An Architectural Pattern for Enterprise Level Monitoring Tools

Olav Skjelkvåle Ligaarden*†, Mass Soldal Lund*, Atle Refsdal*, Fredrik Seehusen*, and Ketil Stølen*†
* *Department for Networked Systems and Services, SINTEF ICT, PO Box 124 Blindern, N-0314 Oslo, Norway*
E-mail: {olav.ligaarden, mass.s.lund, atle.refsdal,fredrik.seehusen, ketil.stolen}@sintef.no
† *Department of Informatics, University of Oslo, PO Box 1080 Blindern, N-0316 Oslo, Norway*

***Abstract*—Requirements from laws and regulations, as well as internal business objectives and policies, motivate enterprises to implement advanced monitoring tools. For example, a company may want a dynamic picture of its operational risk level, its performance, or the extent to which it achieves its business objectives. The widespread use of information and communication technology (ICT) supported business processes means there is great potential for enterprise level monitoring tools. In this paper we present an architectural pattern to serve as a basis for building such monitoring tools that collect relevant data from the ICT infrastructure, aggregate this data into useful information, and present this in a way that is understandable to users.**

***Keywords*-architectural pattern; enterprise level monitoring; software engineering**

I. Introduction

Requirements from laws and regulations, as well as internal business objectives and policies, motivate enterprises to implement advanced monitoring tools. With cloud computing it becomes even more important. An enterprise may take advantage of cloud computing by outsourcing all or parts of its information and communication technology (ICT) supported processes. In that case, monitoring becomes a necessity to ensure that the outsourcing party delivers the agreed performance. When designing an enterprise level monitoring tool based on an ICT infrastructure or a cloud system, we need to take into consideration the characteristics of the infrastructure/system. Events that can be observed directly at the ICT infrastructure level are typically of a low-level nature, such as service calls or responses. The significance of single events or types of events with respect to what we want to monitor and present at the enterprise level cannot usually be understood in isolation, and enterprise information has to be aggregated from a large number of different kinds of events and data. Extracting useful information from the available data in a manual fashion may be costly and unfeasible in practice. A main purpose of an enterprise level monitoring tool is therefore to close the gap between the low-level data obtained from the ICT infrastructure and the comprehension of human operators and decision makers. Ideally, such a monitoring tool will be able to collect all relevant input from the running ICT infrastructure, process this input so that it is turned into useful information, and present this information in a way that is understandable for those who need it. The desired level of detail depends on the role of the user. For example, a security engineer may want to know how often a port in a firewall is opened, while the company manager wants a high-level assessment of the information security risk to which the company is exposed.

Typically, a main objective for wanting to monitor something is to ensure that a certain value stays within an acceptable range. For example with respect to performance, we may want to monitor the average time delay between two steps of a business process, and to take suitable action if this delay is too long. This means that the monitor should automatically give a warning if the average delay is too long.

Irrespective of the particular kind of enterprise level monitoring tool that is being built and deployed, we consider the following capabilities to be core features of such a tool:

- Collect low-level data from the ICT infrastructure.
- Aggregate the collected low-level data.
- Evaluate the aggregated data.
- Present the aggregated data and the evaluation results to different types of enterprise users.
- Present the most recent aggregated data and evaluation results.
- Configure the tool with respect to:
  - The low-level data that should be collected.
  - How the low-level data should be aggregated into information that is relevant and useful.
  - How aggregated data should be evaluated.
  - The kind of aggregated data and evaluation results that should be presented and how this should be made comprehensible to different types of enterprise users.

In this paper we present an architectural pattern that should serve as a basis for building enterprise level monitoring tools with the above features. The pattern identifies the core components and shows how these components interact. We believe that this pattern is a good starting point for building specialized monitoring tools within various kinds of domains and enterprises.

The architectural pattern has been developed with the aim of fulfilling the following characteristics:

- It should serve as a basis for building monitoring tools within a wide range of domains and enterprises.
- It should facilitate modularity and reuse.

978-1-4577-0647-9/11/$26.00 ©2011 IEEE

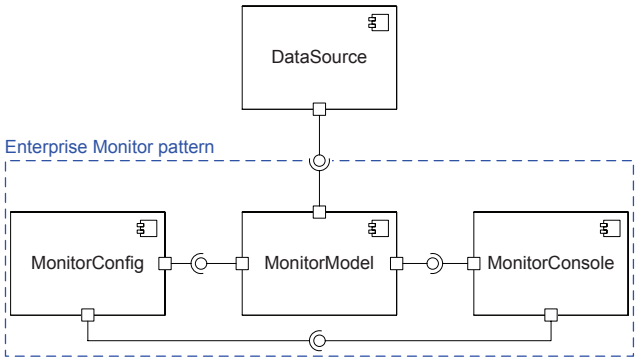

Figure 1. The components of the Enterprise Monitor pattern and its environment

- It should be scalable (handle growth in size or complexity).

The origin of the pattern is a risk monitor that we developed in the MASTER [1] research project. We believe that the risk monitor exhibits a number of features that are not specific to the monitoring of risks, but general to a broad class of enterprise level monitoring tools. We have therefore generalized the architecture of the risk monitor into an architectural pattern for this class of tools. In Section II we present this pattern. In Section III we demonstrate the pattern by showing the risk monitor as an instance, and we exemplify the use of the risk monitor in a health care scenario. After presenting related work in Section IV we conclude in Section V.

II. Architectural pattern

In the following we present our architectural pattern using part of the pattern template presented in [2]. According to this template, a pattern is described by the use of fourteen different categories. We do not use all categories since not all of them are relevant. The two categories *Context* and *Problem* have already been covered in Section I, while the *Example* category is covered in Section III.

A. Name and short summary

The ***Enterprise Monitor*** architectural pattern divides an enterprise level monitoring tool into three components: *MonitorModel* which contains the core monitoring functionality and data; *MonitorConsole* which presents core monitoring data in a specific way to a group of enterprise users; and *MonitorConfig* which is used by enterprise analysts to set up the enterprise level monitoring tool and to configure it during run-time.

B. Solution

In Fig. 1 is a UML [3] component diagram which shows the components of the Enterprise Monitor pattern as well as the component DataSource, which is part of the pattern's environment. The Enterprise Monitor collects low-level data from DataSource during run-time. DataSource may for instance be a sensor network, an ICT infrastructure, a cloud system, a database, and so on. The external behavior of the different components is defined by provided and required interfaces. A provided interface, represented by a full circle, describes services that a component provides to other components, while a required interface, represented by a half circle, describes services that a component requests from other components. The interfaces are again connected to ports on the component.

The component MonitorModel contains the core monitoring functionality and data. An enterprise analyst configures MonitorModel using the component MonitorConfig. This component is also used by the analyst to re-configure MonitorModel during run-time. The MonitorModel is independent of how core monitoring data is presented to different enterprise users.

An Enterprise Monitor may have several MonitorConsoles, but for simplicity, only one is shown in Fig. 1. Each MonitorConsole presents core monitoring data in a specific way to a group of enterprise users. The MonitorConfig configures how each MonitorConsole should present core monitoring data to its enterprise users. The MonitorModel must keep track of which MonitorConsoles that depend on the different core monitoring data. Each MonitorConsole therefore notifies the MonitorModel about what core monitoring data it depends on. When core monitoring data is updated, the MonitorModel notifies the MonitorConsoles. These MonitorConsoles then retrieve the data and present it to their enterprise users. This change-propagation mechanism is described in the *Publisher-Subscriber*[1] design pattern [2].

C. Structure

In this section we specify the structure of the pattern by the use of Class-Responsibility-Collaborator (CRC) cards [4] and a UML class diagram.

In Fig. 2 CRC-cards describing the three components are shown, while in Fig. 3 a UML class diagram describing how the components are related to each other is provided. Each CRC-card describes the responsibilities of one component and specifies its collaborating components. In the UML class diagram each component is represented by a class. The different classes are related to each other by associations, where each end of the association is given a multiplicity.

The responsibilities of MonitorConfig are to create a data config and a number of presentation models, and to configure the MonitorModel and MonitorConsoles. The relation between a data config and a presentation model is important for understanding the structure of the pattern. In Fig. 4 a UML

[1]Also referred to as Observer.

class diagram that shows the relation between a presentation model and a data config is provided. We distinguish between basic and composite indicators. By a basic indicator we mean a measure such as the number of times a specific event generated by the ICT infrastructure has been observed within a given time interval, the timing of an event, the load on the network at a particular point in time, or similar. The basic indicators are the low-level data. A composite indicator is an aggregation of basic indicators. The information carried by a composite indicator should be both relevant and useful for an enterprise user. The basic indicators and the aggregation functions are defined in the data config. A presentation model specifies how results from the monitoring should be presented to the enterprise user. Each presentation model is parameterized with respect to the composite indicators in the data config. The presentation models can therefore easily be updated when the composite indicators change. This parameterization is exemplified in Fig. 5. The figure shows excerpts from a data config, in the form of a file, and a presentation model. An enterprise user is presented the presentation model with the value of `f1` inserted. In this example we only use a textual presentation model, but other kinds of models, such as graphical presentation models, may be used as well. We can also consider letting different values trigger different presentations. For instance in the example in Fig. 5, the enterprise user may be presented with a warning if the value of `f1` exceeds some threshold.

The MonitorModel keeps a registry, referred to as `observerList` in Fig. 3, to keep track of the composite indicators that the different MonitorConsoles depend on. The core monitoring data `cmd` contains the basic and composite indicators used by MonitorModel, as well as other data used during the monitoring. The MonitorConsoles realize the interface Observer and therefore implements the `update` procedure. A MonitorConsole's `update` procedure is invoked by the `notify` procedure if one or more of the MonitorConsole's composite indicators have been updated. The `update` procedure will then retrieve the updated composite indicators from MonitorModel, and the MonitorConsole will update its display.

D. Dynamics

This section presents typical scenarios, modeled by the use of UML sequence diagrams, that describe the dynamic behavior of the pattern. Each entity in a sequence diagram is a component and is represented by a dashed, vertical line called a lifeline, where the box at its top specifies which entity the lifeline represents, its name as well as its type separated by a colon. Entities interact with each other through the transmission and reception of messages, which are shown as horizontal arrows from the transmitting lifeline to the receiving lifeline.

The sequence diagram in Fig. 6 describes the initial configuration of the MonitorModel and MonitorConsoles,

Class MonitorConfig	***Collaborators*** — MonitorModel — MonitorConsole
Responsibility — Creates data config and presentation models. — Configures and re-configures MonitorModel. — Configures and re-configures MonitorConsoles.	

Class MonitorModel	***Collaborators*** — MonitorConsole — MonitorConfig
Responsibility — Retrieves updated basic indicators from some data source. — Aggregates basic indicators into composite indicators. — Registers and unregisters dependent MonitorConsoles. — Notifies dependent MonitorConsoles about updated composite indicators.	

Class MonitorConsole	***Collaborators*** — MonitorModel — MonitorConfig
Responsibility — Evaluates composite indicators. — Presents composite indicators and evaluation results by the use of a presentation model to its enterprise users. — Retrieves updated composite indicators from MonitorModel. — Implements the update procedure.	

Figure 2. CRC-cards for the three components

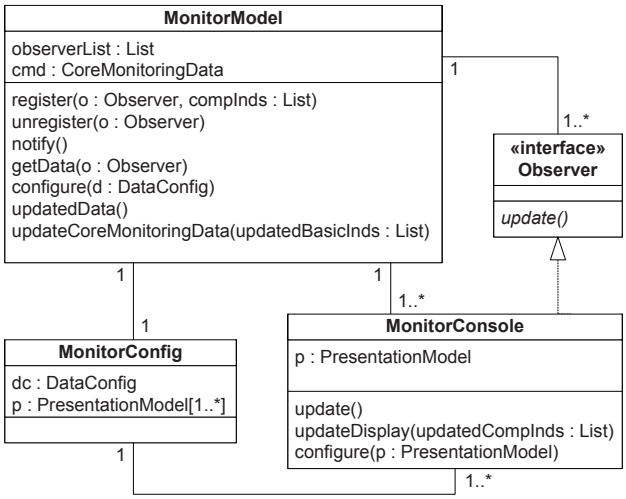

Figure 3. Components and their relations

while the sequence diagram in Fig. 7 describes how the displays of the MonitorConsoles are updated when basic indicators are updated. For simplicity, we only use one MonitorConsole.

In Fig. 6 the entity `config` configures the entity `model` by the use of a data config `d`. During configuration, `model` will set up a connection with some data source, specified by a sequence diagram `Data source setup` (not shown in this paper) referred to by the `ref` construct in the diagram.

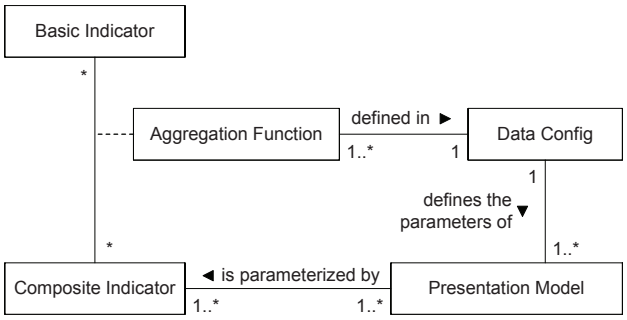

Figure 4. The relation between presentation model and data config

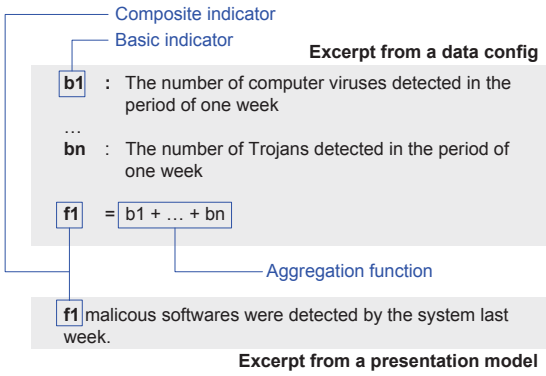

Figure 5. Parametrization exemplified

`config` also configures the entity `console` by the use of a presentation model `p`. After configuration, the `console` registers with the `model` in order to receive updated composite indicators. The parameter `compInds` specifies the composite indicators for which `console` wants to receive updates.

The interaction specified in Fig. 7 starts by the `model` receiving a message from the data source (not shown in the diagram) saying that there are updated basic indicators. This message is sent when one or more of the basic indicators have been updated. The `model` then retrieves these basic indicators (`updatedBasicInds`), and updates its core monitoring data by updating its basic and composite indicator values. The `model` then invokes the `notify` procedure, which invokes the `update` procedure of `console` (if the presentation model of `console` is parameterized by one or more of the updated composite indicators), which again retrieves updated composite indicators (`updatedCompInds`) from the `model` and updates its presentation model based on this data. During the execution of `updateDisplay`, the `console` may evaluate one or more of the updated composite indicators and use the evaluation results when updating the display.

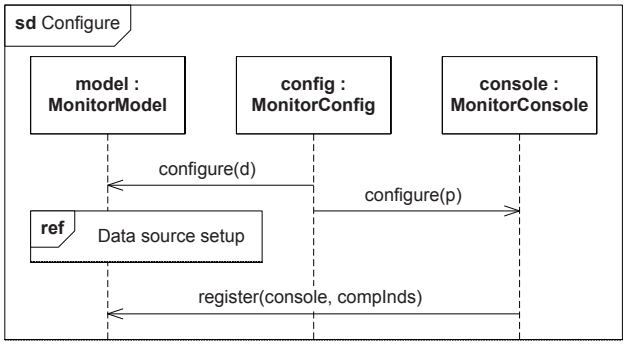

Figure 6. Sequence diagram "Configure"

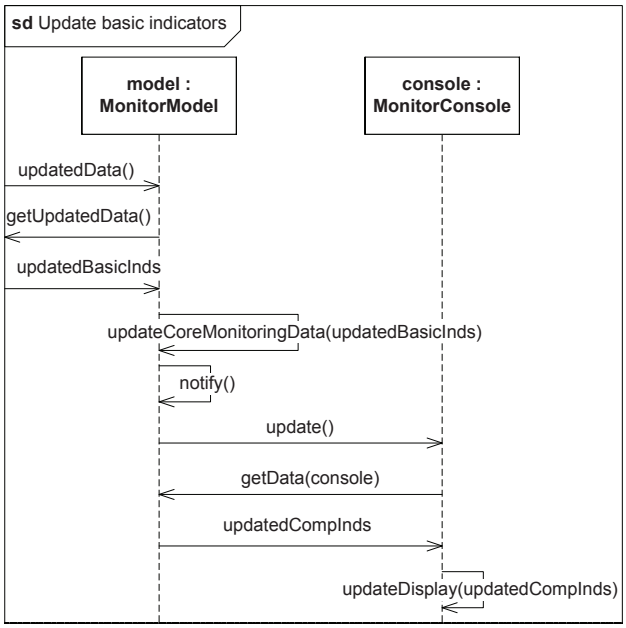

Figure 7. Sequence diagram "Update basic indicators"

E. Implementation

In this section we provide guidelines for implementing the pattern in the form of code fragments written in Java. The code fragments serve only as suggestions, but they high-light some important aspects that should be considered when implementing the pattern. In the code fragments we use *bi* and *ci* to refer to basic and composite indicators, respectively. In Fig. 8 some of the code that may be used to implement MonitorModel is provided. The Maps `biVals` and `ciVals` use basic and composite indicator names, respectively, as keys, while the values assigned to the indicators during monitoring are used as values. The indicator names should be specified in the data config used to configure the MonitorModel. The Map `ciDefs` uses composite indicator names as keys, while the values are the definitions of the composite indicators. This Map

is used when aggregating basic indicators. The last Map `observers` uses Observer objects as keys, where each object refers to a MonitorConsole, while the values are Maps. Each of these Maps uses composite indicator names as keys and Booleans as values. The value is `true` if the MonitorConsole has not retrieved the latest updated value of the composite indicator, and `false` otherwise. The Map `observers` keeps track of which MonitorConsoles have registered with MonitorModel, and makes sure that MonitorConsoles *only* retrieve composite indicators that have been updated.

The function `updateCoreMonitoringData` is executed after the MonitorModel has retrieved updated basic indicators from some data source. It takes a Map of basic indicator names and their updated values as input. The function updates the values of the basic indicators in the Map `biVals`. Then it checks for each composite indicator if it needs to be updated. This is done by checking whether the definition of the composite indicator refers to any of the basic indicators in `biNames`. If this is the case, the value of the composite indicator in `ciVals` is updated. The function `aggregate` produces this value based on the definition `d` and the basic indicator values stored in `biVals`. At the end, the function `notify` is called with the names of the updated composite indicators as parameter.

For each Observer object in the Map `observers` the function `notify` sets the value in `obsMap` to `true` if the key is a member of the set `updatedCis`. If `obsMap` contains one or more `true` values and if `obsMap` did not contain any `true` values before `notify` started updating its values, then `update` for the Observer `o` is called. `update` will not be called if `updateCalled` equals `true` since this means that `update` has already been called and that Observer `o` has not retrieved the updated composite indicators yet.

The definition of `update` is shown in Fig. 9. MonitorConsole implements the `update` function since it implements the Observer interface. The return value of calling the `getData` function, shown in Fig. 8, is a Map of composite indicators names and their updated values. The Map `updatedCis` is used to update the display of MonitorConsole. We can see from the definition of `getData`, in Fig. 8, that only updated composite indicators are retrieved. All values in the Map `obsMap` are `false` when `getData` returns, since Observer `o` has retrieved all updated composite indicators.

F. Consequences

In this section we present the consequences in the form of benefits and liabilities of applying the pattern.

The application of the Enterprise Monitor pattern has the following **benefits**:

- *Separation of MonitorModel and MonitorConsoles*. The pattern separates user interfaces from the core

```
class MonitorModel {
  private Map<String,Object> biVals = null;
  private Map<String,Object> ciVals = null;
  private Map<String,Definition> ciDefs = null;
  private Map<Observer,Map<String,Boolean>>
    observers = null;

  private void updateCoreMonitoringData(
    Map<String,Object> updatedBis){

    Set<String> updatedCis;
    //Construct updatedCis
    Set<String> biNames = updatedBis.keySet();

    for(String name : biNames){
      biVals.put(name, updatedBis.get(name));
    }

    for(String name : ciDefs.keySet()){
      Definition d = ciDefs.get(name);

      if(ciNeedsUpdate(d,biNames)){
        ciVals.put(name, aggregate(d));
        updatedCis.add(name);
      }
    }
    notify(updatedCis);
  }

  private void notify(Set<String> updatedCis){
    boolean updateCalled;

    for(Observer o : observers.keySet()){
      Map<String,Boolean> obsMap = observers.get(o);
      updateCalled = obsMap.containsValue(true);

      for(String name : updatedCis){
        if(obsMap.get(name) != null){
          obsMap.put(name,true);
        }
      }

      if(!updateCalled && obsMap.containsValue(true)){
        o.update();
      }
    }
  }

  public Map<String,Object> getData(Observer o){
    Map<String,Object> updatedCis;
    //Construct updatedCis
    Map<String,Boolean> obsMap = observers.get(o);

    for(String name : obsMap.keySet()){
      if(obsMap.get(name)){
        updatedCis.put(name,ciVals.get(name));
        obsMap.put(name,false);
      }
    }

    observers.put(o,obsMap);
    return updatedCis;
  }
}
```

Figure 8. Some Java code for class MonitorModel

```
interface Observer {
  abstract void update();
}

class MonitorConsole implements Observer {
  private MonitorModel m;
  private PresentationModel p;

  public void update(){
    Map<String,Object> updatedCis = m.getData(this);
    updateDisplay(updatedCis);
  }

  private void updateDisplay(
    Map<String,Object> updatedCis){

    //Update the presentation model "p" based
    //on "updatedCis".
    //Display the updated model.
  }
}
```

Figure 9. Some Java code for interface Observer and class MonitorConsole

monitoring functionality and data.

- *Synchronized MonitorConsoles*. The change-propagation mechanism ensures that all MonitorConsoles are notified about updates of their data at the correct time.
- *Efficient updates*. MonitorConsoles are only notified when their data have been updated. The MonitorConsoles retrieve only the data that have been updated and nothing more.
- *Re-configuration*. By using a data config and presentation models it is possible to re-configure the enterprise level monitoring tool during run-time.
- *Creation of MonitorConsoles during run-time*. It is possible to create new MonitorConsoles and register them with the MonitorModel during run-time.

The **liabilities** of the pattern are as follows:

- *Scalability*. Using the pattern to build a very large enterprise level monitoring tool may lead to high complexity and a large number of MonitorConsoles. It may then be better to build separate tools for the different monitoring tasks.
- *Close coupling of MonitorModel to MonitorConfig*. The MonitorModel depends on the specific language in which the data config is expressed. Change of language may most likely require changing the code of the MonitorModel.
- *Re-configuration*. If the MonitorModel is re-configured it may also be necessary to re-configure a number of the MonitorConsoles. Losing track of which presentation models must be changed when the data config used by the MonitorModel is changed is a potential consequence.

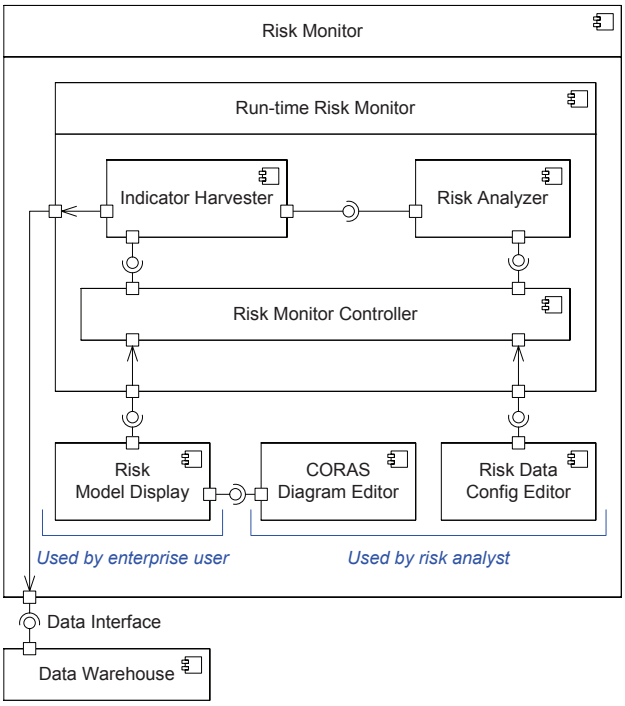

Figure 10. Component diagram of the risk monitor

G. Related patterns

The *Model-View-Controller* [2] (MVC) architectural pattern solves a related problem. This pattern was used as inspiration when we designed our pattern. MVC divides an interactive application into a model (core functionality and data) and different user interfaces, where the user interfaces presents the same core data in different ways. If the core data is updated in the model or in a user interface, then this change should be reflected in all user interfaces. As for our pattern, MVC uses the *Publisher-Subscriber* design pattern [2] to implement the change-propagation mechanism. This mechanism ensures consistency between the model and user interfaces.

III. Demonstration of Architectural Pattern

The risk monitor is an implementation of the architectural pattern presented in this paper. It is a monitor specialized at aggregating basic indicators related to risk and presenting the results in risk models. In particular, the language used for the presentation models is the CORAS risk modeling language. For more information on the language and the use of indicators in risk analysis, we refer to [5] and [6], respectively. The structure of the risk monitor is shown in Fig. 10 in the form of a UML component diagram.

The Run-time Risk Monitor contains the components that play a part at run-time, while the three components Risk Model Display, CORAS Diagram Editor, and Risk Data Config Editor provide a graphical user interface to the user;

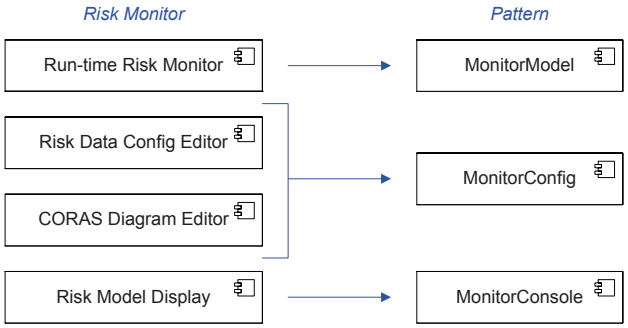

Figure 11. Correspondence between risk monitor components and pattern components

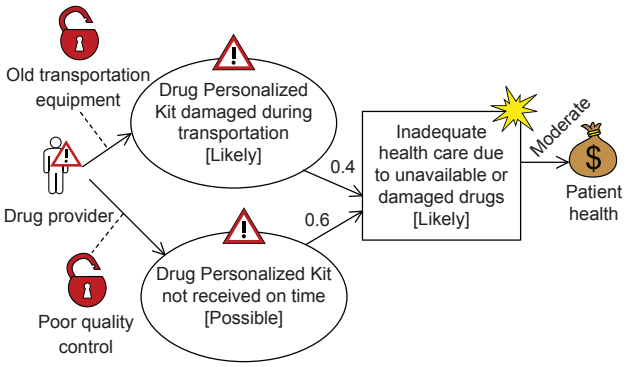

Figure 12. Example CORAS threat diagram

they are either used to configure the monitor or view the output of the monitor. Fig. 11 shows the correspondence between the components of the risk monitor and the components described in the pattern.

In the Risk Monitor the Indicator Harvester retrieves updated basic indicators from a data repository, referred to as Data Warehouse in the diagram. The Risk Analyzer aggregates the basic indicators into composite indicators. The Risk Monitor Controller configures the Run-time Risk Monitor based on the data config provided by the Risk Data Config Editor, while the CORAS Diagram Editor configures the Risk Model Display by the use of a risk model.

The Risk Monitor implements its components as plug-ins to Eclipse. As a result, the Risk Monitor appears to the risk analyst as a standalone Eclipse tool with the three front-end components: CORAS Diagram Editor, Risk Data Config Editor, and Risk Model Display, as well as basic functionality to communicate with the Risk Monitor Controller, available as a integrated whole. The Risk Model Display is primarily used by the enterprise user during run-time, but can also be used by the risk analyst for testing when configuring the Risk Monitor.

In the following we concentrate on the configuration of the Risk Monitor, but first we give a little introduction to risk analysis by the use of CORAS.

A prerequisite for doing risk monitoring is to do a risk analysis. The outcome of a risk analysis is usually a description of threats to the target of analysis, vulnerabilities that the threats might exploit, unwanted incidents initiated by the threats, risks (which are usually defined as the likelihood of an unwanted incident and its consequence for a specified asset), and treatment of the risks.

In the CORAS methodology for model-based risk analysis [5], threats, threat scenarios (series of events initiated by threats, leading to unwanted incidents), vulnerabilities, unwanted incidents, and risks are modeled using a number of diagrams, referred to as CORAS diagrams. In Fig. 12, a simple CORAS threat diagram is shown. The example is adapted from one of the example cases used in the MASTER [1] project and concerns a hospital that has outsourced its drug stock management. A Drug Personalized Kit contains the drugs needed in the treatment of a particular patient. A drug provider is responsible for delivering Drug Personalized Kits to the hospital within a given deadline after receiving an order from the hospital. The diagram documents a part of a risk analysis where the focus is on protecting the asset "Patient health". The unwanted incident "Inadequate health care due to unavailable or damaged drugs" may harm this asset. Two different threat scenarios which may lead to this unwanted incident have been identified, each represented by an oval with a warning sign. Both threat scenarios are initiated by the unintentional threat "Drug provider". The first scenario is initiated via the vulnerability "Old transportation equipment", while the second is initiated via the vulnerability "Poor quality control". Moreover, the threat scenarios and unwanted incident, as well as the relations between them, are annotated with quantitative or qualitative likelihood (i.e. frequency or probability) values, and the relation between the unwanted incident and the asset is annotated with a consequence value. Fig. 12 represents a snapshot view at the point in time when the risk analysis was conducted, and the assigned likelihood and consequence values are assumed to apply at that point. The goal of the Risk Monitor, however, is to obtain risk pictures that are updated as the situation changes. In the Risk Monitor we achieve this by using values that are derived from basic indicators that represent observable and measurable properties of the target. These basic indicators are aggregated and presented as part of the risk picture.

To configure the Risk Monitor, the risk analyst needs to create a data config, used by the Run-time Risk Monitor, and a presentation model, used by the Risk Model Display. The risk analyst creates the data config by the use of the Risk Data Config Editor. In the data config, the risk analysts define:

- the basic indicators that should be monitored; and
- variables (composite indicators), aggregated from basic

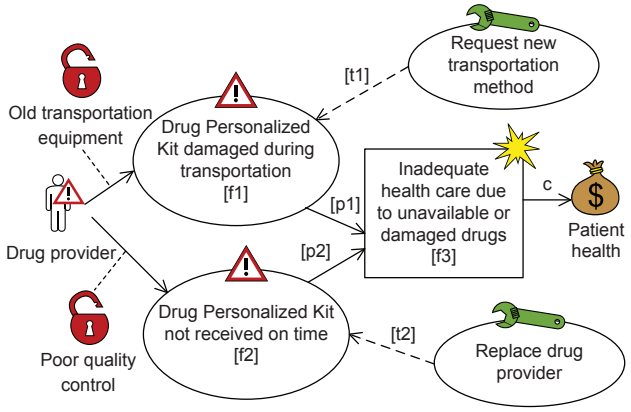

Figure 13. Example presentation model in the form of a CORAS threat diagram, annotated with variables

indicators, for likelihood values, consequence values, trigger conditions, and risk levels.

The presentation model used is a risk model annotated with variables from the data config. The risk analyst creates the presentation model by the use of the CORAS Diagram Editor. In Fig. 13 an example presentation model is provided, while a small excerpt of an example data config, which provides definitions for how to assign values to some of the variables used in the presentation model, is shown in Fig. 14.

The diagram in Fig. 13 is similar to the diagram in Fig. 12, but it differs in two ways. First, the likelihood and consequence values have been replaced by variables whose values will be dynamically updated during run-time. For variable names we follow a convention where `b` is a basic indicator, `f` is a frequency, `p` is a probability, `t` is a triggering condition, `c` is a consequence, and `r` is a risk level.

Second, treatments have been attached to the two threat scenarios. Each of these will be shown to the enterprise user only if the Boolean variables `t1` and `t2` have been assigned the value `true`.

From Fig. 14 we can see that the variables `f1`, `f2`, `f3`, `p1`, and `p2` are the likelihood variables that occur in Fig. 13, while `b1`, `b2`, and `b3` are the basic indicators whose values will be obtained via the Indicator Harvester. `t1` and `t2` are the Boolean variables that also occur in Fig. 13. The variable `r`, that is used in the definition of `t1` and `t2`, will be 0 if the risk level associated with the unwanted incident is acceptable and 1 otherwise. Note that the definitions of `r`, `p1`, `p2`, and `c` have been omitted from the excerpt to save space.

The variables in Fig. 13 will be substituted for actual values retrieved from the Run-time Risk Monitor during run-time, while treatments will appear in the diagram if the retrieved Boolean variables have the value `true`. We do not illustrate this further, as these diagrams will be similar

```
...
indicator b1 = "The number of DPKs rejected on
                arrival due to damage"
indicator b2 = "The number of DPKs that arrive
                between 1 and 6 hours after deadline"
indicator b3 = "The number of DPKs that arrive
                more than 6 hours after deadline"
variable  f1 = b1;
variable  f2 = (b2 / 2) + b3;
variable  f3 = (f1 * p1) + (f2 * p2);
variable  t1 = r == 1 and f1 >= 15;
variable  t2 = r == 1 and f2 >= 20;
...
```

Figure 14. Excerpt from the data config used by the Run-time Risk Monitor

to Fig. 12, with the possible addition of treatments.

IV. Related work

To the best of our knowledge, there exist no other architectural patterns that address the problem of building enterprise level monitoring tools with the core features described in Section I. As already mentioned, the MVC [2] architectural pattern solves a related problem. Our pattern and MVC are related, since MVC was used as inspiration when we designed our pattern. One of the main differences between the two patterns is that in MVC all the user interfaces display the same core data, while in our pattern the MonitorConsoles can display different core monitoring data, which means that they need to be updated differently.

Design patterns specifically targeted on building software health monitoring applications are described in [7]. It focus on design patterns for sensors collecting information about the internal state and operation of software and how this information can be combined into software health indicators describing different aspects of software health. Design patterns [8] describe relationships between classes and objects. Our pattern is at the architectural level, since it describes relationships between components. Examples of architectural patterns can be found in [2]. Another approach for creating monitoring applications is described in [9], which presents a tool framework called Mozart that uses a model driven approach to create performance monitoring applications using indicators.

Within business intelligence (BI) there are many tools that do some form of enterprise level monitoring. In the following we mention some of the most central ones. One type of tools is digital dashboards [10]. A dashboard should present monitored data in a highly visual and intuitive way so that managers can monitor their progress towards their identified goals. Other types of tools that do a sort of monitoring are data [11] and process [12] mining tools. Within business intelligence, data mining uses techniques from statistics and artificial intelligence to identify interesting patterns in often large sets of business data. Process mining is used to extract information about business process by the use of event logs. Both data and process mining use historical data, while

in our approach we deal with real-time data as well as historical data. Another type of tools that rely on monitoring is business performance management [13] tools. These tools are used to monitor, control, and manage the implementation of business strategies.

A monitoring tool closely related to our pattern is the MASTER ESB [14]. This tool is used to monitor compliance with access and usage policies in a system. The tool monitors low-level evidence data that is aggregated into meaningful evidence on how different parties comply with the policies. This evidence is then evaluated, and actions against compliance violations may be taken.

A central aspect of our pattern is the aggregation of data into more meaningful information. Fault trees [15], Markov models [16], and Bayesian networks [17] all specify some form of aggregation, and have support for updating the output values when the input values changes. In [18], fault trees are used with influence diagrams [19], a graphical language originally designed for supporting decision making based on the factors influencing the decisions. These diagrams are connected to leaf nodes in fault trees. The influencing factors contribute to the probabilities of the leaf nodes, and these probabilities are propagated to the unwanted incident specified at the root of the tree. In this case, the fault tree specifies how the low-level data at the leaf nodes should be aggregated into something more meaningful. It is possible to monitor these influencing factors as well as the input values for the other approaches that aggregate data.

The pattern presented in this paper uses basic and composite indicators. Indicators are also used by other monitoring approaches, such as in [20] where low-level indicators are aggregated to create high-level indicators. These high-level indicators are used to monitor business performance. The low-level and high-level indicators are equivalent to our two types of indicators.

V. Conclusion

In this paper we have presented an architectural pattern called *Enterprise Monitor* that may serve as a basis for building enterprise level monitoring tools. The pattern identifies the core components and shows how they interact. The applicability of the pattern is demonstrated by an instantiation in the form of a risk monitor.

We believe the architectural pattern captures the core features of enterprise level monitoring tools. The MonitorModel collects relevant low-level data in the form of basic indicators from the ICT infrastructure and aggregates the basic indicators into composite indicators. The MonitorConsoles retrieve the most recent updated composite indicators from MonitorModel, evaluate them if needed, and update the displays used by their enterprise users based on the composite indicators and evaluation results. The enterprise analyst can quite easily configure the two above-mentioned components by using a data config and presentation models. The analyst also has the option of re-configuring the components during run-time. Since the architectural pattern captures all the core features and since these features are general, the pattern can be used as a starting point for building specialized monitoring tools within various kinds of domains and enterprises.

The pattern supports modularity by separating the MonitorConsoles from the MonitorModel, making it quite easy to introduce new MonitorConsoles. There are however some dependencies between the MonitorModel and the MonitorConfig. The MonitorModel depends on the specific language that the data config is written in. If this language is changed, then most likely the code of the MonitorModel must be changed as well.

The design pattern also promotes reuse of components. It is possible to make very general implementations of some of the components since configurations are used to specialize the components. For instance, it is possible to make a very general MonitorModel, which becomes specialized when configured.

Today, enterprises are starting to take advantage of the possibilities provided by cloud computing. This will result in increased need and demand for enterprise level monitoring and new challenges with respect to the capabilities of enterprise level monitoring tools. In future work we will identify and address these challenges.

Acknowledgments

The research on which this paper reports has been carried out within the DIGIT project (180052/S10), funded by the Research Council of Norway, and the MASTER and NESSoS projects, both funded from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreements FP7-216917 and FP7-256980, respectively.

References

[1] "MASTER: Managing Assurance, Security and Trust for sERvices," http://www.master-fp7.eu/, Accessed: 2011-07-11 14:00PM CEST.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, 1st ed. Wiley, 1996.

[3] Object Management Group (OMG), "Unified Modeling Language Specification, version 2.0," 2004.

[4] K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," in *Proceeding of Object-oriented programming systems, languages and applications (OOPSLA '89)*. ACM, 1989, pp. 1–6.

[5] M. S. Lund, B. Solhaug, and K. Stølen, *Model-Driven Risk Analysis: The CORAS Approach*, 1st ed. Springer, 2010.

[6] A. Refsdal and K. Stølen, "Employing Key Indicators to Provide a Dynamic Risk Picture with a Notion of Confidence," in *Proceedings of Third IFIP WG 11.11 International Conference (IFIPTM'09)*. Springer, 2009, pp. 215–233.

[7] A. Lau and R. Seviora, "Design Patterns for Software Health Monitoring," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*. IEEE, 2005, pp. 467–476.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.

[9] M. Abe, J. Jeng, and Y. Li, "A Tool Framework for KPI Application Development," in *Proceedings of the IEEE International Conference on e-Business Engineering (ICEBE'07)*. IEEE, 2007, pp. 22–29.

[10] C. Dover, "How Dashboards Can Change Your Culture," *Strategic Finance*, vol. 86, no. 4, pp. 43–48, 2004.

[11] Y. Fu, "Data Mining," *IEEE Potentials*, vol. 16, no. 4, pp. 18–20, 1997.

[12] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters, "Workflow Mining: A Survey of Issues and Approaches," *Data Knowledge Engineering*, vol. 47, no. 2, pp. 237–267, 2003.

[13] M. N. Frolick and T. Ariyachandra, "Business Performance Management: One Truth," *Information Systems Management*, vol. 23, no. 1, pp. 41–48, 2006.

[14] B. Crispo, G. Gheorghe, V. D. Giacomo, and D. Presenza, "MASTER as a Security Management Tool for Policy Compliance," in *Towards a Service-Based Internet*, ser. Lecture Notes in Computer Science, E. D. Nitto and R. Yahyapour, Eds. Springer Berlin / Heidelberg, 2010, vol. 6481, pp. 213–214.

[15] International Electrotechnical Commission, "Fault Tree Analysis (FTA)," 1990.

[16] International Electrotechnical Commission, "Application of Markov Techniques," 1995.

[17] D. Lidley, *Introduction to Probability and Statistics from a Bayesian Viewpoint*. Cambridge University Press, 1965.

[18] J. Spouge and E. Perrin, "Methodology Report for the 2005/2012 Integrated Risk Picture for Air Traffic Management in Europe," EUROCONTROL, Tech. Rep., 2006, eEC Technical/Scientific Report No. 2006-041.

[19] R. A. Howard and J. E. Matheson, "Influence Diagrams," *Decision Analysis*, vol. 2, no. 3, pp. 127–143, 2005.

[20] P. Huang, H. Lei, and L. Lim, "Real Time Business Performance Monitoring and Analysis Using Metric Network," in *Proceedings of IEEE International Conference on e-Business Engineering (ICEBE'06)*. IEEE, 2006, pp. 442–449.