A framework for monitorable services implementation

David Cardoso
Universidade Federal de Pernambuco
Recife, Pernambuco
Email: dmc4@cin.ufpe.br

Augusto Sampaio
Universidade Federal de Pernambuco
Recife, Pernambuco
Email: acas@cin.ufpe.br

Abstract—Since the very first graphical user interfaces, progress indicators have been widely used to provide feedback regarding the execution of long-running system tasks. In order to make progress feedback possible, the involved services must provide on-line monitoring capabilities. However, on larger concurrent and object-oriented implementations, as services execution involve multiple interactions between various components and abstraction layers, the crosscutting nature of the monitoring concerns can introduce some challenges to the software development — such as code quality degradation, absence of standardized code patterns, and loss of development productivity. In this context, after analyzing some possible solutions to the monitoring problem, we propose a general framework to support the development of monitorable services, as well as some extended libraries that are used to illustrate a concrete implementation. We also evaluate the proposed solution through a real case study performed in a private software development organization.

I. INTRODUCTION

The execution of a software service can require more processing time than originally expected by the service consumer (possibly the end-user). In other cases, the service consumer may already expect a time-consuming execution. In both cases, despite expectations, some applications display information regarding the progress of the service execution, by explicitly displaying a percentage of progress or by using graphic representations such as progress bars, as shown in Figure 1 along with other monitoring information such as the elapsed time, an estimate of remaining time, a brief description of the current task being performed, etc.

Fig. 1. Progress bar example

The usage of percent-done progress indicators in information systems is not new. In fact, it dates from the very first graphical user interfaces [1]. Practical experience and formal experiments show that progress indicators are an important and useful user interface tool, and that they enhance the attractiveness and effectiveness of programs that incorporate them [2], [3].

Despite the advances on computer performance in general, in the last decades, which led to better processing response times, still more recent formal experiments [4] demonstrate that progress indicators continue to be an important graphical technique for improving human-computer interactions, with substantial impact on users’ decisions to continue rather than break off software usage, therefore explaining why percent-done progress indicators are so much widespread in current information systems.

Nowadays, the act of displaying a graphical percent-done indicator in a user interface is often considered a task of trivial complexity, even for novice software developers. This is due to the immense variety of easy, ready-to-use and self-contained components that handle the calculations, scaling and perform all the painting — such as the JProgressBar Class of the Java API [5] and the ProgressBar Class of the Microsoft .NET Framework [6]. With most of these components, all the software developer needs to do is to write code for updating the progress bar component with the current progress percentage.

These components often enable very effective reuse and play an important role on the user-interface part of the monitoring concern. However, despite their usefulness in facilitating renderization in the user interfaces, such components are not concerned with the difficulties involved in gathering on-line execution data and translating them into a single overall percentage. Nevertheless, on larger and more complex software systems, this is exactly when the difficulties may arise: services execution may involve the interaction of many different components, layers and subsystems, and therefore it can require very hard work to gather detailed information upon the execution of lower-level layers, control the whole execution chain, and translate them all into a single overall progress percentage, readily available on a higher-level layer such as a user interface progress bar.

We use the term monitorable service to denote services that explicitly enable their consumers to extract execution information, such as the percentage of overall completion along with business-specific notification messages, making it easy for providing on-line feedback to the end-user, registering...
information on systems’ event log for later verification, and estimating remaining execution time.

Unfortunately, however, reuse opportunity is very limited when it comes to providing the software development community with solutions specifically designed to aid the development of such monitorable services. Even though the progress indicators are often present on information systems, the development strategies being adopted are often non-systematic and lack standardization.

The main difficulties regarding monitorable services implementation relate to (1) concern modularization; (2) code standardization; (3) development productivity; (4) increasing maintenance costs; (5) estimating and coordinating tasks and subtasks; and (6) generating accurate and uniform monitoring results. In fact, these concerns only become of real matter on large systems, with complex architectures, thousands of lines of code, and involving many developers, as further explained in later sections.

As a result of this work, some contributions can be enumerated: (1) a characterization of the monitoring problem, by identifying some possible approaches and summarizing their main benefits and weaknesses; (2) a proposed solution to the monitoring problem, consisting of a generic framework along with concrete libraries and code patterns; and (3) an evaluation of the proposed solution through a case study performed in a private software development organization.

In Section II we discuss some possible approaches and related work regarding the monitoring problem. Section III presents a proposed solution consisting of a generic framework and extended libraries, while Section IV summarizes the case study performed to evaluate the proposed solution. Finally, in Section V we draw some final remarks, along with future works.

II. POSSIBLE APPROACHES AND RELATED WORK

In order to characterize the monitoring problem, we summarize four possible approaches: (1) Ad-hoc; (2) Eclipse; (3) Aspect; and (4) Metadata. The approaches (1) and (2) are actually adopted in real-life projects, while the approaches (3) and (4) were devised as a result of our technical discussions. Although there are certainly many other approaches to address the monitoring concern, we consider these sufficient for representing the nature of monitorable service implementations.

A. The Ad-hoc Approach

We use the term ad-hoc approach to designate the process of implementing monitorable services without the systematic adoption of reuse. In this case, no standards, patterns, frameworks, APIs nor tools are used to support the software development — or, at least, none specially designed to deal with the monitoring concerns.

On simple systems — no distribution, few layers and short call stacks —, or in a situation where the system under consideration is already developed and the monitoring requirement is needed just for one or two services with low complexity and implemented as a few lines of code, it may be more practical and productive to adopt the ad-hoc approach.

However, the adoption of the ad-hoc approach is often unconscious. Software designers often focus on business and functional requirements and simply do not realize the cross-cutting nature of the monitoring concern and its potential for increasing code complexity. Listing 1 shows a fictitious example — for didactical purposes — that updates a graphical percent-done indicator in a user interface.

```java
void executeService() {
    lbl.setText("Starting operation...");
    progressBar.setValue(0);
    progressBar.setTotal(100);
    // (...) business-specific code here
    lbl.setText("Accessing database");
    // (...) business-specific code here
    progressBar.setValue(10);
    int i = 0;
    for (Thing thing : thingList) {
        // (...) business-specific code here
        i++;
        lbl.setText("Doing thing... + i);
        progressBar.setValue(10 + i*90/thingList.size());
    }
    progressBar.setValue(100);
    lbl.setText("Operation finished");
}
```

Listing 1. Example of ad-hoc code updating a progress bar

In a real scenario, such ad-hoc implementations could meaningfully increase in complexity, if the business logic involved chained execution, concurrency, alternative flows, etc. Although we could mention many weaknesses of such ad-hoc implementations, let us point out only a few:

- **Core business decoupling.** It does not stimulate decoupling of core business logic and monitoring-related code.
- **Support for layered and modularized architectures.** Large systems often implement their business-specific rules on a multi-layer (or multi-component) architecture [9]. The ad-hoc approach does not deal properly with modularized business code, as it can be hard for the lower-level components to share detailed monitoring data with the higher-level components.
- **Limited reuse.** Without making use of any reusable patterns or components, the code becomes replicated among distinct monitorable service implementations (scattering).

Therefore, a non-systematic approach will rarely handle the monitoring problem in a proper manner, except for very simple scenarios such as the one shown in Listing 1.

B. The Eclipse Approach

The Eclipse API [9] is a feature-rich component library providing the Java Development Community with a variety of tools for software development common use. Among its various features, it provides some facilities to the implementation of progress execution feedback to the end-user. The Eclipse approach relies on the concept of Progress Monitors, provided by the Eclipse API. A Progress Monitor is a callback interface that allows a long-running task to report execution
progress and respond to cancelation request. Typically, a UI Component (the consumer) will create a monitor instance and pass it to a low-level component (the long-running service provider) that does not know or care about the UI. Thus, this callback interface is an abstraction that allows for decoupling of UI and non-UI components [10]. The IProgressMonitor interface is shown in Figure 2 [11].

Any monitorable service implementation needs to receive an IProgressMonitor as parameter in order to send progress feedback data. During the monitorable service execution, the service provider needs to properly invoke the monitor interface methods in order to report the current execution state. All the interface methods are intended for the service provider to actively invoke. As an effect of each method call, the service consumer becomes aware of a new progress state update being reported. Therefore, the service provider is absolutely active, while the service consumer is absolutely passive — as it does nothing but receive the progress reports. Listing 2 shows a fictitious code.

```java
void doThings(List<Thing> thingList, IProgressMonitor monitor) {
    boolean canceled = false;
    try {
        monitor.beginTask(thingList.size())
        for (Thing t : thingList) {
            if (monitor.isCanceled()) {
                canceled = true;
                break;
            }
            t.doSomething();
            monitor.worked(1);
        }
    }
    finally {
        if (canceled)
            monitor.setCanceled();
        else
            monitor.done();
    }
}
```

Listing 2. Example of implementation using IProgressMonitor

Each monitor instance must comply with a strictly defined lifecycle, as shown in Figure 3. One might notice that not all the monitor interface methods appear in the state diagram — isCanceled, setTaskName and subTask were suppressed. The suppressed methods are not mandatory in the monitor lifecycle, since they play a secondary role in the monitor protocol. In addition, they do not trigger any state transition.

Long-running operations often reuse other operations as part of their implementation. In a typical object-oriented architecture, methods that represent higher-level services have their implementation based on a deep method invocation chain where each called method represents a lower-level service. Similarly, in service-oriented architectures, services are expected to be built on top of other services, as an aggregation of lower-level services [12]. For dealing with chained method calls, the Eclipse API provides a specific type of monitor that attaches to the main monitor and works as a subtask of the main task, the SubProgressMonitor class [13]. Listing 3 shows an example of how it works.

```java
void transfer(Account targetAccount, double value, IProgressMonitor monitor) {
    try {
        monitor.beginTask("Transferring...", 100);
        SubProgressMonitor debitSubMonitor = new SubProgressMonitor(monitor, 50);
        try {
            this.debit(value, debitSubMonitor);
        }
        finally {
            debitSubMonitor.done();
        }
        SubProgressMonitor creditSubMonitor = new SubProgressMonitor(monitor, 50);
        try {
            targetAccount.credit(value, creditSubMonitor);
        }
        finally {
            creditSubMonitor.done();
        }
    }
    finally {
        monitor.done();
    }
}
```

Listing 3. Example of cascade monitoring

Although the Eclipse approach provides some facilities for implementing monitorable services, by decoupling service consumers and providers, facilitating cancelation handling and supporting subtasks, it still has some drawbacks, including the following:

- **It degrades business code legibility.** Creating and finishing tasks, calculating total work items, creating try-finally blocks, verifying cancelation state and updating task worked items. These concerns require some lines of code that have nothing to do with the core business implementation, making the resulting code more difficult to understand and maintain.

- **It affects method signatures.** Every monitorable method must receive its own progress monitor. This rule is very "architecture-intrusive", as it forces changes to method specifications by adding a parameter that does not directly relate to the core business implementation.
• It does not support distributed service monitoring. The strategy is based on an interface callback which is passed by reference to the service provider. In most distributed service-oriented architectures, callbacks and object references are simply not supported due to the cross-platform difficulties that typically occur on distributed scenarios.

• It requires explicit binding of tasks and subtasks. The majority of the code implementation for a service runs in sequence, not concurrently. Thus, if a new task is created while the previous one is not yet finished, in a non-concurrent environment, the new task is obviously a subtask of the previous one. The API does not track these relations automatically and the programmer has to explicitly link the subtasks to the main monitor.

• It does not track tasks finished with errors. Almost every service implementation has fault or error conditions which may lead to execution abrupt termination. The monitor lifecycle does not distinguish the tasks completed with errors from those completed with success.

• It depends on work items estimates. Progress feedback reporting is based on calls to the worked method. Since there is no automation, the programmer needs to reasonably distribute work items among the calls to the worked method so that the progress execution is reported uniformly along the whole task execution.

C. The Aspect Approach

According to Kiczales [7], object-oriented programming does not completely avoid code scattering and tangling when dealing with crosscutting concerns, irrespective of the efforts towards good design and recommended patterns. In fact, code tangling and scattering clearly occurs on monitorable service implementations. Therefore, one could adapt the Eclipse approach by defining one single aspect that affects every monitorable method. The aspect could be like that presented in Listing 4.

```java
aspect MonitoringAspect {
  pointcut monitorableMethods(): ...

  + around(IProgressMonitor monitor) :
    monitorableMethods() {
    if (monitor == null)
      monitor = new NullProgressMonitor();
    try {
      int methodTotalCost = ... // get the total cost
      monitor.beginTask(methodTotalCost);
      return proceed(); // business-specific logic
    }
    finally {
      if (monitor.isCanceled())
        monitor.setCanceled();
      else
        monitor.done();
    }
  }
}
```

Listing 4. Monitorable aspect

Thus, by using the MonitoringAspect, the example in Listing 2 would be rewritten as in Listing 5.

```java
void doThings(List<Thing> thingList, IProgressMonitor monitor) {
  if (monitor == null)
    monitor = new NullProgressMonitor();
  for (Thing t : thingList) {
    if (monitor.isCanceled()) break;
    t.doSomething();
    monitor.worked(1);
  }
}
```

Listing 5. Monitorable service using aspect

In the given examples, the monitor instance is shared by both the monitorable method and the advice implementation. The advice implementation focuses on the monitoring-related overhead that happens before and after the method executes. Using aspect-oriented programming improves code modularity, as the monitoring code is meaningly separated from the business-specific code. However, one should consider two difficulties to deal with:

• It is hard to generalize. Since the aspect is generic, it should affect various monitorable services in the whole application. This means it cannot make any assumptions about the amount of work and the nature of the monitorable method. Setting the total amount of work to the monitor is a task intimately coupled to the method semantics and inputs. Therefore, we would need additional architectural elements for the aspect to gather abstract information from the concrete method, such as the total amount of work needed by the method. One can overcome this difficulty by defining an abstract aspect implementation concerned only with the main monitoring-related code, and then inheriting concrete aspects according to business-specific needs. Therefore, each concrete aspect implementation would properly handle the cost assignment. Alternatively, it is possible to use metadata-based conventions [14], such as annotations, and associate monitoring-related data to the monitorable methods, readily available for the generic aspect implementation to access — this approach is discussed further.

• Not all subtasks are methods. Aspects use pointcuts to determine points of additional code injection, while pointcuts define sets of methods to crosscut. The problem relies on establishing a one-to-one mapping between method calls and tasks. For example, one could create many subtasks within a single method implementation, without necessarily calling a new method for each subtask. In this case, a single method call should generate a whole tree of subtasks. It is possible to overcome this difficulty for new service implementations, by introducing a coding convention that guarantees that every subtask is implemented as a new method invocation, forcing such a one-to-one mapping. For turning existing service implementations into monitorable services, this approach can require some refactoring, but it is still possible.
D. The Metadata Approach

One notable characteristic of the previous approaches is that they do not impose very specific coding styles, therefore enabling the developers to implement the business logic in their preferred way. This is possible because all the monitoring-related logic are made explicit by adding code for invoking methods provided by the monitoring library. However, despite the benefits of free coding style, the resulting monitoring-related code is often tangled with the core business code.

The Metadata approach is based upon specifying coding conventions that allow the monitoring library (classes and aspects in charge of monitoring) to dynamically infer the monitoring context at runtime, by gathering context-specific information from metadata elements associated to the language static structures (such as classes and methods), by making use of reflection and annotation techniques. The idea is based upon the principles of Convention over Configuration [15] and Attribute-oriented programming [14].

Most of the code tangling on the presented approaches derive from generating subtasks and updating the progress execution status. In order to eliminate it, one can establish a coding convention to guarantee that every atomic task (something that contributes to the overall progress execution) will be implemented as a separate method call. Therefore, by using this convention, it is possible to define a one-to-one mapping between the method calls and the monitoring subtasks. The Metadata Approach can be combined to any of the other approaches, including the solution proposed in the next section, by introducing some conventions and coding style. By using metadata in conjunction with the aspect-oriented approach, the example presented in Listing 3 would be rewritten as in Listing 6.

```java
@Monitorable(operation="Transferring...", cost=100)
void transfer(Account targetAccount, double value, IProgressMonitor monitor) {
    this.debit(value, monitor);
    targetAccount.credit(value, monitor);
}

@Monitorable(operation="Processing_debit...", cost=50)
void debit(double value, IProgressMonitor monitor) {
    ...
}

@Monitorable(operation="Processing_credit...", cost=50)
void credit(double value, IProgressMonitor monitor) {
    ...
}
```

Listing 6. Example of monitorable service using metadata

For the code in Listing 6 to work properly, one would need to intercept every method invocation marked with the Monitorable annotation. This can be performed by an aspect-oriented approach, code weaving or instrumentation technique. Once intercepted at run-time, it is possible to reflectively gather the monitoring cost and operation description from the annotation attributes. Although this approach can meaningly improve code modularity and legibility, we still identify two major drawbacks:

- Costs can depend on the execution context. Annotations are designed for associating static data such as constant values. However, sometimes, the total cost of a method can depend upon the parameters and other contextual information which is only available at runtime, just as in the example of Listing 2 in which the total cost depends upon the size of a list. With an aspect-oriented approach, it is possible to deal with such a situation by using aspect inheritance, by implementing a specific aspect for the method. This would resolve the code tangling problem, but compromises regularity: some monitorable methods will only require an annotation, while others will require a custom aspect extension. Even if the costs can be assigned constant values regardless of the parameters, its cost can depend upon the context in which it is being invoked. For example, although the credit has been assigned a cost of 50 in the context of a transfer operation, it could be assigned a cost of 10 in the context of another financial operation. Again, when combined with an aspect-oriented approach, it is possible to deal with this problem by extending a customized aspect and carefully specifying the pointcut with cflow [16] option, leaving the cost assignment as an aspect decision, but it would add complexity and compromise regularity.

- It is difficult to implement monitoring a posteriori. Since the Metadata Approach relies on coding conventions, such as requiring developers to split up methods in a specific manner whenever they need to provide progress reports, it can be difficult to turn an existing non-monitorable implementation into a monitorable one, possibly requiring lots of manual refactoring.

III. PROPOSED SOLUTION

Our proposed solution to address the monitoring concern is organized in three distinct components: (1) generic framework; (2) service provider library; and (3) service consumer library. The framework specifies a set of abstract classes for representing the core monitoring-related concepts, defining the base protocol that rules consumer-provider interactions. The extended libraries provide concrete implementation logic for managing progress states, aggregating subtasks, and generating progress feedback.

A. The framework

The framework provides the core concepts in the domain of execution monitoring by defining classes and interfaces that represent generalized abstractions and should be reused and extended by applications that need to enable monitoring support. The framework aims at serving as basis for general development scenarios. Therefore, its usage does not impose specific algorithms, heuristics or design assumptions. It supports distribution and concurrency while introducing no specific coding conventions. Different applications in different contexts may want to add additional features and change the default behavior regarding execution monitoring.
Since it is designed to minimize restrictions to the coding style, it should be easy to adjust and adapt an existing non-monitorable implementation, in order to make it monitorable, by using the framework to support it. For this requirement to hold, a monitorable system architecture should not be distinguishable from a non-monitorable system architecture. By architecture, we mean the system’s layers, boundaries, facade interfaces and core contracts.

The framework model relies upon the concept of Operations. Non-trivial services often have their execution logically divided into Operations (steps) which could be individually monitored by a given service consumer. Operations are a set of tasks or subtasks performed by a service during its execution. Operations are semantically complete and independent from the major service semantics. For example, an application service that performs updates to a database after gathering information from the internet can be logically divided into some Operations whose execution could be monitored by the end-user: establishing connection to the internet; performing authentication; downloading web pages; processing information; establishing database connection; creating database transaction; executing SQL queries; etc. Operations have their own inputs and outputs; they start and finish execution and work as a monitorable service inside another service. The Operation concept is part of the framework fundamental package, shown in Figure 4.

The IOperation interface is the foundation for the monitoring framework. Every monitorable service, even the most trivial one, must have exactly one operation (the main operation) representing the overall execution progress of a service. Every other operation will be a child (or descendant) of the main operation. Given an IOperation, it is possible to monitor its state changes and track the execution progress. The framework supports concurrency by allowing operations to contain as many concurrent logical suboperations as desired. It supports distribution by allowing asynchronous message exchange instead of direct method calls (service providers need no direct object reference to consumers, but requires a channel to deliver the messages).

The ProgressState class represents a snapshot of the execution progress of an operation in a given instant. It provides the amount of steps done (work items) and the total steps needed for the operation to complete. These two values, together, determine the percentage of completeness of the operation.

One can register itself as observer to receive updates to a given IOperation instance. We use the Observer Pattern [17]. By extending the IOperationStartSubject interface, the IOperation interface notifies all registered observers of whenever a new child operation starts execution. In addition, by extending the IOperationStateSubject interface, it notifies registered observers of whenever the operation progress changes — which can be a direct result of updating the operation, or a cascading effect of the updates to the child operations. All the notifications are asynchronous, so not to interfere on the core business application performance.

The framework establishes a predefined lifecycle for the service providers to report current operation states to the consumers. The EOperationState enumeration specifies the seven possible states for an operation. Figure 5 illustrates these states along with all possible state transitions.

Core concepts to the framework, besides representing operations and states, are the interfaces that enable consumers and providers to communicate and exchange monitoring information. Figure 6 presents the main entities for the consumer-provider interactions. The IServicesMonitor interface represents the boundary between consumers and providers. Consumers receive monitoring information through an instance of IServicesMonitor. Providers also update monitoring information through an instance of the same IServicesMonitor. By using a monitor, one can register itself to be notified whenever
a service main operation starts (the monitor interface also extends IOperationStartSubject). It allows accessing the IOperation instance for the main operation and allows consumer to send cancelation requests to the running operation. The IMonitorableServicesProvider interface must be implemented by every Service Provider with monitorable services, forcing it to provide an IServicesMonitor instance for the consumer to interact. For example, a class implementing a Facade interface \[17\] should also implement the IMonitorableServicesProvider interface if its services are monitorable, as shown in Figure 7. Typically, there is a one-to-one relation between IMonitorableServicesProvider and IServicesMonitor.

Since the framework and libraries were implemented on top of Microsoft .NET Framework \[18\], using the C# programming language \[19\], we make use of the using statement in order to make the resulting source code cleaner. Although we use a specific language construct, it is easily convertible into a try-finally statement, as explained in \[20\], and we only use it for the convenience of improving legibility and reducing the amount of lines of code.

The service provider library relies on the idea of creating explicit code blocks for each monitoring operation. As the code blocks can be naturally chained, it is possible to seamlessly bind parent and child operations based upon the execution call stack. Listing \[7\] shows the main code pattern for creating operations.

```csharp
void GeneralService()
{
    using (CustomOperation mainOperation = this.Monitor.
        createOperation("Processing...", 100)) {
        // (...) Business-specific logic
        mainOperation.setSucceeded();
    }
}
```

Listing 7. Main code pattern for monitorable services implementation

One can create as many operation blocks as desired. The library classes provide some facilities for that purpose, such as the createOperation method. Suboperations can be created both as a natural result of the execution call stack, or by explicitly opening chained code blocks within the same monitorable method implementation. Progress updates to the suboperations are recursively propagated to the parent operations. The monitor engine provides the progress aggregation logic and computes the overall progress percentage. Therefore, the credit/debit example of Listing \[4\] could be rewritten as in Listing \[8\].

```csharp
void transfer(Account targetAccount, double value)
{
    using (CustomOperation mainOp = this.Monitor.
        createOperation("Transferring...", 100)) {
        this.debit(value);
        targetAccount.credit(value);
        mainOp.setSucceeded();
    }
}
```

Listing 8. Example of seamless suboperation

In the example of Listing \[8\] the costs of the debit and credit operations are both assigned to 50, while the operation of the transfer method is assigned to 100. In fact, the sum of the
suboperation costs must be the same as the cost of the parent operation, so that the resulting progress percentage is properly calculated. However, such a restriction creates a very fragile dependency between the three method implementations. Such kind of code should never be encouraged, because it can easily generate bugs when different developers happen to change the code. The reason is that it violates a basic principle of design: service provider implementations should make no assumptions regarding the possible service consumers. In this case, both credit and debit implementations use 50 as cost because they are supposed to be called from within the transfer method. This problem becomes meaningly amplified if the involved methods are reused by multiple consumers with distinct execution contexts. In order to deal with this situation, the library provides facilities to bind execution costs at the invocation level by using the method createChildChainedOperation. In this case, the example should be rewritten as in Listing 9.

```csharp
void transfer(Account targetAccount, double value) {
    using(CustomOperation mainOp = this.Monitor.
        createOperation("Transferring...", 100)) {
        mainOp.createChildChainedOperation(50);
        mainOp.createChildChainedOperation(50);
        targetAccount.credit(value);
        mainOp.setSucceeded();
    }
}
void debit(double value) {
    using(CustomOperation mainOp = this.Monitor.
        createOperation("Debits...", 100)) {
        mainOp.setSucceeded();
    }
}
void credit(double value) {
    using(CustomOperation mainOp = this.Monitor.
        createOperation("Credits...", 100)) {
        mainOp.setSucceeded();
    }
}
```

Listing 9. Example of seamless suboperation

Since the progress reports are performed by an event-oriented approach which is handled by the monitor, the monitorable method signatures are not affected. The library also provides classes to act as observers of the monitor and automatically display the progress state updates and notifications. Figure 8 shows the consumer library class diagram.

The NotificationsViewer component provides a graphical interface for displaying services notifications. It displays a tree of notifications generated by the service provider while running a service. Each tree node represents a notification, and its color and icon changes according to the notification type and state, as shown in Figure 9.

The ProgressViewer component can be easily added to a user interface of any system and bound to any object implementing IMonitorableServicesProvider. Once added to the application, it receives updates and displays the currently running operation as a progress bar, as shown in Figure 10.

Let us point out some facilities provided by the proposed solution: (1) support for layered and modularized architectures; (2) reduced code scattering (since the core monitoring logic is encapsulated within the framework and library logic); (3) standardized code patterns for creating operations; (4) reduced code tangling (through compact code blocks); (5) minimal impact on application architecture (since it does not affect method signatures); (6) support for concurrency and distribution; (7) seamless binding of operations and suboperations based on the call stack; (8) native support for cancelation and error handling; (9) free coding style (since it does not add constraints to method implementations).

Since the framework is only concerned with generic monitoring concepts, one can easily extend it in order to create a customized monitoring strategy, or introduce different approaches that can be optimal for specific development contexts. This characteristic allows the proposed solution to incorporate other approaches. For example, one could extend it to implement both the Aspect and Metadata approach, while still reusing the core monitoring logic for notifying and aggregating progress updates.

IV. Evaluation

In order to evaluate the effectiveness of the proposed solution, we performed a case study at SUATI [21] — Suporte Avancado em Tecnologia da Informacao —, a software development company based in Recife, during the months of July
and August 2011. The company has nearly 8 years of experience on the development of software systems. The project in the context of which the case study took place consists of more than 600k lines of code (LoCs). The company development team is organized in four autonomous development cells, each one containing a leader and nearly 8 software developers. One of the cells was about to start the development of a new system module where the monitoring requirement was considered important, because the project involved complex risk analysis algorithms, time-consuming simulations of financial scenarios and database-intensive operations, which take significant time to execute. With respect to the monitoring requirement, the final goal was to enable the display of a percent-done indicator along with the remaining execution time estimates during the risk analysis simulation.

The major objective of the study is to analyze the efficiency of the solution in supporting the implementation of monitorable services, with focus on productivity, code tangling and monitoring consistency and quality. Monitorable services implementation only becomes a real concern in the context of medium to large systems, with thousands of lines of code and hundreds of developer working hours, involving time-consuming services for which the users expect to receive on-line progress feedback. The case study provided a real-life scenario and thus the proper circumstances where monitoring can be applied.

We defined three metrics as basis for the case study: (1) the time spent on implementation; (2) number of lines of code (LoCS); and (3) service execution time. These metrics were used to base the case study hypotheses, serving as indicators of (1) productivity; (2) relative code tangling; and (3) monitoring consistency. For data gathering and analysis of these metrics, we used three professional tools: (1) Microsoft Team Foundation Server 2010 [22] — which is Microsoft’s source code repository, activity and version control system, helping on tracking and registering the implementation time spent by the team; (2) The “code metrics” tool of Microsoft Visual Studio 2010 Ultimate Edition [23] — which is Microsoft’s generic IDE for development, assisting on counting the lines of code of the system under development; and (3) the EQATEC Profiler [24] — which is a professional tool for profiling and analyzing execution time of the methods and call stack during application execution. The team leader was responsible for collecting and summarizing the data gathered from the developers.

Before we could draw testable hypotheses, we needed to establish an objective criteria for determining what “an acceptable monitoring implementation regarding productivity and code quality with respect to tangling” actually means. All we knew was that the ad-hoc approach was definitely not sufficient. To gain sensibility over the metrics, we have gathered information about the quality and development time of recent company projects involving implementation of monitorable services to some extent, considering the current and the last year, and involving distinct company development teams. We analyzed 8 distinct projects towards the metrics, and used the best project values as reference to compare our proposed solution. This led us to the following null hypotheses (the ones intended to be rejected [25]):

\[ H_{0a} : \frac{\mu_{\text{monitoring\_implementation\_time}}}{\mu_{\text{total\_implementation\_time}}} > 0.015 \quad (1) \]
\[ H_{0b} : \frac{\mu_{\text{monitoring\_implementation\_LoCs}}}{\mu_{\text{total\_implementation\_LoCs}}} > 0.01 \quad (2) \]
\[ H_{0c} : \left| \frac{\mu_{\text{estimated\_execution\_time}}}{\mu_{\text{actual\_execution\_time}}} - 1 \right| > 0.07 \quad (3) \]

The monitorable services were part of a module corresponding to nearly 50k LoCs. The set of methods that corresponded to the monitorable service implementation ended with precisely 14,491 LoCs and thus they represent a significant portion of the module. In fact, those services represent the heart of the module, which is to perform the calculations and risk analysis simulations under thousands of generated scenarios.

As expected from large systems with an object-oriented design, the execution of a higher-level service is based upon the execution of various smaller services on the lower layers. Different objects and instances communicate, exchange data and request processing in order for a service to properly execute. For this reason, the 14,491 LoCs are spread among 1,087 different methods. During the execution, one method calls others and so on, extending the execution call stack. During the service execution, we observed a maximum call stack level of 24 methods in the execution chain. Since there was no recursion, this number was considered high by the company developers, when comparing to the typical call stack length in other previous projects they participated. More detailed information about the case study is available in [26].

Let us analyze the results obtained for each of the three metrics under evaluation:

- **Code quality (with respect to tangling).** The monitoring implementation using the proposed solution required 124 LoCs. In comparison with the total 14,491 LoCs required for the entire monitorable service implementation, it represented 0.86% of the entire code, which is less than the 1.5% established as our acceptance threshold. Thus, the null hypothesis \( H_{0a} \) is rejected.

- **Implementation time.** The entire service implementation required 860 hours spread upon 26 working days. The monitoring-related implementation consumed 5 hours, representing 0.58% percent of the total effort and less than the 1% established as our acceptance threshold. Thus, the null hypothesis \( H_{0b} \) is rejected.

- **Monitoring quality.** The worst monitoring estimate observed during the service execution predicted the execution to take 941 seconds while it actually took 923 seconds, resulting in a deviation of 1.9%, which is less than the 7% established as our acceptance threshold. Thus, the null hypothesis \( H_{0c} \) is rejected.

The case study succeeded in testing the solution, although it is not sufficient to derive generalized conclusions regarding its
effectiveness for other development contexts. Figure [11] shows a screen snapshot of the system considered in the case study, displaying a percent-done indicator while the server processes the simulation.

![Screen snapshot of the system while the simulation is processing.](image)

Fig. 11. Snapshot of the system while the simulation is processing.

V. Conclusion

We characterized the monitoring problem by summarizing its main development difficulties and discussing some possible implementation approaches. We also presented a solution consisting of a generic framework along with concrete extended libraries to aid the development of monitorable services. Finally, we described a case study that evaluated the proposed solution in the context of a private software development organization.

Related work and some alternative approaches have been discussed in Section [II]. In summary, in the context of large and complex applications, monitorable services implementation often requires some systematic support. Therefore, the ad-hoc approach is not viable for such cases. The Eclipse approach represents a great step towards code standardization and decoupling of service providers and consumers, but it still has some issues related to code tangling, distribution and binding of tasks and subtasks. With respect to the Aspect and Metadata approaches, both can completely eliminate code tangling and automate the binding of tasks and subtasks, but at the price of imposing a specific coding style and introducing constraints to the organization of methods, so to enable a one-to-one mapping between tasks and method calls.

In spite of the commitment to develop the monitoring solution, some possible improvements and related work were identified: (1) optimize code quality, by completely eliminating code tangling between business and monitoring-related code; (2) provide means of automating operation cost assignment, instead of hardcoding numbers which are difficult to estimate and maintain, either by algorithm static analysis or by dynamic recalibration of costs at runtime through profiling techniques; (3) create native programming language constructs in order to deal with monitoring at the programming language level; and (4) evaluate the described approaches, as well as the proposed solution, in distinct software development contexts, involving distinct programming languages and development teams.

REFERENCES