ICT PROJECT 258109
Monitoring Control for Remote Software Maintenance

Project number: 258109
Project Acronym: FastFix
Project Title: Monitoring Control for Remote Software Maintenance
Call identifier: FP7-ICT-2009-5
Instrument: STREP
Thematic Priority: Internet of Services, Software, Virtualisation
Start date of the project: June 1, 2010
Duration: 30 months

D3.3: 1st Prototype of the Context Observer

Lead contractor: TUM
Author(s): Emíttá Guzmán, Amel Mahmuzic, Joao Garcia, Walid Maalej.
Submission date: June 2011
Dissemination level: Public

Project co-funded by the European Commission under the Seventh Framework Programme
© S2, TUM, LERO, INESC-ID, TXT, PRODEVELOP
Abstract: This document summarizes the development work performed in task 3.5 (Development of Context Observer). It presents, by means of a tutorial, how the sensing component of the context system can be used to create sensors which provide context information to the other FastFix components. The source code can be accessed from the FastFix repository.¹

¹https://repository.fastfixproject.eu/svn/fastfix
Contents

1 Introduction .......................................................... 5

2 Architecture of Context Observer .................................. 6
   2.1 Context Observer in Context System ......................... 6
   2.2 Main Classes for Context Observation ....................... 7
   2.3 Context Event Representation ................................ 9
   2.4 Application Independent Sensing ............................ 10
      2.4.1 Separation of Concerns ................................. 10
      2.4.2 Sensor Implementation Strategies ..................... 10
   2.5 Context Observer and Privacy ................................ 11
   2.6 Context Observer and Efficiency ............................ 11
   2.7 Context Observer Metrics .................................... 12

3 How to Create New Sensors ........................................ 14
   3.1 Create New Class and Extend AbstractSensingStrategy .... 14
   3.2 Create and Register Sensor .................................. 17
   3.3 Include Sensed Properties in Ontology ........................ 17
   3.4 Communication Between a non-OSGi Application and Context Observer 18

4 Implemented Sensors ................................................. 19

5 Summary .................................................................. 23

Bibliography ................................................................ 23
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Main concepts of Context System</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>The sensing package of the context system</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Switching sensors on and off</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Context Events in the Ontology</td>
<td>18</td>
</tr>
</tbody>
</table>
1 Introduction

The context observer collects information concerning the application execution, the user interaction and the application environment. It is part of the context system described in D3.2 [2]. In this document we describe the main characteristics of the implemented context observer. Section 2 gives a general overview of the context system and its main components, highlighting the parts that are more important for context observation. This section also describes the main classes and most important methods of the context observer. In Section 3 we present a short tutorial describing how the context observer infrastructure can be used to create new sensors. Section 4 describes the main characteristics of the implemented sensors and Section 5 summarizes the main points presented in this document.
2 Architecture of Context Observer

The context observer is in charge of collecting context data that is later processed and fed to other FastFix components, such as the event correlator, self-healer and fault replicator. In D3.2 [2] the context observer functionality is encompassed in the context system. In this section we review the main concepts of the context system, which are related to context observation. Also, we describe the main classes in charge of context observation, included in the sensing component of the context system, we summarize how context information is represented, and we describe the implemented sensors. The source code that implements the characteristics described in this section is available in the FastFix repository 1.

2.1 Context Observer in Context System

The main concepts of the context system are Targets, ContextEvents, and the ContextBus. Figure 2.1 shows how these concepts are interrelated. A Target is an object which can be monitored in order to elicit context information. Example of targets are applications, a specific interaction between the application and the user, or an artifact. Targets consist of Properties. Property values represent the Target state. Sensors observe the changes of target Properties and are attached to the Targets. Therefore a Sensor is a ContextGenerator. Sensors implement how particular context is elicited. Context is represented by Events. When a Sensor detects changes in the target’s Properties it creates ContextEvents, and adds them to the ContextBus. The context system includes three different components: the context hub where the context events are collected and diffused, the sensing component which collects the context information, and the processing component where new information is generated. In this document we describe how the Sensing component (in charge of context observation) is implemented in FastFix; further information concerning the other components can be found in D3.2. The concepts most relevant for context observation are the ContextEvent, Target, Property, and Sensor. In the next section we describe how these concepts were translated into classes.

1https://repository.fastfixproject.eu/svn/fastfix
2.2 Main Classes for Context Observation

Figure 2.2 depicts the major classes of the sensing component, which we discuss in the following.
D3.3: 1st Prototype of the Context Observer

Target

A Target is something that we monitor in order to deliver context information (e.g. an application, or a user interaction). A Target possesses specific properties. The Property values represent the state of the Target. A Target can be attached to Sensors, which observe the value of its Properties.

Sensor

A Sensor is a type of ContextGenerator that is attached to a Target. For example, user interactions represent a particular Target. The Sensor generates ContextEvents every time the sensed Target changes its Properties. The main methods are:

- void reConfigure(SensingStrategy) initializes the Sensor with the SensingStrategy, the object that will do the sensing job.
- boolean isActive() returns the current state of the sensor, i.e. if the sensor is currently detecting values or not.
- boolean isDataSensed() returns true if the sensor has already sensed some values. Sensors can continuously sense value changes, based on their sensitivity (i.e. if there is a change that is higher than the sensitivity, a context event is created). Sensors can also make a sensing break, e.g. for performance reasons if another sensor is detecting semantically similar context information.
- void dataSensed() throws SensingException is called if the sensing strategy detects changes to the properties of the observed target.

SensingStrategy

The objects of this class implement how the context information is detected by a sensor. A SensingStrategy is a programmatic implementation of a Sensor. Sensor and SensingStrategies are separated because Sensors can adopt different strategies to observe the same type of ContextEvent. The SensingStrategy implements how value changes of the sensor’s target will be detected.

- void start() initializes the strategy, e.g. registers this object as a listener, creates communication channels, or threads.
- void stop() stops this strategy, e.g. unregistering it and closing open communication channels.
• **senseValues(String, String)** atomic sensing operation, where a name, value pair is created. The first parameter refers to the name of the property being sensed. The second parameter refers to the value of the property being sensed.

• **getPropertyNames()** delivers the names of the properties, whose values will be sensed by the sensor. These properties must be defined in an ontology.

• **getSensedData()** returns the data sensed by the sensor.

While the sensing component specifies the interfaces needed for the sensing strategies, the concrete strategy implementations will be included in separate sub-components that can be added and substituted at runtime.

### 2.3 Context Event Representation

Context events are the basis for context observation. In D3.2 we defined context events as a representation of context relative to time and representing “a change”. Each context event $e$ consists of the following information:

1. A time stamp $t_e$ specifying the absolute time when the context event occurred. This allows us to globally order a sequence of context events in time.

2. A duration $d_e$ specifying how long the context event is valid, that is holds a determinate value collected by the sensor.

3. A semantic context information $c_e$, describing the context event and its detected values. This information is represented as a set of RDF statements that instantiate a FastFix ontology.

The context observers use RDF in order to represent and store context events. This is because FastFix uses RDF ontologies as a data representation mechanism, and as a consequence other components expect data represented in an RDF format. The storage of the information in RDF format opens the possibility of reasoning mechanisms over the sensed data. An example of how this is done is the following:

```xml
<pre:e1 rdf:type art:MethodCall>
<pre:e1 int:hasTimeStamp 12222222200>
<pre:e1 int:hasDuration 200>
<pre:e1 int:concerns fastfix:fastfixed_app?name=appTxt>
<pre:fastfixed_app?name=appTxt rdf:type art:Application>
```

The example shows that the context event `pre:e1` has the type `art:MethodCall`. It gives the context event’s timestamps and its duration. It also represents the object
concerned with the event, `fastfix:fastfixed_app?name=appTxt` which has the type `art:Application`.

## 2.4 Application Independent Sensing

FastFix should be used to maintain arbitrary applications. The sensing component therefore needs to be able to collect context information independently from the target application. We realize this independence through the following techniques:

### 2.4.1 Separation of Concerns

We realize this independence through the use of a strategy pattern [1] in the sensing component. The strategy pattern in the sensing component decouples the `Sensor` from its implementation. `SensingStrategy` implementations for a determinate `Sensor` can therefore be developed for arbitrary applications. The pattern further allows for the addition and exchange of the `SensingStrategies` at runtime.

### 2.4.2 Sensor Implementation Strategies

How a target application may be observed depends strongly on the target application itself. Some applications provide the possibility to install plugins that may serve as information providers for the context observer. In general, we distinguish the following observation approaches:

- **Plugin based.** If the target application provides means to include external plugins, `SensingStrategies` may observe application specific information exposed by an additional FastFix plugin.

- **Instrumenting execution environment.** This observation approach uses facilities given by the runtime environment to observe the target application.

- **Instrumenting applications.** In this approach the developers of the target application instrument their application in order to collect information.

- **Bytecode instrumentation.** The instrumentation of bytecode allows to inject functionality into an already compiled application. Using this technique, functionality can be included in a target application to provide information to a `SensingStrategy`.

Whenever possible the instrumentation of the execution environment over the specific application will be preferred, in order to maintain the `SensorStrategies` as general as
possible, the use of bytecode injection and middleware sensors (e.g. log sensors) will also be preferred over application specific sensors. Application specific sensors will be implemented when the performance of the sensor changes drastically when applying methods to keep it general.

2.5 Context Observer and Privacy

In order to assure that end-users’ data is collected only when they agree, we allow users to switch the different sensors on and off. This allows for transparency and gives end-users control over what data is collected. Figure shows how the “Search” sensor can be turned off through a graphical user interface.

![Sensors interface]

Figure 2.3: Switching sensors on and off

2.6 Context Observer and Efficiency

In order to assure that the context observer is performant and does not affect the users experience with the end application, we implemented the following techniques:
Lazy loading  Lazy loading, also known as dynamic function loading, is a mode that allows specifying which components of a program should not be loaded into memory by default when a program starts. Normally, the system loader automatically loads the initial program and all of its dependent components at the same time. In lazy loading, dependents are only loaded as they are specifically requested. Lazy loading is available in OSGi and in particular in Eclipse. We use Lazy loading to improve the performance of the context observer. If a sensor, for example, is of no interest for a determinate application it will not be loaded into the memory.

Sensors’ Relationships  The Targets that are observed by the Sensors are semantically interrelated. These relationships are described in the FastFix ontologies. If two pieces of the context information should be pushed to the ContextBus and one is a refinement of the other, only the refinement is pushed. For example, if a sensor $S_1$ detects that the user is formatting a document $d$ and a sensor $S_2$ detects at the same time that the user is changing the color of some text located in document $d$ then only the context information of the sensor $S_2$ is fired, since changing text color is a refinement activity of formatting documents. In this manner, less context “noise” is created, which leads to a more efficient interpretation and communication of context.

Computation Intensive Operation at the Initialization  The context observer is designed in such way, that computation intensive transactions such searching for relationships in the ontology $O(n^2)$ are performed at the moment of starting the respective bundle. The most important one is the initialization of the targets based on the ontological description of their relationships. At runtime, targets become a template, which immediately format data values from the sensors into semantically rich statements. This is similar to the class loading mechanism and the instantiation of objects in object-oriented programming languages.

2.7 Context Observer Metrics

The Context Observer was implemented in Java using OSGi$^2$ and the Eclipse IDE$^3$. The code is available in the FastFix repository$^4$. Table 2.1 shows some metrics obtained from the code implementing the context observer. The metrics were obtained running the CodePro Analytiix 7.0$^5$ tool.

---

$^2$http://www.osgi.org/Main/HomePage
$^3$http://www.eclipse.org/
$^4$https://repository.fastfixproject.eu/svn/fastfix
$^5$http://code.google.com/javadevtools/codepro/doc/index.html
### Table 2.1: Context observer metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>12.674</td>
</tr>
<tr>
<td>Number of Characters</td>
<td>781,073</td>
</tr>
<tr>
<td>Number of Comments</td>
<td>3,633</td>
</tr>
<tr>
<td>Number of Constructors</td>
<td>167</td>
</tr>
<tr>
<td>Number of Fields</td>
<td>660</td>
</tr>
<tr>
<td>Number of Methods</td>
<td>1,326</td>
</tr>
<tr>
<td>Number of Types</td>
<td>267</td>
</tr>
<tr>
<td>Average Lines of Code Per Method</td>
<td>6.72</td>
</tr>
</tbody>
</table>
3 How to Create New Sensors

In this section we explain how sensors can be created using the existing context observer main classes by means of a tutorial. Throughout the tutorial we use the UseIDEPart sensor which is in charge of monitoring IDE interactions on the Eclipse environment as our illustrating example. The basic steps for creating a sensor are summarized in the following list:

1. Create an Eclipse plugin project
2. Create a new class and extend it from the AbstractSensingStrategy
   a) Implement the start(), stop() and getPropertyNames() methods of the AbstractSensingStrategy
   b) Each time the strategy needs to send the elicited information to the context bus, call the senseValue() and getSensor().dataSensed() methods belonging to the AbstractSensingStrategy
3. Create and register sensor
4. Include sensed properties in ontology

In the following sections we describe steps 2, 3, and 4 from this procedure. We assume that the execution of step 1 has been successful.

3.1 Create New Class and Extend AbstractSensingStrategy

A new class which extends the abstract class AbstractSensingStrategy needs to be created. The AbstractSensingStrategy is defined in the org.teamweaver.context.sensing package. Extend the following abstract methods of the AbstractSensingStrategy class:

- **public void start()**: Initializes the sensing strategy when the sensor is started, e.g. registers this object as a listener, creates communication channels, or threads.
- **public void stop()**: Stops the strategy strategy when sensor is turned off, e.g. unregistering it as listener, closing open communication channels.
• protected Set<String> getPropertyNames(): This method delivers the names of the properties, whose values will be sensed by the sensor. These properties must be defined in an ontology, as we will describe further in the tutorial.

• public void senseValue() and getSensor().dataSensed(): Every time the SensingStrategy needs to send data to the ContextBus (in order for it to be processed and sent to the other FastFix components) two methods need to be called: senseValue() and getSensor().dataSensed(). Exactly how this is done will become more clear once we have explained the UseIDEPart example.

UseIDEPart Example

In order to illustrate the substeps that need to be taken when creating the SensingStrategy we will use the UseIDEPart sensor as an example. The class related to this sensor is the WorkbenchPartSensing.java from the org.teamweaver.context.sensing.eclipse package. This class extends the abstract class AbstractSensingStrategy and implements two interfaces IPartListener2 (from org.eclipse.ui) and IWindowListener (from java.awt.event):

public class WorkbenchPartSensing extends AbstractSensingStrategy implements IPartListener2, IWindowListener

This class extends the abstract methods of the class AbstractSensingStrategy:

• public void start(): The WorkbenchPartSensing object registers in the PlatformUI.getWorkbench() of Eclipse as a Listener because it implements the IPartListener2 and IWindowListener interfaces, through these interfaces the sensor will receive notifications whenever an action occurs on the IDE.

public void start() {
    IWorkbenchWindow[] workbenchWindows = PlatformUI.getWorkbench().getWorkbenchWindows();
    for (int i = 0; i < workbenchWindows.length; i + +) {
        IWorkbenchWindow workbenchWindow = workbenchWindows[i];
        workbenchWindow.getPartService().addPartListener(this);
    }
    PlatformUI.getWorkbench().addWindowListener(this);
}

• public void stop(): The WorkbenchPartSensing object unregisters in the PlatformUI.getWorkbench() of Eclipse as a Listener
public void stop() {
    IWorkbenchWindow[] workbenchWindows =
    PlatformUI.getWorkbench().getWorkbenchWindows();
    for (int i = 0; i < workbenchWindows.length; i++) {
        IWorkbenchWindow workbenchWindow = workbenchWindows[i];
        workbenchWindow.getPartService().removePartListener(this);
    }
    PlatformUI.getWorkbench().removeWindowListener(this);
}

• protected Set<String> getPropertyNames(): The values of the attributes Type and concerns are returned every time this method is called

protected Set<String> getPropertyNames() {
    Set<String> retVal = new HashSet<String>();
    retVal.add("Type");
    retVal.add("concerns");
    return retVal;
}

• public void senseValue() and getSensor().dataSensed(): After a part of the IDE is activated, the Eclipse platform notifies the observer (in this example the WorkbenchPartSensing) and the method partActivated(...) is called. This method then calls the method sense(...) which is a private method in this extension of the AbstractSensingStrategy class. The method sense() calls senseValue(name, value) from the AbstractSensingStrategy in order to locate the sensed values in the buffer. After that, this.getSensor().dataSensed() must be called in order to notify the sensor that data are sensed. These data will then be saved in the RDF store. For example:

private void sense(String type, IWorkbenchPart part) {
    this.senseValue("Type", type);
    this.senseValue("concerns", ArtefactManager.getURI(part));
    this.getSensor().dataSensed();
}
3.2 Create and Register Sensor

After the sensing strategy has been created, the sensor must be created and registered. In the class EclipseSensorFactory a SensingFactory is instantiated (using a Singleton pattern [1]). This Factory is responsible for creating and registering the sensor with a sensing strategy. A sensor is created with the method `createSensor(...)`, which has the following parameters:

- Strategy of the sensor
- URI of the sensor in the ontology (see following section)
- Boolean value: set to true when the monitored event concerns an artifact and to false when it does not concern an artifact
- Boolean value: false

In the following example we show how a sensor can be created. Behind this sensor, there is WorkbenchPartSensing as a strategy.

```java
SensorFactory sManager = SensorFactory.getInstance();
sManager.createSensor(new WorkbenchPartSensing(), DataStore.INT_NS + "UseIDEPart", true, false);
```

3.3 Include Sensed Properties in Ontology

The ontology includes a description of the monitored context events and their properties. Once the sensor is created, the name of the context event it monitors and its properties need to be introduced in the ontology. The easiest way to do this is through the ontology tool Protegé. Figure 3.1 shows an example of the taxonomy of some of the context events for FastFix. The UseIDE context event, which has served as a running example through the tutorial is highlighted in the Figure.
3.4 Communication Between a non-OSGi Application and Context Observer

In order to make communication between a non-OSGi application and the context observer, which is OSGi based, an external monitoring component for the target application which sends the collected data, and an internal sensor which accepts the data from the external plugin needs to be created. The best way for making this communication is with sockets. This means that an internal sensor (in the context observer) will act as a server which accepts the data on a configurable port and an external sensor will act as a client which sends the data on the same port.
4 Implemented Sensors

Different partners of the consortium contributed to the creation of a set of sensors, involving user-interaction and application execution sensors. This serves as a proof of concept for the context observer conceptual model. The set includes general sensors, which will be useful independently from the error type and concrete target application. We also plan to implement other sensors on a need basis during the second reporting period of the project. In this section, we briefly list the implemented sensors and discuss how each of them is implemented.

Browser usage

This sensor detects browsing actions in Internet Explorer and FireFox. The sensor works for Internet Explorer 6 and 7 as well as for FireFox 2 and 3, for Windows, Mac and Linux. The sensors captures the URL that is accessed by the user (i.e. when the user sends a specific HTTP request). For Internet Explorer the sensors monitors the URL bar and the usage of the browsers buttons. For FireFox the sensors is realized as a plugin in the the XPI language. The sensor can be configured by using regular expressions to detects specific types of web interactions such as web search, or usage of a particular web-based applications. In this case other types of context events can be created.

Clipboard

Detects a cut, copy and paste command in an RCP editor, the source and destination documents as well as the number of pasted lines. The method postExecuteSuccess creates a ClipboardEvent if a cut, copy or paste command was executed in the Eclipse editor.

ClipboardSensing is a listener of the Eclipse ICommandService. This service provides services related to the command architecture within the Eclipse workbench and can be used to access the set of commands and command categories. This sensor currently works for RCP under Windows, Mac and Linux.
D3.3: 1st Prototype of the Context Observer

Commands

Detects user commands in an RCP workbench, that is all menu items of the applications as well as the actions provided in the main application window (e.g. save, open, print and close). Commands executed from the main menu, from buttons in the main workbench, as well as from short cuts are detected. **CommandSensing** is an **IExecutionListener** (a listener to the execution of commands which will be notified if a command is about to execute, and when that execution completes), which is registered to the command service of RCP.

Document change

This sensor detects document changes in an application without giving any information about the structure and content of the sensed document. The sensor observes the document in focus, characters entered by the keyboard and the inactivity of the user. For efficiency and privacy reasons, the single characters entered in the keyboard are not sensed. Instead, the sensors manages a counter using a boolean operator. This sensor includes a strategy for RCP applications and a strategy for non RCP applications on Windows. The later strategy monitors the size of particular file handles that are recognized as documents (no binaries).

Document read

Detects when the user is currently reading a document. This sensor first detects open files. If the file is in the focus, the sensor listens for keyboard and mouse events. The sensor creates a read event in case the user is not editing the file, and the user is active (i.e. scrolls, selects, unselects). The sensor also allows for entering N characters after T time, e.g. when the users press the keyboard by mistakes. This sensors is currently implemented for RCP, and is tested for Mac, Windows and Linux. However the implementation can easily be adjusted to other environments that offer a mouse and keyboard API.

File manipulation

The file manipulation sensor detects if a document is opened, closed, or changed. It currently involves two strategies. The first is a periodical file-sensing strategy, based on file handle utilities (UNIX’s ls/of and Windows file handles). The second is a dynamic OS-level file-sensing strategy, based on DTrace.
The periodical file-sensing strategy periodically checks file handles, which meets specific conditions (for example file extensions, size, name, or path). This strategy is able to report a file which was opened or closed in a given time-interval. The provided implementation senses for certain filetypes, such as pdf, txt and doc files. File-operations occurring between an interval can not be sensed with this approach.

The dynamic OS-level file-sensing strategy depends on an own scripting-language (called D), which can dynamically track nearly every event on the OS-level. The sensor allows for the detection of opening and closing events of files.

**Inactivity**

This sensor detects the user inactivity inside Eclipse. A user is inactive if no interactions with the keyboard, the mouse, and other UI input devices are sensed for a given time interval.

**Log events**

The log sensor checks for specific events in the log of an application. The events can be on any type and are given in the form of a regular expression when the sensor is being created. This sensor is useful for detecting errors, exceptions or specific messages being logged. This sensor can monitor any type of log and it processes information line by line. The sensor works for Windows and Unix environments.

**Method calls**

This sensor monitors all method calls which occurs during an application execution. In order to avoid performance issues it can be configured to only monitor the call of specific method. The sensor is implemented using bytecode injection and works in Windows and Mac operating systems.

**SAP**

To explore the possibilities of sensing rich client applications, we implemented a proof of concept for the SAP client. This sensor monitors the user client as well as the SAP server to detect particular interactions. The knowledge gained by implementing this sensor might be useful for detecting infrastructure communication errors (e.g. between the front-end and backend of a rich client application)
D3.3: 1st Prototype of the Context Observer

System information

This sensor detects metadata about the user environment, i.e. Operating System Name, Version, Java Version etc. This sensor works for Java applications in general. It continuously queries this meta information from the JVM.

User input

This sensor monitors the textual input a user gives into an application. It is implemented using bytecode injection. Currently this sensor supports Java applications and has been tested in Windows and Mac environments.

Workbench part

This sensor detects which part of an RCP Workbench is currently activated. Some windows include a complex structure with several parts. This sensor enables us to check if the user changes the focus without changing the windows (e.g. between a package explorer and editor parts). The sensor is implemented as a IWorkbenchPart listener in RCP, and works for Windows, Mac, and Linux. The sensor can be easily extended for any UI framework that offers window part APIs (e.g. Swing).

Workbench window

Detects the manipulation of RCP windows (open, close, minimize, maximize). This sensor differs from a general application running sensor since an RCP process can include several workbench windows. This is a proof of concept for any virtual machine that manages its windows independently from the Operating System.
5 Summary

In this document we summarized the development work of the context observer prototype. We described the architecture of the context observer and presented some metrics of its implementation; we also explained how to use the context observer to create new sensors, and described the implemented sensors. This prototype is a first iteration of the context observer and more work will be done in the following months to implement more sensors for the concrete scenarios described in D2.3 [3], as well as to improve the performance of the sensors, and to make the sensors available on more platforms and application versions.
Bibliography

