



Architecture-driven assistance for fault-localization tasks

Álvaro Soria, J. Andrés Díaz-Pace and Marcelo R. Campo

ISISTAN Research Institute, Faculty of Sciences, UNICEN University, Campus Universitario, (B7001BOO) Tandil, Buenos Aires, Argentina

E-mail: asoria@exa.unicen.edu.ar

Abstract: Finding software faults is a problematic activity in many systems. Existing approaches usually work close to the system implementation and require developers to perform different code analyses. Although these approaches are effective, the amount of information to be managed by developers is often overwhelming. This problem calls for complementary approaches able to work at higher levels of abstraction than code, helping developers to keep intellectual control over the system when analyzing faults. In this context, we present an expert-system approach, called FLABot, which assists developers in fault-localization tasks by reasoning about faults using software architecture models. We have evaluated a prototype of FLABot in two medium-size case studies, involving novice and non-novice developers. We compared time consumed, code browsed and faults found by these developers, with and without the support of FLABot, observing interesting effort reductions when applying FLABot. The results and lessons learned have shown that our approach is practical and reduces the efforts for finding individual faults.

Keywords: software architecture, fault analysis, tool support, Use-Case Maps, Java

1. Introduction

Finding and repairing software faults is typically a tedious, expensive and time-consuming activity, not only during the development of a system but also during its further maintenance and evolution. The effectiveness of fault localization depends on both developers' expertise and appropriate tool support. In particular, developers need to understand the structure of the software system under analysis and make judgments about suspicious regions of the code (i.e. code that is likely to contain faults). Unfortunately, the complexity of current applications still makes it difficult for developers to fix bugs properly and in time. Furthermore, factors such as the high mobility of developers and a very competitive software market aggravate the problem, particularly in small organizations that are more likely to suffer from 'organizational amnesia' (Kransdorff, 1998; Othman & Hashim, 2004).

Over the last years, many efforts have been spent on testing and debugging approaches to detect errors or unexpected behaviours in software applications (Korel & Laski 1990; Lieberman, 1997; Myers *et al.*, 2004; Zeller, 2006). Several tools, such as *FINDBUGS*¹ or *QJ-PRO*,² have become popular in development environments. All these automated approaches are good at detecting errors in code sentences, but the intervention of the developer is still necessary in the process, because she has to reason about potential causes for an error and pinpoint suspicious code regions. For example, when a programmer tries to fix an

error reported by a code analysis tool on a given module, the faults may come from modules with no apparent links to the inspected module. In this context, tools relying solely on code analysis do not help developers to manage the cause-effect relationships between errors and faults. As another example, these tools have trouble to deal with faults that are the result of missing code (Wong & Debroy, 2009) (e.g. missing an assignment or a conditional statement by error can result in certain critical statements to be unexecuted). Even in medium-size code bases, the information needed for a fault diagnosis can impose a high cognitive load on the human performing the analysis. In practice, these situations are often solved using expert knowledge, either about the system structure or typical fault patterns.

From a usability perspective, we argue that a developer should have a 'high-level' debugging tool able to provide her with educated guesses about code regions where faults are likely located. We see two main benefits in this fault-localization approach. Firstly, it would permit a manageable exploration (and prioritization) of the application code based on its underlying design structure. Secondly, once the developer identifies a suspicious region of the system as relevant, it would permit a deeper investigation of the suspicious code by means of standard debugging tools.

We experienced the fault-localization problems above during consulting work for a software company that developed a workflow management framework (Campo *et al.*, 2002). The framework was designed following an event-based architectural pattern (Buschmann *et al.*, 1996) because it offered good flexibility to meet the business goals of the company. On the downside, the applications built on top of the framework were very hard to debug, because the order of activation of components cannot be known a priori.

¹FindBugs homepage. <http://findbugs.sourceforge.net/>

²QJ-Pro homepage. <http://qjpro.sourceforge.net/>

Despite the use of code inspections, error logs and debugging tools, the application developers spent a lot of time to reproduce ‘component activation patterns’ leading to particular errors. Given this situation, developers relied on the judgments of expert architects. These architects usually pointed out faults in components based on their knowledge of the framework design, rather than on a deep analysis of code or error logs. This reasoning process is a form of root cause analysis (Agarwal *et al.*, 2004; Andersen & Fagerhaug, 2006), in which an expert maps error symptoms to high-level components and checks the communications of those components with other components in order to figure out parts of the design or code where faults might be hidden. The expert architects selectively inspected system logs (or other software artefacts) to make hypotheses about the presence or absence of faults in certain components. Yet more interestingly, we observed that the software architecture design worked as a ‘mental model’ of the system on which the architects performed their analysis. On the basis of this architectural metaphor, we have developed a tool called *FLABot* (Soria *et al.*, 2009) to assist developers in finding faulty functions in object-oriented applications.

The approach proposed by *FLABot* relies both on an architectural model of the system, documented with Use-Case Maps (UCMs) (Buhr, 1998), and on knowledge rules for exploring the code as guided by these UCMs. Architectural models provide a high-level view of the system, reducing the complexity of understanding what the system is doing and supporting the fault reasoning process (Casanova *et al.*, 2011). Initially, the architect specifies the main design structure and functional scenarios of her application using UCMs and also maps those UCMs to the application implementation (Java code). These inputs are fed into the *FLABot* tool. At debugging time, the developer selects a responsibility that she perceived as problematic and asks the *FLABot* tool to explore other responsibilities in the UCMs that may ‘explain’ the problems. A key UCM concept for *FLABot* is that of *architectural responsibility*, as a coarse-grained piece of functionality (e.g. a function, activity or action) carried out by a software component. Internally, *FLABot* is equipped with a heuristic algorithm that detects error states in related responsibilities. Some parts of this algorithm can be customized with expert knowledge. To assess error states, *FLABot* collects runtime information from system executions and applies a rule-based filtering schema. As output of its analysis, *FLABot* suggests a set of responsibilities that are possible causes of the error to the developer. Furthermore *FLABot* inserts breakpoints in the code for that set of responsibilities, allowing the developer to continue inspecting the application using a standard (code-level) debugger. A novel aspect of our assistive approach is that it raises the abstraction level of fault-localization tasks to a level where the developer can analyze faults in terms of the architectural structure of the system. Thus, *FLABot* complements existing code-level tools and makes the fault-localization process more effective.

The main concepts of *FLABot* and a proof-of-concept were described in prior work (Soria *et al.*, 2009). In this article, we provide an in-depth description of the *FLABot* tool infrastructure and elaborate on the fault-localization

heuristic and the knowledge rules used by the current prototype. We also report on an additional case study and lessons learned using the tool. These findings have corroborated our initial results, showing that the approach can certainly reduce the developer’s efforts spent on finding candidate faults (in combination with existing debugging tools).

The rest of the article is organized into five sections. Section 2 explains the kind of assistance provided by *FLABot* through a motivating example. Section 3 describes the main components of the *FLABot* infrastructure. Section 4 discusses the case studies and lessons learned. Section 5 covers related work. Finally, Section 6 presents the conclusions of the work.

2. Using architectural information to approximate faults

Fault localization is the activity of identifying the locations of faults in software programs. In this work, we use the dependability terminology given in (Laprie & Randell, 2004). An error is a system state that might cause a failure. A failure occurs when some functionality delivered by the system deviates from the expected functionality. In this context, a fault is the cause of an error in the system. As pointed out in (Wong & Debroy, 2009), fault localization can be divided into two parts: (a) the identification of code that is suspicious of containing faults; and (b) the actual code examination to decide whether it contains faults. The *FLABot* approach deals mostly with the first part.

A premise for *FLABot* is that expert architects normally use the *software architecture* of a system as an aid for performing cause-effect reasoning about system faults (Burkhardt *et al.*, 1998). The software architecture comprises the high-level structure(s) and the key design decisions of a system (Bass *et al.*, 2003). Three major benefits of software architecture are the following: (a) it helps organizations to maintain intellectual control over the software; (b) it supports communication among stakeholders; and (c) it permits early analysis of quality attributes (e.g. availability, modifiability, etc.). In practice, the software architecture is captured by an architecture specification (or architecture documentation), which describes the main components, coarse-grained functions, interactions among them and design constraints (Clements *et al.*, 2002). From this perspective, the system implementation must adhere to the architectural design prescriptions. That is, the implementation should be divided into the elements prescribed by the architecture; these elements must interact with each other in the prescribed fashion, and each element should fulfil its responsibilities to the others as dictated by the architecture (Bass *et al.*, 2003).

Figure 1 shows the context of the *FLABot* tool, how the software architecture information drives the fault-localization process and the articulation of *FLABot* with standard debugging tools. We distinguish two phases: (a) the architecture definition phase; and (b) the debugging phase. The first phase is a preparation for the second phase, in which the assistance takes place. In the architecture definition phase, the architect’s first task is to document the architecture of the system. The Unified Modelling

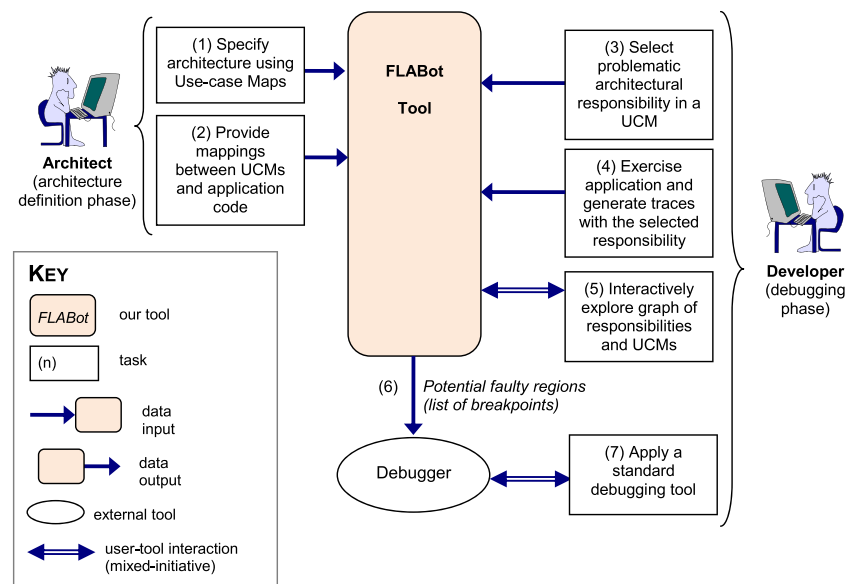


Figure 1: Main actors and tasks in the FLABot approach.

Language (UML) notation appears to be a natural choice for this matter. Nonetheless, it has been argued (Buhr, 1999; Bordeleau & Cameron, 2000) that UML behavioural diagrams (e.g. sequence, collaboration or activity diagrams) are suitable for describing detailed interactions (such as messages exchanged between objects), and we are instead interested in high-level behavioural scenarios showing interactions among architectural components and their responsibilities.

Use-Case Maps provide an intuitive, high-level notation for both structural and behavioural aspects of a system. Basically, UCMs capture a set of related use cases that cut across the component structure and touch responsibilities in the components. A UCM responsibility represents a cohesive subset of the behaviour defined by a component (e.g. a high-level function, activity or action). This notion is based on (Wirfs-Brock & McKean, 2002) but applied to architectural components in general instead of just object-oriented elements. UCMs are typically derived from informal requirements or use cases, and the UCM responsibilities are inferred from these requirements.

The architect's second task is to map the architectural components and responsibilities of the UCMs to an object-oriented implementation (in our case, Java classes and methods). For the two tasks, we assume the availability of the source code implementing the application, and a degree of correspondence between the architectural specification and the actual system implementation. In situations where the application code exists but the architectural documentation is poor or unreliable, reconstruction and conformance tools (e.g. LATTIX,³ STRUCTURE101,⁴ etc.) should be applied before running FLABot.

In the debugging phase, the developer's first task is to select an architectural element (e.g. a UCM responsibility) in which an error has been observed. The developer's second task is to provide FLABot with a set of application

execution traces involving the selected responsibility. To do so, the application code is instrumented according to the mappings between UCMs and code (defined by the architect in the first phase). The developer runs the application and FLABot logs events of interest (including error conditions), generating a set of execution traces. Once FLABot collected these traces, the third task is the search for faults that involves both the FLABot assistant and the developer. This search follows a mixed-initiative modality of assistance (Wilkins & desJardins, 2001), in which the developer makes some principled decisions (e.g. about certain UCM responsibilities), they trigger analysis and suggestions from FLABot, which leads to subsequent developer's decisions, and so forth. FLABot will analyze the differences between UCMs and corresponding execution traces, returning a set of responsibilities whose misbehaviour could explain the error. These architectural responsibilities direct the developer to potential faults in the code. Actually, FLABot inserts breakpoints in those code regions that are relevant to the problem. The developer's last task is to perform a detailed analysis of the code marked by the assistant via a standard debugging tool.

In the following sub-sections, we go through an example that uses the FLABot prototype to find a fault in a simple user account management (UAM) system. This example helps to understand the architecture definition and debugging phases in more detail.

2.1. Specifying the architecture with UCMs

The UAM system is organized around a model-view-controller (MVC) architectural pattern (Buschmann *et al.*, 1996), as shown in the component diagram of Figure 2. The architect creates and manipulates the architecture specification using an Eclipse-based editor that is part of the FLABot tool. This editor supports both UCM and UML2 component diagrams. UML2 component diagrams (Ambler, 2005) (shown at the top in Figure 2) capture the structure of the system in terms of components that interact via provided/required interfaces. Components represent

³Lattix homepage. <http://www.lattix.com/>

⁴Structure101 homepage. <http://www.headwaysoftware.com/products/>

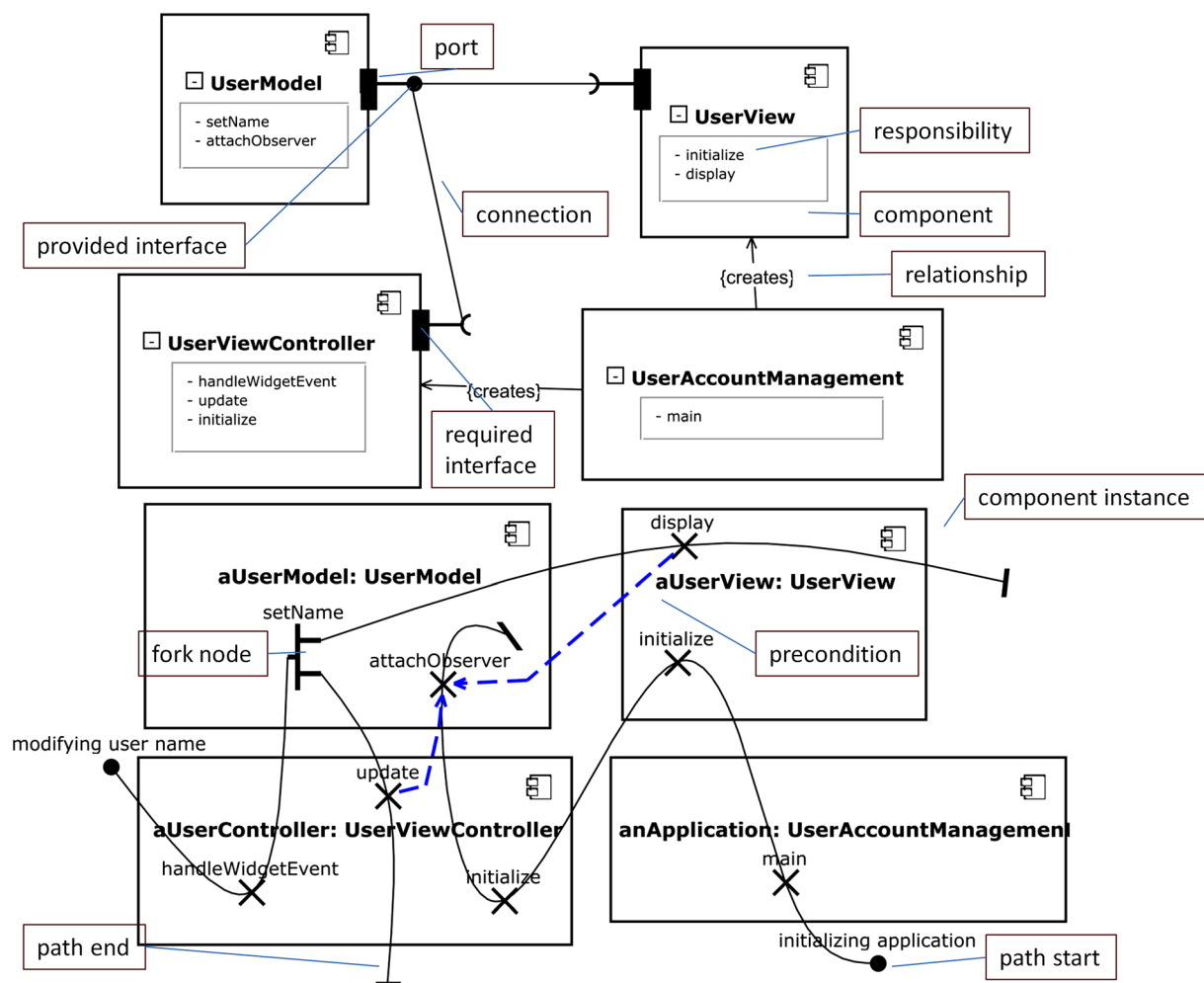


Figure 2: Component diagram and Use-Case Maps specification of the user account management architecture.

units of computation and state. In general, *FLABot* can deal with any kind of component-based modelling of the system, as provided by the architect.⁵ In particular, the UAM architecture has four main components: *UserModel*, *UserView*, *UserAccountManager* and *UserViewController*, which instantiate the different parts of the MVC pattern. In *FLABot*, a component is also a container of responsibilities, which are allocated by the architect in order to capture functional aspects of the system. In the case of the UAM example, the architect has chosen to model and allocate functions such as *setName*, *display*, *handleWidgetEvent* and *attachObserver*, among others. Note that some of these functions come from the problem domain (e.g. *setName* and *display*), while others are derived from the MVC mechanisms (e.g. *attachObserver* and *update*). The components and responsibilities can later appear in several UCM diagrams, as the architect specifies the behavioural scenarios of the system.

A UCM diagram models a set of related scenarios. A UCM scenario is represented by paths that show the progression of causes and effects among responsibilities. Figure 2 (in the succeeding texts) shows a UCM diagram for a *modifying-user-name* scenario in the UAM system. There are two paths of responsibilities involving the four

components. The responsibility *handleWidgetEvent* (in *UserViewController*) translates the events produced by a user's action to responsibility *setName* (in *UserModel*), which then notifies to the *UserView* and *UserViewController* components. In the UCM context, scenarios are more abstract than UML sequence diagrams, allowing developers to think architecturally about functionality without considering (at this stage) the actual object-oriented implementation of the paths. Alternative implementations of the same UCM scenario are possible, as long as the causal flow of responsibilities is ensured. *FLABot* leverages on this feature to support the fault-localization reasoning process.

Responsibilities in paths can be connected according to 'coupling patterns' (e.g. fork and join nodes, preconditions, etc.). In the UCM of Figure 2, responsibility *setName* is represented as a fork node meaning that the change propagation might happen in parallel. In addition, due to the prescriptions of the MVC used in the UAM example, the model only notifies changes to components that have been registered for notifications. Thus, responsibilities *display* and *update* will be executed only if the *UserView* and *UserViewController* instances have been previously added as observers of *UserModel*. The registration is modelled by the *initializing-application* path, whereas the MVC prescription is modelled by two preconditions (depicted as dashed arrows in Figure 2) from responsibilities *update* and *display* to *attachObserver*.

⁵Note, however, that architectural patterns are not first-class modelling entities in the tool.

After creating the UCM diagrams, the architect needs to relate architectural responsibilities and components with object-oriented counterparts in the implementation. Normally, the architect has knowledge of these relationships, and she works together with the developers to define the mappings to Java code using the architectural editor of *FLABot*. A variety of mapping relationships between architecture and code are possible. In practice, it is a good idea to exploit ‘regularities’ in the mappings, such as coding policies. For example, a policy can be that each component always maps to one principal Java class (or Java interface), which may rely on other subsidiary classes to realize the responsibilities of the component. When coding policies are used judiciously to reduce complexity, many mappings end up in one-to-one or one-to-many cases.

The UML class diagram of Figure 3 shows a possible MVC implementation, in which simplified mappings from UCM responsibilities to Java methods are graphically shown using solid arrows. For instance, responsibility *setName* (in *UserModel*) is mapped to methods *setName()* and *notify()* in classes *UserModel* and *Observable*, respectively. As we will explain later in Section 3.3, the mappings are useful for gathering runtime system information related to possible faults.

2.2. *FLABot*’s assistance at work

Let’s assume the UAM system has been correctly documented with UCMs, and a bug report arrives during system maintenance. This bug report states that when the ‘user name’ is changed, the GUI is not properly refreshed and still shows the old ‘user name’. Let’s also consider that a developer, who is not familiar with the UAM implementation, has to deal with this bug report. According to the UAM architecture specification, the GUI is represented by the *UserView* component and the *display* responsibility is supposed to update the ‘user name’. After analyzing the bug report, the developer decides to investigate the wrong behaviour in *display*. To do so, she flags *display* in the UCM editor and asks the *FLABot* assistant to start its diagnosis from that responsibility. Before proceeding to the debugging session, the developer

exercises an instrumented version of UAM with test cases for *UserView*, in such a way the misbehaviour of *display* is exposed. The resulting execution traces are inputted into *FLABot*.

In the debugging session, *FLABot* starts searching for points at which the system did not execute as expected, which could have ramifications to responsibility *display*. To hint problems, the assistant moves backwards following the relationships among responsibilities in the UCMs. This analysis is inspired by fault trees (Sullivan *et al.*, 1999), in which one starts with an (undesired) system state and then attempts to determine causes for that state. In a fault tree, the root represents a particular system state and each event that could affect that state is modelled as a child node. Any parent node specifies the rules that trigger its state, based on conditions observed in its children. Essentially, a fault tree analysis assesses the probability of the root state by means of a top-down chain of reasoning. An expert architect could use a fault tree as her mental model to track down the error in responsibility *display*. However, an expert will normally assess each node in the tree using different heuristic rules based on her experience. The *FLABot* assistant implements a fault-searching system that emulates the expert’s reasoning, although it is limited to a small number of heuristics (see description in Sections 3.2 and 3.3). Figure 4 (at the top) shows the search results for the GUI-update error in terms of the UCM diagram, and the steps performed by *FLABot* when traversing the UCM, allowing the developer to track the progress in the fault-localization process.

While exploring the causes for a fault, an expert often makes judgments about whether particular responsibilities have evidence of errors. This evidence can be obtained from code inspections and test cases exercising specific code sections. For instance, if methods *setName()* or *notify()* reported abnormal executions, the architect can infer that responsibility *setName* (in *UserModel*) is in some error state and needs further inspection. Note that, by selectively probing the code, the architect prunes the fault tree, because a responsibility whose methods executed correctly can be considered as an ‘end point’ in the search. To emulate this human capability, the *FLABot* assistant complements its

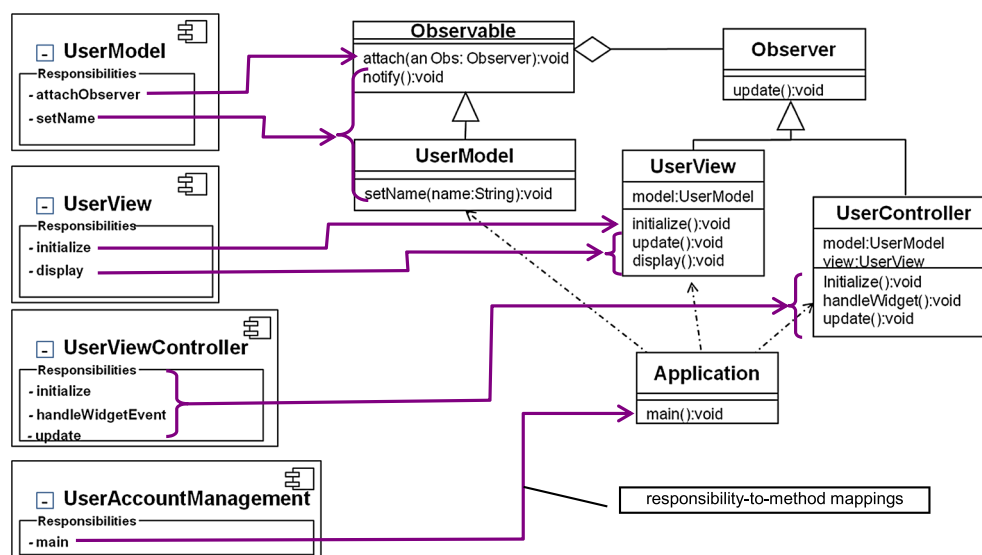


Figure 3: *Mappings from architectural responsibilities to a Java implementation.*

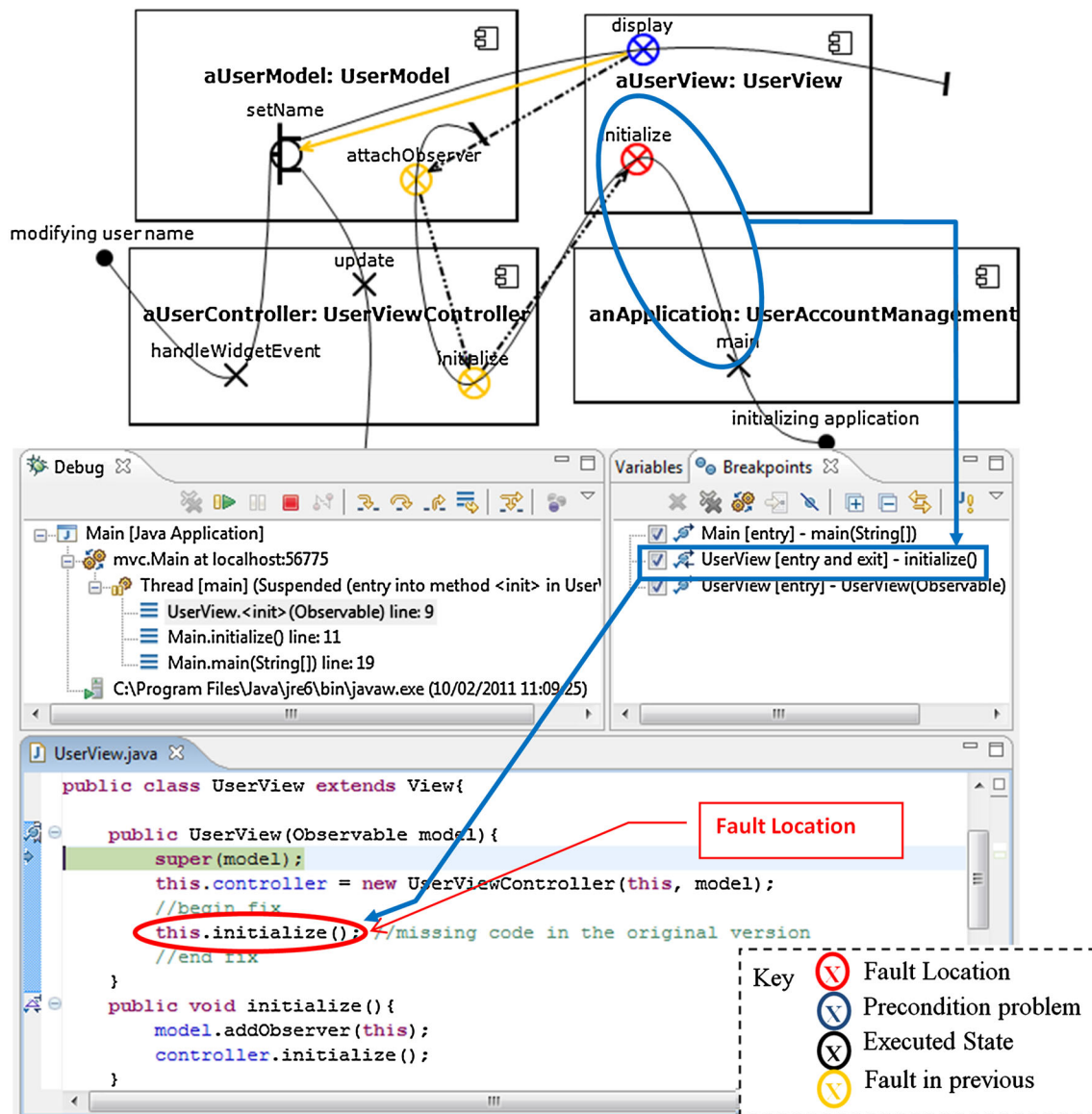


Figure 4: Interactive search for faults in the user account management system.

responsibility analysis with application execution information. We refer to the evidence of errors in a responsibility as the *error status* for that responsibility. Basically, any UCM responsibility that maps to Java code is also associated to a gauge that monitors the execution(s) of that code. We refer to this gauge as *fault estimator*, and it will report the status of the responsibility upon request of the *FLABot* assistant. As we will explain in Sections 3.2 and 3.3, the responsibility status is an indicator for the fault-proneness of a responsibility. Coming back to Figure 4, there is a fault estimator for each responsibility in the UCM. For example, the fault estimator for `setName` graphically reports that the responsibility executed without problems.

The *FLABot* assistant continues its search until it comes up with a set of faulty responsibilities. Crossed circles on responsibilities show the diagnosis steps, and colours give clues about the nature of the fault under analysis. In Figure 4, the diagnosis can be interpreted as follows. The black circle on `setName` indicates that this responsibility was analyzed and works properly, so it requires no further analysis. The blue crossed circle on `display` indicates that there is a problem with some preconditions of this responsibility, and the slashed arrow points at the suspicious

responsibility. The yellow crossed circles on responsibilities `attachObserver` (in *UserModel*) and `initialize` (in *UserViewController*) mean that those responsibilities were not executed due to a fault in some previous responsibility. The culprit is responsibility `initialize` (in *UserView*), and the red crossed circle on it indicates a fault in the code for that responsibility. *FLABot* reached this conclusion after detecting that responsibility `main` (in *UserAccountManager*) has executed correctly. To sum up, *UserView* is not updated because the execution of the *initializing-application* path stopped at responsibility `initialize` (in *UserViewController*); and therefore, *UserView* cannot receive notifications from *UserModel*.

At the bottom of Figure 4, we show the set of breakpoints suggested by *FLABot* for the faulty responsibilities along with the fault location in the code. This is the output of the *FLABot* diagnosis for the GUI-update error. To help the developer with the final task of examining the code to locate the specific place of the fault, *FLABot* inserted three breakpoints in methods `initialize()`, `UserView(Observable)` and `main()`. The first two breakpoints (mapped to responsibility `initialize`) correspond to the place in which *FLABot* detected the deviation from the intended

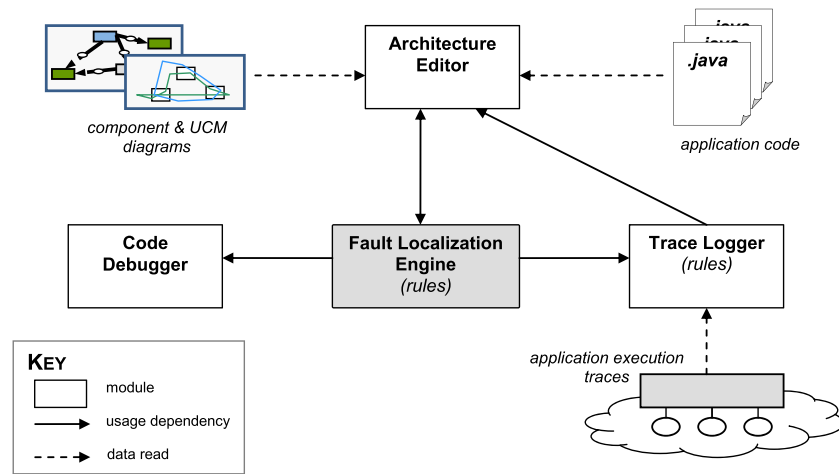


Figure 5: *FLABot* architecture

behaviour, while the remaining breakpoint was inserted because responsibility *main* is part of the execution context of responsibility *initialize*. The rationale is that the fault can be either in the communication between *main* and *initialize*, or in the code implementing *initialize*. In our example, the Eclipse debugger inserted breakpoints that direct the developer to the constructor of class *UIView*, where an invocation to method *initialize()* seems to be missing. A quick code examination reveals that the missing code prevents responsibility *setName* from activating responsibility *display*, so the developer can finally fix the error. Despite its simplicity, this example illustrates the complexity of finding faults, especially if developers have to perform the search by reading and understanding code sentences.

3. Design of the *FLABot* tool

The *FLABot* prototype⁶ is designed as an Eclipse plugin that comprises three main modules: *ArchitectureEditor*, *TraceLogger* and *FaultLocalizationEngine* (Figure 5). The *ArchitectureEditor* module provides a graphical editor for the creation of UCMs and component diagrams and supports the mapping of UCM elements to Java code. Furthermore, this module acts as a repository of the system's architectural information, so that the *TraceLogger* and *FaultLocalizationEngine* modules can access that information for their work. The *TraceLogger* is responsible for the application instrumentation, which is configured according to the responsibility mappings defined with the *ArchitecturalEditor*. The *TraceLogger* is also equipped with predefined fault estimators that translate low-level runtime events into responsibility statuses. The implementation of these fault estimators is based on rules specified by a system expert (Section 3.3). The *FaultLocalizationEngine* module takes the UCMs provided by the *ArchitectureEditor* and is in charge of running the fault-localization algorithm on them. When necessary, the *FaultLocalizationEngine* consults the status of UCM responsibilities by pulling data from the *TraceLogger*. The fault-localization algorithm is also implemented as a rule-based system (Section 3.2). At the end

of a fault-localization session, the *FaultLocalizationEngine* passes a list of breakpoints to the *CodeDebugger* module, which inserts the breakpoints in the code. *CodeDebugger* is actually the Java debugger of the Eclipse platform.

A technical objective when designing the *FaultLocalizationEngine* was to allow developers to combine information from various sources (e.g. architectural model, execution traces, developer's inputs, etc.), while keeping a good separation of concerns. Thus, the implementation of the *FaultLocalizationEngine* is intentionally flexible for accommodating those requirements.

3.1. UCMs as a responsibility graph

The purpose of the *ArchitectureEditor* module is to build a graph of responsibilities out of all the component and UCM diagrams. As the diagrams are being edited and the editors store the architectural information (and particularly, the responsibilities) into a common repository. Architectural responsibilities can take part in different types of relationships, depending on how responsibilities influence each other. We distinguish four types of responsibility relationships, namely: functional dependency, refinement dependency, precondition dependency and constraint dependency. These dependency types are navigated differently by the fault-localization algorithm in the *FaultLocalizationEngine*. The four types are summarized in the succeeding texts.

A functional dependency between a pair of responsibilities *R1-R2* on the same UCM path means that *R2* happens after *R1* in the execution flow. This is the usual interpretation of responsibilities for UCMs (Buhr, 1998). For example, there is a functional dependency between *initialize* and *attachObservers* in the UCM path *initializing-application* in Figure 2. A refinement dependency appears when a responsibility *R1* is linked to a set of responsibilities $\langle R2, Rn \rangle$ in a different UCM, representing a functional explosion of *R1*. That is, $\langle R2, Rn \rangle$ provides a detailed view of the computations of *R1*. This kind of relationship is also known as 'stub' in the UCM jargon, and it permits to organize UCM diagrams at different abstraction levels. For example, responsibility *display* in Figure 2 can be refined by another UCM containing responsibilities that update specific GUI widgets. A precondition dependency between responsibilities *R1* and *R2* says that *R1* cannot execute if *R2* has not executed

⁶*FLABot* homepage: <http://www.exa.unicen.edu.ar/isistan/flabot/>

previously. A precondition allows the specification of temporal activations of responsibilities. Coming back to the MVC pattern in Figure 2, the view will not display its contents if it is not attached to some model. This situation is modelled by a precondition dependency of *display* on *attachObservers*. Conversely, a constraint dependency between responsibilities R1 and R2 says that R2 cannot execute if R1 has executed previously, which means that the behaviour of R1 invalidates the behaviour of R2. For example, a constraint dependency between responsibility *detachObserver* (not shown in Figure 2) in *UserModel* and *display* means that *display* is unable to execute, if *UserView* has been detached as observer of *UserModel* by executing *detachObserver*.

These four dependency types among responsibilities are mainly intended to support behavioural scenarios of the system. The tool users generally leverage on the dependencies either to capture business rules of the domain, or to model some architectural properties (e.g. registration of views in MVC). In our experience, functional dependencies are the most commonly used in order to flesh out UCMs. The use of the remaining dependency types is not mandatory.

3.2. The fault-localization algorithm

The *FaultLocalizationEngine* implements a heuristic algorithm that performs fault analysis based on architectural specifications. This algorithm departs from a responsibility with errors, and then performs an automatic backward search through the graph of UCMs, labelling each responsibility with information about its execution status. A responsibility status can take one of three possible values: ‘executed’, ‘not-executed’, or ‘faulty’. In addition, a responsibility labelled as faulty can come from errors either in a previous responsibility along the UCM path (‘faulty-by-previous’) or in a preconditions upon other responsibility (‘faulty-by-precondition’). A pseudocode of the algorithm, called INSPECT, is presented in Figure 6.

INSPECT is a breadth-first search in the responsibility graph, which takes a problematic responsibility (input parameter *currentR*) and returns a collection of faulty responsibilities (output parameter *causes*). At first, the status of the current responsibility is determined by function *EstimateRuntimeStatus*, which relies on runtime information provided by the *TraceLogger* (input parameter *Obs*). If the status of the current responsibility is not executed, the algorithm starts a new inspection step by calling function *InspectContextOfResponsibility* (Figure 7). This function analyses the predecessors of *currentR* based on the dependency type. After all predecessors are assessed (eventually involving recursive inspections of other responsibilities), function *ApplyEvaluationRules* (Figure. 8) re-evaluates the current responsibility based on the predecessors’ statuses so as to determine the final status of *currentR*. The rules used by function *ApplyEvaluationRules* are the following. In case *currentR* has a functional or precondition dependency with a previous responsibility in a path, *InspectContextOfResponsibility* inspects firstly the previous responsibility. In case the previous responsibility is actually a UCM stub that represents a refinement dependency, the processing of *InspectContext-OfResponsibility* is slightly different (line 10 in Figure 7). This function evaluates whether the previous responsibility to the sub was executed (variable *inResponsibility*), and if that happened, then analysis continues with the paths contained within the Stub. Examples of stub analysis were considered in the case studies (Section 4.2). Finally, If *currentR* has no other dependency (or all of them have been already visited and have executed), *Apply-EvaluationRules* marks the current responsibility as faulty.

To clarify the algorithm, we describe next the steps involved in the GUI-update error of Figure 4 and the sequence of explored responsibilities by the fault-localization strategy (Figure 9). Let’s assume that we call INSPECT with *display* (in *UserView*) as the input (problematic) responsibility

```

Heuristic INSPECT(currentR, UCMs, Obs)

in:  currentR = <id,status,type> ∈ UCMs - the main responsibility under analysis.
      UCMs - the graph of responsibilities (derived from the UCM diagrams). Each node corresponds to a
      responsibility R (of the form <id,status,type>) and each directed edge RD corresponds to a
      responsibility dependency (functional, refinement/stub, precondition, constraint).
      Obs = <logE1, logE2, ..., logEn> - a sequence of log entries representing the execution of code associated
      to the responsibilities in UCMs.

out:  causes = {r1, r2, ..., rn} - a set of responsibility nodes ri ∈ UCMs that are labeled as faulty (and potentially
      explain the error in currentR).

pre:  The status of currentR = <id,status,type> is void (undetermined). Obs is assumed to be
      non-empty and contain methods with mappings from currentR.

post: The status of currentR = <id,status,type> is changed, after executing INSPECT.

1:  begin-procedure
2:  // Determine status of currentR based on the execution log. This status can be any of
3:  // three values, namely: executed, faulty or not-executed.
4:  currentR.status ← EstimateRuntimeStatus(currentR, Obs)
5:  if (currentR.status = executed) then
6:    return (∅) // No cause of problems in currentR is detected
7:  else if (currentR.status = faulty) then
8:    return ({currentR}) // The root cause of the problem is currentR
9:  else // if currentR did not execute, the node is expanded to the context of its predecessors
10:   // in the graph, and those responsibilities are further analyzed in order to derive causes
11:   causes ← InspectContextOfResponsibility(currentR, UCMs, Obs)
12:   // The causes just obtained are re-assessed with respect to the main responsibility currentR
13:   causes ← ApplyEvaluationRules(currentR, causes, UCMs)
14: end-if-else
15: return (causes)
16: end-procedure

```

Figure 6: Pseudocode of the fault-localization heuristic.


```

Procedure INSPECTCONTEXTOFRESPONSIBILITY(currentR, UCMs, Obs)

in:   currentR = <id,status,type> ∈ UCMs - the main responsibility whose context is analyzed.
      UCMs - the graph of responsibilities (derived from the UCM diagrams). Each node corresponds to a
      responsibility R (of the form <id,status,type>) and each directed edge RD corresponds to a
      responsibility dependency (functional, refinement/stub, precondition, constraint).
      Obs = logE1, logE2, ... logEn - a sequence of log entries representing the execution of code associated
      to the responsibilities.

out:  causes = {r1, r2, ... rn} - a set of responsibility nodes ri ∈ UCMs that are labeled as faulty (and potentially
      explain the error in currentR).

pre:  The status of currentR = <id,status,type> is not-executed (as determined by EstimateRuntimeStatus).

post: All the predecessors of currentR in UCMs are analyzed by invoking INSPECT.

local: previousR = <id,status,type> - a responsibility node that precedes currentR in UCMs
      inResponsibility = <id,status,type> - responsibility node previous to a stub in UCMs
      outResponsibility = <id,status,type> - responsibility node that corresponds to a stub output in UCMs

1: begin-procedure
2: causes ← ∅
3: // Analyze causes for currentR being not-executed by looking at previous responsibilities (in the graph)
4: if ( (currentR.type = responsibility) or (currentR.type = and-fork) or (currentR.type = or-fork)
5:   or (currentR.type = and-join) ) // Cases of normal responsibilities, Join or Fork nodes in the UCMs
6:   foreach previousR in Predecessors(currentR, UCMs) do
7:     // Recursively inspect and label all the predecessors of currentR
8:     causes ← causes U INSPECT (previousR, UCMs, Obs)
9:   end-foreach
10: else if (currentR.type = stub) // Case of stub for a refinement UCM
11:   inResponsibility ← GetInResponsibility(currentR, UCMs)
12:   causes ← causes U INSPECT (inResponsibility, UCMs, Obs)
13:   // If there is a fault previous to the stub node, then no analysis is required for the stub.
14:   // Otherwise, the problem is in the stub paths.
15:   if (inResponsibility ∈ causes)
16:     return (causes)
17:   else // the stub is expanded and its responsibility paths are analyzed for faults.
18:     outResponsibility ← GetOutResponsibility(currentR)
19:     causes ← causes U INSPECT (outResponsibility, UCMs, Obs)
20:   end-if-else
21: end-if
22: return (causes);
23: end-procedure

```

Figure 7: Rules for navigating the graph based on the type of the responsibility under analysis.

```

Procedure APPLYEVALUATIONRULES(currentR, UCMs, causes)

in:   currentR = <id,status,type> ∈ UCMs - the main responsibility whose status is to be determined.
      UCMs - the graph of responsibilities (derived from the UCM diagrams). Each node corresponds to a
      responsibility R (of the form <id,status,type>) and each directed edge RD corresponds to a
      responsibility dependency (functional, refinement/stub, precondition, constraint).
      causes = {r1, r2, ... rn} - a set of responsibility nodes ri ∈ UCMs that are labeled as faulty (and potentially
      explain the error in currentR).

out:  causes = {r1, r2, ... rn} - a set of responsibility nodes ri ∈ UCMs that are labeled as faulty (and potentially
      explain the error in currentR).

pre:  The status of currentR = <id,status,type> is not-executed. All predecessors of currentR have been
      inspected and their statuses are known.

post: The status of currentR is modified to one of three possible values: faulty-by-previous, faulty, or faulty-
      by-precondition. These values correspond to the diagnosis marks shown by the FLABot UI.

local: previousR = <id,status,type> - a responsibility node that precedes currentR in UCMs

1: begin-procedure
2: foreach previousR in Predecessors(currentR, UCMs) do
3:   if previousR ∈ causes then // the fault was detected in a predecessor of currentR
4:     currentR.status ← faultyByPrevious
5:     causes ← causes U currentR // currentR is in the faulty path.
6:     return (causes)
7:   end-if
8: end-for
9: // If there is no fault in the previous responsibilities, then currentR is likely to have the fault
10: currentR.status ← faulty
11: causes ← causes U currentR
12: return (causes)
13: end-procedure

```

Figure 8: Rules for assessing a given responsibility, after evaluating its precedent responsibilities.

(step 1 in Figure 9). Initially, function *EstimateRuntimeStatus* (line 4 in Figure 6) performs a diagnosis of *display* and sets its error status (details about *EstimateRuntimeStatus* are provided in Section 3.3). Let's say that *display* is reported as

'not-executed' by *EstimateRuntimeStatus*. This means that *TraceLogger* could not find evidence accounting for the execution of *display* in the application traces. So far, the decision of making *display* responsible for the fault is left open

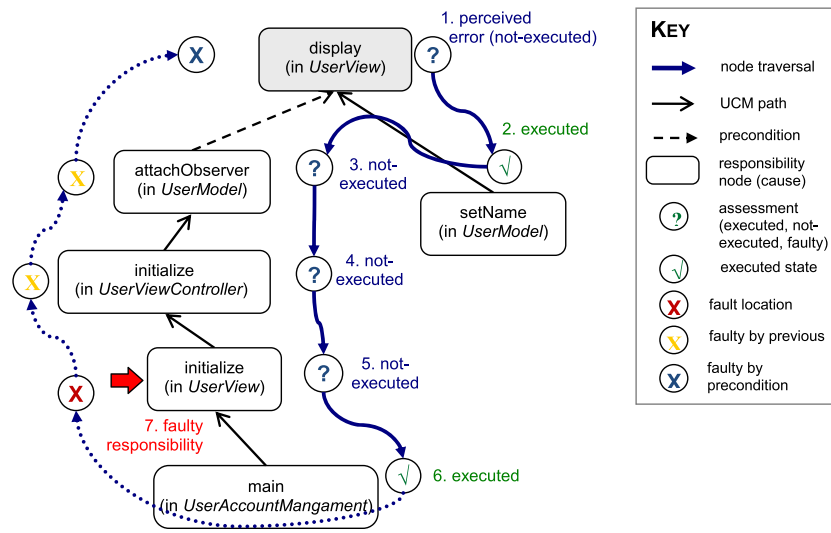


Figure 9: Sequence of responsibilities explored by the fault-localization algorithm.

(question mark in Figure 9). The status of *display* will be re-considered after assessing its previous responsibilities in the UCMs. Here, *InspectContextOfResponsibility* (line 11 in Figure 6) analyses previous responsibilities according to their dependency type with *display* (*currentR*). According to the UCM of Figure 2, *display* is influenced by previous responsibilities *setName* (in *Model*) and *attachObserver* (also in *Model*). Note that the first responsibility is on the same UCM path as *display*, while the second one is on a different UCM path but still linked to *display* through a precondition dependency. Here, function *Predecessors* picks *setName* as the current candidate for a new inspection step (a recursive *Inspect* invocation) and leaves *attachObserver* pending.

Following the steps shown in Figure 9, let's assume that *setName* is marked as 'executed' based on the execution traces. The algorithm considers that the system has functioned well at that point, so no further inspection is needed on this branch (step 2 in Figure 9). Therefore, the algorithm goes back to *InspectContextOfResponsibility* and checks if other responsibilities prevented *display* from being executed. A new inspection step starts with *attachObserver* (in *UserModel*) as the current responsibility (step 3 in Fig. 9). Function *EstimateRuntimeStatus* reports that *attachObserver* did not execute. The decision for a fault in *attachObserver* is left open, and *InspectContextOfResponsibility* moves backward in the *initialize-application* path to figure out what caused the error status of *attachObserver*. The inspection continues with *initialize* (in *UserViewController*), determining that it is 'not-executed', and then moves to *initialize* (in *UserView*) obtaining the same result (steps 4–6 in Figure 9). Function *InspectContextOfResponsibility* reaches responsibility *main* (in *UserAccountManagement*), and *EstimateRuntimeStatus* finds evidence in the log showing that the system executed without problems (step 6 in Figure 9). At this point, function *InspectContextResponsibilities()* goes back to *initialize* (in *UserView*) looking for any other responsibility that might invalidate *initialize* for execution. Because no other responsibilities are pending for inspection, the control is returned to the main *INSPECT* in which *initialize* is the current responsibility. Here, *INSPECT* calls function *ApplyEvaluationRules* to determine the status of *initialize*, based on the evidence just obtained from *main*. As a result, the algorithm

marks *initialize* (in *UserView*) as a possible cause for the GUI-update error (step 7 in Figure 9). This status is propagated to resolve open decisions in pending responsibilities (call-backs represented by dotted arrow in Figure. 9). In our example, the pending statuses are labelled as *faultyByPrevious*, and the fault-localization algorithm is finished.

When it comes to mixed-initiative, the *INSPECT* algorithm allows the developer to supervise the decisions made by the algorithm. After functions *EstimateRuntimeStatus* or *ApplyEvaluationRules* compute the status of a responsibility, the algorithm suspends itself and shows the current results to the developer. The developer can opportunistically agree on the statuses inferred by the assistant, or she can modify results based on her own criterion. In practice, the developer just wants to double-check certain responsibilities when the algorithm has marked them as faulty. The level of mixed-initiative is configurable, so that routinary inspection steps are automatically executed by *FLABot* without user's intervention.

The *INSPECT* algorithm is implemented in *JavaLog* (Amandi *et al.*, 2004), an integration between Java and Prolog that exploits the advantages of the logic and object-oriented paradigms. We codified the exploratory parts of the search using Prolog. Examples of Prolog rules are the prioritization of predecessors by function *InspectContextOfResponsibility*, and the assessment of a responsibility by function *ApplyEvaluationRules*. These two functions currently have default implementations. We envision that an expert can provide other evaluation rules, for instance, to capture architecture-specific properties. In spite of these customization points, there are parts of the algorithm that are well known, such as the traversal of UCMs, the graphical interaction between user and tool, or the invocations of function *EstimateRuntimeStatus* to the module *TraceLogger*. We programmed these parts in Java to achieve efficiency.

Regarding the computational efforts of the *INSPECT* algorithm, we can think of it as a decision tree in which each branching operation basically evaluates the status of responsibilities. The computational cost of an operation depends on the evaluation of queries on a set of method executions associated to each responsibility. In this context,

the worst-case complexity for analyzing a given responsibility is proportional to the largest depth of its decision tree, which is the number of all related responsibilities in a UCM multiplied by the number of invocations of the mapped methods stored in the application log.

3.3. Implementation of fault estimators

Error detection is a prerequisite for relating misbehaviours in the UCMs with faults in the code. This aspect is covered by the *TraceLogger* module, which is designed to gather runtime information about the application behaviour into a log and then perform an offline log analysis. For each responsibility under inspection (by the *FaultLocalizationEngine*), function *EstimateRuntimeStatus* () asks the *TraceLogger* to determine the error status based on the log entries. That is, the possibility of a responsibility being ‘executed’, ‘faulty’ or ‘not-executed’ is computed by probing the execution context of the methods associated to that responsibility (e.g. normal method execution, no execution of certain methods, execution with exceptions, etc.). The pseudocode of function *EstimateRuntimeStatus* is shown in Figure 10. Note the mixed-initiative hook (line 6) in which the tool shows the responsibility status (as inferred from the log) to the user, so that she can confirm it (or modify it) based on her knowledge of the application design and runtime information (parameter *Obs*).

Internally, *TraceLogger* relies on fault estimators for assessing error statuses. In particular, a *fault estimator* knows how to take specific pieces of information from the log, how to aggregate those pieces into events of interest and how to interpret patterns of events into a meaningful responsibility status. Our notion of fault estimator is similar to the notion of gauge (Garlan *et al.*, 2001) or fault locator (Steimann *et al.*, 2008). The mappings of responsibilities to code serve here for configuring the fault estimators with the code sections to be monitored. For each UCM responsibility, we have a fault estimator with a set of probes that listen to code-level events from Java methods for that responsibility (e.g. object creation, method call, method return, exceptions, type casting, field access, etc.).

The instrumentation mechanism provided by *TraceLogger* is based on an infrastructure of probes as described in (Garlan *et al.*, 2001). For program instrumentation, we used the *Javassist* toolkit (Chiba, 2000), which permits Java bytecode manipulation at load-time. When the developer launches the

instrumented version of the application from Eclipse, the probes save all the events of interest generated by the application to a log. Note that the instrumentation is selective in that probes are deployed only in those methods mapped from a UCM with problematic responsibilities. We assume that the developer knows how to exercise application functions that correspond to these problematic responsibilities. That is, if the developer is interested in inspecting a UCM responsibility that performs a wrong view update, she should consequently exercise application functions related to the implementation of that view update.

Inside the log, the events captured when executing an application are organized into traces for further analysis. A trace defines a sequence of ‘executed’ responsibilities. Figure 11 shows a log fragment generated by the execution of the UAM system, and how a trace summarizes code-level information in terms of responsibilities. The box atop in Figure 11 contains runtime events for the GUI-update error, and the middle box shows their correspondence with UCM responsibilities (remember the mappings in Figure 3). For example, responsibility *setName* holds references to two events logged from the execution of methods *setName(String)* and *notify()*, while responsibilities *handleWidgetEvent* and *main* hold references to events from methods *handleWidgetEvent(String)* and *main(String[])*, respectively.

Each fault estimator is equipped with a set of rules that recognize patterns of runtime events (e.g. method name is valid, parameter values and return are not null, no exceptions thrown, etc.). We refer to these patterns of events as *execution conditions*. Then, the responsibility status is actually the result of evaluating concrete execution conditions. The bottom-level box in Figure 11 shows a possible fault estimator for responsibility *setName* and simplified Prolog rules over execution conditions. In our case, when events 3 and 4 are filtered from the log, the estimator evaluates its rules (via the query *?-status('setName',Status)*, line 5 in Figure 11) and reports that *setName* has been successfully executed. In this particular example, the rules are expressed in terms of execution conditions for methods *setName(String)* and *notify()*, which are actually the mappings defined for responsibility *setName*. Alternatively, the responsibility could be faulty if, for example, method *setName(String)* were invoked with a null argument, or if some exception were logged by the probes. In general, different execution conditions for the

```

Procedure ESTIMATERUNTIMESTATUS(currentR, Obs)
  in:  currentR = <id,status,type> ∈ UCMs - the main responsibility whose status is to be determined.
       Obs = <logE1, logE2 ... logEn> - a sequence of log entries representing the execution of code associated
       to the responsibilities.

  out: Status ∈ {executed, non-executed, faulty} - the execution status of currentR.

  local: estimatorRules - set of Prolog rules for assessing currentR. These rules are based on its mapped
         methods, and they are configured by an expert.

  pre: The status of currentR = <id,status,type> is void (undetermined).
  post: The status of currentR is determined, after evaluating estimatorRules on Obs

1: begin-procedure
2: estimatorRules ← GetRules(currentR) // The rules for the fault estimator(s) for currentR
3: prologEngine.loadRules(estimatorRules)
4: prologEngine.setFacts(Obs) // The world state on which the status rules will be executed
5: prologEngine.query("status(currentR, Status)") // The evaluation of the rules itself
6: status ← askUserToConfirm(currentR, Status.value(), Obs) // GUI displayed in the debugger
7: return (status)
8: end-procedure

```

Figure 10: Pseudocode of the responsibility assessment (fault estimators) based on the log.

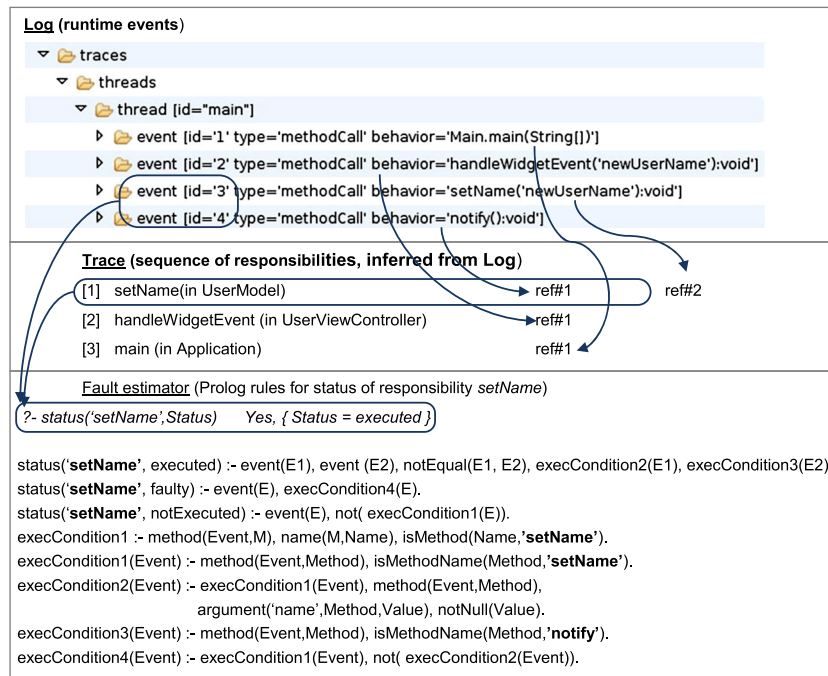


Figure 11: Aggregation of runtime events into an error status by a fault estimator.

events can be combined in the rules, and these rules can be part of different fault estimators. The events, execution conditions and rules work integrated with the *JavaLog* framework mentioned in Section 3.2.

Note that the architect or other expert (or maybe the developer inspecting a given UCM) is responsible for the configuration of the rules for the fault estimators associated to each responsibility. In fact, the architect can provide rules that express properties of architectural patterns, as in our example of rules earlier (the architect must know that the pattern is being used in the UCMs though). Anyway, this customization of rules is not mandatory for the workings of the fault-localization algorithm. In order to facilitate the task, *TraceLogger* provides predefined schemas of rules and execution conditions, which are set by default to all the UCM responsibilities. When a schema is applied to a specific responsibility, the variables of the rules of this schema get bound to the responsibility name and method names (or signature), based on the mappings provided by the architect. In our example, the mappings of responsibility *setName* will instantiate the schema of rules shown in Figure 11 (bottom box). Furthermore, an expert can often customize the resulting schemas to include other conditions/rules, as she sees fit with the UCM or application being analyzed.

4. Evaluation

To evaluate the *FLABot* approach, we carried out experiments on two case studies, applying the tool to detect predefined faults. The objective was to assess the effectiveness of the fault-localization heuristic based on UCMs and runtime information when compared with a traditional fault-localization activity performed by developers.

The first case study is a telecommunication accounting system, called *G2*, developed by an Argentine software company. The *G2* implementation has around 43.000 lines

of code. The second case study is the *ArchitectureEditor* module described in Section 3, whose implementation has around 37.000 lines of code [without considering the Java source code of the Eclipse Modeling Framework (EMF)/ Graphical Editing Framework (GEF)]. For each case study, we selected six developers. The participants for *G2* were required to have programming experience in both Java and the Eclipse environment, and they came from a graduate course on software design offered at UNICEN University. The participants for *ArchitectureEditor* were required to have some experience in real-world projects and came from software companies collaborating with our research group. This way, we can evaluate *FLABot* with participants used to debug Java code in a work environment.

The six participants of each case study were divided into two groups: *non-FLABot-supported* and *FLABot-supported*. The first group (three participants) was asked to use the Eclipse debugger along with a set of design documents, which included UCMs and architecture-to-code mappings describing the high-level design of the system. The second group (three participants) was asked to use the *FLABot* tool in addition to the Eclipse debugger. This second group had also access to design documentation of the corresponding system. Prior to the fault-localization exercise, all participants received a 1-hour training session on the high-level design and functionality of the assigned system.

We selected three specific faults from each case study based on common bug reports from *G2* and *ArchitectureEditor*. We randomly assigned to every participant an application version containing one single fault, and gave the participants 120 minutes to diagnose the fault (with a possible extension of 100 minutes, in case they could not meet the deadline). To monitor the debugging sessions, we have integrated the Mylyn⁷ plug-in

⁷Mylyn homepage. <http://www.eclipse.org/mylyn/>

into the Eclipse environment, because Mylyn can keep track of code sections accessed by a developer within Eclipse. This way, we collected information about the Java classes/methods browsed by each participant. Once the debugging sessions ended, we used that information to compute three metrics: non-blank, non-commented lines of code (NLoC) of Java methods inspected by a developer, cyclomatic complexity (CC) of these methods and time taken by a developer to find a given fault in the code. When using *FLABot*, the time consumed by the developers also included the time for exercising the application in order to gather the execution traces.

Because the *FLABot* algorithm is heuristic, and also depends on the provided UCMs and application traces, we cannot guarantee its correctness. For this reason, we evaluated the effectiveness of the *FLABot* assistance in terms of whether specific faults were pointed out correctly by the fault-localization heuristic. Furthermore, we analyzed the time metric for each fault, with and without *FLABot* support. Note that we were not interested in the tool performance per se, but rather we considered the time metric as an indicator of the human effort to produce a fault diagnosis. To this end, we complemented the evaluation with an additional criterion: the code elements (e.g. Java methods) browsed by a developer when searching for a particular fault. This code criterion was based on the NLoC and CC metrics (per method).

Two assumptions were made regarding the architectural specification of *G2* and *ArchitectureEditor* (consistently with Figure 1). Firstly, we assumed that the architectural diagrams were created beforehand (by the architect) and made available to the developers. Thus, we did not consider the time spent in specifying the UCMs as part of the fault-localization activity. Secondly, we assumed that the architectural diagrams were in alignment with the implementation. These two assumptions should be interpreted from an architecture-centric perspective, in which architectural artefacts are produced early in the development cycle and drive further development activities (Bass *et al.*, 2003). Certainly, differences between architecture and implementation are likely to occur and they

should be managed properly. The reader can refer to (Diaz-Pace *et al.*, 2012) for an example of a tool to synchronize UCMs with code.

4.1. Case study #1: modelling the *G2* server

G2 is designed as a Web-enabled system based on a three-tier architecture and an event-based mechanism. Basically, there is a central server that interacts with clients and telephonic booths located in several countries in South America. Clients make requests for accounting information to the server via Web browsers. The telephonic booths send information about calls to the server via Internet. The server is divided into several subsystems that encapsulate the business logic. These subsystems are built on top of the *Bubble* architectural framework (Campo *et al.*, 2002). The system implementation consisted of 900 Java classes, mostly related to server-side functionality.

In this case study, the development team had followed an architecture-driven approach, but the existing documentation relied on UML sequence diagrams rather than on UCMs. We asked the architects of *G2* to translate those sequence diagrams to UCMs, and they also specified the UCM mappings to Java classes. Building the initial architecture required around one and a half hours, and the modelling consisted of 34 UCMs. Figure 12 shows examples of UCM diagrams created for *G2*. The left side diagram describes a typical functional scenario for *G2*, called *adding-a-Country*; while the right side diagram captures a part of the underlying framework, which is instantiated by the functional scenario.

On the right side diagram, four components of the *Bubble* framework are identified: *Agent*, *ContainerAgent*, *Task* and *Sensor*. The basic interactions between these components are modelled by four UCM scenarios. Scenario *running-tasks* specify the responsibility path for an agent that performs a given task, according to its current execution context. Scenario *configuring-application-agents* describe the responsibilities for initializing the agent's sensors to listen to events and also specify the configuration of agent's tasks. Finally, scenario *managing-flows-of-events* model the

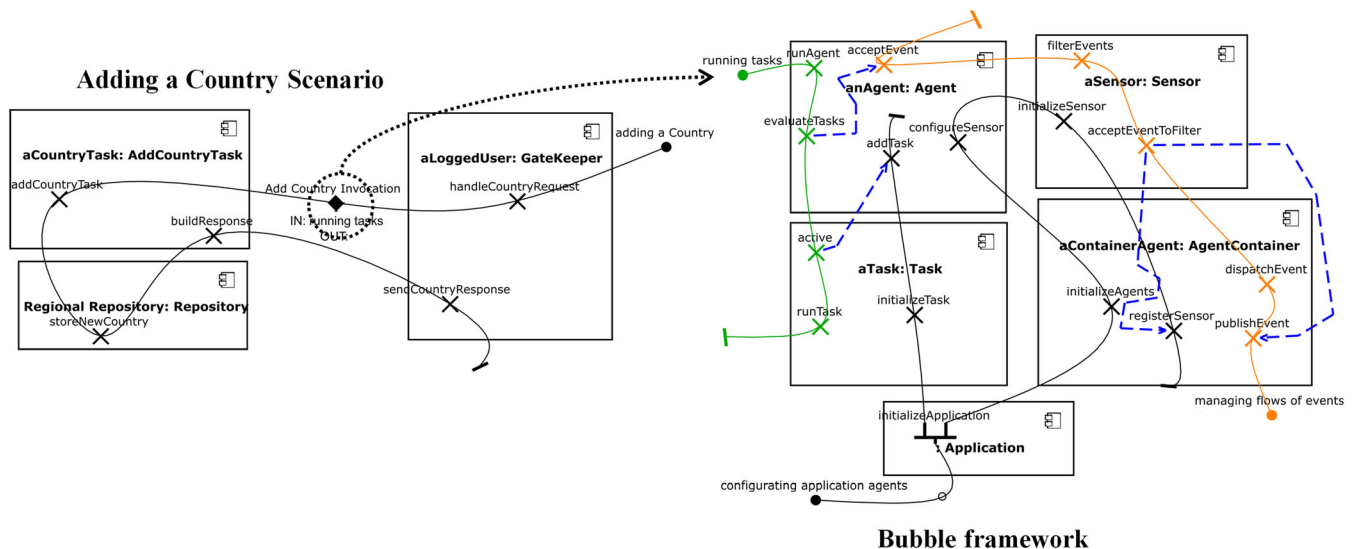


Figure 12: User account management modelling of *Adding a Country Scenario* as an instantiation of *Bubble*.

publish-subscribe communication schema between agents, in which a container agent will distribute events to registered sensors within the container. On the left side, scenario *adding-a-Country* starts with a client request processed at responsibility *handleCountryRequest* (in component *GateKeeper*) and then involves other responsibilities that take care of the registration of a particular country in the system. Note that the UCM scenario *adding-a-Country* is intentionally coarse-grained and relies on a UCM stub for details of the communication between *GateKeeper* and *AddCountryTask*. This stub provides a link to the scenario *running-tasks* of the *Bubbleframework* (on the left side diagram).

4.1.1. Simulating faulty scenarios For the functional scenario *adding-a-Country*, we selected three faults from the *G2* bug tracker system. For each fault, we checked out the corresponding file revisions in order to simulate a situation in which the fault affects the normal behaviour of the system. The details of the faults are the following:

- **Fault #1:** ‘Adding a new country seems not to update the list of countries’.

The configuration code for *AddCountryTask* is never called. This happens because the code responsible for including *AddCountryTask* into the tasks of the agent component is missing.

- **Fault #2:** ‘When adding a new country, the system did not return any page’.

An exception is raised when *AddCountryTask* reads input data due to a bad initialization of the new country object.

The treatment of this exception is erroneously masked in some method, so the details of the exception are never logged.

- **Fault #3:** ‘A country is successfully registered, but the acknowledge page did not appear’.

This fault is similar to the previous one from the point of view of the user. Nonetheless, the effect is produced by a different bug in the code. We altered method *run()* of *AddCountryTask* so that it does not generate the response event sending the acknowledge page back to the user. Thus, the new country is (internally) registered, but it is not noticeable to the user.

The types of faults mentioned earlier are common in event-based systems, and they appear frequently in *G2* when novice developers have to add new functionality to the system following the prescriptions of the *Bubble* framework. Although the faults can be dealt with manually, the process can be time-consuming, especially for developers not familiar with the design rules of the system. Figure 13 (at the bottom) shows the output of the *FLABot* debugging session for fault #1 based on the two UCM diagrams explained in Figure 12.

The observed error is initially in responsibility *sendCountryResponse*. *FLABot* went back in the *adding-a-country* path up to the *Add-Country-Invocation* stub. Here, *FLABot* changed the focus of its analysis to path *running-tasks* and went to responsibility *active*. At this responsibility, *FLABot* detected that *evaluateTasks* was executed and then inspected *addTask*. After detecting that responsibility *addTask* was not executed, it is established as a precondition for *active*. Finally, *FLABot* stopped at *initializeApplication*, which is the first responsibility in the path, given that all its

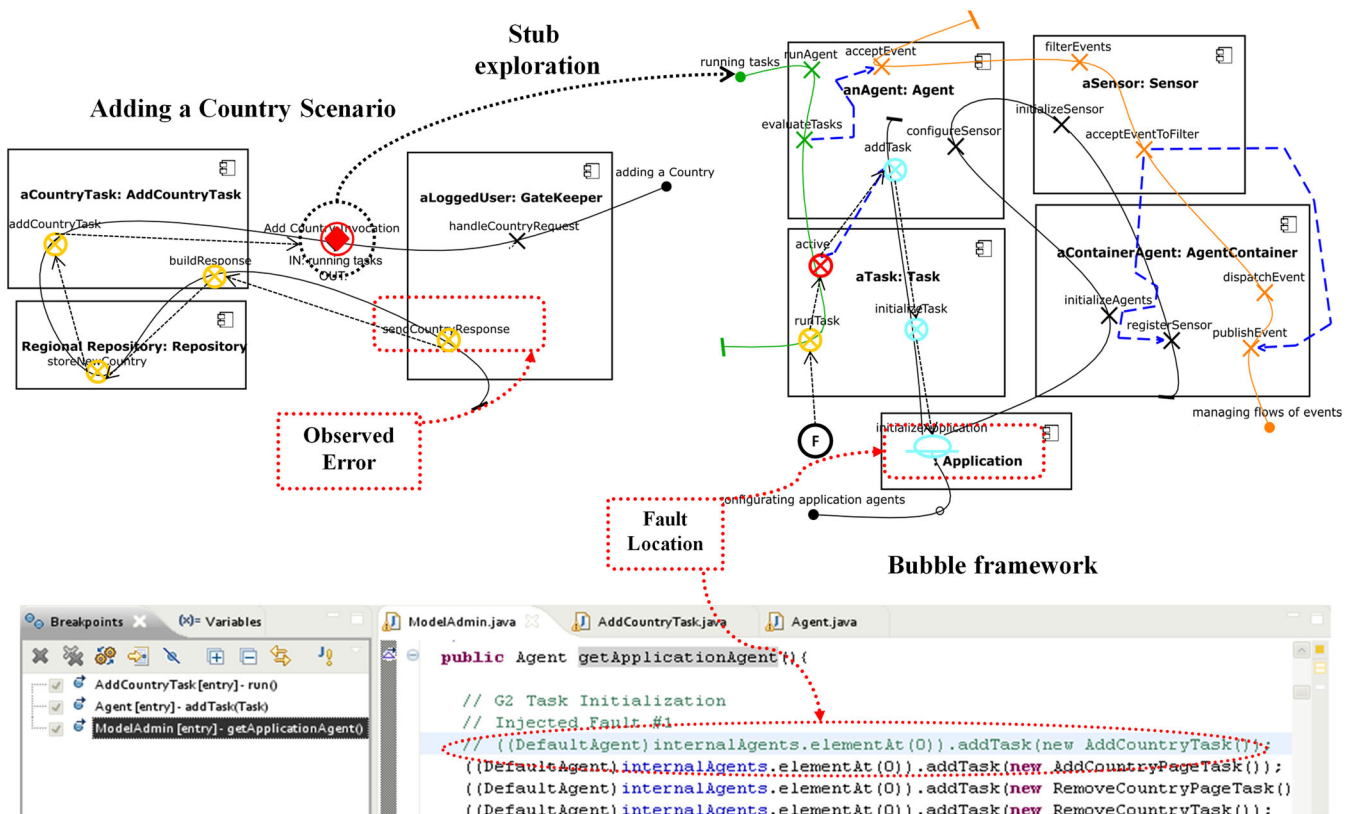


Figure 13: *FLABot* debugging session for fault #1 in the *G2* system.

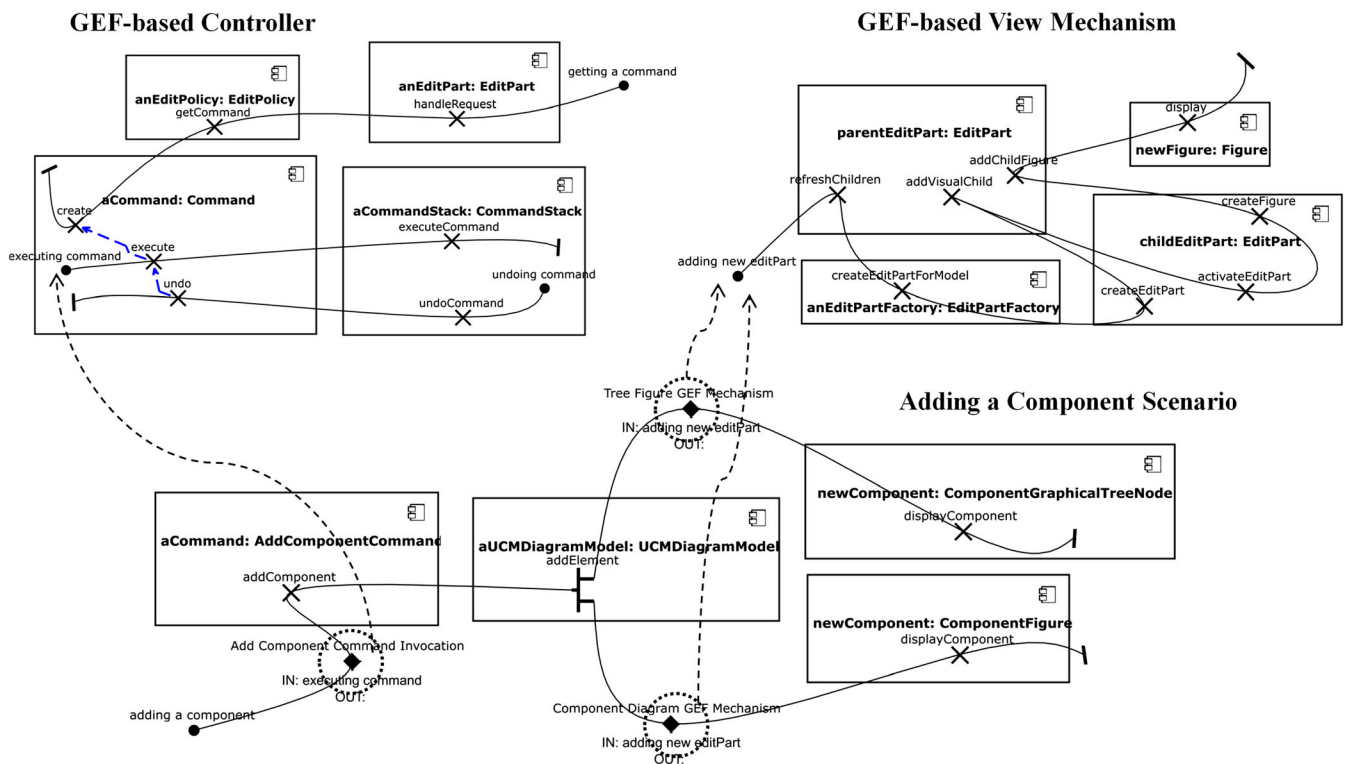


Figure 14: User account management modelling of Architectural Editor as an instantiation of Graphical Editing Framework.

successors were not executed. When following the corresponding breakpoints derived by *FLABot* in the Java code, we can quickly see a configuration problem for *AddCountryTask*. Note that this error appears at the application level (responsibility *sendCountryResponse* in *GateKeeper*), but the source of the problem is at the framework level (responsibility *initializeApplication* in *Application*). This type of fault can be difficult for developers who are not familiar with the framework.

4.2. Case study #2: modelling the architecture editor

ArchitectureEditor is based on the GEF⁸. The architecture follows an MVC architectural pattern that consists of three main parts: a graphical editor for UML2 component diagrams, a graphical editor for UCMs and a core model that stores all the information created with the editors. Being an academic project, the *ArchitectureEditor* included already a baseline of UCMs as part of the architectural documentation, which drove programmers in the implementation of the system. This architecture was modelled with 25 UCMs. The Java implementation for *ArchitectureEditor* consisted of 594 classes (without considering GEF classes). Figure 14 shows the main UCMs for the *ArchitectureEditor*. The top-left block models the execution of commands via GEF; the top-right block models the graphical view of the editor (called *EditParts* in GEF terminology); and the bottom block corresponds to the instantiation of the editor model with specific functionality for adding a component to a diagram. This instantiation is specified using UCM stubs.

Eclipse provides specific guidelines for model elements, graphical views and commands participating in the MVC pattern (Moore *et al.*, 2004). Although GEF makes no assumptions about the model structure, GEF requires the model to store the user's data (or domain-specific information) as well as any graphical properties of the views (e.g. colour, size, position, etc). When building a graphical application, the views (components *EditPart*) are implemented using figures (component *Figure* in Figure 14). There is only one view for a model element in a given diagram. The *EditParts* are organized using a composite design pattern. The *parentEditPart* plays the role of the composed component while the *childrenEditPart* plays the child role. In order to get the correct *Editpart* and *Figure* instances for a new model element, a Factory pattern is used. That is, *parentEditPart* asks component *EditPartFactory* to create those instances.

The functional scenario *adding-a-component* (at the bottom of Figure 14) works as follows. The UCM departs from responsibility *addComponent* in *AddComponentCommand*, which translates the user request to a new component element in the UCM model (responsibility *addElement* in *UCMDiagramModel*). After responsibility *addElement*, the flow of the scenario continues in the stub *Tree-Figure-GEF-Mechanism*. This behaviour is triggered by the model that activates the GEF features for adding and displaying a new figure. This figure represents the new component in both the Outline view and the UCM editor. The stub is linked to responsibility *refreshChildren* (*parentEditPart* instance in component *EditPart*). A factory component configures the figure for a given component (responsibility *createEditPartForModel* in *EditPartFactory*). Then, the parent *EditPart* adds the new component (i.e. the *childEditPart* and *newFigure* instances) and calls responsibility *display* for drawing the component just created.

⁸GEF (Graphical Editing Framework) homepage. <http://www.eclipse.org/gef/>

4.2.1. *Simulating faulty scenarios* Based on the architectural context depicted in Figure 14, we selected three faults from the *FLABot* bug tracker system. These faults are examples of developers' misunderstanding of the Eclipse-GEF rules for instantiating visual applications. Similar to the G2 Server case study, we checked out the corresponding file revisions, in order to simulate a situation in which the fault affects the normal behaviour of the editor. The details of the faults are the following:

- **Fault #4:** 'Adding a component to the component diagram canvas works strangely'.

When the command for adding a component is called twice, a new component is added in the outline view but that component never shows up in the canvas. To produce this effect, we omitted a call to *model.setDiagram(diagram)* in method *redo()* of class *AddComponentCommand*. The source of this problem is the GEF-based commands (top-left block in Figure 14).

- **Fault #5:** 'A new component is added to the Outline view but it is not displayed in the component diagram canvas'.

This fault is similar to the previous one from the point of view of the user. Nonetheless, the effect is produced by a different bug in the code. We altered method *createEditPart(EditPart context, Object model)* to return a

null value when invoked from class *EditPartFactory*, instead of returning a *ComponentEditPart* as expected by the client object. Thus, the source of this problem is the GEF-based views (top-right block in Figure 14).

- **Fault #6:** 'The command for adding components gets suddenly disabled in the GUI'.

To produce this problem, we modified the initialization of instances of *AddComponentCommand*. Specifically, class *ComponentEditPart* creates these instances erroneously, because a null model is passed on the *AddComponentCommand* constructor. Again, the source of the problem is the GEF-based commands in Figure 14.

As a diagnosis example, Figure 15 shows the debugging trace for fault #5 using the *FLABot* tool. The input is an observed error at responsibility *displayComponent* (in *ComponentFigure*). In scenario *adding-a-component*, *FLABot* went back up to the stub *Component-Diagram-GEF-Mechanism*. Here, *FLABot* analyzed the path *adding-new-editpart* until reaching responsibility *createEditPartForModel* (in *EditPartFactory*), because all previous responsibilities were not executed. At *createEditPartForModel*, *FLABot* found evidence of its execution and showed the status to the developer. The developer decided to mark responsibility *createEditPartForModel* as faulty, because an inspection of the associated code showed a null return value in method *createEditPart(EditPart context, Object model)* when an

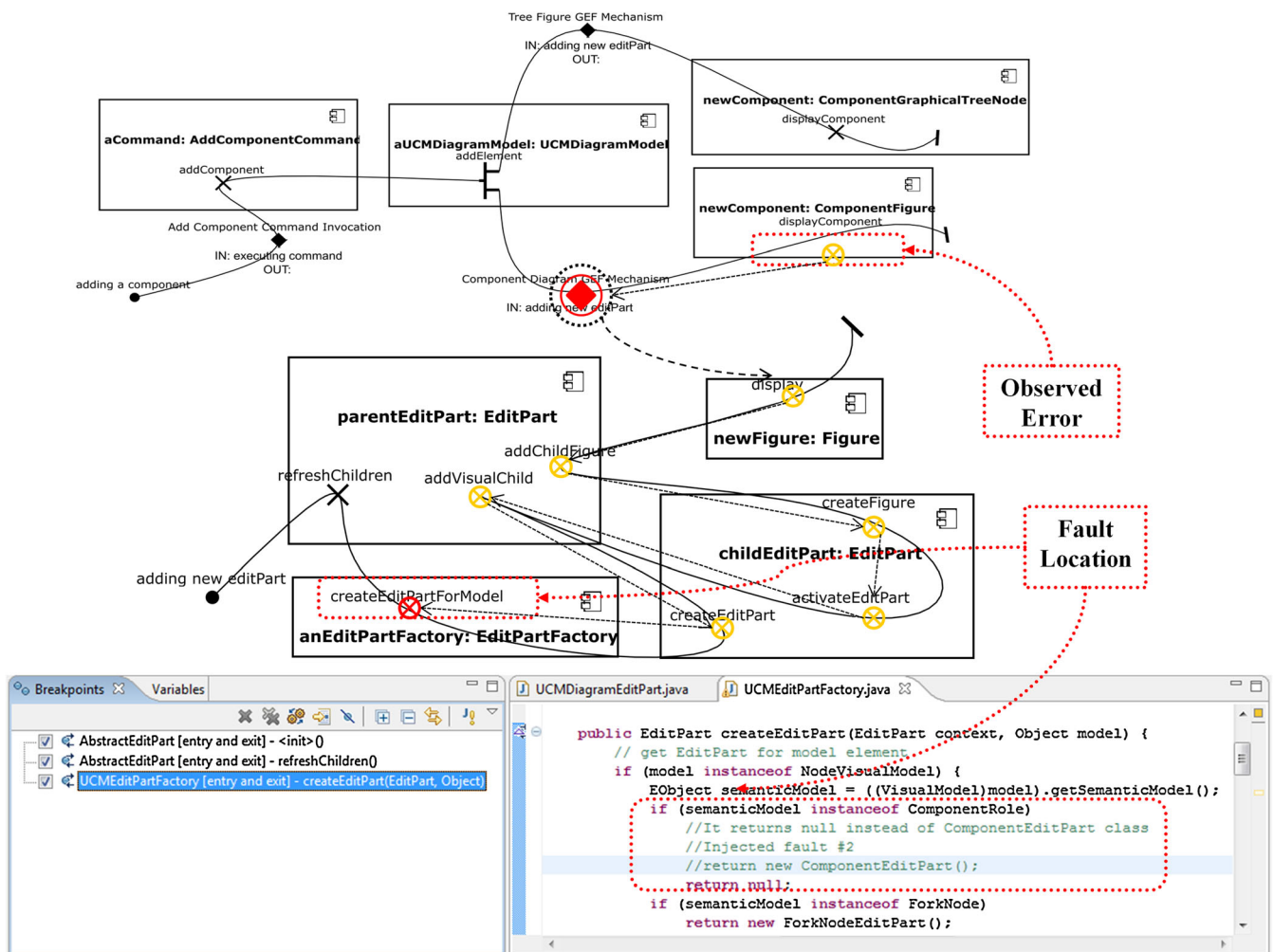


Figure 15: *FLABot* debugging session for fault #5 in the *Architecture Editor* system.

instance of *ComponentEditPart* class should be the expected return.

At the bottom of Figure 15, we show the set of breakpoints suggested by *FLABot* for fault #5. *FLABot* inserted three breakpoints in methods *AbstractEditPart()*, *refreshChildren()* and *createEditPart(EditPart, Object)*. The first two breakpoints (mapped to responsibilities *createEditPart* and *refreshChildren*, respectively) correspond to the execution context of responsibility *createEditPartForModel*. The third breakpoint (mapped to responsibility *createEditPartForModel* in component *EditPartFactory*) is the place in which *FLABot* detected the deviation from the intended behaviour. A quick code examination of *createEditPart(EditPart, Object)* reveals that the 'If' branch testing for class *ComponentRole* (representing a graphical component in *ArchitectureEditor*) returns a null value instead of the corresponding instance; so the developer can finally fix the error.

4.3. Metrics

In this section, we analyze the efforts spent by the participants across the two case studies. Figure 16 summarizes the metrics *Time-Consumed* (a), *NLoC* (b) and *Cyclomatic Complexity* (c) for the localization of the six faults, in the experiments with and without *FLABot*.

For the time consumed per fault, the results show that the participants using *FLABot* spent less time for localizing the faults than the participants without tool assistance (see graph (a) in Figure 16). In particular, we noticed that for the G2 Server case study the *FLABot* participants spent around 30 minutes to perform the fault-localization tasks, but we found more variability in the time required by non-*FLABot* participants. Note that all the participants in the G2 Server case study found the faults within the initial allotted time. For the Architecture Editor case study, all the participants using *FLABot* succeeded in finding the faults before the deadline (120 minutes), with an average of 72 minutes. Unlike the G2 Server case study, the fourth and sixth participants (who did not use *FLABot*) exceed

the allotted time. The fourth participant did detect fault#4 with 100 minutes of extra time. However, the sixth participant could not find fault #6 before the deadline plus the extra time. An interesting finding here was that the developer using *FLABot* identified fault #6 in a short period of time.

Regarding the code browsed by the participants, the *FLABot-supported* participants analyzed fewer Java sentences (see graph (b) *NLoC* in Figure 16) than the other participants. Except for fault#1 in which the *NLoC* difference was marginal, we observed a reduction factor of 2–3 times with *FLABot* assistance. The number of breakpoints (not shown in Figure 16) inserted by *FLABot* in the *ArchitectureEditor* case study was around 4–6 breakpoints per fault, which is consistent with the 3–5 breakpoints per fault obtained in the G2 case study. We conjecture that the number of breakpoints is dependent on the characteristics of the UCM specification as well as on the mapping policy, rather than on the *NLoC* of the system. Nonetheless, not all the breakpoints were correctly placed in the code by the tool. This problem led to false positives, and the developers had some overhead inspecting each code section manually. In our experiments, the precision of the tool regarding whether it pointed out the Java method/class with a fault was around 70%.

Using the Shapiro–Wilk and Kolmogorov–Smirnov tests, we found that our *Time-Consumed* and *NLoC* measurements (for the six faults, that is, both case studies) are not normally distributed. Along this line, we applied the non-parametric Wilcoxon–Mann–Whitney test to the data series, which reported that the groups using *FLABot* were more effective (i.e. they required less time and browsed fewer lines of code per fault) than the groups without *FLABot* assistance, with statistical significance at the level of 0.05.

It is worth noting that, in the case of the G2 faults, the CC values for the methods analyzed by *FLABot-supported* participants kept lower than the CC values for those methods covered by non-*FLABot-supported* participants.

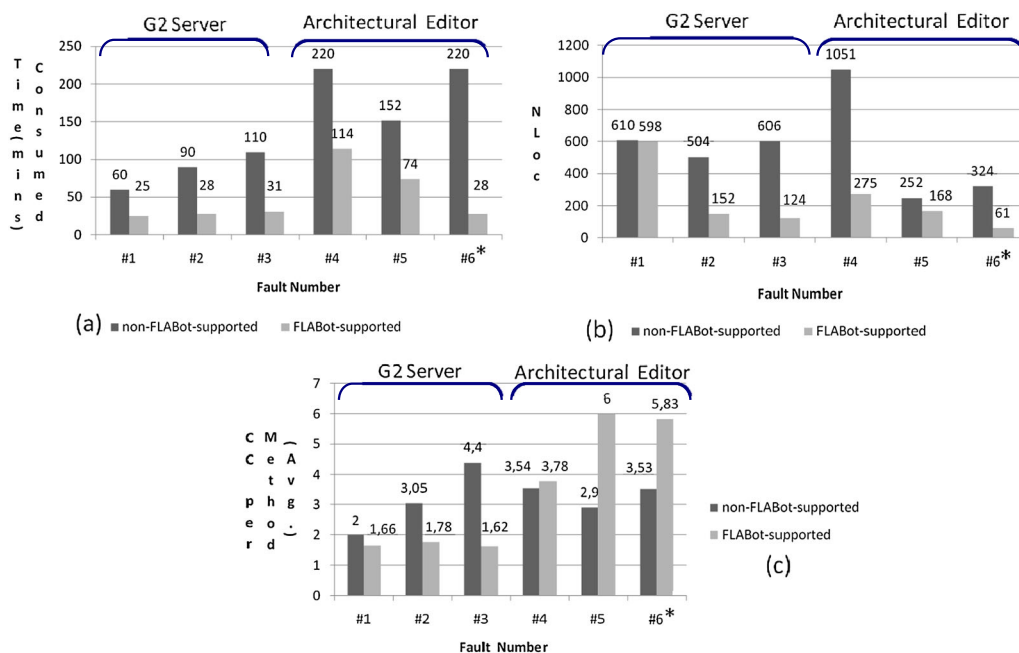


Figure 16: Comparison of the metrics collected from the G2 and Architectural Editor case studies.

However, the opposite trend was observed in the case the *Architecture Editor* faults: the CC values of the *FLABot-supported* participants kept higher than the CC values of the *non-FLABot-supported* participants. We believe the differences of CC values come from the characteristics of the framework supporting each system (see Section 4.4 for more discussion).

4.4. Lessons learned

FLABot indeed facilitates the navigation of the application code and the identification of faults. The participants using *FLABot* reported that having an architectural metaphor (i.e. UCMs) helped them to narrow down the space of possible root causes to search for. The *FLABot* assistant guided the participants to explore a neighbourhood of responsibilities and dependencies in order to approximate faulty Java methods, keeping the participants from an extensive search through the source code. This kind of guidance is particularly useful for novice developers. Despite some lack of precision in the suggestions of the tool, the information recorded from the debugging sessions showed that the participants spent their time focused on specific problems in the code. Furthermore, in post-mortem interviews the participants reported that, as they interacted with the tool, they progressively gained an understanding of the architectural context in which those problems occurred. Anyway, more experiments with subjects and types of faults are needed to generalize these claims.

From the experiments with the different types of faults, we can say that *FLABot* is able to deal with faults either in an application (e.g. the *ArchitectureEditor* or *G2* systems) or in the object-oriented framework supporting that application. The workings of the tool are independent of the architecture under analysis, and the UCM input specification is oriented to behavioural scenarios and not tied to specific architectural patterns (e.g. MVC, event-based systems or any other). In practice, however, there are two constraints to the fault assistance provided by *FLABot*: the UCM modelling efforts and the limited coverage of application traces.

On one hand, as in any documentation process, the architects must take some time to create a base of architectural diagrams that is consistent and provides sufficient information about system functionality. The results of the fault localization depend on these initial specification efforts. Had the UCMs of our two case studies come with a different granularity of architectural components and responsibilities (and thus, with different code mappings), the outputs of *FLABot* would have varied. For instance, very abstract UCMs and mappings (e.g. one-to-many mappings from components to Java classes) often lead to many false negatives (i.e. *FLABot* cannot find the source of the error). Detailed UCMs with inaccurate mappings to code are likely to increase the false positives (i.e. *FLABot* mistakes a correct system behaviour for an error). False positives are costly in terms of time wasted by the developer on re-diagnosis. As rule of thumb for UCM modelling, we recommend architects to avoid very coarse-grained components and responsibilities, in order to make the heuristic more precise regarding the identification of faulty responsibilities and associated code.

On the other hand, the application traces work like ‘test suites’ for the UCMs, providing an approximate (runtime) view of the implemented system. As consequences of using this approximation for the analysis, certain faults might be difficult to expose, and conversely, some misbehaviours cannot be always tracked back to errors. The precision of *FLABot* gets affected, because the tool might misinterpret the actual statuses of responsibilities and give erroneous diagnoses. Anyway, the analysis of runtime information is useful for exposing behavioural patterns between responsibilities that are not easily discernible from method calls in the code (e.g. frameworks such as *Spring* that supports dynamic object composition through XML files).

Furthermore, we observed a relationship between the CC variations and the design characteristics of the frameworks used in the two case studies. The *G2* case study was built on top of a white-box framework (Fayad *et al.*, 1999) in which functionality is mainly reused by applications via class inheritance. On the contrary, the GEF infrastructure for the *ArchitectureEditor* is a black-box framework in which functionality tends to be reused via object composition. In the former case, a developer often looks for faults in concrete application-level subclasses, and then she eventually moves up in the hierarchy to inspect framework-level classes. Because application-level subclasses can have access to the inherited implementation, the complexity of application-level methods is likely to remain low. This fact would explain the low CC values obtained in *G2*. In the latter case, application-level subclasses use well-defined interfaces to deal with the composition and configuration of instances, which often leads to complex method implementations. Thus, high CC values are expected when those methods are inspected during fault localization. This fact supports the results of the *ArchitectureEditor* case study.

4.5. Threats to validity

We have identified a number of threats to validity in our experiments. Regarding the participants’ performance (internal validity), the main threats include the following: (a) the participants’ knowledge of UCM notation; (b) the selection of the faults; and (c) tooling issues such as the code instrumentation or the mixed-initiative schema. As for the generalization of the results (external validity), the main threats have to do with (a) the profiles of the developers chosen for the study; (b) design aspects of the applications being tested; and (c) the limited size of our application sample. We elaborate on these threats later.

In general, the UCM notation is relatively easy to use for developers, although it is not a mainstream notation. A prerequisite for applying *FLABot* (and also an adoption challenge) is the need of architects and developers trained in UCM modelling. In addition, we should recall that the UCM notation admits various ‘modelling options’. Two modelling choices for the same architectural behaviour can have differences in their level of abstraction, and therefore, their mappings to code might also be different. The mappings are also related to the fault estimators, which infer (and abstract) responsibility statuses based on runtime events. Along this line, the abstraction and mapping factors do affect the processing steps, and thus the results of the

fault-localization heuristic. Given the possible levels of abstraction involved, it is difficult to generalize the *FLABot* approach to other systems or architectural specifications. Anyway, we believe that the ‘architectural abstraction’ is a key aspect of the *FLABot* approach, as it allows tracking down types of faults in which the errors and their root causes are in different modules, or faults that involve missing code, among others. In such cases, the architectural structure provides a quicker ‘chain of reasoning’ to spot candidate fault locations than conventional tools based on following long, detailed code dependencies.

For selecting the six faults used in the case studies, we examined the bug tracking systems and looked for faults such as missing code, which are not easily detectable with standard code analysis tools (e.g. *FindBugs*). A limitation in our evaluation is that we injected only one fault per application version under analysis, each fault happening in a narrow code location (i.e. a small set of methods). In the search in the responsibility graph, this kind of faults might lead to one or more (faulty) responsibilities. In the case of multiple faults (at different code locations) responsible for an error, the behaviour of the *FLABot* algorithm is still to be studied.

We also have to factor in the impact of code instrumentation both on the application behaviour and on the size of the logs. Although our instrumentation strategy per UCM has shown no observable effects in system executions, there is still a change to influence certain errors or the order of events logged by the tool. The log size adds overhead to the processing time of the fault-localization algorithm. So far, processing times have been marginal when considered within the whole fault-localization tasks performed by the participants. Experiments to determine the scalability of the tool with bigger logs are still pending.

Regarding the selection of participants, we considered both developers with moderate experience in Java programming in real projects and junior developers. If we had had another group of developers, *FLABot* would have performed differently due to variations in motivation and background of the participants. Also, participants are unlikely to behave in the same way when using the mixed-initiative feature during a debugging session. The results of our evaluation might not generalize to developers who have been maintaining a code base for a while, as they often gain insights about typical ‘problem patterns’. In such cases, a manual code revision could be more effective than using the *FLABot* tool.

In the previous subsection, we pointed out framework design issues related to the working of the *FLABot* tool. A caveat of the CC metric is that it does not take into account the complexity of the interactions in an object-oriented application. Other design issues that might have effects on the tool include the use of third-party libraries and the mapping decisions for libraries or framework classes. Although *FLABot* is good at focusing the developer’s attention on a set of methods relevant to the fault at hand, it does mean that those methods are always easy to understand for the developer. The effort required by a developer to understand how a (relevant) method works varies from one application to another. Application-specific design aspects are currently not taken into account in the *FLABot* assistance.

5. Related work

There are many semi-automated approaches in the literature that seek to reduce debugging efforts by helping developers to test, diagnose and repair problems in software systems. Fault localization falls in the diagnosis category, and the main problem-solving strategy is to depart from an observed error (e.g. a faulty run) and then make predictions about the root causes for that error. Early fault-localization approaches were variations of program-slicing techniques (Korel & Laski 1990; Weiser, 1984; Agrawal *et al.*, 1995). Further developments include: model-based debugging (Köb & Wotawa, 2006), program spectra (Harrold *et al.*, 1998; Renieris & Reiss, 2003) and delta debugging (Cleve & Zeller, 2005). These approaches require different degrees of knowledge about the system’s internal structure and behaviour. A common aspect of all these approaches is their strong reliance on code information (either static or runtime aspects), in order to find potential causes for errors.

Program slicing is a debugging technique proposed by Weiser (Weiser, 1984), in which a program is decomposed into slices for analysis. Each slice is computed for a particular variable and contains all the executable statements that might affect the value of the variable at a given program location. Because these slices are computed at compilation time, they often contain more statements than those that actually affect a faulty run. To narrow down the statements in a slice, Korel and Laski (1990) and also Agrawal *et al.* (1995) extended Weiser’s static concept of slice to a dynamic one, in which a slice contains all the executed statements influencing a faulty run. Studies like (Renieris & Reiss, 2003) and others have shown that faults correlate with differences in traces between a correct and faulty runs. Agrawal *et al.* have looked at differences in execution slices for faulty and non-faulty runs, in order to identify faults that appear only in the code for an execution slice of a faulty run. Nonetheless, because dynamic program slicing is still focused on executed statements, it does not always capture unexpected behaviours caused by the omission of statements that should be part of a correct run. In the *FLABot* approach, the correct runs are given at the architectural level (i.e. the UCMs under investigation), while the faulty runs still operate at the code level.

An improvement to program slicing is the use of program spectra (Harrold *et al.*, 1998). A program spectrum is an execution profile that indicates which parts of a program are active during a run. Fault localization entails identifying the part of the program whose activity correlates the most with the detection of errors. Renieris and Reiss (2003) have developed an approach to select a single run closest to the faulty run (from a set of correct runs), so as to circumscribe the program features that are more likely to contain the fault. Here, different distance-based criteria can be used for the run selection. Alternatively, Cleve and Zeller (2005) proposed the identification of causes by looking at program states. This technique is known as delta debugging. Delta debugging focuses on cause transitions, which are moments in time where a cause originates. A prerequisite of Renieris’s and Cleve’s approaches is the availability of one failing test case and at least one successful test run. Finding a successful test case might be unfeasible if a new version of the program is considered.

When comparing the earlier approaches to *FLABot*, we see two main differences. Firstly, existing approaches are

programming-language specific (e.g. C and Java). Thus, developers are forced to work closely to the system implementation, at the risk of losing the global context of the problem being solved. Secondly, existing approaches cannot always deal with errors caused by missing code because the 'faulty' code is not in the source code, as pointed out in Wong's survey (Wong & Debroy, 2009). Even though the missing code may affect a part of the program and give developers some clues about the fault, these approaches cannot identify the missing code fragment except for some model-based debugging techniques. An adoption barrier for model-based debugging is that developers must understand the logical models used for representing the system behaviour in order to understand the results of the diagnosis. We believe that having a high-level model of the system dependencies (like the one provided by the UCMs) is a better alternative to help developers with their fault-localization (cognitive) process.

Since the early 1980s some knowledge-based approaches (Lukey, 1980; Sedlmeyer *et al.*, 1983; Pau, 1986; Hunt, 1997) have been developed to automate the fault-localization process. These approaches require expert developers to manually provide knowledge about the expected program behaviour and a set of classified faults. The faults and behaviour are specified through assertions for code fragments and faulty situations in the program. These aspects limit the approaches to handle only restricted classes of faults and simple programs (when compared with today's programs). Our approach tries to overcome those limitations by specifying the knowledge about the system behaviour in architectural terms. That is, the intended behaviours are described in the form of architectural scenarios (UCMs), and the assertions as execution conditions for the methods implementing key responsibilities in those scenarios.

From a different perspective, architecture-level dependency analyses have attempted to increase the level of abstraction of fault-localization techniques. Stafford and Wolf (1998) propose a technique, called chaining, that builds graphs of components capturing their relationships in a formal architectural description language (ADL). For a particular issue, these graphs (also known as slices) can support developers in the navigation of the components related to that issue. Zhao (1997) builds slices capturing particular relationships between components and connectors in the Acme ADL. Here, the developer can navigate the components within an architectural slice calculated for a particular component. Unlike *FLABot*, these two approaches consider only components (not responsibilities), and the construction of slices is static in nature. The static analysis may produce more suspicious architectural elements than those actually needed to find faults. In order to reduce the number of suspicious components, some authors take advantage of the 'executable' capability of certain ADLs. A dynamic architecture slicing approach is used by Kim *et al.* (2000) for computing slices based on the sequence of components visited when executing an ADL specification. Particularly aimed at debugging, Wong *et al.* (2005) use program-execution slicing for architectures specified in Specification and Description Language (SDL), based on the similarity between the textual SDL representation and the C language. The SDL statements are prioritized according to their presence in slices of faulty and correct runs. One problem of Wong's approach is that the prioritization depends on the faulty and non-faulty runs

selected by the developer. Different run choices lead to different sets of possible faulty statements. Furthermore, Kim's and Wong's approaches do not consider architecture-implementation relationships like *FLABot* does and offer little assistance to the developer for diagnosing problems detected at the architectural level in the code.

There are also promising applications of model checking to find errors in architectural descriptions. Basically, an error is found by exploring all the execution states of a system and by verifying whether the state violates a given property. These approaches differ in the way they define (or extract) the logical model of the architecture. He *et al.* (2002) compute a finite-state model based on the semantics of an architectural specification, while Garlan *et al.* (2003) build a set of finite-state models containing the general mechanisms of implicit-invocation systems. From the adoption viewpoint, a perceived drawback of model checking is that a developer must be familiar with the logic formalism used to specify properties in the generated model, and she must understand the results generated by the model-checking tool. Recent approaches are working towards lowering this barrier. We think that *FLABot* takes a more practical approach than model checking, because the UCM notation serves both to describe the architectural behaviour and present the diagnosis of errors to the developer, without requiring high levels of formality in the specifications.

6. Conclusions

In this article, we have presented an architecture-driven tool approach to aid fault localization. The *FLABot* tool analyzes architectural responsibilities based on a combination of information from design elements, code and executions of the system. The design information comes from the main architectural scenarios, which are described with UCMs. The code information comes from predetermined mappings between architecture and implementation elements. The system execution information comes from the instrumentation of the application code. *FLABot* is not intended to replace code-level analysis tools, but rather to complement them, by leveraging on architectural information to guide the search for faults. Note that the faults detectable by *FLABot* need to be somehow linked to architectural elements, such as responsibilities and components realizing those responsibilities.

The main contribution of *FLABot* is that it assists the developer in the exploration of a complex object-oriented application and puts marks in code regions that might be responsible for errors. Because it performs a heuristic search, the outputs provided by *FLABot* to developers should be interpreted as hints and not as precise fault locations. The idea is to strike a balance between precision in fault localization and savings in developer's efforts, when she needs to quickly inspect the application. Departing from the marks (breakpoints) generated by *FLABot*, the developer can readily apply conventional debuggers or advanced code analysis tools.

The results of applying the *FLABot* prototype in medium-size case studies, like the ones reported in this article, have been encouraging. Provided the architectural specification of the system, we have evidence that *FLABot* contributes to reduce the time and code browsed to find particular faults, especially when used by novice developers. Furthermore, we observed that the provision of an architectural metaphor such as the UCMs facilitates fault-

localization tasks. The UCMs play a communicational role because reasoning architecturally about faults frees developers from many programming language details. We speculate that senior people (e.g. managers, chief architects, IT architects, etc.) can also benefit from the *FLABot* approach. They can apply their problem-solving skills about faults and solutions and codify this knowledge via *FLABot* rules in order to give advice to less experienced developers.

Nonetheless, *FLABot* still has some limitations and open issues that we expect to address in future versions of the tool. Firstly, the modelling efforts for specifying a consistent set of UCMs along with architecture-implementation mappings are still important. We envision guidelines for determining the minimum architectural and functional aspects of an application that an architect should input into *FLABot*. Secondly, *FLABot* requires the developer to exercise the instrumented application with system scenarios related to each of the faults. An improvement for this problem is a mechanism to associate semi-automated test cases (e.g. *JUnit* tests) with the UCMs. Thirdly, the identification of anomalies in traces (Dallmeier *et al.*, 2005) is another complementary technique for *FLABot* to determine error states in responsibilities. Fourthly, we are interested in considering ‘testability metrics’ for predicting bug-prone code regions (Khoshgoftaar *et al.*, 1995; Giger *et al.*, 2012), so as to inform the fault-localization heuristic regarding responsibilities mapped to those regions. Alternatively, the assistance could be improved by incorporating feedback from previous fault-localization experiences (e.g. via case-based reasoning techniques). At last, we plan to investigate how the *FLABot* approach can inform developers on system areas being sensitive to faults due to incorrect architectural decisions.

Overall, this work argues for the centrality of software architecture throughout the development lifecycle. *FLABot* is an interesting example of how architectural design knowledge, appropriately embedded into a tool, helps people to understand the relationships between errors and faults in a system without a deep knowledge of its implementation. We envision that this kind of high-level analyzers and debuggers will enjoy increasing attention in complex systems, in support for practices such as architecture documentation and conformance checking.

Acknowledgements

This work was partially supported by Intel Corporation as part of a research cooperation project with the ISISTAN Research Institute (UNICEN University). The authors are grateful to Enrique da Costa, Agustin Persson, Franco Scavuzzo and Martin Blech, who greatly helped in the development of the *FLABot* prototype. The authors would also like to thank the anonymous reviewers for their valuable comments to improve the quality of this manuscript.

References

AGARWAL, M.K., K. APPLEBY, M. GUPTA, G. KAR, A. NEOGI, A. SAILER (2004) Problem determination using dependency graphs and run-time behavior models. 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management,

DSOM 2004, Davis, CA, USA, November 15–17, 2004. Lecture Notes in Computer Science, Springer Berlin Heidelberg, 171–182.

AGRAWAL, H., J. HORGAN, S. LONDON, W. WONG (1995) Fault localization using execution slices and dataflow tests, Proc. of the 6th IEEE Int. Symposium on Software Reliability Engineering, Toulouse, France. 143–151.

AMANDI, A., M. CAMPO, A. ZUNINO (2004) JavaLog: A framework-based integration of Java and Prolog for agent-oriented programming. Computer Languages, Systems and Structures. ISSN 0096-0551.

AMBLER, S. (2005) *The Elements of UML 2.0 Style*, Cambridge University Press.

ANDERSEN, B., T. FAGERHAUG (2006) *Root Cause Analysis: Simplified Tools and Techniques*, 2nd edn, Milwaukee, USA: ASQ Quality Press.

BASS, L., P. CLEMENTS, R. KAZMAN (2003) *Software Architecture in Practice*, 2nd edn, Boston, USA: Addison-Wesley.

BORDELEAU, F., D. CAMERON (2000) On the relationship between use case maps and message sequence charts, in: VERIMAG, IRISA, SDL Forum, E. Sherratt (Ed.), 2nd Workshop of the SDL Forum Society on SDL and MSC, SAM, pp. 123–138.

BUHR, R.J.A. (1998) Use case maps as architectural entities for complex systems, *IEEE Transactions on Software Engineering*, **24**(12), 1131–1155.

BUHR, R.J.A. (1999) Making behaviour a concrete architectural concept, 32nd Annual Hawaii Conference on System Sciences, HICSS’99, 05 Jan, Hawaii, USA.

BURKHARDT, J.M., F. DETIENNE, S. WIEDENBECK (1998) The effect of object-oriented programming expertise in several dimensions of comprehension strategies, in *Proceedings of the 6th International Workshop on Program Comprehension*. IWPC ’98.

BUSCHMANN, F., R. MEUNIER, H. ROHNERT, P. SOMMERLAD, M. STAL (1996) *Pattern-Oriented Software Architecture: A System of Patterns*, New York, NY, USA: Wiley & Sons, Inc.

CAMPO, M., J.A. DIAZ-PACE, M. ZITO (2002) Developing object-oriented enterprise quality frameworks using proto-frameworks, *Software Practice & Experience*, **32**(8), 1–7.

CASANOVA, P., B. SCHMERL, D. GARLAN, R. ABREU (2011) Architecture-based run-time fault diagnosis. In *Proceedings of the 5th European Conference on Software Architecture (ECSA’11)*. I. Crnkovic, V. Gruhn, M. Book, (Eds.), Berlin, Heidelberg: Springer-Verlag, 261–277.

CHIBA, S. (2000) Load-time structural reflection in java. *Proc. of the 14th European Conference on Object-Oriented Programming Lecture Notes In Computer Science*, London: Springer-Verlag, 313–336.

CLEMENTS, P., D. GARLAN, L. BASS, J. STAFFORD, R. NORD, J. IVERS, R. LITTLE (2002) *Documenting Software Architectures: Views and Beyond*, Boston, USA: Addison-Wesley.

CLEVE, H., A. ZELLER (2005) Locating causes of program failures. 27th International Conference on Software Engineering, 15–21 May, 342–351.

DALLMEIER, V., C. LINDIG, A. ZELLER (2005) Lightweight defect localization for java. *Proc. of 19th European Conference on Object-Oriented Programming*, ECOOP 2005. Black A.P. (Ed.). Lecture Notes in Computer Science, 3586, Heidelberg: Springer-Verlag Berlin, 528–550.

DIAZ-PACE, J.A., A. SORIA, G. RODRIGUEZ, M. CAMPO (2012) Assisting conformance checks between architectural scenarios and implementation, *Journal of Information and Software Technology*, **54**, 448–466, 2012, ISSN: 0950-5849, doi: 10.1016/j.infsof.2011.12.005.

FAYAD, M., D. SCHMIDT, R. JOHNSON (1999) *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, Chichester; UK: John Wiley & Sons Ltd.

GARLAN, D., B. SCHMERL, J. CHANG (2001) Using gauges for architecture-based monitoring and adaptation, in *Proceedings Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia.

GARLAN, D., S. KHERSONSKY, J.S. KIM (2003) Model checking publish-subscribe systems, Proc. of the 10th International SPIN Workshop on Model Checking of Software (SPIN 03). Portland, Oregon.

GIGER, E., M. D’AMBROS, M. PINZGER, H.C. GALL (2012) Method-level bug prediction, in *Proceedings of the ACM-IEEE*

- international symposium on Empirical software engineering and measurement (ESEM '12). ACM, New York, NY, USA, 171–180.
- HARROLD, M.J., G. ROTHERMEL, R. WU, L. YI (1998) An empirical investigation of program spectra, in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*. Montreal, Quebec, Canada. ACM, New York, NY, 83–90.
- HE, X., J. DING, Y. DENG (2002) Model checking software architecture specifications in SAM, in *Proceedings of the 14th international conference on Software engineering and knowledge engineering*.
- HUNT, J. (1997) Case based diagnosis and repair of software faults, *Expert Systems*, **14**, 100–110.
- KHOSHGOFTAAR, T.M., R.M. SZABO, J.M. VOAS (1995) Detecting program modules with low testability. *Proc. of the International Conference on Software Maintenance (ICSM '95)*, Washington, DC, USA: IEEE Computer Society, 242.
- KIM, T., Y.T. SONG, L. CHUNG, D.T. HUYNH (2000) Software architecture analysis: a dynamic slicing approach, *Journal of Computer & Information Science*, **1**, 91–103.
- KÖB, D., F. WOTAWA (2006) Fundamentals of debugging using a resolution calculus. In *Fundamental Approaches to Software Eng. Lecture Notes in Computer Science*. Heidelberg: Springer-Verlag Berlin, 278–292.
- KOREL, B., J. LASKI (1990) Dynamic slicing of computer programs, *Journal of Systems and Software*, **13**, 187–195.
- KRANSDORFF, A. (1998) *Corporate Amnesia: Keeping Know-How in the Company*, Butterworth-Heinemann.
- LAPRIE, J., B. RANDELL (2004) Basic concepts and taxonomy of dependable and secure computing, *IEEE Transactions on Dependable and Secure Computing*, **1**, 11–33.
- LIEBERMAN, H. (1997) The debugging scandal, *Communications of the ACM*, **40**(4), 15–29.
- LUKEY, F.J. (1980) Understanding and debugging programs, *International Journal of Man-Machine Studies*, **12**, 189–202.
- MOORE, B., D. DEAN, A. GERBER, G. WAGENKNECHT, P. VANDERHEYDEN (2004) Eclipse development using the graphical editing framework and the eclipse modeling framework, IBM Redbook, <http://www.redbooks.ibm.com/abstracts/sg246302.html>.
- MYERS, G.J., C. SANDLER, T. BADGETT, T.M. THOMAS (2004) *The Art of Testing*, 2nd edn, John Wiley & Sons, Inc.
- OTHMAN, R., N.A. HASHIM (2004) Typologizing organizational amnesia, *The Learning Organization*, **11**, 273–284.
- PAU, L.F. (1986) Survey of expert systems for fault detection, test generation and maintenance, *Expert Systems*, **3**, 100–110.
- RENIERIS, M., S. REISS (2003) Fault localization with nearest neighbor queries. In 18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6–10 October 2003, Montreal, Canada, IEEE Computer Society, 30–39.
- SEDLMEYER, R.L., W.B. THOMPSON, P.E. JOHNSON (1983) Knowledge-based fault localization in debugging, in *Proceedings of the Symposium on High-level debugging (SIGSOFT '83)*. New York, USA, 25–31.
- SORIA, A., J.A. DIAZ-PACE, M. CAMPO (2009) Tool support for fault localization using architectural models, in *Proceedings 13th Conference on Software Maintenance and Reengineering (CSMR'09)*. Reengineering Forum (REF) & IEEE Computer Society. Fraunhofer IESE, Kaiserslautern, Germany, 59–68.
- STAFFORD, J.A., A.L. WOLF (1998) Architecture-level dependence analysis in support of software maintenance, in *Proceedings of the third international workshop on Software architecture (ISAW '98)*. Orlando, Florida, United States, 129–132.
- STEIMANN, F., T. ENGELEN, M. SCHAAF (2008) Towards raising the failure of unit tests to the level of compiler-reported errors. In *'Objects, Components, Models and Patterns'*, Lecture Notes in Business Information Processing, Heidelberg: Springer-Verlag Berlin, 11, 60–79.
- SULLIVAN, K.J., J.B. DUGAN, D. COPPIT (1999) Developing a high-quality software tool for fault tree analysis, *ISSRE '99 Proceedings of the 10th International Symposium on Software Reliability Engineering*, IEEE Computer Society, 01 Nov, 222–231.
- WEISER, M. (1984) Program slicing, *IEEE Transactions on Software Engineering*, **10**, 352–357.

- WILKINS, D.E., M. DESJARDINS (2001) A call for knowledge-based planning, *AI Magazine*, **22**, 100–115.
- WIRFS-BROCK, R., A. MCKEAN (2002) *Object Design: Roles, Responsibilities, and Collaborations*, Boston, USA: Addison-Wesley.
- WONG, W.E., V. DEBROY (2009) A survey of software fault localization. Tech. Rep. UTDCS-45-09. University of Texas at Dallas.
- WONG, W.E., Y. QI, T. SUGETA, J.C. MALDONADO (2005) Smart debugging software architectural design in SDL, *Journal of Systems and Software*, **76**, 15–28.
- ZELLER, A. (2006) *Why Programs Fail: A Guide to Systematic Debugging*, San Francisco, USA: Morgan Kaufmann Publishers.
- ZHAO, J. (1997) Using dependence analysis to support software architecture understanding, *New Technologies on Computer Software*, In M. Li (Ed.), International Academic Publishers, pp.135–142.

The authors

Álvaro Soria

Álvaro Soria received the Computer Engineer degree from Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA), Tandil, Argentina, in 2001, and the PhD degree in Computer Science at the same university in 2009. Since 2001, he has been part of Instituto de Sistemas Tandil (ISISTAN) Research Institute, UNCPBA. He is currently a professor at UNICEN University (Tandil, Argentina) and also a research fellow of the National Council for Scientific and Technical Research of Argentina (CONICET). His research interests include software architectures, quality-driven design, object-oriented frameworks and fault localization. Contact him at asoria@exa.unicen.edu.ar

J. Andrés Díaz-Pace

J. Andres Diaz-Pace received the Computer Engineer degree from Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA), Tandil, Argentina, in 1999 and the PhD degree in Computer Science at the same university in 2004. He is currently a professor at UNICEN University (Tandil, Argentina) and also a research fellow of the National Council for Scientific and Technical Research of Argentina (CONICET). From 2007 to 2010, he was a member of the technical staff at the Software Engineering Institute (SEI, Pittsburgh, USA), with the Research, Technology and System Solutions Program. His primary research interests are quality-driven architecture design, AI techniques in design, architecture-based evolution and conformance. He has authored several publications on topics of design assistance and object-oriented frameworks. Contact him at adiaz@exa.unicen.edu.ar

Marcelo R. Campo

Marcelo Campo received the Computer Engineer degree from Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA), Tandil, Argentina, in 1988 and the PhD degree in Computer Science from Instituto de Informática de la Universidad Federal de Rio Grande do Sul (UFRGS), Brazil, in 1997. He is currently an Associate Professor at Computer Science Department and Director of the ISISTAN Research Institute of the UNICEN University in Tandil, Argentina. His research interests include intelligent aided software engineering, software architecture and frameworks, agent technology and software visualization. Contact him at mcampo@exa.unicen.edu.ar