UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SMELLS ARQUITETURAIS DE MONITORAMENTO EM SISTEMAS ADAPTATIVOS

MARCEL AKIRA SERIKAWA

ORIENTADOR: PROF. DR. VALTER VIEIRA DE CAMARGO

São Carlos - SP
Agosto/2016

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SMELLS ARQUITETURAIS DE MONITORAMENTO EM SISTEMAS ADAPTATIVOS

MARCEL AKIRA SERIKAWA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software
Orientador: Prof. Dr. Valter Vieira de Camargo

São Carlos - SP
Agosto/2016

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária UFSCar
Processamento Técnico
com os dados fornecidos pelo(a) autor(a)

Serikawa, Marcel Akira
S485s Smells arquiteturais de monitoramento em sistemas adaptativos / Marcel Akira Serikawa. -- São Carlos : UFSCar, 2016.
97 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2016.

1. Smell Arquitetural . 2. Sistemas Auto-Adaptativos . 3. Monitoramento. I. Título.

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia

Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a defesa de Dissertação de Mestrado do candidato Marcel Akira Serikawa, realizada em 19/08/2016.

Prof. Dr. Valter Vieira de Camargo
(UFSCar)

Prof. Dr. Fabiano Cutigi Ferrari
(UFSCar)

****

Prof. Dr. Edson de Oliveira Júnior
(UEM)

Certifico que a sessão de defesa foi realizada com a participação à distância do membro Prof. Dr. Edson de Oliveira Júnior . Depois das arguições e deliberações realizadas, o participante à distância está de acordo com o conteúdo do parecer da comissão examinadora redigido no relatório de defesa do aluno Marcel Akira Serikawa.

Prof. Dr. Valter Vieira de Camargo
Coordenadora da Comissão Examinadora
(UFSCar)

A todos que me apoiaram.

AGRADECIMENTO

Agradeço à minha família. Ao meu pai e minha mãe que sempre me deram o suporte necessário. As minhas irmãs por me mostrarem o valor de se compartilhar. A Cris, minha companheira nas maiores e melhores aventuras de minha vida. A Sofia por simplesmente se importar. Ao Gabriel e Marco por me ensinarem os mais diversos assuntos.

Agradeço aos novos amigos desta jornada. Ao professor Valter que me orientou nessa difícil construção de conhecimento, os amigos de laboratório, André, Bruno M., Bruno G., Daniel, Durelli e Renato. Aos meus amigos que conheci nos corredores do departamento, em especial ao Thiago, Régis e Matheus.

Agradeço aos velhos amigos. Mateus, Teruo, Renatinha e Hatano que assim como eu resolveram encarar o curso de computação e torceram pelo meu sucesso. Ao Digão e o Johnny pela companhia nas viagens que tivemos e teremos.

Um grande obrigado a todos.

Se vi mais longe foi por estar apoiado sobre ombros de gigantes.

Isaac Newton

RESUMO

Sistemas Adaptativos são capazes de se adaptarem às mudanças de seu ambiente de execução ou das necessidades do usuário. Monitores são essenciais para o desenvolvimento desse tipo de sistema, pois, são responsáveis por coletar e pré-processar dados do contexto. Em um levantamento de sistemas realizado em repositórios e na literatura especializada, notou-se que monitores são por vezes projetados e implementados de uma forma inadequada, exibindo as seguintes características: i) ficam obscuros no código-fonte; ii) possuem uma taxa de monitoramento única e iii) são forçados a ter uma ordem de execução pré-determinada. Isso leva a dificuldades na manutenção, evolução e muitas vezes problemas relacionados a desempenho. Decisões de projetos que levam a essas dificuldades podem ser caracterizadas como *smells* arquiteturais. A documentação de *smells* auxilia desenvolvedores a identificar oportunidades de refatorações do sistema, bem como evidenciar práticas que devem ser avaliadas durante o projeto e desenvolvimento de novos sistemas. Portanto, nesta dissertação de mestrado são propostos dois *smells* arquiteturais: o *Obscure Monitor* e o *Oppressed Monitors*. O primeiro ocorre quando os monitores não estão evidentes no sistema e o segundo ocorre quando os monitores estão sujeitos a uma mesma taxa de monitoramento e também possuem uma ordem de execução rígida. Como avaliação preliminar foi realizado um estudo exploratório em um sistema adaptativo chamado *PhoneAdapter*. Para verificar a influência da presença desses *smells* foram realizadas cinco atividades de manutenção em duas versões desse sistema, a original com os *smells* e uma refatorada na qual os *smells* foram removidos. Os resultados obtidos indicam que a manutenção e evolução de tais sistemas são facilitadas na maioria dessas atividades.

Palavras-chave: Smell Arquitetural, Sistemas Auto-Adaptativos, Monitoramento.

ABSTRACT

Adaptive systems are able to adapt themselves according to changes in its execution environment or the user's needs. Monitors are essential for the development of such systems because they are responsible for collecting and pre-processing the context data. By a search in systems found in repositories and literature, it was observed that monitors are sometimes designed and implemented in an inappropriate way, exhibiting the following characteristics: i) they are Obscure in the source code; ii) they have a unique monitoring rate and iii) they are forced to have a pre-determined execution order. These characteristics lead to difficulties in maintenance, evolution and often problems related to performance. Design decisions that lead to these difficulties can be characterized as architectural smells. The documentation of smells helps developers identifying refactoring opportunities of a system and also highlights practices that should be analyzed during the design and development of new systems. Therefore, this master thesis proses two architectural smells for adaptive systems: the Obscure Monitor and Oppressed Monitors. The first occurs when the monitors are not evident in the system and the second occurs when the monitors are subject to the same monitoring rate and have a strict execution order. In order to check the influence of the presence of theses smells five maintenance activities were applied in two versions of a system called *PhoneAdapter*, the original version with the presence of the smells and the refactored one in which the smells were removed. The results indicate that the maintenance and evolution of the refactored system are facilitated in most activities.

Keywords: Architectural Smell, Self-Adaptive System, Monitoring.

LISTA DE FIGURAS

Figura 2.1 - Sistema de controle com realimentação adaptado de Ogata *et al.* (2003) 22

Figura 2.2 - Atividades realizadas por um *loop* de controle adaptado de Brun *et al.* (2009).....23

Figura 2.3 - Modelo conceitual MAPE-K proposto por Kephart e Chess (2003).....24

Figura 2.4 - Ciclo de Monitoramento (fonte: elaborada pelo autor) 28

Figura 2.5 - Smell Arquitetural Ciclo entre Classes *(Lippert e Roock 2006)*.....30

Figura 2.6 - Padrão de projeto Subject/Observer (Gamma 1995).....31

Figura 2.7 - Catálogo de *smells* arquiteturais adaptado de Lippert e Roock (2006)..32

Figura 3.1 - Diagrama de classes da aplicação PhoneAdapter (fonte: elaborada pelo autor).....36

Figura 3.2 - Classe ContextManager do PhoneAdapter (fonte: elaborada pelo autor) 38

Figura 4.1 - Diagrama de Classes do sistema AIAS (fonte: elaborada pelo autor) ...52

Figura 4.2 - Esquema de monitoramento do sistema AIAS.....52

Figura 4.3 - Trechos da Classe ScriptBot do sistema AIASProject (fonte: elaborada pelo autor) 53

Figura 4.4 - Diagrama de classes do sistema Dc4citites (fonte: elaborada pelo autor) 56

Figura 4.5 - Execução do monitoramento do sistema Dc4cities (fonte: elaborada pelo autor).....57

Figura 4.6 - Trecho da Classe OptimizationManager do sistema dc4cities (fonte: elaborada pelo autor) 58

Figura 4.7 - Diagrama de classes do sistema Self-protect-project (fonte: elaborada pelo autor) 60

Figura 4.8 - Execução do monitoramento do sistema Self-protect-project (fonte: elaborada pelo autor) 60

Figura 4.9 - Trecho da Casse MainActivity do sisema Self-protect (fonte: elaborada pelo autor) 61

Figura 4.10 - Diagrama de classes do sistema Novi (fonte: elaborada pelo autor) ...63

Figura 4.11 - Execução do monitoramento do sistema Novi (fonte: elaborada pelo autor).....64

Figura 4.12 - Trecho da Classe SensorService do sistema Novi (fonte: elaborada pelo autor) 65
Figura 4.13 - Arquitetura do sistema SmarTrAC (Fan *et al.* 2015) 67
Figura 4.14 - Modelo do objeto sentinela adaptado de Biegel e Cahill (2004) 68
Figura 4.15 - Arquitetura Behavior-based programming adaptado de (Arkin 1998) 69
Figura 5.1 - Diagrama de classes do PhoneAdapter a) versão original, b) versão refatorada (fonte: elaborada pelo autor) 72
Figura 5.2 - Template da classe Monitor (fonte: elaborada pelo autor) 73
Figura 5.3 - Monitor de dia da semana (fonte: elaborada pelo autor) 75
Figura 5.4 - Trechos relacionados ao monitor de bateria no PhoneAdapter (fonte: elaborada pelo autor) 78
Figura 5.5 - Adição de novo monitor no sistema original (fonte: elaborada pelo autor) 80
Figura 5.6 - Classe BatteryMonitor (fonte: elaborada pelo autor) 81
Figura 5.7 - Código-fonte relacionado ao pré-processamento do dia da semana 81
Figura 5.8 - Definição de diferentes taxas de monitoramento (fonte: elaborada pelo autor) 83
Figura 5.9 - Alteração da taxa de monitoramento no sistema refatorado (fonte: elaborada pelo autor) 84
Figura 5.10 - Alterações na classe BatteryMonitor (fonte: elaborada pelo autor) 86
Figura 5.11 - Alterações na classe GPSMonitor (fonte: elaborada pelo autor) 86
Figura 6.1 - Padrão de projeto *master/slave (Weyns et al. 2013)* 90
Figura 6.2 - Padrão SAS Monitor (Abuseta e Swesi, 2015) 91

LISTA DE TABELAS

Tabela 2.1 - Exemplo de sensores físicos disponíveis 27
Tabela 3.1 - Trecho de código e o monitor correspondente 37
Tabela 4.1 - Trabalhos selecionados a partir de artigos 50
Tabela 4.2 - Sistemas selecionados para verificação dos *smells* 50
Tabela 4.3 - Evidência da presença dos *smells* no sistema AIASProject 54
Tabela 4.4 - Evidência da presença dos *smells* em Dc4cities 58
Tabela 4.5 - Evidência da presença de *smells* no sistema Self-protect 62
Tabela 4.6 - Evidência da presença dos *smells* no sistema Novi 66
Tabela 5.1 - Tarefas de manutenção 76

SUMÁRIO

CAPÍTULO 1 - INTRODUÇÃO....13

1.1 Contexto....13
1.2 Motivação....15
1.3 Objetivos....17
1.4 Organização do Trabalho....18

CAPÍTULO 2 - FUNDAMENTAÇÃO TEÓRICA....19

2.1 Considerações Iniciais....19
2.2 Sistemas Adaptativos....20
2.2.1 *Loop* de Controle....21
2.2.2 Interesse de Monitoramento....26
2.3 *Smells* Arquiteturais....29
2.4 Considerações Finais....32

CAPÍTULO 3 - *SMELLS* ARQUITETURAIS DE MONITORAMENTO....34

3.1 Considerações Iniciais....34
3.2 Exemplo Motivacional PhoneAdapter....35
3.3 Smell Obscure Monitor....40
3.3.1 Identificação do Smell....41
3.3.2 Impactos na Qualidade....42
3.4 Smell Opressed Monitors....43
3.4.1 Identificação do Smell....44
3.4.2 Impactos na Qualidade....45
3.5 Considerações Finais....46

CAPÍTULO 4 - EVIDÊNCIA DA EXISTÊNCIA DOS *SMELLS* PROPOSTOS....47

4.1 Considerações Iniciais....47
4.2 Metodologia de Levantamento....48

4.3 Projetos Selecionados de Repositórios 51
4.3.1 AIASProject 51
4.3.2 Zanshin 54
4.3.3 Dc4cities 55
4.3.4 Self-protect-project 59
4.4 Projetos Selecionados de Artigos 62
4.4.1 Novi 62
4.4.2 SmarTrAC 65
4.4.3 Context-aware Framework 67
4.4.4 Behavior-Based Program 68
4.5 Considerações Finais 69

CAPÍTULO 5 - AVALIAÇÃO PRELIMINAR DA MANUTENABILIDADE EM SISTEMAS ADAPTATIVOS 70

5.1 Considerações Iniciais 70
5.2 Refatoração do PhoneAdapter 71
5.3 Tarefas de Manutenção 75
5.3.1 Adicionar novo monitor 77
5.3.2 Alterar a lógica de pré-processamento 79
5.3.3 Definir diferentes taxas de monitoramento 82
5.3.4 Definir ordem de execução entre monitores 85
5.4 Considerações Finais 87

CAPÍTULO 6 - TRABALHOS RELACIONADOS 88

6.1 Considerações Iniciais 88
6.2 Trabalhos com Padrões de Projetos 89
6.3 Trabalhos com *Smells* Arquiteturais 92
6.4 Considerações Finais 93

CAPÍTULO 7 - CONCLUSÃO 94

7.1 Considerações iniciais 94
7.2 Contribuições 95
7.3 Limitações 95
7.4 Trabalhos Futuros 96
7.5 Publicações 97

REFERÊNCIAS..98

Capítulo 1

INTRODUÇÃO

1.1 Contexto

Sistemas Adaptativos (SAs) são capazes de modificar sua estrutura ou comportamento em tempo de execução a partir de dados coletados do contexto em que estão inseridos. O objetivo é melhorar a qualidade do serviço oferecido e/ou oferecer ao usuário uma experiência mais agradável do uso do sistema (Oppermann 1994, McKinley *et al.* 2004, Hussein *et al.* 2010).

O desenvolvimento de Sistemas Adaptativos teve grande influência da área de Engenharia de Controle (Brun *et al.* 2009). Um conceito muito utilizado nessa área é o de *loop* de controle (*feedback control loop)*, que é responsável por monitorar os valores de saída de um sistema a partir de sensores, para em seguida avaliar se os valores estão de acordo com os esperados. Caso não estejam, o *loop* de controle realiza ajustes nos parâmetros de forma que o sistema produza resultados cada vez mais próximos aos valores de saída esperados (Ogata *et al.* 2003). Assim, o *loop* de controle se caracteriza como um ciclo de adaptação que pode ser dividido em quatro atividades, em que cada uma delas possui um elemento abstrato responsável pela sua execução. Esses elementos e suas respectivas atividades são as seguintes (IBM 2005, Brun *et al.* 2009):

1) Monitores - os monitores são responsáveis pela atividade ou interesse de monitoramento, envolvendo a coleta de dados a partir de sensores;

2) Analisadores - os analisadores são responsáveis pela atividade de análise que compara os valores coletados dos sensores com valores de referência;
3) Planejadores - os planejadores são responsáveis pela atividade de planejamento que verifica a viabilidade em realizar as adaptações;
4) Executores - os executores são responsáveis por aplicar, no sistema gerenciado, o plano de adaptação gerado pelos planejadores.

Projeto de sistemas adaptativos é uma linha de pesquisa que estuda formas alternativas de organização da arquitetura desse tipo de sistema. O objetivo é fazer com que a manutenção e a evolução sejam mais gerenciáveis e produtivas, bem como prover melhorias no desempenho. Muitos autores concordam que *loop*s de controle são abstrações fundamentais nesse tipo de sistema e que devem ficar evidentes no código-fonte (Kephart e Chess 2003, IBM 2005, Brun *et al.* 2009, Weyns *et al.* 2013). Seguindo essa linha, um modelo conceitual bastante difundido é o MAPE-K proposto pela IBM (Horn 2001, IBM 2005). O objetivo desse modelo é estruturar sistemas adaptativos de forma que os elementos do *loop* de controle (Monitores, Analisadores, Planejadores e Executores) fiquem evidentes no código-fonte (Brun *et al.* 2009).

Dentre os elementos/abstrações dos *loop*s de controle, o monitor é o foco de pesquisa deste trabalho. Monitores são responsáveis por realizar três principais tarefas (Gil de La Iglesia 2014, Abuseta e Swesi 2015): i) observar e coletar dados provenientes de sensores; ii) realizar o pré-processamento desses dados e iii) enviar as informações processadas para os demais elementos do *loop* de controle. Deste ponto em diante, essas tarefas são denominadas de lógica de monitoramento, lógica de pré-processamento e lógica de envio de dados.

A partir do levantamento e análise do interesse de monitoramento dos sistemas adaptativos, observou-se que os monitores são compostos por uma parte estática e uma comportamental. A estática é o código-fonte que representa suas tarefas (lógica de monitoramento, pré-processamento e envio de dados). Dependendo do projeto, essas lógicas podem estar modularizadas em uma única classe ou espalhadas e entrelaçadas em diversas outras classes.

A parte comportamental é representada pelo código-fonte que i) determina a taxa de monitoramento e ii) especifica a independência de execução dos monitores. A taxa de monitoramento é o intervalo de tempo com que os dados são coletados.

Por exemplo, um monitor de movimento coleta os dados de um sensor com uma taxa de monitoramento de um minuto, ou seja, a cada intervalo de um minuto os dados são coletados. Quanto à independência da execução dos monitores, um monitor é considerado independente se sua execução não depende de dados de outros monitores. Ou seja, em nenhum trecho de seu código-fonte é necessária a utilização de dados provenientes de outro monitor.

Em outra linha de pesquisa, *bad smells* são práticas de programação inadequadas que podem levar a problemas de manutenção, reúso e desempenho (Fowler 2009). O fato de um *bad smell* estar documentado e publicado na literatura auxilia mantenedores a encontrar mais facilmente locais do código-fonte que podem gerar problemas ao se realizarem manutenções e evoluções.

Os *smells* mais conhecidos são aqueles propostos por Fowler (2009) que são focados em elementos de código-fonte, como por exemplo, métodos e classes. Os *smells* de Fowler também são independentes de domínio, isto é, eles podem estar presentes em sistemas adaptativos, sistemas multi-agentes, sistemas embarcados, sistemas de informação dentre outros. Quando os *smells* buscam evidenciar indícios de problemas em níveis mais altos de abstração, são conhecidos como *Smells* Arquiteturais. S*mells* arquiteturais são práticas de programação inadequadas que afetam a arquitetura de um sistema e que também podem impactar as atividades de manutenção e evolução (Stal 2007, Garcia *et al.* 2009). A diferença é que ocorrem em abstrações mais altas que o código-fonte, como camadas, componentes, pacotes dentre outros.

Portanto, a presente dissertação de mestrado investiga a existência de decisões no projeto do interesse de monitoramento que possam impactar negativamente nas evoluções e manutenções no interesse de monitoramento de sistemas adaptativos, podendo assim, serem caracterizadas como *smells* arquiteturais no interesse de monitoramento.

1.2 Motivação

O fator motivador deste trabalho são evidências de que existem formas inadequadas de projeto e implementação de monitores que podem levar a

problemas de manutenção, evolução e reúso. Assim, julga-se ser relevante caracterizá-las e documentá-las como *smells* arquiteturais de monitores em SAs.

As evidências mencionadas foram obtidas com base em uma análise realizada em oito (8) sistemas adaptativos nos quais se observou que os monitores são, muitas vezes, projetados e implementados de forma a apresentarem as seguintes características: i) não estão evidentes no código-fonte; ii) são forçados a possuir uma mesma taxa de monitoramento e iii) possuem uma ordem de execução pré-determinada.

A primeira característica está relacionada à modularização inadequada dos monitores, fazendo com que seu código-fonte seja obscuro. Isto é, o código-fonte relacionado ao elemento monitor que envolve a lógica de monitoramento, pré-processamento e envio dos dados está implementado de forma entrelaçada com o código de outros monitores ou outros elementos. Como consequência a compreensibilidade, reusabilidade e manutenibilidade relacionadas ao elemento monitor são prejudicadas. Muitos trabalhos já publicados propõem que ausência de modularização resulta em problemas de manutenção (Parnas 1972, Sanchez e Mahoney 1996, Kiczales *et al.* 2001).

A segunda característica está relacionada aos monitores possuírem a mesma taxa de monitoramento. Esta característica pode resultar nos seguintes problemas:

- Consumo inadequado de recursos. Caso a taxa de monitoramento esteja adequada ao monitor cujos dados se alteram com maior frequência, os monitores os quais os dados se alteram em uma frequência menor serão executados desnecessariamente;
- Perda de dados de contexto. Caso a taxa de monitoramento esteja adequada para o monitor cujos dados se alterem com pouca frequência, os monitores com dados que alteram em uma frequência maior deixarão de coletar diversos dados;
- Taxas de monitoramento individuais. Evoluções no sistema para adaptar as taxas de monitoramento de cada monitor individualmente são dificultadas.

A terceira característica está relacionada aos monitores possuírem uma ordem predeterminada e imutável em tempo de execução. Com base nas análises realizadas, observou-se que essa característica pode resultar nos seguintes problemas:

- Inativação de monitores. Esse problema pode ocorrer quando monitores anteriores não liberam o controle para monitores subsequentes;
- Geração de contextos errôneos. Esse problema pode ocorrer quando a coleta dos dados por um determinado monitor é feita em um momento tardio, pois monitores anteriores mantiveram o controle por muito tempo.

Como esses problemas não estão documentados na literatura, muitos desenvolvedores podem continuar a projetar o interesse de monitoramento com as mesmas características mencionadas. Dessa forma, observou-se a oportunidade de documentar esses problemas na forma de *smells* arquiteturais. Assim, o principal ponto de argumentação deste trabalho reside sobre duas hipóteses:

1- Monitores mal modularizados e obscuros no código-fonte prejudicam a análise do sistema e também atividades de manutenção que envolvam tais elementos. Isso é sustentado por meio da observação de trabalhos que defendem separação de interesses e modularização (Parnas 1972, Sanchez e Mahoney 1996, Kiczales *et al.* 2001);

2- Monitores que não expõem uma forma de controlar suas taxas de monitoramento e execução independente levam a dificuldades de manutenção e impossibilidades de adaptações que lidam com essas taxas.

1.3 Objetivos

Esta dissertação possui quatro objetivos que podem ser categorizados em específicos e gerais. Os objetivos específicos estão relacionados com o que é descrito e proposto nesta dissertação e os objetivos gerais estão relacionados ao impacto que se espera que os *smells* propostos tenham no desenvolvimento e manutenção de sistemas adaptativos.

Os objetivos específicos são:

- Mostrar evidências de que existem formas inadequadas de implementação de monitores em sistemas adaptativos, sendo possível documentá-la na forma de *smells* arquiteturais;

- Mostrar evidências de que a presença desses *smells* leva a dificuldades de manutenção;

Os objetivos gerais são:

- Facilitar a identificação dos *smells* em sistemas existentes para que o sistema possa ser refatorado;
- Atentar engenheiros de software para o projeto do interesse de monitoramento em novos sistemas;

1.4 Organização do Trabalho

Esta dissertação está organizada em sete capítulos. No primeiro capítulo estão apresentados o contexto, a motivação e os objetivos do trabalho.

No Capítulo 2 são apresentados dois principais conceitos abordados neste trabalho. O primeiro conceito abordado são os Sistemas Adaptativos, enfatizando a composição do elemento *loop* de controle e principalmente o interesse de monitoramento. O segundo conceito abordado são os *Smells* Arquiteturais.

No Capítulo 3 são descritos os dois *smells* arquiteturais relacionados ao interesse de monitoramento em sistemas adaptativos: o *Obscure Monitor* e o *Oppressed Monitors*.

No Capítulo 4 são analisados sistemas adaptativos que evidenciam a presença dos *smells* arquiteturais propostos nesta dissertação.

No Capítulo 5 é realizada uma avaliação preliminar da manutenabilidade em duas versões do sistema adaptativo PhoneAdapter: a versão original com os *smells* e a versão refatorada sem os *smells*.

No Capítulo 6 são mostrados os principais trabalhos relacionados que foram encontrados na literatura, assim como uma discussão das semelhanças e diferenças de cada um em relação a esta dissertação.

Por fim, no Capítulo 7 são apresentadas as principais contribuições do trabalho, suas limitações e propostas de trabalhos futuros.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

Os objetivos deste capítulo são apresentar os principais conceitos relacionados ao desenvolvimento deste trabalho, sendo esses: Sistemas Adaptativos e *Smells* Arquiteturais. Na seção de Sistemas Adaptativos objetiva-se fornecer embasamento teórico para o melhor entendimento do interesse de monitoramento. Para isso são apresentadas as principais características e propriedades dos Sistemas Adaptativos, bem como o elemento *loop* de controle. Na seção de *Smells* Arquiteturais objetiva-se caracterizar e exemplificar tais *smells*.

O capítulo está organizado em quatro seções. Na Seção 2.1 são realizadas as considerações iniciais. Na Seção 2.2 são abordados os principais conceitos relacionados ao desenvolvimento de Sistemas Adaptativos, sendo ela dividida em duas subseções. Na Subseção 2.2.1 são apresentados os principais conceitos do *Loop* de Controle e na Subseção 2.2.2 são descritos em maiores detalhes o Interesse de Monitoramento. Na Seção 2.3 é apresentado o conceito de Smells Arquiteturais e a sua importância no processo de manutenção e evolução de sistemas. Por fim, na Seção 2.4 são mostradas as considerações finais do capítulo.

2.2 Sistemas Adaptativos

Diversas áreas estão interessadas no desenvolvimento e pesquisas de Sistemas Adaptativos, dentre essas podem-se citar a Computação Autonômica, Sistemas Embarcados, Redes de Computadores, Robôs Autônomos, Sistemas Multi-agentes, Inteligência Artificial, Computação Ubíqua, Linha de produto de software dentre outras (De Lemos *et al.* 2013). Por ser um tema estudado por diversas áreas o termo sistema adaptativo é um conceito muito amplo e com diversas definições.

Neste sentido é importante diferenciar os termos sistemas adaptáveis e sistemas adaptativos. Oppermann (1994) define que um sistema é adaptável se ele provê ao usuário ferramentas que possibilitem realizar alterações em algumas características do sistema. Portanto, o usuário está no controle da adaptação. Por outro lado, um sistema será adaptativo se for capaz de alterar suas próprias características de acordo com a necessidade do contexto ou usuário. Essa auto-adaptação é a principal característica de sistemas adaptativos.

Sob outra perspectiva, McKinley *et al.* (2004) sugerem que os sistemas adaptativos podem ser divididos em duas categorias: Adaptação por parâmetros e Adaptação composicional. Na primeira categoria, o sistema é responsável por adaptar variáveis do programa determinando assim um novo comportamento, como por exemplo, um robô móvel pode aumentar sua velocidade de acordo com o tipo de terreno. Enquanto que na segunda categoria é possível que sejam adicionadas novas estruturas ao sistema em tempo de execução. Por exemplo, um novo algoritmo pode ser implementado e adicionado ao sistema. É importante observar que as adaptações em ambos os programas ocorrem em tempo de execução.

No contexto deste trabalho o termo Sistema Adaptativo refere-se a sistemas capazes de realizar alterações em seu comportamento por meio de ajustes em seus parâmetros. Esses ajustes ocorrem de acordo com o contexto em que o sistema é executado, sem a necessidade da intervenção do usuário. Para isso, Sistemas Adaptativos devem realizar o monitoramento constante de dados do ambiente ou do próprio sistema e a partir desses dados coletados realizar adaptações em seu comportamento sem a necessidade de intervenção do usuário.

Na perspectiva da Engenharia de Software, Sistemas Adaptativos têm sido explorados por diversos aspectos, como a Engenharia de Requisitos, Desenvolvimento de Modelos e Arquiteturas, Programação orientada a Eventos e Componentes, Teste, dentre outros. Apesar de vários anos de pesquisa sob esses aspectos, o desenvolvimento de Sistemas Adaptativos ainda é uma tarefa desafiadora. Alguns destes desafios são (De Lemos *et al.* 2013):

- **Lidar com conflitos de objetivos em diferentes *loop*s de controle.** Por exemplo, a adaptação gerada por um *loop* é conflitante com a adaptação gerada por outro *loop*, levando a disputa pela prioridade sobre determinado recurso;
- **Monitorar diversos interesses com precisão.** O monitoramento é uma atividade que envolve uma troca entre precisão e desempenho. Dependendo da quantidade de dados monitorados e da precisão desejada, pode ser gerado um gasto indesejado de processamento, comprometendo a qualidade de serviço. Isto é, o gasto na utilização do processamento para realizar o monitoramento reduz recursos para a execução do sistema, prejudicando sua qualidade de serviço;
- **Reutilizar os *loop*s de controle.** Os *loop*s são desenvolvidos de maneira *adhoc* para o sistema, exigindo pouca ou nenhuma modularização dos interesses. Levando, assim, a uma dificuldade em reutilizar esses elementos em outros sistemas.

Com base nesses desafios, diversos pesquisadores (Garlan *et al.* 2004, IBM 2005, Brun *et al.* 2009) descrevem os benefícios de especificar o *loop* de controle e seus principais elementos de forma explícita no código-fonte. Além de afirmarem que definir as interações entre os elementos do *loop* explicitamente da fase de projeto até a implementação possibilita a melhora dos atributos de qualidade e o gerenciamento da complexidade.

2.2.1 *Loop* de Controle

Uma característica comum a todos os sistemas adaptativos é a capacidade de tomar decisões que irão alterar o seu comportamento em tempo de execução. Essas decisões se baseiam nos dados coletados do ambiente e do próprio sistema

(Brun *et al.* 2009). Por exemplo, no caso de um robô móvel autônomo, o robô deve constantemente verificar a presença de obstáculos em sua trajetória. Caso encontre algum obstáculo ele deverá adaptar seu comportamento com o objetivo de desviar do obstáculo. Uma situação similar pode ocorrer em um serviço de internet, no qual um servidor fica constantemente verificando o tráfego de dados em sua rede e em casos de tráfego intenso, requisita que um novo servidor seja ativado para evitar a lentidão nos dados transmitidos.

Estes sistemas possuem a capacidade de adaptar-se a diferentes informações do contexto que estão sendo executados. O elemento responsável por realizar este processo de adaptação é conhecido como *loop* de controle. O conceito de *loop* de controle na computação teve muita influência da Engenharia de Controle, em especial dos sistemas de controle com realimentação conhecidos também como *feedback control loop*.

Na Figura 2.1 é mostrada uma representação típica de um sistema de controle com realimentação. Esse sistema contém três elementos principais representados por retângulos: o Sistema, Sensor e Controlador. O Sistema é responsável por realizar os processos relacionados ao requisito do domínio, envolvendo os processos para transformar o valor de entrada no valor de saída. O Sensor é responsável por coletar informações do valor de saída gerando o valor medido. Esse valor medido é então comparado a um valor de referência tendo como saída um erro, processo representado pelo círculo preenchido na figura. Por fim, o Controlador recebe como entrada o erro e realiza ajustes no Sistema alterando seus parâmetros para que sejam produzidos melhores valores de saída conforme as necessidades (Ogata *et al.* 2003).

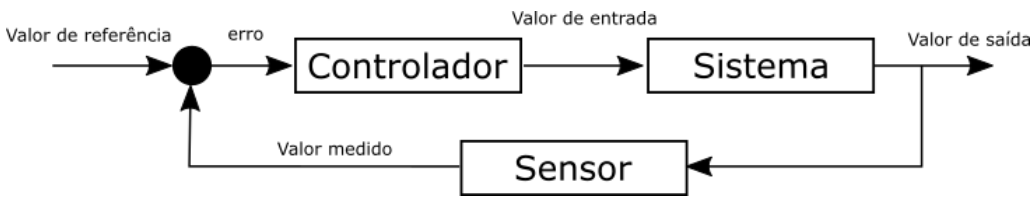


Figura 2.1 - Sistema de controle com realimentação adaptado de Ogata *et al.* (2003)

Um exemplo de um sistema de controle com realimentação é o sistema de ar-condicionado no qual o valor de entrada refere-se ao ar do ambiente que entra no sistema e o valor de saída é o ar resfriado que volta para o ambiente. No caso do ar-

condicionado, um sensor de temperatura verifica a temperatura do ar resfriado pelo sistema. A temperatura é então comparada com o valor ajustado, dessa comparação pode haver três possíveis resultados: a temperatura do ar está acima do esperado, abaixo do esperado ou está igual aos valores esperados. A partir deste resultado, o controlador do ar-condicionado define os novos parâmetros para o sistema a fim de gerar um valor de saída correspondente aos desejados.

Conforme pode ser observado na Figura 2.2, o processo executado por um *loop* de controle pode ser dividido em quatro atividades principais (Brun *et al.* 2009): coletar, analisar, decidir e agir. A atividade Coletar é responsável por coletar os dados por meio de sensores e em seguida enviá-los para a atividade Analisar. Na atividade Analisar os dados são processados, esse processamento pode envolver a execução de diversos algoritmos de análise. Com os resultados da análise dos dados, a atividade Decidir verifica a necessidade de realizar adaptações. Por fim, caso adaptações sejam necessárias, essas serão executadas pela atividade Agir, na qual são efetivadas as adaptações necessárias.

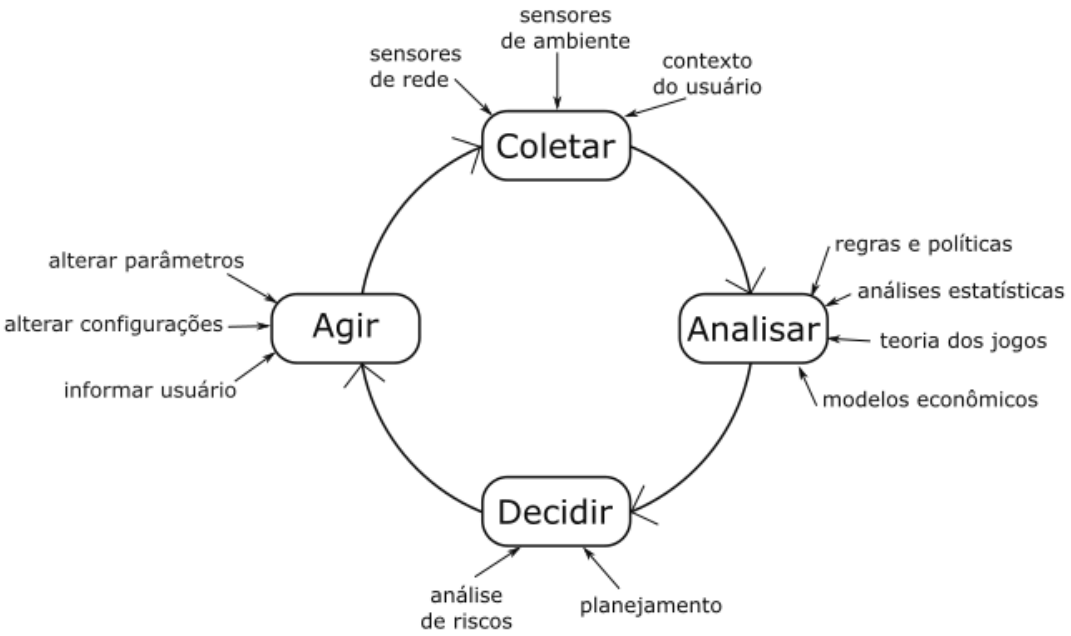


Figura 2.2 - Atividades realizadas por um *loop* de controle adaptado de Brun *et al.* (2009)

Inspirada no sistema de controle com realimentação e das atividades realizadas pelo *loop* de controle, a IBM (IBM 2005) propôs um modelo conceitual de *loop* de controle para o desenvolvimento de sistemas autonômicos, conhecido como

MAPE-K representado na Figura 2.3. A principal proposta desse modelo é evidenciar o *loop* de controle e seus elementos, proporcionando melhor gerenciamento no desenvolvimento de sistemas adaptativos.

A Figura 2.3 é dividida em duas partes, subsistema gerenciado e subsistema gerenciador. Na parte inferior está representado o subsistema gerenciado, nesse subsistema estão localizados os sensores que são responsáveis por informar dados do contexto em que o sistema é executado e os atuadores que são responsáveis por efetivar as adaptações. Na parte superior é representado o subsistema gerenciador que segue o modelo conceitual MAPE-K. Como é mostrada na Figura 2.3 parte superior, o subsistema gerenciador possui quatro elementos básicos *Monitor*, *Analisador*, *Planejador* e *Executor (MAPE)* e uma base de *Conhecimento (K)*. Esses elementos serão detalhados em seguida.

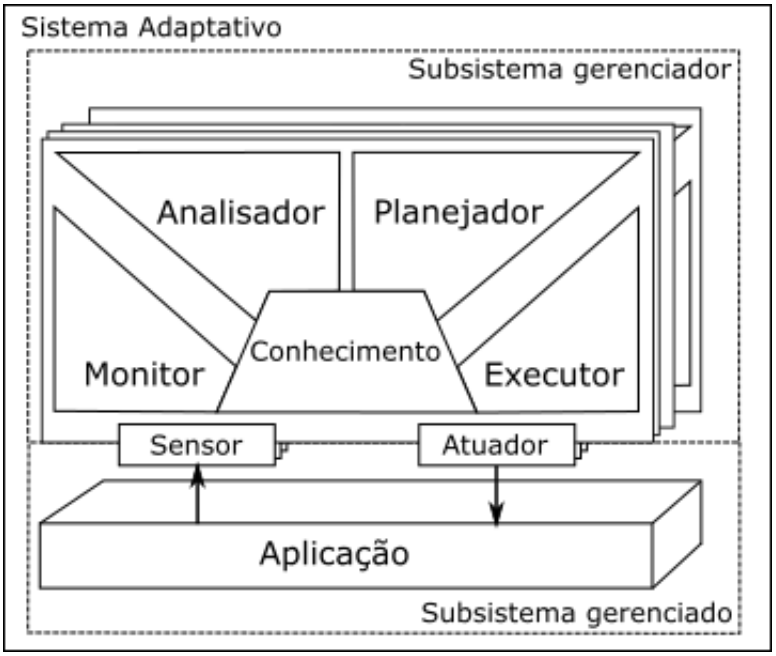


Figura 2.3 - Modelo conceitual MAPE-K proposto por Kephart e Chess (2003)

Sensores e os Atuadores são chamados de *touch points*, pois conectam o subsistema gerenciador com o subsistema gerenciado. Sensores são responsáveis por fornecer dados ao subsistema gerenciador. Esses dados podem ser referentes ao subsistema gerenciado ou do ambiente de execução. Atuadores são responsáveis por efetivar as adaptações no subsistema gerenciado.

Monitores são responsáveis por coletar as informações dos Sensores, realizar o pré-processamento dos dados e enviá-los aos demais elementos do *loop* de controle interessados. Em um monitor, a coleta dos dados deve ser realizada de maneira constante em paralelo a execução do sistema. O pré-processamento é realizado quando os dados coletados pelos monitores necessitam de modificações para poderem ser utilizados pelos demais elementos do *loop* de controle. Como, por exemplo, conversões, padronização, agregação de valores, aplicação de filtros, dentre outros.

Analisadores são responsáveis por processar os dados enviados pelos monitores e determinar se adaptações na aplicação são necessárias. Esse elemento analisa os dados recebidos comparando-os com um valor de referência previamente determinado, caso após análise seja identificada a necessidade de adaptações, o elemento é responsável por iniciar o processo de adaptação no *loop* de controle.

Planejadores são responsáveis em planejar o processo necessário para adaptar o estado atual do sistema para um estado no qual os requisitos sejam satisfeitos. O planejador pode realizar três ações: (i) manter o sistema como está, pois já satisfaz as condições esperadas; (ii) adicionar novos recursos caso o sistema não satisfaça os objetivos; (iii) remover recursos sobressalentes. Definido o plano de adaptação, este é enviado para o executor.

Executores são responsáveis por realizar as ações definidas pelo Planejador. Essas ações podem envolver um ou mais comandos de adição ou remoção de recursos. Dessa forma é responsabilidade do executor verificar se os recursos já estão disponíveis para adição/remoção.

A base de Conhecimentos é a representação abstrata de toda informação necessária para o bom funcionamento do *loop* de controle. Isto pode envolver dados do subsistema gerenciado, contexto, objetivos e recursos utilizados ou disponíveis. Com esse conhecimento é possível que as adaptações sejam executadas de maneira adequada.

Nesta dissertação o elemento Monitor é o foco dos estudos. Isto se deve ao fato de ele ser o primeiro elemento necessário para tornar um sistema adaptativo, além de ser o elemento que inicia o processo de adaptação realizado pelo *loop* de controle (Lalanda *et al.* 2013).

2.2.2 Interesse de Monitoramento

Os sensores são um elemento intrínseco na composição de um monitor, ou seja, para existir um monitor é necessário que exista pelo menos um sensor definido. Contudo o contrário nem sempre é verdadeiro, isto é, a presença de um sensor não indica necessariamente a existência de um monitor.

Os sensores são definidos como os elementos responsáveis por fornecer os dados do sistema e do ambiente de execução (Brun *et al.* 2009) podendo ser divididos em três grupos (Baldauf *et al.* 2007):

- **Sensores físicos.** Os sensores físicos são os utilizados com maior frequência, eles representam os diversos hardwares disponíveis atualmente para coletar as informações do ambiente. A Tabela 2.1 mostra alguns exemplos desses sensores.
- **Sensores virtuais.** Os sensores virtuais proveem informações dos sistemas de software ou serviços, ou seja, não existe componente de hardware dedicado para coleta dos dados. Por exemplo, um sensor que verifica o tempo de execução de um método em um sistema, o valor de banda utilizado em uma rede de computadores, dentre outros.
- **Sensores lógicos.** Os sensores lógicos são a composição de vários outros sensores para prover dados em maior nível de abstração. Por exemplo, um sensor para detectar a localização de um funcionário. Essa coleta pode ser por meio de um cartão RFID, dados de seu calendário, dados do GPS de seu celular, dentre outros. Gerando assim dados precisos sobre a localização do funcionário.

Na Tabela 2.1 são mostrados alguns exemplos de possíveis sensores físicos. Esta tabela é dividida em duas colunas, na primeira coluna contém o contexto em que os sensores podem estar inseridos e na segunda coluna alguns exemplos de sensores existentes capazes de prover dados de tais contextos.

Tabela 2.1 - Exemplo de sensores físicos disponíveis

Tipo de Contexto	Exemplo de sensores disponíveis
Luminosidade	Fotodiodos, sensores de cor, Infravermelho e Ultravioleta.
Imagens	Câmeras fotográficas e de vídeo.
Áudio	Microfones.
Movimento	Acelerômetros, sensores de movimento, sensores de campo magnético.
Localização	GPS (Global Positioning System) e GSM (Global System for Mobile Comunications)
Atributos físicos	Barômetros e termômetros.

Segundo Gil de La Iglesia (2014), o relacionamento entre sensores e monitores pode ser realizado de duas formas: Ativo - reativo ou Reativo - ativo. No primeiro caso, o sensor cria um evento e fornece os dados para o monitor, isto é, quando um sensor capta alguma alteração no contexto ele informa esta alteração ao monitor. Por outro lado, na segunda forma, o monitor é o responsável por coletar os dados dos sensores. Isto é, de tempos em tempos o monitor deve iniciar o processo de verificação. No contexto deste trabalho observou-se principalmente o relacionamento Reativo - ativo.

Esta diferenciação é importante, pois algumas plataformas como o *Android*[1]*e Lejos* fornecem componentes de código-fonte do tipo *Listener*, que são componentes responsáveis por "observar" os sensores de hardware. Em muitos casos esses *Listener* são referidos como sensores capazes de gerar um evento quando existe alguma alteração no sensor. Contudo, a partir da definição de monitores os *Listeners* fazem o papel de um monitor de hardware.

O monitor é o elemento responsável por verificar ativamente os dados dos sensores, ou seja, é sua responsabilidade definir quando e como realizar a coleta dos dados, o pré-processamento dos dados, se necessário, e enviá-los para os demais elementos do *loop* de controle.

Na Figura 2.4 está representada a execução de um clico de monitoramento, isto é, o processo executado pelo monitor para a coleta, pré-processamento e o envio dos dados. O ciclo se inicia pela etapa de aquisição dos dados fornecidos pelo

[1] https://developer.android.com/google/index.html

sensor, chamado de lógica de monitoramento. Esta lógica refere-se ao processo de instanciar o hardware do sensor e atribuir os valores coletados a uma variável do monitor. Em seguida, esta variável contendo os dados do sensor pode então ser filtrada, padronizada ou acrescida de valores, processo denominado de lógica de pré-processamento. Por fim, os dados coletados são enviados aos demais elementos do *loop* de controle.

Nesta etapa, o envio dos dados aos demais elementos pode ser desenvolvido de duas maneiras: ativado por tempo ou por evento. O monitor por tempo transmite seus dados constantemente conforme um período de tempo determinado, enquanto que o monitor por evento envia seus dados quando um evento é disparado a partir dos dados coletados (Gil de La Iglesia 2014).

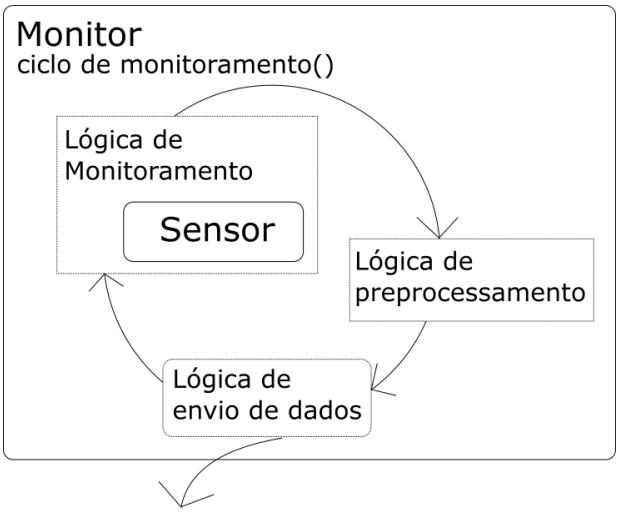


Figura 2.4 - Ciclo de Monitoramento (fonte: elaborada pelo autor)

Por exemplo, em um monitor ativado por tempo as informações geradas são enviadas frequentemente para os demais elementos com base em um período de tempo determinado. Por outro lado, em um monitor ativado por evento, o monitor realiza a leitura do sensor, verifica se esta leitura é significativa e caso seja um evento é gerado para o envio dos dados aos demais elementos do *loop* de controle.

A utilização de cada tipo de monitor depende exclusivamente do tipo de dado coletado e do sistema envolvido. Isto é, em alguns sistemas talvez seja necessário que o *loop* de controle seja disparado mesmo com a leitura de dados repetidos, pois esses dados podem ser armazenados para gerarem estatísticas ou participarem nos algoritmos de aprendizado. Por outro lado, em outros sistemas não é interessante

que leituras em sequência repetidas disparem todo o processo do *loop* de controle, pois dados repetidos podem representar um mesmo contexto o que não geraria novas adaptações ao sistema. Logo, a execução do processo do *loop* de controle seria um gasto desnecessário de recursos.

Como se pode ver o desenvolvimento do elemento monitor possui diversas particularidades, podendo ser realizado de diversas maneiras. Mas, apesar de alguns autores proporem padrões para o desenvolvimento destes elementos (Ramirez e Cheng 2010, Weyns *et al.* 2013, Abuseta e Swesi 2015), observou-se que eles não foram utilizados em nenhum dos sistemas analisados. A partir da análise, pode se notar também que em muitos casos eles são projetados de maneira que podem levar a dificuldades de manutenção e evolução.

2.3 *Smells* Arquiteturais

A Arquitetura de Software surgiu como uma parte crucial ao desenvolvimento de sistemas. O desenvolvimento pode ser dividido em duas partes: uma que consiste no entendimento do problema e outra na construção da solução. A arquitetura de software está intrinsicamente relacionada com o entendimento do problema, pois por meio da arquitetura, podem-se evidenciar as regras de negócio e projetar os requisitos do sistema. Nessa primeira parte, o sistema é definido por meio de abstrações sem a necessidade dos detalhes de implementação tendo assim um maior foco nas interações e no comportamento dos componentes (de Andrade *et al.* 2014).

Apesar do termo arquitetura de software ser utilizado há anos não existe ainda um consenso a respeito de sua definição (Kruchten *et al.* 2006). Uma das definições mais difundidas do termo arquitetura de software refere-se à estrutura do sistema consistindo: nos elementos de software, propriedades externamente visíveis e o relacionamento entre estes elementos (Bass *et al.* 2003). Outra definição proposta pelo SEI (*Software Engineering Institute*), sugere que arquitetura de software seja uma representação do sistema que auxilia no entendimento de como ele deve se comportar. Outros autores vão além e afirmam que a arquitetura de

software também envolve as principais decisões realizadas no processo de desenvolvimento e suas subsequentes evoluções do sistema (Taylor *et al.* 2009).

Alguns autores dizem que é possível identificar práticas na arquitetura que podem impactar negativamente na realização de manutenções e evoluções futuras. Essas práticas são denominadas *Smells* Arquiteturais (Lippert e Roock 2006, Stal 2007, Garcia *et al.* 2009). Seguindo a linha de Beck *et al.* (1999) que definem *smells* de código como indicativos da necessidade de refatorações presentes no código-fonte, os *smells* arquiteturais representam indicativos da necessidade de refatorações arquiteturais (Mo *et al.* 2015). Portanto, a principal diferença entre ambos os *smells* é o nível de abstração em que são encontrados. Os *smells* de código-fonte estão relacionados a estruturas em nível de código-fonte como classe, métodos e parâmetros, por outro lado os *smells* arquiteturais estão relacionados a abstrações arquiteturais como o relacionamento entre classes, componentes, camadas e filtros. (Lippert e Roock 2006, Garcia *et al.* 2009).

Um exemplo de *smell* de código é o Código Duplicado que se refere à existência de uma mesma estrutura de código presente em mais de um lugar no sistema o que dificulta a manutenção e a evolução do sistema (Beck *et al.* 1999). Um exemplo de *smell* arquitetural é o Ciclo entre Classes que indica um relacionamento cíclico entre duas ou mais classes, como é mostrado na Figura 2.5. Na parte superior da figura é mostrado o ciclo mais simples possível, entre duas classes apenas. Na parte inferior é mostrado um ciclo com mais de duas classes, representando a possibilidade de ocorrer em diversas classes (Lippert e Roock 2006).

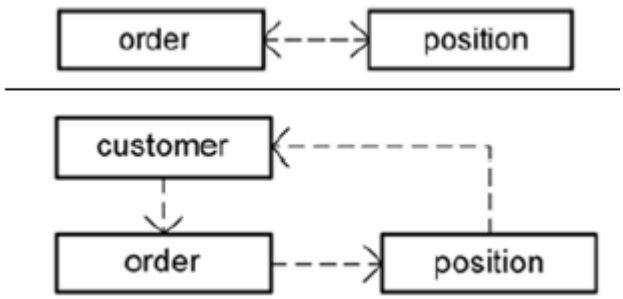


Figura 2.5 - Smell Arquitetural Ciclo entre Classes *(Lippert e Roock 2006)*

A presença do *Smell* Ciclo entre Classes leva e efeitos negativos em diversos atributos de qualidade dos quais podemos citar:

- Compreensibilidade, pois para o completo entendimento da execução de uma classe obriga o entendimento das demais classes do ciclo;
- Manutenibilidade, pois alterações em classes em um ciclo podem impactar todas as classes que fazem parte desse ciclo, podendo gerar um comportamento imprevisto do sistema;
- Reusabilidade, somente o reúso do ciclo como um todo é possível devido às dependências.

Apesar de ambos os tipos de *smells*, de código e arquitetural, impactarem os atributos de qualidade, é importante salientar que eles não representam problemas do sistema. Apenas indicam construções que podem afetar negativamente atributos de qualidade, como a compreensibilidade, testabilidade, evolução e reusabilidade. Isto é, nem sempre a presença de um *smell* é incorreta, por exemplo, considere o caso do padrão de projeto *Observer* (Gamma 1995), representado na Figura 2.6. Nessa figura é possível verificar que a classe *Subject* possui uma lista de *Observers* e na classe *Observer* no método *Update()* faz uma chamada do método *GetState()* da classe *Subject*. Isso caracteriza o *smell* Ciclo entre classes, contudo não significa que a utilização deste padrão seja incorreta.

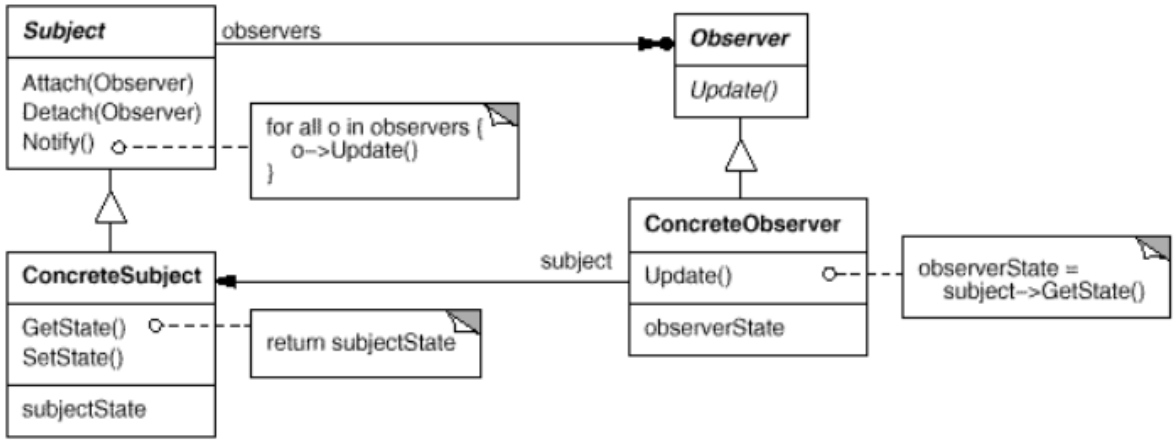


Figura 2.6 - Padrão de projeto Subject/Observer (Gamma 1995)

Alguns *smells* arquiteturais foram catalogados por Lippert e Roock (2006). Esses autores os classificaram de acordo com as abstrações em que são encontrados. Na Figura 2.7 são mostrados os *smells* agrupados em caixas que

representam os níveis de abstrações: classes, pacotes, subsistemas e camadas. Pode-se observar que os *smells* catalogados são bastante genéricos podendo ser aplicados a qualquer tipo de sistema. Como por exemplo, classes e pacotes não utilizados, pacotes com nome não significativo, pacotes e subsistemas pequenos ou grandes demais, violação de camadas, dentre outros.

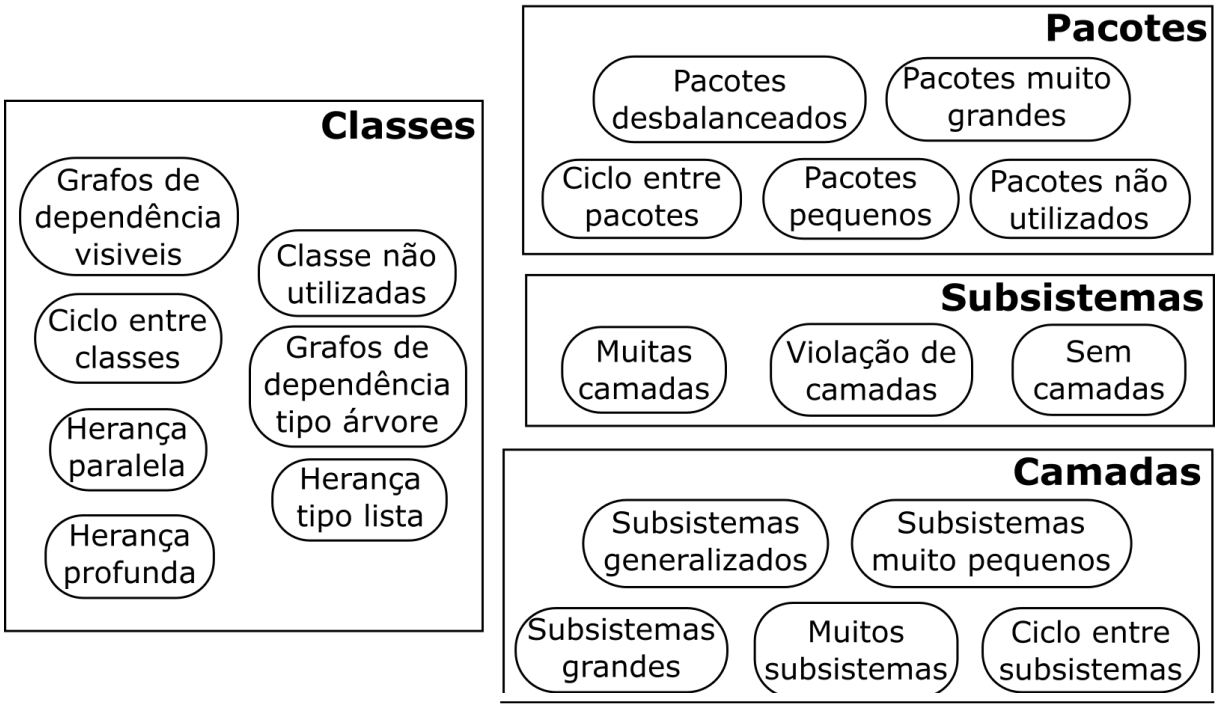


Figura 2.7 - Catálogo de *smells* arquiteturais adaptado de Lippert e Roock (2006)

2.4 Considerações Finais

Neste capítulo foram abordados os tópicos de Sistemas Adaptativos e *Smells* Arquiteturais. No primeiro tópico abordado definiu-se que Sistemas Adaptativos são sistemas capazes de alterar o seu comportamento de acordo com o contexto em que é executado sem a necessidade da intervenção do usuário. Tais sistemas podem ser divididos em dois subsistemas: o gerenciado e o gerenciador. O subsistema gerenciador representa o *loop* de controle, responsável pelas atividades de monitorar, analisar, planejar e executar as adaptações no subsistema gerenciado. A atividade de monitorar ou o interesse de monitoramento é responsável por coletar, pré-processar e enviar os dados, dando inicio ao processo de adaptação.

No segundo tópico abordado definiu-se que *Smells* Arquiteturais são práticas na arquitetura de um sistema que podem gerar problemas na manutenção e evolução desses sistemas. Portanto, os *Smells* Arquiteturais assim como os *smells* de código-fonte representam indícios da necessidade de se realizar refatorações com o intuito de melhorar as qualidades internas do software.

Capítulo 3

SMELLS ARQUITETURAIS DE MONITORAMENTO

3.1 Considerações Iniciais

O objetivo deste capítulo é propor dois *smells* arquiteturais no interesse de monitoramento em sistemas adaptativos. Na seção Exemplo Motivacional PhoneAdapter é apresentado o sistema adaptativo PhoneAdapter utilizado para facilitar a descrição dos *smells*. Nas duas seções subsequentes são apresentados os *smells Obscure Monitor* e *Oppressed Monitors* propostos. Nessas seções são fornecidos detalhes e características sobre cada *smell*, é proposto um guia para auxiliar na identificação do *smell* e são discutidos os impactos na manutenibilidade e evolução dos sistemas com a presença dos *smells*.

O capítulo está organizado em cinco seções. Na seção 3.1 são realizadas as considerações iniciais. Na Seção 3.2 é apresentado o exemplo motivacional PhoneAdapter. Na Seção 3.3 é proposto o *smell Obscure Monitor*. Na Seção 3.4 é proposto o *smell Oppressed Monitors*. Por fim, na Seção 3.5 são mostradas as considerações finais do capítulo.

3.2 Exemplo Motivacional PhoneAdapter

O *PhoneAdapter* (Sama *et al.* 2010) é uma aplicação desenvolvida para a plataforma *Android*, tendo como objetivo realizar adaptações no perfil de configuração do dispositivo móvel tendo como base os dados coletados do contexto e regras previamente definidas pelo usuário.

O perfil de configuração do dispositivo determina o comportamento do aparelho móvel, podendo envolver a intensidade de brilho da tela, o volume do aparelho, qual o modo de toque de chamadas (tocar, silencioso ou vibrar), o descobrimento de dispositivos *Bluetooth* dentre outros. Por exemplo, o perfil de configuração pode definir que o aparelho tenha a menor intensidade de brilho da tela, esteja no modo silencioso e com o descobrimento de dispositivos Bluetooth e GPS desligado com a finalidade de reduzir o gasto da bateria.

Os dados coletados do contexto pelo *PhoneAdaper* são provenientes do dispositivo *Bluetooth*, GPS, dia da semana e hora atual. Isto é, o sistema deve monitorar quatro interesses. É importante observar que os interesses monitorados possuem características de dados bastante distintas, por exemplo, os dados do dia da semana e de dispositivos Bluetooth podem variar muito pouco, contudo a hora do sistema e os dados do GPS podem se alterar constantemente.

As regras definidas previamente pelo usuário são definições de perfis de configuração criadas conforme o contexto do celular. Isto é, o usuário é capaz de definir um determinado comportamento do aparelho celular para diferentes contextos em que o sistema é executado. Por exemplo, o celular deve ficar no modo silencioso quando a localização lida pelo GPS seja a localização do trabalho usuário.

Portanto, o *PhoneAdapter* é caracterizado como um sistema adaptativo, pois realiza o monitoramento do contexto de maneira constante e em paralelo com a execução do sistema. Além disso, adapta o sistema automaticamente conforme os dados coletados do contexto.

O subsistema gerenciador do *PhoneAdapter* possuí três classes principais como é mostrado no diagrama de classes simplificado na Figura 3.1. A figura é dividida em duas partes, sendo que na parte superior estão representadas as três classes principais do subsistema gerenciador e na parte inferior uma classe representando o subsistema gerenciado. Neste diagrama optou-se em mostrar

apenas as classes mais relevantes à discussão, sendo excluídas as classes apenas relacionadas à aplicação, como por exemplo, as classes relacionadas a telas da aplicação.

As classes que envolvem o subsistema gerenciador são a `ContextManager`, `AdaptationManager` e `MyDbAdapter`. Seguindo a sequência do *loop* de controle, o primeiro passo é monitorar os dados, este processo é realizado pela classe `ContextManager`. Após a coleta dos dados é necessário analisar a necessidade de adaptações com base em valores de referência, este processo é realizado pela classe `AdaptationManager`, ela também é responsável por planejar qual das adaptações será realizada e executar as adaptações selecionadas alterando os parâmetros do perfil de configuração. Por fim, a classe `MyDbAdapter` é a responsável por recuperar do banco de dados os valores de referência bem como os perfis de configuração previamente definidos pelo usuário. Como o foco deste trabalho é analisar o monitoramento, a classe `ContextManager` é descrita em maiores detalhes a seguir.

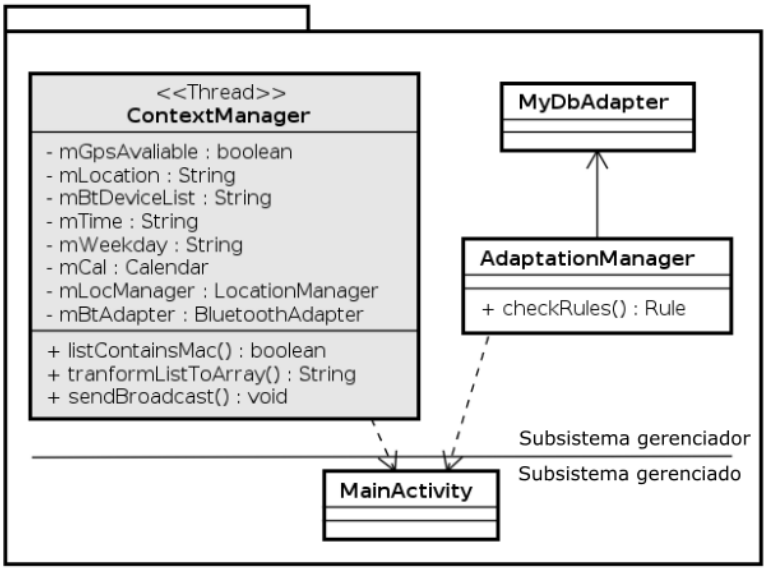


Figura 3.1 - Diagrama de classes da aplicação PhoneAdapter (fonte: elaborada pelo autor)

A classe `ContextManager` é responsável por monitorar quatro interesses: GPS, Bluetooth, horário e dia da semana. Os dados monitorados são armazenados, respectivamente, nos atributos: `mLocation`, `mBtDeviceList`, `mTime` e `mWeekday`. Para a coleta desses dados há a necessidade de instanciar os sensores responsáveis por prover os dados, neste caso é utilizado o atributo `mLocManager`

para o sensor de GPS, o `mBtAdapter` para o sensor de Bluetooth, o atributo `mCal` para o sensor de hora e dia da semana. Nesta classe também estão presentes os métodos `listContainsMac()` e `transListToArray()` relacionados ao pré-processamento do bluetooth. O método `sendBroadcast()` está relacionado ao envio dos dados coletados para a classe `AdaptationManager`.

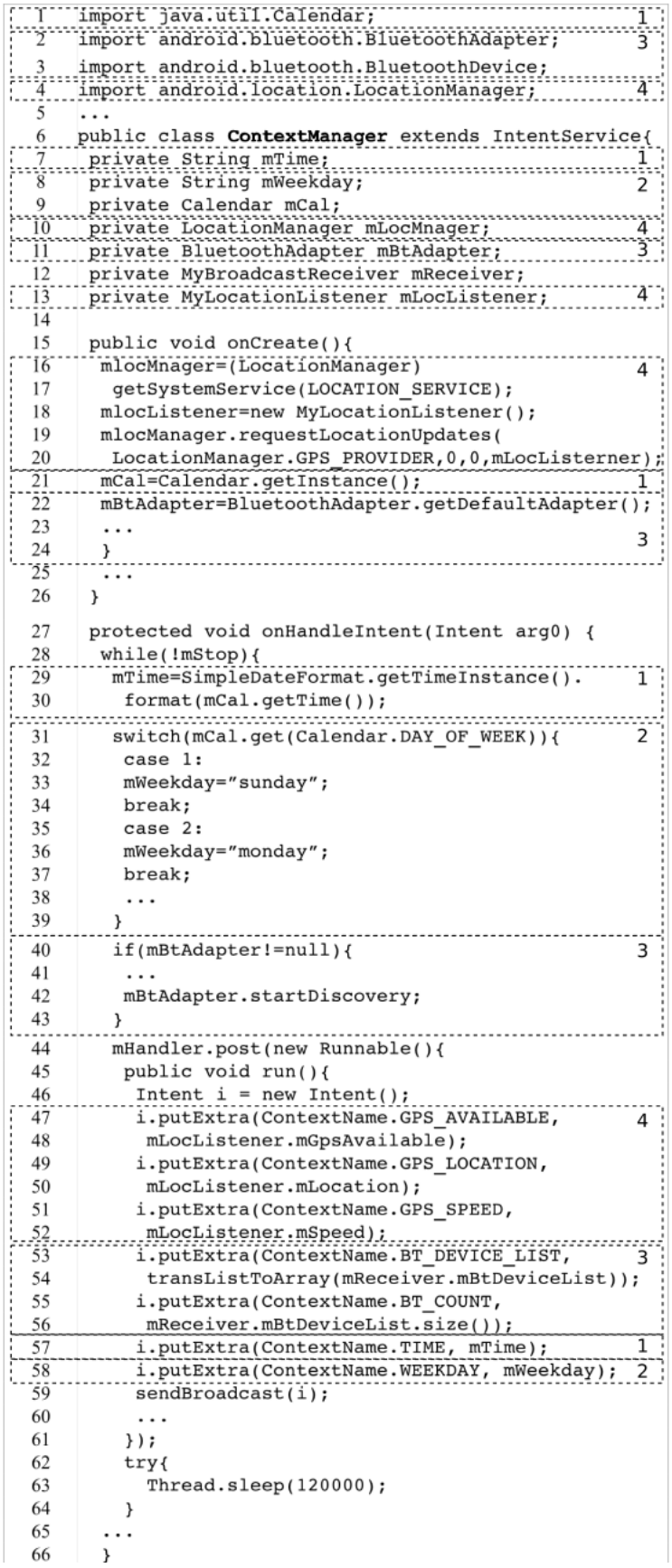
Para evidenciar algumas características importantes desta classe, na Figura 3.2 está representado um trecho de código da classe `ContextManager` na qual estão realçadas caixas pontilhadas numeradas de um a quatro. Cada caixa representa os trechos de código relacionados a um monitor como listados na Tabela 3.1.

Tabela 3.1 - Trecho de código e o monitor correspondente

Trecho	Monitor
1	Horário
2	Dia da semana
3	Bluetooth
4	GPS

Na Figura 3.2 é mostrado que a classe `ContextManager` estende *IntentService* que é uma classe semelhante a classe Thread do JAVA, isto permite que esta classe seja executada em paralelo à aplicação, uma característica importante do monitoramento de sistema adaptativos. O código-fonte é descrito em detalhes a seguir:

- Da linha 1 até a 4 estão as importações das APIs (Interface de Programação de Aplicativos) dos sensores de GPS, Bluetooth e Calendar relacionados ao processo de monitoramento;
- Da linha 7 até a 13 estão os atributos relacionados ao instanciação dos sensores e os atributos relacionados ao armazenamento dos dados provenientes dos sensores, já descritos anteriormente;



```
1   import java.util.Calendar;                                    1
2   import android.bluetooth.BluetoothAdapter;                    3
3   import android.bluetooth.BluetoothDevice;
4   import android.location.LocationManager;                      4
5   ...
6   public class ContextManager extends IntentService{
7    private String mTime;                                        1
8    private String mWeekday;                                     2
9    private Calendar mCal;
10   private LocationManager mLocMnager;                          4
11   private BluetoothAdapter mBtAdapter;                         3
12   private MyBroadcastReceiver mReceiver;
13   private MyLocationListener mLocListener;                     4
14
15   public void onCreate(){
16    mlocMnager=(LocationManager)                                4
17     getSystemService(LOCATION_SERVICE);
18    mlocListener=new MyLocationListener();
19    mlocManager.requestLocationUpdates(
20     LocationManager.GPS_PROVIDER,0,0,mLocListerner);
21    mCal=Calendar.getInstance();                                 1
22    mBtAdapter=BluetoothAdapter.getDefaultAdapter();
23    ...
24    }                                                           3
25    ...
26   }
27   protected void onHandleIntent(Intent arg0) {
28    while(!mStop){
29     mTime=SimpleDateFormat.getTimeInstance().                  1
30      format(mCal.getTime());
31     switch(mCal.get(Calendar.DAY_OF_WEEK)){                    2
32      case 1:
33      mWeekday="sunday";
34      break;
35      case 2:
36      mWeekday="monday";
37      break;
38      ...
39     }
40     if(mBtAdapter!=null){                                      3
41      ...
42      mBtAdapter.startDiscovery;
43     }
44     mHandler.post(new Runnable(){
45      public void run(){
46       Intent i = new Intent();
47       i.putExtra(ContextName.GPS_AVAILABLE,                    4
48        mLocListener.mGpsAvailable);
49       i.putExtra(ContextName.GPS_LOCATION,
50        mLocListener.mLocation);
51       i.putExtra(ContextName.GPS_SPEED,
52        mLocListener.mSpeed);
53       i.putExtra(ContextName.BT_DEVICE_LIST,                   3
54        transListToArray(mReceiver.mBtDeviceList));
55       i.putExtra(ContextName.BT_COUNT,
56        mReceiver.mBtDeviceList.size());
57       i.putExtra(ContextName.TIME, mTime);                     1
58       i.putExtra(ContextName.WEEKDAY, mWeekday);               2
59       sendBroadcast(i);
60       ...
61      });
62      try{
63        Thread.sleep(120000);
64      }
65    ...
66   }
```

Figura 3.2 - Classe ContextManager do PhoneAdapter (fonte: elaborada pelo autor)

- Da linha 15 até a 24 está o método `onCreate()` que é o primeiro método a ser executado o realizar a instanciação da classe `ContextManager`, similar a um método construtor. Neste método são inicializados todos os sensores e variáveis que serão utilizados durante a execução da classe;
- Da linha 27 até a 66 está o método `onHandleIntent()`, sendo ele o método principal da classe, no qual toda a lógica da classe é executada. Este método semelhante ao método `run()` da Thread, no qual ao final de sua execução a classe é finalizada. Portanto para evitar que a classe seja finalizada, é definido um *loop* `while` da linha 28 até a linha 65 o qual é executado constantemente enquanto a aplicação estiver ativa;
- Da linha 29 até a 69 estão as execuções da lógica de monitoramento, pré-processamento e envio dos dados de todos os monitores estão agrupados dentro do *loop* `while`;
- Na linha 63 existe uma pausa na execução da classe de 2 minutos (120000 ms), isto é, o ciclo do *loop* ocorre a cada período de dois minutos. Ou seja, este tempo definido é referente a taxa de execução dos monitores, a taxa de monitoramento;
- Da linha 67 até a 74 o método `transListToArray()` está relacionado ao pré-processamento dos dados obtidos pelo sensor de Bluetooth, este método só foi inserido para demonstra que o pré-processamento também pode ser executado em um método ou classe externa, mas deve ser chamado dentro da execução do *loop*.

A construção do interesse de monitoramento do *PhoneAdapter* possui três características importantes a serem destacadas:

1) A independência de execução entre os monitores. Na Figura 3.2 são mostradas caixas numeradas referentes ao código de cada monitor, sendo possível verificar que não existem caixas sobrepostas, isto é, em nenhum momento seja na lógica de monitoramento, pré-processamento ou envio dos dados, um monitor influencia na execução de outro monitor, garantindo a independência entre os monitores.

2) A falta de modularização dos monitores. A partir das caixas numeradas também é possível verificar que o código referente a cada monitor está espalhado e entrelaçado na classe `ContextManager`, ou seja, o código de cada monitor apesar de ser independente não está bem modularizado.
3) A execução de todos os monitores está submetida à execução de um mesmo *loop* e uma mesma taxa de monitoramento. Excluindo-se a instanciação dos sensores, as demais lógicas de cada monitor são executadas sob um mesmo *loop* `while`, portanto todos os monitores estão submetidos à frequência de execução deste *loop*.

3.3 Smell Obscure Monitor

O *smell* arquitetural *Obscure Monitor* ocorre em um Sistema Adaptativo quando um monitor independente é implementado com uma granularidade incorreta tornando-se obscuro no código-fonte. Isto é, no paradigma da programação orientada a objetos este elemento não é uma classe evidente no código-fonte. Ou seja, os monitores são implementados em uma mesma classe, podendo até mesmo estar implementados em classes não relacionadas apenas com o interesse de monitoramento. É importante ressaltar que mesmo em sistemas que possuam uma classe na qual o interesse de monitoramento é modularizado, a granularidade proposta neste trabalho é que cada monitor independente seja expresso como uma classe individual. Ou seja, cada sensor monitorado deve ser considerado um interesse separado, portanto devidamente modularizado em uma classe.

No exemplo motivacional, o *PhoneAdapter*, as caixas numeradas evidenciam cada monitor presente na aplicação. Nesse exemplo, os monitores se encontram espalhados e entrelaçados na classe `ContextManager` tornando-os obscuros no código-fonte. Dessa forma, tarefas de manutenção relacionadas aos monitores ou o reúso de um único monitor são dificultadas, por exemplo, caso queira utilizar-se do monitor de dia da semana em outro projeto, o código-fonte deverá ser reescrito, pois este componente não está modularizado.

Alguns autores já argumentam que os elementos do *loop* de controle devem ser modularizados como entidades de primeira classe, tornando-os evidentes no

código-fonte (Garlan *et al.* 2004, Brun *et al.* 2009, Garcia *et al.* 2009, Weyns *et al.* 2013). Seguindo essa mesma linha de pensamento, este *smell* propõe a necessidade de evidenciar cada interesse monitorado em um elemento monitor separado, ou seja, cada monitor deve estar modularizado em classes específicas.

3.3.1 Identificação do Smell

As orientações a seguir objetivam auxiliar o desenvolvedor na identificação do elemento monitor em um sistema, possibilitando-o analisar a presença ou não do *smell Obscure Monitor.* Com a identificação dos monitores é possível verificar se o esse elemento se encontra entrelaçado e espalhado pelo código-fonte do sistema. Para facilitar o processo de identificação, esse foi dividido em três passos:

1. **Identificar todos os sensores utilizados pelo sistema.** Como descrito anteriormente, sabe-se que o elemento monitor está intrinsecamente relacionado com um sensor, o qual provê ao monitor dados do contexto. Isto é, um monitor de proximidade necessita de um sensor ultrassônico para realizar as leituras de distância, assim como um monitor de presença necessita de um sensor de movimento. Sabendo-se disso é possível concluir que a presença de um sensor pode, portanto evidenciar a presença de um monitor.
   Como os sensores de ambiente são hardwares específicos de plataforma, sendo esses abstraídos em APIs, uma maneira eficaz de identificar os sensores é por meio da documentação da plataforma que deve conter as classes e pacotes utilizados para a implementação dos sensores.

2. **Identificar as classes que utilizam sensores.** Admitindo-se que realizado o passo anterior, existe uma lista contendo todos os possíveis sensores utilizados pelo sistema. Nesse passo, então, será realizada a busca por classes que implementem ou fazem uso de uma ou mais das APIs presentes na lista. Realizando-se então a busca por sensores em todas as classes.

3. **Identificar o código do monitor.** Neste último passo é realizada a identificação propriamente dita do elemento monitor. Devido à especificidade de cada monitor, esta tarefa é de responsabilidade do desenvolvedor. Ele deve identificar no

código-fonte a lógica relacionada a cada elemento monitor, o que inclui a lógica de monitoramento, pré-processamento e de envio dos dados coletados.

Alguns métodos existentes podem auxiliar neste processo de identificação dos trechos de código-fonte relacionados a cada monitor como. Por exemplo, o método Grafo Def-Uso (*Def-use Graph*) proposto por Rapps e Weyuker (1982) que permite que sejam exploradas as interações que envolvem a atribuição de uma variável e seus subsequentes usos.

Após realizar estes três passos para a identificação dos monitores o desenvolvedor deve verificar se esse elemento está evidente ou obscuro no código-fonte indicando a presença do *Obscure Monitor*.

3.3.2 Impactos na Qualidade

Os principais impactos relacionados com a presença do *smell Obscure Monitor* estão relacionados à reusabilidade e compreensibilidade do elemento monitor. O impacto sobre a reusabilidade é devido à falta de modularização deste elemento o que reduz a possibilidade em utilizar o elemento em outros sistemas. Por exemplo, em um novo sistema há a necessidade de se implementar um monitoramento de distância ultrassônico. Esse mesmo tipo de monitor já está implementado em outro sistema, porém encontra-se obscuro no código-fonte o que impossibilita reutilizar o código já desenvolvido e utilizado. Desta forma, a solução será desenvolver novamente o código do monitor no novo sistema.

O impacto sobre a compreensibilidade está relacionado à especificidade de cada monitor. Isto é, os dados monitorados podem estar relacionados a interesses bem diferentes como um monitor para localização geográfica e outro monitor relacionado à detecção de movimento. Nesse caso, cada monitor realiza um tipo de pré-processamento específico ao dado monitorado, podendo até mesmo utilizar de algoritmos e fórmulas complexas. Estando esses códigos entrelaçados a compreensibilidade do funcionamento da classe será comprometida.

3.4 Smell Opressed Monitors

O *smell Oppressed Monitors* é caracterizado por um conjunto de monitores que exibem três características principais:

1. Os monitores são independentes entre si quanto aos dados pré-processados;
2. Os monitores possuem a mesma taxa de monitoramento e;
3. A ordem de execução dos monitores é predeterminada e não modificável em tempo de execução.

Essas características ocorrem principalmente pela implementação dos monitores em uma única estrutura de *loop* reduzindo a autonomia individual de cada um deles. É importante salientar que essas características podem ocorrer mesmo que os monitores estejam todos modularizados em classes separadas. O principal componente neste *smell* é a presença de uma única estrutura responsável por gerenciar todos os monitores.

A primeira característica que diz respeito à independência entre os monitores. Essa situação ocorre com frequência, contudo caso exista a dependência entre monitores, haverá a necessidade de uma ordem de execução. Sendo assim, a refatoração deve ser realizada atentando-se a esta situação.

A segunda característica relacionada aos monitores possuírem a mesma taxa de monitoramento é causada devido a todos os monitores compartilharem a mesma estrutura de *loop*. Com isso, todos os monitores são executados de acordo com a frequência do *loop* e não uma definida exclusivamente para eles. Isto gera a captura de dados desnecessária para alguns dos monitores resultando em um gasto de recursos desnecessário. Em muitas situações a melhor alternativa é que cada monitor possua a sua própria taxa de monitoramento possibilitando que a taxa de monitoramento seja alterada durante a execução do sistema.

A terceira característica que diz respeito à ordem de execução restrita ocorre quando os monitores são dispostos em sequência em uma mesma estrutura de *loop*. Isso ocorre ao colocar os monitores em uma lista fixa ou quando o código-fonte dos monitores está implementado de maneira estruturada. Assim, um monitor só pode iniciar após a execução do monitor anterior e assim por diante. Esta última característica pode gerar dois problemas:

1. **Interrupção na execução do monitoramento.** Caso um monitor gere um erro, todos os monitores da sequência estarão comprometidos, podendo não ser executados.
2. **Geração de contextos errôneos.** Se um monitor tiver uma execução lenta isto aumentará a chance dos monitores em sequência de coletarem dados que não correspondem mais ao mesmo contexto dos monitores anteriores, gerando contextos errôneos.

Em resumo o *smell Oppressed Monitors* envolve duas principais abstrações: um gerenciador de monitores e os monitores. O gerenciador de monitores é responsável por coordenar a execução de todos os monitores. Portanto, para a refatoração deste *smell* é necessária à remoção do gerenciador de monitores tornando cada monitor independente. É importante ressaltar que tanto o gerenciador de monitores quanto os monitores podem ser implementados como classes ou métodos.

No caso do exemplo motivacional, o *PhoneAdapter*, a classe `ContextManager` é responsável por gerenciar os monitores. Nessa classe pode-se observar que os monitores evidenciados nas caixas numeradas estão submetidos a um mesmo *loop* e a uma mesma taxa de monitoramento. Desta forma, os monitores perdem sua independência de execução o que pode levar a dificuldades em aplicar evoluções como, por exemplo, definir uma taxa de monitoramento individual para cada monitor.

3.4.1 Identificação do Smell

A identificação deste *smell* se baseia em encontrar um componente responsável por gerenciar os monitores. Isto pode ser realizado por meio das seguintes orientações:

1. **Identificar todos os monitores.** Para realizar este passo são utilizadas as mesmas orientações para encontrar os monitores, descritas na seção anterior do *Smell Obscure Monitor*: Identificar os sensores utilizados pelo sistema, Identificar as classes que utilizam sensores e por fim, Identificar o código do monitor.

2. **Identificar classes que se relacionem com monitores.** Após encontrar todos os monitores no passo anterior, neste passo é necessário identificar o gerenciador de monitores. Para este passo é importante verificar se os monitores encontrados se encontram modularizados ou não. Caso os monitores não se encontrem modularizados apresentando o *smell Obscure Monitor*, em geral, o *Oppressed Monitors* é expresso na mesma classe em que se encontram os monitores. Para verificar-se a presença do *smell*, deve existir uma estrutura de *loop* que esteja englobando todos os monitores ao mesmo tempo. Por outro lado, caso os monitores se encontrem modularizados, o gerenciador de monitores pode ser outra classe possuindo relacionamento com os diversos monitores. Para encontrar esta classe basta procurar por classes que se relacionem com estes monitores.

3. **Identificar o gerenciador de monitores.** Após realizar o passo anterior, como resultado, são identificadas as classes que utilizam um ou mais sensores. A classe com maior probabilidade de apresentar o *Oppressed Monitors* será a classe que possui maior número de relacionamento com os monitores. Determinada essa classe é necessário verificar a sua lógica de execução. Como descrito anteriormente, o *Oppressed Monitors* será expresso caso nesta classe exista um *loop* que submeta todos os monitores a uma mesma taxa de monitoramento e uma ordem de execução fixa.

3.4.2 Impactos na Qualidade

Em algumas situações o *smell Oppressed Monitors* é aceito, por exemplo, quando os monitores são similares necessitando uma mesma taxa de monitoramento. Nessa situação o *smell* é aceitável, pois ele cria apenas uma *thread* para realizar o monitoramento, evitando assim a criação de *threads* separadas para cada monitor o que pode ocasionar a uma disputa excessiva pelo controle do processador, conhecido também como "*overhead*". Contudo, em outros casos em que os monitores possuem requisitos muito diversos, por exemplo, quando há necessidade de monitorar o dia da semana e o posicionamento global (GPS). No monitor do dia da semana, os dados se alteram a cada 24h não havendo a necessidade de uma taxa de monitoramento frequente. Enquanto que o monitor de

GPS pode ter seus dados alterados a cada instante. Nesse caso, a presença do *Oppressed Monitors* pode gerar o gasto desnecessário de recursos ao definir-se uma taxa de monitoramento que comporte as mudanças de contexto do GPS, ou por outro lado, pode-se definir uma taxa de monitoramento menor condizente com as alterações no contexto dos dias da semana levando a uma perda de dados gerados pelo GPS. Quando isso ocorre, respeita-se a taxa de frequência do monitor com maior alteração de dados, neste caso o GPS.

Outro impacto importante está relacionado a dependência de execução entre os monitores. Ou seja, como são executados em uma sequência lógica os monitores se tornam dependentes da execução do monitor anterior. Portanto, alterações de evolução ou manutenção em algum dos monitores pode impactar na correta captação de contextos pelos demais monitores. Isto é, caso um monitor tenha sua lógica alterada levando um longo tempo para realizar seu processamento, os demais monitores em sequência podem demorar a serem executados coletando um contexto que não condiz mais com o contexto coletado pelos monitores executados anteriormente. Esta situação pode levar a comportamentos não esperados.

3.5 Considerações Finais

Neste capítulo foram propostos dois *smells* arquiteturais relacionados ao interesse de monitoramento em sistemas adaptativos. A proposta destes *smells* foi realizada por meio de uma análise em sistemas adaptativos encontrados na literatura e em repositórios de código-fonte. O *smell Obscure Monitor* está relacionado à disposição do código-fonte dos monitores, enquanto que o *Oppressed Monitor* está relacionado ao seu comportamento. Contudo, uma limitação é que apesar da análise realizada é possível que existam outros *smells* relacionados ao interesse de monitoramento não encontrado, envolvendo outras características não identificadas nos sistemas analisados.

Capítulo 4

EVIDÊNCIA DA EXISTÊNCIA DOS *SMELLS* PROPOSTOS

4.1 Considerações Iniciais

O objetivo deste capítulo é apresentar evidências da presença dos *smells* propostos neste trabalho. Na seção Metodologia e Levantamento são detalhados os procedimentos realizados para a busca e seleção dos sistemas avaliados. Na seção Projetos Selecionados de Repositórios são mostrados os *smells* nos sistemas selecionados na seção anterior. Para cada sistema há uma breve descrição de suas funcionalidades, um detalhamento da implementação do interesse de monitoramento e por fim os *smells* são evidenciados. Na seção Projetos Selecionados de Artigos são evidenciados a presença dos *smells* em sistemas descritos em artigos da literatura.

O capítulo está organizado em cinco seções. Na Seção 4.1 são realizadas as considerações iniciais. Na Seção 4.2 são descritos os passos para a seleção dos sistemas avaliados. Na Seção 4.3 são descritos e analisados sistemas adaptativos encontrados em repositórios de código-fonte *online*. Na Seção 4.4 são descritos e analisados sistemas adaptativos encontrados a partir de artigos. Por fim, na Seção 4.5 são mostradas as considerações finais do capítulo.

4.2 Metodologia de Levantamento

O levantamento dos sistemas adaptativos foi realizado por meio de uma busca em dois repositórios *online.* Esses repositórios armazenam programas de código-fonte aberto e acessíveis ao público. Os repositórios utilizados foram: Bitbucket[2] e Github[3]. Os sistemas levantados passaram por um processo de seleção e foram escolhidos apenas sistemas adaptativos cujo interesse de monitoramento é executado de forma independente, isto é, em paralelo com a aplicação base. Assim a aplicação gera constantemente dados e os monitores são os responsáveis por capturar esses dados de tempos em tempos. Dessa forma, não foram considerados sistemas cujo interesse de monitoramento é dependente da execução da aplicação base. Isso ocorre quando o monitoramento só é executado quando os dados são gerados. O levantamento dos sistemas foi dividido em cinco (5) etapas listadas a seguir:

1. **Busca por projetos em repositórios**. Para realizar a busca utilizou-se o termo "*adaptive system*". Foi adicionado apenas o filtro por linguagem de programação sendo apenas selecionados os projetos desenvolvidos em JAVA. A escolha em analisar apenas sistemas desenvolvidos nessa linguagem deve-se ao fato de ser considerada a linguagem de programação mais utilizada segundo o índice Tiobe[4]. Nesta etapa foram selecionados 85 sistemas, cinco projetos encontrados no repositório Bitbucket e 80 projetos no Github.

2. **Remoção de projetos repetidos.** Nesta etapa projetos com o mesmo nome e descrição foram analisados verificando o código-fonte. Em casos que os projetos possuíssem o mesmo nome e código-fonte, apenas um foi selecionado, sendo excluídos os demais repetidos. Ao fim desta etapa restaram 62 projetos.

3. **Remoção de projetos a partir das descrições.** Nesta etapa os projetos foram selecionados a partir de suas descrições. Foram removidos os projetos cuja a

[2] https://bitbucket.org/

[3] https://github.com/

[4] *www.tiobe.com/tiobe_index*

descrição não estava relacionada a sistemas adaptativos. Quando o projeto não possuía descrição era automaticamente adicionado à próxima etapa de seleção. Ao fim desta etapa restaram 39 projetos.

4. **Remoção de projetos incompletos.** Nesta etapa analisou-se o código-fonte de todos os projetos e foram excluídos projetos vazios ou que continham classes incompletas. É importante ressaltar que projetos que continham classes incompletas, contudo não interferiam na análise do interesse de monitoramento não foram excluídos. Ao fim desta etapa restaram 33 projetos.

5. **Análise dos Projetos.** Como resultado das etapas anteriores restou 33 projetos a serem avaliados. Esses projetos foram verificados um a um analisando-se o código-fonte de cada projeto. Nessa análise procurou-se a presença de um subsistema gerenciador. Dessa forma, foram excluídos sistemas cujo processo de adaptação era resultado do emprego de condicionais simples do tipo *if/else* e *switch/case*. Isto é, não foram considerados sistemas que a adaptação estava relacionada apenas com a execução de uma condicional em trechos específicos do código-fonte, pois este tipo de adaptação não exige a execução de um monitoramento constante. Ao fim desta etapa foram selecionados quatro projetos que apresentavam algum subsistema gerenciador.

No sentido de completar o conjunto de sistemas obtidos, também foram considerados artigos científicos que apresentavam trechos de código de sistemas adaptativos. Os seguintes artigos listados na Tabela 4.1 foram analisados porque apresentavam sistemas adaptativos e era possível observar ou inferir detalhes do interesse de monitoramento. É importante ressaltar que esses sistemas encontrados a partir de artigos foram incluídos durante o processo de elaboração do projeto, ou seja, não foi realizado um levantamento sistemático para a seleção de tais sistemas. Estes artigos foram encontrados a partir de referências utilizadas na elaboração do presente estudo.

Na Tabela 4.2 são mostrados os sistemas selecionados a partir de repositórios e os selecionados a partir de artigos. Esta tabela é dividida em duas colunas, na qual a primeira coluna possui os sistemas selecionados a partir dos

repositórios e a segunda coluna os sistemas selecionados a partir de outros meios como na leitura de artigos da área.

Tabela 4.1 - Trabalhos selecionados a partir de artigos

Trabalhos selecionados	Referência
Behavior-based robotics	Arkin (1998)
A framework for developing mobile, context-aware applications.	Biegel e Cahill (2004)
Context-aware adaptive applications: Fault patterns and their automated identification.	Sama *et al.* (2010)
SmarTrAC: A smartphone solution for context-aware travel and activity capturing.	Fan *et al.* (2015)

Tabela 4.2 - Sistemas selecionados para verificação dos *smells*

Sistemas Selecionados	
Repositórios online	**Outros meios**
AIASProject	Novi (Amarasekara *et al.* 2014)
Zanshin	SmarTrAC (Fan *et al.* 2015)
Dc4cities	Context-aware framework (Biegel e Cahill 2004)
Self-protect-project	Behavior-based programming (Arkin 1998)
	PhoneAdapter[5] (Sama *et al.* 2010)

Para um melhor entendimento de cada projeto, o diagrama de classes de cada um foi recuperado utilizando-se a ferramenta MoDisco. Essa ferramenta recebe como entrada o código-fonte JAVA de um sistema e é capaz de gerar o diagrama de classes do sistema por meio de alguns passos de transformação.

Para a visualização do diagrama de classes UML foi utilizada a ferramenta Papyrus. Por meio da visualização do diagrama de classes a verificação da existência de um componente responsável pelo interesse de monitoramento foi facilitada, bem como a sua relação com os demais componentes do sistema, possibilitando assim uma melhor análise.

[5] O sistema PhoneAdapter não será descrito neste capítulo pois já foi descrito como exemplo motivacional no Capítulo 3 - .

4.3 Projetos Selecionados de Repositórios

Nesta seção os sistemas selecionados são apresentados com maiores detalhes, sendo apresentada uma breve descrição sobre as funcionalidades gerais do sistema, parte do diagrama de classes relacionado ao monitoramento e também são apresentados trechos do código-fonte referentes ao elemento monitor.

4.3.1 AIASProject

O projeto AIASProject (Uwa 2013) (*Artificial Inteligence Adaptive System)* é um robô de batalha cuja principal função é localizar outros robôs para duelar. Esse robô é controlado por um sistema adaptativo que utiliza métodos de inteligência artificial para auxiliar na tomada de decisões. As adaptações realizadas por ele envolvem analisar o contexto atual e decidir direções a serem tomadas, a movimentação da arma e controle do radar. Para que essas decisões possam ser tomadas, os dados monitorados são: a posição atual do robô; o sentido da movimentação; a direção da arma e a posição do radar. Dessa forma, esse sistema possui quatro monitores: Monitor de posição atual, Monitor de movimentação, Monitor de direção da arma e Monitor de posição do radar.

Na análise desse sistema foi identificado que esses monitores se encontram obscuros no código-fonte, a leitura dos dados monitorados é realizada em uma sequência pré-definida não alterável durante a execução e possuem a mesma taxa de monitoramento. Portanto, apresenta os *smells Obscure Monitor* e o *Oppressed Monitors*.

Na Figura 4.1 é mostrada uma parte do diagrama de classes do sistema AIASProject. Como pode ser observado, não há classes com nomes que sugerem a presença dos quatro monitores mencionados. Isso é um indicativo de que o código-fonte dos monitores encontra-se obscuro nesse sistema. Também pode ser observado um padrão bastante comum de implementação de sistemas adaptativos, que é a presença de uma thread com um método `run()`. A interface *Strategy* e a classe `StrategyFactory` estão relacionadas ao planejamento e à tomada de decisão executadas pelo robô, e a classe `RobotData` é responsável por armazenar os dados relacionados ao contexto atual do sistema.

A thread `ScriptBot` é responsável por realizar todos os processos necessários para a execução do robô, incluindo todo o interesse de monitoramento, o planejamento das ações a serem tomadas e a execução das ações. Assim, todo o código responsável pelo monitoramento encontra-se dentro dessa classe, mais especificamente dentro do método `run()`.

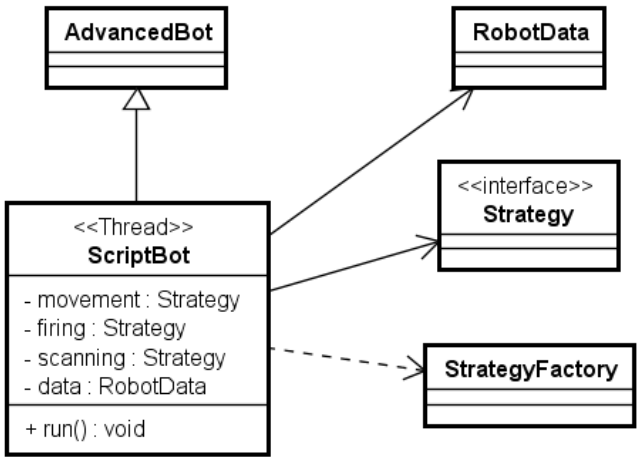


Figura 4.1 - Diagrama de Classes do sistema AIAS (fonte: elaborada pelo autor)

Na Figura 4.2 mostra-se esquematicamente a execução do método `run()`, o qual é responsável por todo o ciclo de adaptação (*loop de controle*), incluindo o interesse de monitoramento. Esse método executa três etapas distintas, as quais podem ser vistas na Figura 4.2 com os nomes: atualizar dados; plano de ação e execução. A caixa posicionada do lado direito da etapa "atualizar dados" representa os dados que são monitorados.

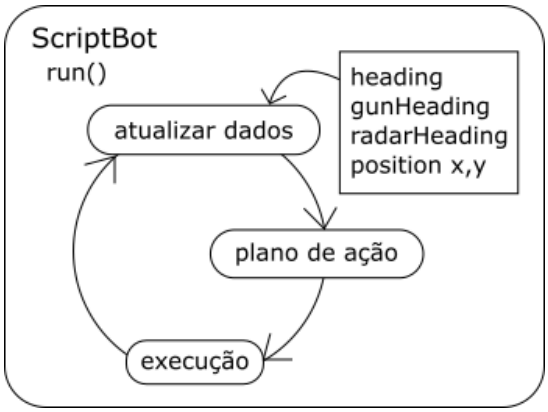


Figura 4.2 - Esquema de monitoramento do sistema AIAS
(fonte: elaborada pelo autor)

Essa figura também procura evidenciar que o método `run()` possui um *loop* que é executado constantemente durante a operação do robô. O monitoramento é realizado na etapa "atualizar dados", em que os dados de direção do robô, dados de direção da arma, dados de direção do radar e dados da posição do robô são coletados. A partir dos dados monitorados é realizada a etapa de “plano de ação” e por fim a etapa de “execução”.

Na Figura 4.3 é mostrado um trecho do código-fonte da classe `ScriptBot`. Nessa figura da linha 5 até linha 10 são carregadas as especificações do robô. Da linha 11 até a linha 30 é executado o *loop* responsável por monitorar os dados, planejar as ações e executar as ações planejadas. Os monitores encontram-se entre as linhas 14 e 18 dispostos na seguinte sequência: Monitor de direção do robô (`data.heading`), Monitor de direção da arma (`data.gunHeading`), Monitor de posição do radar (`data.scannerHeading`) e Monitor de posição atual (`data.x` e `data.y`).

```
1  public class ScriptBot extends AdvancedBot {
2     public void run() {
3     // load from file
4     try {
5     FileReader fr = new
   FileReader(BattleRunner.RobocodePath
6     + "/robots/maxsbots/scriptBot
7     ...
8     // config and setup
9     data = new RobotData();
10    ...
11    while (true) {
12          data.scannedAge++;
13          // update robot data
14          data.heading = this.getHeading();
15          data.gunHeading = this.getGunHeading();
16          data.scannerHeading = this.getRadarHeading();
17          data.x = this.getX();
18          data.y = this.getY();
19          // do movement
20          move = movement.getNextMove(data);
21          this.setAhead(move[0]);
22          this.setTurnLeft(move[1]);
23          // do scanning
24          move = scanning.getNextMove(data);
25          this.setTurnRadarLeft(move[0]);
26          // move gun
27          move = gun.getNextMove(data);
28          this.setTurnGunLeft(move[0]);
```

Especificação dos sensores

Monitores

Figura 4.3 - Trechos da Classe ScriptBot do sistema AIASProject (fonte: elaborada pelo autor)

Na Tabela 4.3 são mostrados os locais do código-fonte do projeto AIAS em que os *smells* podem ser detectados. Essa tabela possui duas partes; a superior trata do *smell Obscure Monitor* e a parte inferior do *smell Opressed Monitors*. Na primeira parte podem-se ver os quatro monitores existentes no sistema, suas lógicas e os locais no código-fonte em que essas lógicas estão presentes. A segunda parte também indica os locais do código-fonte em que o *smell Oppressed Monitors* se torna evidente.

Tabela 4.3 - Evidência da presença dos *smells* no sistema AIASProject

AIAS		
Obscure Monitors		
	Lógica do monitor	**Local no Sistema**
Obscure Monitor Heading	Sensores	Linhas 5 e 6
	Lógica de Monitoramento	Linha 14
	Lógica de Pré-processamento	-
	Lógica de Envio de Dados	-
Obscure Monitor gunHeading	Sensores	Linhas 5 e 6
	Lógica de Monitoramento	Linha 15
	Lógica de Pré-processamento	-
	Lógica de Envio de Dados	-
Obscure Monitor radarHeading	Sensores	Linhas 5 e 6
	Lógica de Monitoramento	Linha 16
	Lógica de Pré-processamento	-
	Lógica de Envio de Dados	-
Obscure Monitor Position	Sensores	Linhas 5 e 6
	Lógica de Monitoramento	Linhas 17 e 18
	Lógica de Pré-processamento	-
	Lógica de Envio de Dados	-
Opressed Monitors		
Ordem de sequência	***Loop* de execução**	**Taxa de monitoramento**
Linhas 14 a 18	Linha 11 a 31	-

4.3.2 Zanshin

O projeto Zanshin é um *framework* que possibilita o desenvolvimento de sistemas adaptativos (Tallabaci e Souza 2013). O *framework* é responsável por

monitorar os resultados da execução de métodos do sistema. A partir desse monitoramento é possível desenvolver estratégias de adaptações.

A partir da análise realizada, observou-se que o ciclo de adaptação é somente ativado quando um método monitorado é executado. Isto é, o monitoramento não é executado de forma constante e sim a partir da execução de um trecho de código-fonte. Portanto apesar de existir um subsistema gerenciador este sistema não se encaixa com as características analisadas no presente trabalho.

4.3.3 Dc4cities

O projeto Dc4cities é um sistema adaptativo para gerenciar a produção de energia elétrica. Para isso, esse sistema monitora tanto o consumo atual quanto a produção de energia elétrica. Esse sistema é capaz de adaptar o fornecimento com base no consumo reduzindo assim os gastos excessivos de recursos ou a falta de energia elétrica.

A partir da análise realizada no sistema observou-se que o interesse de monitoramento está entrelaçado com o interesse dos outros elementos do *loop* de controle na classe `OptimizationManager` sendo este um indicativo da presença do *smell Obscure Monitor*. Também foi observado que o sistema realiza o monitoramento de apenas um tipo de sensor denominado EASC (*Energy Adaptive Software Components*). Esses sensores são responsáveis por fornecer dados sobre o consumo atual de energia de uma região determinada. A classe `OptimizationManager` também é a responsável por gerenciar a execução dos monitores o que indica a existência do *smell Oppressed Monitors*.

Na Figura 4.4 é mostrada uma parte do diagrama de classes referente ao *loop* de controle. A classe `OptimizationManager` é responsável pela execução do *loop* de controle, ela possuí o atributo `monitoringInterval` que define a taxa de monitoramento. O método `startScheduledLoops()` é responsável pela chamadas dos métodos `executeMonitoringLoop()`, `executePowerLoop()` e `executeControlLoop()` cada um desses métodos está relacionado a uma etapa do *loop* de controle. O método `executeMonitoringLoop()` é responsável por realizar o monitoramento, o `executePowerLoop()` é responsável por analisar os

dados coletados e planejar as adaptações e o `executeControlLoop()` é responsável por aplicar as adaptações.

As classes `MonitoringLoop`, `PowerLoop` e `ControlLoop` estão relacionadas aos elementos do *loop* de controle: a classe `MonitoringLoop` está relacionada ao elemento Monitor; a classe `PowerLoop` está relacionada a dois elementos: o Analisador e o Planejador e; a classe `ControlLoop` está relacionada ao elemento Executor.

A classe `EascHandler` é responsável por fornecer os dados de cada componente EASC, esta classe possui um mapa do tipo *hash* contendo todos os componentes EASC. A classe `ServiceConfiguration` é utilizada para instanciar cada componente EASC.

Por meio do diagrama é possível verificar que parte do interesse de monitoramento está entrelaçada com outros interesses do *loop* de controle e que parte está separada na classe `MonitoringLoop`, dando indícios da presença do *Obscure Monitors*.

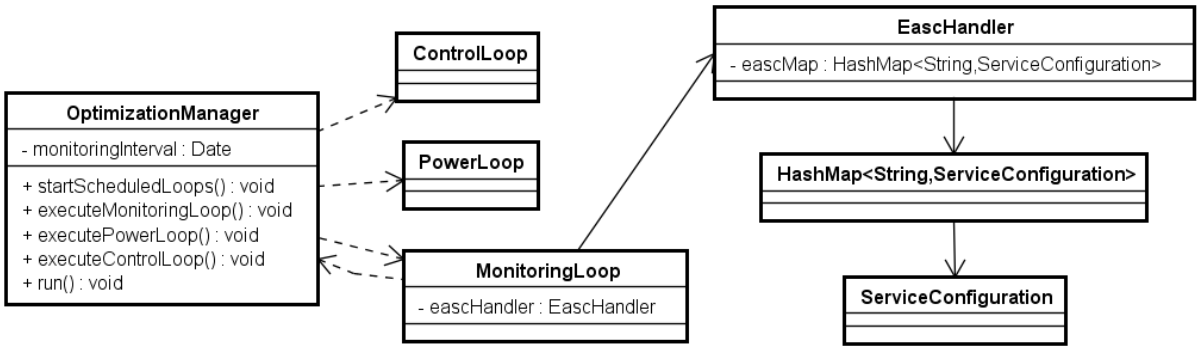


Figura 4.4 - Diagrama de classes do sistema Dc4citites (fonte: elaborada pelo autor)

Na Figura 4.5 é mostrado um esquema da execução do interesse de monitoramento pela classe `OptimizationManager`. Nessa classe o método `startScheduledLoops()` é responsável por executar o monitoramento conforme o período determinado pelo atributo `monitoringInterval`. Na execução do monitoramento, cria-se um objeto `LoopExecutor` do tipo *Runnable*, nesse objeto o método `executeMonitoringLoop()` cria o objeto do tipo `MonitoringLoop`. O objeto do tipo `MonitoringLoop` é então responsável por coletar os dados dos

componentes EASC e definir o novo valor do atributo `monitoringInterval` que irá determinar a nova execução do *loop*.

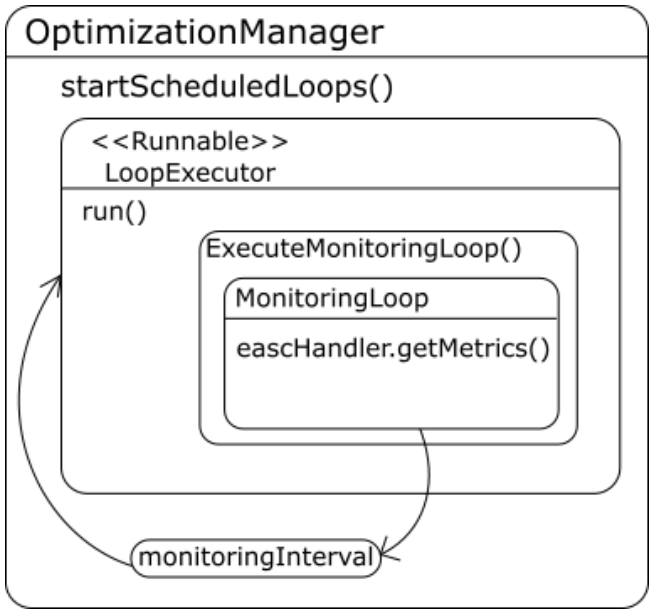


Figura 4.5 - Execução do monitoramento do sistema Dc4cities (fonte: elaborada pelo autor)

Na Figura 4.6 é mostrado um trecho da classe `OptimizationManager` relacionado ao interesse de monitoramento. O método `startScheduleLoops()` na linha 3 a 9 inicia a execução do ciclo de monitoramento criando o objeto `LoopExecutor`. A classe `LoopExecutor` estende *Runnable*, linha 11.

Na Tabela 4.4 estão descritas as principais evidências da presença dos *smells*. A tabela está dividida em duas partes, na primeira contém as evidências relacionadas à presença do *Obscure Monitor*, mostrando as partes de código referentes a quatro monitores obscuros. A segunda parte da tabela possui as evidencias da presença do *Oppressed Monitors*, mostrando trechos do código-fonte que representam o gerenciador de monitores.

```
1  public abstract class OptimizationManager {
2
3     public void startScheduledLoops() {
4           int monitoringInterval =
5                 technicalConfiguration.getMonitoringLoopInterval();
6           LoopExecutor loopExecutor = new LoopExecutor(startAt);
7           taskScheduler.scheduleAtFixedRate(loopExecutor,
8                 startAt.toDate(), monitoringInterval * 60000);
9     }
10
11    private class LoopExecutor implements Runnable {
12
13          private int monitoringLoopInterval =
14                technicalConfiguration.getMonitoringLoopInterval();
15
16          public void run() {
17                DateTime dateNow = nextExecutionDate;
18                nextExecutionDate =
19          nextExecutionDate.plusMinutes(monitoringLoopInterval);
20                executeMonitoringLoop(dateNow);
21          }
22
23          private void executeMonitoringLoop(DateTime dateNow) {
24                MonitoringLoop monitoringLoop = newMonitoringLoop();
25                List<EascMetrics> eascMetrics =
26                monitoringLoop.execute(dateNow);
27          }
28    }
29 }
```

Taxa de monitoramento

Ciclo de monitoramento

Coleta dos dados

Figura 4.6 - Trecho da Classe OptimizationManager do sistema dc4cities (fonte: elaborada pelo autor)

Tabela 4.4 - Evidência da presença dos *smells* em Dc4cities

Dc4cities		
Obscure Monitors		
	Lógica do monitor	**Local no Sistema**
Obscure Monitor Heading	Sensores	Linhas 5 e 6
	Lógica de Monitoramento	Linha 14
	Lógica de Pré-processamento	-
	Lógica de Envio de Dados	-
Opressed Monitors		
Ordem de sequência	***Loop* de execução**	**Taxa de monitoramento**
Linhas 14 a 18	Linha 11 a 31	-

4.3.4 Self-protect-project

O projeto Self-protect-project é um aplicativo desenvolvido para a plataforma Android que realiza a criptografia de dados dependendo do contexto em que o celular é executado. Para isso, esse projeto monitora os dados referentes à posição geográfica, rede de dados, configuração do dispositivo e data, possuindo então quatro monitores: o Monitor de Rede de dados, Monitor de dispositivo, Monitor de GPS e Monitor de data.

Nesse projeto há quatro monitores que se encontram parcialmente modularizados em classes diferentes, a parte do monitor relacionada a instanciar os sensores e a lógica de pré-processamento estão modularizadas em classes diferentes para cada monitor, contudo a lógica de monitoramento e o envio dos dados estão todos localizados em uma única classe. Portanto, a execução dos monitores é realizada em uma única classe que submete os monitores a uma mesma taxa de monitoramento e com uma ordem de execução fixa o que caracteriza a presença do *smell Oppressed Monitors*.

Na Figura 4.7 é mostrado o diagrama de classes relacionado ao interesse de monitoramento. Nesse diagrama é possível observar as classes `ContextNetwork`, `ContextDevice`, `ContextGPS` e `ContextDate` que modularizam cada monitor. No diagrama também é possível verificar que a classe `ActivityFileEncode` se relaciona com as classes dos quatro monitores e a partir de uma análise mais aprofundada do código-fonte desta classe é possível verificar que ela é responsável por gerenciar a execução dos monitores determinando a presença do *smell Oppressed Monitors*.

Na Figura 4.8 é mostrada uma representação esquemática da execução da classe `ActivityFileEncode`. Nessa classe é criado o objeto `UpdateThread` do tipo *Runnable*, como pode ser visto na figura, este objeto possui um ciclo que representa a execução do *loop* de monitoramento. Nesse *loop* realiza-se a chamada ao método `actionThread()` responsável por executar o método `updateAffichange()`. O método `updateAffichange()` é responsável por coletar os dados dos sensores Calendar, GPS, Network e Device nesta ordem. Após a coleta dos dados o *loop* é pausado por um segundo (1000ms) e em seguida o *loop* é iniciado novamente.

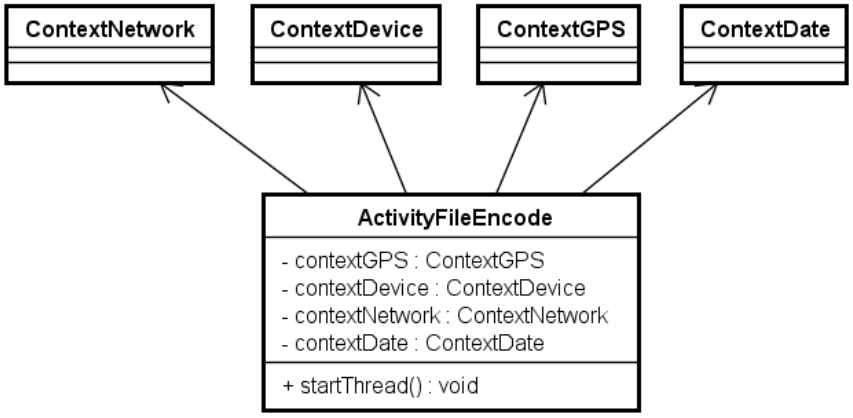


Figura 4.7 - Diagrama de classes do sistema Self-protect-project (fonte: elaborada pelo autor)

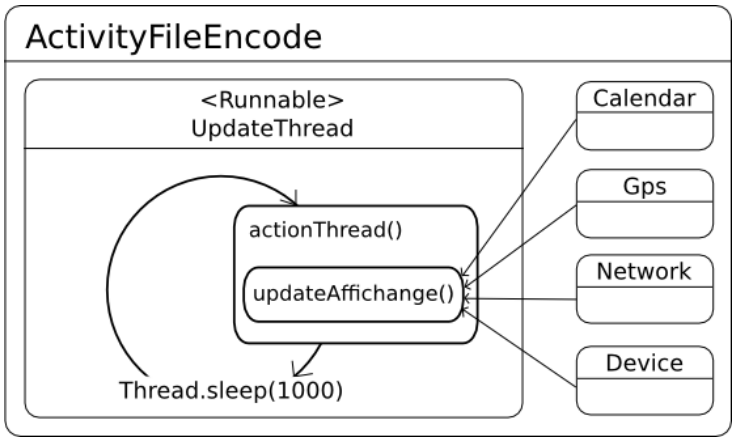


Figura 4.8 - Execução do monitoramento do sistema Self-protect-project (fonte: elaborada pelo autor)

Na Figura 4.9 é mostrado um trecho da classe `ActivityFileEncode`. Nas linhas 3 a 6 estão os atributos que irão receber os objetos das classes responsáveis por fornecer os dados do contexto. No método `onCreate()` da linha 9 a 16, que é executado na criação da classe `ActivityFileEncode()`, são criados os objetos responsáveis pelo fornecimento dos dados. O método `startThread()` da linha 29 a 33 é responsável por criar a *thread* `UpdateThread` responsável por realizar o monitoramento. Nessa *thread* está contido um *loop* que é executado a cada um segundo (1000ms), linha 39. Este *loop* é responsável por atualizar os dados por meio do método `updateAffichange()` executado da linha 18 a 27.

```
1  public class ActivityFileEncode extends ActivityFile {
2     ...
3     private ContextGPS contextGPS = null;                          Monitores
4     private ContextDevice contextDevice = null;
5     private ContextNetwork contextNetwork = null;
6     private ContextDate contextDate = null;
7     private Thread threadUpdate = null;
8
9     protected void onCreate(Bundle savedInstanceState) {
10          ...
11          contextGPS = new ContextGPS(this);
12          contextNetwork = new ContextNetwork(this);
13          contextDevice = new ContextDevice();
14          contextDate = new ContextDate();
15          ...
16    }                                                    Coleta dos dados
17
18    public void updateAffichage(){
19          viewTime.setText(contextDate.getTime());
20          viewDate.setText(contextDate.getDate());
21          viewLatitude.setText(contextGPS.getLatitude(0));
22          viewLongitude.setText(.getLongitude(0));
23          viewWifi.setText(contextNetwork.getCurrentNetwork());
24          viewManufacturer.setText(contextDevice.getManufact());
25          viewModel.setText(contextDevice.getBuild());
26          ...
27    }
28    ...
29     public void startThread(){
30          Runnable runnableThread = new UpdateThread();
31          threadUpdate= new Thread(runnableThread);
32          threadUpdate.start();
33    }
34
35    class UpdateThread implements Runnable{
36        public void run() {
37            while(!Thread.currentThread().isInterrupted()){
38                actionThread();
39                Thread.sleep(1000); // Pause of 1 Second
40          ...
41    }                                                  Ciclo de monitoramento
42
43    public void actionThread() {
44        runOnUiThread(new Runnable() {
45            public void run() {
46                updateAffichage();
47        });
48    }
49    ...
50 }
```

Figura 4.9 - Trecho da Casse MainActivity do sisema Self-protect (fonte: elaborada pelo autor)

Na Tabela 4.5 estão descritas as principais evidências da presença dos *smells*, essa tabela está dividida em duas partes na primeira contém as evidências relacionadas à presença do *smell Obscure Monitor*, mostrando as partes de código

referentes a quatro monitores. A segunda parte da tabela possui as evidências da presença do *Oppressed Monitors*, mostrando trechos do código-fonte que representam o gerenciador de monitores.

Tabela 4.5 - Evidência da presença de *smells* no sistema Self-protect

Self-protect-project		
Obscure Monitors		
	Lógica do monitor	**Local no Sistema**
Obscure Monitor GPS	Sensores	ContextGPS
	Lógica de Monitoramento	Linha 21 e 22
	Lógica de Pré-processamento	ContextGPS
	Lógica de Envio de Dados	Linha 21 e 22
Obscure Monitor Network	Sensores	ContextNetwork
	Lógica de Monitoramento	Linha 23
	Lógica de Pré-processamento	ContextNetwork
	Lógica de Envio de Dados	Linha 23
Obscure Monitor Device	Sensores	ContextDevice
	Lógica de Monitoramento	Linha 24 e 25
	Lógica de Pré-processamento	ContextDevice
	Lógica de Envio de Dados	Linha 24 e 25
Obscure Monitor Date	Sensores	ContextDate
	Lógica de Monitoramento	Linhas 19 e 20
	Lógica de Pré-processamento	ContextDate
	Lógica de Envio de Dados	Linhas 19 e 20
Opressed Monitors		
Ordem de sequência	***Loop* de execução**	**Taxa de monitoramento**
Linhas 19 a 25	Linha 37 a 39	Linha 39

4.4 Projetos Selecionados de Artigos

4.4.1 Novi

O projeto Novi (Amarasekara *et al.* 2014) é um sistema desenvolvido para a plataforma Android que objetiva melhorar a experiência do usuário realizando

adaptações no dispositivo a partir de configurações previamente realizadas pelo usuário. Para isso, o dispositivo monitora o recebimento de chamadas, o sensor de proximidade, o sensor de aceleração e o sensor de gravidade. Possuindo então quatro monitores: o Monitor de ligações, Monitor de proximidade, Monitor de Aceleração e Monitor de gravidade.

A partir da análise realizada no sistema pode-se verificar que os interesses monitorados estão parcialmente modularizados em classes individuais. Essas classes são responsáveis por instanciar os sensores e realizar parte do pré-processamento. Verificou-se também a presença de uma classe responsável por gerenciar a execução dos monitores, o que indica a presença do *smell Oppressed Monitors*.

Na Figura 4.10 é mostrada uma parte do diagrama de classes relacionada ao interesse de monitoramento. Por meio do diagrama é possível verificar que existe uma classe responsável por cada interesse monitorado, neste caso as classes `CallHelper`, `ProximityHelper`, `AccelerometerHelper` e `GravityHelper`. Outra informação relevante é a existência da classe `SensorService` que estende *Service.* Portanto, esta classe é executada em paralelo à execução do sistema principal, além disso, essa classe se relaciona com as classes dos interesses monitorados, sendo um indicativo da presença do *smell Oppressed Monitors*.

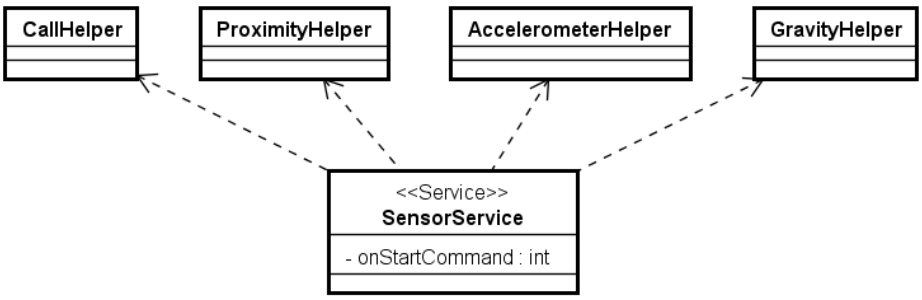


Figura 4.10 - Diagrama de classes do sistema Novi (fonte: elaborada pelo autor)

Na Figura 4.11 é mostrada uma representação esquemática do processo de execução do método `onStartCommand()` da classe `SensorService`. Nessa figura as caixas menores representam os monitores utilizados na execução do método. Esse método é executado constantemente enquanto o serviço é requisitado, sendo responsável por executar os monitores na seguinte sequência:

monitoramento de chamada (`CallHelper`), monitoramento de proximidade (`ProximityHelper`), monitoramento de aceleração (`AccelerometerHelper`) e monitoramento da gravidade (`GravityHelper`).

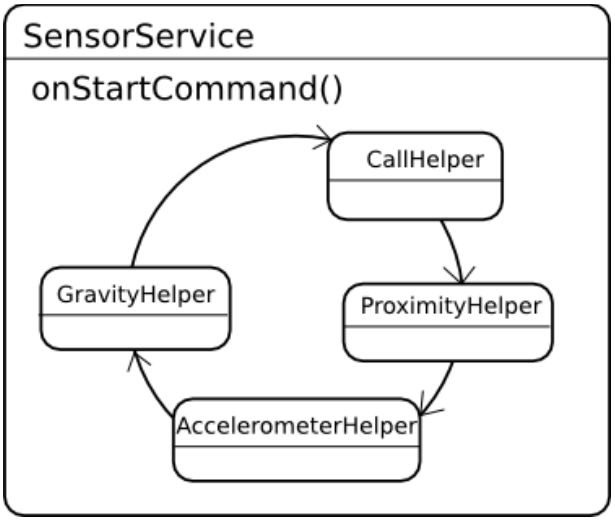


Figura 4.11 - Execução do monitoramento do sistema Novi (fonte: elaborada pelo autor)

Na Figura 4.12 são mostrados trechos da classe `SensorService` que estende *Service*. Essa classe é responsável por monitorar e armazenar os dados do contexto em paralelo a execução da aplicação. A importação dos quatro monitores é mostrada na linha 2 até a 6. O monitoramento de chamadas é realizado na linha 11 e 12, o monitoramento da proximidade na linha 14 até 18, o monitoramento do acelerômetro da linha 21 até a 24 e o monitoramento da gravidade da linha 25 até a 31. Estes trechos referem-se a execução dos monitores e o envio dos dados, as demais lógicas de pré-processamento e o carregamento dos monitores estão localizados em suas respectivas classes. Esta estrutura do código determina a sequência em que o monitoramento é executado.

Na

Tabela 4.6 estão descritas as principais evidências da presença dos *smells*, esta tabela está dividida em duas partes na primeira contém as evidencias relacionadas à presença do *smell Obscure Monitor*, mostrando as partes de código referentes a quatro monitores obscuros. A segunda parte da tabela possui as evidencias da presença do *smell Oppressed Monitors*, mostrando trechos do código-fonte que representam o gerenciador de monitores.

```
1  ...
2  import com.integra.novi.util.AccelerometerHelper;
3  import com.integra.novi.util.CallHelper;
4  import com.integra.novi.util.GravityHelper;
5  import com.integra.novi.util.ProximityHelper;
6
7  public class SensorService extends Service {
8     ...
9    public int onStartCommand(Intent intent, int flags, int startId) {
10   ...
11   IHelper ch = new CallHelper(this);                    Monitor de chamadas
12   ch.register();
13
14   if (pref.getBoolean(SettingsProvider.
15               SETTINGS_MOTION_PROXIMITY_TAP, false)){
16     ProximityHelper ph = new ProximityHelper(this);
17     ph.register();                          Monitor de proximidade
18   }
19
20   AccelerometerHelper ac = new AccelerometerHelper(this);
21   ac.setShuffleEnabled(pref.getBoolean(
22   SettingsProvider.SETTINGS_MOTION_SHUFFLE, false));
23   ac.register();                          Monitor de aceleração
24
25   if (pref.getBoolean(SettingsProvider.
26               SETTINGS_MOTION_FACEDOWN, false) ||
27      pref.getBoolean(SettingsProvider.
28               SETTINGS_MOTION_FACEUP,false)) {
29     GravityHelper gh = new GravityHelper(this);
30     gh.register();
31   }                                       Monitor de gravidade
32
33   return Service.START_REDELIVER_INTENT;
34   }
35 }
```

Figura 4.12 - Trecho da Classe SensorService do sistema Novi (fonte: elaborada pelo autor)

4.4.2 SmarTrAC

SmarTrAC é uma aplicação desenvolvida para dispositivos móveis responsável por coletar dados de sensores, determinar a atividade sendo realizada e adaptar-se conforme regras pré-definidas (Fan *et al.* 2015). Os dados são monitorados a cada trinta segundos, sendo eles a latitude, longitude, velocidade, precisão e acelerômetro.

Tabela 4.6 - Evidência da presença dos *smells* no sistema Novi

Novi		
Obscure Monitors		
	Lógica do monitor	**Local no Sistema**
Obscure Monitor Call	Sensores	`ContextGPS`
	Lógica de Monitoramento	Linha 11 e 12
	Lógica de Pré-processamento	`ContextGPS`
	Lógica de Envio de Dados	Linha 12
Obscure Monitor Proximity	Sensores	`ContextNetwork`
	Lógica de Monitoramento	Linha 14 a 18
	Lógica de Pré-processamento	`ContextNetwork`
	Lógica de Envio de Dados	Linha 17
Obscure Monitor Accelerometer	Sensores	`ContextDevice`
	Lógica de Monitoramento	Linha 20 a 23
	Lógica de Pré-processamento	`ContextDevice`
	Lógica de Envio de Dados	Linha 23
Obscure Monitor Gravity	Sensores	`ContextDate`
	Lógica de Monitoramento	Linhas 25 a 31
	Lógica de Pré-processamento	`ContextDate`
	Lógica de Envio de Dados	Linha 30
Opressed Monitors		
Ordem de sequência	***Loop* de execução**	**Taxa de monitoramento**
Linhas 14 a 18	Linha 9 a 35	-

Na Figura 4.13 é mostrada uma representação da arquitetura do sistema que possui quatro principais componentes, o *Sensor Data Capture*, *Sensor Data Processor*, *Main Database* e *User Interface*. O componente *Sensor Data Capture* é responsável por monitorar os dados e aplicar filtros de transformações. A coleta dos dados é realizada pelo *Sensor listener* que coleta os dados dos sensores embutidos do dispositivo móvel a cada trinta segundos, armazenando-os no *Main Database*.

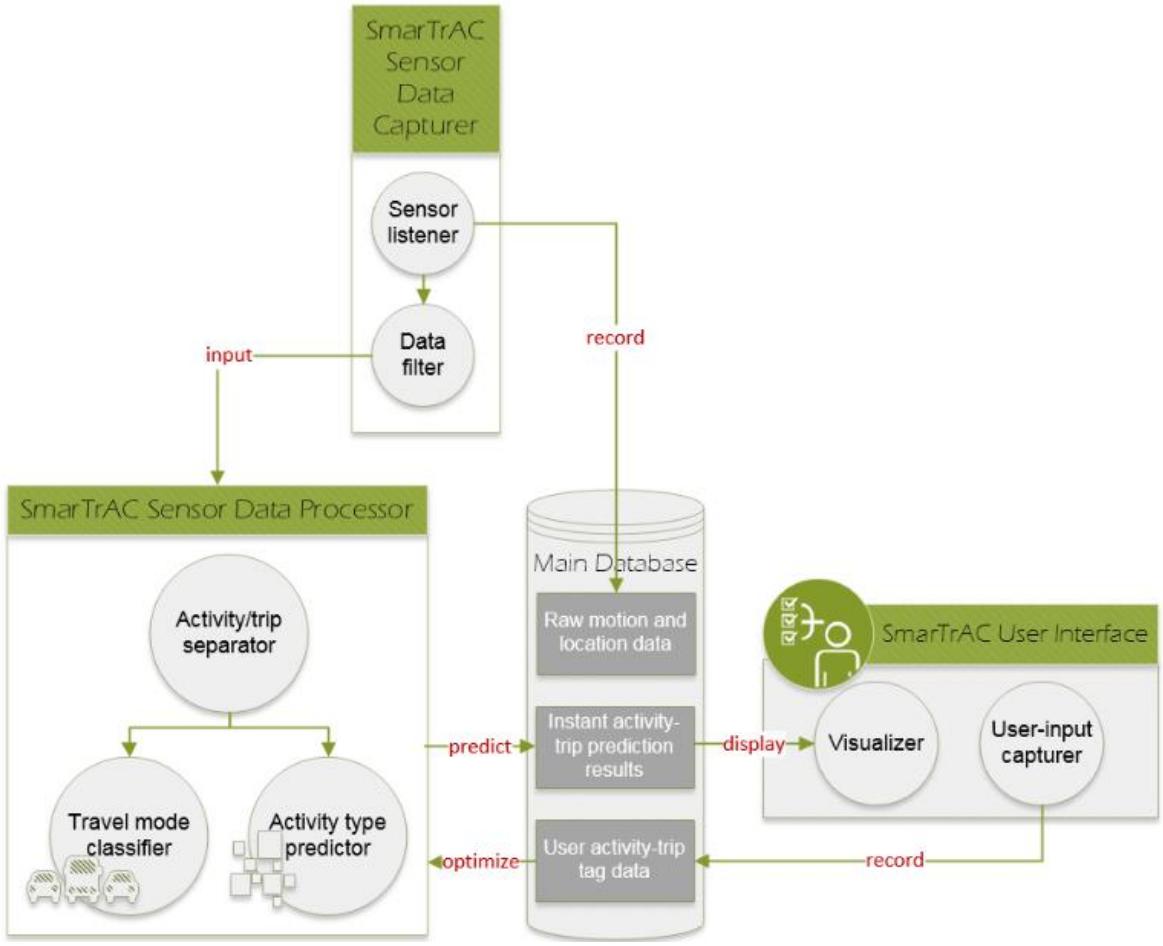


Figura 4.13 - Arquitetura do sistema SmarTrAC (Fan *et al.* 2015)

A partir da análise realizada no trabalho, a coleta dos dados de todos os sensores é realizada pelo *Sensor Listener* o que indica a presença do *smell Obscure Monitor*. Além disso, a taxa de monitoramento é única indicando também a presença do *smell Oppressed Monitors*. Nessa análise não foi encontrado indicativo que sugere uma ordem de execução dos monitores, contudo por serem executados em uma única classe e com uma taxa de monitoramento única acredita-se que também possuem uma ordem de execução fixa.

4.4.3 Context-aware Framework

No trabalho de Biegel e Cahill (2004), os autores propõem um framework para aplicações ciente de contexto. Nesse *framework* é utilizado o objeto *sentient* para a coleta de dados do contexto. Como mostrado na Figura 4.14, esse objeto é responsável por monitorar os dados de diversos sensores e assim produzir contexto

com maior nível de abstração, possibilitando assim definir ações a serem realizadas pelos atuadores.

O objeto *sentient* é responsável realizar o monitoramento de todos os sensores indicando a presença do *smell Obscure Monitor*, pois apenas um elemento é responsável por monitorar diversos tipos de dados. Este objeto também nos indica a presença do *smell Oppressed Monitor*, pois segundo os autores, o objeto.

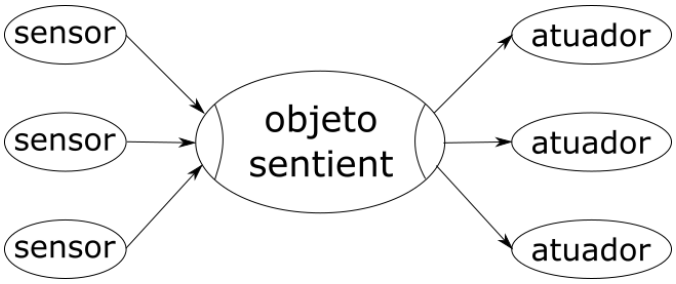


Figura 4.14 - Modelo do objeto sentinela adaptado de Biegel e Cahill (2004)

4.4.4 Behavior-Based Program

O estilo de programação Behavior-based é muito utilizado na programação de sistemas robóticos, possibilitando certo grau de autogerenciamento (Arkin 1998). Esse estilo de programação é baseado em comportamentos reativos, isto é, a partir de leituras de sensores o sistema determina um comportamento a ser executado.

Na Figura 4.15 é mostrada a arquitetura utilizada no estilo de programação *behavior-based*. O retângulo *Sensores* representa os sensores responsáveis por fornecer dados do ambiente para cada *Behavior*. Os *Behaviors* são responsáveis por modularizar a execução de cada comportamento realizado pelo sistema. É importante salientar que apesar de não estar demonstrado na figura, cada *Behavior* pode possuir um ou mais sensores. O elemento *Coordenador* que engloba todos os *Behaviors* é responsável por gerenciar a execução dos *Behaviors*. Os Atuadores são os elementos que irão executar o comportamento determinado por cada *Behavior*.

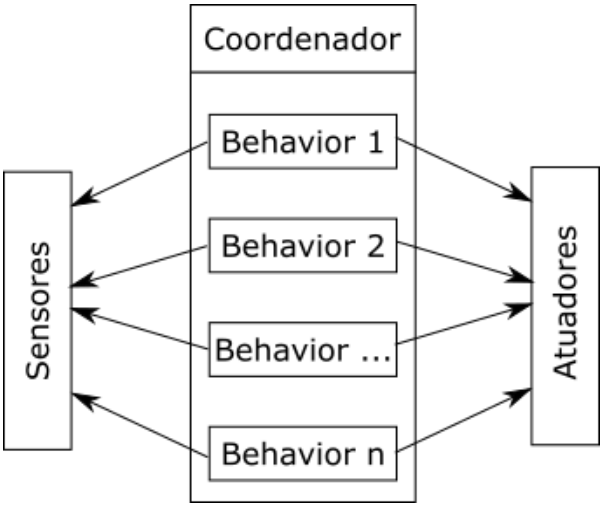


Figura 4.15 - Arquitetura Behavior-based programming adaptado de (Arkin 1998)

A partir da análise realizada, o *smell Obscure Monitor* está presente neste estilo de programação, pois cada *behavior* é responsável por monitorar os dados dos sensores necessários a sua execução. Dessa forma é comum que dois ou mais *behaviors* necessitem de dados de um mesmo sensor, o que leva a ocorrência da presença de código-duplicado, o que não ocorreria com o interesse monitorado modularizado. A presença do *smell Oppressed Monitors* ocorre em decorrência do monitoramento ser ordenado em uma fila de execução no elemento *Coordenador*. Além disso, os monitoramentos não possuem uma taxa de monitoramento individual evidente.

4.5 Considerações Finais

Neste capítulo foi apresentado o levantamento de sistemas realizado para a análise da implementação do interesse de monitoramento em sistemas adaptativos. Para todos os sistemas selecionados foi incluída uma descrição e análise das principais características relacionadas ao monitoramento em cada sistema.

Capítulo 5

AVALIAÇÃO PRELIMINAR DA MANUTENABILIDADE EM SISTEMAS ADAPTATIVOS

5.1 Considerações Iniciais

Neste capítulo apresentam-se alguns cenários de manutenção com o objetivo de analisar impactos causados pela presença dos *smells*. Para realizar essa análise, na seção Refatoração do *PhoneAdapter* é apresentada a refatoração aplicada para remover os *smells* obtendo, assim, uma versão refatorada sem *smells*. Na seção Tarefas de Manutenção são apresentadas as manutenções aplicadas nas duas versões do sistema *PhoneAdapter*; a versão original e a refatorada. Quanto à versão refatorada, sabe-se que a refatoração de smells arquiteturais pode ser realizada de diversas maneiras, assim, optou-se por uma alternativa simples e direta para a remoção dos *smells* com base em alguns trabalhos (Ramirez e Cheng 2010, Gil de La Iglesia 2014, Abuseta e Swesi 2015). É importante salientar que não é o objetivo deste capítulo propor refatorações ou padrões de projetos para solucionar os *smells*.

O capítulo está organizado em quatro seções. Na Seção 5.1 são realizadas as considerações iniciais. Na Seção 5.2 é descrito o processo de refatoração do *PhoneAdapter*. Na Seção 5.3 são apresentadas as tarefas de manutenção. Por fim, na Seção 5.4 são mostradas as considerações finais do capítulo.

5.2 Refatoração do PhoneAdapter

Nesta seção é demonstrada uma possível alternativa de refatoração que foi aplicada no *PhoneAdapter* com o intuito de remover os *smells Obscure Monitor* e *Oppressed Monitors*. É importante salientar que não é objetivo desta dissertação propor refatorações específicas para os *smells* apresentados, visto que essa atividade pode ser realizada de diversas maneiras dependendo grandemente das características do sistema e do contexto atual.

A estratégia de refatoração que foi adotada possui dois objetivos principais:

i. Tornar o elemento monitor evidente no código-fonte modularizando-o em uma entidade de primeira classe; removendo assim o *smell Obscure Monitor*.
ii. Tornar os monitores independentes com relação à ordem de execução e taxa de monitoramento.

Na Figura 5.1 são mostrados os diagramas de classes da versão refatorada e da versão original, que contém os *smells*. A versão original é mostrada na parte superior e a versão refatorada na parte inferior da figura. Optou-se por mostrar novamente o diagrama de classes da versão original para facilitar a comparação, sem que seja necessário deslocamento até a figura da página 36.

Na versão refatorada a classe `ContextManager` foi removida e em seu lugar foram criadas quatro novas threads representando cada um dos monitores (`MonitorWeekday`, `MonitorGPS`, `MonitorTime`, `MonitorBluetooth`) existentes no *PhoneAdapter*. Os métodos e atributos que estavam na classe `ContextManager` foram separados e colocados em suas respectivas classes. No entanto, o método `sendBroadcast()` foi replicado em todas as classes sendo ele o responsável por enviar os dados monitorados para a classe `AdaptationManager`. Ao realizar este processo de tornar os Monitores evidentes o *smell Obscure Monitor* foi resolvido.

A solução para o *smell Oppressed Monitors* precisa garantir que os monitores possam ser executados independentemente e que possam ter suas próprias taxas de monitoramento. Para que seja possível garantir o primeiro ponto, uma premissa é que cada classe que representa um monitor seja uma thread separada. Além disso,

para permitir que cada monitor exponha sua taxa de monitoramento, deve-se criar uma variável dedicada a isso em cada um deles.

a) Original

b) Refatorada

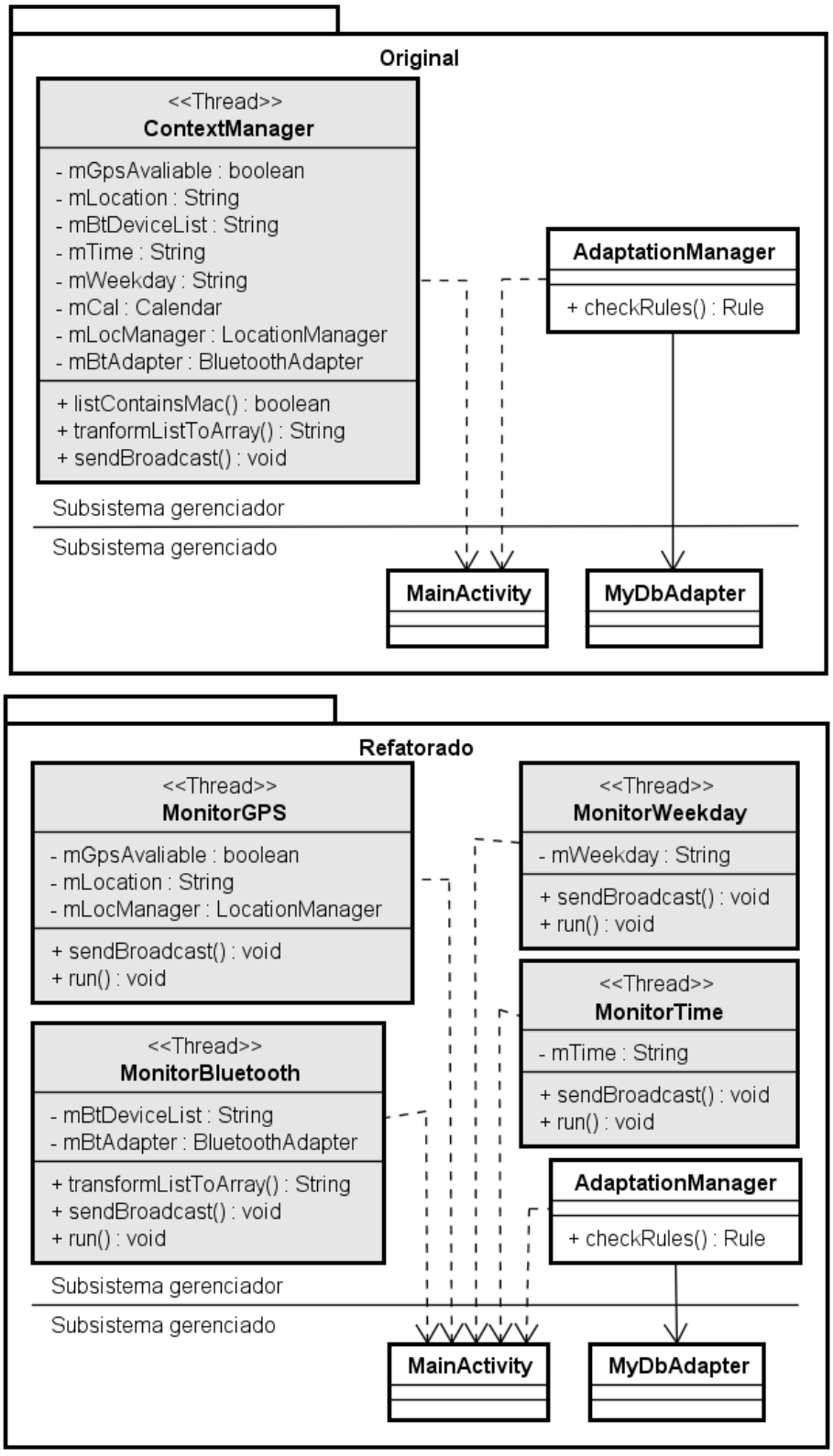


Figura 5.1 - Diagrama de classes do PhoneAdapter a) versão original, b) versão refatorada (fonte: elaborada pelo autor)

Para facilitar o processo de refatoração demonstrado nesta seção foi criado um *template*, que pode ser visto na Figura 5.2. Esse *template* foi inspirado nas principais características do elemento monitor encontradas nos trabalhos de Ramirez e Cheng (2010), Gil de La Iglesia (2014) e Abuseta e Swesi (2015), descritas a seguir:

1) **Um monitor deve ser executado como um processo paralelo ao subsistema gerenciado.** Por exemplo, na programação com linguagem JAVA, esta classe deve implementar a interface *Runnable* ou estender a classe `Thread`. Já para a plataforma Android o monitor deve ser executado estender a classe *Service* ou *IntentService*.

2) **Um monitor deve instanciar os sensores antes de iniciar o processo de coleta de dados.** Portanto, um local adequado para serem instanciados é em um método construtor ou que seja chamado na criação da classe.

3) **Um monitor deve possuir um *loop* que tenha uma taxa de monitoramento.** Essa estrutura e a taxa de monitoramento determinam a execução da coleta de dados constante.

```
1  //Classe de execução paralela ao sistema
2  public class monitorName extends [IntentService/Thread/...]{
1
2  int pollingRate;
3
4  //Método de inicialização da classe
5  public [construtor/onCreate](int pollingRate) {
6     this.pollingRate = pollingRate;
7     //monitoring logic (initiate the sensor)
8  }
9
10 //Métodos de execução da classe
11 protected void [run/start/onHandleIntent/...](){
12    while(appOn) {
13       //monitoring, preprocessing, broadcasting logic
14       //polling rate
15    ...
16 }
17 }
```

Figura 5.2 - Template da classe Monitor (fonte: elaborada pelo autor)

Na Figura 5.3 é mostrada a classe `WeekdayMonitor` do *PhoneAdapter*, que é a versão refatorada. Este trecho de código está relacionado à caixa pontilhada número 2 (dois) na Figura 3.2, página 38.

Para aplicar esta refatoração o primeiro passo foi identificar o monitor de dia da semana, para isso foram seguidos os passos descritos na identificação do elemento monitor da Subseção 3.3.1. Identificando-se assim o sensor responsável por prover o dia da semana, neste caso o sensor é representado pela API do JAVA `java.util.Calendar`. A partir da identificação da API, foi localizado então o trecho de código relacionado à implementação deste sensor, que no caso é realizado por meio do atributo `mCal` (linha 21, Figura 3.2) localizado no método `onCreate()`.

Após identificar o atributo que instancia o sensor é então localizado o código-fonte relacionado à lógica de monitoramento, pré-processamento e envio de dados. Para isso é necessário verificar todo código relacionado ao atributo `mCal`. Algumas evidências podem auxiliar neste processo, pois esta lógica deve estar inserida em uma estrutura de *loop*.

Neste caso, a lógica de monitoramento é realizada pelo código `mCal.get(Calendar.DAY_OF_WEEK)` na linha 31, Figura 3.2. A lógica de pré-processamento é realizada pela estrutura de código *switch/case* da linha 31 até 39, Figura 3.2. Por fim, a lógica de envido de dados está localizada dentro do objeto *runnable* `mHandler` na linha 44, Figura 3.2. Após identificar todas as partes deste elemento monitor, o seu código foi movido para o seu respectivo local no template criado anteriormente.

Este processo foi executado para cada um dos monitores, removendo todo o código da classe `ContextManager`. É importante ressaltar que ao realizar esta refatoração outras adequações no código-fonte do sistema devem ser realizadas para permitir o correto funcionamento da aplicação, como por exemplo, o novo monitor criado deve ser instanciado na classe principal da aplicação, mesma classe na qual era instanciado a classe `ContextManager`.

5.3 Tarefas de Manutenção

Nesta seção é apresentado um estudo cujo objetivo foi averiguar o impacto que a presença dos *smells* causa durante determinadas atividades de manutenção e evolução de um sistema. Para isso, foram idealizadas as tarefas de manutenção listadas na Tabela 5.1. Esse conjunto de tarefas procurou cobrir a maior parte das possibilidades de manutenção que podem ocorrer no interesse de monitoramento. A tabela contém quatro tarefas de manutenção e uma breve descrição de cada uma, sendo que na primeira coluna estão listados os nomes de cada tarefa e na segunda coluna contém a descrição. Essas tarefas foram aplicadas na versão original do *PhoneAdapter* e em uma versão refatorada como descrita na seção anterior.

```
1  import java.util.Calendar;
2  ...
3  public class MonitorWeekday extends IntentService {
4  //attributes
5      Calendar mCal;
6      int pollingRate;
7      ...
8      public MonitorWeekday(int pollingRate) {
9           this.pollingRate = pollingRate;
10         //monitoring logic (initiate the sensor)
11         mCal=Calendar.getInstance();
12    }
13
14     protected void onHandleIntent(Intent intent) {
15         while(appOn) {
16             //monitoring logic
17             weekday=mCal.get(Calendar.DAY_OF_WEEK);
18             switch(weekday) {
19                 case 1: mWeekday="sunday";
20                 break;
21                 ...     }
22             //broadcast logic
23             mHandler.post(new Runnable() {
24                 public void run() {
25                     Intent i=new Intent();
26                     i.putExtra(
27                     ContextName.WEEKDAY.mWeekday);
28                     sendBroadcast(i);   }
29             });
30             //polling rate
31             Thread.sleep(pollingRate);
32         }
33     ...
34  }
```

Figura 5.3 - Monitor de dia da semana (fonte: elaborada pelo autor)

É importante notar que todas as tarefas listadas estão relacionadas ao interesse de monitoramento, ou seja, não foram aplicadas tarefas em outros elementos do *loop* de controle como o analisador, planejador e executor. Como não houve alterações realizadas nestes elementos durante a aplicação da refatoração não foi objetivo verificar as influências das atividades de manutenção em tais elementos. Também é importante salientar que no sistema refatorado ambos os *smells* foram removidos, além disso, as tarefas de refatoração aqui propostas não possuem o intuito de exercitar um dos *smells* em específico. Portanto, no caso de melhorias no processo de manutenção do sistema refatorado não é possível definir qual dos *smells* teve maior influência. Isto é, as melhorias podem ser devido tanto à remoção do *smell Obscure Monitor* ou do *Oppressed Monitor*, podendo também ser em consequência da remoção de ambos.

Tabela 5.1 - Tarefas de manutenção

Tarefa de manutenção	Descrição
Adicionar novo monitor	Surge a necessidade de monitorar um novo sensor, necessitando assim a implementação de um novo elemento monitor a ser adicionado no interesse de monitoramento.
Alterar a lógica de pré-processamento	Uma manutenção comum é o desenvolvimento de novos algoritmos de pré-processamento para que os dados coletados caracterizem de forma mais apropriada o contexto, necessitando da alteração da lógica de pré-processamento dos monitores.
Definir diferentes taxas de monitoramento	Para um uso melhor dos recursos cada monitor deve possuir uma taxa de monitoramento adequada ao seu tipo de dado monitorado.
Definir ordem de execução entre monitores	Os monitores não devem agir de forma independente, possuindo assim uma ordem de execução que deve ser determinada.

Nas subseções seguintes, as tarefas listadas na Tabela 5.1 serão descritas em detalhes com um exemplo prático, são apresentadas também as alterações realizadas em ambos os sistemas: original e refatorado, com uma avaliação descritiva dos impactos notados durante o processo de evolução. Por fim, é

realizada uma discussão e conclusão a respeito de qual sistema a manutenção foi menos invasiva, isto é, que menos altera código já existente.

5.3.1 Adicionar novo monitor

Nesta tarefa de manutenção o sistema deve começar a realizar o monitoramento do nível de carga da bateria. Com o monitoramento do nível da bateria é possível implementar diversas novas adaptações para otimizar o uso dos recursos do sistema. Como por exemplo, em um dispositivo móvel, com nível de bateria baixo a intensidade de brilho da tela pode ser reduzida ou periféricos podem ser desativados como a rede *wireless*.

Na Figura 5.4 é mostrado o código-fonte necessário para a implementação do monitor de bateria no PhoneAdapter. Esse código deve ser adicionado tanto na versão original quanto na versão refatorada. Essa figura está dividida em quatro partes, na primeira é mostrado o código-fonte para criar o sensor de bateria, na segunda parte a lógica de monitoramento na qual são coletados dois dados do monitor de bateria, na terceira parte o pré-processamento do dado no qual converte os dados monitorados para o valor da porcentagem da bateria atual. Por fim na última parte a lógica de envio dos dados que é realizada da mesma por meio de um objeto para *broadcast*.

É importante ressaltar que as modificações aqui demonstradas dizem respeito somente às alterações relacionadas ao monitoramento. Ou seja, para que as alterações sejam realmente efetivas é necessário realizar alterações na classe `AdaptationManager` e `MyDbAdapter` de forma que estas aceitem os novos dados monitorados. No entanto, essas alterações não serão abordadas por não fazerem parte do escopo deste presente trabalho.

Código-fonte para instanciar um sensor

```
Intent batteryStatus = context.registerReceiver(
                       null, new IntentFilter(
                       Intent.ACTION_BATTERY_CHANGED));
```

Exemplo de código-fonte para captura de dados

```
int level = batteryStatus.
            getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
int scale = batteryStatus.
        getIntExtra(BatteryManager.EXTRA_SCALE, -1);
```

Exemplo de código-fonte para pré-processamento dos dados

```
float batteryPct = level / (float)scale;
```

Código para envio dos dados

```
i.putExtra(ContextName.Battery, batteryPct);
```

Figura 5.4 - Trechos relacionados ao monitor de bateria no PhoneAdapter (fonte: elaborada pelo autor)

Alterações no sistema original:

No sistema original, para seguir o padrão de implementação, todo o código-fonte relacionado com o novo monitor foi adicionado à classe `ContextManager`. O código relacionado à criação do sensor de bateria foi inserido no método `onCreate()`. Os trechos relacionados à lógica de monitoramento e pré-processamento foram inseridos dentro do *loop* `while` junto com a execução dos demais monitores, a localização do código irá definir a ordem em que este monitor será executado, ficando a critério do desenvolvedor. Neste caso optou-se por executar o Monitor de Bateria em primeiro. Por fim para o envio dos dados monitorados foi adicionado mais um atributo ao container responsável por enviar os dados. Essas modificações podem ser vistas na Figura 5.5 que mostra partes da classe `ContextManager`. Os retângulos nessa figura evidenciam os trechos de código-fonte relacionados ao monitor de bateria.

Alterações no sistema refatorado:

No sistema refatorado, a adição do novo monitor requer que uma nova classe de monitoramento seja criada, seguindo o *template* do monitor mostrado na Figura 5.2, os trechos de código devem ser adicionados conforme as orientações descritas no *template*, por fim é obtida a classe `BatteryMonitor` como a representada na

Figura 5.6. Após o desenvolvimento da classe, essa deve ser instanciada na classe principal da aplicação, `MainActivity`, juntamente com a instanciação dos demais monitores.

Análise das Alterações:

A inserção de um novo monitor é simples de ser realizada em ambos os sistemas, a principal diferença é o quanto cada alteração é invasiva ao código-fonte já existente. No caso do sistema original, a inserção é bastante invasiva, pois altera vários trechos de código da classe `ContextManager`. Contudo, no sistema refatorado todo código-fonte novo foi adicionado a uma classe separada, sendo apenas necessário adicionar a criação do novo monitor na classe `MainActivity`. Portanto é possível verificar que as alterações no sistema original são mais invasivas que as realizadas no sistema refatorado, essa diferença é importante, pois facilita na identificação de possíveis problemas que possam ocorrer no período de desenvolvimento.

5.3.2 Alterar a lógica de pré-processamento

Nesta tarefa é realizada uma simples alteração relacionada ao pré-processamento do dado coletado pelo monitor do dia da semana. É requisitado ao sistema que ao monitorar o dia da semana e este dia for uma sexta-feira 13, deve-se acrescentar um asterisco na frente do dado produzido. Apesar de esta tarefa ser simples, ela serve para exercitar possíveis alterações no pré-processamento dos monitores que podem envolver a adição de fórmulas e algoritmos mais complexos. Sedo assim, com uma tarefa mais simples é possível entender e analisar melhor quais seriam as alterações necessárias.

Para realizar esta tarefa, é necessário então alterar o trecho do pré-processamento relacionado à sexta-feira, o código-fonte que deve ser adicionado no lugar do existente é mostrado na Figura 5.7. Nesse código existe uma simples condicional que verifica se o dia do mês é igual a treze, caso verdadeiro é adicionado o sinal asterisco junto ao nome da semana.

```
1  public class ContextManager extends IntentService {
2     private Intent batteryStatus;
3     private float batteryPct;
4     private boolean mGpsAvailable;
5     private String mLocation;
6     private double mSpeed;
7     ...
8
9     public void onCreate() {
10          ...
11          batteryStatus = getApplicationContext().
12                registerReceiver(null, new IntentFilter(
13                Intent.ACTION_BATTERY_CHANGED));
14          mLocManager=(LocationManager)
15                getSystemService(LOCATION_SERVICE);
16          mLocListener=new MyLocationListener();
17          ...
18    }
19
20    ...
21    protected void onHandleIntent(Intent arg0) {
22
23          while(!mStop){
24                int level = batteryStatus.getIntExtra
25                (BatteryManager.EXTRA_LEVEL, -1);
26
27                int scale = batteryStatus.getIntExtra
28                (BatteryManager.EXTRA_SCALE, -1);
29
30                batteryPct = level / (float)scale;
31
32                mCal=Calendar.getInstance();
33                mTime=mTimeFormat.format(mCal.getTime());
34
35                switch(mCal.get(Calendar.DAY_OF_WEEK)){
36                case 1:
37                      mWeekday="sunday";
38                      break;
39                ...
40
41          mHandler.post(new Runnable() {
42                public void run() {
43                Intent i=new Intent();
44                ...
45                i.putExtra(ContextName.TIME,mTime);
46                i.putExtra(ContextName.WEEKDAY, mWeekday);
47                i.putExtra(ContextName.BATTERY, batteryPct);
48                sendBroadcast(i);
49                ...
50          });
51          ...
52 }
```

Figura 5.5 - Adição de novo monitor no sistema original (fonte: elaborada pelo autor)

```
1  public class BatteryMonitor extends IntentService{
2  int level, scale, pollingRate;
3  Intent batteryStatus;
4
5  public void onCreate() {
6           batteryStatus = context.registerReceiver(
7                           null, new IntentFilter(
8                           Intent.ACTION_BATTERY_CHANGED));
9           pollingRate = 120000;
10 }
11
12 protected void onHandleIntent(Intent arg0){
13  while(appOn) {
14    level = batteryStatus.
15              getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
16    scale = batteryStatus.
17              getIntExtra(BatteryManager.EXTRA_SCALE, -1);
18    batteryPct = level / (float)scale;
19
20    mHandler.post(new Runnable() {
21                  public void run() {
22                      Intent i=new Intent();
23                      i.putExtra(
24                       ContextName.Battery,
25                       batteryPct);
26              });
27
28    Thread.sleep(pollingRate);
29  }
30 }
```

Figura 5.6 - Classe BatteryMonitor (fonte: elaborada pelo autor)

Alterações no sistema refatorado:

No sistema refatorado as alterações da lógica de pré-processamento do dia da semana estão localizadas na classe `WeekdayMonitor`. As alterações de código-fonte são as mesmas realizadas no sistema original, basta localizar o pré-processamento relacionado ao dia desejado e alterado o código-fonte adicionando o trecho representado na Figura 5.7.

```
1  Case 6:
2     if(mCal.get(Calendar.DAY_OF_MONTH) == 13){
3           mWeekday="friday*";
4     }else{
5           mWeekday="friday";
6     }
7     break;
```

Figura 5.7 - Código-fonte relacionado ao pré-processamento do dia da semana (fonte: elaborada pelo autor)

Análise das Alterações:

As alterações realizadas nos dois sistemas são similares, contudo é importante observar que na versão original do sistema as alterações são bem mais invasivas, pois o processo inclui alterar a classe que contém todos os monitores. Um eventual erro na execução da alteração pode gerar um comportamento indesejado dos outros monitores, ou até mesmo interrompendo a execução do interesse de monitoramento. Por outro lado, no sistema refatorado, como as alterações são localizadas apenas no monitor de dia da semana, caso o processo de alteração gere algum problema, este será facilmente localizado.

É importante ressaltar que apesar da alteração proposta nesta tarefa de manutenção seja simples, em sistemas maiores estas alterações podem envolver a inclusão de lógicas mais complexas, adicionar estas lógicas em uma mesma classe de monitoramento irá dificultar ao realizar novas manutenções e evoluções.

5.3.3 Definir diferentes taxas de monitoramento

Nesta tarefa é requisitado ao sistema que reduza o gasto de bateria. Para isso o Monitor de GPS deve operar em uma frequência menor que a atual. No caso a frequência do Monitor de GPS deve ser alterada para cada nove minutos (atualmente a é realizada a cada dois minutos).

Alterações no sistema original:

No sistema original devem ser realizadas duas alterações na classe `ContextManager`. A primeira alteração diz respeito à inserção de uma lógica condicional para executar o monitor de GPS somente a cada nove minutos. Ou seja, o monitor deverá verificar se o tempo decorrido foi de nove minutos e em caso verdadeiro executar o monitoramento do GPS. Um exemplo de lógica possível a ser adicionada é mostrado na Figura 5.8. Nessa figura os retângulos evidenciam a lógica que deve ser adicionada, no primeiro retângulo é inserido um contador que armazena o intervalo de tempo de cada execução do ciclo de monitoramento. O segundo e o terceiro retângulos evidenciam os condicionais que verificam se o monitoramento deve ser realizado. Por fim, o último retângulo evidencia a alteração que deve ser realizada na taxa de monitoramento.

```
1  public class ContextManager extends IntentService {
2
3     protected void onHandleIntent(Intent arg0) {
4           int elapsedTime = 0;
5           while(!mStop){
6                 elapsedTime++;
7                 if(elapsedTime % 2 == 0){
8                       mCal=Calendar.getInstance();
9                       mTime=SimpleDateFormat.
10                            getTimeInstance().
11                            format(mCal.getTime());
12                      switch(mCal.get(Calendar.DAY_OF_WEEK)){
13                      case 1:
14                            mWeekday="sunday";
15                            break;
16                      ...
17                      if(mBtAdapter != null){
18                      ...
19                }
20
21                if(elapsedTime % 9 == 0){
22                      mGpsAvailable = mLocListener.mGpsAvailable;
23                      mLocation = mLocListener.mLocation;
24                      mSpeed = mLocListener.mSpeed;
25                      elapsedTime = 0;
26                }
27                ...
28                Thread.sleep(TimeUnit.MINUTES.toMillis(1));
29                ...
30          }
31    }
32 }
```

Figura 5.8 - Definição de diferentes taxas de monitoramento (fonte: elaborada pelo autor)

A segunda alteração realizada é relacionada à taxa de monitoramento. Observe que o *loop* é executado a cada dois minutos, desta forma a condicional adicionada para a execução do Monitor de GPS nunca será satisfeita. Para que o Monitor de GPS seja executado a cada nove minutos é necessário que a taxa de monitoramento seja compatível com o tempo de execução do desejado ao Monitor de GPS. Isto é, com o *loop* executando a cada dois minutos, os monitores sempre serão executados em períodos múltiplos de dois. Logo é necessário adicionar uma taxa de monitoramento que seja compatível com a taxa de monitoramento de todos os monitores. No caso utilizou-se a frequência de monitoramento de um minuto. Para evitar que os demais monitores sejam executados a cada um minuto, um condicional semelhante ao adicionado no Monitor de GPS deve ser adicionado a todos os demais monitores.

Alterações no sistema refatorado:

No sistema refatorado as alterações para determinar a execução do monitor de GPS a cada nove minutos deve ser realizada na classe `MonitorGPS`. Nessa classe basta alterar o atributo `pollingRate` para nove minutos conforme é mostrado na Figura 5.9. Nessa figura o retângulo evidencia a linha que deve ser alterada para modificar a taxa de monitoramento, é possível observar também que o atributo `pollingRate` possui o método `setPollingRate()` o que permite a alteração dessa taxa de monitoramento em tempo de execução do sistema.

Análise das Alterações:

Nesta tarefa as alterações realizadas nos sistemas são bastante diferentes. É importante ressaltar que este tipo de tarefa é uma possível evolução do sistema adaptativo. Com isso é permitido que em determinados contextos o sistema seja capaz de alterar a frequência de execução de cada monitor em específico. Por exemplo, em um contexto no qual o nível de bateria esteja baixo, pode ser requisitado ao sistema uma adaptação para diminuir a taxa de monitoramento ou até mesmo interromper a execução de um monitor com o intuito de reduzir os gastos da bateria. Exigindo então que cada monitor possua um tempo de execução diferente.

```
1  public class GPSMonitor extends IntentService{
2
3  private int pollingRate;
4  ...
5
6  public void setPollingRate(int pollingRate){
7     this.pollingRate = pollingRate;
8  }
9
10 public int getPollingRate(){
11    return this.pollingRate
12 }
13
14 public void onCreate() {
15    ...
16    pollingRate = (int) TimeUnit.MINUTES.toMilli(9);
17 }
18 ...
19 }
```

Figura 5.9 - Alteração da taxa de monitoramento no sistema refatorado (fonte: elaborada pelo autor)

As alterações no sistema original são bastante impactantes, considerando que poderia haver a necessidade de introduzir diferentes tempos de execução para cada um dos monitores, então este processo deveria ser refeito por completo. Ou seja, pensar em uma frequência de execução do *loop* que seja compatível a todos os monitores e adicionar uma condicional a cada monitor. Por outro lado, no sistema refatorado como cada monitor já possui uma taxa de monitoramento definida as alterações são simples de serem realizadas.

5.3.4 Definir ordem de execução entre monitores

Há situações em que os monitores possuem dependência entre si e precisam ser executados em uma ordem predeterminada. Assim, nesta tarefa de manutenção, o objetivo é comparar o esforço de se definir essa ordem de execução entre as duas versões do sistema. Neste caso é requisitado ao sistema que o monitor de GPS só deve ser executado se e somente se o nível da bateria esteja acima dos cinquenta por cento. Ou seja, a execução do monitor de bateria deve ocorrer antes da execução do Monitor de GPS.

Alterações no sistema original:

No sistema original, como os monitores já possuem uma sequência de execução intrínseca a sua estrutura, basta alterar a ordem de execução do monitoramento, colocando o código-fonte do Monitor de Bateria para ser executado antes do código-fonte do monitor de GPS. É importante lembrar que para cumprir o requisito será necessário também adicionar uma condicional antes da execução do monitor de GPS para verificar os valores lidos no Monitor de Bateria.

Alterações no sistema refatorado:

No sistema refatorado uma das possíveis soluções é adicionar um método que ativa a execução do ciclo de monitoramento do Monitor de Bateria e como resposta são fornecidos os valores obtidos. Este método será executado antes da execução do Monitor de GPS. O código-fonte do método adicionado a classe `BatteryMonitor` é mostrado na Figura 5.10. As alterações realizadas na classe do `GpsMonitor` são mostradas na Figura 5.11.

```
1  public class BatteryMonitor extends IntentService{
2  int level, scale, pollingRate;
3  Intent batteryStatus;
4  ...
5  public Floaat executeMonitor(){
6     level = batteryStatus.
7              getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
8     scale = batteryStatus.
9              getIntExtra(BatteryManager.EXTRA_SCALE, -1);
10    batteryPct = level / (float)scale;
11    return batteryPCT;
12 }
13 ...
```

Figura 5.10 - Alterações na classe BatteryMonitor (fonte: elaborada pelo autor)

Análise das Alterações:

As alterações realizadas no sistema original apesar de serem mais invasivas por serem realizadas na classe `ContextManager` que engloba todos os monitores demonstrou ser mais simples de ser realizada pois é necessário apenas trocar a ordem do código-fonte e adicionar uma condicional. No sistema refatorado as alterações exigem a implementação de novo código-fonte o que acaba sendo mais impactante. Uma possível alternativa poderia ser criar uma nova regra de adaptação na qual se define interromper a execução do monitor de GPS quando o nível da bateria estivesse menor que o limite determinado, contudo para realizar tal adaptação haveria a necessidade de refatorar o sistema como um todo. Como o objetivo do trabalho foi apenas o interesse de monitoramento não foram analisadas as possíveis refatorações em outras partes do sistema.

```
1  public class GPSMonitor extends IntentService{
2  ...
3  public void onHandleIntent(Intent args){
4     while(appON){
5        if(batteryMonitor.executeMonitor() < 0.5){
6        ...
7        }
8     ...
9     }
10 }
11 ...
```

Figura 5.11 - Alterações na classe GPSMonitor (fonte: elaborada pelo autor)

5.4 Considerações Finais

Neste capítulo foi descrito a refatoração realizada com o intuito de remover a presença dos *smells Obscure Monitor* e *Oppressed Monitors* da aplicação *PhoneAdapter*. É importante salientar que não foi objetivo do capítulo propor uma refatoração para solucionar nenhum dos *smells*. Também foi apresentado no capítulo quatro tarefas de manutenções para poder compara a sua realização em ambos os sistemas, original e refatorado. Na maioria das tarefas propostas o sistema refatorado demonstrou várias vantagens, principalmente por tratar-se de alterações menos invasivas ao sistema.

Capítulo 6

TRABALHOS RELACIONADOS

6.1 Considerações Iniciais

Neste capítulo são descritos os principais trabalhos relacionados que foram encontrados na literatura. Ressalta-se que não é conhecida a existência de trabalhos com o objetivo de identificar a presença de *smells* arquiteturais no interesse de monitoramento de sistemas adaptativos. Contudo, trabalhos que abordam o desenvolvimento de padrões de projeto para o interesse de monitoramento e a identificação de *smells* arquiteturais deram suporte ao desenvolvimento deste trabalho, assim, neste capítulo são mostradas as principais relações encontradas.

O capítulo está organizado em quatro seções. Na Seção 6.1 são realizadas as considerações iniciais. Na Seção 6.2 são apresentados trabalhos relacionados ao projeto do interesse de monitoramento. Na Seção 6.3 são apresentados trabalhos relacionados à identificação de *smells* arquiteturais. Por fim, na Seção 5.4 são mostradas as considerações finais do capítulo.

6.2 Trabalhos com Padrões de Projetos

Padrões de projetos são soluções utilizadas com frequência para problemas recorrentes no desenvolvimento de sistemas (Gamma 1995). Portanto, o estudo na área de padrões de projetos indica a necessidade de prover soluções para problemas que ocorrem com frequência em um determinado contexto. Nesse contexto, alguns autores identificaram padrões de projetos com o intuito de tornar o desenvolvimento do interesse de monitoramento mais adequado, o que indica a existência de soluções não tão adequadas que podem ser descritas como *smells* arquiteturais.

Weyns *et al.* (2013) analisaram o desenvolvimento de sistemas adaptativos e afirmam que em sistemas complexos há a necessidade de descentralizar os *loop*s de controle. Isto é, as adaptações dos sistemas tornam-se muito complexas e difíceis de serem gerenciadas em um único *loop* de controle centralizado. Ao identificarem este problema os autores propõem cinco padrões de projetos para *loop*s de controle descentralizados: *coodinated control*, *information sharing*, *master/slave*, *regional planning* e *hierarchical control*.

Apesar de esses padrões objetivarem solucionar o relacionamento entre diversos *loop*s de controle, o padrão *master/slave* representado na Figura 6.1, propõe que o monitoramento deve ser realizado em nodos específicos do sistema. Por exemplo, em casos que a transferência de dados para pré-processamento é muito custosa. Apesar do foco dos monitores distribuídos estar relacionado ao custo da transmissão de dados, é possível observar que os monitores são independentes.

A Figura 6.1 está dividida em duas partes, na parte superior é mostrado como o padrão é estruturado e na parte inferior é mostrado um exemplo concreto da construção do padrão.

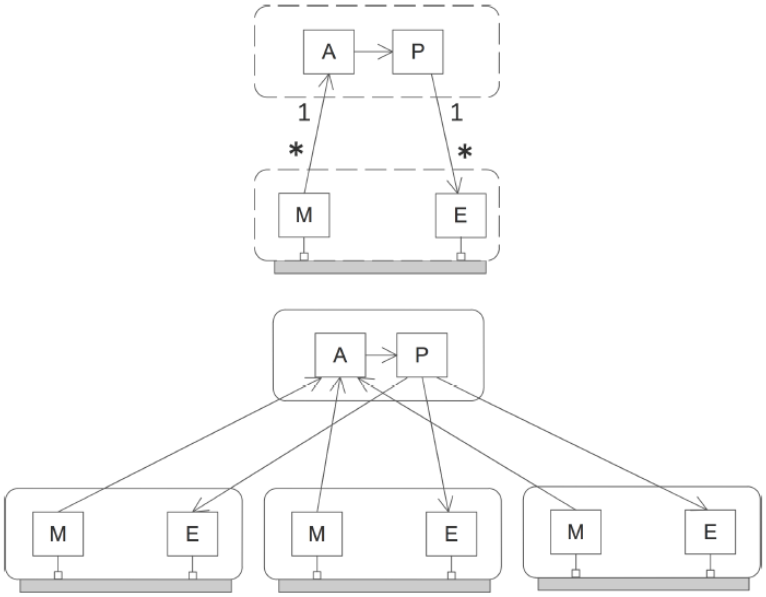


Figura 6.1 - Padrão de projeto *master/slave (Weyns et al. 2013)*

Em outro trabalho na linha de padrões, Abuseta e Swesi (2015) sugerem um padrão de design para a implementação dos elementos do *loop* de controle em sistemas auto-adaptativos. O padrão relacionado com o elemento monitor é chamado de SAS Monitor. Segundo os autores, este padrão é utilizado para estabelecer as relações entre os elementos que participam na realização do processo de monitoramento.

Na Figura 6.2 está representado o diagrama de classes para a composição do padrão sugerido no do elemento monitor. A proposta dos autores é que a classe Sensor, que estende a classe Thread, seja executada constantemente, sendo ela responsável por instanciar um sensor de contexto e coletar os dados provenientes deste sensor. Os autores ainda definem que o sensor pode enviar os dados aos monitores de duas maneiras: ativados por evento ou por tempo. Nesta figura também é possível observar que o relacionamento sensor e o monitor seguem o padrão *Observer*, ou seja, os autores sugerem que o monitoramento seja passivo, isto é, quando o sensor envia seus dados ao monitor. Neste padrão, os autores sugerem que o elemento monitor seja responsável por coletar os dados de diversos sensores, e a partir deles gerar o estado atual do sistema.

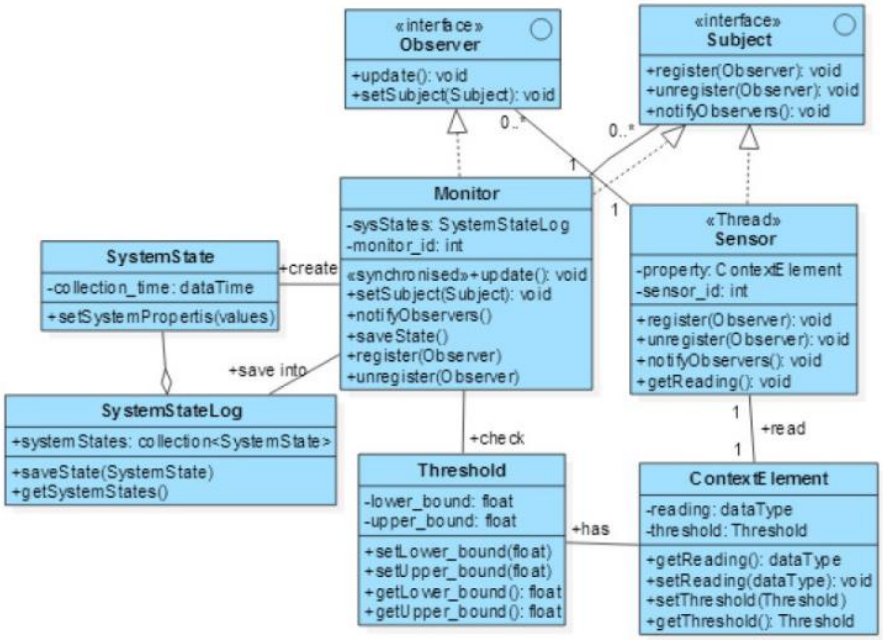


Figura 6.2 - Padrão SAS Monitor (Abuseta e Swesi, 2015)

Os demais elementos são brevemente descritos a seguir: o `ContextElement` é a classe que representa um elemento do contexto monitorado, sendo o `Threshold` os valores de referência para análise de tal contexto. As classes `SystemState` e `SystemStateLog` estão relacionadas a representação e ao armazenamento do contexto atual do sistema respectivamente.

No padrão sugerido pelos autores, é possível notar que a classe Sensor se refere ao componente monitor que é proposto neste trabalho, pois é a classe responsável por instanciar os sensores do contexto. Outra observação importante é que os autores sugerem que a classe monitor seja responsável por monitorar vários sensores ao mesmo tempo, o que pode gerar problemas relacionados à modularização como os propostos no *smell Obscure Monitor*.

Os trabalhos realizados na identificação de padrões em sistemas adaptativos indicam a necessidade de estudar como esses sistemas são desenvolvidos e proporcionar soluções que facilitem esse desenvolvimento. A identificação de *Smells* Arquiteturais segue uma linha muito próxima, pois auxilia na identificação de estruturas que podem necessitar de refatorações, podendo ser aplicados os padrões como solução.

6.3 Trabalhos com *Smells* Arquiteturais

Por meio de um levantamento da literatura em *Smells* Arquiteturais e Sistemas Adaptativos não foram identificados estudos que propõe *smells* arquiteturais específicos a Sistemas Adaptativos. Contudo trabalhos relacionados à identificação de *smells* arquiteturais serviram como inspiração para o desenvolvimento deste trabalho, esses trabalhos são descritos a seguir.

Garcia *et al.* (2009) catalogaram quatro *smells* arquiteturais para sistemas genéricos: *Connector Envy, Scattered Functionality, Ambiguous Interfaces e Extraneous Connector*. Para cada um desses *smells* é apresentado uma descrição detalhada, os impactos no sistema com a presença dos *smells* e um diagrama esquemático para auxiliar os arquitetos de software a detectarem a presença do *smell*. Os autores sugerem que os *smells* arquiteturais podem ser detectados tanto na fase de projeto do sistema quanto em uma arquitetura recuperada de um sistema já existente. Esse trabalho proporcionou bases teóricas que auxiliaram na descrição e elaboração dos *smells* propostos nesta dissertação.

Seguindo essa linha, de Andrade *et al.* (2014) conduziram um estudo exploratório objetivando caracterizar *smells* arquiteturais em Linha de Produto de Software. Para realizar este estudo os autores utilizaram um sistema de Linha de Produto de Software com código-fonte aberto. A análise do projeto foi dividida em duas partes. Na primeira parte foram identificados os *smells* genéricos catalogados em Garcia *et al.* (2009), na segunda parte foi catalogado um *smells* específicos para Linhas de Produto de Software, o *Feature Concentration*. Esse *smell* é decorrente da granularidade utilizada em alguns componentes que dificultam o entendimento e o reúso dos componentes. Assim como os autores propuseram um *smell* arquitetural específicos de domínio, esta dissertação propõe a existência de *smell* arquiteturais específicos a sistemas adaptativos.

Em uma linha similar a identificação de *smells* arquiteturais, Mo *et al.* (2015) propuseram a existência de *hot-spot* em arquiteturas de software. Os autores definem *hot-spot* como partes de uma arquitetura que ocasionam altos custos para manutenção do sistema. Os autores descreveram dois *hot-spots*: o *Unstable Interface* e o *Implicit Cross-module Dependency*. No estudo os *hot-spots* identificam os locais do sistema nos quais existem maior chance do surgimento de erros ou

exigirem alterações do código-fonte. A partir de uma avaliação os autores determinaram que os *hot-spots* definidos são responsáveis pela maioria dos problemas relacionados à manutenção. Portanto, os *hot-spots* assim como os *smells* arquiteturais representam partes da arquitetura que podem levar a dificuldades em tarefas de manutenção.

Apesar do principal foco de trabalhos que analisam a arquitetura de sistemas e identificação de *smells* ser a refatoração de sistemas, estes trabalhos também provêm práticas que devem ser bastante analisadas no desenvolvimento de novos projetos.

6.4 Considerações Finais

Neste capítulo foram abordados trabalhos relacionados que inspiraram e deram bases ao desenvolvimento da presente dissertação. Os trabalhos relacionados a padrões de projeto de sistemas adaptativos dão fortes indícios da necessidade de analisar soluções recorrentes relacionadas ao desenvolvimento desses sistemas. Sobre uma perspectiva similar é importante também identificar situações que podem ocasionar futuros problemas no ciclo de vida de um sistema, isto é, em suas tarefas de manutenção. Nesse aspecto, trabalhos relacionados à *smells* arquiteturais possibilitam identificar estruturas em sistemas que podem levar a problemas de manutenção.

Capítulo 7

CONCLUSÃO

7.1 Considerações iniciais

O interesse de monitoramento é o primeiro passo para o desenvolvimento de um sistema adaptativo. Nesta dissertação de mestrado verificou-se o projeto e desenvolvimento do interesse de monitoramento em diversos sistemas adaptativos encontrados em repositórios e em publicações da área. A partir desta verificação observou-se a existência de práticas no projeto deste interesse que podem impactar negativamente tarefas de evolução e manutenção em tais sistemas. Sendo então propostas como dois *smells* arquiteturais no interesse de monitoramento de sistemas adaptativos.

Após um estudo exploratório em que foram realizadas tarefas de manutenção em um sistema adaptativo ciente de contexto, o Phoneadapter, foi verificado que os *smells* propostos podem impactar negativamente em algumas tarefas de manutenção. Proporcionando evidências da possibilidade de que tais práticas realmente representem *smells* arquiteturais.

7.2 Contribuições

Esta dissertação de mestrado propôs aprofundar os conhecimentos no desenvolvimento de sistemas adaptativos, com o foco no interesse de monitoramento. Dois *smells* arquiteturais denominados *Obscure Monitor* e o *Oppressed Monitors* foram documentados, evidenciando que existem formas inadequadas na implementação de monitores em sistemas adaptativos. No Capítulo 3, os *smells* são descritos e caracterizados com o intuito de auxiliar desenvolvedores na análise de possíveis estruturas arquiteturais que podem influenciar nos atributos de qualidade interna do sistema. Nesse capítulo também é fornecido um guia para identificação de cada *smell* com o intuito de auxiliar desenvolvedores na verificação de sistemas adaptativos. No Capítulo 4 foram evidenciados alguns sistemas que apresentam a presença do *smells* descritos. No Capítulo 5, foi realizada uma avaliação preliminar para verificar o impacto da presença dos *smells* em tarefas de manutenção.

Portanto, as principais contribuições nesta dissertação de mestrado são para o avanço da área de refatoração, principalmente no contexto de sistemas adaptativos. Em especial, refatorações no interesse de monitoramento, pois os *smells* descritos contribuem na identificação de problemas relacionados a construção e desenvolvimento dos monitores. A descrição dos *smells* auxilia o engenheiro de software na tomada de decisões quanto à necessidade de realizar refatorações, e até mesmo no desenvolvimento de novos projetos de sistemas adaptativos. Uma vez que monitores independentes e com taxa de monitoramento próprio facilitam em desenvolver adaptações relacionadas aos próprios monitores.

7.3 Limitações

As principais limitações do trabalho são listadas a seguir:

- Foram apenas analisados sistemas desenvolvidos com a linguagem JAVA, portanto os *smells* definidos podem ser específicos para esta linguagem. Contudo os *smells* definidos estão relacionados a como os Monitores

estão estruturados, logo acredita-se que estes *smells* também estejam presentes em sistemas desenvolvidos com linguagens similares orientadas a objetos.

- Embora foram encontrados sistemas com a presença dos *smells*, não foi conduzida uma avaliação aprofundada no sentido de evidenciar que a presença dos *smells* realmente é problemática. Além disso, também não foi feito um estudo no sentido de mostrar que a presença dos *smells* pode ser inadequada para determinadas situações, contudo não apresentar problemas para outras.
- Na análise preliminar realizada, as tarefas e os impactos apenas avaliaram o interesse de monitoramento, ou seja, o impacto real da refatoração no sistema como um todo não foi avaliado. Assim, é possível que alguma das tarefas realizadas podem ter sua execução facilitada no sistema refatora quanto ao interesse de monitoramento mas afetar de maneira negativa manutenções em outras partes do sistema.

7.4 Trabalhos Futuros

Como trabalho futuro é necessário a realização de um estudo empírico qualitativo com especialistas para saber se os smells representam realmente *smells* arquiteturais do interesse de monitoramento em sistemas adaptativos. Após esta verificação é possível pensar em duas principais vertentes para continuidade das pesquisas:

A primeira é aprofundar estudos relativos ao interesse de monitoramento. Para isso, é necessário verificar a influência dos *smells* propostos nos d elementos do *loop* de controle. Definidas as principais influências é possível então propor formas de refatorar os monitores para que se adequem a algum padrão proposto, caso houver.

Uma segunda vertente é aprofundar o estudo na busca por mais *smells* arquiteturais, analisando os demais elementos do *loop* de controle: Analisador, Planejador e Executor. Verificar se existem formas de implementação nesses

elementos que comprometam a realização de manutenções e evoluções do sistema, para então também caracterizá-los como *smells* arquiteturais.

7.5 Publicações

Os resultados obtidos foram divulgados nos seguintes trabalhos:

- SERIKAWA, M. A. *et al.* Guidelines for Modularizing the Monitor Component when Refactoring Adaptive Systems. In: Segunda Escola Latino-Americana de Engenharia de Software (ELAES), 2015, Porto Alegre, Brazil.

- SERIKAWA, M. A. *et al.* Towards the Characterization of Monitor *Smells* in Adaptive Systems. In 10th Brazilian Symposium on Components, Architecture, and Reuse (SBCARS), 2016, Maringá, Brazil. (aceito para publicação)

- DE PAULA, M. H. *et al.* SARA: Uma Arquitetura de Referência para Facilitar Manutenções em Sistemas Robóticos Autoadaptativos. In IV Workshop on Software Visualization, Evolution and Maintenance (VEM), 2016, Maringá, Brazil. (aceito para publicação)

REFERÊNCIAS

Abuseta, Y. e K. Swesi (2015). "Design Patterns for Self Adaptive Systems Engineering." International Journal of Software Engineering Applications Vol. 06(No. 4).

Amarasekara, B., D. Jayasuriya e Nishami (2014, 2014). "Novi." Retrieved 2016, 2016, from https://sourceforge.net/projects/novi/.

Arkin, R. C. (1998). Behavior-based robotics, MIT press.

Baldauf, M., S. Dustdar e F. Rosenberg (2007). "A survey on context-aware systems." International Journal of Ad Hoc and Ubiquitous Computing 2(4): 263-277.

Bass, L., P. Clements e R. Kazman (2003). "Software architectures principles and practices." Addison-Wesley.

Beck, K., M. Fowler e G. Beck (1999). "Bad smells in code." Refactoring: Improving the design of existing code: 75-88.

Biegel, G. e V. Cahill (2004). A framework for developing mobile, context-aware applications. Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on, IEEE.

Brun, Y., G. D. M. Serugendo, C. Gacek*, et al.* (2009). Engineering self-adaptive systems through feedback loops. Software engineering for self-adaptive systems, Springer: 48-70.

de Andrade, H. S., E. Almeida e I. Crnkovic (2014). Architectural bad smells in software product lines: An exploratory study. Proceedings of the WICSA 2014 Companion Volume, ACM.

De Lemos, R., H. Giese, H. A. Müller*, et al.* (2013). Software engineering for self-adaptive systems: A second research roadmap. Software Engineering for Self-Adaptive Systems II, Springer: 1-32.

Fan, Y., J. Wolfson, G. Adomavicius*, et al.* (2015). "SmarTrAC: A smartphone solution for context-aware travel and activity capturing."

Fowler, M. (2009). Refactoring: improving the design of existing code, Pearson Education India.

Gamma, E. (1995). Design patterns: elements of reusable object-oriented software, Pearson Education India.

Garcia, J., D. Popescu, G. Edwards*, et al.* (2009). Identifying architectural bad smells. Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on, IEEE.

Garlan, D., S.-W. Cheng, A.-C. Huang*, et al.* (2004). "Rainbow: Architecture-based self-adaptation with reusable infrastructure." Computer 37(10): 46-54.

Gil de La Iglesia, D. (2014). "A formal approach for designing distributed self-adaptive systems."

Horn, P. (2001). "Autonomic computing: IBM\'s Perspective on the State of Information Technology."

Hussein, M., J. Han e A. Colman (2010). Context-Aware Adaptive Software Systems: A System-Context Relationships Oriented Survey, Technical Report# C3-516_01, Swinburne University of Technology.

IBM (2005). An architectural blueprint for autonomic computing. IBM Publication.

Kephart, J. O. e D. M. Chess (2003). "The vision of autonomic computing." Computer 36(1): 41-50.

Kiczales, G., E. Hilsdale, J. Hugunin*, et al.* (2001). An overview of AspectJ. European Conference on Object-Oriented Programming, Springer.

Kruchten, P., H. Obbink e J. Stafford (2006). "The past, present, and future for software architecture." IEEE Software 23(2): 22-30.

Lalanda, P., J. A. McCann e A. Diaconescu (2013). Autonomic Computing, Springer.

Lippert, M. e S. Roock (2006). Refactoring in large software projects: performing complex restructurings successfully, John Wiley & Sons.

McKinley, P. K., S. M. Sadjadi, E. P. Kasten*, et al.* (2004). "Composing adaptive software." IEEE Computer: 56-64.

Mo, R., Y. Cai, R. Kazman*, et al.* (2015). Hotspot patterns: The formal definition and automatic detection of architecture smells. Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on, IEEE.

Ogata, K., P. Á. Maya e F. Leonardi (2003). Engenharia de controle moderno, Prentice Hall.

Oppermann, R. (1994). "Adaptively supported adaptability." International Journal of Human-Computer Studies 40(3): 455-472.

Parnas, D. L. (1972). "On the criteria to be used in decomposing systems into modules." Communications of the ACM 15(12): 1053-1058.

Ramirez, A. J. e B. H. Cheng (2010). Design patterns for developing dynamically adaptive systems. Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, ACM.

Rapps, S. e E. J. Weyuker (1982). Data flow analysis techniques for test data selection. Proceedings of the 6th international conference on Software engineering, IEEE Computer Society Press.

Sama, M., S. Elbaum, F. Raimondi*, et al.* (2010). "Context-aware adaptive applications: Fault patterns and their automated identification." IEEE Transactions on Software Engineering 36(5): 644-661.

Sanchez, R. e J. T. Mahoney (1996). "Modularity, flexibility, and knowledge management in product and organization design." Strategic management journal 17(S2): 63-76.

Stal, M. (2007). Software architecture refactoring. Tutorial, in The International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA).

Tallabaci, G. e V. E. S. Souza (2013). Engineering adaptation with Zanshin: an experience report. Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, IEEE Press.

Taylor, R. N., N. Medvidovic e E. M. Dashofy (2009). Software architecture: foundations, theory, and practice, Wiley Publishing.

Uwa, M. M. (2013). "AIASProject." from https://github.com/maltic/AIASProject.

Weyns, D., B. Schmerl, V. Grassi*, et al.* (2013). On patterns for decentralized control in self-adaptive systems. Software Engineering for Self-Adaptive Systems II, Springer: 76-107.