Evolving and Composing Frameworks with Aspects: The MobiGrid Case

Cidiane Lobato¹  Alessandro Garcia²  Uirá Kulesza¹  Arndt von Staa¹  Carlos Lucena¹

¹Pontifical Catholic University of Rio, Brazil
²Lancaster University, United Kingdom
{cidiane, uira, arndt, lucena}@inf.puc-rio.br, garciaa@comp.lancs.ac.uk

Abstract
Software frameworks enable modular, large-scale reuse by both providing a core architecture addressing recurring concerns in a certain domain and a set of variability options. However, the high volatility of requirements nowadays often imposes a number of framework changes with an architecture-wide impact. In order to avoid the framework design erosion, the modularity and stability of its core architecture implementation must be preserved. With aspect-oriented programming (AOP) promising superior software evolvability, there is a need for verifying its efficacy to enhance or not framework architecture stability. This paper presents a systematic case study where we have compared the evolution of OO and aspectual versions of a code mobility framework, called MobiGrid. Our analysis was driven by the application of heterogeneous evolutionary changes to MobiGrid, such as feature extensions and compositions with a second framework. Our analysis is also rooted at a comprehensive suite of conventional quantitative stability and modularity indicators.

1. INTRODUCTION
Software frameworks [4, 7, 14, 19] represent a common and important technology to implement program families. They enable modular, large-scale reuse through a core software architecture and different variability options to the target applications. However, the framework design gradually evolves to cope with new stakeholders’ concerns [13, 24]. Its longevity is highly dependent on the ability of implementations mechanisms to sustain the framework modularity and stability in the presence of evolving concerns. This is a deep concern to framework engineers as the nature of certain changes is not localized, such as introduction of new broadly-scoped concerns and the composition of the target framework with new frameworks addressing additional concerns. For instance, the implementation of frameworks composition is often intrusive [17] and usage of OO mechanisms and patterns might lead to several undesirable consequences, including invasive changes, ripple effects, and unplugability of the integration code [17, 21].

In this context, it is important to systematically verify the suitability of aspect-oriented programming (AOP) [16] to enhance framework implementation evolvability in scenarios involving: (i) the composition between two or more different frameworks; and (ii) the extension of existing features provided by the frameworks. AOP supports the encapsulation of crosscutting concerns into new modular units – the aspects – through new composition mechanisms, such as pointcut-advice and inter-type declarations. Some works [6, 9, 17, 26] have started to explore the use of AOP to improve the isolation of specific concerns in framework design. However, none of them has analyzed the impact of AOP on framework evolution scenarios. Most of them are either of methodological nature [17, 26] or are not focused on objectively assessing the role of AOP on sustaining framework architecture modularity and reducing observance of ripple effects across the core architecture and optional modules [9, 25].

This paper presents a case study that quantitatively and qualitatively assesses the positive and negative impacts of AOP on a number of widely-scoped framework modifications. Our investigation focused on four evolution requirements of a code mobility framework, called MobiGrid [2], which has been originally implemented in Java. We have systematically observed the evolution of two MobiGrid versions based on respectively Java and AspectJ [15]. Our goal was to observe to what extent AOP mechanisms provide or not enhanced framework modularity and stability in the presence of evolution tasks. Such tasks emerged from the need to satisfy four requirements that are typical in framework evolution scenarios: (1) the isolation of existing mobility-specific concerns in the MobiGrid architecture, (2) a modular introduction of new mobility-specific concerns, (3) an overall enhanced variability of certain tangled and non-tangled concerns in the original MobiGrid implementation, and (4) the composition of MobiGrid with a new communication and mobility object-oriented framework, called JADE [4]. The design stability evaluation of the OO and the aspect-oriented (AO) versions was based on conventional suites of modularity and change impact metrics.

This paper is organized as follows. Section 2 presents study setting issues and an analysis of the framework evolution requirements. Section 3 presents our OO and AO solutions for the framework evolution task. Section 4
presents the measurement results and general analysis. Section 5 discusses limitations of our study and contrasts it with related work. Finally, section 6 presents the concluding remarks.

2. STUDY SETTING

This section discusses the procedures and reasons related to the choice of our case study, the MobiGrid framework. It also discusses its original design and the modularity breakdowns that motivated the requirements for evolving the framework.

2.1 Selection of the Case Study

The case study chosen for assessing the impact of AOP on framework evolution scenarios was the MobiGrid framework [2]. It is a medium-size mobile agent system within a grid environment project called InteGrade [12]. A mobile agent-based system (MAS) consists of a mobility platform and mobile agents being executed on it [4, 19]. In the MobiGrid case, the mobile agent code is defined from the OO framework provided by the Aglets platform [19]. The agents’ mobility strategy encompasses a number of steps, such as its instantiation, initialization, destruction, and migration. The instantiation of a mobile agent in such OO frameworks is made only once when it is created. Every agent receives a unique id and an initial state. Migration is performed each time when an agent arrives at a new host. Destruction means that an agent terminates all its activities and frees all the resources it was using. Migration represents a transfer of an agent from one host to another. In the MobiGrid, agents are used to encapsulate and execute long processing tasks using the idle cycles of a network of personal workstations. The agents can migrate whenever the local machine is requested by its user.

We have selected a case study from the MAS domain because MASs are very complex systems that often require evolution involving more than one framework. For example, because of the recurring openness and heterogeneity requirements of MASs, a MAS framework often needs to be integrated with more than one mobility and communication platforms, such as JADE [4] and Aglets [19]. In fact, since the mobility-specific strategies in MAS often result in a high coupling between the underlying models of mobility platforms and the design of these systems, a number of crosscutting concerns are frequently tangled and scattered over MAS implementation. These problems decrease the modularity and variability of frameworks in general, not only in the MAS domain.

The MobiGrid case involved the composition of two relevant services: (i) the basic grid services necessary to structure and organize tasks to allow their execution in a network of workstations; and (ii) the mobility services responsible to adapt existing services provided by agent platforms to make possible the seamless migration of tasks from a workstation to another one. Several instances of OO framework evolution problems, as discussed in the introduction, were addressed to deal with the composition of these services. In particular, the history of the MobiGrid framework involved a number of widely-scoped refactorings in its original architecture and variability points. The driving force was the need for integrating MobiGrid with Aglets, JADE and other platforms. This required to maximize the independence of its core design from specific platforms and APIs. In order to achieve this goal, a set of MobiGrid evolution requirements were derived:

1. modularization of the code mobility, that is, an explicit separation between the MobiGrid mobility concerns and the other framework concerns;
2. a modular introduction of mobility capabilities into the MobiGrid stationary agents;
3. an enhanced MobiGrid variability in terms of a flexible choice of distinct mobility platforms;
4. a modular handling of typical framework composition issues [21], such as the overlapping of MobiGrid functionalities with those from the OO frameworks provided by the Aglets and JADE.

The next sections discuss a relevant slice of the framework original design (Section 2.2), and the challenges associated with satisfying the four evolution requirements mentioned above (Section 2.3).

2.2 The MobiGrid Original Design

Figure 1 presents a partial view of the MobiGrid original design. The TaskAgent and the TaskState classes allow the specification of a task and its state, to be moved together with an Aglets mobile agent. The TaskAgent class extends the abstract Aglet class, which is the class for mobile agent specification on the Aglets platform.

The ManagerAgent class allows the instantiation of a manager, which is a stationary agent that manages references to tasks in a client machine in the grid. The manager exchanges messages with tasks during registering, migration and cloning scenarios. The Server class represents an agent server that is installed on each grid machine to provide Aglets resources to MobiGrid. It is associated with the AgletRuntime and AgletContext objects, which provide an execution environment for managers and tasks. When a machine is requested by its user, the server asks the evacuation of the local tasks. These tasks query their managers, which communicate with InteGrade infrastructure looking for idle machines to which they may be dispatched. The proxies, ids and listeners are Aglets-specific mobility issues used in the MobiGrid. The proxies (AgletProxy) allow messaging among tasks, managers and servers; the ids (AgletId) identify tasks, managers and servers; and the listeners trigger specific procedures at special moments (e. g. after migration).

2.3 The MobiGrid Modularity Breakdowns

From the description above, we can observe that the Aglets framework imposes architectural restrictions on the MobiGrid design. In order to introduce the mobility capabilities, developers must: (1) declare that the agent classes extend specific Aglets API classes, (2) implement the ab-
stract methods, (3) specify the implementation of the interfaces, and (4) explicitly invoke the Aglets mobility methods on the MobiGrid classes, which are not created to address mobility concerns. Thus, the usage of the Aglets does not allow the reuse and explicit handling of agents’ mobility strategy as variability points. MobiGrid is also platform dependent and it can not rely on the JADE framework [4] as an alternative to implement the mobility capabilities. This variability is required due to the different Aglets and JADE characteristics, which are relevant according to specific application demands. Finally, the lack of proper modularization also makes the composition of MobiGrid and Aglets with infrastructures addressing other concerns more difficult, such as learning and role-oriented coordination of multiple agents.

In fact, although part of the mobility concerns are localized in Aglets mobility-specific classes, mobility-specific code replicates and spreads across core classes in the MobiGrid design. The strong dependence between the MobiGrid and Aglets platform is mainly caused by the excessive use of inheritance relationships between the classes providing the mobility (Aglet framework) and the basic grid services (MobiGrid framework). Figure 2 represents each modularity breakdown with a number surrounded by a circle. The use of inheritance to incorporate the mobility capabilities also results in code replication as well as code tangling and scattering (problem ⑦). The agent classes also need to hold explicit references to Aglets mobility elements (e.g., aglet ids and proxies) as attributes (problem ⑧). These classes also manage these elements by Aglets-specific mobility methods (problem ⑨). As a consequence, the basic functionalities, the context-specific services, and the messaging are amalgamated to mobility methods.

Certain classes also have to implement the Serializable interface for allowing the objects, which are part of the agent, to be moved across hosts (problem ⑩). The Serializable interface is just a representative example: OO APIs from platforms usually provide a number of interfaces with methods that are implemented by systems in order to allow actions to be automatically executed at specific moments through the mobile agent lifecycle.

For instance, MobiGrid procedures are automatically executed immediately after cloning (TaskCloneListener interface) or just before migration (TaskMobilityListener interface).

![Figure 1. The MobiGrid Original Design](image)

A single method implements the conditions governing on when a task should move (evacuate()). There would be no problem with it, except for the fact that in evolution scenarios, the code related to migration decision is replicated on several agent type methods. These methods contain mobility-specific code relative to contextual decisions on when an agent should move to a remote environment (problem ⑪), or when should go back to the home location (problem ⑫). In addition, there can be a spread of usual preconditions and post-conditions when an agent moves to another host (problem ⑬).

Beyond the architectural restrictions on the agent design, the pervasiveness of mobility frameworks also introduces conceptual problems in the definition of agents and its potential roles. For example, the ManagerAgent class has to extend the Aglet class even this agent type corresponding to a stationary one. The Aglet class is required for using the platform’s communication capabilities. All these problems decrease the system evolvability, since adding or removing the mobility code from classes requires invasive changes in those classes.
3. EVOLUTION STEPS

Due to the high coupling between the Aglets and MobiGrid, we have decided to evolve this framework taking into account the requirements listed in Section 2.1 and described in Section 2.3. In the refactoring process, we have generated two versions of the MobiGrid architecture. In a first moment, our goal was to assess the applicability of OO mechanisms and patterns in order to explicitly separate the mobility concern (Section 3.1). After that, we provided an alternative version using AOP (Section 3.2) with the aim of improving the separation of mobility crosscutting concerns and integration between MobiGrid and distinct mobility platforms. Next subsections present, respectively, the OO and AO resulting versions from refactoring and evolving the MobiGrid original framework.

3.1 The OO Refactored Version of MobiGrid

Figures 3 and 4 illustrate the architectural design and detailed design of the OO refactored version of the MobiGrid framework. Both follow a simplified UML notation. The MobiGrid architecture, illustrated in Figure 3, is now composed of four component types: (1) MobiGrid encapsulates the basic concerns of an agent-based application, including knowledge base, and autonomy, adaptation, and autonomy-specific behaviors; (2) MobilityProtocol modularizes the concerns relative to the agents’ mobility strategies; (3) MobilityManagement provides the integration between MobiGrid and distinct platforms; and (4) MobilityPlatform represents a specific platform.

The IMobileAgentProtocol interface from the MobilityProtocol component delegates to IReferenceManager from the MobilityManagement component, the mobility services invoked by the MobiGrid; this delegation is independent from platform-specific issues. The IMobileAgent is the interface which is responsible for delegating to a specific platform the invoked services; to do that, it uses the IPlatformServices interface provided by the MobilityPlatform component.

Alternative interfaces are used in order to provide the integration between MobiGrid and distinct platforms, represented by the MobilityPlatform component. Different mobility services provided by many interfaces of the MobilityPlatform component are invoked by the MobilityManagement component. The IReferenceTable interface, for instance, is used to abstract the context and messaging services provided by different platforms.

The MobilityProtocol interfaces modularize the mobility-specific concerns using different interfaces. These interfaces, illustrated in Figure 3, implement the OO Observer pattern [8] in order to make the information relative to the agent lifecycle available.

Figure 4 presents the design elements which were used to integrate the MobiGrid with distinct platforms in the first step of the refactoring process. These elements correspond to the MobilityManagement implementation. Note that the proposed structure can be visualized as an OO platform framework. The hot spots (or variability points) corresponding to each platform are as follows:

- **ReferenceManager** class is responsible for (1) the reference table instantiation and its update when mobile agents are instantiated, initialized or destructed, and (2) response to common requests, such as getting the agent list in a specific context. It is a stationary agent that is a singleton.

- **ReferenceCreator** class is responsible for (1) the context creation for agent execution at a specific host, (2) the agent instantiation on this context, and (3) the starting of services for instantiated agents. This class is implemented as a singleton for each context.

- **MobileAgent** class is responsible for delegating to a specific platform the agent services provided by its...
interface, including methods, such as move(). This class communicates with the ReferenceManager in order to reply to common requests, such as getting the agent list in a specific context.

- Event class allows detection of relevant platform-specific join points, such as the agent initialization.

MessageParser class executes parsing of platform-specific message formats.

Beyond the MobilityManagement design, Figure 4 shows that the MobileElement interface allows MobileAgent objects to represent the MobiGrid mobile elements, such as the TaskAgent objects. In addition, the EventListener interface makes the information relative to a mobile element’s lifecycle available to MobiGrid. The MobileElement and EventListener interfaces correspond to the implementation of the MobilityProtocol component. Note that these elements can be visualized as an OO application framework. The framework hot spots correspond to the procedures that a mobile element can execute at special stages of an agent lifecycle.

3.2 The AO Refactored Version of the MobiGrid

In a second step of the refactoring process, we worked on the definition of the aspect-oriented version of the framework design with the same goals in mind (Section 2.1). Figures 5 and 6 show, respectively, the architectural design and the detailed design of the AO MobiGrid version. We applied the patterns used in the MobiGrid refactoring based on the OO mechanisms, but now implementing the Observer [8] in an AO version. More specifically, we implemented the MobilityManagement and the MobilityProtocol components and their strategy relative to the platform event propagation taking into account the AOP. This action resulted in the implementation of the IReferenceObserver and the MobilityProtocol event observers, such as the IInitializationEvent interface, as crosscutting interfaces (Figure 4). In other words, the IReferenceObserver interface crosses join points as calls of mobility platform services in order to maintain a consistence with the platform runtime. This is possible because the IReferenceObserver interface is also affected by the interfaces that allow the agent instantiation, departure, arrival, and destruction handling in the MobilityProtocol component. Note that this implementation contrasts with the IReferenceObserver and the MobilityProtocol interfaces in the OO MobiGrid refactoring (Figure 3), where observers are implemented following an OO version of the Observer pattern [8]. This architectural decision aims to decrease the strong coupling between MobilityPlatform, MobilityManagement and MobilityProtocol components existing in the OO version.

In addition, in order to explicitly separate the MobiGrid mobility-specific concerns, we implemented the MobilityProtocol component in an AO version. This action resulted in the IMobileElement implementation as a crosscutting interface (Figure 5). This interface is used to determine when and how a mobile agent is instantiated on a platform to represent a MobiGrid-specific agent. It is also used to affect mobility join points in order to determine when agents should move. Figure 6 presents a partial view of the MobilityProtocol design using AOP. It corresponds to the abstract Mobility aspect and its subsaspects. The abstract Mobility aspect contains pointcuts which capture the instantiation, migration, initialization and destruction of mobile agents. It also contains the advice associated with these pointcuts. The advices are implemented following an abstract pattern: mobility-relevant events are picked out, conditions are checked and the appropriate mobility-specific methods are invoked. For example, the migration_() after advice is responsible for checking the need for the agent roaming and for calling the mobility platform actions, such as the doAfterArrivalHost() method.

Note that the Mobility abstract aspect defines a series of variabilities, such as: (1) the elements to be defined as mobile or serializable (using the declare implements statement), (2) the instantiation abstract methods, (3) the initialization abstract methods, (4) the migration decision points implemented as abstract pointcuts, and (5) the definition of which objects will be moved together with the mobile element. Therefore, in order to introduce mobility-specific concerns into a stationary agent, the application designers only have to specify the Mobility hot spots for each element with mobility requirements: (1) for each class with mobility requirements (e.g. TaskAgent in Figure 6), we specify a Mobility subsaspect and declare that this class implements the MobileElement interface through an inter-type declaration; and (2) we make concrete the Mobility interface for each Mobility subsaspect (we specify the instantiation and migration pointcuts and the methods corresponding to procedures invoked on the agent advice (e.g. doAfterArrivalHost()).

4. RESULTS AND LESSONS LEARNED

This section presents primarily a general analysis (Section 4.1) of the OO and AO versions we have implemented in the refactoring process of the MobiGrid architecture (Section 3). Then we analyze a suite of quantitative modularity and stability indicators obtained from the measurements on the two refactored versions. The metrics are complementarily applied to
both design and implementation elements that were modified during the changes. The modularity metrics quantified classes and packages changed in the implementation artifacts (Section 4.2). The stability measures focused on the coarse-grained elements of the OO and AO architectural representations in order to capture major instabilities in both architecture designs (Section 4.3).

4.1 General Analysis

Core Architecture Instability. First, both designs, even following different implementations for the MobilityManagement component, have provided the integration between MobiGrid and distinct mobility platforms (Section 2.1). The evolution requirement related to an enhanced MobiGrid variability in terms of a flex-

Figure 3. The OO MobiGrid Architecture

Figure 4. The Integration of the MobiGrid with Distinct Platforms

Figure 5. The AO MobiGrid Architecture
ible choice of distinct mobility platforms is thus addressed in both OO and AO implementations. However, the OO refactored version (Figure 3) did not promote an explicit modular treatment of the MobiGrid mobility concerns (Section 4.2). Even though separation of mobility concerns is achieved to some extent in the OO version, it leads to a significant increase on coupling between the MobiGrid and OO mobility frameworks. This limitation also hinders a modular introduction of mobility capabilities into the MobiGrid agents, thereby causing some instability of the MobiGrid core implementation in the presence of such a change.

In this perspective, our study involved a composition of two calling frameworks. As a consequence, it makes varying the adopted platform more cumbersome.

**Composing Calling Frameworks.** On the other hand, the overlapping of MobiGrid functionalities with those from the MobilityProtocol component also affects the handling of typical framework composition issues [21]. In this perspective, our study involved a composition of two calling frameworks. Calling frameworks [21] are responsible for controlling and invoking application modules, as opposed to called frameworks, which are passive entities that are called by other application parts. When MAS frameworks (e.g., MobiGrid), are combined with OO platform calling frameworks (e.g., Aglets) in order to implement the agent's mobility capabilities, MASs reuse the mobility functionality from these frameworks through inheritance relationships, which brings a strong coupling between the frameworks composed. As a consequence, it makes varying the adopted platform more cumbersome.

**Aspects as Framework Glue.** To overcome this problem of composition of the calling frameworks, the AO MobiGrid architecture used the AO implementation of the Adapter pattern [8] in order to expose the services of different mobility platforms to the MobiGrid component (Figure 4). The IMobileElement interface represents this adapter element in the AO design (Figure 4) and it makes the use of a mobility calling framework possible as if it was a called framework. In this way, the Mobility aspect (Figures 4 and 6) can work as a modular bridge between the MobiGrid and the OO platform frameworks (not only the Aglets one), by modularizing all the mobility code and calling the mobility services using the MobileElement interface. As a consequence, the pluggability of the inter-framework composition code is more effectively addressed in the AO version. This observation is supported by the modularity measures discussed in the next section.

**4.2 Modularity Analysis**

Table 1 presents a brief definition of each metric we have applied and associates them with the attributes measured by each one. They are traditional modularity metrics for coupling, cohesion, size, and separation of concerns (SoC), and their in-depth discussion is out of this paper scope; refer to [5, 10] for further details about the metrics. Their choice was driven by the fact they have already been used in several experimental studies and proven to be effective quality indicators (e.g., [5, 9, 10, 18]). Table 2 presents the collected data, and the results are discussed in the following.

**Unavoidable Mobility Diffusion in the Core Framework Architecture.** The SoC measures were obtained for quantifying the separation of the mobility concern, since it was the main target in the four evolution requirements (Section 2.2). The number of components (CDC) implementing mobility in the OO version is 7. This number significantly decreases to 3 if we use aspects to modularize the mobility concerns.

The inferiority of the OO version is granted to three main reasons: (1) agent basic classes are part of the core framework architecture, but they present a high degree of mobility-specific tangling and scattering; (2) without aspects, there is a need for event listeners triggering mobility-specific actions – the use of pointcut-advice eliminates this need in the AO version; and (3) the use the OO framework provided by a platform forces the implementation of additional classes (hot spots) to manage information relative to the platform runtime. Even though the OO architecture has achieved a considerable separation of mobility behaviors, the number of mobility-specific operations (CDO) and transition points (CDLOC) are also lower in the AO version. The reason is that the later has significantly enhanced the localization of code elements represented as $\odot_1$, $\odot_2$, $\odot_3$, $\odot_4$, $\odot_5$, and $\odot_6$ in Figure 2.

**Additional Inter-Component Dependencies.** The use of aspects decouples the mobility concerns from the agent basic functionalities; each of the mobility and basic concerns can be associated with other components in a more condensed way. In fact, it can be observed through the CBC measures that are significantly
lower in the AO MobiGrid version. Some of the extra dependencies in the OO version are localized in the elements that implement the “glue” for composing MobiGrid with Aglets, JADE or alternative platforms. DIT also makes it explicit that the OO platform frameworks inevitably rely on OO mechanisms, such as inheritance and delegation, thereby highlighting the gravity of architectural restrictions discussed in Section 2.3. In the AO framework design, the AO mechanisms (Sections 3.1 and 3.2) minimizes the overuse of inheritance, and reduces coupling of crosscutting behavior that should be exposed as a framework’s variability point; the use of inheritance is restricted to the framework users’ hierarchies addressing non-crosscutting features. In other words, the use of AOP allows for more flat hierarchies instead of flat hierarchies.

Table 1. The Modularity Metrics Suite

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concern Diffusion over Components (CDC)</td>
<td>Classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them.</td>
</tr>
<tr>
<td>Concern Diffusion over Operations (CDO)</td>
<td>Methods and advice whose main purpose is to contribute to the implementation of a concern and the number of other operations that access them.</td>
</tr>
<tr>
<td>Concern Diffusions over LOC (CDLOC)</td>
<td>Transition points for each concern through the lines of code. Transition points are points in the code where there is a “concern switch”.</td>
</tr>
<tr>
<td>Coupling Between Components (CBC)</td>
<td>Other classes and aspects to which a class or an aspect is coupled.</td>
</tr>
<tr>
<td>Depth Inheritance Tree (DIT)</td>
<td>How far down in the inheritance hierarchy a class or aspect is declared.</td>
</tr>
</tbody>
</table>

Table 2. The Modularity Measurement Results

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Metric</th>
<th>OO</th>
<th>AO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concern Diffusion over Components (CDC)</td>
<td>7</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Concern Diffusion over Operations (CDO)</td>
<td>50</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Concern Diffusions over LOC (CDLOC)</td>
<td>11</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Coupling Between Components (CBC)</td>
<td>34</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>Depth Inheritance Tree (DIT)</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Vocabulary Size (VS)</td>
<td>20</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Lines of Code (LOC)</td>
<td>699</td>
<td>706</td>
<td></td>
</tr>
<tr>
<td>Number Attributes (NOA)</td>
<td>26</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>Weighted Operations per Component (WOC)</td>
<td>74</td>
<td>201 (78)</td>
<td></td>
</tr>
</tbody>
</table>

Aspect Interface Complexity. LOC and NOA metrics show that AO and OO MobiGrid implementations are similar in the number of lines of code and attributes of each component. However, the most interesting size measures are the ones quantifying the complexity of operations’ interface. If we consider the operations introduced by a mobility aspect (hot spot) into an application’s mobile element, WOC is 201; this number is 78 if we consider only the non-aspect elements, such classes (in this case, the complexity is similar to the OO version). As discussed before (Section 3), the framework design concentrates the mobility services abstracted from platforms into an interface that has to be implemented by all mobile elements. Whenever an intertype declaration is used for introducing the mobility capabilities into the otherwise stationary agent, 31 units should be added to the WOC.

4.3 Stability Analysis

Our architecture-level stability analysis relies on a suite of typical change impact measures [25], such as number of components added, changed or removed, number of operations added, changed or removed, and number of relationships (via inheritance and reference) added, changed or removed. The purpose of using these metrics is to quantitatively assess the architectural propagation effects when applying the various evolution steps. For these metrics, the lower the impact measures the more stable and resilient the design is to a certain change. Table 3 presents the collected data after we have applied the suite of metrics to both changed OO and AO architectural representations (Figures 3 and 4).

More Operations Changed in the OO Design. We can observe that the number of components is the same for both OO and AO architectures: the MobilityManagement and MobilityProtocol components are added, and the MobilityPlatform and MobiGrid components remain. On the other hand, the number of added
and changed operations is significantly different for the OO and AO versions of the MobiGrid refactoring. This number is higher in the OO version for a main reason: without aspects, there is need of a separate pair of interfaces for event propagation in both Mobility Management and MobilityProtocol components in the OO MobiGrid refactoring (Figure 3). It results from the implementation of the Observer pattern [8] with more classes than in an AO version.

<table>
<thead>
<tr>
<th>Table 3. The Stability Measurement Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric</td>
</tr>
<tr>
<td>Added Components</td>
</tr>
<tr>
<td>Changed Components</td>
</tr>
<tr>
<td>Removed Components</td>
</tr>
<tr>
<td>Added Operations</td>
</tr>
<tr>
<td>Changed Operations</td>
</tr>
<tr>
<td>Removed Operations</td>
</tr>
<tr>
<td>Added Relationships</td>
</tr>
<tr>
<td>Changed Relationships</td>
</tr>
<tr>
<td>Removed Relationships</td>
</tr>
</tbody>
</table>

OO and AO Architecture Ripple Effects. In addition, the resulting number of changed relationships reflects our general analysis of the OO and AO versions implemented in the refactoring process (Section 4.1). Using AOP, it is possible to effectively decouple the mobility-specific concerns from the MobiGrid through the use of a single interface, the IMobileElement in the MobilityProtocol component (Figure 4). In this case, the IApplicationAgent interface is changed to not implement the inheritance relationship. In contrast, this and the other original relationships are also present in the OO refactored version, even with changes implemented in order to not refer to a specific platform. In the OO version, the platform models still influence the MAS design, such as the MobiGrid. Thus, the AO refactored version is more resilient to changes than the OO version. However, note that the removal of operations and relationships can be visualized as an introduction of instability on the AO MobiGrid design as well. We have noticed that some of these side effects tend to occur in the AO design when the focus of a change is a non-crosscutting concern; removal of non-crosscutting behavior might cause non-obvious interference with the specification of pointcuts in the aspects.

5. RELATED WORK

Only a few related works that discuss framework evolution and composition issues have been published. Gurp & Bosch [13] present a set of practical guidelines to build OO frameworks, which aims to improve their evolvability. Many of their guidelines nowadays represent common practice in industry, such as to design role-oriented framework interfaces and the use of role inheritance to combine different roles. Besides their advantages, we believe the adoption of these guidelines is not sufficient to deal with many problems of framework evolution, such as the presence of optional features [3, 20] and the existence of crosscutting composition between frameworks; it was in turn a key issue in the MobiGrid refactorings.

Mattsson et al. [21] have analyzed the problems and causes that occur in OO frameworks compositions and proposed several OO solutions. In a recent study, Kulesza et al [17] have revisited the Mattsson et al's study to evaluate the modularity properties of framework composition OO solutions proposed by them. Their analysis [17] has shown that 6 out of 9 solutions described by those authors have poor modularity and a crosscutting nature, requiring invasive internal changes in the framework code. It reinforces the motivation for investigating aspect-oriented programming to modularize crosscutting framework variabilities and code for framework composition. However, Kulesza’s study is not focused on framework evolution and modularity assessment in the presence of changes. On the other hand, our study quantified the modularity and stability improvements brought by the AO implementation of the MobiGrid. Moreover, our investigation allowed us to understand how AO modularization of the crosscutting composition code in brought the additional benefit of better modularizing the variability of using different mobility platform frameworks.

Recent research work [17, 20, 22, 23] on the development of software family and product line architectures has addressed the decomposition of flexible architectures into features. Feature-oriented approaches (FOAs) have been proposed [23] to deal with the encapsulation of program features that can be used to extend the functionality of existing base program. Batory et al. [3] argue the advantages that FOAs have in respect to OO frameworks to design and implement product-lines. Mezini and Ostermann [22] have identified that FOAs are only capable of modularizing hierarchical features. They do not support the specification of crosscutting features. They propose CaesarJ [22], an AO language that combines ideas from both AspectJ and FOAs, to provide a better support to manage variability in product lines. More recently, Apel & Batory [1] have proposed the Aspectual Mixin Layers approach to also allow the integration between aspects and refinements. They propose: (i) to use aspects to modularize crosscutting features in the implementation level; and (ii) to use mixins/refinements to structure classes and aspects in a set of architectural components.

Recent research work [17, 20, 22, 23] on the development of software family and product line architectures has addressed the decomposition of flexible architectures into features. Feature-oriented approaches (FOAs) have been proposed [23] to deal with the encapsulation of program features that can be used to extend the functionality of existing base program. Batory et al. [3] argue the advantages that FOAs have in respect to OO frameworks to design and implement product-lines. Mezini and Ostermann [22] have identified that FOAs are only capable of modularizing hierarchical features. They do not support the specification of crosscutting features. They propose CaesarJ [22], an AO language that combines ideas from both AspectJ and FOAs, to provide a better support to manage variability in product lines. More recently, Apel & Batory [1] have proposed the Aspectual Mixin Layers approach to also allow the integration between aspects and refinements. They propose: (i) to use aspects to modularize crosscutting features in the implementation level; and (ii) to use mixins/refinements to structure classes and aspects in a set of architectural components.
that address the product line features. These authors have also used size metrics to quantify the number of components and lines of code associated with mixins and aspects in the implementation of a product line of overlay networks. Their case study, however, has not considered a significant suite of software metrics and did not address evolution scenarios and stability metrics.

Zhang and Jacobsen [26] propose the Horizontal Decomposition method (HD), a set of principles guiding the definition of functionally coherent core architecture and customizations of it. The core is customized with aspects implementing orthogonal functionality. The authors have applied systematically their approach in the AO refactoring of a middleware [26] and the Prevayler persistence framework [11]. Only the Prevayler refactoring study developed by these authors has applied software metrics to compare the modularity of the AO and OO framework implementations. However, they have not also addressed evolution scenarios and stability metrics on their case studies. They do not also investigate framework composition scenarios.

6. CONCLUSION

The high volatility of requirements nowadays imposes a number of evolution tasks with a crosscutting impact on the framework’s core and optional modules; changes in the crosscutting concerns should have a minimal impact on the implementation of the other concerns and vice-versa. This paper presented a case study where we have compared AO and OO solutions to satisfy certain types of requirements, which are recurring in realistic framework evolutions.

We have observed that the AO version improved the modularity and stability of the MobiGrid architecture for most of the change scenarios. The core architecture of the AO framework allowed for enhanced separation of mobility-specific concerns that typically cut across framework classes. In addition, it allows the introduction of new crosscutting concerns in a transparent way to the other framework concerns.

AO design also reduces the intrusiveness of the code used to compose two or more frameworks. The application of conventional suites of metrics quantitatively pointed out that the AO architecture was more modular and resilient to changes, especially when the target is a crosscutting concern. However, some instabilities were observed in the AO design when operations and relationships realizing non-crosscutting concerns needed to be changed.

7. REFERENCES


