa. A Figura 36 a apresenta as opções de adaptação presentes no *PhoneAdapter* que são: volume, vibração e modo avião, enquanto a Figura 36 b mostra as condições de contexto (i.e.: disponibilidade de GPS, velocidade, localização, dispositivos Bluetooth, tempo e dia da semana) as quais podem ser monitoradas sozinhas ou em conjunto com vistas a disparar as adaptações selecionadas pelo usuário.



(a) (b)

Figura 36 a) tela de criação de perfil de adaptação b) tela de cadastro de filtro de contexto para definir momento de adaptação.

5.2.1. Reengenharia

O código do aplicativo *PhoneAdapter* está organizado em três pacotes: *activity*, *context* e *database*. A Figura 37 mostra o conjunto de classes que compõem cada pacote do aplicativo. As classes do pacote *activity* são atividades que repre-

[26] https://github.com/Advanse-Lab/*PhoneAdapter*/releases/tag/v1.5

sentam os cadastros básicos do aplicativo exceto pela classe *MainActivity* que inicializa os serviços de segundo plano responsáveis pelo mecanismo de autoadaptação. As classes do pacote *context* são os serviços de segundo plano responsáveis pelo mecanismo de autoadaptação (i.e.: *ContextManager e AdaptatioManager*) e interfaces que definem valores constantes utilizados no mecanismo de autoadaptação (i.e.: *ContextName, ContextOperator e ContextType*). Por fim, as classes do pacote *database* são responsáveis pela criação e atualização da base de dados do aplicativo (i.e.: *MyDbAdapter* e *MyDbHelper*).

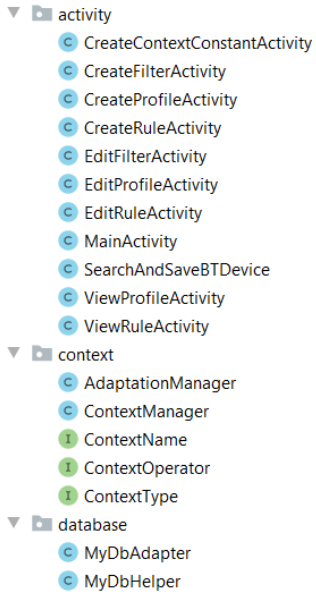


Figura 37 Classes e interfaces do aplicativo *PhoneAdapter*.

Serikawa et al. (SERIKAWA et al., 2016) identificaram dois problemas neste aplicativo: **Monitores Oprimidos** e **Monitor Obscuro** (ver Seção 2.3.2). Os dois problemas foram identificados na classe *ContextManager* que é responsável pela aquisição e difusão dos dados, onde obriga os sensores a coletarem dados na mesma frequência além deixar a lógica de monitoramento, pré-processamento e difusão dos dados dispersa sem a caracterização do monitor propriamente. Enquanto San Martín et al. (SAN MARTÍN et al., 2020) identificaram o problema **Executores e Atuadores Mistos** neste mesmo aplicativo, porém em local diferente, no método *onReceive* da classe *MyBroadcastReceiver*. Esta classe é uma classe interna à classe *AdaprationManager*. Deste modo, nossa estratégia de reengenharia pôde ser aplicada nas classes do pacote *context*.

Ao aplicar o subprocesso recuperar da nossa estratégia, obtivemos o modelo da Figura 38 com cinco agentes e três papéis responsáveis pela lógica de adaptação.

Apesar de aparentemente analisarmos somente duas classes, cada uma das classes analisadas possui classes internas, a classe *ContextMananger* possui as classes *MyBroadcastReceiver* e *MyLocationListener*. Enquanto a classe AdaptationManager possui a classe interna *MyBroadcastReceiver*.

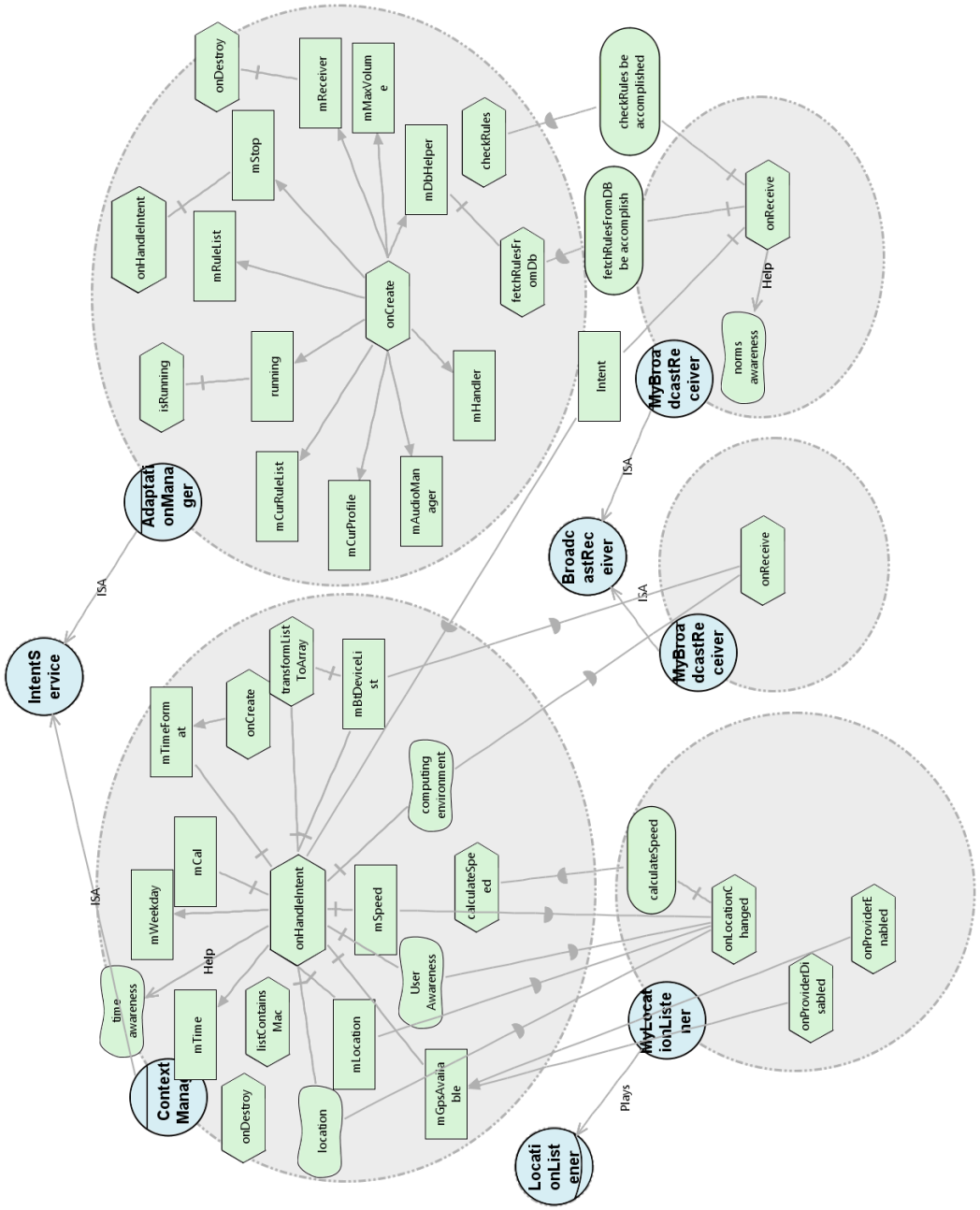


Figura 38 Modelo de metas *AS-IS* em i* do *PhoneAdapter*.

Após a recuperação do modelo i* *AS-IS*, vimos que um mesmo agente (i.e.: *ContextManager*) concentrava mais de um RNF de consciência operacionalizado (i.e.: problema monitores oprimidos) e identificamos uma operacionalização de consciência de norma no agente *MyBroadcastReceiver* em meio ao planejamento e

execução das adaptações (i.e.: problema monitor obscuro, executores e atuadores mistos, alto acoplamento entre as funções de análise, planejamento e execução do MAPE). Devido a estes problemas, confirmamos a necessidade de reorganizar o sistema e iniciamos o subprocesso especificar e geramos o modelo i* *TO-BE* da Figura 39. Em seguida, redesenhamos este modelo para selecionar as tecnologias que seriam adotadas para a reimplementação do aplicativo. Esta etapa foi muito importante para definirmos as características técnicas dos agentes e revisar as operacionalizações de consciência utilizadas. O novo modelo *TO-BE* com detalhes técnicos é apresentado na Figura 40.

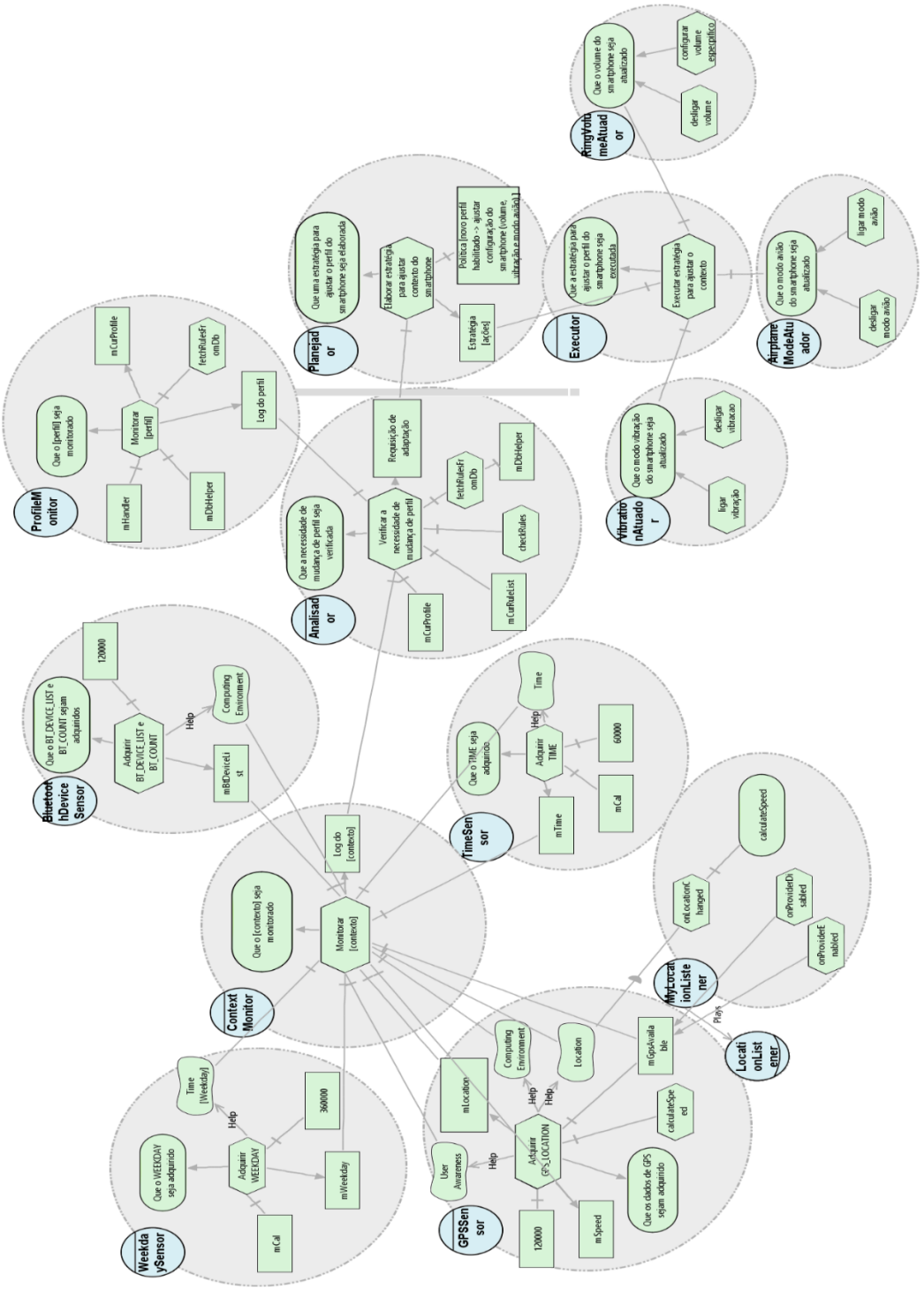


Figura 39 Modelo i* *TO-BE* após especificação do *PhoneAdapter*.

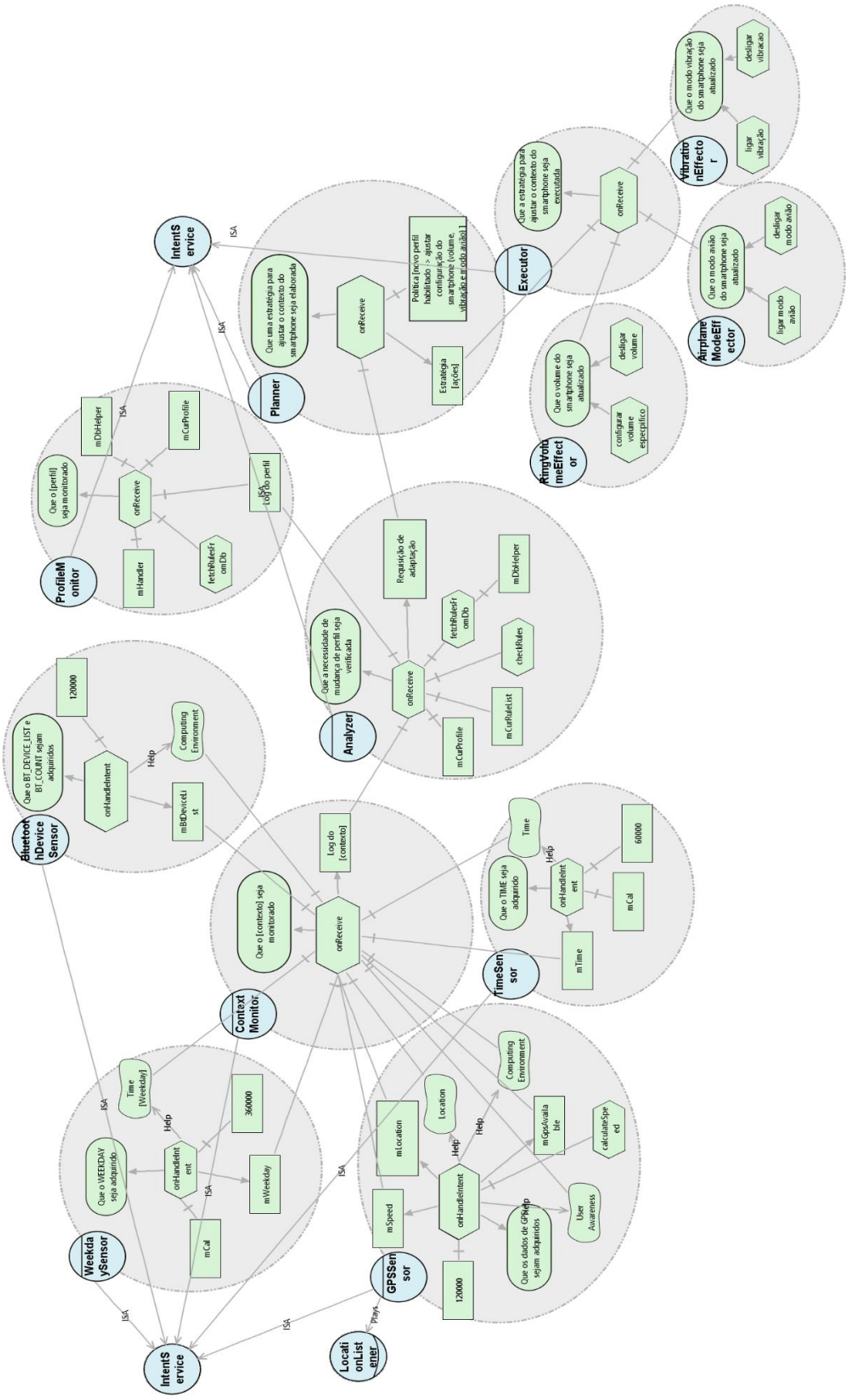


Figura 40 Modelo i* *TO-BE* redesenhado.

Após redesenhar o aplicativo, reimplementamos o mesmo seguindo as escolhas técnicas realizadas no modelo i* *TO-BE* redesenhado da Figura 40, além dos *guidelines* e *templates* de classe sugeridos em nossa estratégia. A seguir, apresentamos um trecho do código do sensor de tempo, classe *TimeSensor* (Listagem 10). Este sensor de tempo adquire o novo valor do tempo a cada um minuto. Conforme o modelo i* *TO-BE* da Figura 40, implementamos também os sensores GPSSensor, BluetoothDeviceSensor e WeekdaySensor (ver Apêndice 1).

```
196.  @Actor(name = "TimeSensor",type = ActorType.AGENT)
197.  @Goal(name = "That time be acquired",actor = "TimeSensor",description = "")
198.  @Softgoal(name = "Time Awareness",actor = "TimeSensor")
199.  public class TimeSensor extends IntentService {
200.
201.     //atributos
202.     ...
203.     @Override
204.  @Contribution(type = ContributionType.HELP,softgoal = "Time Awareness")
205.     @MeansEnd(end = "That time be acquired",endType = MeansEndType.GOAL)
206.     protected void onHandleIntent(Intent intent) {
207.        while(!mStop){
208.           mCal=Calendar.getInstance();
209.           mTime=mTimeFormat.format(mCal.getTime());
210.           mHandler.post(new Runnable() {
211.              @Override
212.              public void run() {
213.  /* broadcast new context*/
214.                 Intent i=new Intent();
215.                 i.setAction(ContextName.TIME);
216.                 i.putExtra(ContextName.TIME,mTime);
217.                 sendBroadcast(i);
218.  ...
219.              } });
220.           try{
221.              Thread.sleep(60000);
222.           } catch(Exception e){
223.              Log.e("PhoneAdapter.error", "Thread sleep exception");
224.           }
225.        }
226.     }
227.  ...
228.  }
```

Listagem 10 Classe *TimeSensor*.

A consciência de tempo, operacionalizada na classe *TimeSensor*, é utilizada pelo monitor de contexto do smartphone, ContextMonitor (Listagem 11). Conforme o modelo *TO-BE* (Figura 40), implementamos também o monitor de perfil denominado ProfileMonitor (ver Apêndice 1).

```
229.  @Actor(name = "ContextMonitor",type = ActorType.AGENT)
230.  @Goal(name = "That monitoring the smartphone context be accomplished",actor =
231.  "ContextMonitor",description = "")
232.  public class ContextMonitor extends IntentService  {
```

```
233.        private boolean mGpsAvailable;
234.        private String mLocation;
235.        private double mSpeed;
236.        private String mTime;
237.        private String mWeekday;
238.        private Handler mHandler;
239.        private MyBroadcastReceiver mReceiver;
240.        private boolean mStop;
241.        private static final String TAG = "PhoneAdapterContextLog";
242.        private static boolean running;
243.        private String[] deviceMacList;
244.        private int deviceCount;
245.
246.        public ContextMonitor() {
247.            super("ContextMonitor");
248.        }
249.
250.        @Override
251.        public void onCreate() {
252.            super.onCreate();
253.            mHandler = new Handler();
254.            mReceiver = new MyBroadcastReceiver();
255.            IntentFilter iFilter = new IntentFilter();
256.            iFilter.addAction(ContextName.GPS_LOCATION);
257.            iFilter.addAction(ContextName.GPS_AVAILABLE);
258.            iFilter.addAction(ContextName.GPS_SPEED);
259.            iFilter.addAction(ContextName.TIME);
260.            iFilter.addAction(ContextName.WEEKDAY);
261.            iFilter.addAction(ContextName.BT_DEVICE_LIST);
262.            iFilter.addAction(ContextName.BT_COUNT);
263.            iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
264.        ...
265.        }
266.
267.        @Override
268.        public void onDestroy() {
269.            try {
270.                unregisterReceiver(mReceiver);
271.            } catch (Exception e) {
272.            }
273.            /**stop foreground service**/
274.            stopForeground(true);
275.            ContextMonitor.running = false;
276.            super.onDestroy();
277.        }
278.
279.        @SuppressLint("LongLogTag")
280.        @Override
281.        protected void onHandleIntent(Intent arg0) {
282.            while (!mStop) {  }
283.        }
284.        private void broadcastSystemStateLog() {
285.            mHandler.post(new Runnable() {
286.                @Override
287.                public void run() {
288.                    /* broadcast new context*/
289.                    Intent i = new Intent();
290.                    i.setAction("edu.hkust.cse.PhoneAdapter.newContext");
291.                    i.putExtra(ContextName.GPS_AVAILABLE, mGpsAvailable);
292.                    i.putExtra(ContextName.GPS_LOCATION, mLocation);
```

```
293.                i.putExtra(ContextName.GPS_SPEED, mSpeed);
294.                i.putExtra(ContextName.BT_DEVICE_LIST, deviceMacList);
295.                i.putExtra(ContextName.BT_COUNT, deviceCount);
296.                i.putExtra(ContextName.TIME, mTime);
297.                i.putExtra(ContextName.WEEKDAY, mWeekday);
298.                sendBroadcast(i);
299.             }
300.          });
301.       }
302.
303.    ...
304.       private class MyBroadcastReceiver extends BroadcastReceiver {
305.
306.          @Override
307.    @MeansEnd(end = "That monitoring the smartphone context be accomplished",endType
308.    = MeansEndType.GOAL)
309.          public void onReceive(Context c, Intent i) {
310.             String action = i.getAction();
311.             switch (action) {
312.                case "edu.hkust.cse.PhoneAdapter.stopService": {
313.                   mStop = true;
314.                   stopSelf();
315.                   break;
316.                }
317.                case ContextName.GPS_LOCATION: {
318.                   mLocation = i.getStringExtra(ContextName.GPS_LOCATION);
319.                   mSpeed = i.getDoubleExtra(ContextName.GPS_SPEED,0.0);
320.                   boolean gpsAvailable=i.getBooleanExtra(ContextName.GPS_AVAILABLE,
false);
321.                   broadcastSystemStateLog();
322.                   break;
323.                }
324.                case ContextName.TIME: {
325.                   mTime = i.getStringExtra(ContextName.TIME);
326.                   broadcastSystemStateLog();
327.                   break;
328.                }
329.    ...
330.                default:{}
331.             }}}
332.    }
```

Listagem 11 Classe *ContextMonitor*.

O monitor de contexto, classe *ContextMonitor*, submete o log do estado do sistema com relação ao contexto0 do smartphone e o analisador, classe *Analyzer* (Listagem 12), verifica se alguma regra de mudança de perfil foi satisfeita pois isto indica a necessidade de adaptação. Caso haja regra de mudança de perfil satisfeita, a classe *Analyzer* envia uma requisição de adaptação que contém as regras satisfeitas. Esta requisição de adaptação é tratada pelo planejador, classe *Planner* (Listagem 13). A classe *Planner* cria uma estratégia com base na premissa de que somente uma regra de adaptação pode ser aplicada e envia a estratégia elaborada para o *Executor* (Listagem 14)..

```
@Actor(name = "Analyzer",type = ActorType.AGENT)
@Goal(name = "Analyze profile change needs be accomplished",actor =
"Analyzer",description = "")
public class Analyzer extends IntentService {
...
   private ArrayList<Rule> fetchRulesFromDb() { ... }

   private class MyBroadcastReceiver extends BroadcastReceiver {

      @Override
@MeansEnd(end = "Analyze profile change needs be accomplished",endType =
MeansEndType.GOAL)
      public void onReceive(Context c, Intent i) {
         String action = i.getAction();
         if (action.equals("edu.hkust.cse.PhoneAdapter.newContext")) {

               if (mCurRuleList.size() > 0) {
                  satisfiedRuleList = checkRules(mCurRuleList, i);
                  if (satisfiedRuleList.size() > 0) {
                     mHandler.post(new Runnable() {
                        @Override
                        public void run() {
                           /* broadcast new context*/
                           Intent i = new Intent();
                           i.setAction("edu.hkust.cse.PhoneAdapter.adaptationRequest");
                           i.putExtra("symptom", "newProfileEnabled");
                           i.putExtra("satisfiedRuleList",
transformListToArray(satisfiedRuleList));
                           sendBroadcast(i);

                        }
                     });
                  }   }
         } else if (action.equals("edu.hkust.cse.PhoneAdapter.profileChange")) {...
         } else if (action.equals("edu.hkust.cse.PhoneAdapter.stopService")) { ...
         } else {   //nothing  }
      }

      private int[] transformListToArray(ArrayList<Rule> rules) {...    }
   private static ArrayList<Rule> checkRules(ArrayList<Rule> ruleList, Intent i)  ...}
   public static double calculateDist(String lastLoc, String curLoc) { ... }
   public static boolean macListContainsMac(String[] list, String mac) { ... }
   private static int compareTime(String t1, String t2) {...  }
   private static int compareWeekday(String wd1, String wd2) { ... }
}
```

Listagem 12 Classe *Analyzer*.

```
@Actor(name = "Planner",type = ActorType.AGENT)
@Goal(name = "Strategy for profile change be accomplished",actor =
"Planner",description = "")
public class Planner extends IntentService {
      //atributos
   ...
   @Override
   public void onCreate() {
      ...
      IntentFilter iFilter = new
 IntentFilter("edu.hkust.cse.PhoneAdapter.adaptationRequest");
      iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
      ...
```

```
391.        }
392.
393.        @Override
394.        protected void onHandleIntent(Intent intent) { while (!mStop) {  } }
395.     ...
396.        private class MyBroadcastReceiver extends BroadcastReceiver {
397.
398.           @Override
399.     @MeansEnd(end = "Strategy for profile change be accomplished",endType =
400.     MeansEndType.GOAL)
401.           public void onReceive(Context context, Intent i) {
402.              String action = i.getAction();
403.              if (action.equals("edu.hkust.cse.PhoneAdapter.adaptationRequest")) {
404.                 while (true) {
405.                    String symptom = i.getStringExtra("symptom");
406.                    createStrategy(symptom, i);
407.                 }
408.              } else if (action.equals("edu.hkust.cse.PhoneAdapter.stopService")) {
409.                 mStop = true;  }
410.           }   }
411.
412.        private void createStrategy(String symptom, Intent i) {
413.           switch (symptom) {
414.              case "newProfileEnabled": {
415.                 int[] ruleList = i.getIntArrayExtra("satisfiedRuleList");
416.                 satisfiedRuleList = fetchRulesFromDb(ruleList);
417.                 if (satisfiedRuleList.size() == 1) {
418.                    broadcastStrategy(satisfiedRuleList.get(0));
419.                 } else {
420.                    ...
421.                    } else { ... }
422.                 }
423.                 break;
424.              }
425.              default: {
426.                 break;
427.              }       }   }
428.
429.        private void broadcastStrategy(Rule satisfiedRule) {
430.           volumeLevel = (satisfiedRule.newProfile.ringVolume) * 1.0 / 100;
431.           airplaneMode = satisfiedRule.newProfile.airplaneMode;
432.           vibratiom = satisfiedRule.newProfile.vibration;
433.
434.           mHandler.post(new Runnable() {
435.              @Override
436.              public void run() {
437.                 /* broadcast new strategy*/
438.                 Intent i = new Intent();
439.                 i.setAction("edu.hkust.cse.PhoneAdapter.strategy");
440.                 i.putExtra("vibratiom", vibratiom == 1 ? true : false);
441.                 i.putExtra("airplaneMode", airplaneMode == 1 ? true : false);
442.                 i.putExtra("ringVolume", volumeLevel);
443.                 i.putExtra("newProfile", satisfiedRuleList.get(0).newProfile.id);
444.                 sendBroadcast(i);
445.              }
446.           });
447.        }}
```

Listagem 13 Classe *Planner*.

Uma vez que o Executor (Listagem 14) recebeu a estratégia ele dispara os pedidos de execução das ações para os atuadores. Um dos atuadores implementados é o *RingVolumeEffector* (Listagem 15) que é responsável por adaptar o volume do toque do smartphone conforme o perfil que foi habilitado.

```
448.  @Actor(name = "Executor",type = ActorType.AGENT)
449.  @Goal(name = "The profile change strategy be carried out",actor =
450.  "Executor",description = "")
451.  public class Executor extends IntentService {
452.  ...
453.     public Executor() {
454.        super("Executor");
455.     }
456.
457.     @Override
458.  public void onCreate() {
459.        ...
460.           IntentFilter iFilter = new IntentFilter("edu.hkust.cse.PhoneAdapter.strategy");
461.           iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
462.           ...
463.     }...
464.     @Override
465.     protected void onHandleIntent(Intent intent) { while (!mStop) { } }
466.
467.     private class MyBroadcastReceiver extends BroadcastReceiver {
468.        @Override
469.  @MeansEnd(end = "The profile change strategy be carried out",endType =
470.  MeansEndType.GOAL)
471.        public void onReceive(Context context, Intent i) {
472.           String action = i.getAction();
473.           switch (action) {
474.              case "edu.hkust.cse.PhoneAdapter.stopService": {
475.                 mStop = true;
476.                 stopSelf();
477.                 break;
478.              }
479.              case "edu.hkust.cse.PhoneAdapter.strategy": {
480.                 executeStrategy(i);
481.                 break;
482.              }
483.              default: {  }  }  }
484.     }
485.     private void executeStrategy(Intent strategy) {
486.        ...
487.        volumeLevel = strategy.getDoubleExtra("ringVolume",0.0);
488.        mHandler.post(new Runnable() {
489.           @Override
490.           public void run() {
491.              /* broadcast new context*/
492.              Intent i = new Intent();
493.              i.setAction("edu.hkust.cse.PhoneAdapter.volumeLevel");
494.              i.putExtra("volumeLevel", volumeLevel);
495.              sendBroadcast(i);
496.           }
497.        });
498.        ...    }}
```

Listagem 14 Classe *Executor*.

```
499.  @Actor(name = "RingVolumeEffector",type = ActorType.AGENT)
500.  @Goal(name = "That the ring volume be adjusted",actor =
501.  "RingVolumeEffector",description = "")
502.  public class RingVolumeEffector extends IntentService {
503.     ...
504.
505.     @Override
506.     public void onCreate() {
507.     ...
508.     IntentFilter iFilter = new IntentFilter("edu.hkust.cse.PhoneAdapter.volumeLevel");
509.        iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
510.        registerReceiver(mReceiver, iFilter);
511.       ...
512.     }
513.     @Override
514.     protected void onHandleIntent(Intent intent) {
515.        while (!mStop) {
516.        }
517.     }
518.
519.     private class MyBroadcastReceiver extends BroadcastReceiver {
520.        @Override
521.        public void onReceive(Context context, Intent i) {
522.           String action = i.getAction();
523.           switch (action) {
524.              case "edu.hkust.cse.PhoneAdapter.stopService": {
525.                 mStop = true;
526.                 stopSelf();
527.                 break;
528.              }
529.              case "edu.hkust.cse.PhoneAdapter.volumeLevel": {
530.                 double ringVolume = i.getDoubleExtra("ringVolume",0.0);
531.                 int volume = (int) (ringVolume * mMaxVolume);
532.                 if (volume > 0) {
533.                    ringVolumeOn(volume);
534.                 } else {
535.                    ringVolumeOff();
536.                 }
537.                 break;
538.              }
539.              default: {
540.              } }
541.        } }
542.
543.  @MeansEnd(end = "That the ring volume be adjusted",endType =
544.  MeansEndType.GOAL)
545.     private void ringVolumeOff() {
546.        mAudioManager.setStreamVolume(AudioManager.STREAM_RING, 0,
547.   AudioManager.FLAG_SHOW_UI);
548.     }
549.
550.  @MeansEnd(end = "That the ring volume be adjusted",endType =
551.  MeansEndType.GOAL)
552.     private void ringVolumeOn(int volume) {
553.        mAudioManager.setRingerMode(AudioManager.RINGER_MODE_NORMAL);
554.        mAudioManager.setStreamVolume(AudioManager.STREAM_RING, volume,
555.  AudioManager.FLAG_SHOW_UI);
556.     }
557.  }
```

Listagem 15 Classe *RingVolumeEffector*.

As informações coletadas pelos sensores, logs de estado do sistema e do ambiente, requisições de adaptação, sintomas, políticas e estratégias, que costumam estar na base de conhecimento do padrão MAPE ou MAPE-K(IBM, 2006), estão sendo enviadas como notificação e os agentes se cadastraram para receber as notificações que tenham interesse. As alternativas de adaptação (“políticas”) e os valores de referência (“sintomas”) que disparam a autoadaptação, por serem definidos pelo usuário do aplicativo, são armazenados na base de dados. A seguir apresentamos as classes de modelo que são as abstrações que representam um conjunto de sintomas (i.e.: *Rule* (Listagem 16) + *Filter* (Listagem 17)) e alternativas de adaptação do *smarphone* (i.e.: *Profile* (Listagem 18)). Estas classes eram privadas e estavam implementadas junto à classe *AdaptationManager* do sistema original. Na reimplementação do sistema, tornamos estas classes públicas pois elas são classes de modelo que representam abstrações importantes no mecanismo de autoadaptação.


```
@Resource(name = "Policy")
public class Rule {
   public int id;
   public String ruleName;
   public Profile currentProfile;
   public ArrayList<Filter> filterList;
   public Profile newProfile;
   public int priority;
}
```

Listagem 16 Classe *Rule*.

```
@Resource(name = "Symptom")
public class Filter{
      public int contextType;
      public int contextOp;
      public String contextValue;
}
```

Listagem 17 Classe *Filter*.

```
@Resource(name = "Adaptation Alernatives")
public class Profile {
   public int id;
   public String profileName;
   public int ringVolume;
   public int airplaneMode;
   public int vibration;

}
```

Listagem 18 Classe Profile.

Ao final da reimplementação do sistema, o pacote *context*, que contém as classes responsáveis pelo mecanismo de autoadaptação, possui quinze classes e três

interfaces com valores de constantes (ver Figura 41). Apesar do aumento de número de classes, elas estão mais coesas e menos acopladas. O código do *PhoneAdapter* pós-reengenharia está disponível no GitHub[27].

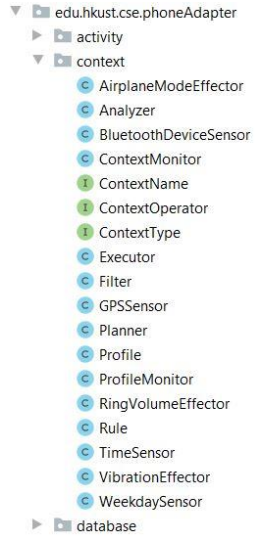


Figura 41 Classes criadas ou adaptadas no *PhoneAdapter*.

5.2.2. Teste do *PhoneAdapter*

Como dito anteriormente, o *PhoneAdapter* permite adaptar o perfil do smartphone (i.e.: volume da campainha, modo vibração e modo avião) conforme o contexto definido pelo usuário (i.e.: disponibilidade de GPS, localização, tempo, velocidade, dia da semana, quantidade de dispositivos *Bluetooth* e/ou um dispositivo *Bluetooth* específico). A seguir mostramos um teste de uso do aplicativo após a reengenharia, no qual criamos um perfil, o qual demos o nome "*Baby Sleeping*", e definimos uma regra para a ativação desse perfil a qual demos o nome de "*Baby_Sleep_Time*".

A Figura 42, apresentamos os passos para a definição do perfil "*Baby Sleeping*" com as adaptações necessárias para o smartphone a partir do horário que o bebê está dormindo. Os testes foram realizados no simulador do Android Studio. A primeira tela é a tela principal do aplicativo com os menus principais. A segunda tela é a tela de criação de um novo perfil. A terceira tela apresenta os dados preen-

[27] https://github.com/professoraanamoura/PhoneAdapter_MAPE

chidos: nome (“*Baby Sleeping*”), volume do toque (“0”), modo de vibração (“ligado”) e modo avião (“desligado”). A quarta tela é a tela de visualização dos perfis criados, onde podemos visualizar o perfil “*Baby Sleeping*” criado com sucesso. O *PhoneAdapter* pós reengenharia continua possibilitando que os usuários definam perfis de adaptação do smartphone.

Figura 42 Criação de novo perfil de adaptação do smartphone.

Após a criação do perfil “*Baby Sleeping*”, criamos as regras de adaptação que informam o perfil de origem, o novo perfil, a prioridade e os filtros que indicam a necessidade de mudança de perfil. A Figura 43 apresenta a criação da regra

"*Baby_Sleep_Time*". A primeira tela desta figura é a tela principal do aplicativo, onde selecionamos a opção "*Create Rule*". A segunda tela é a tela de criação da regra, onde preenchemos as informações.

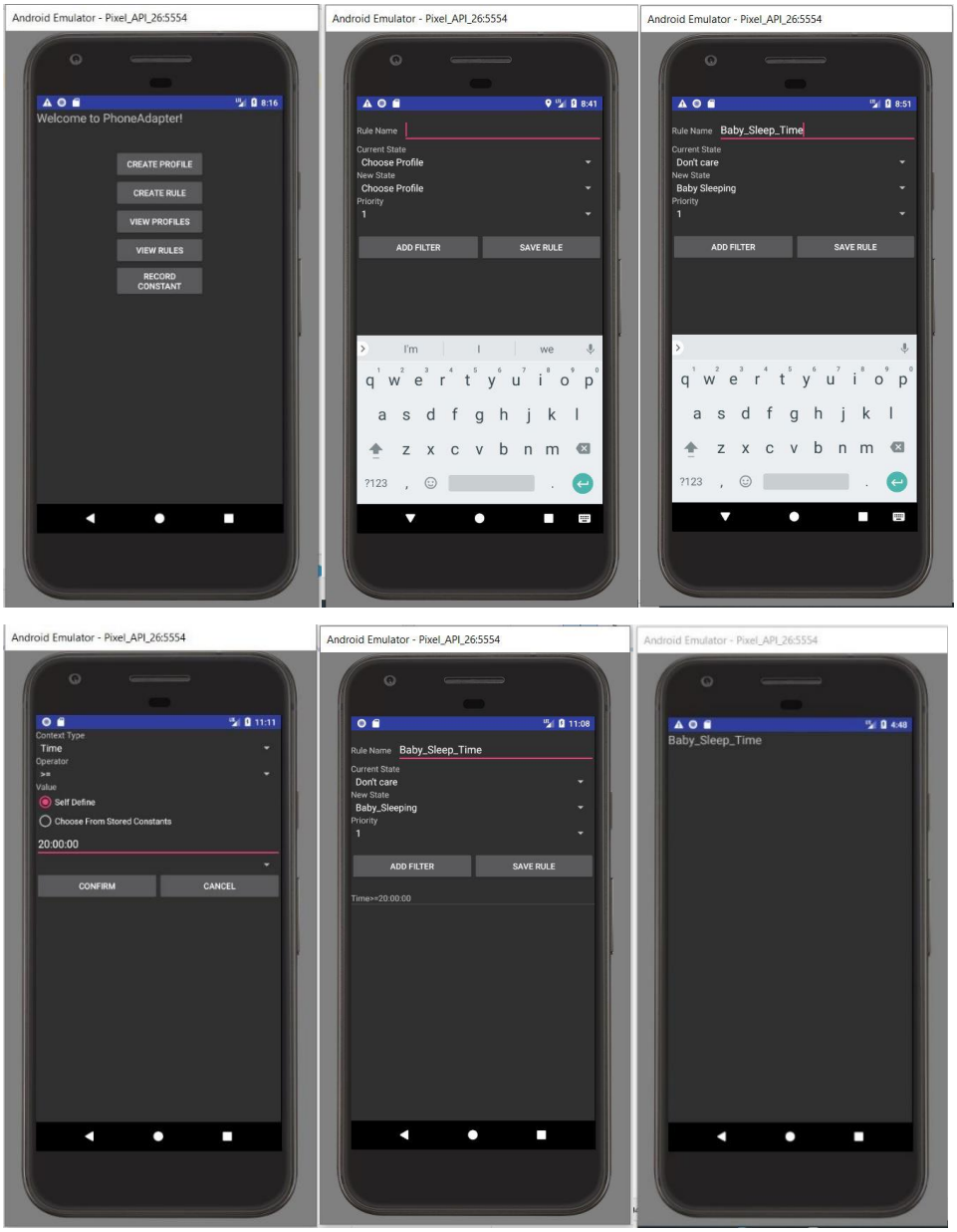


Figura 43 Criação de regras de contexto que disparam uma adaptação.

A terceira tela mostra os dados da regra preenchidos: perfil de origem ("*Don't care*"), perfil de destino ("*Baby Sleeping*") e prioridade ("1"). Em seguida, clicamos na opção "*Add Filter*" e a quarta tela é apresentada para preenchermos o filtro, onde escolhemos o filtro "*Time*", o operador ">=" e o valor "20:00:00".

Após a criação do filtro, ele aparece na quinta tela da Figura 43 junto aos dados iniciais da regra. Consequentemente, a sexta tela lista a regra

"*Baby_Sleep_Time*" já criada. O perfil criado é acionado a partir do horário de 20:00 e a adaptação causada é redução do volume da campainha do smartphone para 0 (zero) e o modo vibração é ativado conforme mostra a Figura 44.

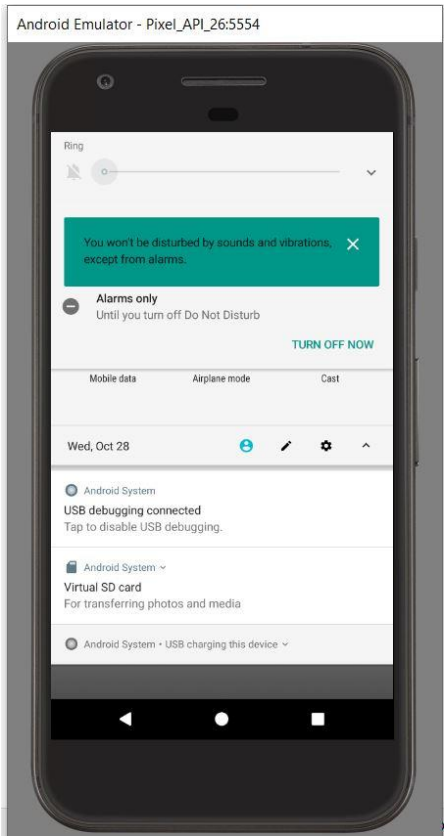


Figura 44 Perfil do smartphone adaptado.

Adicionalmente, escrevemos mensagens de log no código do novo sistema para mostrar as mensagens trocadas entre os agentes do MAPE. Conforme apresentado na Figura 45, assim que a regra é criada, já conseguimos ver pelo log da aplicação os passos realizados até a adaptação (ver seta vermelha na Figura 45) do smartphone ser realizada.

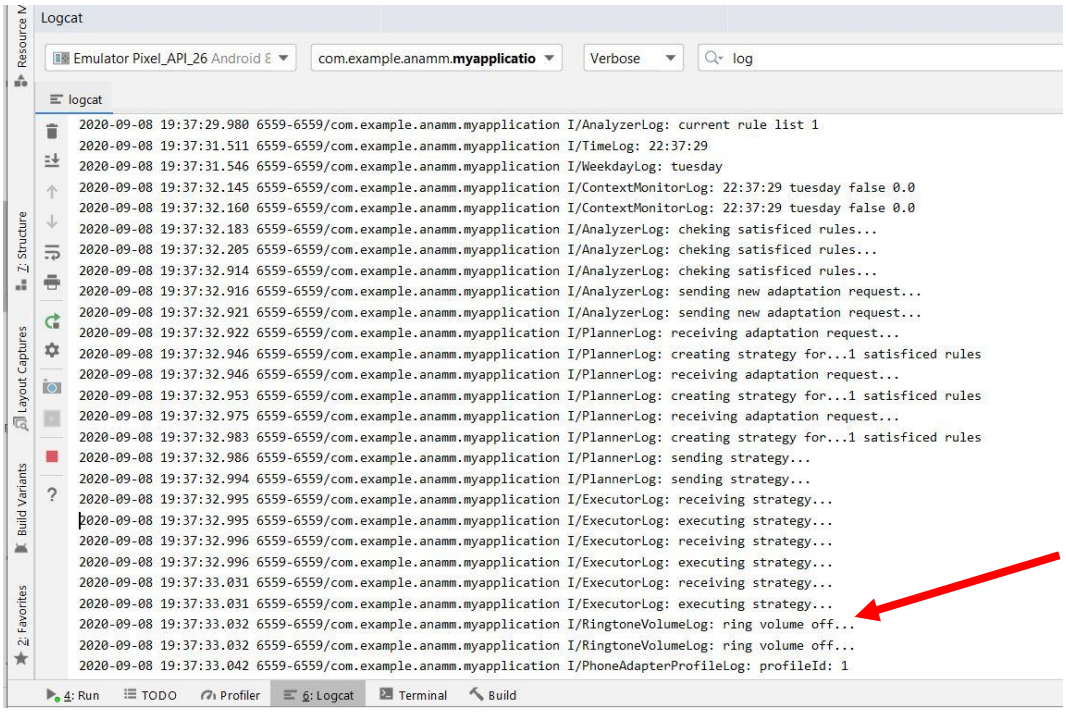


Figura 45 Log de adaptação do smartphone.

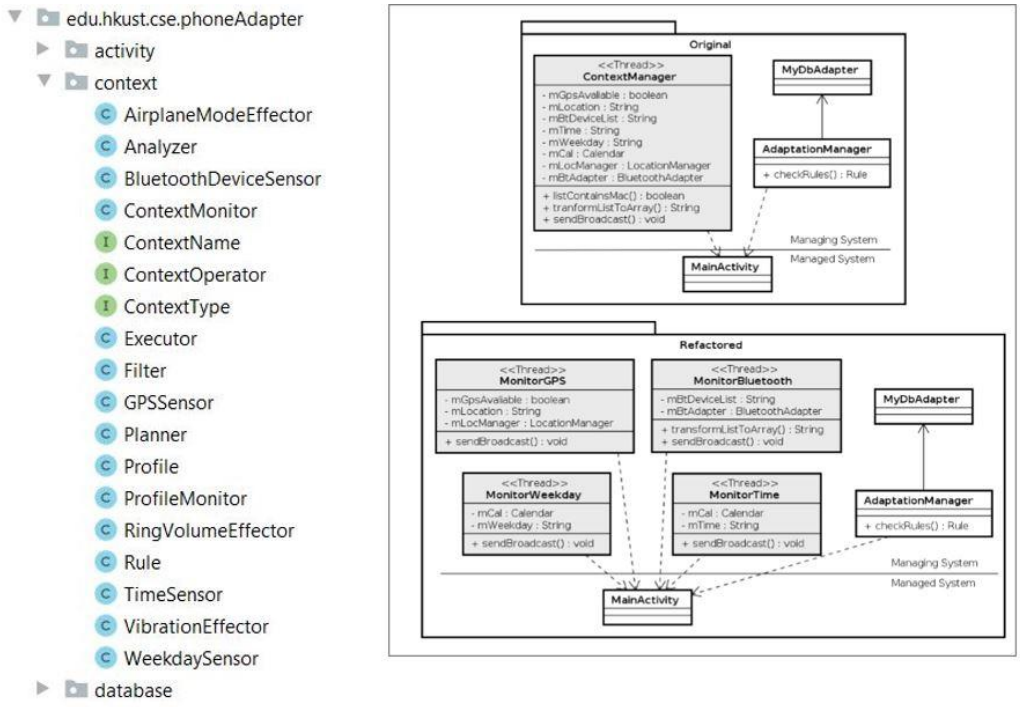
5.2.3. Avaliação

Quando comparamos a reorganização do aplicativo que foi alcançada pela nossa estratégia com o resultado da refatoração realizada por Serikawa et al. (SERIKAWA et al., 2016), relativo as funções de sensoriamento e monitoramento do MAPE (IBM, 2006), é possível notar que, assim como estes pesquisadores, removemos os problemas de “Monitores Oprimidos” e “Monitor Obscuro”.

Conforme Figura 46, apesar de Serikawa et al. terem removido os problemas citados, eles deixaram os monitores e sensores mistos e isto criou um acoplamento entre estes dois conceitos do MAPE que dificulta o seu reuso, modificação e testes de modo independente.

Observando os resultados das duas estratégias na Figura 46, é possível notar que, utilizando nossa estratégia de reengenharia, foram criados os sensores *BluetoothDeviceSensor*, *GPSSensor*, *TimeSensor* e *WeekdaySensor* e os monitores *ContextMonitor* e *ProfileMonitor*. Deste modo, separamos as responsabilidades de sensores e monitores, identificamos o monitoramento do perfil que estava obscuro na

classe *AdaptationManager* e tivemos a oportunidade de revisar as operacionalizações utilizadas com o reuso do conhecimento presente no catálogo de consciência.

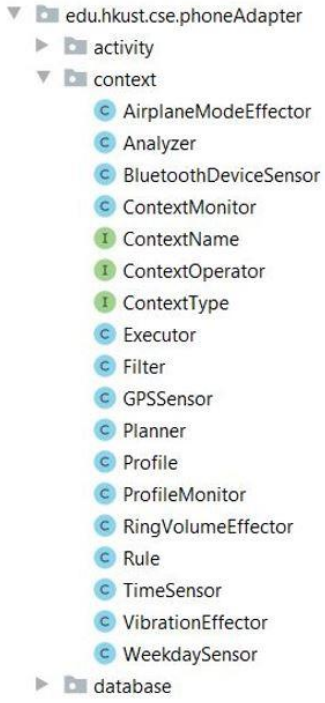


PhoneAdapter após nossa estratégia de reengenharia

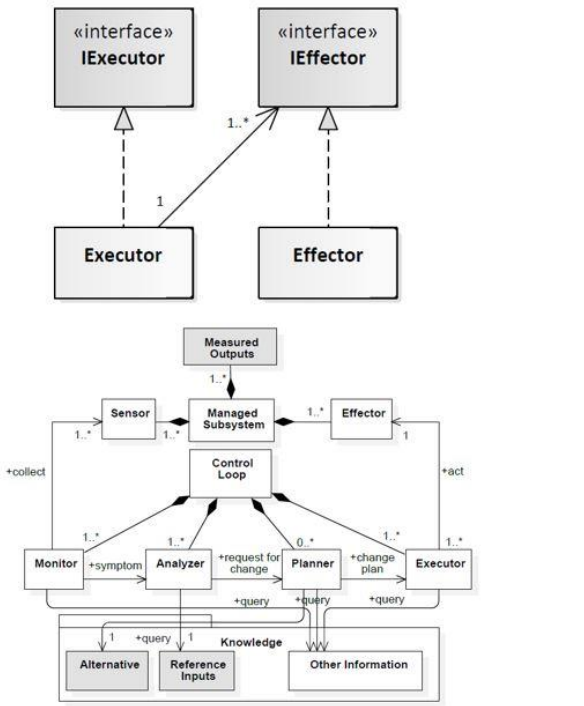
PhoneAdapter após a refatoração proposta por Serikawa et al.

Figura 46 Comparação com os resultados da refatoração de Serikawa et al (SERIKAWA et al., 2016).

Quando comparamos os nossos resultados ao resultado da refatoração realizada por San Martín et al. (SAN MARTÍN et al., 2020) no *PhoneAdapter*, relativo as funções de executores e atuadores do MAPE (IBM, 2006), ver Figura 47, é possível notar que, assim como estes pesquisadores, removemos o problema de "Executores e Atuadores Mistos" e, além disso, melhoramos a modularidade dos valores de referência tornando públicas as classes *Rule* e *Filter* e das alternativas de adaptação que estão definidas na classe *Profile*, a qual também foi tornada pública. Estas classes estavam com visibilidade privada dentro da classe *AdaptationManager* e isto torna mais difícil a sua localização, reuso e realização de testes.



PhoneAdapter após nossa estratégia de reengenharia



Separação de executores e atuadores e caracterização de informações da base de conhecimento proposta por San Martin et al.

Figura 47 Comparação com os resultados da refatoração de San Martín et al (SAN MARTÍN et al., 2020).

Dado que realizamos a reengenharia do aplicativo *PhoneAdapter*, a partir deste momento, verificamos as hipóteses acerca desta pesquisa.

Hipótese 1: reutilizar meta conhecimento de consciência de *software* pode auxiliar na identificação de metas flexíveis e operacionalizações de consciência em sistemas autoadaptativos.

Com vistas a avaliar a nossa estratégia sob a perspectiva da hipótese 1, recuperamos um modelo de metas i* *AS-IS* do sistema autoadaptativo real chamado *PhoneAdapter* com as metas flexíveis de consciência operacionalizadas no código, conforme mostra a Figura 38. Para identificar as metas flexíveis operacionalizadas, foi preciso o uso do catálogo de consciência de *software* de modo estruturado com as possíveis operacionalizações de consciência para aplicativos Android em Java. Estruturamos esse conhecimento com o uso do formato NFRJson. A identificação da meta flexível além da operacionalização propriamente auxiliou na revisão das operacionalizações utilizadas. No exemplo da Figura 48, apresentamos a identificação da consciência de tempo, na classe *ContextManager*. Deste modo, evidenci-

amos que reutilizar o conhecimento de consciência de *software* auxiliou na identificação de metas flexíveis e operacionalizações de consciência implementadas no código fonte.

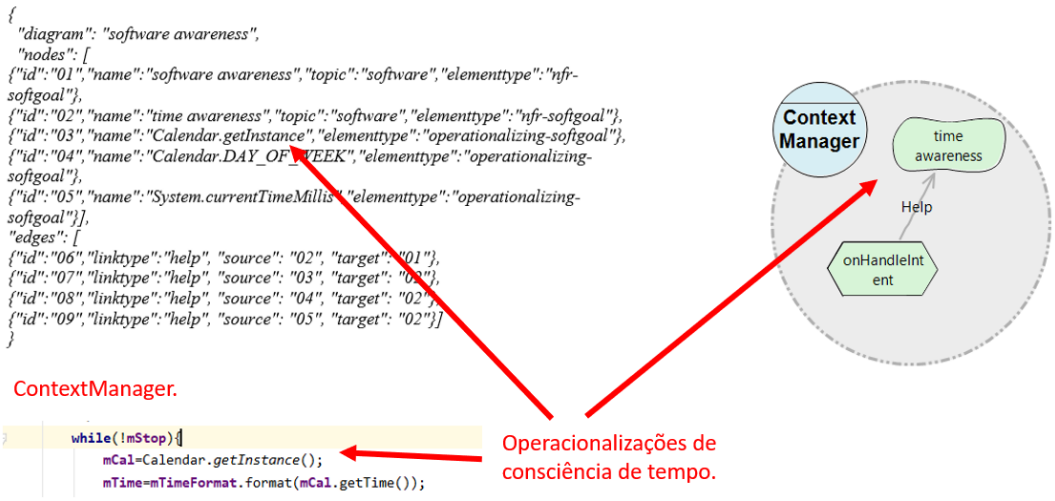


Figura 48 Exemplo de identificação de consciência de tempo.

Hipótese 2: reutilizar a meta arquitetura MAPE de sistemas autonômicos em modelos de metas pode ajudar na reorganização da modularidade de sistemas autoadaptativos, diminuindo os problemas arquiteturais recorrentes na literatura.

Para avaliar a nossa estratégia, conforme o disposto na hipótese 2, reespecificamos o sistema PhoneAdapter2 através do reuso da meta arquitetura MAPE através de *SRconstructs*. Comparando a distribuição de responsabilidades das classes responsáveis pelo mecanismo de autoadaptação antes da reengenharia (ver modelo i* *AS-IS* na Figura 38) e após a reengenharia (ver modelo *TO-BE* na Figura 40), notamos que as classes estão mais coesas e as responsabilidades estão separadas conforme o padrão MAPE.

Como descrito anteriormente, Serikawa et al. (SERIKAWA et al., 2016) identificaram, no *PhoneAdapter*, os problemas **Monitor Obscuro** e **Monitores Oprimidos**. Enquanto, San Martín et al. (SAN MARTÍN et al., 2020) identificaram, no *PhoneAdapter*, o problema **Executores e Atuadores Mistos**. Avaliando o novo sistema sob a perspectiva destes problemas de implementação do MAPE recorrentes na literatura que foram identificados no *PhoneAdapter*, como mostra a Figura 49, o novo sistema não possui o problema de **Monitor Obscuro** pois os sensores e monitores estão individualizados e caracterizados desde o modelo até o código. Isto

significa que estas abstrações estão mais fáceis de serem reutilizadas, compreendidas e mantidas/evoluídas. É importante ressaltar também que cada um dos sensores e monitores possuem frequência individualizada e alguns são disparados sob demanda, decorrente disso, não há mais o problema de **Monitores Oprimidos** e eles estão menos acoplados, ocasionando menos gasto de recurso desnecessariamente.

Devido à reorganização, o novo sistema também não possui mais o problema de **Executores e Atuadores Mistos** pois existe uma separação clara destes elementos e os mesmos são acionados sob demanda. Conforme mostra a Figura 50, esta separação ocorre desde o modelo até o código. Além disso, nossa estratégia de reengenharia deixou ainda mais caracterizadas as abstrações da base de conhecimento referente as entradas de referência e alternativas de adaptação.

A Tabela 9 demonstra uma análise mais detalhada sobre as razões pelas quais o novo sistema não possui os problemas de implementação do MAPE recorrentes na literatura.

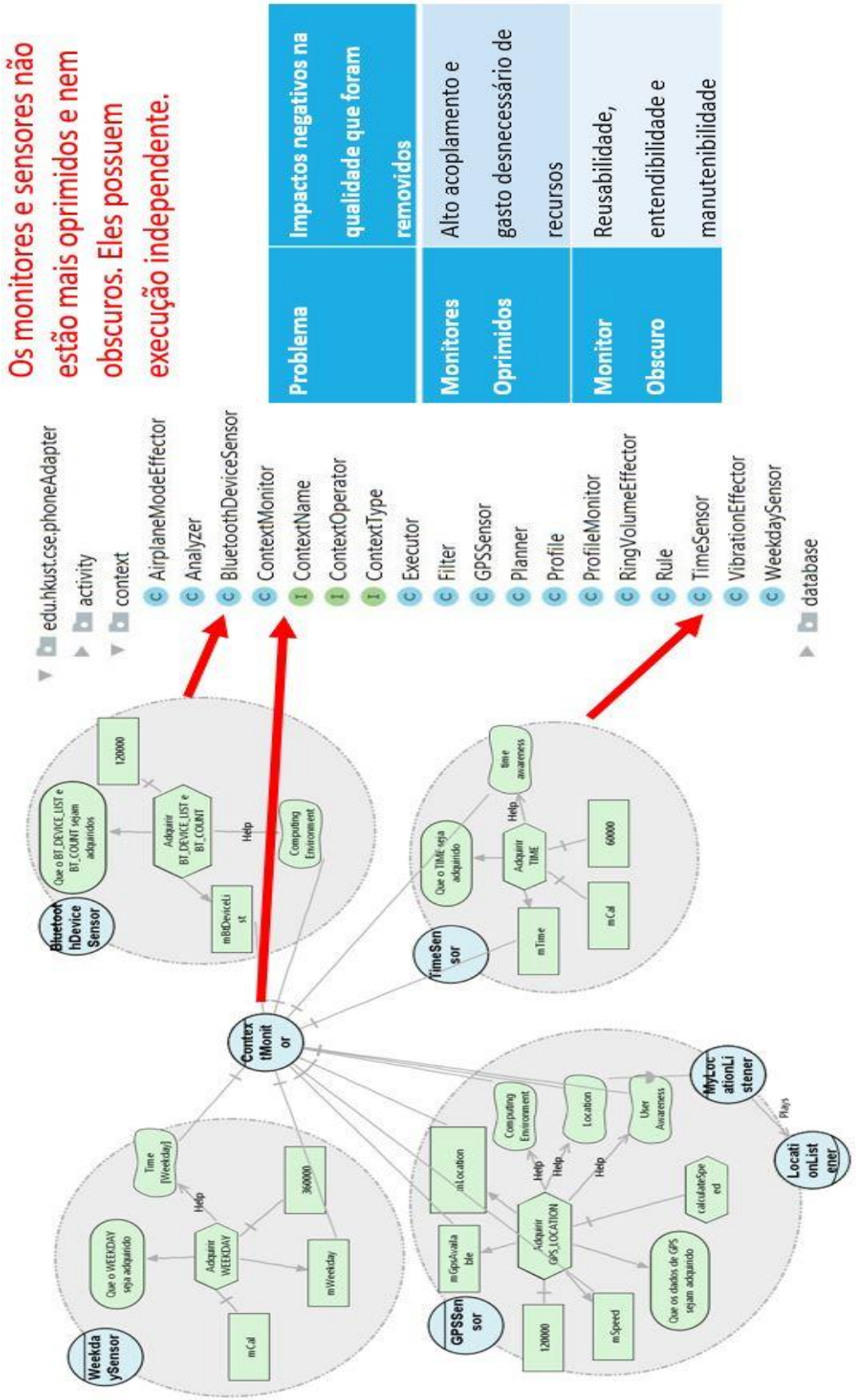


Figura 49 Remoção dos problemas apontados por Serikawa et al. (SERIKAWA et al., 2016)

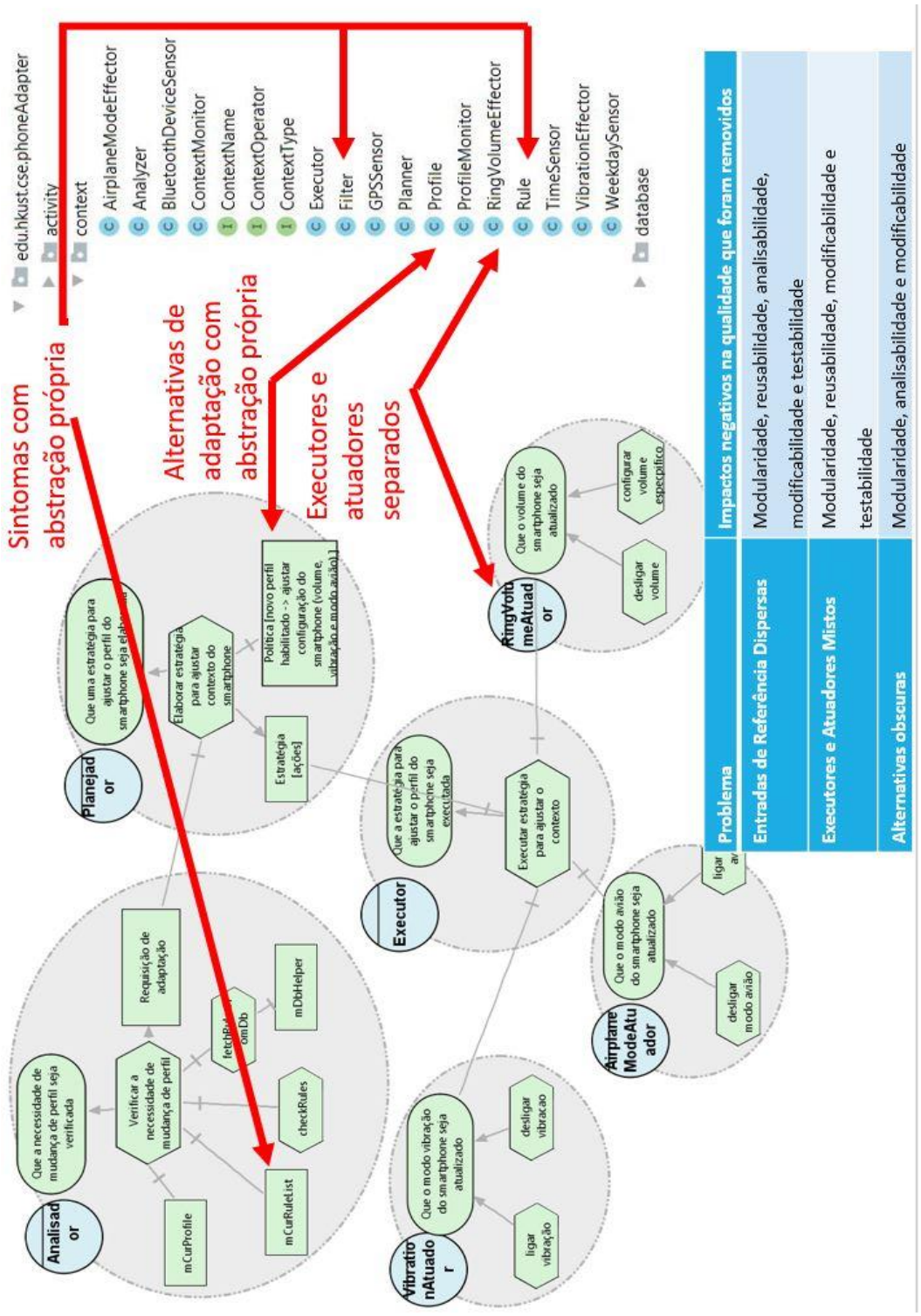


Figura 50 Remoção dos problemas apontados por San Martín et al. (SAN MARTÍN et al., 2020)

Tabela 9 Análise do novo sistema sob a perspectiva dos problemas de implementação do MAPE.

Problema	Atributos de qualidade	Modelo	Implementação
Monitores Oprimidos	Alto acoplamento	Os agentes sensores e monitores são especificados separadamente e possuem metas específicas. Conforme o modelo *TO-BE* da Figura 39, criamos os agentes para os sensores *WeekdaySensor*, *TimeSensor*, *BluetoothDeviceSensor* e *GPSSensor* e para os monitores ContextMonitor e ProfileMonitor. Deste modo, os agentes podem se comunicar para atender as dependências entre eles sem que precisem conhecer detalhes da implementação um do outro e sem a necessidade da imposição de uma ordem de execução específica.	Os sensores foram reimplementados como especificado e redesenhado nos modelos *TO-BE (ver* Figura 39 *e* Figura 40*)*: *TimeSensor WeekdaySensor*, *GPSSensor* e *BluetoothDeviceSensor* (ver Figura 41). Nós também reimplementamos os monitores de modo independente, conforme especificado e redesenhado nos modelos *TO-BE (ver* Figura 39 *e* Figura 40*)*: *ContextMonitor* e *ProfileMonitor* (ver Figura 41).
	Gasto desnecessário de recursos	Os agentes sensores alcançam sua meta em uma frequência independente a qual é determinada de acordo com o tipo de consciência e o tópico ao qual se refere. No modelo *TO-BE* da Figura 39, os sensores *WeekdaySensor*, *TimeSensor*, *BluetoothDeviceSensor* e *GPSSensor* alcançam suas metas com as respectivas frequências de 360000ms, 60000ms, 120000ms e 120000ms.	Cada sensor adquire os dados de contexto os quais são responsáveis em frequências independentes: *WeekdaySensor*: 360000ms, *TimeSensor*: 60000ms, *BluetoothDeviceSensor:* 120000ms, *GPSSensor:* 120000ms (ver Apêndice 1).
		Os agentes monitores geram logs do estado do sistema à medida que recebem o recurso esperado dos sensores os quais ele monitora, conforme os monitores *ContextMonitor* e *ProfileMonitor* da Figura 39 que geram novos logs de estado do sistema à medida que recebem os dados (“recursos”) adquiridos pelos sensores que eles monitoram.	Os monitores geram logs do sistema à medida que recebem uma notificação dos monitores os quais se inscreveram para receber notificações: *ContextMonitor* (Listagem 11), Inscrição para receber notificações(ver Apêndice 1), Recebimento de notificação e geração de log: linhas 284 a 331, *ProfileMonitor* (ver Apêndice 1), Inscrição para receber notificações (ver Apêndice 1), Recebimento de notificação e geração de log: (ver Apêndice 1).
Monitor Obscuro	Reusabilidade	Os monitores e sensores são especificados como agentes em separado que podem ser reutilizados. Conforme o modelo *TO-BE* da Figura 39, criamos os agentes independentes para os	Os sensores foram reimplementados como especificado e redesenhado nos modelos *TO-BE (ver* Figura 39 *e* Figura 40*)* e isto facilita o reuso, o entendimento do código e a manutenção:

		sensores *WeekdaySensor*, *TimeSensor*, *BluetoothDeviceSensor* e *GPSSensor* e para os monitores *ContextMonitor* e *ProfileMonitor*.	*TimeSensor* (Listagem 10), *WeekdaySensor* (ver Apêndice 1), *GPSSensor* (ver Apêndice 1), *BluetoothDeviceSensor* (ver Apêndice 1). Nós também reimplementamos os monitores de modo independente, conforme especificado e redesenhado nos modelos *TO-BE (ver* Figura 39 *e* Figura 40*)*: *ContextMonitor* (Listagem 11), *ProfileMonitor* (ver Apêndice 1).
	Entendibilidade	Os monitores e sensores são especificados como agentes independentes e isto facilita a compreensão do "*rationale*" de cada agente. Conforme o modelo *TO-BE* da Figura 39, criamos os agentes independentes para os sensores *WeekdaySensor*, *TimeSensor*, *BluetoothDeviceSensor* e *GPSSensor* e para os monitores ContextMonitor e ProfileMonitor.	
	Manutenibilidade	Os monitores e sensores são especificados como agentes independentes e isto facilita a localização do local ("*rationale*") onde a manutenção deve ocorrer e minimiza o impacto no "*rationale*" de outros agentes. Conforme o modelo *TO-BE* da Figura 39, criamos os agentes independentes para os sensores *WeekdaySensor*, *TimeSensor*, *BluetoothDeviceSensor* e *GPSSensor* e para os monitores ContextMonitor e ProfileMonitor.	
Entradas de Referência Dispersas	Modularidade	A lista de sintomas que é utilizada como valores de referência para analisar o log do estado do sistema está modularizada como o recurso "*mCurRuleList*" do analisador que é consumido pela tarefa "Verificar a necessidade de mudança de perfil", conforme mostra o modelo *TO-BE* da Figura 39. Esta modularidade minimiza as chances de efeitos colaterais em manutenções realizadas sobras esta abstração, poisos valores de referência ("sintomas") não espalhados em outros agentes.	A lista de sintomas está representada como o atributo "*mCurRuleList*" da classe *Analyzer* (Listagem 12) que é a implementação do agente analisador do modelo *TO-BE* da Figura 39. Esta lista de regras é lida a partir do banco de dados e é utilizada para verificar se existe alguma regra ("sintoma") que indique a necessidade de mudança de perfil do smartphone. A verificação é feita no método *chekRules*.
	Reusabilidade	A lista de sintomas que é utilizada como valores de referência para analisar o log do estado do sistema está definida como o recurso "*mCurRuleList*" do analisador e possui interfaces bem definidas. Isto possibilita o seu reuso em outros contextos, por exemplo, o planejador que recebe um conjunto de sintomas identificados junto à requisição de adaptação (ver *TO-BE* da Figura 39).	A lista de sintomas está representada como o atributo "*mCurRuleList*" da classe *Analyzer* (Listagem 12) e parte dela, *satisfiedRuleList*, é reutilizada na classe *Planner* (Listagem 13) quando a mesma planeja uma estratégia de adaptação.
	Analisabilidade	Com o uso de uma lista de sintomas bem definida como o recurso "*mCurRuleList*" do analisador, é possível analisar como	A lista de sintomas está representada como o atributo "*mCurRuleList*" da classe *Analyzer* (Listagem 12) e esta lista é composta

		adicionar novos valores de referência (“sintomas”) sem que a capacidade de compreensão destes valores de referência seja prejudicada (ver *TO-BE* da Figura 39).	por elementos bem definidos do tipo *Rule* (Listagem 16). Com esta estrutura bem definida é possível analisar como adicionar novas regras (“sintomas”) e como alterar a estrutura de uma regra (“sintoma”).
	Modificabilidade	Com o uso de uma lista de sintomas bem definida como o recurso “*mCurRuleList*“ do analisador (ver *TO-BE* da Figura 39), é possível realizar modificações nos valores de referência (“sintomas”) em locais bem definidos e isto diminui o tempo da manutenção e o risco de afetar outros módulos inesperados.	A lista de sintomas está representada como o atributo “*mCurRuleList*“ da classe *Analyzer* (Listagem 12) e esta lista é composta por elementos bem definidos do tipo *Rule* (Listagem 16). Com esta estrutura bem definida é possível modificar o conjunto de regras (“sintomas”) assim como alterar a estrutura de uma regra (“sintoma”).
	Testabilidade	Com o uso de uma lista de sintomas bem definida como o recurso “*mCurRuleList*“ do analisador e posteriormente é reutilizada no planejador ao elaborar uma estratégia de adaptação com base na requisição de adaptação que contém os sintomas identificados (ver *TO-BE* da Figura 39), é possível criar casos de teste mais simples para um número bem definido de módulos.	A lista de sintomas está representada como o atributo “*mCurRuleList*“ da classe *Analyzer* (Listagem 12) e esta lista é composta por elementos bem definidos do tipo *Rule* (Listagem 16). Esta mesma estrutura é reutilizada na classe *Planner* (Listagem 13). Com esta estrutura bem definida é possível criar casos de teste mais simples e que envolvam um número de módulos bem definido.
Executores e Atuadores Mistos	Modularidade	Os agentes executores e atuadores são especificados em separado com situações de dependência bem definidas. Conforme modelo *TO-BE* da Figura 39, criamos um agente Executor e três agentes atuadores: RingVolumeEffector, AirplaneModeEffector e VibrationEffector.	O executor e os atuadores foram reimplementados, como especificado e redesenhado nos modelos *TO-BE (ver* Figura 39 *e* Figura 40): Executor (Listagem 14), RingVolumeEffector (Listagem 15), AirplaneModeEffector (ver Apêndice 1) e VibrationEffector (ver Apêndice 1).
	Reusabilidade	Como os agentes executores e atuadores estão especificados em separado com situações de dependência bem definidas, conforme modelo *TO-BE* da Figura 39, é possível reutilizar estes agentes mais facilmente.	Como o executor e os atuadores foram reimplementados em separado, como especificado e redesenhado nos modelos *TO-BE (ver* Figura 39 *e* Figura 40), é possível reutilizá-los mais facilmente. As classes Executor (Listagem 14), RingVolumeEffector (Listagem 15), AirplaneModeEffector (ver Apêndice 1) e VibrationEffector (ver Apêndice 1) podem ser reutilizadas em separado

			pois as dependências entre o executor e os atuadores são resolvidas através de notificação entre as classes e isto reduz bastante o acoplamento.
	Modificabilidade	Como os agentes executores e atuadores estão especificados em separado com situações de dependência bem definidas, conforme modelo *TO-BE* da Figura 39, é possível modificar o executor e/ou os atuadores sem afetar um ao outro.	Como o executor e os atuadores foram reimplementados em separado, como especificado e redesenhado nos modelos *TO-BE* (*ver* Figura 39 *e* Figura 40), é possível modificá-los sem afetar um ao outro. As classes *Executor* (Listagem 14), *RingVolumeEffector* (Listagem 15), *AirplaneModeEffector* ((ver Apêndice 1) e *VibrationEffector* (ver Apêndice 1) podem ser modificadas em separado pois as dependências entre o executor e os atuadores são resolvidas através de notificação entre as classes e isto reduz bastante o acoplamento.
	Testabilidade	Como os agentes executores e atuadores estão especificados em separado com situações de dependência bem definidas, conforme modelo *TO-BE* da Figura 39, é possível criar casos de teste mais simples para testá-los em separado ou em conjunto.	Como o executor e os atuadores foram reimplementados em separado, como especificado e redesenhado nos modelos *TO-BE* (*ver* Figura 39 *e* Figura 40), é possível criar casos de teste mais simples e testar as classes em separado ou em conjunto.
Alternativas obscuras	Modularidade	Como as alternativas de adaptação estão caracterizadas como o recurso política no agente planejador (ver modelo *TO-BE* da Figura 39), isto melhora a modularidade das alternativas que deixam de estar obscuras no código. Isto facilita a adição, alteração e remoção de alternativas de adaptação	Implementamos o agente planejador através da classe *Planner* (Listagem 13). Nesta classe, criamos o método *createStrategy* (linhas 412 a 427) que garante que apenas uma regra de adaptação será aplicada. Esta regra possui o *Profile* (Listagem 18) que será aplicado ao smartphone e as configurações deste *Profile* (Listagem 18) são passadas ao executor (linhas 429 a 447) para atualizar o smartphone. Deste modo, qualquer nova configuração de adaptação de perfil poderá ser adicionada ao *Profile* (Listagem 18) e deverá ser considerada na classe *Planner* (Listagem 13) e *Executor* (Listagem 14) além de criar uma nova classe *Effector* para executar a nova ação.
	Analisabilidade	Como as alternativas de adaptação estão caracterizadas como o recurso política no agente planejador (ver modelo *TO-BE* da	Implementamos o agente planejador através da classe *Planner* (Listagem 13). Nesta classe, criamos o método *createStrategy* que garante que apenas uma regra de adaptação será aplicada.

		Figura 39), isto facilita a compreensão das alternativas que deixam de estar obscuras.	Esta regra possui o *Profile* (Listagem 18) que será aplicado ao smartphone e as configurações deste *Profile* (Listagem 18) são passadas ao executor para atualizar o smartphone. Deste modo, a compreensão dos locais onde podemos adicionar, remover ou alterar alternativas é mais facilmente analisado.
	Modificabilidade	Como as alternativas de adaptação estão caracterizadas como o recurso política no agente planejador (ver modelo *TO-BE* da Figura 39), isto facilita a modificação das alternativas que deixam de estar obscuras.	Implementamos o agente planejador através da classe *Planner* (Listagem 13). Nesta classe, criamos o método *createStrategy* que garante que apenas uma regra de adaptação será aplicada. Esta regra possui o *Profile* (Listagem 18) que será aplicado ao smartphone e as configurações deste *Profile* (Listagem 18) são passadas ao executor para atualizar o smartphone. Deste modo, a modificação das alternativas pode ser realizada em locais específicos e mais fáceis de serem identificados.

Hipótese 3: futuras evoluções de sistemas autoadaptativos podem ser mais amenas após a reengenharia ser aplicada.

Para avaliar a nossa estratégia sob a perspectiva da hipótese 3, fizemos um estudo exploratório para analisar o impacto de algumas tarefas de manutenção na versão original do *PhoneAdapter* e na versão pós-reengenharia. Este mesmo tipo de avaliação foi adotado por Serikawa (SERIKAWA et al., 2016).

Tabela 10 Lista de tarefas de manutenção.

Id	Tarefa	Descrição
1	Adicionar um novo tipo de sensor	Adicionar sensor de fone de ouvido conectado.
2	Mudar a lógica de processamento do sensor	Alterar a aquisição do dado dia da semana para somente enviar o dia da semana quando houver mudança.
3	Configurar frequências diferentes para os sensores	Ajustar a frequência de leitura de dados de GPS para 9 min, ao invés de ser feita a qualquer instante, para economizar bateria.
4	Adicionar novo tipo de adaptação	Adicionar possibilidade de adaptar o tamanho da fonte do smartphone.

Tarefa 1: adicionar sensor de fone de ouvido conectado.

Adicionar novos sensores pode ser uma tarefa comum em sistemas autoadaptativos. Um novo sensor para monitorar se o fone de ouvido está conectado pode ser útil para ajustar automaticamente o volume, por exemplo e desativar o modo vibração. Fizemos esta tarefa no contexto de reengenharia do *PhoneAdapter* para auxiliar na mobilidade urbana de pessoas com limitações visuais (MOURA et al., 2019).

Impacto no sistema original: no sistema original, esta alteração é feita na classe *ContextManager*. A lógica de instanciação do sensor deve ser feita no método *onCreate*. A lógica de leitura do estado do fone de ouvido deve ser feita no método *onHandleIntent* dentro do laço de repetição que agrupa os outros sensores. Também deve ser adicionado mais este dado na lógica de difusão. Por fim, é preciso revisar o código responsável pelo uso de mais este sensor.

Impacto no sistema após-reengenharia: deve ser criada uma nova classe conforme o *tenplate* da Listagem 4 (Seção 4.3). A lógica de instanciação do sensor deve ser feita no método *onCreate* desta nova classe. A lógica de leitura do estado do

fone de ouvido deve ser feita no método *onHandleIntent*, assim como a lógica de difusão do dado.

A mudança no sistema pós-reengenharia é menos invasiva do que no sistema original pois cria-se uma classe nova ao invés de alterar a classe *ContextManager* que possui outros sensores. Além disso, o novo sensor pode ter uma frequência independente de aquisição de dados.

No sistema original, a mudança obriga o novo sensor a ter a mesma frequência de leitura dos demais e torna a ContextManager ainda mais complexa com tantos sensores.

Tarefa 2: alterar a aquisição do dado dia da semana para somente enviar o dia da semana quando houver mudança.

Alterar a lógica de aquisição de dados pode ser importante para minimizar o consumo de recursos, por exemplo. Nesta tarefa hipotética, queremos que o dia da semana seja repassado para os demais elementos se realmente ocorreu uma mudança desde de a última mudança informada. Isso evitará que os demais elementos do mecanismo de adaptação sejam disparados desnecessariamente.

Impacto no sistema original: no sistema original, esta mudança não poderia ser realizada pois os sensores estão oprimidos em um único laço de repetição e a lógica de difusão dos dados envia sempre a lista completa de dados adquiridos. Além disso, o tempo sempre será um valor novo se comparado ao último valor lido e isso é suficiente para gerar um novo contexto independente dos demais valores terem sofrido alteração.

Impacto no sistema após-reengenharia: no novo sistema, seria adicionado um teste de comparação entre o valor lido e a última atualização passada no método *onHandleIntent* da classe *WeekdaySensor*. Caso seja um novo valor de dia da semana, a lógica de difusão do dado é acionada.

Este tipo de ajuste só é possível quando os sensores são independentes um do outro, deste modo a modificação só é possível no sistema pós-reengenharia.

Tarefa 3: ajustar a frequência de leitura de dados de GPS para 9 min ao invés de 2 min para economizar bateria.

Evoluções para minimizar o consumo de recursos são muito importantes. Nesta tarefa, especificamente, ajustamos a frequência de leitura do GPS para ser realizada a cada nove minutos (540000 ms) ao invés de ser feita a qualquer instante.

Impacto no sistema original: no sistema original, esta alteração é feita no método *onCreate* da classe *ContextManager*. É preciso ajustar o tempo de solicitações de atualizações ao configurar o recebimento das atualizações de localização com o método *requestLocationUpdates*. Este método recebe como parâmetros o tipo de provedor a ser utilizado, o tempo mínimo entre as atualizações e a distância mínima percorrida entre atualizações e o objeto do tipo *LocationListener* que receberá as atualizações. Deste modo, informaremos o valor 540000 no parâmetro tempo mínimo entre atualizações.

Impacto no sistema após-reengenharia: no sistema pós-reengenharia, a atualização é no mesmo método, porém deverá ser feito no método *onCreate* a classe *GPSSensor*. Além disso, ajustaremos o tempo de frequência de envio dos dados de GPS para 540000 ms também. Para evitar enviar atualizações desnecessárias.

Ainda que as atualizações sejam similares, a identificação do local de alteração pode ser mais fácil na classe que trata unicamente a leitura de dados de GPS. Adicionalmente, no sistema novo, podemos também ajustar o tempo de envio dos valores referente aos dados de GPS para nove minutos, uma vez que os sensores estão caracterizados e individualizados.

Tarefa 4: adicionar possibilidade de adaptar o tamanho da fonte do smartphone.

Adicionar novas possibilidades de adaptação podem ser tarefas frequentes em sistemas autoadaptativos. Em (OLIVEIRA; MOURA; LEITE, 2018), realizamos a reengenharia do *PhoneAdapter* para adicionar requisitos de acessibilidade conforme o contexto do usuário através de um SIG hibrido do RNF de consciência e de acessibilidade de *software*. Nesta reengenharia, um dos requisitos foi adicionar a adaptação do tamanho da fonte do smartphone conforme o contexto escolhido pelo usuário.

Impacto no sistema original: a reengenharia foi realizada sobre o sistema original após utilizar o nosso subprocesso recuperar para identificar as metas flexíveis

de consciência operacionalizadas no código fonte. Quanto ao mecanismo de adaptação, a nova adaptação foi adicionada no método *onReceive* da classe *MyBroadcastReceiver* e nos atributos da classe Profile, ambas classes internas da classe *AdaptationManager*.

Impacto no sistema após-reengenharia: no novo sistema, esta alteração seria realizada nos atributos da classe Profile, que deixou de ser uma classe interna para ser uma classe pública, no método *createStrategy* da classe *Planner*, no método *executeStrategy* da classe Executor e na adição de um método na classe *Effector* referente a execução do ajuste do tamanho da fonte.

O ajuste no sistema original é mais complexo devido ao baixo grau de modularidade do sistema. Todo o código referente a análise, planejamento e execução estão nos métodos *onReceive* e *checkRules* da classe *AdaptationManager* a qual possui um grande número de linhas. No sistema novo, as alterações são em locais de mais fácil localização devido a nova modularidade do sistema.

6
Conclusão

Neste capítulo, apresentamos as conclusões referente a este trabalho aliadas as contribuições e limitações da pesquisa. Propomos também algumas sugestões de trabalhos futuros.

6.1.Considerações Finais

Esta pesquisa apresentou uma estratégia para a reengenharia de sistemas autoadaptativos guiada pelo requisito não funcional de consciência de *software*. A estratégia é composta por quatro subprocessos, a saber: **recuperar** um modelo de metas *AS-IS* com os RNF de consciência de *software* operacionalizados; **especificar** o modelo *TO-BE* em direção ao padrão MAPE através do reuso de *SRconstructs* ; **redesenhar** o modelo *TO-BE* adicionando detalhes de tecnologia que serão utilizados na reimplementação do mecanismo de autoadaptação do sistema e das operacionalizações de consciência; e a reimplementação do sistema conforme algumas *guidelines* e *templates* sugeridos.

Conforme apresentado, sistemas autoadaptativos têm crescido em número e complexidade. Apesar de pesquisadores buscarem soluções que amenizem a construção de tais sistemas, o uso dos padrões de referência recomendados pela literatura (e.g.: GORE (ALI; DALPIAZ; GIORGINI, 2010; BARESI; PASQUALE; SPOLETINI, 2010; CUNHA, 2014; SOUZA; MYLOPOULOS, 2012) e MAPE (IBM, 2006)) não é trivial. Tamanha complexidade tem levado a construção de soluções com problemas estruturais e qualitativos recorrentes na literatura (SAN MARTÍN et al., 2020; SERIKAWA et al., 2016) que dificultam sua evolução. Adicionalmente, poucos trabalhos abordam estratégias de evolução com vistas a reparar tais problemas, como as que são sugeridas em (SAN MARTÍN et al., 2020; SERIKAWA et al., 2016).

Uma vez que acreditamos que a evolução deve ser guiada desde requisitos para que haja uma consistência entre as metas, arquitetura e implementação, o maior esforço desta pesquisa se concentrou na criação de uma estratégia orientada a metas para guiar a reorganização de sistemas legados autoadaptativos em direção padrão MAPE, promovendo a remoção dos problemas recorrentes na literatura acerca da implementação do MAPE e mantendo a rastreabilidade entre o modelo de metas, a arquitetura e a implementação.

Esta estratégia nos deu a oportunidade de adquirir a consistência entre as metas, a arquitetura e a implementação, que é reconhecido como um problema em aberto conforme apresentação da pesquisadora Liliana Pasquale na premiação MIT (*Must Influential Paper Award*) no RE'20 (PASQUALE, 2020). Para tal, utilizamos o RNF de consciência de *software* como um dos grandes pilares da estratégia. Como resultado, três dos quatro subprocessos da estratégia são baseados em modelos de metas com reuso do catálogo de consciência de *software*. Por fim, o quarto e último subprocesso é guiado pelos modelos de metas produzidos nos subprocessos anteriores além dos *guidelines* e *templates* sugeridos.

6.2.Contribuições

As contribuições desta pesquisa são destacadas com base nos pontos em aberto nos trabalhos relacionados que foram apresentados no Capítulo 1.

Uma estratégia sistemática de reengenharia de sistemas auto adaptativos guiada pelo requisito não funcional de consciência de software: a nossa estratégia de reengenharia relaciona subprocessos bem definidos, executados com critérios, métodos e técnicas estabelecidas com vistas a apoiar a reengenharia de sistemas autoadaptativos guiando-se pelo RNF de consciência de *software*. A estratégia cobre desde o resgate da intencionalidade do sistema alvo com as metas flexíveis de consciência de *software* operacionalizadas, passando pela especificação do novo sistema em direção ao padrão MAPE e pelo redesenho do novo sistema com a seleção das tecnologias que serão utilizadas na reimplementação e na revisão das operacionalizações de consciência, até a sugestão de *guidelines* e *templates* para a reimplementação do novo sistema. Nossa estratégia foi elaborada para realinhar os sistemas

autoadaptativos com as boas práticas recomendadas pela literatura para tais sistemas (i.e.: GORE e MAPE), removendo problemas recorrentes na implementação do MAPE que foram apontados pela literatura. É importante ressaltar que a nossa estratégia é parcialmente automatizável.

Apoio à manutenção da consistência entre metas, arquitetura e implementação: A estratégia apresentada permite um realinhamento de sistemas autoadaptativos com suas metas e com o padrão arquitetural MAPE e mantém uma rastreabilidade para o código, auxiliando a manutenção da consistência entre metas, arquitetura e código. O realinhamento é feito ao longo dos subprocessos propostos, onde o realinhamento com o modelo de metas é feito no subprocesso recuperar e o realinhamento com a arquitetura MAPE é feito nos subprocessos especificar e redesenhar. A rastreabilidade é mantida através das heurísticas de mapeamento de código OO para i* e do conjunto de anotações de código framework JiStar que são utilizados ao longo dos subprocessos recuperar e reimplementar.

Um conjunto de heurísticas para a recuperação de modelos de metas em i* para código OO: criamos um conjunto de heurísticas para o mapeamento automatizado de elementos do paradigma OO para modelos de metas i*. Este conjunto de heurísticas (MOURA, 2017), apresentado em versão revisada e estendida aqui neste trabalho, é utilizado para resgatar a intencionalidade de sistemas legados OO de modo automatizado. Apesar de ter sido criado no contexto da reengenharia de sistemas autoadaptativos, este conjunto de heurísticas pode ser aplicado a outros tipos de sistemas OO.

Estratégia para o mapeamento de RNFs: criamos uma estratégia para mapear RNFs operacionalizados no código de sistemas legados que é baseada no reuso do conhecimento presente em catálogos de método modelados através do SIG proposto no NFR framework (CHUNG et al., 2000). Este tipo de catálogo permite a organização de técnicas de desenvolvimento. Além disso, criamos anotações de código para que o engenheiro de *software* possa indicar operacionalizações específicas no código que não estejam adicionadas ao catálogo. Esta estratégia é parte do subprocesso recuperar da nossa estratégia, onde a utilizamos para identificar RNFs de consciência de *software*. Esta estratégia pode ser utilizada para identificar outros

RNFs catalogados e/ou anotados no código fonte, além do RNF de consciência de *software*.

JiStar - Um framework para manter* a rastreabilidade de código OO Java para modelos de metas i: o *framework* JiStar (MOURA; LEITE, 2020) permite manter a rastreabilidade entre o código OO Java e modelos de metas i* de modo semiautomatizado. Este *framework* é composto por um conjunto de anotações ("metainformações") e um exportador de modelos i*. É possível exportar modelos de metas i* nos formatos iStarML, PiStar e HTML, utilizando o nosso conjunto de heurísticas, as anotações e/ou ambos. O *framework* JiStar está disponível no github em https://github.com/RE-Projects/JiStar. Em nosso estudo de viabilidade do *framework* JiStar com base no aplicativo *RioBus*, identificamos indício de que este *framework* pode auxiliar no resgate da intencionalidade de sistemas OO em Java, mantendo uma rastreabilidade entre o código e o modelo de metas.

NFRJson – Um formato leve para representar Softgoal Interdependency Graphs: o NFRJson permite representar *Sofgoal Interdependency Graphs* (SIGs) em formato json que é mais leve que xml, por exemplo, e é compreensível por máquinas e seres humanos. O formato é proposto através de um json *schema* que está disponível no github em https://github.com/RE-Projects/NFRJson.

***Aplicação da estratégia de reengenharia*:** fizemos uso da nossa estratégia, ao longo desta pesquisa, com vistas a realizar a reengenharia de sistemas autoadaptativos para diferentes finalidades. Em (OLIVEIRA; MOURA; LEITE, 2018), realizamos a reengenharia do *PhoneAdapter* para adicionar novos requisitos de acessibilidade de acordo com a necessidade de cada usuário. Para isto, fizemos uso das nossas heurísticas (MOURA, 2017) e do subprocesso recuperar para resgatar as metas flexíveis de consciência operacionalizadas em um modelo de metas i**AS-IS*, por conseguinte, criamos um SIG de correlação entre o SIG de consciência (CUNHA, 2014) de *software* e o SIG de acessibilidade (OLIVEIRA et al., 2016) para a reespecificação do sistema alvo da reengenharia. Posteriormente, implementamos os novos requisitos de acessibilidade conforme a combinação de qualidades selecionadas no SIG hibrido. Em (MOURA et al., 2019), fizemos uso da nossa estratégia de reengenharia completa com a primeira versão de *SRconstructs* baseados no MAPE, ao longo de uma nova reengenharia do *PhoneAdapter*, com vistas a mostrar

que podemos utilizar o RNF a consciência de *software* na superação de alguns desafios da IoT ("*Internet of Things*") para desenvolver soluções para pessoas com limitações. Tais desafios foram propostos por Domingo (DOMINGO, 2012) e Khedo (KHEDO, 2006). Os resultados obtidos nestes trabalhos contribuíram para a melhoria da estratégia.

6.3. Limitações do Trabalho

A estratégia apresentada possui algumas limitações já identificadas.

Necessidade de conhecimento do domínio por parte do engenheiro de software: embora a nossa estratégia possua heurísticas para a extração automática de um modelo de metas, é imprescindível que o engenheiro de *software* tenha algum conhecimento do domínio e da linguagem de programação utilizada pois várias operacionalizações de consciência podem não estar catalogadas como tal. Além disso, uma das fontes de informação importantes no uso das anotações de código do *framework* JiStar é o conhecimento tácito dos especialistas do sistema.

Tipo de Paradigma: também limitamos a nossa estratégia ao paradigma OO, o que nos permitiu criar um conjunto de heurísticas de mapeamento entre o *framework* i* e representações OO. Como o *framework* i* é bastante difundido para a modelagem de SMA e estes, por sua vez, podem ser implementados sobre plataformas OO como JADEX(BRAUBACH; POKAHR; LAMERSDORF, 2004) e JADE (BELLIFEMINE et al., 2002). Apesar disso, acreditamos que seja possível criar um conjunto de heurísticas de outros paradigmas para elementos do *framework* i*. Nossa crença é baseada no trabalho (YU et al., 2005) que recupera modelos de metas em GSP (HUI; LIASKOS; MYLOPOULOS, 2003) a partir de sistemas legados estruturados e não estruturados para fins de reengenharia. Ainda que o GSP e *framework* i* sejam linguagem diferentes, as duas permitem representar a intencionalidade do *software* a seu modo.

Generalidade da estratégia: embora tenhamos buscado a generalidade no mapeamento de elementos do paradigma OO para elementos do *framework* i*, atingimos uma generalidade parcial. A estratégia é voltada para sistemas autoadaptativos OO,

mas o seu ferramental de apoio é para sistemas escritos em Java e criamos *templates* para a reimplementação do código com base nessa linguagem.

Esforço na avaliação e evolução dos modelos gerados: os modelos i* *AS-IS* e *TO-BE* trabalhados a longo dos subprocessos da estratégia de reengenharia têm uma tendência a serem extensos à medida que aplicamos a estratégia em sistemas de médio e grande porte ou sistemas que possuem um grande número de participantes no mecanismo de autoadaptação, conforme observamos em nossos estudos. Modelos extensos são difíceis de serem visualizados, compreendidos e evoluídos. Para mitigar tal dificuldade, sugerimos o uso da estrutura canônica *SDsituation* (PADUA; OLIVEIRA; CYSNEIROS, 2006), que define um bloco de elementos de dependência com intencionalidade situacional compartilhada, pois os modelos poderão ser analisados e evoluídos parcialmente ao invés de um único modelo por inteiro.

Esforço na criação de SIGs em NFRJson sem uma ferramenta de modelagem*:* apesar de propormos um formato para a especificação de SIGs em json (i.e.: NFRJson) que pode ser validado pelo nosso NFRJson *schema*, a construção de tais arquivos exige um grande esforço a *depender* do tamanho do SIG que se desejar representar. Deste modo, entendemos que um suporte ferramental seria de grande valia.

6.4. Trabalhos Futuros

Os estudos realizados e a constante revisão da literatura nos permitiram identificar oportunidades de melhoria e possibilidades de extensão desta pesquisa. Estas podem ser tratadas através dos seguintes trabalhos futuros.

Definição de um conjunto de heurísticas para mapeamento de outros paradigmas: uma possível oportunidade de tornar o nosso trabalho mais abrangente é criar um conjunto de heurísticas para realizar o mapeamento de outros paradigmas de programação para os elementos do *framework* i*, como por exemplo, o paradigma estruturado.

Criar plugins para uso do framework JiStar em IDEs de modo integrado: apesar do *framework* i* poder ser utilizado como uma biblioteca (jar) no sistema, temos a

intuição de que pode ser produtivo e incentivador, para engenheiros de *software*, ter à disposição um *wizard* para aplicar as anotações. Os *wizards* são de uso opcional, de tal modo que engenheiros de *software*, após adquirirem experiência com o uso do *framework* JiStar, possam optar pelo seu uso ou não.

Geração de código a partir do modelo TO-BE: como sugerimos *templates* de classe para a implementação dos agentes participantes do mecanismo de autoadaptação conforme modelo MAPE, seria possível, em alguma ferramenta de modelagem com suporte a i*, gerar o esqueleto destas classes. A *depender* do número elementos, esta solução pode poupar bastante esforço dos engenheiros de *software* que participam do time de reimplementação.

Evolução do framework de rastreabilidade de código OO para modelos i* para outras linguagens de programação: Escolhemos a linguagem Java pela nossa familiaridade/cultura e pelo seu amplo uso na indústria e na academia. Entretanto, existe a oportunidade de estendermos o nosso *framework* para outras linguagens de programação OO como .Net, Python, entre outras.

Conversão de modelos de SIG para NFRJson: apesar da vantagem de poder descrever um SIG em um formato compreensível por seres humanos e máquinas, a *depender* do tamanho do SIG, esta pode ser uma tarefa onerosa. Quando criamos as operacionalizações em Android para o SIG de consciência de *software*, tivemos essa percepção. Idealizamos então uma oportunidade de melhoria que é a conversão de modelos de SIG para o formato NFRJson. Deste modo, a especificação do SIG deveria ser feita em uma ferramenta visual e posteriormente este poderá ser convertido/exportado para o formato NFRJson. Uma das ferramentas que encontramos na literatura para a modelagem de SIG é a *RE-Tools* (SUPAKKUL; CHUNG, 2012), que é um toolkit para modelagem de requisitos em diferentes notações incluindo o NFR *framework*. Apesar da sua compatibilidade apenas com a versão 1.0 da ferramenta de modelagem StarUML[28], esta poderia ser uma opção de ferramenta para criarmos uma forma de exportar o SIG para o formato NFRJson.

[28] http://sourceforge.net/projects/staruml/

Referências Bibliográficas

ABUSETA, Y.; SWESI, K. Design patterns for self adaptive systems engineering. **arXiv preprint arXiv:1508.01330**, 2015.

ALENCAR, F. M. et al. **New Mechanism for the Integration of Organizational Requirements and Object Oriented Modeling.** WER. **Anais**...2003

ALI, R.; DALPIAZ, F.; GIORGINI, P. A goal-based framework for contextual requirements modeling and analysis. **Requirements Engineering**, v. 15, n. 4, p. 439–458, 2010.

ARCAINI, P.; RICCOBENE, E.; SCANDURRA, P. **Modeling and analyzing MAPE-K feedback loops for self-adaptation**. Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. **Anais**...IEEE Press, 2015

BARESI, L.; PASQUALE, L.; SPOLETINI, P. **Fuzzy goals for requirements-driven adaptation**. 2010 18th IEEE International Requirements Engineering Conference. **Anais**...IEEE, 2010

BELLIFEMINE, F. et al. Jade programmer's guide. **Jade version**, v. 3, p. 13–39, 2002.

BIEGEL, G.; CAHILL, V. **A framework for developing mobile, context-aware applications**. Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. Proceedings of the. **Anais**...IEEE, 2004

BOEHM, B. W.; BROWN, J. R.; LIPOW, M. **Quantitative evaluation of software quality**. Proceedings of the 2nd international conference on Software engineering. **Anais**...1976

BRABERMAN, V. et al. **MORPH: a reference architecture for configuration and behaviour self-adaptation**. Proceedings of the 1st International Workshop on Control Theory for Software Engineering. **Anais**...ACM, 2015

BRAUBACH, L.; LAMERSDORF, W.; POKAHR, A. Jadex: Implementing a bdi-infrastructure for jade agents. 2003.

BRAUBACH, L.; POKAHR, A.; LAMERSDORF, W. **Jadex: A short overview**. Main Conference Net. ObjectDays. **Anais**...2004

BRESCIANI, P. et al. Tropos: An agent-oriented software development methodology. **Autonomous Agents and Multi-Agent Systems**, v. 8, n. 3, p. 203–236, 2004.

CÁMARA, J. et al. Incorporating architecture-based self-adaptation into an adaptive industrial software system. **Journal of Systems and Software**, v. 122, p. 507–523, 2016.

CARES, C. et al. Towards interoperability of i* models using iStarML. **Computer Standards & Interfaces**, v. 33, n. 1, p. 69–79, 2011.

CASTRO, J. et al. Integration of i* and object-oriented models. **Social modeling for requirements engineering**, p. 457–484, 2011.

CASTRO, J.; ALENCAR, F.; CYSNEIROS, G. Closing the gap between organizational requirements and object oriented modeling. **Journal of the Brazilian Computer Society**, v. 7, n. 1, p. 05–16, 2000.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse engineering and design recovery: A taxonomy. **IEEE software**, v. 7, n. 1, p. 13–17, 1990.

CHUNG, L. et al. Non-functional requirements in software engineering (the kluwer international series in software engineering volume 5). 1999.

CHUNG, L. et al. Softgoal interdependency graphs. In: **Non-Functional Requirements in Software Engineering**. [s.l.] Springer, 2000. p. 47–88.

CHUNG, L.; DO PRADO LEITE, J. C. S. On non-functional requirements in software engineering. In: **Conceptual modeling: Foundations and applications**. [s.l.] Springer, 2009. p. 363–379.

CHUNG, L.; NIXON, B. A.; YU, E. **Using non-functional requirements to systematically select among alternatives in architectural design**. Proc. 1st Int. Workshop on Architectures for Software Systems, Seattle, Washington. **Anais**...1995

CUNHA, H. DE S. **Desenvolvimento de Software Consciente com Base em Requisitos**. [s.l.] PUC-Rio, 2014.

CUNHA, H.; DO PRADO LEITE, J. C. S. **Reusing non-functional patterns in i∗ modeling**. Requirements Patterns (RePa), 2014 IEEE 4th International Workshop on. **Anais**...IEEE, 2014Disponível em: <http://ieeexplore.ieee.org/abstract/document/6894840/>. Acesso em: 11 abr. 2017

DALPIAZ, F.; FRANCH, X.; HORKOFF, J. istar 2.0 language guide. **arXiv preprint arXiv:1605.07767**, 2016a.

DALPIAZ, F.; FRANCH, X.; HORKOFF, J. **iStar tutorial online**. Disponível em: <https://www.dropbox.com/s/4l2k4tbywb8wekk/iStar-tutorial-online.pdf?dl=0>. Acesso em: 27 mar. 2020b.

DALPIAZ, F.; GIORGINI, P.; MYLOPOULOS, J. **An architecture for requirements-driven self-reconfiguration**. International Conference on Advanced Information Systems Engineering. **Anais**...Springer, 2009Disponível em: <http://link.springer.com/chapter/10.1007/978-3-642-02144-2_22>. Acesso em: 19 abr. 2017

DARDENNE, A.; VAN LAMSWEERDE, A.; FICKAS, S. Goal-directed requirements acquisition. **Science of computer programming**, v. 20, n. 1–2, p. 3–50, 1993.

DE LEMOS, R. et al. Software engineering for self-adaptive systems: A second research roadmap. In: **Software Engineering for Self-Adaptive Systems II**. [s.l.] Springer, 2013. p. 1–32.

DIAZ, O. Object-oriented systems: a cross-discipline overview. **Information and Software Technology**, v. 38, n. 1, p. 47–57, 1996.

DOMINGO, M. C. An overview of the Internet of Things for people with disabilities. **Journal of Network and Computer Applications**, v. 35, n. 2, p. 584–596, 2012.

FRANCO BEDOYA, Ó. H. et al. **istarjson: A lightweight data-format for i* models**. iStar 2016: Proceedings of the Ninth International i* Workshop, co-located with the 24rd International Requirements Engineering Conference (RE 2016): Beijing, China, September 12-13, 2016. **Anais**...CEUR-WS. org, 2016

FUXMAN, A. et al. Specifying and analyzing early requirements in Tropos. **Requirements Engineering**, v. 9, n. 2, p. 132–150, 2004.

GAMMA, E. et al. **Design patterns: Abstraction and reuse of object-oriented design**. European Conference on Object-Oriented Programming. **Anais**...Springer, 1993

GAMMA, E. **Design patterns: elements of reusable object-oriented software**. [s.l.] Pearson Education India, 1995.

GARLAN, D. et al. Rainbow: Architecture-based self-adaptation with reusable infrastructure. **Computer**, v. 37, n. 10, p. 46–54, 2004.

GARLAN, D.; SCHMERL, B.; CHENG, S.-W. Software architecture-based self-adaptation. In: **Autonomic computing and networking**. [s.l.] Springer, 2009. p. 31–55.

GIORGINI, P.; MYLOPOULOS, J.; SEBASTIANI, R. Goal-oriented requirements analysis and reasoning in the tropos methodology. **Engineering Applications of Artificial Intelligence**, v. 18, n. 2, p. 159–171, 2005.

GONÇALVES, E. et al. **A Catalogue of iStar Extensions.** WER. **Anais**...2018

GU, T.; PUNG, H. K.; ZHANG, D. Q. **A middleware for building context-aware mobile services**. 2004 IEEE 59th Vehicular Technology Conference. VTC 2004-Spring (IEEE Cat. No. 04CH37514). **Anais**...IEEE, 2004

HORKOFF, J. et al. **Goal-oriented requirements engineering: A systematic literature map**. 2016 IEEE 24th International Requirements Engineering Conference (RE). **Anais**...IEEE, 2016

HUI, B.; LIASKOS, S.; MYLOPOULOS, J. Goal skills and preference framework. **RE'03**, p. 117–126, 2003.

IBM. An architectural blueprint for autonomic computing. **IBM White Paper**, v. 31, n. 2006, p. 1–6, 2006.

IGLESIA, D. G. D. L.; WEYNS, D. MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, v. 10, n. 3, p. 15, 2015.

ISO/IEC 25010:2011. Disponível em: <https://tinyurl.com/y6e6wru2>. Acesso em: 1 fev. 2019.

KANDÉ, M. M. **A concern-oriented approach to software architecture**. [s.l.] Technische Universität Berlin, 2003.

KAZMAN, R.; WOODS, S. G.; CARRIÈRE, S. J. **Requirements for integrating software architecture and reengineering models: CORUM II**. Proceedings Fifth Working Conference on Reverse Engineering (Cat. No. 98TB100261). **Anais**...IEEE, 1998

KHEDO, K. K. **Context-aware systems for mobile and ubiquitous networks**. Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies, 2006. ICN/ICONS/MCL 2006. International Conference on. **Anais**...IEEE, 2006

KRAMER, J.; MAGEE, J. **Self-managed systems: an architectural challenge**. 2007 Future of Software Engineering. **Anais**...IEEE Computer Society, 2007

LEITE, J. C. S. DO P. Working results on software re-engineering. **ACM SIGSOFT Software Engineering Notes**, v. 21, n. 2, p. 39–44, 1996.

LEWIS, T. L.; PÉREZ-QUIÑONES, M. A.; ROSSON, M. B. **A comprehensive analysis of object-oriented design: towards a measure of assessing design ability**. 34th Annual Frontiers in Education, 2004. FIE 2004. **Anais**...IEEE, 2004

MAHMOUD, A. **An information theoretic approach for extracting and tracing non-functional requirements**. Requirements Engineering Conference (RE), 2015 IEEE 23rd International. **Anais**...IEEE, 2015

MALTA, Á. et al. **iStarTool: Modeling Requirements using the i* Framework.** IStar. **Anais**...2011

MISELDINE, P.; TALEB-BENDIAB, A.; RANDLES, M. Rainbow: An approach to facilitate restorative functionality within distributed autonomic systems. **Proceedings of PGNet**, 2005.

MOURA, A. M. DA M. **Awareness Driven Software Reengineering**. Requirements Engineering Conference (RE), 2017 IEEE 25th International. **Anais**...IEEE, 2017

MOURA, A. M. DA M. et al. **Improving Urban Mobility for the Visually Impaired using the Awareness Quality**. Proceedings of the XVIII Brazilian Symposium on Software Quality. **Anais**...2019

MOURA, A. M. DA M.; LEITE, J. C. S. DO P. **JiStar - Rastreabilidade Entre Código Java e Modelos de Metas i***. . In: WER. 2020

OIVO, M. et al. **Software engineering research strategy: Combining experimental and explorative research (eer)**. International Conference on Product Focused Software Process Improvement. **Anais**...Springer, 2004

OLIVEIRA, A. DE P. A.; LEITE, J. C. S. DO P.; CYSNEIROS, L. M. **Método ERi* c-Engenharia de Requisitos Intencional.** WER. **Anais**...2008

OLIVEIRA, A. P. A. Engenharia de Requisitos Intencional: Um método de elicitação, modelagem e Análise de Requisitos. **PhD Tese, PUC-Rio**, 2008.

OLIVEIRA, A. P. A. et al. **i* Diagnoses: A Quality Process for Building i* Models**. PROCEEDINGS OF THE FORUM AT THE CAISE'08 CONFERENCE. **Anais**...Citeseer, 2008

OLIVEIRA, R. F. DE; MOURA, A. M. DA M.; LEITE, J. C. S. P. **Reengineering for Accessibility: A Strategy Based on Software Awareness**. Proceedings of the 17th Brazilian Symposium on Software Quality. **Anais**...ACM, 2018

OLIVEIRA, R. et al. **Eliciting accessibility requirements an approach based on the NFR framework**. Proceedings of the 31st Annual ACM Symposium on Applied Computing. **Anais**...ACM, 2016

PADUA, A.; OLIVEIRA, A.; CYSNEIROS, L. M. Defining strategic dependency situations in requirements elicitation. 2006.

PASQUALE, L. **Fuzzy Goals for Requirements-Driven Adaptation "Most Influential Paper" from the 18th IEEE International Requirements Engineering Conference (RE 2010)**Zurich, Switzerland, 3 set. 2020. Disponível em: <https://drive.google.com/file/d/1JRyGktbuN-RHUz7KSGM5a_jJNecvSKNI3/view>. Acesso em: 9 ago. 2020

PIMENTEL, J.; CASTRO, J. **pistar tool–a pluggable online tool for goal modeling**. 2018 IEEE 26th International Requirements Engineering Conference (RE). **Anais**...IEEE, 2018

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

ROZANSKI, N.; WOODS, E. **Software systems architecture: working with stakeholders using viewpoints and perspectives**. [s.l.] Addison-Wesley, 2011.

SAN MARTÍN, D. et al. **Characterizing Architectural Drifts of Adaptive Systems**. 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). **Anais**...IEEE, 2020

SAYÃO, M.; DO PRADO LEITE, J. C. S. Rastreabilidade de requisitos. **RITA**, v. 13, n. 1, p. 57–86, 2006.

SERIKAWA, M. A. et al. **Towards the Characterization of Monitor Smells in Adaptive Systems**. Software Components, Architectures and Reuse (SBCARS), 2016 X Brazilian Symposium on. **Anais**...IEEE, 2016Disponível em: <http://ieeexplore.ieee.org/abstract/document/7789839/>. Acesso em: 11 abr. 2017

SERRANO, M.; LEITE, J. C. S. DO P. **Development of Agent-Driven Systems: from i* Architectural Models to Intentional Agents Code.** iStar. **Anais**...Citeseer, 2011a

SERRANO, M.; LEITE, J. C. S. DO P. **Capturing transparency-related requirements patterns through argumentation**. Requirements Patterns (RePa), 2011 First International Workshop on. **Anais**...IEEE, 2011bDisponível em: <http://ieeexplore.ieee.org/abstract/document/6046723/>. Acesso em: 16 out. 2017

SOUZA, S.; MYLOPOULOS, J. Requirements-based software system adaptation. 2012.

SUPAKKUL, S. et al. **An NFR pattern approach to dealing with NFRs**. Requirements Engineering Conference (RE), 2010 18th IEEE International. **Anais**...IEEE, 2010

SUPAKKUL, S.; CHUNG, L. **The RE-Tools: A multi-notational requirements modeling toolkit**. 2012 20th IEEE International Requirements Engineering Conference (RE). **Anais**...IEEE, 2012

SUPRIANA, I.; SURENDRO, K. Self-adaptive Software Modeling Based on Contextual Requirements. **Telkomnika**, v. 16, n. 3, 2018.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. Introdução à engenharia de software experimental. 2002.

WALCZAK, S. Knowledge acquisition and knowledge representation with class: the object-oriented paradigm. **Expert Systems with Applications**, v. 15, n. 3–4, p. 235–244, 1998.

YU, E. Modelling strategic relationships for process reengineering. 1995.

YU, Y. et al. **Reverse engineering goal models from legacy code**. Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on. **Anais**...IEEE, 2005

Apêndice 1 Classes Criadas no *PhoneAdapter*

A1.1. Sensores

```
582.  @Actor(name = "TimeSensor",type = ActorType.AGENT)
583.  @Goal(name = "That time be acquired",actor = "TimeSensor",description = "")
584.  @Softgoal(name = "Time Awareness",actor = "TimeSensor")
585.  public class TimeSensor extends IntentService {
586.
587.     private boolean mStop;
588.     private static final String TAG = "TimeLog";
589.     private String mTime;
590.     private Calendar mCal;
591.     private Handler mHandler;
592.     private SimpleDateFormat mTimeFormat;
593.     private static boolean running;
594.
595.     public TimeSensor() {
596.        super("TimeSensor");
597.     }
598.
599.     public void onCreate() {
600.        super.onCreate();
601.        mTimeFormat=new SimpleDateFormat("HH:mm:ss");
602.        mHandler=new Handler();
603.                    IntentFilter iFilter = new IntentFilter();
604.                    iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
605.        mStop=false;
606.        TimeSensor.running = true;
607.     }
608.
609.
610.     @Override
611.  @Contribution(type = ContributionType.HELP,softgoal = "Time Awareness")
612.     @MeansEnd(end = "That time be acquired",endType = MeansEndType.GOAL)
613.     protected void onHandleIntent(Intent intent) {
614.        while(!mStop){
615.           mCal=Calendar.getInstance();
616.           mTime=mTimeFormat.format(mCal.getTime());
617.
618.
619.           mHandler.post(new Runnable() {
620.              @Override
621.              public void run() {
622.                 /* broadcast new context*/
623.                 Intent i=new Intent();
624.                 i.setAction(ContextName.TIME);
625.                 i.putExtra(ContextName.TIME,mTime);
626.                 sendBroadcast(i);
627.
628.
629.                 Log.i(TAG, mTime.toString());
630.
631.
632.              }
```

```
633.              });
634.              try{
635.                 Thread.sleep(60000);
636.              } catch(Exception e){
637.                 Log.e("PhoneAdapter.error", "Thread sleep exception");
638.              }
639.           }
640.        }
641.        public static boolean isRunning(){
642.           return TimeSensor.running;
643.        }
644.
        }

645.     @Actor(name = "WeekdaySensor",type = ActorType.AGENT)
646.     @Goal(name = "That Weekday be acquired",actor = "WeekdaySensor",description = "")
647.     @Softgoal(name = "Time Awareness",topic = "Weekday",actor =
648.     "WeekdaySensor")public class WeekdaySensor extends IntentService {
649.        private String mWeekday;
650.        private String newWeekday;
651.        private Calendar mCal;
652.        private Handler mHandler;
653.        private boolean mStop;
654.        private static boolean running;
655.        private static final String TAG = "WeekdayLog";
656.
657.        public WeekdaySensor() {
658.           super("WeekdaySensor");
659.        }
660.
661.        @Override
662.        public void onCreate() {
663.           super.onCreate();
664.           mCal = Calendar.getInstance();
665.           mStop = false;
666.           mHandler = new Handler();
667.           mWeekday = null;
668.           newWeekday = null;
669.           WeekdaySensor.running = true;
670.        }
671.
672.        @Override
673.     @Contribution(type = ContributionType.HELP,softgoal = "Time Awareness")
674.        @MeansEnd(end = "That Weekday be acquired",endType = MeansEndType.GOAL)
675.        protected void onHandleIntent(Intent intent) {
676.           while (!mStop) {
677.              mCal = Calendar.getInstance();
678.              switch (mCal.get(Calendar.DAY_OF_WEEK)) {
679.                 case 1: newWeekday = "sunday"; break;
680.                 case 2: newWeekday = "monday"; break;
681.                 case 3: newWeekday = "tuesday";break;
682.                 case 4: newWeekday = "wednesday"; break;
683.                 case 5: newWeekday = "thursday"; break;
684.                 case 6: newWeekday = "friday";  break;
685.                 case 7: newWeekday = "saturday"; break;
686.                 default:
687.              }
688.
689.              if (mWeekday == null || !newWeekday.equalsIgnoreCase(mWeekday)) {
690.                 mWeekday = newWeekday;
```

```
691.                mHandler.post(new Runnable() {
692.                   @Override
693.                   public void run() {
694.                      /* broadcast new context*/
695.                      Intent i = new Intent();
696.                      i.setAction(ContextName.WEEKDAY);
697.                      i.putExtra(ContextName.WEEKDAY, mWeekday);
698.                      sendBroadcast(i);
699.
700.                   }
701.                });
702.             }
703.             try {
704.                Thread.sleep(360000);
705.             } catch (Exception e) {
706.                Log.e("PhoneAdapter.error", "Thread sleep exception");
707.             }
708.          } }
709.       public static boolean isRunning() {
710.          return WeekdaySensor.running;
711.       }
712.    }

713.    @Actor(name = "GPSSensor",type = ActorType.AGENT)
714.    @Goal(name = "That data based on GPS be acquired",actor = "GPSSensor",description
715.    = "")
716.    @Softgoal(name = "Location",topic = "User",actor = "GPSSensor")
717.    @Softgoal(name = "Computing environment",topic = "GPS avaiable",actor =
718.    "GPSSensor")
719.    @Softgoal(name = "User awareness",topic="Speed",actor = "GPSSensor")
720.    public class GPSSensor extends IntentService implements LocationListener{
721.
722.       private String mLocation;
723.       private double mSpeed;
724.       private long mLastTime;
725.       private String mLastLocation;
726.       private boolean mStop;
727.       private Handler mHandler;
728.       private static boolean running;
729.       private static final String TAG = "GPSLog";
730.       private LocationListener mLocListener;
731.       private LocationManager mLocManager;
732.       private boolean mGpsAvailable;
733.
734.
735.       public GPSSensor() {
736.          super("GPSSensor");
737.       }
738.
739.       @Override
740.       public void onCreate() {
741.          super.onCreate();
742.          super.onCreate();
743.          mLocManager = (LocationManager) getSystemService(LOCATION_SERVICE);
744.          mLocListener = this;
745.          if (ContextCompat.checkSelfPermission(this,
746.    Manifest.permission.ACCESS_FINE_LOCATION) ==
747.    PackageManager.PERMISSION_GRANTED
748.                && ContextCompat.checkSelfPermission(this,
749.    Manifest.permission.ACCESS_COARSE_LOCATION) ==
```

```
750.    PackageManager.PERMISSION_GRANTED) {
751.            mLocManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0,
752.    0, mLocListener);
753.        }
754.
755.        mStop=false;
756.        mHandler=new Handler();
757.        mLocation = new String();
758.
759.        GPSSensor.running = true;
760.    }
761.
762.    public int onStartCommand(Intent intent, int flags, int startId) {
763.    //      mGoogleApiClient.connect();
764.        return super.onStartCommand(intent, flags, startId);
765.
766.    }
767.
768.    public void onDestroy() {
769.        try {
770.            mLocManager.removeUpdates(mLocListener);
771.        } catch (Exception e) {  }
772.        stopForeground(true);
773.        GPSSensor.running = false;
774.        super.onDestroy();
775.    }
776.
777.    public static boolean isRunning() {
778.        return GPSSensor.running;
779.    }
780.    @Override
781.    @MeansEnd(end = "That data based on GPS be acquired",endType =
782.    MeansEndType.GOAL)
783.    protected void onHandleIntent(@Nullable Intent intent) {
784.        while (!mStop) {
785.
786.            mHandler.post(new Runnable() {
787.                @Override
788.                public void run() {
789.                    /* broadcast new context*/
790.                    Intent i = new Intent();
791.                    i.setAction(ContextName.GPS_LOCATION);
792.                    i.putExtra(ContextName.GPS_LOCATION, mLocation);
793.                    i.putExtra(ContextName. GPS_SPEED, mSpeed);
794.                    i.putExtra(ContextName. GPS_AVAILABLE, mGpsAvailable);
795.                    sendBroadcast(i);
796.
797.                }
798.            });
799.
800.            try {
801.                Thread.sleep(120000);
802.            } catch (Exception e) {
803.            }
804.        }
805.    }
806.
807.    @Contribution(type = ContributionType.HELP,softgoal = "User awareness")
808.        private double calculateSpeed(String lastLoc, String curLoc, int duration){
809.        double lat1=0.0;
```

```
810.            double lat2=0.0;
811.            double long1=0.0;
812.            double long2=0.0;
813.            String[] temp=lastLoc.split(",");
814.            if(temp.length==2){
815.               lat1=Double.parseDouble(temp[0]);
816.               long1=Double.parseDouble(temp[1]);
817.            }
818.
819.            temp=curLoc.split(",");
820.            if(temp.length==2){
821.               lat2=Double.parseDouble(temp[0]);
822.               long2=Double.parseDouble(temp[1]);
823.            }
824.
825.            /*transform to radians */
826.            lat1=(lat1*Math.PI)/180.0;
827.            lat2=(lat2*Math.PI)/180.0;
828.            long1=(long1*Math.PI)/180.0;
829.            long2=(long2*Math.PI)/180.0;
830.
831.            double dlat=lat2-lat1;
832.            double dlong=long2-long1;
833.            double a=
834.     Math.sin(dlat/2)*Math.sin(dlat/2)+Math.sin(dlong/2)*Math.sin(dlong/2)*Math.cos(lat1)
835.     *Math.cos(lat2);
836.            double c=2*Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
837.            double d=6371*c;
838.            double result=d/(duration/1000.0);
839.            return result;
840.         }
841.
842.
843.         @Override
844.
845.     @Contribution(type = ContributionType.HELP,softgoal = "Location")
846.         public void onLocationChanged(Location loc) {
847.            mLocation=loc.getLatitude()+","+loc.getLongitude();
848.            /* calculate speed and update current time */
849.            long curTime=System.currentTimeMillis();
850.            if(mLastTime!=0 && mLastLocation!=null){
851.               int duration=(int)(curTime-mLastTime);
852.               mSpeed=calculateSpeed(mLastLocation, mLocation, duration);
853.            } else{
854.               mSpeed=-1;
855.            }
856.            mLastLocation=mLocation;
857.            mLastTime=curTime;
858.         }
859.
860.         @Override
861.     @Contribution(type = ContributionType.HELP,softgoal = "Computer environment")
862.         public void onProviderEnabled(String provider) {
863.            mGpsAvailable = true;
864.            mLastLocation=null;
865.         }
866.
867.         @Override
868.         public void onProviderDisabled(String provider) {
869.            mGpsAvailable = false;
```

```
870.        }
871.        @Override
872.        public void onStatusChanged(String arg0, int arg1, Bundle arg2) {    }
873.    }

874.    @Actor(name = "BluetoothDeviceSensor",type = ActorType.AGENT)
875.    @Goal(name = "That bluetooth device list be acquired",actor =
876.    "BluetoothDeviceSensor",description = "")
877.    @Softgoal(name = "Computing environment",topic = "Bluetooth device",actor =
878.    "BluetoothDeviceSensor")
879.    public class BluetoothDeviceSensor extends IntentService {
880.
881.        private ArrayList<String> mBtDeviceList;
882.        private BluetoothAdapter mBtAdapter;
883.        private Handler mHandler;
884.        private boolean mStop;
885.        private MyBroadcastReceiver mReceiver;
886.        private static boolean running;
887.        private static final String TAG = "BluetoothDeviceLog";
888.
889.        public BluetoothDeviceSensor() {
890.            super("BluetoothDeviceSensor");
891.        }
892.
893.        @Override
894.        public void onCreate() {
895.            mHandler = new Handler();
896.            mBtDeviceList = new ArrayList<String>();
897.            mBtAdapter = BluetoothAdapter.getDefaultAdapter();
898.            if (mBtAdapter == null) {
899.                Toast.makeText(getApplicationContext(), "Bluetooth is not supported on this
900.    device!", Toast.LENGTH_SHORT).show();
901.            } else {
902.                mBtAdapter.enable();
903.            }
904.
905.            mReceiver = new MyBroadcastReceiver();
906.            IntentFilter iFilter = new IntentFilter();
907.            iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
908.            mStop = false;
909.            registerReceiver(mReceiver, iFilter);
910.            if (mBtAdapter != null) {
911.                if (mBtAdapter.isEnabled()) {
912.                    iFilter.addAction(BluetoothDevice.ACTION_FOUND);
913.                    iFilter.addAction(BluetoothAdapter.ACTION_DISCOVERY_STARTED);
914.                    iFilter.addAction(BluetoothAdapter.ACTION_DISCOVERY_FINISHED);
915.                }
916.            }
917.
918.            BluetoothDeviceSensor.running = true;
919.        }
920.
921.        @Override
922.    @MeansEnd(end = "That bluetooth device list be acquired",endType =
923.    MeansEndType.GOAL)
924.        protected void onHandleIntent(Intent intent) {
925.            while (!mStop) {
926.                if (mBtAdapter != null) {
927.                    /* start a new thread to perform Bluetooth device search */
928.                    if (!mBtAdapter.isEnabled()) {
```

```
929.                    mHandler.post(new Runnable() {
930.                        @Override
931.                        public void run() {
932.                            Toast.makeText(getApplicationContext(), "enabling bt adapter.",
933.    Toast.LENGTH_SHORT).show();
934.                        }
935.                    });
936.                    mBtAdapter.enable();
937.                }
938.                mBtAdapter.cancelDiscovery();
939.                mBtAdapter.startDiscovery();
940.            }
941.
942.            mHandler.post(new Runnable() {
943.                @Override
944.                public void run() {
945.                    /* broadcast new context*/
946.                    Intent i = new Intent();
947.                    i.setAction(ContextName.BT_DEVICE_LIST);
948.                    i.putExtra(ContextName.BT_COUNT, mBtDeviceList.size());
949.                    String[] btDeviceList = transformListToArray(mBtDeviceList);
950.                    i.putExtra(ContextName.BT_DEVICE_LIST, btDeviceList);
951.                    sendBroadcast(i);
952.                    Log.i(TAG, ContextName.BT_COUNT +" - " + mBtDeviceList.size());
953.                    Log.i(TAG, ContextName.BT_DEVICE_LIST+" - " + btDeviceList);
954.                }
955.            });
956.        }
957.        try{
958.            Thread.sleep(120000);
959.        } catch(Exception e){
960.            Log.e("PhoneAdapter.error", "Thread sleep exception");
961.        }
962.    }
963.
964.    @Override
965.    public void onDestroy() {
966.        try {
967.            unregisterReceiver(mReceiver);
968.        } catch (Exception e) {
969.            mHandler.post(new Runnable() {
970.                @Override
971.                public void run() {
972.                    Toast.makeText(getApplicationContext(), "Failed to unregister broadcast
973.    receiver!", Toast.LENGTH_SHORT).show();
974.                }
975.            });
976.        }
977.        /**stop foreground service**/
978.        stopForeground(true);
979.
980.        BluetoothDeviceSensor.running = false;
981.
982.        super.onDestroy();
983.    }
984.
985.    private boolean listContainsMac(ArrayList<String> macList, String mac) {
986.        for (int i = 0; i < macList.size(); i++) {
987.            if (macList.get(i).equals(mac)) {
988.                return true;
```

```
989.              }
990.           }
991.           return false;
992.        }
993.
994.        private String[] transformListToArray(ArrayList<String> list) {
995.           String[] s = new String[list.size()];
996.           for (int i = 0; i < list.size(); i++) {
997.              s[i] = list.get(i);
998.           }
999.           return s;
1000.       }
1001.       public static boolean isRunning(){
1002.          return BluetoothDeviceSensor.running;
1003.       }
1004.       private class MyBroadcastReceiver extends BroadcastReceiver {
1005.
1006.          @Override
1007.    @Contribution(type = ContributionType.HELP,softgoal = "Computing environment")
1008.          public void onReceive(Context c, Intent i) {
1009.             String action = i.getAction();
1010.             if (action.equals(BluetoothAdapter.ACTION_DISCOVERY_STARTED)) {
1011.                //mBtDeviceList=new ArrayList<String>();
1012.             } else if (action.equals(BluetoothDevice.ACTION_FOUND)) {
1013.                BluetoothDevice device =
1014.     i.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
1015.                if(!listContainsMac(mBtDeviceList, device.getAddress())){
1016.                         mBtDeviceList.add(device.getAddress());
1017.                }
1018.             } else if (action.equals(BluetoothAdapter.ACTION_DISCOVERY_FINISHED)) {
1019.             } else if (action.equals("edu.hkust.cse.PhoneAdapter.stopService")) {
1020.                mStop = true;
1021.                stopSelf();
1022.             } else {
1023.             }
1024.          }
1025.       }
1026.
1027.    }
```


A1.2. Monitores

```
1028.    @Actor(name = "ContextMonitor",type = ActorType.AGENT)
1029.    @Goal(name = "That monitoring the smartphone context be accomplished",actor =
1030.    "ContextMonitor",description = "")
1031.    public class ContextMonitor extends IntentService  {
1032.       private boolean mGpsAvailable;
1033.       private String mLocation;
1034.       private double mSpeed;
1035.       private String mTime;
1036.       private String mWeekday;
1037.       private Handler mHandler;
1038.       private MyBroadcastReceiver mReceiver;
1039.       private boolean mStop;
1040.       private static final String TAG = "PhoneAdapterContextLog";
1041.       private static boolean running;
1042.       private String[] deviceMacList;
```

```
1043.     private int deviceCount;
1044.
1045.     public ContextMonitor() {
1046.        super("ContextMonitor");
1047.     }
1048.
1049.     @Override
1050.     public void onCreate() {
1051.        super.onCreate();
1052.        mHandler = new Handler();
1053.        mReceiver = new MyBroadcastReceiver();
1054.        IntentFilter iFilter = new IntentFilter();
1055.        iFilter.addAction(ContextName.GPS_LOCATION);
1056.        iFilter.addAction(ContextName.GPS_AVAILABLE);
1057.        iFilter.addAction(ContextName.GPS_SPEED);
1058.        iFilter.addAction(ContextName.TIME);
1059.        iFilter.addAction(ContextName.WEEKDAY);
1060.        iFilter.addAction(ContextName.BT_DEVICE_LIST);
1061.        iFilter.addAction(ContextName.BT_COUNT);
1062.        iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
1063.
1064.        registerReceiver(mReceiver, iFilter);
1065.        mStop = false;
1066.        mLocation = new String();
1067.
1068.        ContextMonitor.running = true;
1069.     }
1070.
1071.     @Override
1072.     public void onDestroy() {
1073.        try {
1074.           unregisterReceiver(mReceiver);
1075.        } catch (Exception e) {
1076.        }
1077.        /**stop foreground service**/
1078.        stopForeground(true);
1079.        ContextMonitor.running = false;
1080.        super.onDestroy();
1081.     }
1082.
1083.     @SuppressLint("LongLogTag")
1084.     @Override
1085.     protected void onHandleIntent(Intent arg0) {
1086.        while (!mStop) {  }
1087.     }
1088.     private void broadcastSystemStateLog() {
1089.        mHandler.post(new Runnable() {
1090.           @Override
1091.           public void run() {
1092.              /* broadcast new context*/
1093.              Intent i = new Intent();
1094.              i.setAction("edu.hkust.cse.PhoneAdapter.newContext");
1095.              i.putExtra(ContextName.GPS_AVAILABLE, mGpsAvailable);
1096.              i.putExtra(ContextName.GPS_LOCATION, mLocation);
1097.              i.putExtra(ContextName.GPS_SPEED, mSpeed);
1098.              i.putExtra(ContextName.BT_DEVICE_LIST, deviceMacList);
1099.              i.putExtra(ContextName.BT_COUNT, deviceCount);
1100.              i.putExtra(ContextName.TIME, mTime);
1101.              i.putExtra(ContextName.WEEKDAY, mWeekday);
1102.              sendBroadcast(i);
```

```
1103.             }
1104.         });
1105.     }
1106.
1107.  ...
1108.     private class MyBroadcastReceiver extends BroadcastReceiver {
1109.
1110.         @Override
1111.     @MeansEnd(end = "That monitoring the smartphone context be accomplished",endType
1112.     = MeansEndType.GOAL)
1113.         public void onReceive(Context c, Intent i) {
1114.             String action = i.getAction();
1115.             switch (action) {
1116.                 case "edu.hkust.cse.PhoneAdapter.stopService": {
1117.                     mStop = true;
1118.                     stopSelf();
1119.                     break;
1120.                 }
1121.                 case ContextName.GPS_LOCATION: {
1122.                     mLocation = i.getStringExtra(ContextName.GPS_LOCATION);
1123.                     mSpeed = i.getDoubleExtra(ContextName.GPS_SPEED,0.0);
1124.                     boolean gpsAvailable=i.getBooleanExtra(ContextName.GPS_AVAILABLE,
false);
1125.                     broadcastSystemStateLog();
1126.                     break;
1127.                 }
1128.                 case ContextName.TIME: {
1129.                     mTime = i.getStringExtra(ContextName.TIME);
1130.                     broadcastSystemStateLog();
1131.                     break;
1132.                 }
1133.                 case ContextName.WEEKDAY: {
1134.                     String newWeekday = i.getStringExtra(ContextName.WEEKDAY);
1135.                     if (mWeekday == null || !newWeekday.equalsIgnoreCase(mWeekday))
1136.                         mWeekday = newWeekday;
1137.                     broadcastSystemStateLog();
1138.                     break;
1139.                 }
1140.                 case ContextName.BT_DEVICE_LIST: {
1141.                     deviceMacList=i.getStringArrayExtra(ContextName.BT_DEVICE_LIST);
1142.                     deviceCount=i.getIntExtra(ContextName.BT_COUNT,0);
1143.                     broadcastSystemStateLog();
1144.                     break;
1145.                 }
1146.                 default: ;
1147.             }
1148.         }
1149.     }
1150.     }

1151.     @Actor(name = "ProfileMonitor",type = ActorType.AGENT)
1152.     @Goal(name = "That monitoring the profile change be accomplished",actor =
1153.     "ProfileMonitor",description = "")
1154.     public class ProfileMonitor extends IntentService {
1155.         private int profileId;
1156.         private static boolean running;
1157.         private MyBroadcastReceiver mReceiver;
1158.         private Handler mHandler;
1159.         private boolean mStop;
1160.         private static final String TAG = "PhoneAdapterProfileLog";
```

```
1161.
1162.       public ProfileMonitor() {
1163.          super("ProfileMonitor");
1164.       }
1165.
1166.
1167.       @Override
1168.       public void onCreate() {
1169.          super.onCreate();
1170.          mHandler = new Handler();
1171.          mReceiver = new MyBroadcastReceiver();
1172.          IntentFilter iFilter = new IntentFilter();
1173.          iFilter.addAction("edu.hkust.cse.PhoneAdapter.profileChange");
1174.          iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
1175.          registerReceiver(mReceiver, iFilter);
1176.          mStop = false;
1177.          ProfileMonitor.running = true;
1178.       }
1179.       public static boolean isRunning() {
1180.          return ProfileMonitor.running;
1181.       }
1182.       @Override
1183.       public void onDestroy() {
1184.          try {
1185.             unregisterReceiver(mReceiver);
1186.          } catch (Exception e) {
1187.          }
1188.          /**stop foreground service**/
1189.          stopForeground(true);
1190.          ProfileMonitor.running = false;
1191.          super.onDestroy();
1192.       }
1193.
1194.       @SuppressLint("LongLogTag")
1195.       @Override
1196.       protected void onHandleIntent(Intent arg0) {
1197.          while (!mStop) {
1198.          }
1199.       }
1200.
1201.       private void broadcastSystemStateLog() {
1202.          mHandler.post(new Runnable() {
1203.             @Override
1204.             public void run() {
1205.                /* broadcast new context*/
1206.                Intent i = new Intent();
1207.                i.setAction("edu.hkust.cse.PhoneAdapter.profileChange");
1208.                i.putExtra("profileId", profileId);
1209.                sendBroadcast(i);
1210.
1211.
1212.             }
1213.          });
1214.       }
1215.
1216.       private class MyBroadcastReceiver extends BroadcastReceiver {
1217.
1218.          @Override
1219.
1220.          public void onReceive(Context context, Intent i) {
```

```
1221.            String action = i.getAction();
1222.            switch (action) {
1223.               case "edu.hkust.cse.PhoneAdapter.stopService": {
1224.                  mStop = true;
1225.                  stopSelf();
1226.                  break;
1227.               }
1228.               case "edu.hkust.cse.PhoneAdapter.profileChange": {
1229.
1230.                  profileId = i.getIntExtra("newProfile", 0);
1231.                  broadcastSystemStateLog();
1232.                  break;
1233.               }
1234.               default: {
1235.               }
1236.            }
1237.         }
1238.      }
}
```

A1.3. Analisador


```
1239.   @Actor(name = "Analyzer",type = ActorType.AGENT)
1240.   @Goal(name = "Analyze profile change needs be accomplished",actor =
1241.   "Analyzer",description = "")
1242.   public class Analyzer extends IntentService {
1243.      private MyDbAdapter mDbHelper;
1244.      private Handler mHandler;
1245.      private MyBroadcastReceiver mReceiver;
1246.      private ArrayList<Rule> mRuleList;
1247.      private Profile mCurProfile;
1248.      private ArrayList<Rule> mCurRuleList;
1249.      private AudioManager mAudioManager;
1250.      private int mMaxVolume;
1251.      private boolean mStop;
1252.      private static final String TAG = "AnalyzerLog";
1253.      private ArrayList<Rule> satisfiedRuleList;
1254.      private static boolean running;
1255.
1256.      public Analyzer() {
1257.         super("Analyzer");
1258.      }
1259.
1260.      @Override
1261.      public void onCreate() {
1262.         super.onCreate();
1263.   ...
1264.         mStop = false;
1265.         Analyzer.running = true;
1266.      }
1267.
1268.      @Override
1269.      protected void onHandleIntent(Intent intent) {
1270.         while (!mStop) {
1271.         }
1272.      }
```

```
1273.
1274.     @Override
1275.     public void onDestroy() {
1276.  ...
1277.         super.onDestroy();
1278.     }
1279.
1280.     public static boolean isRunning() {
1281.         return Analyzer.running;
1282.     }
1283.
1284.     private ArrayList<Rule> fetchRulesFromDb() {
1285.  ArrayList<Rule> list = new ArrayList<Rule>();
1286.         Cursor c = mDbHelper.fetchAllEnabledRules();
1287.         if (c.getCount() > 0) {
1288.             c.moveToFirst();
1289.             while (!c.isAfterLast()) {
1290.                 Rule r = new Rule();
1291.                 Profile curP = new Profile();
1292.                 Profile newP = new Profile();
1293.                 ArrayList<Filter> fList = new ArrayList<Filter>();
1294.                 r.id = c.getInt(c.getColumnIndex(MyDbAdapter.KEY_ROW_ID));
1295.                 r.ruleName =
1296.  c.getString(c.getColumnIndex(MyDbAdapter.KEY_RULE_NAME));
1297.                 r.priority = c.getInt(c.getColumnIndex(MyDbAdapter.KEY_PRIORITY));
1298.
1299.                 /*retrieve profiles, current state might be do not care */
1300.                 Cursor tempC;
1301.                 if (c.getLong(c.getColumnIndex(MyDbAdapter.KEY_CURRENT_STATE_ID))
1302.  == -1) {
1303.                     curP.profileName = "general";
1304.                     /* ring volume should be the percentage of the max, not the absolute value */
1305.                     int volume =
1306.  mAudioManager.getStreamVolume(AudioManager.STREAM_RING);
1307.                     curP.ringVolume = (int) ((volume * 1.0 / mMaxVolume) * 100);
1308.
1309.                     curP.airplaneMode = Settings.System.getInt(getContentResolver(),
1310.  Settings.System.AIRPLANE_MODE_ON, 0);
1311.
1312.                     if (mAudioManager.getRingerMode() ==
1313.  AudioManager.RINGER_MODE_NORMAL || mAudioManager.getRingerMode() ==
1314.  AudioManager.RINGER_MODE_VIBRATE) {
1315.                         curP.vibration = 1;
1316.                     } else {
1317.                         curP.vibration = 0;
1318.                     }
1319.                 } else {
1320.                     tempC =
1321.  mDbHelper.fetchProfile(c.getLong(c.getColumnIndex(MyDbAdapter.KEY_CURRENT_S
1322.  TATE_ID)));
1323.                     if (tempC.getCount() <= 0) {
1324.                         c.moveToNext();
1325.                         continue;
1326.                     } else {
1327.                         tempC.moveToFirst();
1328.                         curP.id =
1329.  tempC.getInt(tempC.getColumnIndex(MyDbAdapter.KEY_ROW_ID));
1330.                         curP.profileName =
1331.  tempC.getString(tempC.getColumnIndex(MyDbAdapter.KEY_PROFILE_NAME));
1332.                         curP.ringVolume =
```

```
1333.  tempC.getInt(tempC.getColumnIndex(MyDbAdapter.KEY_RING_VOLUME));
1334.                  curP.airplaneMode =
1335.  tempC.getInt(tempC.getColumnIndex(MyDbAdapter.KEY_AIRPLANE_MODE));
1336.                  curP.vibration =
1337.  tempC.getInt(tempC.getColumnIndex(MyDbAdapter.KEY_VIBRATION));
1338.                  tempC.close();
1339.                }
1340.              }
1341.
1342.              r.currentProfile = curP;
1343.              tempC =
1344.  mDbHelper.fetchProfile(c.getLong(c.getColumnIndex(MyDbAdapter.KEY_NEW_STATE
1345.  _ID)));
1346.              if (tempC.getCount() <= 0) {
1347.                c.moveToNext();
1348.                continue;
1349.              } else {
1350.                tempC.moveToFirst();
1351.                newP.id =
1352.  tempC.getInt(tempC.getColumnIndex(MyDbAdapter.KEY_ROW_ID));
1353.                newP.profileName =
1354.  tempC.getString(tempC.getColumnIndex(MyDbAdapter.KEY_PROFILE_NAME));
1355.                newP.ringVolume =
1356.  tempC.getInt(tempC.getColumnIndex(MyDbAdapter.KEY_RING_VOLUME));
1357.                newP.airplaneMode =
1358.  tempC.getInt(tempC.getColumnIndex(MyDbAdapter.KEY_AIRPLANE_MODE));
1359.                newP.vibration =
1360.  tempC.getInt(tempC.getColumnIndex(MyDbAdapter.KEY_VIBRATION));
1361.                r.newProfile = newP;
1362.                tempC.close();
1363.              }
1364.
1365.              /* set Filter list */
1366.              tempC =
1367.  mDbHelper.fetchFilterByRuleId(c.getLong(c.getColumnIndex(MyDbAdapter.KEY_ROW
1368.  _ID)));
1369.              if (tempC.getCount() <= 0) {
1370.                c.moveToNext();
1371.                continue;
1372.              } else {
1373.                tempC.moveToFirst();
1374.                while (!tempC.isAfterLast()) {
1375.                  Filter f = new Filter();
1376.                  f.contextType =
1377.  tempC.getInt(tempC.getColumnIndex(MyDbAdapter.KEY_CV_TYPE));
1378.                  f.contextOp =
1379.  tempC.getInt(tempC.getColumnIndex(MyDbAdapter.KEY_CV_OPERATOR));
1380.                  f.contextValue =
1381.  tempC.getString(tempC.getColumnIndex(MyDbAdapter.KEY_VALUE));
1382.                  fList.add(f);
1383.                  tempC.moveToNext();
1384.                }
1385.                tempC.close();
1386.              }
1387.              r.filterList = fList;
1388.              list.add(r);
1389.              c.moveToNext();
1390.            }
1391.            c.close();
1392.          }
```

```
1393.         return list;
1394.     }
1395.
1396.     private class MyBroadcastReceiver extends BroadcastReceiver {
1397.
1398.         @Override
1399.  @MeansEnd(end = "Analyze profile change needs be accomplished",endType =
1400.  MeansEndType.GOAL)
1401.         public void onReceive(Context c, Intent i) {
1402.             String action = i.getAction();
1403.             if (action.equals("edu.hkust.cse.PhoneAdapter.newContext")) {
1404.
1405.                 if (mCurRuleList.size() > 0) {
1406.                     satisfiedRuleList = checkRules(mCurRuleList, i);
1407.                     if (satisfiedRuleList.size() > 0) {
1408.                         mHandler.post(new Runnable() {
1409.                             @Override
1410.                             public void run() {
1411.                                 /* broadcast new context*/
1412.                                 Intent i = new Intent();
1413.                                 i.setAction("edu.hkust.cse.PhoneAdapter.adaptationRequest");
1414.                                 i.putExtra("symptom", "newProfileEnabled");
1415.                                 i.putExtra("satisfiedRuleList",
1416.  transformListToArray(satisfiedRuleList));
1417.                                 sendBroadcast(i);
1418.
1419.                             }
1420.                         });
1421.
1422.                     }
1423.
1424.                 }
1425.
1426.
1427.             } else if (action.equals("edu.hkust.cse.PhoneAdapter.profileChange")) {
1428.                 int profileId = i.getIntExtra("profileId",0);
1429.                 mRuleList = fetchRulesFromDb();
1430.                 mCurRuleList = new ArrayList<Rule>();
1431.                 for (int j = 0; j < mRuleList.size(); j++) {
1432.                     if (mRuleList.get(j).currentProfile.id == profileId) {
1433.                         mCurRuleList.add(mRuleList.get(j));
1434.                     }
1435.                 }
1436.
1437.             } else if (action.equals("edu.hkust.cse.PhoneAdapter.stopService")) {
1438.                 mStop = true;
1439.             } else {
1440.                 //nothing
1441.             }
1442.
1443.         }
1444.
1445.         private int[] transformListToArray(ArrayList<Rule> rules) {
1446.  ...
1447.     }
1448.
1449.
1450.     private static ArrayList<Rule> checkRules(ArrayList<Rule> ruleList, Intent i) {
1451.         ArrayList<Rule> result = new ArrayList<Rule>();
1452.
```

```
1453.         /* get context data from intent */
1454.         boolean gpsAvailable = i.getBooleanExtra(ContextName.GPS_AVAILABLE, false);
1455.         String gpsLocation = i.getStringExtra(ContextName.GPS_LOCATION);
1456.         double gpsSpeed = i.getDoubleExtra(ContextName.GPS_SPEED, 0.0);
1457.         String[] deviceMacList = i.getStringArrayExtra(ContextName.BT_DEVICE_LIST);
1458.         int count = deviceMacList.length;
1459.         String time = i.getStringExtra(ContextName.TIME);
1460.         String weekday = i.getStringExtra(ContextName.WEEKDAY);
1461.
1462.         /* go through each active Rule and evaluate whether its predicate (a conjunction of
1463.    filters) is satisfied */
1464.         for (int j = 0; j < ruleList.size(); j++) {
1465.            ArrayList<Filter> fList = ruleList.get(j).filterList;
1466.            boolean f = true;
1467.            for (int k = 0; k < fList.size(); k++) {
1468.               /* if any Filter is not satisfied,break and set f to false */
1469.               Filter Filter = fList.get(k);
1470.               switch (Filter.contextType) {
1471.                  case ContextType.GPS_ISVALID:
1472.                     int value = Integer.parseInt(Filter.contextValue);
1473.                     boolean bool = (value == 1) ? true : false;
1474.                     if (Filter.contextOp == ContextOperator.EQUAL) {
1475.                        if (gpsAvailable != bool) {
1476.                           f = false;
1477.                           break;
1478.                        }
1479.                     }
1480.                     if (Filter.contextOp == ContextOperator.NOTEQUAL) {
1481.                        if (gpsAvailable == bool) {
1482.                           f = false;
1483.                           break;
1484.                        }
1485.                     }
1486.                     break;
1487.
1488.                  case ContextType.GPS_LOCATION:
1489.                     if (gpsLocation == null || !gpsLocation.contains(",")) {
1490.                        f = false;
1491.                        break;
1492.                     } else {
1493.                        if (Filter.contextOp == ContextOperator.EQUAL) {
1494.                           if (calculateDist(Filter.contextValue, gpsLocation) > 0.05) {
1495.                              f = false;
1496.                              break;
1497.                           }
1498.                        }
1499.                        if (Filter.contextOp == ContextOperator.NOTEQUAL) {
1500.                           if (calculateDist(Filter.contextValue, gpsLocation) <= 0.05) {
1501.                              f = false;
1502.                              break;
1503.                           }
1504.                        }
1505.                     }
1506.                     break;
1507.
1508.                  case ContextType.GPS_SPEED:
1509.                     switch (Filter.contextOp) {
1510.                        case ContextOperator.EQUAL:
1511.                           if (gpsSpeed != Double.parseDouble(Filter.contextValue)) {
1512.                              f = false;
```

```
1513.                     }
1514.                     break;
1515.                  case ContextOperator.GREATER:
1516.                     if (gpsSpeed <= Double.parseDouble(Filter.contextValue)) {
1517.                        f = false;
1518.                     }
1519.                     break;
1520.                  case ContextOperator.GREATER_EQUAL:
1521.                     if (gpsSpeed < Double.parseDouble(Filter.contextValue)) {
1522.                        f = false;
1523.                     }
1524.                     break;
1525.                  case ContextOperator.NOTEQUAL:
1526.                     if (gpsSpeed == Double.parseDouble(Filter.contextValue)) {
1527.                        f = false;
1528.                     }
1529.                     break;
1530.                  case ContextOperator.SMALLER:
1531.                     if (gpsSpeed >= Double.parseDouble(Filter.contextValue)) {
1532.                        f = false;
1533.                     }
1534.                     break;
1535.                  case ContextOperator.SMALLER_EQUAL:
1536.                     if (gpsSpeed > Double.parseDouble(Filter.contextValue)) {
1537.                        f = false;
1538.                     }
1539.                     break;
1540.                  default:
1541.                     break;
1542.
1543.               }
1544.               break;
1545.
1546.            case ContextType.BLUETOOTH:
1547.               if (Filter.contextOp == ContextOperator.EQUAL) {
1548.                  /* if mac list does not contain the context value, f=false */
1549.                  if (!macListContainsMac(deviceMacList, Filter.contextValue)) {
1550.                     f = false;
1551.                     break;
1552.                  }
1553.               }
1554.               if (Filter.contextOp == ContextOperator.NOTEQUAL) {
1555.                  /* if mac list contains the context value, f=false */
1556.                  if (macListContainsMac(deviceMacList, Filter.contextValue)) {
1557.                     f = false;
1558.                     break;
1559.                  }
1560.               }
1561.               break;
1562.
1563.            case ContextType.BLUETOOTH_COUNT:
1564.               switch (Filter.contextOp) {
1565.                  case ContextOperator.EQUAL:
1566.                     if (count != Integer.parseInt(Filter.contextValue)) {
1567.                        f = false;
1568.                     }
1569.                     break;
1570.                  case ContextOperator.GREATER:
1571.                     if (count <= Integer.parseInt(Filter.contextValue)) {
1572.                        f = false;
```

```
1573.                    }
1574.                    break;
1575.                 case ContextOperator.GREATER_EQUAL:
1576.                    if (count < Integer.parseInt(Filter.contextValue)) {
1577.                       f = false;
1578.                    }
1579.                    break;
1580.                 case ContextOperator.NOTEQUAL:
1581.                    if (count == Integer.parseInt(Filter.contextValue)) {
1582.                       f = false;
1583.                    }
1584.                    break;
1585.                 case ContextOperator.SMALLER:
1586.                    if (count >= Integer.parseInt(Filter.contextValue)) {
1587.                       f = false;
1588.                    }
1589.                    break;
1590.                 case ContextOperator.SMALLER_EQUAL:
1591.                    if (count > Integer.parseInt(Filter.contextValue)) {
1592.                       f = false;
1593.                    }
1594.                    break;
1595.              }
1596.              break;
1597.
1598.           case ContextType.TIME:
1599.              int compResult = compareTime(time, Filter.contextValue);
1600.              switch (Filter.contextOp) {
1601.                 case ContextOperator.EQUAL:
1602.                    if (compResult != 0) {
1603.                       f = false;
1604.                    }
1605.                    break;
1606.                 case ContextOperator.GREATER:
1607.                    if (compResult != 1) {
1608.                       f = false;
1609.                    }
1610.                    break;
1611.                 case ContextOperator.GREATER_EQUAL:
1612.                    if (compResult < 0) {
1613.                       f = false;
1614.                    }
1615.                    break;
1616.                 case ContextOperator.NOTEQUAL:
1617.                    if (compResult == 0) {
1618.                       f = false;
1619.                    }
1620.                    break;
1621.                 case ContextOperator.SMALLER:
1622.                    if (compResult != -1) {
1623.                       f = false;
1624.                    }
1625.                    break;
1626.                 case ContextOperator.SMALLER_EQUAL:
1627.                    if (compResult == 1 || compResult == -2) {
1628.                       f = false;
1629.                    }
1630.                    break;
1631.              }
1632.              break;
```

```
1633.
1634.                   case ContextType.WEEKDAY:
1635.                      int comp = compareWeekday(weekday, Filter.contextValue);
1636.                      switch (Filter.contextOp) {
1637.                         case ContextOperator.EQUAL:
1638.                            if (comp != 0) {
1639.                               f = false;
1640.                            }
1641.                            break;
1642.                         case ContextOperator.GREATER:
1643.                            if (comp != 1) {
1644.                               f = false;
1645.                            }
1646.                            break;
1647.                         case ContextOperator.GREATER_EQUAL:
1648.                            if (comp < 0) {
1649.                               f = false;
1650.                            }
1651.                            break;
1652.                         case ContextOperator.NOTEQUAL:
1653.                            if (comp == 0) {
1654.                               f = false;
1655.                            }
1656.                            break;
1657.                         case ContextOperator.SMALLER:
1658.                            if (comp != -1) {
1659.                               f = false;
1660.                            }
1661.                            break;
1662.                         case ContextOperator.SMALLER_EQUAL:
1663.                            if (comp == 1 || comp == -2) {
1664.                               f = false;
1665.                            }
1666.                            break;
1667.                         default:
1668.                            break;
1669.                      }
1670.                      break;
1671.
1672.                   default:
1673.                      break;
1674.                }
1675.             }
1676.             if (ruleList.get(j).newProfile.id != mCurProfile.id) {
1677.                result.add(ruleList.get(j));
1678.             }else{
1679.                Log.i(TAG,"Profile is already updated.");
1680.             }       }
1681.          return result;
1682.       }
1683.       public static double calculateDist(String lastLoc, String curLoc) {
1684.       ...
1685.       }
1686.       public static boolean macListContainsMac(String[] list, String mac) {
1687.    ...
1688.       }
1689.       private static int compareTime(String t1, String t2) {
1690.          ...
1691.       }
1692.       private static int compareWeekday(String wd1, String wd2) {
```

```
...
    }}
```

A1.4. Planejador

```
@Actor(name = "Planner",type = ActorType.AGENT)
@Goal(name = "Strategy for profile change be accomplished",actor =
"Planner",description = "")
public class Planner extends IntentService {
    private double volumeLevel;
    private int vibratiom;
    private int airplaneMode;
    private MyBroadcastReceiver mReceiver;
    private static final String TAG = "PlannerLog";
    private MyDbAdapter mDbHelper;
    private AudioManager mAudioManager;
    private Handler mHandler;
    private int mMaxVolume;
    private Profile mCurProfile;
    private ArrayList<Rule> mCurRuleList;
    private ArrayList<Rule> mRuleList;
    private ArrayList<Rule> satisfiedRuleList;
    private boolean mStop;
    private static boolean running;

    public Planner() {
        super("Planner");
    }

    @Override
    public void onCreate() {
        super.onCreate();
        ...
        mReceiver = new MyBroadcastReceiver();
        IntentFilter iFilter = new
 IntentFilter("edu.hkust.cse.PhoneAdapter.adaptationRequest");
        iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
        registerReceiver(mReceiver, iFilter);
        ...
        mStop = false;
        Planner.running = true;
    }

    @Override
    public void onDestroy() {
      ...
        super.onDestroy();
    }

    public static boolean isRunning() {
        return Planner.running;
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        while (!mStop) {
        }
    }
```

```
1748.     private ArrayList<Integer> transformArrayToList(int[] rules) {
1749.         ArrayList<Integer> ruleList = new ArrayList<Integer>();
1750.         for (int i = 0; i < rules.length; i++) {
1751.             ruleList.add(new Integer(rules[i]));
1752.         }
1753.         return ruleList;
1754.     }
1755.
1756.     private ArrayList<Rule> fetchRulesFromDb(int[] ruleIdList) {
1757.         ...
1758.         return list;
1759.     }
1760.
1761.     private class MyBroadcastReceiver extends BroadcastReceiver {
1762.
1763.         @Override
1764. @MeansEnd(end = "Strategy for profile change be accomplished",endType =
1765. MeansEndType.GOAL)
1766.         public void onReceive(Context context, Intent i) {
1767.             String action = i.getAction();
1768.             if (action.equals("edu.hkust.cse.PhoneAdapter.adaptationRequest")) {
1769.                 while (true) {
1770.                     String symptom = i.getStringExtra("symptom");
1771.                     createStrategy(symptom, i);
1772.
1773.                 }
1774.             } else if (action.equals("edu.hkust.cse.PhoneAdapter.stopService")) {
1775.                 mStop = true;
1776.             }
1777.         }
1778.
1779.     }
1780.
1781.     private void createStrategy(String symptom, Intent i) {
1782.         switch (symptom) {
1783.             case "newProfileEnabled": {
1784.                 int[] ruleList = i.getIntArrayExtra("satisfiedRuleList");
1785.                 satisfiedRuleList = fetchRulesFromDb(ruleList);
1786.                 if (satisfiedRuleList.size() == 1) {
1787.
1788.                     broadcastStrategy(satisfiedRuleList.get(0));
1789.
1790.                 } else {
1791.                     //pick up the Rule with highest priority
1792.                     ArrayList<Rule> candidate = new ArrayList<Rule>();
1793.                     int min = Integer.MAX_VALUE;
1794.                     for (int j = 0; j < satisfiedRuleList.size(); j++) {
1795.                         if (satisfiedRuleList.get(j).priority < min) {
1796.                             min = satisfiedRuleList.get(j).priority;
1797.                         }
1798.                     }
1799.                     for (int j = 0; j < satisfiedRuleList.size(); j++) {
1800.                         if (satisfiedRuleList.get(j).priority == min) {
1801.                             candidate.add(satisfiedRuleList.get(j));
1802.                         }
1803.                     }
1804.                     // size cannot be zero, at least one
1805.                     if (candidate.size() == 1) {
1806.                         broadcastStrategy(candidate.get(0));
1807.
```

```
1808.                 } else {
1809.                     //randomly pick one
1810.                     Random rand = new Random();
1811.                     int choice = rand.nextInt(candidate.size());
1812.                     //perform adaptation and update current profile and current Rule list
1813.                     broadcastStrategy(candidate.get(choice));
1814.
1815.                 }
1816.             }
1817.             break;
1818.         }
1819.         default: {
1820.             break;
1821.         }
1822.     }
1823. }
1824.
1825. private void broadcastStrategy(Rule satisfiedRule) {
1826.     volumeLevel = (satisfiedRule.newProfile.ringVolume) * 1.0 / 100;
1827.     airplaneMode = satisfiedRule.newProfile.airplaneMode;
1828.     vibratiom = satisfiedRule.newProfile.vibration;
1829.
1830.     mHandler.post(new Runnable() {
1831.         @Override
1832.         public void run() {
1833.             /* broadcast new strategy*/
1834.             Intent i = new Intent();
1835.             i.setAction("edu.hkust.cse.PhoneAdapter.strategy");
1836.             i.putExtra("vibratiom", vibratiom == 1 ? true : false);
1837.             i.putExtra("airplaneMode", airplaneMode == 1 ? true : false);
1838.             i.putExtra("ringVolume", volumeLevel);
1839.             i.putExtra("newProfile", satisfiedRuleList.get(0).newProfile.id);
1840.             sendBroadcast(i);
1841.         }
1842.     });
1843.
1844. }
1845. }
```


A1.5. Executor

```
1846. @Actor(name = "Executor",type = ActorType.AGENT)
1847. @Goal(name = "The profile change strategy be carried out",actor =
1848. "Executor",description = "")
1849. public class Executor extends IntentService {
1850. ...
1851.     public Executor() {
1852.         super("Executor");
1853.     }
1854.
1855.     @Override
1856.     public void onCreate() {
1857.         super.onCreate();
1858.
1859.         mHandler = new Handler();
1860.
1861.         mReceiver = new MyBroadcastReceiver();
```

```
1862.         IntentFilter iFilter = new IntentFilter("edu.hkust.cse.PhoneAdapter.strategy");
1863.         iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
1864.         registerReceiver(mReceiver, iFilter);
1865.
1866.         mStop = false;
1867.         Executor.running = true;
1868.      }
1869.
1870.      @Override
1871.      public void onDestroy() {
1872.     ...
1873.         super.onDestroy();
1874.      }
1875.
1876.      public static boolean isRunning() {
1877.         return Executor.running;
1878.      }
1879.
1880.      @Override
1881.      protected void onHandleIntent(Intent intent) {
1882.         while (!mStop) {
1883.         }
1884.      }
1885.
1886.      private class MyBroadcastReceiver extends BroadcastReceiver {
1887.         @Override
1888.   @MeansEnd(end = "The profile change strategy be carried out",endType =
1889.   MeansEndType.GOAL)
1890.         public void onReceive(Context context, Intent i) {
1891.            String action = i.getAction();
1892.            switch (action) {
1893.               case "edu.hkust.cse.PhoneAdapter.stopService": {
1894.                  mStop = true;
1895.                  stopSelf();
1896.                  break;
1897.               }
1898.               case "edu.hkust.cse.PhoneAdapter.strategy": {
1899.                  executeStrategy(i);
1900.                  break;
1901.               }
1902.               default: {
1903.               }
1904.            }
1905.         }
1906.      }
1907.
1908.      private void executeStrategy(Intent strategy) {
1909.         vibratiom = strategy.getBooleanExtra("vibration",false);
1910.         mHandler.post(new Runnable() {
1911.            @Override
1912.            public void run() {
1913.               /* broadcast new context*/
1914.               Intent i = new Intent();
1915.               i.setAction("edu.hkust.cse.PhoneAdapter.vibration");
1916.               i.putExtra("vibratiom", vibratiom);
1917.               sendBroadcast(i);
1918.            }
1919.         });
1920.         airplaneMode = strategy.getBooleanExtra("airplaneMode",false);
1921.         mHandler.post(new Runnable() {
```

```
1922.            @Override
1923.            public void run() {
1924.               /* broadcast new context*/
1925.               Intent i = new Intent();
1926.               i.setAction("edu.hkust.cse.PhoneAdapter.airplaneMode");
1927.               i.putExtra("airplaneMode", airplaneMode);
1928.               sendBroadcast(i);
1929.            }
1930.         });
1931.         volumeLevel = strategy.getDoubleExtra("ringVolume",0.0);
1932.         mHandler.post(new Runnable() {
1933.            @Override
1934.            public void run() {
1935.               /* broadcast new context*/
1936.               Intent i = new Intent();
1937.               i.setAction("edu.hkust.cse.PhoneAdapter.volumeLevel");
1938.               i.putExtra("volumeLevel", volumeLevel);
1939.               sendBroadcast(i);
1940.            }
1941.         });
1942.         newProfile = strategy.getIntExtra("newProfile",0);
1943.         mHandler.post(new Runnable() {
1944.            @Override
1945.            public void run() {
1946.               /* broadcast new context*/
1947.               Intent i = new Intent();
1948.               i.setAction("edu.hkust.cse.PhoneAdapter.profileChange");
1949.               i.putExtra("newProfile", newProfile);
1950.               sendBroadcast(i);
1951.            }
1952.         });
1953.      }
1954.   }
```

A1.6. Atuadores

```
1955.   @Actor(name = "RingVolumeEffector",type = ActorType.AGENT)
1956.   @Goal(name = "That the ring volume be adjusted",actor =
1957.   "RingVolumeEffector",description = "")
1958.   public class RingVolumeEffector extends IntentService {
1959.      ...
1960.
1961.      public RingVolumeEffector() {
1962.         super("RingVolumeEffector");
1963.      }
1964.
1965.
1966.      @Override
1967.      public void onCreate() {
1968.         super.onCreate();
1969.
1970.         mHandler = new Handler();
1971.
1972.         mReceiver = new MyBroadcastReceiver();
1973.         IntentFilter iFilter = new IntentFilter("edu.hkust.cse.PhoneAdapter.volumeLevel");
1974.         iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
1975.         registerReceiver(mReceiver, iFilter);
1976.         mAudioManager = (AudioManager)
1977.   this.getSystemService(Context.AUDIO_SERVICE);
```

```
1978.          mMaxVolume =
1979.    mAudioManager.getStreamMaxVolume(AudioManager.STREAM_RING);
1980.          mStop = false;
1981.          RingVolumeEffector.running = true;
1982.       }
1983.
1984.       @Override
1985.       public void onDestroy() {
1986.          ...
1987.        super.onDestroy();
1988.       }
1989.
1990.       public static boolean isRunning() {
1991.          return RingVolumeEffector.running;
1992.       }
1993.
1994.       @Override
1995.       protected void onHandleIntent(Intent intent) {
1996.          while (!mStop) {
1997.          }
1998.       }
1999.
2000.       private class MyBroadcastReceiver extends BroadcastReceiver {
2001.          @Override
2002.          public void onReceive(Context context, Intent i) {
2003.             String action = i.getAction();
2004.             switch (action) {
2005.                case "edu.hkust.cse.PhoneAdapter.stopService": {
2006.                   mStop = true;
2007.                   stopSelf();
2008.                   break;
2009.                }
2010.                case "edu.hkust.cse.PhoneAdapter.volumeLevel": {
2011.                   double ringVolume = i.getDoubleExtra("ringVolume",0.0);
2012.                   int volume = (int) (ringVolume * mMaxVolume);
2013.                   if (volume > 0) {
2014.                      ringVolumeOn(volume);
2015.                   } else {
2016.                      ringVolumeOff();
2017.                   }
2018.                   break;
2019.                }
2020.                default: {
2021.                }
2022.             }
2023.          }
2024.       }
2025.
2026.    @MeansEnd(end = "That the ring volume be adjusted",endType =
2027.    MeansEndType.GOAL)
2028.       private void ringVolumeOff() {
2029.          mAudioManager.setStreamVolume(AudioManager.STREAM_RING, 0,
2030.     AudioManager.FLAG_SHOW_UI);
2031.       }
2032.
2033.    @MeansEnd(end = "That the ring volume be adjusted",endType =
2034.    MeansEndType.GOAL)
2035.       private void ringVolumeOn(int volume) {
2036.          mAudioManager.setRingerMode(AudioManager.RINGER_MODE_NORMAL);
2037.          mAudioManager.setStreamVolume(AudioManager.STREAM_RING, volume,
```

```
2038.    AudioManager.FLAG_SHOW_UI);
2039.        }
2040.
2041.    }

2042.    @Actor(name = "VibrationEffector",type = ActorType.AGENT)
2043.    @Goal(name = "That the vibration mode be adjusted",actor =
2044.    "VibrationEffector",description = "")
2045.    public class VibrationEffector extends IntentService {
2046.        private AudioManager mAudioManager;
2047.        private MyBroadcastReceiver mReceiver;
2048.        private static final String TAG = "VibrationEffectorLog";
2049.        private Handler mHandler;
2050.        private boolean mStop;
2051.        private static boolean running;
2052.
2053.        public VibrationEffector() {
2054.            super("VibrationEffector");
2055.        }
2056.
2057.
2058.        @Override
2059.        public void onCreate() {
2060.            super.onCreate();
2061.
2062.            mHandler = new Handler();
2063.
2064.            mReceiver = new MyBroadcastReceiver();
2065.            IntentFilter iFilter = new IntentFilter("edu.hkust.cse.PhoneAdapter.volumeLevel");
2066.            iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
2067.            registerReceiver(mReceiver, iFilter);
2068.            mAudioManager = (AudioManager)
2069.    this.getSystemService(Context.AUDIO_SERVICE);
2070.            mStop = false;
2071.            VibrationEffector.running = true;
2072.        }
2073.
2074.        @Override
2075.        public void onDestroy() {
2076.            /* unregister broadcast receiver and close database */
2077.            try {
2078.                unregisterReceiver(mReceiver);
2079.            } catch (Exception e) {
2080.            }
2081.
2082.            /**stop foreground service**/
2083.            stopForeground(true);
2084.            VibrationEffector.running = false;
2085.            super.onDestroy();
2086.        }
2087.
2088.        public static boolean isRunning() {
2089.            return VibrationEffector.running;
2090.        }
2091.
2092.        @Override
2093.        protected void onHandleIntent(Intent intent) {
2094.            while (!mStop) {
2095.            }
2096.        }
```

```
2097.
2098.     private class MyBroadcastReceiver extends BroadcastReceiver {
2099.         @Override
2100.         public void onReceive(Context context, Intent i) {
2101.             String action = i.getAction();
2102.             switch (action) {
2103.                 case "edu.hkust.cse.PhoneAdapter.stopService": {
2104.                     mStop = true;
2105.                     stopSelf();
2106.                     break;
2107.                 }
2108.                 case "edu.hkust.cse.PhoneAdapter.vibration": {
2109.                     boolean vibration = i.getBooleanExtra("vibration",false);
2110.                     if (vibration) {
2111.                         vibrationOn();
2112.                     } else {
2113.                         vibrationOff();
2114.                     }
2115.                     break;
2116.                 }
2117.                 default: {
2118.                 }
2119.             }
2120.         }
2121.     }
2122.
2123.     @MeansEnd(end = "That the vibration mode be adjusted",endType =
2124.     MeansEndType.GOAL)
2125.         private void vibrationOn() {
2126.             mAudioManager.setVibrateSetting(AudioManager.VIBRATE_TYPE_RINGER,
2127.      AudioManager.VIBRATE_SETTING_ON);
2128.         }
2129.
2130.     @MeansEnd(end = "That the vibration mode be adjusted",endType =
2131.     MeansEndType.GOAL)
2132.         private void vibrationOff() {
2133.             mAudioManager.setVibrateSetting(AudioManager.VIBRATE_TYPE_RINGER,
2134.     AudioManager.VIBRATE_SETTING_OFF);
2135.         }
2136.     }

2137.     @Actor(name = "AirplaneModeEffector",type = ActorType.AGENT)
2138.     @Goal(name = "That the airplane mode be adjusted",actor =
2139.     "AirplaneModeEffector",description = "")
2140.     public class AirplaneModeEffector extends IntentService {
2141.
2142.         private MyBroadcastReceiver mReceiver;
2143.         private static final String TAG = "AirplaneModeEffectorLog";
2144.         private Handler mHandler;
2145.         private boolean mStop;
2146.         private static boolean running;
2147.
2148.         public AirplaneModeEffector() {
2149.             super("AirplaneModeEffector");
2150.         }
2151.
2152.
2153.         @Override
2154.         public void onCreate() {
2155.             super.onCreate();
```

```
2156.
2157.         mHandler = new Handler();
2158.
2159.         mReceiver = new MyBroadcastReceiver();
2160.         IntentFilter iFilter = new IntentFilter("edu.hkust.cse.PhoneAdapter.volumeLevel");
2161.         iFilter.addAction("edu.hkust.cse.PhoneAdapter.stopService");
2162.         registerReceiver(mReceiver, iFilter);
2163.
2164.         mStop = false;
2165.         AirplaneModeEffector.running = true;
2166.       }
2167.
2168.       @Override
2169.       public void onDestroy() {
2170.         ...
2171.         super.onDestroy();
2172.       }
2173.
2174.       public static boolean isRunning() {
2175.         return AirplaneModeEffector.running;
2176.       }
2177.
2178.       @Override
2179.       protected void onHandleIntent(Intent intent) {
2180.         while (!mStop) {
2181.         }
2182.       }
2183.
2184.       private class MyBroadcastReceiver extends BroadcastReceiver {
2185.         @Override
2186.         public void onReceive(Context context, Intent i) {
2187.           String action = i.getAction();
2188.           switch (action) {
2189.             case "edu.hkust.cse.PhoneAdapter.stopService": {
2190.               mStop = true;
2191.               stopSelf();
2192.               break;
2193.             }
2194.             case "edu.hkust.cse.PhoneAdapter.airplaneMode": {
2195.               boolean airplaneMode = i.getBooleanExtra("airplaneMode",false);
2196.               if (airplaneMode) {
2197.                 airplaneModeOn();
2198.               } else {
2199.                 airplaneModeOff();
2200.               }
2201.               break;
2202.             }
2203.             default: {
2204.             }
2205.           }
2206.         }
2207.       }
2208.
2209.     @MeansEnd(end = "That the airplane mode be adjusted",endType =
2210.     MeansEndType.GOAL)
2211.       private void airplaneModeOn() {
2212.         if (Settings.System.getInt(getContentResolver(),
2213.     Settings.System.AIRPLANE_MODE_ON, 0) == 0) {
2214.           Settings.System.putInt(getContentResolver(),
2215.     Settings.System.AIRPLANE_MODE_ON, 1);
```

```
2216.            Intent intent = new Intent(Intent.ACTION_AIRPLANE_MODE_CHANGED);
2217.            intent.putExtra("state", true);
2218.            sendBroadcast(intent);
2219.        }
2220.    }
2221.
2222.    @MeansEnd(end = "That the airplane mode be adjusted",endType =
2223.    MeansEndType.GOAL)
2224.    private void airplaneModeOff() {
2225.        if (Settings.System.getInt(getContentResolver(),
2226.    Settings.System.AIRPLANE_MODE_ON, 0) == 1) {
2227.            Settings.System.putInt(getContentResolver(),
2228.    Settings.System.AIRPLANE_MODE_ON, 0);
2229.            Intent intent = new Intent(Intent.ACTION_AIRPLANE_MODE_CHANGED);
2230.            intent.putExtra("state", false);
2231.            sendBroadcast(intent);
2232.        }
2233.    }
2234.
2235.    }
```