Towards the Characterization of Monitor Smells in Adaptive Systems

Marcel A. Serikawa, André de S. Landi, Bento R. Siqueira,
Renato S. Costa, Fabiano C. Ferrari, Ricardo Menotti, Valter V. de Camargo
Department of Computing - Federal University of São Carlos
São Carlos, São Paulo - Brazil
Email: {marcel.serikawa, andre.landi, renato.costa, fabiano, menotti, valter}@dc.ufscar.br

***Abstract*—Adaptive Systems (ASs) can adapt themselves to a changing environment or new user needs. Monitors are essential in AS, being responsible for collecting and processing data from environment. There exist different kinds of monitors with distinct characteristics. Based on a literature review, we have noticed that Monitors are usually designed and implemented in an inadequate way: i) making them obscure in the source-code; ii) compelling all of them to have the same polling rate and also iii) predetermining the execution order among them. This leads to maintenance, evolution and performance problems. Besides, based on our observations, this erroneous way monitors are implemented follows a pattern and it is a recurrent practice. Therefore, we believe it can be classified as Monitor Smells of Adaptive Systems. In this paper we present two architectural smells we have identified: the Obscure Monitor and the Oppressed Monitors. The first smell occurs when the monitors are not evident in the source-code. The second smell occurs when monitors are compelled to have the same poling rate and an immutable execution order at runtime. The presence of these smells compromises the reusability, evolvability and maintainability. We have also conducted an exploratory study by comparing the impact of maintenance tasks in the original version of an AS called `PhoneAdapter` with a refactored version, in which the smells were removed. The results indicate the maintenance is facilitated in the version without the smells.**

I. Introduction

Adaptive Systems (AS) are able to modify their structure or behavior at runtime, in order to provide a better user experience and quality of service [1]–[3]. Although they are usually referred as a unique system, conceptually they can be viewed as two distinct sub-systems: the managed sub-system and the managing sub-system [4], [5]. The managed one is where the user requirements reside, i.e., the application itself. Yet, the managing one comprises all the source-code responsible for the adaptations.

The design of ASs is a research topic aiming at structuring them in a way to make maintenance and evolution a more manageable and productive task. ASs are inherently composed by control loops (CLs) and they must be designed/implemented making the CL abstractions evident in the source-code [3], [5], [6]. CL is a subject from the Control Engineering field and comprises an adaptation cycle that act over a process/system to be adapted [3]. The most important concepts/abstractions of CLs are Monitors, Analyzers, Planners and Executors.

Monitors are software components that are responsible for observing, collecting, processing and broadcasting the data provided by sensors. They provide information about the environment or the system itself [7] and can be classified as dependent or independent. The dependent ones need to consume data from other monitors to proceed with their operation. On the other hand, the independent ones do not need interact with other monitors to perform their processing. The focus on this paper is the second type, the independent.

In general, monitors have two important facets; a static one and a behavioral one. The static facet is concerned with the source-code responsible for the monitoring logic, the processing logic and the broadcasting logic. These compose the main functionality of a monitor. On the other hand, the behavioral facet is concerned with the source-code responsible for: i) defining a polling rate and ii) guarantee the independent execution for the independent monitors. Polling rate is the frequency in which the monitor captures data from sensors. For example a polling rate of 0.5 seconds means that the sensor data are collected at every 0.5 seconds. The independent execution is concerned with guarantee that independent monitors do not need to wait for others to finish their work. Examples of Monitors are GPS Monitor, Bluetooth Monitor, Ultrasonic Monitor, Laser Monitor, Touch Monitor and WeekDay Monitor. Each of these demands different behavioral characteristics, for example, normally it makes no sense a Bluetooth and a WeekDay monitors have the same polling rate.

We conducted a literature review concentrated on analyzing the way monitors are designed/implemented in ASs. We have observed that, in some cases, they are designed in an inadequate way, leading to the following problems: i) the monitoring, processing and broadcasting logic get obscured in the source-code; ii) all independent monitors are oppressed to have the same polling rate and iii) an unmodified execution order is imposed to them. Normally, the source of the first problem is not implementing monitors as first class entities and the source of the second and third problems are the erroneous employment of a loop structure or a central component that manages all of the monitors.

Based on our investigations, the erroneous way they are implemented follows a pattern and can be documented as Architectural Smells for ASs. Architectural smells are design decisions that are non-obvious and leads to significant detri-

978-1-5090-5086-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SBCARS.2016.19

mental impacts on maintainability [8]. To remediate architectural smells or architectural deviations, larger refactorings are required, known as Architectural Refactoring (AR).

In this paper we present two architectural smells called Obscure Monitor and Oppressed Monitors. The first occurs when the monitors are not evident in the source-code, i.e., they are not implemented as first-class entities. The second occurs when independent monitors are compelled to have the same polling rate and a predetermined execution order among them.

Furthermore, an exploratory study is also presented , where we have refactored an AS called `PhoneAdapter` and compared it with its original version. We have applied and analysed some maintenance tasks in both systems. The results show that the maintenance tasks in the refactored system are less invasive to the system.

This paper is divided as follows. In section II, a brief background for this paper. In section IV, the two smell proposal for ASs. In section III, a practical example of both smells. In section V, the refactoring for a system with the smells. In section VI, an exploratory study to exemplify and evaluate the smells impact. In section VII, relevant works related to architectural smells. In section VIII, a discussion and conclusion about the smells and the exploratory study.

II. Adaptive Systems and Control Loops

Adaptive systems (ASs) are able to modify their own behavior and/or structure in response to their perception of the environment and the system itself [2]. In most of the cases they are viewed as a unique system, however they may be conceptually interpreted as two systems: the managed system and the managing system. The managed one is the application itself, i.e., where resides the user requirements. Normally, it operates independently. On the other hand, the managing system is responsible for monitoring the managed one and trigger adaptations on it, changing its behavior when necessary. Another important conceptual division is that the managing system can be divided in "control loops" (CL).

CL is a largely used concept in Control Theory, being the generic mechanism responsible for making systems adaptable [3], [5]. The adaptation process is accomplished as follows: i) the CL receives as input data through sensors; ii) then it evaluates if this data complies with the system goals; iii) if it does not, the CL plans possible adjustments targeting the objectives; iv) after planning the CL executes the adaptations on the managed system [3]. Therefore, the CL cycle can be conceptually divided in four concerns: monitoring, analysing, planning and executing as precognized by the MAPE-K conceptual model shown in Figure 1.

Figure 1 shows a schematic view of an AS divided in two parts. The upper part represents the managing sub-system containing the following components: monitor, analyzer, planner, executor and a knowledge based component. The first four elements represent the control loop sub-process. There are also the Sensors and Actuators that links the managing subsystem to the managed one, which is represented by the box

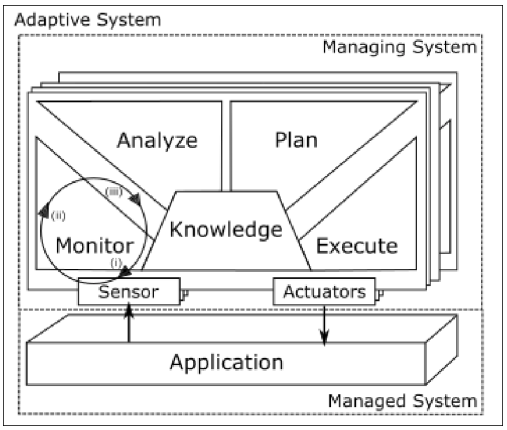

Fig. 1. MAPE-K schematic view

Application in the lower part of the figure. As it is represented in this figure, an adaptive system usually has more than one control loop, each one is related to a different property, i. e., a control loop for security, battery management and others.

TABLE I
Some examples of Sensors APIs

Platform	Sensor Type	API available classes/packages
Android	Accelerometer	Sensor.TYPE_ACCELEROMETER
	GPS	android.location.GpsStatus.Listener
	Bluetooth	android.bluetooth
LEJOS	Touch	lejos.nxt.TouchSensor
	Light	lejos.nxt.LightSensor
	Distance	lejos.nxt.UltrasonicSensor
JAVA ME	General I/O	com.oracle.deviceaccess
	Analog Input	com.oracle.deviceaccess.adc
	Power Management	com.oracle.deciceaccess.power

Monitor is an important CL abstraction, since it is responsible for initiating the adaptation cycle, as showed in Figure 1. Monitor behavior follows behavior follows a pattern, called here "Monitoring cycle", which can be schematically viewed over the monitor component in Figure 1. This cycle includes three steps: i) collecting data from sensors; ii) preprocessing these data and iii) broadcasting the results to other components. Monitors have a close relationships with sensors which are usually available as APIs (Application Program Interfaces) providing the communication between hardware and software [7]. Table I shows some examples of sensor APIs. The first column contains the platform of the provided sensor APIs; the second contains the type of sensor; and the last column shows the class or packages to instantiate the sensors.

According to our investigation, a possible classification for monitors is regarding their dependency, being classified as dependents or independents. Dependent monitors do not operate alone, that is, they depend from the data provided by other monitors, so their execution needs to be scheduled with other monitors. The independent ones do not need to be synchronized with others to operate as they do not dependent neither on the data nor on the control flow of other monitors. The independent monitors are the focus of this paper, so from

this point on, every time we mention the term "monitor" we are referring to the independent ones.

Also in our investigation, we have identified that Monitors have two important facets: the static one and the behavioral one. The static facet is concerned with the source-code responsible for the monitoring logic, the processing logic and the broadcasting logic. These compose the main functionality of a monitor. The monitoring logic is all the source-code responsible for getting the data provided by the sensor API. The preprocessing logic performs conversions, standardization, aggregation and filtering of the data collected from sensors [7]. The broadcasting logic involves sending the data collected for the next components.

The behavioral facet is concerned with the monitor execution. This facet involves the source-code responsible for: i) managing a polling rate and ii) managing the execution order among them or guarantee their independence operation. Polling rate is the frequency that monitors capture data from sensors - what triggers their monitoring cycle. For example, the polling rate of a Battery Monitor can be set for 5 minutes while a Weekday Monitor can have a polling rate of 12 hours. Therefore, each monitor can have its own polling rate. In the case of independent monitors, there is no order among them. So, the way these monitors are implemented must guarantee their execution freedom. Examples of possible monitors are GPS Monitor, Battery Monitor, Wifi Monitor, Ultrasonic Monitor, Camera Monitor and WeekDay Monitor.

The monitoring concern can also be classified as independent and dependent. The independent monitoring is able to be executed by itself independent of the managed subsystem. On the other hand, the dependent monitoring executes only when the managed subsystem requires, similar to aspect oriented programming (AOP) [9].

III. Motivating Example

In order to exemplify the smells described we have used an AS called `PhoneAdapter` [10]. It characterizes itself as "adaptive" because it uses context information to adapt the mobile phone profile. Phone profile is a configuration that determine the phone's behavior. This configuration may involve the display intensity, ring tone volume, vibration mode, Bluetooth discovery and others. Due to its adaptability capacity, the application automatically changes between profiles based on rules previously set. The selected profile prevails until a more suitable one is activated through other rules. The rules are based on context readings from Bluetooth, GPS sensors and the internal clock of the phone [10]

Figure 2 presents part of the class diagram of the `PhoneAdapter`. This figure is divided in two parts, the upper part representing the Original system and the lower part representing the Refactored system, which will be discussed in Section VI. As it is shown in the upper part of this figure, the managing system involves the `ContextManager` and the `AdaptationManager` classes.

The `ContextManager` class is responsible for collecting and preprocessing the monitored data. Therefore, in the

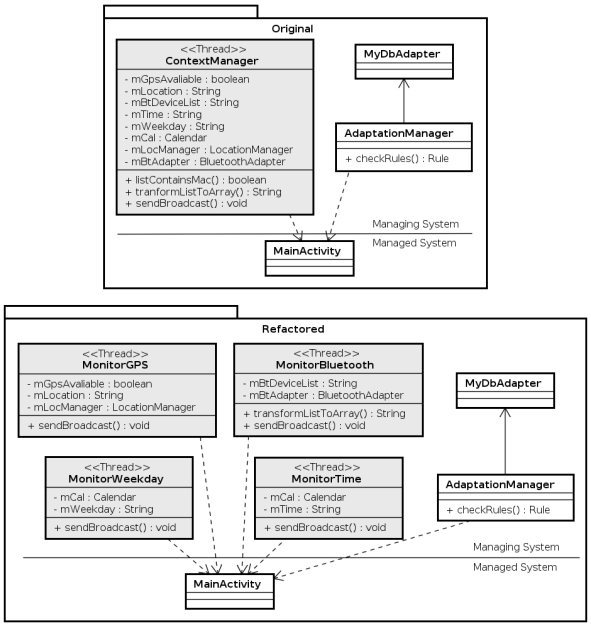

Fig. 2. `PhoneAdapter` class diagram Original and Refactored.

`PhoneAdapter`, the monitoring concern is designed and implemented in this class. The `AdaptationManager` is responsible for managing the necessary adaptations according to rules stored in the database accessed by the `MyDbAdapter` class. The `MainActivity` class represents the managed system. Notice that in `ContextManager` class, there are attributes and methods related to the data monitored, for example, the `mLocation` is related to GPS position while the method `listContainsMac()` is related to list all active Bluetooth devices. This system is quite representative and follows a conventional architecture commonly found in Context-Aware AS [10].

Figure 3 shows the code snippet of this class. It encompasses distinct types of Monitors that are: GPS, Bluetooth and Calendar. In this figure each different monitor is highlighted in a numbered box from 1 (one) to 4 (four): 1) Time Monitor; 2) Weekday Monitor; 3) Bluetooth Monitor; 4) GPS Monitor. It is possible to see that the same monitor is composed by pieces of code that are not modularized, that is, they are spread and tangled with the code of other monitors. For example, the Time monitor is composed by five boxes - five code snippets. The same occurs with the others.

This class extends `IntentService` which is equivalent to extend a `Thread` class in Java. That is, this class is executed in parallel to the main application, which is normally a common characteristic of monitors. The `ContextManager` class is describe as follows:

- from line 1 to 4 there are import statements related to three sensors APIs: GPS, bluetooh and calendar.
- from line 7 to 13 there are the attributes related to sensor instantiation, the `mCal` for the time and days of the week;

```
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     mDtAdapter.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   }
67  private String transListToArray(                            3
68                  ArrayList<String> list){
69   String[] s=new String[list.size()];
70   for(int i=0;i<list.size();i++){
71    s[i]=list.get(i);
72    }
73   return s;
74  }
75 }
```

Fig. 3. `ContextManager` class snippet code

the `mLocManager` and `mLocListener` related to the GPS; the `mBtAdapter` and `mReceiver` related to the Bluetooth device. Also there are the `mTime` and `mWeekday` for storing the time and day of the week respectively.

- on line 15 the method (`onCreate()`) is executed during the class instantiation, loading all necessary sensors.
- on line 27 the method (`onHandleIntent()`) is responsible to execute the class main logic. While this method is being executed this class object will be active and when this method finishes its execution the object is destroyed. Therefore, at line 28, there is the loop statement responsible to keep this class active while the application is running.
- from line 29 to 65 there are all monitors grouped inside the loop in a sequence. This loop contains the static facet of a monitor (monitoring logic, preprocessing logic and the broadcasting logic). For example the Time Monitor (box number 1): in line 30 the `mCal.getTime()` snippet code is the monitoring logic to acquire data from time sensor. In line 29, the `SimpleDateFormat` object is responsible to the preprocessing logic of previous data, specifying a data format. In line 57 the time data collected and preprocessed is added in a content, to be sent by broadcasting logic.
- on line 63 it is set the period of time that this loop is executed, that is, the polling rate for every monitor.
- from 67 to 74 the a method (`transListToArray()`) preprocess the bluetooth data and is called inside the loop.

IV. Monitor Smells

This section presents the Obscure Monitor and the Oppressed Monitors smells. For each of them it is given a description, identification guidelines and the quality impact and trade-off.

A. Obscure Monitor Smell

Description: The Obscure Monitor smell is related to the static facet of the monitor (monitoring logic, broadcasting logic and preprocessing logic). It is characterized when a monitor is not implemented as a first-class entity, making the source-code of this facet tangled with the source-code of other components. In object-oriented systems, that means there is not a class or interface that represent the monitor, but a set of lines of code inside arbitrary classes or methods.

In the Motivating Example shown in Section III, the Obscure Monitor is presented in the `ContextManager` class shown in Figure 3. For example, regarding the Bluetooth Monitor (box number 3), all of its source-code is spread in four boxes. The first one, from line 22 to 24, encapsulates the first part of the monitoring logic. The second one, from line 40 to 43, encapsulates the second part of the monitoring logic. The third one, from line 53 to 56, the broadcasting logic. The last one, from line 67 to 74, the preprocessing logic. As can be observed, the same occurs with the other 3 monitors.

Identification Guidelines: Next, there are some guidelines to assist in the identification of the Obscure Monitor smell.

1) Identifying all the sensors used by the system. This is the first step because monitors rely on the existence of sensors for

gathering relevant data. Therefore an alternative to identify all sensors is making a search in platform documentation because sensors are usually implemented using specific platform APIs as the ones listed in Table I.

2) Identifying all the classes that import/use the sensors found in the step before. The next step is to identify which classes are using/instantiating sensors. Classes that instantiate a sensor are very likely to have one or more monitors related to it. A possible way for identifying these classes is investigating the system class by class. Another possible alternative for automatically identifying is using a mining tool. In this step, it is important to identify classes that are using more than one sensor.

3) Identifying the monitors logic. The last step is identifying the monitoring, preprocessing and broadcasting logic related to each monitor. This task exclusively depends of the developer who have to search all source-code related to each monitor. Therefore, requires a previous knowledge about the system.

Listing 1 shows an algorithm that implements the above guidelines and can be used for detecting the Obscure Monitor smell. It receives two inputs provided by the user and generates one output. As input, the `allSystemClasses` represents a list of all classes of the system and the `sensorToSearch` represents a list of the sensors to be searched. This is usually all sensors of a specific platform. As output the `classesWithSensor` represents a list of all classes that have at least one sensor imported. The algorithm execution gets all classes in the `allSystemClasses`, one by one and searches for each sensor listed in the `sensorToSearch`. Then if a sensor is found in the current searched class, this class will be added in the `classesWithSensor`. It will result in a list with classes that have any type of sensor. Therefore, the class that has more occurrences in this list possibly indicates the Obscure Monitor smell presence.

```
Input ListSystemClasses allSystemClasses,
      ListSensorClasses sensorToSearch;
Output ListSelectClasses classesWithSensor;

    for(class in allSystemClasses){
        for(sensor in sensorToSearch){
            if(class has sensor)
            add class in classesWithSensor
        }
    }

    return classesWithSensor;
```

Listing 1. Obscured Monitors Smell Mining Algorithm

Quality Impact and trade-off: The software quality attributes that the Obscure Monitor smell may direct affect are monitor reusability, understandabilty and maintainability.

The impact in reusability is because of the lack of modularization. For instance, if an obscure monitor has to be reused in a different system, the monitor logic will have to be re-implemented because it is coupled with non related code.

The impact in understandability is related to specificity of each monitor. That is the data being monitored is usually from very different concerns, for example, geographic location and motion detection, then each of these data have specific ways to be preprocessed. Therefore, each monitor may employ a particular complex algorithm strategy and/or formulas. So, if they are tangled in the same class, the understandability of this code will be compromised.

The impact in maintainability and evolution of the system is related to the lack of modularization. That is, if a monitor need to be changed it may directly affect the behavior of others monitors once they are highly coupled.

B. Oppressed Monitors Smell

Description: The Oppressed Monitors smell is related to the behavior facet (polling rate and execution order). It is characterized by a set of monitors that exhibits the following three main characteristics: i) they are independent from each other concerning the data manipulated; ii) they have the same polling rate and iii) the execution order of the monitors is predetermined in compilation time and unmodifiable in runtime. This usually happens due to bad implementation decisions leading all monitors to be confined in a unique loop that takes from them their autonomy.

The first characteristic is regard that they are independent from each other. Most of the times this is conventional situation, i.e., they can operate without dependencies from other monitors. The dependency between monitors usually occurs at data level, i.e., when a monitor depends on the data captured by other monitors.

The second characteristic (having the same polling rate) is caused by employing a looping that wraps all monitors together. Therefore, all monitors inside the loop perform their execution in a polling rate imposed by the loop and not decided by themselves. This leads to an unnecessary data capturing frequency for some monitors impacting the performance and loss of resources. In many situations the best alternative is allowing for each monitor to have its own polling rate, being possible to change it in runtime.

The last characteristic, regarding the predetermined execution order, occurs because monitors are sequentially disposed inside of the same loop. Therefore, a specific monitor can only start its operation when the previous one has released the control. This may lead to two problems: i) if a monitor crashes, all monitoring will be affected being not executed; ii) if a monitor has a slow execution it will increase the likelihood of other monitors getting out of date data and, consequently, generating erroneous contexts.

In summary, the Oppressed Monitors smell can be understood as two abstractions: monitors and a manager. The monitor manager is responsible for the execution of all monitors. This distinction is important because both can be implemented as classes or methods. It is important to notice that even if all monitors are designed/implemented as first class entities, this smell can be present. Therefore, the main goal when refactoring this smell is to remove the monitor manager, making the monitors independent.

In the Motivating Example in Section III, the Oppressed Monitors smell is characterized by the while loop from line 28 to 66. This loop imposes the same polling rate to all the four monitors, i.e., oppressing all of them to perform the monitoring

logic at every 2 (two) minutes, represented by the 120000 ms at line 63. Besides, this loop also predetermines the execution order for all the four monitors, as follows: Time Monitor, Weekday Monitor, Bluetooth Monitor and GPS Monitor. For example, the Weekday Monitor only is allowed to start its operation when the Time Monitor finishes its execution. This is a strong indication of a problem since these two monitors are independent and should have very different polling rates. For example the Weekday Monitors could have a polling rate of 24 hours while the Time Monitors of 2 minutes.

Identification Guidelines: The guidelines provided as follows aims at supporting the identification of a structure that manage all monitors.

1) Identifying all the monitors. The first step is to search for all monitors components. The algorithm in Listing 1 can support this search as it helps to find classes that use a sensor, therefore, it helps in finding the monitors.

2) Identifying classes that use monitors. After finding all monitors, the next step is to verify if there is a structure that wraps them. The monitors can be structured in two ways: obscure monitors or modularized monitors. If monitors are obscure, usually the monitor manager can be found in the same class of monitors as a loop statement. However, if the monitors are modularized, the algorithm in Listing 1 can be adapted to search for monitor classes instead of sensors. If a class that wraps all monitors is found, it is necessary to verify if its logic submit them to a same polling rate and a predetermined execution order.

Quality Impact and trade-off: In some cases the Oppressed Monitors smell is acceptable, like when there are simple monitors with similar polling rate. It is acceptable because creating separated monitors as parallel processes may lead to an overhead problem. However, in other cases, when the monitors require very different polling rate. For example, the Weekday Monitor and the GPS Monitor. In the first monitor, data changes at every twenty four hours meanwhile depending on the situation the GPS data may change every minute or less. In this situation, with the presence of the Oppressed Monitors smell, the loop structure needs to comply with the GPS high rate of change leading to useless Weekday monitoring process. This leads to unnecessary waste of resources because the Weekday Monitor is operating in a polling rate faster than necessary.

Another quality impact of monitors in the same loop structure is that they are highly coupled. One becoming dependent of the previous execution. This execution dependence may affect the data gathered by other monitors leading to an incorrect behavior of the managed sub-system. For example, in a robotic system, the GPS Monitor is executed and then in the sequence the distance monitor, if the GPS takes too long to be preprocessed the gathered distance will be an erroneous data. This may lead to an unexpected behavior.

C. Smell Evidence in other Systems

A real smell is recurrent in practice. In order to obtain evidences of that, we have employed two searching strategies. The first one was concentrated on analyzing AS-focused papers in which there was source code snippets showing parts of the monitoring concern. The second one was concentrated on searching for ASs in source-code repositories, so that we could analyze the source-code of them for detecting how monitors were implemented. As a result we have found some applications and frameworks/architectures that exhibit the mentioned smells, as it is presented in Table II. This table is divided in two parts; the upper one shows applications while the lower one shows Frameworks/Architectures. Besides, the first column contains the name of the application/framework, the second one contains a brief description and last one presents the classes/elements in which both smells were found. It is important to highlight that the presence of the smells in frameworks leadS to strong impacts. That happens because they are used as base for developing applications, and consequently, all the applications developed with their support will present the smells.

V. Refactoring PhoneAdapter

In this section we demonstrate a possible way for refactoring applications in order to solve the smells presented in this paper. We exemplify our strategy with the `PhoneAdapter` Application since it presents both smells the Obscured Monitor smell and Oppressed Monitors smell. Our refactoring strategy is based on two main tasks: i) creating a class for each existing monitor of the system - what solves the Obscure Monitor smell and ii) eliminate loops that manage the monitors - what breaks out the centralized nature and solves the Oppressed Monitors smell. Therefore, the new design must guarantee that each monitor is evident in the code as first-class entity, independent in terms of data and have its own polling rate.

The lower part of the Figure 2 shows the refactored version. As can be seen, the `ContextManager` class no longer exist. This class was broken into four new monitor classes (`MonitorWeekday`, `MonitorGPS`, `MonitorTime`, `MonitorBluetooth`), making the monitors evident in the source-code as first-class entities. Methods and attributes related to each monitor are now in their respective classes. For instance, the method `sendBroadcast()` is responsible to send the collected data to the `AdaptationManager` class. This strategy of slitting a unique class that was encapsulating all the monitoring concern into new ones solves the Obscure Monitor smell. To solve the Opressed Monitors smell we have to guarantee that there is no loop that controls the monitors, so each monitor must have its own parallel process. This is not shown in this figure.

To make our refactoring suggestion easier, we have created a simple template class to be completed, this template is shown in Listing 2 and it is an evolution from a previous publication [18]. The template is based on the main characteristics of monitors [7], [19], [20], that are listed as follows:

1) It has to be a parallel process to the managed system. For example, in Java this class should implement `Runnable`

TABLE II
APPLICATIONS, FRAMEWORKS AND ARCHITECTURES WITH MONITOR SMELLS

Applications	**Description**	**Present Smells**
AIASProject [11]	An adaptive robotic application for artificial intelligence test.	Both smells are present in the *ScriptBot* class which involves five monitors.
PhoneAdapter [10]	An adaptive Context-Aware application for mobile phones where adaptations are guided by previously defined rules.	Both smells are present in the `ContextManager` class which involves four monitors.
Smartrac [12]	A mobile Context-Aware application for Travelling and Activity Capturing	Both smells are present in the `SensorDataCapture` class which involves two monitors
Vehicle Recommendation [13]	A vehicle self-configuring application based on previous driver preferences.	Both smells are present in the `DriverLearning` module which may involves multiple monitors.
Framework	**Description**	**Present Smells**
Zanshin [14]	A framework for developing adaptive software.	Oppressed Monitors is present in the `MonitorThread` class which activate monitors one by one.
CAA Framework [15]	A framework for the development of context-aware mobile applications.	Both smells are present in the object Sentinel which may be responsible for multiple monitor.
SOCAM [16]	A framework for the development of context-aware mobile services.	Both smells are present in the `ContextInterpreter` class which may be responsible for multiple monitors.
Behavior-based [17]	An architecture used in autonomous robotic field.	Oppressed Monitor is present in the behavior manager class which activate each behavior one by one.

or extend `Thread`. For the Android platform, it should implement the `IntentService`.

2) Sensors must be instantiated in the beginning of the monitoring process. Therefore, this can be done in the constructor method or in a loader method.

3) It has to have a constant loop structure in order to keep the data up to date. That is, this loop must contain the monitoring logic of how data are gathered.

4) The loop must be triggered by a polling rate that determines when the monitoring logic is executed. That is, the polling rate sets the frequency of the loop with the monitoring logic.

The guidelines we have followed for conducting this refactoring has 3 main steps: (i) identifying the sensor responsible to get data on the original code with the supportive algorithm in Listing 1; (ii) identifying all code related to this sensor, this includes attributes, code snippets and methods; finally (iii) separating the monitoring logic, the preprocess logic and broadcast logic from the code identified in the last step.

```
public class monitorName [extends/implements] [Runnable/
    Thread/IntentService/...] {
    //attributes like ''int pollingRate = ?;"
    public monitorName() {
        //monitoring logic (initiate the sensors here)
    }
    protected void [run/onHandleIntent/...](<parameters>) {
        while(appOn) {
            //monitoring, preprocessing, broadcasting logic
  here
            //polling rate
        ...
    }
```

Listing 2. Template class for Monitors

An example of a refactored monitor is shown in Listing 3. This code snippet is related to the Weekday Monitor (box number 2) in Figure 3. Therefore, this code snippet is the Weekday Monitor refactored.

To implement this refactoring, as first step we identify the sensor responsible to get the day of the week, which in this case is the Java API `import java.util.Calendar`, on line 1. Then, we identify all parts related to this sensor. Like the attribute `mCal` (on line 21) which represent the sensor instantiation, located inside of the loader method (`onCreate()`). The last step is identifying the monitoring, preprocess and broadcast logic. The monitoring logic is done by the `mCal.get(Calendar.DAY_OF_WEEK)` on line 31. The preprocess logic is inside the conditional structure (`switch/case`), from line 31 to 39. Finally, the broadcast logic is inside the runnable object `mHandler` on line 44.

After identifying all parts of this monitor, it is possible to move these code parts to the respective place in the template and then set a new polling rate specific for this monitor. Some corrections in the code maybe necessary in order to keep the correct behavior of the system, like initiate this new monitor in the domain application.

```
import java.util.Calendar;
...
public class MonitorWeekday extends IntentService {
    //attributes
    Calendar mCal;
    ...
    public MonitorWeekday() {
        //monitoring logic (initiate the sensor)
        mCal=Calendar.getInstance();     }

    protected void onHandleIntent(Intent intent) {
        while(appOn) {
            //monitoring logic
            weekday=mCal.get(Calendar.DAY_OF_WEEK);
            switch(weekday) {
                case 1: mWeekday="sunday";
                break;
                ...      }
            //broadcast logic
            mHandler.post(new Runnable() {
                public void run() {
                    Intent i=new Intent();
                    i.putExtra(
                    ContextName.WEEKDAY.mWeekday);
                    sendBroadcast(i);    }
            });
            //polling rate
            Thread.sleep(360000);    }
```

Listing 3. Refactoring example - WeekdayMonitor class

VI. Exploratory Study

This section presents an exploratory study to analyse the impact of some maintenance tasks in two versions of the `PhoneAdapter` application; the original one, containing the smells and the refactored one, whithout the smells. The maintenance tasks proposed are listed in Table III. The first column presents the name of the maintenance task and the second column a short description. As can be seen in this table, the tasks we have chosen represent typical modifications demanded in ASs. All of them are devoted to modify or evolve monitor characteristics.

Notice that we are aware that the maintenance tasks do not try to exercise a specific smell, i.e., if there are improvements in the system, we cannot presume which smell had the most impact. That is, the improvements may be because the Obscure Monitor smell or the Oppressed Monitors smell or both was solved.

Next, we provide a more detail explanation of each maintenance task listed in Table III. For each one, we provide an explanation of the scenario in which this task usually occurs, the impact in the original version of the system, the impact in the refactored version and a conclusion.

TABLE III
Maintenance Tasks

Maintenance Tasks	Description
1- Adding new monitor type	A common new system requirement is monitoring a new sensor type.
2- Changing the monitoring logic	A possible maintenance required is change the monitor logic (monitoring, preprocessing and broadcasting logic).
3- Setting different polling rates	A common evolution in AS is setting different polling rates for each monitor optimizing the resources usage.
4- Setting same polling rate	In this task all monitors have to be triggered at the same time with the same polling rate.
5- Defining a execution sequence	In this task a monitor may require information from another one. In this case it is a execution order maybe required.

1) Adding new monitor type. In this task it is required that the system monitors the battery status in order to optimize the device endurance. To add the Battery Monitor the source-code shown in Listing 4 was added.

```
//Monitoring logic (Sensor instantiation)
Intent batteryStatus = context.registerReceiver(null, new
    IntentFilter(Intent.ACTION_BATTERY_CHANGED));

//Monitoring logic
int level = batteryStatus.getIntExtra(BatteryManager.
    EXTRA_LEVEL, -1);
int scale = batteryStatus.getIntExtra(BatteryManager.
    EXTRA_SCALE, -1);
//Preprocessing data
float batteryPct = level / (float)scale;

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

Listing 4. Battery Monitor implementation

Impacts on the Original system: The monitor is added in the `ContextManager` class. The code for sensor instantiation is placed on method the `onCreate()`. The monitoring logic and preprocessing data are placed inside the while loop in sequence with other monitors. At last, a new broadcasting attribute is added in the broadcast container.

Impacts on the Refactored system: A new monitor class is created with the monitor source-code, following the template shown in Listing 2. Then, in the application main class this monitor is instantiated as a parallel process with other monitors.

Conclusion: The changes made in the original system are more invasive, adding new source-code into an existent class. In the refactored system a new class is created being less invasive to the existent code.

2) Changing the monitor logic. In this task it is required that whenever it is thirteenth Friday the Weekday Monitor has to concatenate an "*" in the day of the week. In order to accomplish this task, it was necessary to change the preprocessing logic to the source-code shown in Listing 5.

```
if(mCal.get(Calendar.DAY_OF_MONTH) == 13){
    mWeekday="friday*";
}else{
    mWeekday="friday";
}
```

Listing 5. Changing the weekday monitor preprocessing logic.

Impacts on the Original system: The change is done in the `ContextManager` class inside the while loop. It is necessary to find the Weekday monitor logic and change the "`case`" statement that is about the Friday for the code in the previous listing.

Impacts on the Refactored system: The change is the same as done in the Original system, however it is done in the Weekday Monitor class.

Conclusion: Even though the changes made in both systems are very similar, changes in the Original system are more invasive and may compromise other monitors in the loop. While in the refactored system, changes are exclusively made in one specific class, therefore, being less invasive in the system.

```
protected void onHandleIntent(Intent arg0){
int count = 0;
while(!mStop){
count++;
if(count \% 9 == 0){
    gpsAvailable = mLocListener.mGpsAvailable;
    gpsValue = mLocListener.mLocation;
    gpsSpeed = mLocListener.mSpeed;      }
... }
```

Listing 6. Setting different polling rate

3) Setting different polling rates. In this task, to reduce the battery expenditure it is required to set the GPS Monitor polling rate for 9 (nine) minutes, while the other monitors stay with the same polling rate of 2 (two) minutes.

Impacts on the Original system: The changes are made on the `ContextManager` class. The first modification is set the polling rate for a frequency that allows even and odd minutes values, for example, one minute. Then it is added an attribute to count loop cycles and when the loop is multiple of nine, it triggers the GPS Monitor as it is shown in Listing 6. A similar logic is done with other monitors in order to be triggered every 2 (two) minutes.

Impacts on the Refactored system: In the Refactored system all monitors have a specific polling rate, therefore it is just set the polling rate for 9 (nine) minutes in `GPSMonitor` class.

Conclusion: In this task there is a big difference between both systems. Notice that if other monitors also need a specific polling rate, at least a if statement should be added for each monitor. Another important observation is that the polling rate of the loop has to be adequate for all monitors. That is, if a specific monitor requires a polling rate of 30 (thirty) seconds, it cannot be greater than 30 (thirty) seconds.

4) Setting the same polling rate for all monitors. In this task it is required that all monitors to be triggered just twice a day at the same time.

Impacts on the Original system: The changes are made on the `ContextManager` class. In the line with the polling rate, method *Thread.sleep()*, it is setted a frequency of twelve hours triggering the monitors twice a day. It is important to notice that monitors are not triggered at same time once they have an execution order.

Impacts on the Refactored system: In the refactored system as the monitors are going to be executed twice a day, it is not necessary to keep their thread waiting in the background, therefore, a good solution is to create a monitor management as it is shown in Listing 7.

```
while(!mStop){
Intent gpsMonitor = new Intent(this, GpsMonitor.class);
Intent weekdayMonitor = new Intent(this, WeekdayMonitor.
    class);
...
startService(gpsMonitor);
startService(weekdayMonitor);
...
Thread.sleep(TimeUnit.MILLISECONDS.convert(12, TimeUnit.
    HOURS);)    }
```

Listing 7. Creating a monitor management.

Conclusion: In this scenario the Oppressed Monitors smell is a good design solution because all monitors require to be under the same polling rate, and depending on the frequency the monitor thread can be created and terminated just for the data collection, reducing the resources expenditures. It is important to notice that on the refactored system it is created a Monitor Manager which represents the implementation of the Oppressed Monitors smell.

5) Defining a Monitor execution sequence. This task requires that the GPS Monitor must not execute when battery level is below 50%. Therefore the Battery Monitor has to be executed before the GPS Monitor once its execution relies on the monitored battery data.

Impacts on the Original system: In order to accomplish this task in the Original System, it is necessary to set the Battery Monitor code before the GPS Monitor, then an if statement in the GPS Monitor to verify the battery level.

Impacts on the Refactored system: On the refactored system the Battery Monitor will require a method to trigger its execution every time the GPS Monitor is executed. Therefore, there will be a method call in the GPS Monitor to update the Battery Monitor data.

Conclusion: In this scenario if only some monitors need to be executed in a determined sequence, it is possible to make an hybrid solution. Monitors that must have an execution order can be structured with a triggering method in order to be called before the polling rate cycle.

VII. Related Works

The works related to our research can be divided in two groups. The first deals with the design of monitors and the second one deals with architectural smells. The second group is devoted to show that most of the works in this area do not present smells that are specific to types of systems. Abuseta and Swesi [21] proposed design patterns for control loop components of self-adaptive systems (SAS). The first design pattern called SAS Monitor is related to the monitor component. As the authors describe, it intends to establish the relationships between the components participating in accomplishing the monitoring activity. In the SAS Monitor design, the monitor component is implemented as a first-class entity. Therefore, if a system does not follow a similar design pattern evidencing the monitor it corroborates with the Obscure Monitor smell described in this paper.

Garcia *et al.* [8], [22], describe four representative architectural smells. The four smells decribed are: *Connector Envy*, *Scattered Functionality*, *Ambiguous Interfaces* and *Extraneous Connector*. For each smell, there is a detailed description; the quality impact and trade-offs; and a schematic diagram in order to help architects detect the architectural smells and assess its impact. The authors conclude that architectural smells can be detected either in the conceptual architecture of a software system or in the recovered architecture of an implemented one. These smells are not described for an specific system domain and can be applied to general systems. However, we believe that AS software may present unique features that may require specific architectural smells.

Mo *et al.* [23], propose two hot-spot patterns: the *Unstable Interface* and *Implicit Cross-module Dependency*. They define these hot-spot patterns as "recurring architecture problems that occur in most complex systems and incur high maintenance costs". Therefore, we also consider these hot-spot as architectural smell. In this study, the authors conclude that these patterns identify the most error-prone and change-prone files, and also pinpoint specific architecture problems that may be the root causes of bug-proneness and change-proneness. As validation they report on an industrial case study to demonstrate the practicality of these hot-spot patterns. The architect and developers confirmed that the hot-spot discovered was the majority of the architecture problems causing maintenance pain. As the authors, we believe that in AS there are also architecture designs that may cause maintenance pain and with an analysis of AS it is possible to identify architectural smells that will improve the system inner qualities.

Andrade *et al.* [24], [25] conduct an exploratory study that aims at characterizing bad smells in Software Product Line (SPL) architectures. They extracted the architecture of an open source SPL project and and analyzed it to investigate the

occurrence or absence of the four smells described in Garcia *et al.* [8], [22]. In addition, they propose a smell specific to the SPL context and discuss possible causes and implications of having those smells in the architecture of a SPL. The results indicate that the granularity of the SPL features may influence on the occurrence of smells. Therefore, the authors found the necessity of identifying specific domain architectural smells for SPL, corroborating with our work of identifying architectural smells for AS field.

All these studies corroborate with our paper goals of describing Monitor Smells in AS. In Abuseta and Swesi [21] it shows the necessity of evidence the monitor component as a first-class entity. In Garcia *et al.* [8], [22] and Mo *et al.* [23] it described the presence of architectural smells for general systems. Andrade *et al.* [24], [25] suggest that there are architectural smells for specific domains. To our knowledge, no studies about AS architectural smells were found.

VIII. Discussion and Conclusions

This paper presents two Architectural Smells for ASs. The Obscure Monitor smell expresses the lack of modularization of monitors while the Opressed Monitors indicates the monitors are compelled to have the same polling rate and they must also follow a strict execution order. As they are architectural smells, it is difficult to assign them to a specific code level structure, as occurs with canonical smells as God Class, Long Method and others. Therefore, these smells can be found in either classes or methods, that is, the smells may be implemented spread in different classes or inside a single method in a unique class.

In order to facilitating the search for these smells, some guidelines are presented and also an algorithm that enables finding monitors. The algorithm provided is helpful in identifying both smells. In the Obscure Monitor smell it helps to identify not modularized monitor, while in the Oppressed monitors it helps to identify the monitors and then the presence of a monitor management.

An important detail is that a system that presents Obscure Monitor smell, usually has also the Oppressed Monitors. However, there are also systems where they occur uniquely, i.e., systems that present only the Obscure Monitor or the Oppressed Monitors.

We have also presented a preliminary evaluation whose goal was to characterize the effort of maintaining ASs with and without the smells. In order to perform this analysis, five maintenance tasks were applied in two versions of an AS, an original and a refactored version. Results have shown differences in the maintainability between both versions, being the refactored one easier to be maintained in most of the maintenance tasks.

IX. Acknowledgements

We would like to thank Fapesp (2012/00494-0).

References

[1] M. Hussein *et al.*, "Context-aware adaptive software systems: A system-context relationships oriented survey," Swinburne University of Technology, Tech. Rep., 2010.

[2] R. De Lemos *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[3] Y. Brun *et al.*, "Engineering self-adaptive systems through feedback loops," in *Software engineering for self-adaptive systems*. Springer, 2009, pp. 48–70.

[4] D. Weyns *et al.*, "A survey of formal methods in self-adaptive systems," in *Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*. ACM, 2012, pp. 67–79.

[5] ——, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems II*, R. de Lemos *et al.*, Eds., vol. 7475 LNCS. Springer Berlin Heidelberg, 2013, pp. 76–107.

[6] J. Cámara *et al.*, "Evolving an adaptive industrial software system to use architecture-based self-adaptation," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2013 ICSE Workshop on*. IEEE, 2013, pp. 13–22.

[7] D. G. D. L. Iglesia and D. Weyns, "Mape-k formal templates to rigorously design behaviors for self-adaptive systems," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 10, no. 3, p. 15, 2015.

[8] J. Garcia *et al.*, "Identifying architectural bad smells," in *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE, 2009, pp. 255–258.

[9] U. Kulesza *et al.*, "The crosscutting impact of the aosd brazilian research community," *Journal of Systems and Software*, vol. 86, no. 4, pp. 905–933, 2013.

[10] M. Sama *et al.*, "Context-aware adaptive applications: Fault patterns and their automated identification," *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 644–661, 2010.

[11] M. M. Uwa. Aiasproject. [Online]. Available: https://github.com/maltic/AIASProject

[12] Y. Fan *et al.*, "Smartrac: A smartphone solution for context-aware travel and activity capturing," 2015.

[13] M. K. Rao and K. O. Prakah-Asante, "Driver behavior based vehicle application recommendation," Mar. 1 2016, uS Patent 9,272,714.

[14] V. Souza. The zanshin framework. [Online]. Available: https://github.com/sefms-disi-unitn/Zanshin

[15] G. Biegel and V. Cahill, "A framework for developing mobile, context-aware applications," in *Proceedings of the PerCom 2004.* IEEE, 2004, pp. 361–365.

[16] T. Gu, H. K. Pung, and D. Q. Zhang, "A middleware for building context-aware mobile services," in *VTC 2004 - Spring*, vol. 5. IEEE, 2004, pp. 2656–2660.

[17] L. E. Parker, "On the design of behavior-based multi-robot teams," *Advanced Robotics*, vol. 10, no. 6, pp. 547–578, 1995.

[18] M. A. Serikawa *et al.*, "Guidelines for modularizing the monitor component when refactoring adaptive systems," in *2nd Latin-American School on Software Engineering*. Porto Alegre, Brazil: UFRGS, 2015.

[19] A. J. Ramirez and B. H. Cheng, "Design patterns for developing dynamically adaptive systems," in *Proceedings of the ICSE 2010*. ACM, 2010, pp. 49–58.

[20] P. Horn, "Ibm perspective on the state of information technology - autonomic computing," October 2001. [Online]. Available: http://www.citeulike.org/group/1604/article/1512356

[21] Y. Abuseta and K. Swesi, "Design patterns for self adaptive systems engineering," *arXiv preprint arXiv:1508.01330*, 2015.

[22] J. Garcia *et al.*, "Toward a catalogue of architectural bad smells," in *Architectures for adaptive software systems*. Springer, 2009, pp. 146–162.

[23] R. Mo, Y. Cai, R. Kazman, and L. Xiao, "Hotspot patterns: The formal definition and automatic detection of architecture smells," in *Proceedings of the WICSA 2015*, 2015, pp. 51–60.

[24] H. S. de Andrade, E. Almeida, and I. Crnkovic, "Architectural bad smells in software product lines: An exploratory study," in *Proceedings of the WICSA 2014*. ACM, 2014, p. 12.

[25] H. Andrade, "Software Product Line Architectures: Reviewing the Literature and Identifying Bad Smells," Master's thesis, Malardalen University, Vasteras, 2013.