

# A prototype to clarify the behavior of Windows services by using dynamic probes in an autonomic architecture

Jess Nielsen<sup>1</sup>, Sufyan Almajali<sup>2</sup>

<sup>1</sup>Trapeze Group Europe A/S, Søren Frichs Vej 38K, DK-8230 Åbyhøj, Denmark

<sup>2</sup>Computer Science Department, Princess Sumaya University, Amman Jordan

This article documents and evaluates some techniques that might be used to build a consolidated log collected from certain Windows services at runtime. The purpose of this is to eliminate the time that is needed today by the systems management staff to locate the root cause of failure. Several issues make it difficult to locate the root cause of failure. Due to very heterogeneous systems, their logs and locations are very different and non-standardized. This makes it difficult to compare the logs and it often requires special knowledge to interpret their contents. Dynamic probes can be used to collect runtime data from running systems to determine their runtime behaviors and states. Information about runtime behaviors and states of the systems is useful as a foundation for a common and consolidated log with a standardized content and such a log will make it easier to compare and analyze the data from the failed systems. It will also make it possible to augment the log with metadata that might describe the recovery actions needed to reestablish the systems from their bad states. In this way the systems management staff will need less time for analyzing and comparing error logs, and knowledge is only required in order to understand the consolidated log.

**Index Terms**—About Autonomic Computing, Dynamic Probes, Runtime behaviors, Windows services

## I. INTRODUCTION

THIS DOCUMENT outlines an approach that can be used to clarify the behavior of Windows services based on potential problems when localizing log information in an IT infrastructure.

Today's systems management is characterized by many systems that are monitored and managed individually, which is based on own observations and experiences from previous jobs inside systems management.

The systems used in an IT infrastructure today are often based on a Microsoft client/server installation where the server-side software often is a Windows Service (i.e. FTP -, SMTP and Web server daemons) because it keeps running even when no user or administrator is logged in at the central console although Linux, Mac and UNIX daemons works similar to Windows services.

Each of these systems owns their own log (or even logs) which typically have their own syntaxes and/or semantics. This requires the IT staff to achieve special knowledge within each system and it makes it difficult to compare the (correct) logs and analyze them.

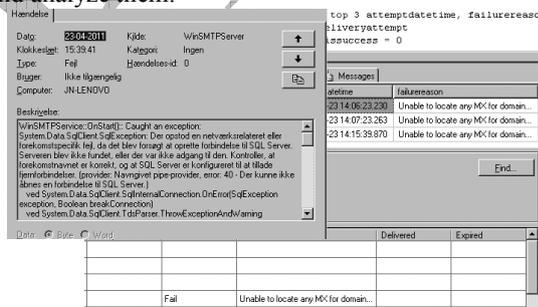


Fig. 1: Various heterogeneous live logs

The logs vary from the Windows Event Viewer, different kind of log files or even built-in logs stored into its respective application database. Usually, they often include neither any description of the error codes other than some few short messages nor any concrete actions needed to recover the respective systems from their bad state.

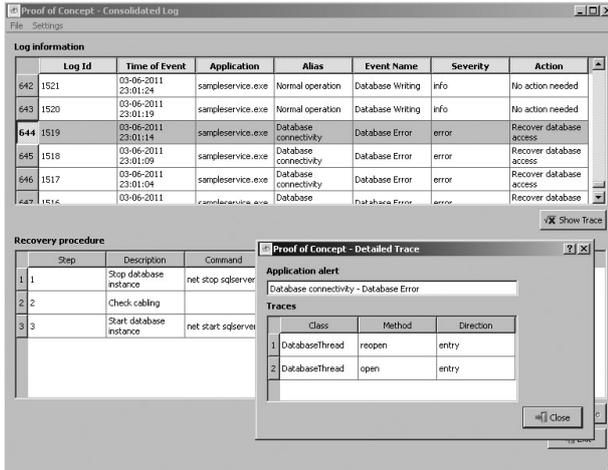
Several techniques exist to solve this. It can be solved by exception handling, data mining or by consolidating the log data, but consolidation is considered the most appropriate because it does not require specific knowledge of the proprietary logs.

Consolidation (or data consolidation) is the procedure through which you install a central data storage unit to keep all databases easily accessible. Consolidation is not unlike data center migrations but the focus is on having the files or data in a specified location rather than actually moving them around. With respect to this informal description and in my opinion, a consolidated log can be defined as outlined in the definition below.

### Definition

*A consolidated log is a central log which merges data from several sources (logs) in a homogeneous and standardized format.*

The introduction of a consolidated log has the goal to save valuable time. The reason for this is that it prevents the search of logs in various locations and analysis of wrong logs by conceptually placing all log information at one place. The knowledge of different systems and particularly their proprietary error codes is no longer needed when all information is in a consolidated log expressed by a common syntax and semantic.



**Fig. 2: A consolidated log**

Secondly, it might become easier to generate a generic rule set based on one common log instead of several logs and the augmentation and mapping of recovery actions also becomes easier when all errors are expressed by the same semantics.

The primary motivation is therefore to make it easier for the systems management staff to manage the systems by introducing a tool or a proof of concept which demonstrates some techniques that can consolidate and standardize the existing log information where possible.

## II. AN AUTONOMIC APPROACH

Consolidated logs are placed on the second level, which is also called the managed level on the autonomic axis [3, p. 6]. Autonomic computing does not directly address this subject (even this approach), but consolidated logs are placed on the managed level on the autonomic axis which makes it related. In other words, consolidated logging can be used in conjunction with or as a foundation for autonomic systems, and that is also the reason why autonomic systems are introduced and used in this article.

A summary of the autonomic systems is outlined to achieve a better understanding of autonomic systems and their architectures. The five levels of the autonomic axis are described with relation to the regular systems. This is continued by a three-layered reference model that is used when summarizing some of the existing approaches. During the walkthrough of the existing approaches, they will be related to the autonomic world and classified in relation to the three-layered reference model.

Within the system management industry, autonomic systems are becoming more and more widespread. Different vendors invent their own system which makes it easier to monitor the individual systems, but even though many vendors have made their own autonomic features to ease the administrator's job and many autonomic components have been built, they do not yet exist in a large-scale and fully autonomic computing system [2, p. 18].

*What exactly are self-managed systems? The vision is of systems which are capable of self-configuration, self-adaptation and self-healing, self-monitoring and self-tuning, and so on, often under the flag of self-\* or autonomic systems [7, p. 1].*

In general, autonomic computing systems have several important self-management properties that characterize all fully autonomic computing systems: self-configuration, self-healing, self-optimization and self-protection.

To categorize systems in the "autonomic world" IBM has spearheaded the autonomic axis that categorizes the systems in five levels. Each level represents a more advanced and refined system. Systems that are categorized at the fourth and fifth level must have all of the self-\* properties to achieve their respective goals as decision making and being business policy driven.

From an architectural perspective these five levels can be mapped into an architectural approach. The lowest level is a "normal" system with no explicit requirements related to self-managed properties. The second level is an operational level that takes care of component control and collection of data. The third and fourth levels are the tactical levels that interpret and execute the plans. The fifth level generates the plans due to time consuming computations to achieve some high level goals.

To achieve the properties of the upper levels it is necessary to achieve the requirements for the lower levels first. The lower level requires data consolidation that implicitly includes a generic collection and organization of data. This initiates the requirements for an architecture which meets these specific requirements and that is why it becomes interesting to investigate some of the existing autonomic architectural approaches.

In general, software architecture defines the structure of the system as depicted by Len Bass et al. in [1], but the self-managed systems state some explicit requirements to achieve the autonomic features.

### **Definition**

*The software architecture of a program or computing system is the structure or structures of the system, which compromise software elements, the externally visible properties of those elements, and the relationships among them [1, p. 3].*

A self-managed system or an autonomic system acts in the same way as a robot. This means that a robot's sense-plan-act (SPA) behavior corresponds exactly to a self-managed system according to Kramer et al. in [7, p. 3]. In other words the architecture for an autonomic system must achieve the same properties as for SPA architectures, but to achieve the properties of the lowest level, an autonomic architecture must match the definition of regular software architectures as well.

In [2] Daniel A. Menascé et al depict three general techniques [2, p. 19 - 20] to help move forward within the research of autonomic computer systems. To turn these techniques into architectural approaches it will require a few other requirements expressed as potential benefits [7, p. 2].

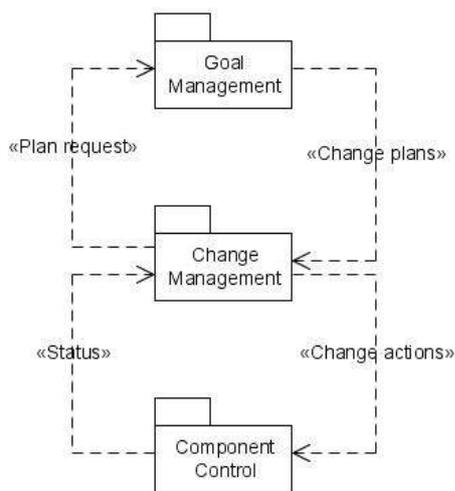
These benefits are used in [7] when they define a three-layered reference model that can be used for autonomic computing. This is also one of the reasons why that particular reference model is introduced in the following.

#### A. Autonomic architectures

In [7] Kramer, et al. depict several approaches summarized from earlier research. This paper will in particular look into the approach described in [7, p. 3] which refers to the research of artificial intelligence and mobile robots made by Erann Gat from Jet Propulsion Laboratory, California Institute of Technology.

*The three-layers are Control: reactive feedback control, Sequencing: reactive plan execution and Deliberation: planning [7, p. 3].*

The three layers of architecture are described in [7, p. 3 - 4]; the bottom layer consists of sensors, actuators and control loops. The bottom layer of a self-managed architecture consists of interconnected components used to accomplish the application function of the system.



**Fig 3: The three-layered architecture**

The bottom layer is the operational layer responsible for component surveillance by reporting the state of the components to the upper layers as well as supporting component creation, deletion and interconnection.

The middle layer is the sequencing layer responsible for execution of the plans when changes in the state of the components are reported by the lower layers. According to Kramer et al., in a self-managed system, this layer is responsible for effecting changes to the underlying component architecture in response to the reported states or in response to new objectives introduced from the upper layers.

The upper or uppermost layer is the deliberation layer which consists of time consuming computations such as planning which states related to which specifications of high level goals. This layer tries to produce a plan to achieve these goals. According to Kramer et al. this layer is responsible for producing plans, and change management plans in response to requests from the layers below and the introduction of new goals.

*We have defined a three-layer reference model – component control, change management and goal management – to provide a context for discussing the main research challenges which self-management poses [7, p. 7].*

In [7, p. 7] Kramer et al. define these three layers as an architectural reference model that can be used for further discussion of research challenges within the subject of self-managed systems. In other words, this defines an architectural reference model that can be used for autonomous systems.

The reference model leads directly to the next section which introduces two autonomic-like systems. The components within these systems will be identified respectively in correspondence to the respective layers in the reference model.

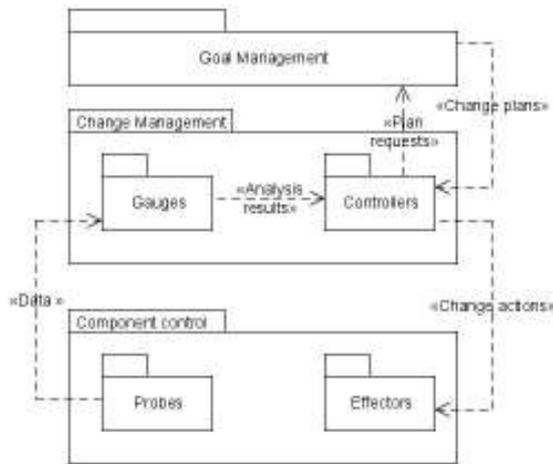
#### B. Monitoring distributed systems

In [4] Kaiser et al. depict an external infrastructure to monitor distributed legacy systems. The infrastructure, named Kinesthetics Extreme, is made with the Java language, but even though this work is centered on C++ language tools, its architectural overview and implementation is still relevant to this subject.

The approach is generally centered on two terms that are related to data collection and categorization; probes that are attached to the target system to collect data and gauges that aggregate, filter and interpret the probed data, but in total five first-class entities are defined in the infrastructure.

The individual components (probes, gauges and controllers) are loosely coupled in the architecture by making them event-based. This is done by a standardized event middleware, which currently is a system called Siena created by U. Colorado that is based on the publisher/subscriber mechanism.

The infrastructure contains an event distiller which supports dynamic rule generation by allowing messages to be sent with XML snippets specifying a rule or a rule segment with the purpose of constructing new rules on the fly or modifying an existing rule. Furthermore, during the development of this infrastructure, learning techniques to build rules in a more autonomic fashion have also been investigated.

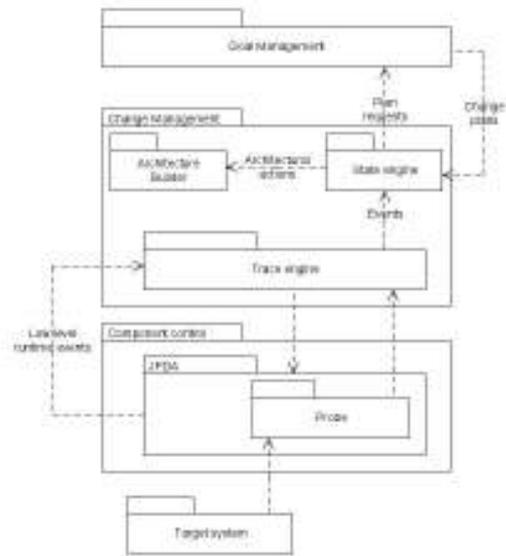


**Fig 4: The infrastructure placed in the three-layered architecture**

The development of the infrastructure indirectly introduces autonomic systems and makes it relevant to place the individual components into the three-layered reference model outlined earlier. In the illustration above, the components are placed into that reference model. The bottom layer takes care of the component control which collects and reports states by the probes and support component creation, deletion and interconnection by the effectors. The middle layer executes the plans by the controllers which act on the results from the analysis produced by the gauges. The top-level is not yet directly addressed, but the controllers accept modification to plans and creation of new plans.

### C. Runtime discovery of architectures

Another autonomic-like system that has been made is a system that is able to discover architectures from running systems, but as with the infrastructure described earlier, this has also been made in Java. This system is called DiscoTect [6] and it has been developed by a group of five individuals from the Carnegie Mellon University. The goal of this system is to ensure consistency by construction, ensure conformance by extracting the architecture from a system's code, using static code analysis and determining the architecture of the system by examining its behavior at runtime.



**Fig. 5: DiscoTect placed in the three-layered architecture**

This architecture does not in the same way introduce the autonomic architecture, but it does have some of the same features. It is primarily centered on collecting data from a running system which can be related to the features in an autonomic system that supports the consolidation of logs located at the second level on the autonomic axis. The trace engine interprets the low level events and parses them onto the state machine. The state engine bridges the events to high level operations used by the architecture builder to form building blocks. These responsibilities place the trace engine, the state engine and the architecture builder on the change management layer.

Emphasis and categorizing systems by certain definitions is one thing, another thing is how to realize and achieve these definitions properly. The research of this subject includes several techniques that have been used to build systems with self-managed properties. Among these techniques are: control theory, queuing models combined with heuristic search techniques, and machine learning [2, p. 19]. These techniques also include collection of data which enables data consolidation from several systems or sources. To do this some techniques that can be used for this purpose will be introduced in the following section.

## III. COLLECTION TECHNIQUES

Common for autonomic architectures is that the lowest level is for data consolidation. In order to be able to perform data consolidation it is necessary to collect data that can be consolidated. In this section a few techniques that can be used to collect data from running applications will be outlined.

In general, data for error handling can be collected in many ways e.g. data mining, exception handling and data collection by dynamic probes, but the most appropriate one that is independent of various log files is the use of dynamic probes and data consolidation.

- Data mining collects data from all existing logs, but in order to do so the knowledge of all logs and their locations must be present. Secondly, a parser of each log that has a different format exists.
- The use of exception handling logs data to the applications own logs instead of a common log used for all applications and it will typically be in a proprietary format, depending on the application which requires a parser for each format.
- Dynamic probes and data consolidation are able to collect data in one format and broadcast the data to one consolidated log i.e. by a message queue, but it requires that the probes can be injected into the application. Secondly, the collected data have to be bridged to higher level data i.e. in an automatic process using pattern matching where patterns are defined either manually or automatically in some way.

The following techniques described in this section are related to the dynamic probes approach and they primarily belong to the lower levels of the autonomic axis. In other words, how would it be possible to collect the states from the components and populate the information about the states to the upper layers?

In the following, it is outlined how some techniques can be used to collect information from a target system. The techniques are directly related to data extraction from a system using aspect weaving. Aspects and aspect weaving will be explained later in this section. The weaving techniques can in general be grouped into two different categories.

- Compile-time / static
- Runtime / dynamic

Normally when doing a compile-time weaving, declarations of elements have to be visible to the source of the target system before it is being compiled. When performing a runtime weaving, this is simply impossible. Either way, it is possible to statically prepare the target system by making it aware of the aspects which is why it is called prepared dynamic weaving.

Generally, runtime weaving does not have the same set of features available as compile-time weaving has. Runtime weaving has several limitations to these features [9] while some must be statically prepared and others are impossible or illegal to use.

A brief summary of the features is that many of them can only be used dynamically if the modules have been prepared for them at compile-time which is also why it is called prepared dynamic weaving.

#### A. Aspect weaving

Aspect weaving is a mix of two techniques; weaving means insert code snippets into the target system, that can be either dynamically or statically, and aspect oriented programming. Today, several approaches exist that implement aspect weaving. One of these approaches is a compile-time technique that is a language extension to C++ called Aspect C++. It is developed by Andreas Gal and Olaf Spinczyk from the University of California, Irvine.

The extension works at compile-time which means it uses static weaving. To work at compile-time, it is necessary to inject source into the target application. The primary reason for focusing at compile-time techniques is due to the additional costs of having the target system perform runtime code manipulation on embedded systems with tight resource constraints [5, p. 53].

Compile-time techniques are often recommended when the target applications run on a platform with limited resources. Instead of monitoring the target application by a third-party tool which requires additional resources that might not be available, it is often more convenient to inject source code into the target application to produce the necessary data. However, due to the nature of a compile-time technique, it requires that the application is stopped and that the source code is available.

As its name implies, the use of aspect weaving introduces aspect oriented programming as a single dimension of functional decomposition is insufficient to emphasize all aspects of a program in a modular way. Aspect Oriented Programming (AOP) tries to solve this.

#### Definition

*An aspect-oriented language environment allows implementing such crosscutting concerns in modular units called aspects ... [5, p. 53]*

The modular units are woven into the target system using an aspect weaver responsible for injecting code fragments into either the source code or the binary of the target application depending on which kinds of weaving are used.

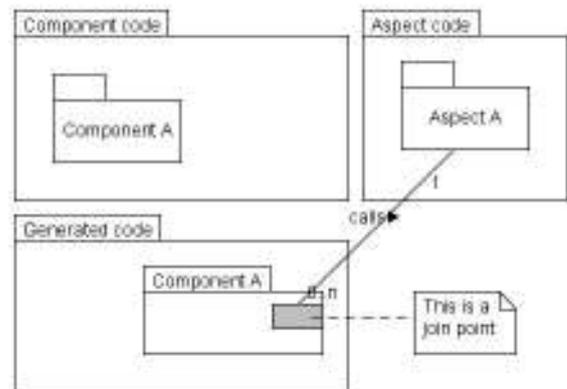


Fig. 6 The weaving aspect

In the illustration above, two tools are often used to generate the merged code; a tool that supports an expressive aspect language, and a tool to take care of the weaving process. The illustration introduces a join point. The definition of a join point is defined by Andreas Gal and Olaf Spinczyk as a reference in the code in the target system which accesses a modular aspect component.

**Definition**

*A join point refers to a method, an attribute, a type (class, struct or union), an object, or a point from which an aspect is accessed [5, p. 54].*

Several join points can be described by an expression called a point cut expression, which defines a point cut as a set of join points. The power of point cuts is that they can describe points in the static structure or in the dynamic control flow of a program.

**Definition**

*A point cut is a set of join points described by a point cut expression [5, p. 54].*

A point cut can be added with an extra trigger-like functionality that can be used to perform logging activity or other activities.

**Definition**

*An advice declaration can be used to specify code that should run when the join points specified by a point cut expression are reached [5, p. 54].*

*An advice is a C function call that replaces a join point in the base program execution [10, p. 5].*

Such trigger-like functionality is called an advice declaration [5, p. 54]. Even though named point cut declarations can exist anywhere, advices can only exist inside an aspect declaration.

As opposed to compile-time techniques, runtime techniques often require an expensive dynamic weaving structure when making dynamic adaptation of crosscutting concerns. When using compile-time techniques the tailoring of a target system needs to be stopped and restarted to make it possible to use the code injected into its source code, but this is not always an option.

When dealing with large applications it often takes a long time to compile and not all applications can be stopped for weaving. This emphasizes the need for runtime techniques because it is not always possible to stop, compile and restart a target system.

The need for runtime weaving exists, but due to the nature of raw data it is much more complicated. A few languages or language extensions support runtime weaving for the C++ language. Some of these languages support only dynamic weaving if the source has been prepared for it while others are able to make dynamic weaving without preparation. In other words, dynamic weavers can be categorized into dynamic weaving and prepared dynamic weaving.

**B. Dynamic weaving**

A few tools to instrument a running system have been found. One of these tools introduces dynamic probes [12]. The tool is a platform-independent interface developed by R. J. More, who is from the USENIX association. Another tool to allow weaving of a running system is an API for runtime code patching that has been developed by Bryan Buck and Jeffrey K. Hollingsworth from the University of Maryland. The Application Programming Interface (API) is named DynInst [14]. The goal of the DynInst API is to provide a machine-independent interface that permits the creation of tools which use runtime code patching and keep a small and easy understandable interface.

This article is primarily based on a non platform-independent C++. These delimitations exclude the API for runtime code patching primarily because it uses assembler code to describe aspects, and secondly even though the API is platform-independent for runtime code patching, it is not a requirement for this article. In other words, this leaves the DynInst as the only valid technique among the two.

The construction of the DynInst interface is divided into three main components. The classes in these components can be categorized into the responsibilities outlined in the taxonomy below, where each responsibility represents a component.

Responsibility	Classes
Manipulation of the code in execution	BPatch, BPatch_thread
Accessing the original program and its data structures.	BPatch_image, BPatch_module and BPatch_function
Construction of new code snippets	BPatch_snippet, BPatch_point

*Responsibilities in DynInst derived from [14, p. 3]*

The uses of the classes also introduce the general terms by the language constructs of this tool. When defining these terms it will become clear that many of them are similar to definitions depicted earlier, but defined with another vocabulary.

Term	Definition	Similar to
Point	A point is a location in a program where weaving can take place [14, p. 2].	A join point
Snippet	A snippet is a representation of a bit executable code to be inserted into a program at a point [14, p. 2].	An advice
Thread	A thread refers to a thread of execution [14, p. 2].	-
Image	Images refer to the static representation of a program on disk. Images contain points where their code can be modified. Each thread is associated with exactly one image [14, p. 2 - 3].	-

*Vocabulary defined by DynInst [14, p. 2 - 3]*

As a matter of fact, DynInst is able to handle viewing of multiple processes. This is done by introducing a thread that refers to a certain thread of execution and from a static point of view an image represents a binary image on a disk such as a dynamic-link library or an executable. Due to the last definitions and this definition itself, a binary image is accessed through a thread and contains points that allow the injection of snippets.

DynInst creates an application thread based on a binary. In the example above it starts a new execution, but it is possible to attach an existing running thread by specifying a file name and process id for the actual thread. Secondly, when searching for procedure points please note that it returns a collection of points. This is because the search returns matches that include functions cloned on different locations and overloaded functions.

### C. Prepared dynamic aspect weaving

Approaches for dynamic aspect weaving have been outlined, but due to some dead ends such as weaving tools that neither can be compiled nor downloaded because they were inaccessible, it has been necessary to investigate another approach.

A Dynamic Aspect C++ (DAC++) language [18], which is an extension to the C++ language, has been developed during a PhD program by Sufyan Al Majali and Tzilla Elrad from Illinois Institute of Technology. DAC++ is using three features to allow dynamic weaving of applications. These features [18, p. 2] include both dynamic loading and dynamic weaving.

The architecture of the approach uses two components: a preprocessor and an AOP engine. The preprocessor generates the meta object data needed at runtime. The metadata is minimal hooks [18, p. 6] inserted into the source code at all possible join points.

Term	Definition	Similar to
Minimal hook	A minimal hook represents a piece of code that allows our system to weave in an aspect at the hook location if needed [18, p. 6].	A join point

*Vocabulary defined by DAC++ [18, p. 6]*

The prepared and compiled binary can now be run with metadata containing information about program classes and methods. This information is used by the dynamic aspect and a designated AOP engine when weaving at runtime.

Linux G++ and DAC++ gave the same performance for these benchmark tests. No overhead was introduced using these benchmark tests. This is excepted as DAC++ generates C++ code that is fed eventually to the G++ compiler. The difference in performance started appearing when we tested the performance of method calls.

Benchmark	Hard Coded Aspect (Static)	DAC++
Method Call 1 (small)	32 seconds	42 seconds 31%
Method Call 2 (average)	36 seconds	44 seconds 22%

*Method Call Performance Evaluation*

The table shows two benchmark tests for method call with 100,000,000 iterations. One is for small methods sizes and the other one for average method sizes. There was no aspect woven in these tests. DAC++ has added extra overhead. This is because DAC++ uses pointer to member function call mechanism. The two benchmarks are strict testing for method call invocation performance.

Please notice that average size methods have less overhead than small ones (31% for small ones and 22% for average ones). This is because the method will have more instructions to execute which requires more CPU cycles to execute the method body rather than just calling the method.

## IV. INFORMATION MAPPING TECHNIQUES

The techniques described collect data from a running application. The collected data is very low level. This makes it difficult for usage on high level decision making as required, when the collected low level data is to be used to determine the states of the applications in order to achieve the requirements for the upper levels on the autonomic axis. To solve this it is necessary to map the low level data into a high level model.

The research outlined in [15] by Robert J Walker et al is primarily a target for construction of architectural views and it is used along with the DiscoTect approach [6]. The technique is used to collect structural information derived from class method events, which can also be used to clarify the behavior of a system by constructing a set of patterns which match the traces. This behavior can be used to tell whether a system is in a bad state or not because prior to any failures comes a certain behavior. The reason why this technique is used, even though this article has nothing to do with construction of architectural views, is to clarify the behaviors prior to such failures by the use of class method events.

The part of the research that will be included is primarily the part about mapping traces [15, p. 5 - 8]. The mapping traces are centered on certain types of events (class method entry and exit, instance method entry and exit, object allocation and de-allocation, thread start and stop). It is exactly these events that can be traced by static and dynamic weaving. In correspondence to the prototype aspect language it will primarily be centered on the first event type, which are class method entry events and class method exit events.

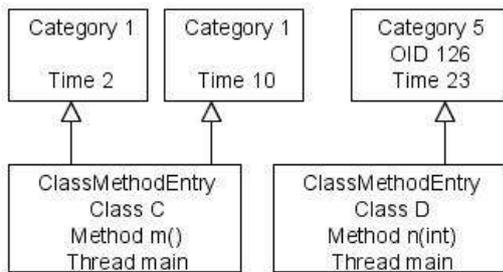


Fig. 7: Encoding scheme [15, p. 7]

The mapping traces are based on an encoding scheme which defines certain encoding events. These events are based on the determination of the patterns that can be used when determining the information needed. The goal of the encoding scheme is to categorize the events and encode the categories. This is done by recording the traces in two streams: an encoding stream and an event stream. The event stream is also based on a sequence of records, but it contains an index to a primitive category [15, p. 6] in the encoding stream and some additional information that depends on the specific type of event.

The mapping specifies a set of primitive categories to an abstract category [15, p. 7] through a partial, ordered specification of matching criteria. In other words each encoding stream record has its class identifier compared against this matching criterion. If it matches it is placed into its respective abstract category.

## V. AN ARCHITECTURAL PROTOTYPE

The previous sections outline an autonomic architecture and some techniques that can be used to construct a consolidated log that will be able to clarify failures from running Windows services. In the following it will be demonstrated by a proof of concept how these techniques can be used to construct such a log.

The proof of concept is primarily centered on two use cases. The first use case is a prerequisite of the other. This is because the first use case outlines the installation while the other outlines the daily usage.

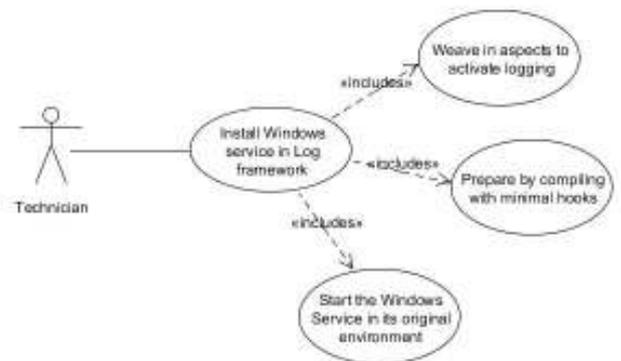


Fig. 8: Log-enabling of a Windows service

The first use case includes the installation or log-enabling of a Windows Service into the consolidated log, it is required that the Windows Service is compiled with the minimal hooks to prepare it for runtime weaving. Whenever this is done the Windows service can be started under its original environment and the aspects can be woven into the binary to activate the logging.

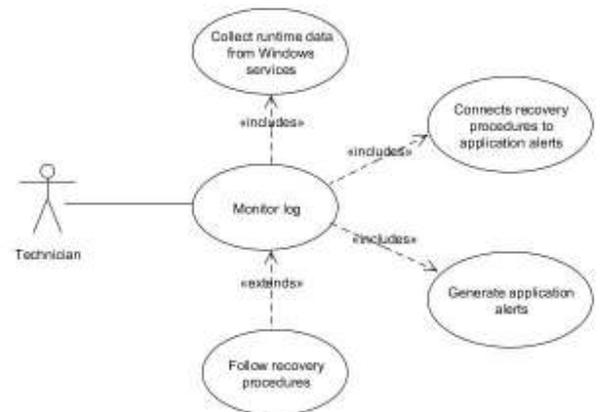


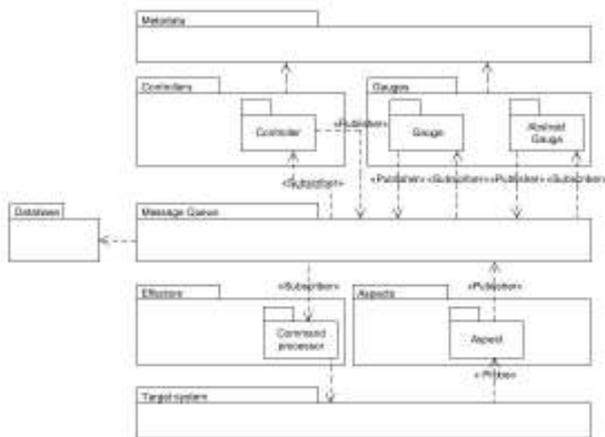
Fig. 9: Daily usage

The second use case deals with the daily usage of the consolidated log. It is primarily based on the operators that monitor the actions and follow a recovery procedure if necessary. The consolidated log includes a collection of runtime data from Windows services, generation of application alerts and recovery procedures.

Finally with these usage cases taken into account, the architecture is divided into the main component types; controllers, gauges, aspects and effectors. These components are connected in a loosely and decoupled structure by a message queue that is working with a publisher/subscriber mechanism [16, p. 339].

The reason for choosing a loosely and decoupled structure is to achieve the same properties as the Kinesthetics Extreme approach [4] that is designed to monitor distributed legacy systems. Of course, this results in an architectural approach similar to the architecture of the Kinesthetics Extreme approach which is designed in a loosely and decoupled structure, generally centered on two terms that are related to data collection and categorization.

In general, the proof of concept contains the following packages. Four of these packages represent a component; the metadata package is a utility package that is used to access and parse the metadata and message documents. The last packages contain the implementation of the message queue, the target system and the metadata repository.

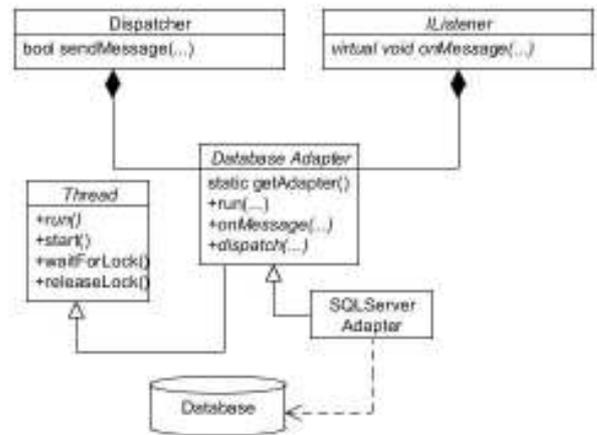


**Fig. 10: Package diagram**

The target application calls the aspects through its join points or minimal hooks. The aspects collect the necessary information such as method and class name published on the message queue by the advice belonging to the relevant aspect. The information published by the aspects is received by the gauges that interpret and transform the low level data to high level events. The high level events are published on the message queue and received by the controllers, which send the actions as commands to the effectors that act as their delegates when reestablishing a target system from its bad state as outlined in the message it has received.

The architecture must be formed in a way that encapsulates the modules in a decoupled structure. The decoupled structure is achieved by introduction of a message queue, which is used to communicate among the individual components.

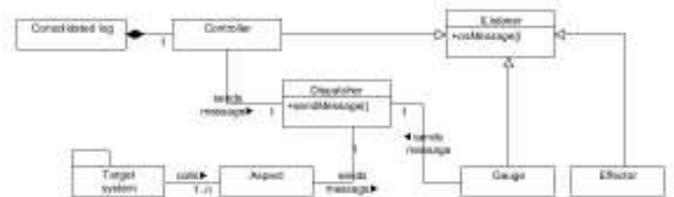
Message queue software from third-party vendors exists, but these have been avoided. The reason for this is to avoid architectural mismatches in relation to the provided and especially the required assumptions [1, p. 457] given by the third-party software.



**Fig. 11: The message queue design**

The foundation of the message queue is based on a repository for messages in transit. The repository holds information contained in the header such as subject, status and sender among others, but of course it holds the message itself as well. The communication mechanism for the message queue is given by an interface through which the data is pushed to the components. The components listen to a subject and receive the respective messages that have been published on that particular subject. In general, the message queue will act as a publisher/subscriber mechanism.

The current implementation of the message queue is based on a third-party database, a Microsoft SQL Server database, as its repository. The database is accessed using an adapter pattern. This is to convert the interface for the database into an interface that matches the architecture of the proof of concept in a more convenient way and to separate the vendor-specific interface from the rest of the proof of concept to allow an easy replacement of the repository.



**Fig. 12: Component and message queue construct**

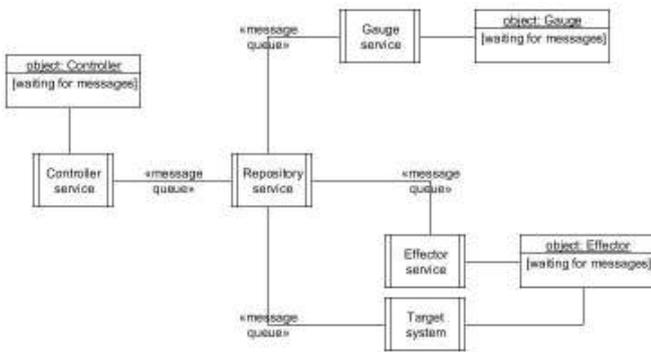
The construction above illustrates how the target system is connected. The target system publishes messages through the aspects on the message queue. The gauges listen on the message queue for any low level data. This low level data is transformed into higher level messages by gauges and published on the message queue to be received by the controllers.

**Definition**

*The consolidated log will act as the controllers through its aggregation. The consolidated log displays a list of high level events and recovery actions received from the aggregated controller.*

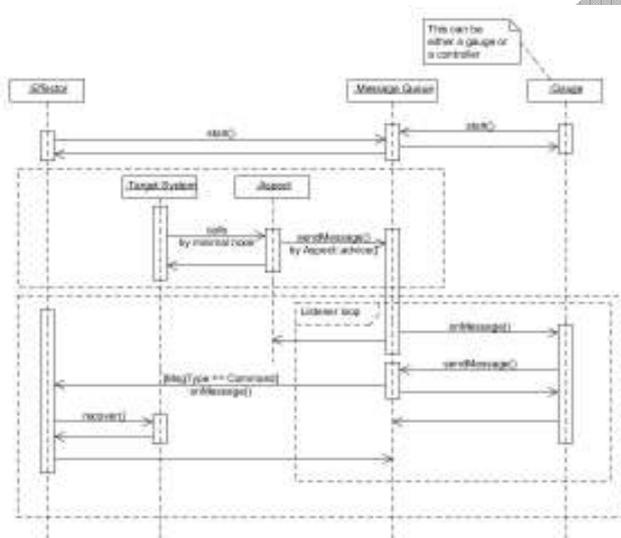
The consolidated log will act as a controller component. It monitors high level messages and application alerts. Depending on the high level messages, it outlines the recovery actions that can be processed manually.

The proof of concept contains the following components. One of these components is the central repository responsible for storing messages in transit, event scheme data and metadata.



**Fig. 13: Component and connectors**

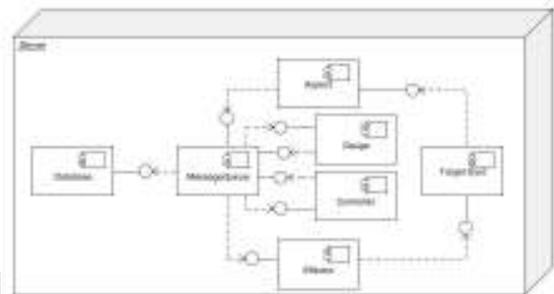
The figure above illustrates how the components interact with each other through the message queue. The message queue itself forms the link between components. The message queue interface is a façade [19, p. 185] that stores the messages in a repository such as an external relational database.



**Fig. 14: Sequence diagram that outlines the communication between components**

In the diagram, it is outlined how a message is dispatched on the message queue. The target system calls the respective aspects through its minimal hooks. The aspects collect desirable low level trace information, which is populated on the message queue by the advice belonging to the respective aspect. The data is received from the message queue by listeners. All components that are going to receive messages from the message queue implement an interface through which the messages are received and the message queue itself makes the connection between each component.

The architecture of the proof of concept allows several setups. All components can be placed on different nodes, all components share one node or only some components share a node while others have their own node. The first setup is described in the following view.



**Fig. 15: A standalone configuration**

This is of course the simplest configuration where all components are placed on the same node as the target system. The node also has a database locally installed which is accessed through an Open Database Connectivity (ODBC) connection.

## VI. IMPLEMENTATION HIGHLIGHTS

Starting from the lower levels, it introduces modular units as aspects. These must be used to collect data used for the consolidated log. Even though this is the lowest level it is also the most important. This is because the lower levels collect the traces that depict the behavior and (indirectly) the states of the target systems.

The modular units are built as aspects which contain business logics that are related to its proper area of responsibility. In this situation, their primary responsibility is to collect data that makes them identifiable as primitive categories to allow data to be stored in the event scheme.

Aspect	class TraceAllAspect : public Aspect { public: void advice(); }; Collects entries of all class method calls
Point cut expression	*

*Aspect, header information*

The aspect outlined above is tracing all on-entry function calls. This will of course produce a very large amount of low level traces and it might be relevant to add a filter to prevent it all to be published on the message queue. However, filtering mechanisms have been omitted from this proof of concept.

```

Advice void advice() {
    std::stringstream ss;
    ss << "<trace>"
        << " <component>"
        << " <filename/>"
        << " <lineno/>"
        << " <executable>"<<getExecFileName()
        <<"</executable>"
        << " <processid>" << ::GetCurrentProcessId()
        <<"</processid>"
        << " <threadid>" << ::GetCurrentThreadId()
        << "</threadid>"
        << " </component>"
        << " <object>"
        << " <class>"<< GetJPCClassName().c_str()
        <<"</class>"
        << " <method event=\"entry\">"
        << GetJPMMethodName().c_str()
        <<" </method>"
        << " </object>"
        << "</trace>\n";

    ofstream File("c:\\message.xml");
    File.write (ss.str().c_str(),strlen(ss.str().c_str()));
    File.close();
    const char* szDefVal =
        "C:\\...\\AspectsExecutable.exe c:\\message.xml";

    DWORD dwSize = 256;
    char szAspectExec[256];

    ::GetPrivateProfileString("Aspect", "ExecCmd", szDefVal,
        szAspectExec,dwSize,
        "InfobusPrototype.ini");

    ::WritePrivateProfileString("Aspect",
        "ExecCmd",
        szAspectExec,
        "InfobusPrototype.ini");

    try { system(szAspectExec); } catch (...) {
    }
}

```

*Aspect, implementation details*

The aspect is applied to the target application by adding a source snippet describing a point cut expression among others to determine the validity for the aspect. This validity depicts when the aspect is to be called by identifying a pattern for class name and method and the weaving type to determine whether it should be called for entry or exit events.

```

Setup TraceAllAspect b;

    MethodPC meth;
    ClassPC classpc;
    PCD pc;
    WeaveSpecs ws;

    DAC_INIT();
    meth.setvalue("");
    classpc.setvalue("");

    pc.setmethodPC(meth);
    pc.setclassPC(classpc);

    ws.setweavetype(Before_T);
    b.setweavespecs(ws);
    b.setpcd(pc);

```

*Enabling the aspect in the application*

To allow the aspect to be used it is necessary to inject the join points into the source code. The table below outlines a snippet of the run() method from one of the thread classes after the injection has been processed.

```

run() public : virtual void run ( ) { _dac_arrayvoidptr
    _dac_array_act_rec;
    _dac_arrayvoidptrnodeptr _dac_ptr2 ,_dac_ptr3;

    _dac_ptr3=NULL;
    if (MethodMOP[15].before)
    { Aspectnodeptr _dac_ptr1;
    _dac_ptr1 = MethodMOP[15].beforeHead; while
    (_dac_ptr1!=NULL)
    { _dac_execute_before(15,_dac_ptr1,_dac_ptr3);
    _dac_ptr1=_dac_ptr1->next; }
    } _org_run ();
    if (MethodMOP[15].after)
    { Aspectnodeptr _dac_ptr1;
    _dac_ptr1 = MethodMOP[15].afterHead; while
    (_dac_ptr1!=NULL)
    { _dac_execute_after(15,_dac_ptr1,_dac_ptr3);
    _dac_ptr1=_dac_ptr1->next; }
    }
}

```

*After the injection of join points for the run() method*

A sequence of traces, collected by the aspect, is analyzed to identify the behavior of the target systems. This makes it possible to construct a pattern which matches that particular sequence by knowing which method calls are produced by a certain behavior. The pattern can further be used to identify high level events and application alerts to which recovery actions can be related.

## VII. EVALUATION

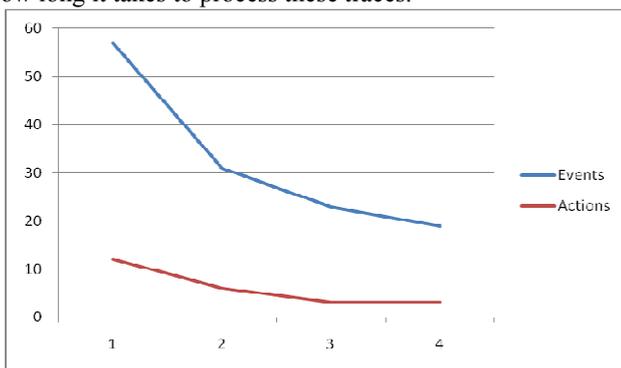
The evaluation primarily focuses on two aspects: performance and adaptability. In a large infrastructure the robustness, adaptability and performance for large throughputs is important. This makes it possible to use various kinds of Windows services.

The following benchmarking is centered on the generation of traces produced by only one Windows service. This is to simplify the scenarios.

In an infrastructure where lots of traces are produced it is necessary for the prototype to be scalable. This is can be expressed by a quality attribute scenario, that makes it possible to measure whether the prototype is scalable or not.

*The number of Windows services and the number of gauges is increased by a factor four to speed up the performance approx. 30 percent while all running consolidated logs keep monitoring all affected Windows services in real-time.*

The Windows Service has generated 226 traces within one minute or to be more precise 59 seconds. The test of the other components is based on these 226 traces, where it is measured how long it takes to process these traces.



**Fig. 16: Processing time when increasing the number of gauges by a factor four.**

The diagram illustrates how long it takes (in seconds) to transform the 226 traces into events and further transform these events into actions, when upscaling the prototype. The number of scalable components has been increased from one to four.

The adaptability issue has been proven by log-enabling a small third-party telnet service. It can be downloaded from [codeproject.com](http://codeproject.com) [20] and called NT Telnet server and client. It is a tiny Windows Service created in the C++ language.

## ACKNOWLEDGEMENT

Thanks to everyone who helped with this article. Especially thanks to Localization Manager Vibeke Batchford at Trapeze Group Europe A/S for proofreading this article. She holds a MA degree in English.

## REFERENCES

- [1] SOFTWARE ARCHITECTURE IN PRACTICE, SECOND EDITION, Bass L, Clements P, Kazman K, Addison Wesley 2003. ISBN: 0-321-15495-9
- [2] AUTONOMIC COMPUTING, DANIEL A. MENASCÉ, JEFFREY O. KEPHART, GEORGE MASON UNIVERSITY, IBM RESEARCH, IEEE COMPUTER SOCIETY, JANUARY / FEBRUARY 2007 (VOL. 11 NO. 1)
- [3] Week 20: Architectural reflection, KLAUS MARIUS HANSEN, UNIVERSITY OF AARHUS, DEPARTMENT OF COMPUTER SCIENCE, [HTTP://WWW.DAIMI.AU.DK/~MARIUS/TEACHING/ASAIP2008/](http://www.daimi.au.dk/~marius/teaching/asaip2008/), VISITED: 21-02-2010
- [4] KINESTHETICS EXTREME: AN EXTERNAL INFRASTRUCTURE FOR MONITORING DISTRIBUTED LEGACY SYSTEMS, Gail Kaiser, Janek Parekh, Philip Gross, Giuseppe Valetto, Columbia University Programming Systems Lab, Department of Computer Science
- [5] ASPECT C++: AN ASPECT-ORIENTED EXTENSION TO THE C++ PROGRAMMING LANGUAGE, Oloaf Sincyk, Andreas Gal, Wolfgang Schröder-Preikschat, University of Magdeburg, University of California
- [6] DISCOTECT: A SYSTEM FOR DISCOVERING ARCHITECTURES FROM RUNNING SYSTEMS, Hoang Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, Rick Kazman, Carnegie Mellon University
- [7] Self-Managed Systems: an Architectural Challenge, JEFF KRAMER, JEFF MAGEE, DEPARTMENT OF COMPUTING, IMPERIAL COLLEGE LONDON
- [8] Dynamic Software Reconfiguration in Software Product Families, HASSAN GOMAA, MOHAMED HUSSEIN, DEPARTMENT OF INFORMATION AND SOFTWARE ENGINEERING, GEORGE MASON UNIVERSITY
- [9] Static and Dynamic Weaving in System Software with AspectC++, PROCEEDINGS OF THE 39TH HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES - 2006, WOLFGANG SCHRÖDER-PREIKSCHAT, DANIEL LOHMANN, FABIAN SCHELER, WASIF GILANI, OLAF SPINCYK, FRIEDRICH-ALEXANDER UNIVERSITY ERLANGEN-NUREMBERG
- [10] An expressive aspect language for system applications with Arachne, RÉMI DOUENCE, THOMAS FRITZ, NICOLAS LORIAN, JEANMARC, MENAUD, MARC S'EGURADEVILLECHAISE, MARIO SÜDHOLT, OBASCO PROJECT, ÉCOLE DES MINES DE NANTES / INRIA
- [11] TinyC2: Towards building a dynamic weaving, ASPECT LANGUAGE FOR C, CHARLES ZHANG, HANS ARNO JACOBSEN, DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING AND DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF TORONTO
- [12] Dynamic probes and generalized kernel hooks interface for Linux, PROCEEDINGS OF THE 4TH ANNUAL LINUX SHOWCASE AND CONFERENCE, ATLANTA, OCTOBER 10-14, 2000, ATLANTA, GEORGIA, R. J. MOORE, USENIX ASSOCIATION, BERKELEY, CA, USA, 2000
- [13] Part of DESIGN METHODS FOR REACTIVE SYSTEMS, R. J. Wieringa, Morgan Kaufmann 2003, ISBN: 1-55860-755-2
- [14] AN API FOR RUNTIME CODE PATCHING, Bryan Buck, Jeffrey K. Hollingsworth, Computer Science Department, University of Maryland
- [15] EFFICIENT MAPPING OF SOFTWARE SYSTEM TRACES TO ARCHITECTURAL VIEWS, Robert J. Walker, Gail C. Murphy, Jeffrey Steinbok, Martin P. Robillard, Department of Computer Science, University of British Columbia
- [16] PART OF Pattern-Oriented Software Architecture. Vol. 1: A System of Patterns, FRANK BUSCHMANN, REGINE MEUNIER, HANS ROHMERT, PETER SOMMERLAND, MICHAEL STAL, ISBN: 0-47195-869-7, JOHN WILEY AND SONS, LTD. APRIL 2006.
- [17] Architecture Decisions: Demystifying Architecture, JEFF TYREE, ART AKERMAN, C, APITAL ONE FINANCIAL IEEE SOFTWARE, MARCH / APRIL 2005
- [18] Dynamic Aspect Oriented C++ for Application Upgrading Without Restarting, SUFYAN ALMAJALI, TZILLA ELRAD, ILLINOIS INSTITUTE OF TECHNOLOGY
- [19] PART OF Design Patterns Elements of Reusable Object-Oriented Software, ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, JOHN VLISSIDES, ISBN: 0-201-63361-2, ADDISON-WESLEY, DECEMBER 1998
- [20] NT TELNET SERVER AND CLIENT, CODEPROJECT.COM, <http://www.codeproject.com/KB/IP/telnetview.aspx>



**Jess Nielsen** is a Senior Developer at Trapeze Group Europe A/S. His research interests include autonomic systems, software architectures and cryptographic. He received his master's degree in IT (Software Development) from Aarhus University, Denmark.  
Contact him at [jessn@bluebottle.com](mailto:jessn@bluebottle.com).



**Sufyan Almajali** is an Assistant Professor at the Computer Science department, Princess Sumaya University for Technology in Jordan. His research interests include Aspect Oriented and Object Oriented Programming, Software Engineering, Compilers and Networking. He received his PhD in Computer Science from Illinois Institute of Technology, Chicago, USA.  
Contact him at [s.almajali@psut.edu.jo](mailto:s.almajali@psut.edu.jo)