Modular Integration Through Aspects: Making Cents of Legacy Systems

Celina Gibbs[+], Daniel Lohmann[*], Chunjian Robin Liu[+], and Yvonne Coady[+]
[+]*University of Victoria, Friedrich-Alexander-University Erlangen-Nuremberg*
celinag@uvic.ca, *lohmann@cs.fau.de*, *clui@cs.uvic.ca*, *ycoady@cs.uvic.ca*

Abstract

Recently, Continua Health Alliance has brought together a powerhouse team, including Cisco, IBM, Motorola and others, for personal telehealth products and services. This team will provide commodity interoperable healthcare devices and services by introducing new connectivity standards for health management tools. But the costs of integrating and configuring disparate system services have proven to be prohibitive in this domain – healthcare processes require extreme agility to assimilate information across traditional boundaries. As a result, these tools must work effectively with dynamic business processes that often elude cost-effective integration themselves. This creates a requirement for software to be fluidly configurable and interoperable in order to best support personalized care with truly integrated solutions. We believe that, without a new technology for the seamless integration of features within healthcare devices, costs associated with attempts to fuse IT with dynamic business processes will continue to be an obstacle in modern patient care.

Aspect-Oriented Software Development (AOSD) is focused on novel notions of modularity that crosscut traditional abstraction boundaries. AOSD techniques and tools, applied at all stages of the software lifecycle, are changing the way software is developed in a wide spectrum of application domains, ranging from embedded systems to enterprise IT. This paper outlines the ways in which aspects could aid the integration and evolution of software used to support modern healthcare practices across this spectrum, with examples at each stage. We believe the key principle of AOSD – the modularization of crosscutting concerns – to be an integral part of the solution to the challenges currently facing modern health service infrastructures.

1 Introduction

Business processes in healthcare are changing rapidly. This is largely due to new technologies that enable processes that were unthinkable not that long ago. Astonishingly, the changes in business processes due to evolving technology also bring about changes in technology due to evolving business processes. Specifically, demands for highly configurable systems to support more personalized healthcare needs fall into this circular relationship. Everything from wireless sensor network technology to adaptive web applications has been put to the test in these dynamic environments. However, the inability for these systems to respond has posed a major challenge in process integration and evolution today.

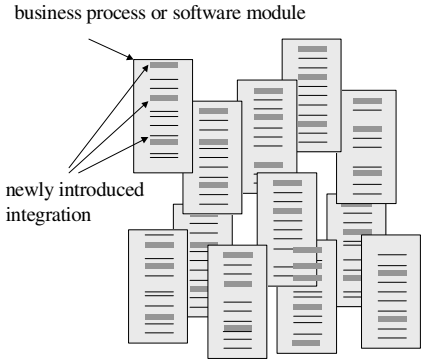

Figure 1 - Scattered and tangled introduction of an integration concern.

Integration is hard – both at the level of business processes and legacy software systems. We further believe that the inherent fusing of the two, and the circular relationship involved, makes it even harder.

[*] This work was conducted during a research stay at University of Victoria, supported by the German Academic Exchange Council (DAAD) under grant no. D/06/40386

1530-1605/07 $20.00 ©2007 IEEE

In this paper we suggest that the nature of the problem is fundamentally the same in both the business process and technology domains: a set of well defined elements (either business processes or software modules) with well defined interfaces begin to decay as new issues for integration are introduced to the system. Support for integration must be scattered across these previously well-defined entities, and becomes tangled within them. This has a costly impact – whether it is at the level of processes or software – and ultimately compromises the structural integrity of the system. Conceptually, this scattering and tangling can be viewed from the perspective shown in Figure 1, where the boxes are an abstract representation of either a business process or a software module, depending on the domain, and the dark lines within the boxes represent the scattered and tangled integration issues within these legacy entities.
A more cost effective way to introduce integration is depicted in Figure 2. Here, the modularity of the legacy system is preserved. The integration issue remains modular, and the legacy entities are still in tact.

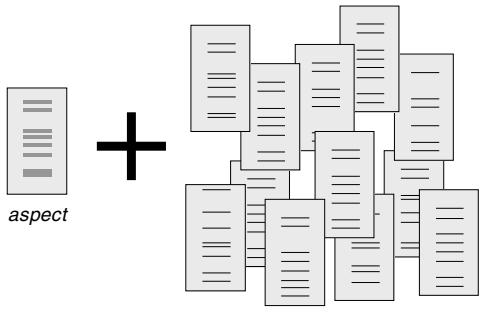

Figure 2 – Modular introduction of an integration concern.

Aspect-Oriented Software Development[1] (AOSD) is focused on novel notions of modularity that crosscut traditional abstraction boundaries. AOSD techniques and tools, applied at all stages of the software lifecycle, are changing the way software is developed in a wide spectrum of application domains, ranging from embedded systems to enterprise IT. This paper outlines the ways in which we believe aspects could aid the integration and evolution of both business processes and software used to support modern healthcare practices across this spectrum, with examples taken from our experience to-date working with aspects in software. We believe the key principle of AOSD – the modularization of crosscutting concerns – to be an integral part of the solution to the challenges currently facing integration and evolution of modern health service infrastructures.

As Figure 2 eludes, the key difference between an aspect-oriented techniques over traditional approaches to modularity is that aspects completely specify not only the concern they modularize (in this case, integration), but also explicitly how that concern interacts with the entities it crosscuts. In software, we understand how to do this in terms of well-defined points in the execution of a program. These points are well defined in terms of the interfaces that are exposed in the system. In business processes we believe this same structure would hold, but we leave it to the experts in the area to define exactly what this may mean in the context of healthcare processes[2]. We believe that process modeling, represented by Petri-net based workflow nets [1], may be a good place to start with further investigation of aspects in this domain.

This paper is organized as follows. First we take from our experience in embedded systems, outlining the technological challenges of viewing a patient as a source of data and the need for AOSD in product lines of this very lowest level of software, the micro-controller (Section 2). We then explore these same principles as they apply to challenges facing configurable communication protocols (Section 3), in these systems where memory, computation and power consumption. Finally, we consider the alternative (or perhaps synergistic) application of AOSD to the challenges of filtering data at the application level (Section 4).

2 Embedded SYSTEMS Level

2.1 The Patient at Home as a Source of Data

Clearly, the most important source of data for clinical services and medical processes in the healthcare system is the patient. Modern diagnostic technologies lead to an increasing amount of electronic data to be collected whenever a patient visits a physician for examination. While most patients see their physician rarely in the first decades of their life, for many the time spent increases exponentially with their age often resulting in necessary hospitalization for constant observation.

[1] www.aosd.net

[2] We are not healthcare professionals, we are computer scientists.

One of the biggest challenges facing our aging society is to find ways to allow seniors to *age in place* and maintain independence as long as possible. There are significant societal and financial benefits to supporting independent living however, there is an associated risk factor. Episodes of confusion and disorientation, undetected heart attacks, and the danger of accidents can put a great deal of stress on involved individuals. From this, there is growing interest in technology that provides constant observation in the home, combined with cognitive systems for a reliable detection of emergency situations.

To support this independent, self-sustaining, yet safe living environment, recording and interpretation of the patient's vital statistics is necessary. A fundamental set of sensors to provide this data, coupled with micro-controllers and transceivers for data integration are typically attached directly to the patient's body. In this scenario these devices constitute the *Patient's Personal Body Network (PPBN)* also known as the *Body Area Network* (BAN).

A constantly worn and working PPBN is the basis for two major functions of the overall system: 1) detection of short-term body function anomalies. 2) providing physicians with long-term data records about the patient's vital statistics.

Short-term anomalies typically indicate an

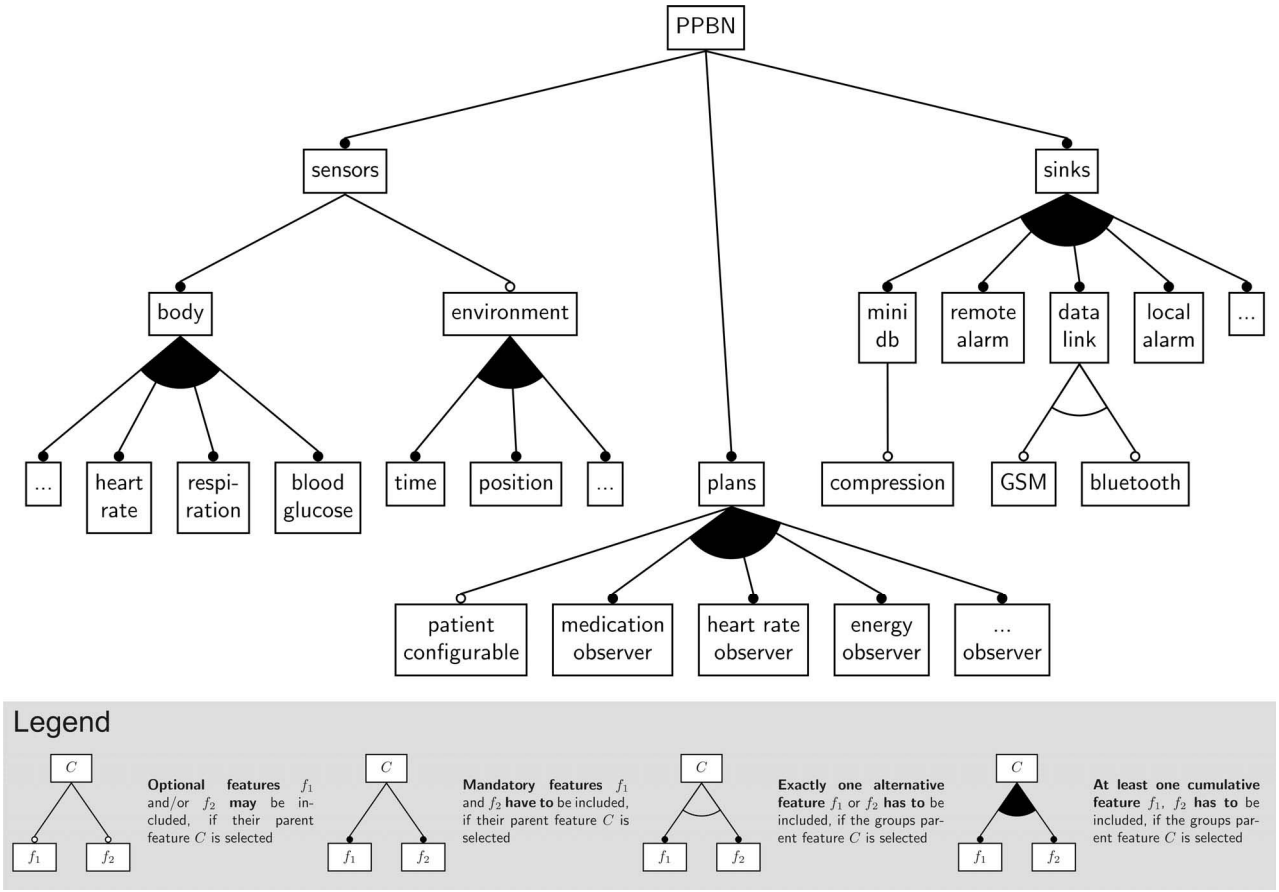

Figure 3 - Example Feature Diagram for a PPBN.
A PPBN consists of at least one *sensor* one *plan* and one *sink*. Sensors acquire information about the patient's body functions (such as *heart rate* or *blood glucose* level) and, optionally, their current environment (such as the current *position* in GPS coordinates). Data acquirement is controlled and integrated by one or more *plans* and, depending on the plan, processed into one or more *sinks* (such as an *alarm* sink to indicate an emergency situation or a *data link* to transfer data to an external device.) Plans observe and react on medically relevant data (such as a *heart rate observer*) as well as on cross-cutting system-eminent state (such as the still available *energy*).

emergency situation that requires some kind of intervention. Depending on the grade of the anomaly, intervention may range from reminding the patient to take some medication up to an automated emergency call for external help.

Long-term data records from PPBNs could be a valuable data source for physicians to detect creeping changes in the aetiopathology of the patient. The integration of this data into the clinical process could allow physicians to optimize medication and treatment.

2.2 Challenges

While the functionality provided by the higher layers of an integrated healthcare system is typically implemented using *general-purpose* commodity hardware technology (e.g. PCs and PDAs), the PPBN clearly is a *special-purpose* system. Patients would be required to wear the PPBN at all times, and therefore it must not hinder them in their daily living. Hence, PPBN components are *deeply embedded systems*, allowing the possibility of implantation. They have to be small, light, and robust. This domain strictly limits resources in terms of memory, computation power, and energy.

The functional requirements on the PPBN, on the other hand, vary widely between patients. The actual body functions to observe as well as the observation frequency and integration algorithms to must be fine-tuned for every patient. As it is simply impossible to find a one-fits-all solution that adheres to hardware and system resource requirements, the PPBN has to be tailored to fulfill exactly the requirements of the specific patient, but nothing more. This leads to a high, but deliberate level of heterogeneity on both, the hardware and the software side. There is a demand for strategies and implementation techniques to systematically deal with this variety in healthcare processes in a current state and future states. Current states, in the sense that the data gathered by PPBNs has to be made available for the physician in charge and well integrated with other available data about the patient's current state; and future states in the sense that physicians must be able to reconfigure a patient's PPBN according to changing requirements.

2.3 Embedded Software Product Lines

There is no one-fits-all recipe for building a software system that fulfills the requirements of all potential applications, while adhering to system resource requirements. The solution is therefore to tailor the PPBN to provide exactly the functionality required. This leads to a *family-based* or *software product-line (SPL)* approach. SPL is an effective approach to increase reuse and quality of software and decrease development time and cost, by sharing architecture and a set of reusable components. In the embedded systems domain, the SPL approach is used particularly for configurable system software, especially operating systems. Well known examples are eCos [7], PURE [3], and TinyOS [14].

By consequently following the SPL-based approach of software development, highly customizable PPBNs are feasible. Variant building, however, is only a first step in the development process. Without being able to organize and manage the many possible variants of the software family in an adequate and user-friendly manner, this approach will be doomed to failure. *Feature modeling* appears to be a promising way to tackle the variability management problem. This technique is understood as "the activity of modeling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model referred to as a feature model" [5]. The goal is to come up with directives for the structural design of a system that meets the requirements and constraints specified by the features. Common is a graphical representation of the feature model in terms of a *feature diagram*. The diagram is of a tree-like structure, with the nodes referring to specific feature categories. Four feature categories are defined: *mandatory*, *optional*, *alternative*, and *or*. A feature diagram describes the options and constraints that shall exist within a system. It models the variable and fixed properties of a family of programs, which implement that system. The feature diagram shown in Figure 4 illustrates the model of the envisioned PPBN.

2.4 The role of AOP in SPL Development

Software product lines are prone to evolution due to the emergence of new requirements on the products in the family. The evolution could be a continuous change, which happens with the maturity of the technology and involves an incremental adoption approach, or it may be radical and force system-wide changes at once. The evolution could be the restructuring or replacement of a feature or multiple features or constantly raising the level of services etc. Moreover, the evolution could affect only a particular module or a number of different modules. When the evolution of a single, related concern requires changes

to multiple modules, it is said to be crosscutting and hence, non-trivial to localize. Often, the unanticipated changes affect multiple modules and result in code tangling, limiting levels of evolvability, variability and granularity and negatively affecting the quality of the SPL. With traditional decomposition approaches, it is difficult to localize and encapsulate these crosscutting concerns. Thus a crucial point in the SPL approach is the mapping of all selectable and configurable features to their corresponding, well-encapsulated implementation modules. Some so called orthogonal features like security, mobility, monitoring, real-time constraints, profiling, or energy management, are typically reflected in many points of the software code. This crosscutting character restricts implementation of such orthogonal features as independent encapsulated entities and thereby limits variability, granularity and evolvability.

AOSD provides mechanisms to encapsulate crosscutting concerns into isolated entities called *aspects*, by specifying the action that will occur at specified points in the system. This control of execution can be applied either statically or at runtime. The encapsulation of crosscutting concerns provided by AOSD allows for the evolution of these concerns in isolation. Hence, evolution is limited only to addition, removal or modification of the concern without affecting the rest of the application. A well-directed application of AOSD principles in the development of software product lines can lead to a higher variability, evolvability and granularity of the selectable system features, as their implementations can not only be encapsulated by classes, but also by aspects. By employing AOSD techniques, the evolution in SPL is principally feature-driven as it is confined to the deployment or the removal of features either statically or at runtime. A well-directed application of AOSD principles in the development of software product lines can therefore enable the development and evolution of optimally patient-tailored, yet resource-thrifty, PPBN systems.

3 Configurable protocols

3.1 Coalescing Patient Data

TinyOS [14] is an operating system (OS) with a modular and communication based design to support distributed data collection, and other requirements specific to wireless sensor networks. This system software may well be the OS of choice for the collection of patient data in commodity wireless sensors.

Development in this embedded systems domain is constrained by memory footprint and power consumption and motivates the component based design of this scaled back OS. The typical 'mote' platform for TinyOS is 10KB of RAM and 100KB of ROM. The event driven design allows for the system resource allocation to be based on the components required by the event allowing for fine-tuning of power consumption between 10uA and 25mA. These events can have a cascading effect, where one event in turn triggers another. This cascading effect introduces difficulty when introducing change to a system, requiring a developer to understand and possibly cascade changes across the system structure.

Aggregate data collection is identified as one of the most widely used services of wireless sensor networks and one of the key services in workflow scenarios of electronic healthcare. Many low-level details are considered in this seemingly small part of a workflow scenario. TinyOS supports this service, considering factors including, but not limited to the number of devices involved, the paths available for data transfer (routing) with complex routing algorithms dependant on system configuration.

Data collection is considered a non-functional requirement along with flow control, error control and security that are introduced in a networking environment such as this. These non-functional requirements within a networking environment introduce complexity with low-level mechanisms of support, overflowing the structural boundaries established by the functional requirements of the system.

Typical ISO and TCP/IP network protocols adhere to a simple hierarchical layering where each layer encapsulates a group of related functions. A CP framework provides a requirement-driven, customizable set of protocols comprised of function specific modules. A specific protocol stack is then just a configuration of the required modules in each protocol layer. The x-kernel [9] is an example of a configurable protocol that follows this type of standard hierarchical structure. Although these CP frameworks are arguably more configurable and extensible than their fixed counterparts, TCP/IP and ISO, they still do not support configuration and encapsulation of fine-grained policy that cuts across multiple layers in the hierarchy. Cactus [8] adheres to the traditional layered architecture in its composition of protocols or services, where each service is composed of multiple

micro-protocols, which are linked to specific events, dynamically triggering a corresponding event handler.

Networking protocols have typically followed a layered, hierarchical architecture imposing strict interfaces at each level. This structure allows the developer to consider each layer in isolation, where an individual layer provides certain services to exactly one layer above. This design provides good separation of concerns, reducing the complexity of the system by shielding the upper layers from the implementation details of those below. These characteristics associated with separation and locality not only aid in the initial development process, but also support future evolution and modification of the system and lends the architecture to reuse.

3.2 Challenges

This hierarchical structure of network protocols often results in requirements associated with low-level mechanisms overflowing boundaries. Non-functional requirements, such as data collection, flow control, error control and security fall into this category. The fine-grained implementation of these low-level mechanisms is thus scattered across multiple layers in the system. By this definition we can say that non-functional requirement *crosscut* the architecture. Regardless of the dominant decomposition of the system architecture, there tend to be requirements that do not fit cleanly into that hierarchical structure and are therefore considered to have an inherently crosscutting structure.

Configurable protocol (CP) frameworks, such as Cactus have been developed to provide a modular representation of non-functional requirements in network architectures. These non-functional requirements in this form, known as *micro-protocols*, better support reusability, configurability, extensibility, and evolvability. Healthcare systems require a high degree of configurability and flexibility across multiple platforms under tight constraints in terms of memory footprint, computational resources and power.

The challenge in a *micro-protocol* implementation is in providing a clear separation of these concerns that do not fall into the dominant structure of the system. The number of potential configurations in addition to the fine-grained nature of their implementation makes these micro-protocols the grounds for a particularly complex system.

The implementation of Cactus is restricted by traditional approaches to capturing modular design and is limited in the following three ways:

- The ability to provide a clear structural view of the relationship between a given micro-protocol, the events affecting it and the strategies to handle those events.
- The ability for developers to customize the events and event handlers within these concrete libraries.
- The ability to provide a clear structural view of the interactions between micro-protocols.

3.3 Solutions

As a small proof of concept example [3], Hiltunen describes the concept of *message stability* in a network service used by many micro-protocols as a *trigger* for related action. How message stability is defined is dependant on multiple factors including general network architecture, the micro-protocol affected and the specific configuration of the micro-protocol.

This paper builds on this proof of concept, considering two implementations of a multicast architecture. The first version defines message stability using a global flag and the second using acknowledgements. The significance of this variation in ubiquitous healthcare systems is directly related to the issue of resource constraints. This state of stability triggers related action that can vary depending on the configuration of the micro-protocol. This example considers both an archiving and a garbage collecting micro-protocol for each version, where archiving would consume more memory and garbage collection would free this constrained resource.

Figure 4 highlights these characteristics in terms of two possible implementations of a multicast service (denoted *V1* and *V2* respectively). In the first version (*V1*), the trigger is a global flag, and the version can be configured to use either archival or garbage collection actions. Similarly, the second version (*V2*) can be configured according to these same two actions, but the trigger is the act of acknowledgment. These versions, V1 and V2, will form the basis for the design and implementation of the configuration aspects that will follow.

service	**Multicast V1**	**Multicast V2**
triggers	global flag	acknowledgement
action	archive / garbage	archive / garbage

Figure 4 - stability micro-protocol characteristics

The trigger factor for each micro-protocol can be defined as the state of message stability. The difference thus lies in the definition of what marks entry to that state. Much like programming to an interface, introducing an *abstract aspect* will force any *subaspect* to concretely define the principled points of execution associated with entry to a stable state.

Figure 5 shows the abstract aspect `MulticastStability` with the single abstract *pointcut*, `msgStability` that will force inherited classes to provide a concrete implementation specifying the principled execution points in the system associated with message stability.

```
public abstract aspect MulticastStability {

   abstract pointcut
       msgStability( Msg m, Client c );
}
```

Figure 5 - Abstract aspect for stability microprotocol.

Figure 6 shows a concrete implementation of the abstract aspect `MulticastStability` for multicast system *V1*, where entry to a stable state is triggered with a global flag. The concrete named *pointcut*, `msgStability`, specifies the execution points for this particular implementation to be any place where the stability flag is set and a stable state is entered. The message and the host as parameters are introduced as parameters within the pointcut, giving access to the instances of those objects. The aspect builds on this named pointcut, specifying the action that will take place upon entry to this stable state. In this case, the advice is archiving the instance of the message once the stable state is reached. The *after* keyword explicitly forces this advice to be triggered only upon entry to the stable state.

```
public aspect microProtocolV1 extends
                MulticastStabilityAspect {

   Archive a = new Archive();

   //joinpoint
   pointcut msgStability(Msg m, Host h):
          set(boolean Host.STABILITY_FLAG())
         && this(m) && target(h);

   //advice
   after(Msg m, Client c):
       msgStability(m, c) {

          //add the message to archive
          a.add(m);
  }
}
```

Figure 6 - Concrete implementation of stability microprotocol for MulticastV1.

Similar to this implementation, multiple configurations of the micro-protocol can be created by simply introducing new concrete aspects and fine-tuning of the pointcut and advice to match the specifications. For example, *V2* would define `msgStability` to occur after all acknowledgements have been sent out (albeit a more complicated pointcut, but still possible considering the high likelihood of an *ack* method for the advice to monitor).

This small proof of concept has shown that in applying AOSD to the domain of CPs is not only possible, but also can clarify the structure as suggested by Hiltunen. Further, this proof of concept indicates that these results are not limited to a single micro-protocol, but might also extend to clarify interaction between existing micro-protocols. The relationship between a given micro-protocol, the events that affect it and the action that they take is localized to one modular unit, allowing the code to emulate the design. In this form, developers can more effectively reason about the relationship between micro-protocols, as their internal structure is better separated, their external interaction is made explicit, and their functionality is exposed in context (within the aspect).

The localized linguistic support provided in the AOSD implementation not only provides developers with structural clarity, but also provides access to their configuration to allow fine-grained customizations to be applied.

This small study suggests that an AOSD approach alleviates the restrictions identified in the Cactus approach in the following three ways:

- Provides a clear structural view of the relationship between a given micro-protocol, the events affecting it and the strategies to handle those events.
- Allows for fine-grained customization of a given micro-protocol configuration.
- Provides a clear structural view of the interactions between multiple micro-protocols.

There are several tradeoffs to consider with this approach. CPs inherently require a fine-grained implementation, which in turn requires fine-grained pointcuts in an AO implementation. Hiltunen suggests that current mechanisms for semantic pointcuts may be sufficient, but studies have shown the contrary. In [13], Siadat *et al* provide results that suggest the amount of refactoring required in their case study to expose sufficient required execution points was intolerable.

This refactoring required maybe alleviated with an AO language with stronger semantic support, but this

also may introduce the issue of performance. For example, even the more semantic based support provided by AspectJ has been shown to incur extra performance. In systems in general, and more specifically network architectures; the performance overhead of an implementation is a serious consideration.

Another consideration in this domain is the varying architectures to consider. CPs encompass the area of embedded devices and also real-time systems, widely used in health care. Specifically, with portable hand-held embedded devices memory footprint is an issue. Currently AO implementations are known to have a significant increase in memory footprint, making their use limited in this area. Resource constraints become an even greater consideration in real-time systems, where deterministic runtimes are necessary for ensuring deadlines are met.

4 Customizable Enterprise IT

4.1 Dynamic Monitor Needs

Traditional system diagnostic and optimization techniques in enterprise solutions to application level software rely on static system structure and static instrumentation. But static approaches are simply no longer sustainable in the evolution of complex, distributed and dynamic systems. This is particularly true in healthcare environments, where monitoring associated with patient status information may change rapidly.

4.2 Challenges

Heterogeneity and predefined abstraction boundaries are actually obstacles to evolution of the system – layering, componentization, and virtualization provide necessary levers for abstraction, but emergent behaviour ultimately impairs the efficacy of local reasoning. Understanding and evolving system behavior thus requires approaches that can flow freely across boundaries and provide comprehensive analysis that can be easily collected, correlated, and subsequently used to adapt applications dynamically, as they are executing. Looking at this problem from another angle, complex system architectures must be designed and documented from the perspective of multiple views for different stakeholders [6]. Furthermore, views may need to be iteratively refined, as focus changes during the process of analyzing interests [4]. Ideally, infrastructure to support views should be able to be easily removed once users no longer need them, and incur little to no performance penalty. Recent technologies such as those employed by JFluid [10] go a long way to demonstrating that dynamic bytecode instrumentation can be both customized and efficient.

4.3 Solution

For a large class of optimization strategies related to unanticipated external environment conditions, optimizations are becoming an increasingly important obstacle to effective evolution. Mixing optimization logic with application logic requires non-local information and makes both of them more difficult to understand, maintain, and evolve, due to the idiosyncratic dependencies on external factors. Optimization code is context dependent and highly sensitive to dynamic factors such as server load, network traffic, and even order of operation completion. These factors make it particularly inefficient to encode certain kinds of optimizations in the absence of a priori knowledge about execution contexts.

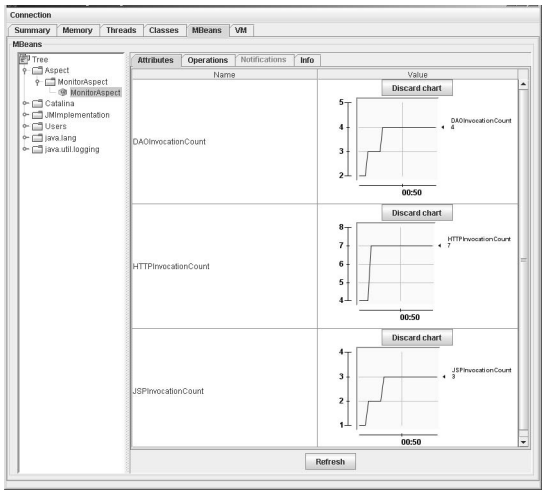

Figure 7 - Data from *MonitorAspect*.

SONAR (Sustainable Optimization and Navigation with Aspects at Runtime) is our current prototype for a fluid and unified framework that allows stakeholders to dynamically explore and adapt meaningful entities that are otherwise spread across predefined abstraction boundaries. This allows for a safe and sound approach to system evolution. Through a combination of dynamic Aspect-Oriented Programming (AOP), Extensible Markup Language (XML), and Java Management Extensions (JMX), SONAR can comprehensively coalesce scattered artifacts, enabling iterative and interactive system-wide investigation and

evolution. SONAR allows stakeholders to easily shift focus between coarser/finer grained, or even crosscutting entities, and presents system diagnostics in a comprehensive, manageable unit.

JMX is used as a means to comprehensively visualize and manage aspects introduced by SONAR. This includes retrieving data from aspects, invoking operations, and receiving event notification. Through JMX, the aspects can be managed by JMX-compatible tools remotely and/or locally. The tool we used is called JConsole which is a JMX-compliant graphical tool for monitoring and management built into Sun's JDK distribution shown in Figure 7.

This figure illustrates how the statistics from three different invocation points collected by *MonitorAspect* can be visualized as line charts in Jconsole and managed in SONAR.

Tracing. In order to demonstrate SONAR's ability to freely navigate across abstraction boundaries in the system, we developed a *TraceAspect*. This aspect is not enabled (i.e., its deployment strategy is *manual*) when the system is started. The stakeholder can thus enable the tracing through domain independent AOP (deploy/undeploy).

The *TraceAspect* reflects a request-centric view of the system, recording key data points as requests are serviced. As a result, it exposes several key configurable options and operations through JConsole, such as the ability to:

- enable/disable tracing to specified classes and/or methods
- apply a filter, to exclude unwanted data
- configure tracing details (timestamps, duration)
- manipulate buffer operations (change the size, clear the buffer) in resource constrained devices

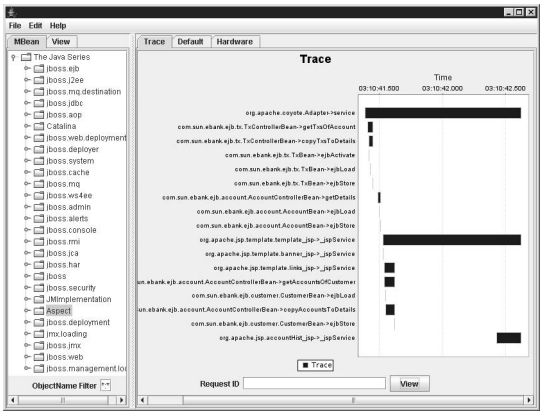

Figure 8 – Serving a request stack trace.

All the above options and operations are accessible through this aspect's JMX interface. All data is stored at the server side and can be retrieved and viewed through JMX management tool.

Figure 8 visually depicts the information collected by SONAR regarding requests to retrieve customer accounts. Each bar indicates the processing time (in milliseconds) of a method, the summation of the time spent in processing its method body and subsequent method calls. The top level is the entry point of serving an HTTP request. Access to SessionBeans, EntityBeans and JSPs are traced to clearly show the processing time in each software layer according to J2EE architecture.

Cross-platform Optimization. We chose *SpringAir*, a web application built upon Spring and the .Net framework, as our second case study to demonstrate cross-platform support for optimization. The basic idea is to retrieve some cached data from remote systems and build a local copy in order to save the time spent on network communication. Optimizations that improve locality can dramatically impact the evolvability of many of today's web-service based applications. These kinds of performance bottlenecks could thus be avoided in the context of time sensitive patient care information retrieval scenarios. The important evidence this work suggests is the efficacy of aspects in the role of high-level system integration and beyond – into the realm of optimization.

Our plans are to investigate how to unify SONAR with heavyweight, low level tool kits in order to provide an efficient means of truly integrating tasks crossing all layers in the software stack, beyond middleware and applications shown here, and down into the protocol and OS layers, as discussed in Sections 2 and 3. We also plan to further investigate a high level language specifically for optimization and navigation into SONAR, based on what we have established so far in terms of semantic representation of system behaviour. We believe this language would require higher-level representation of aspect compositions in order to provide comprehensive management. For example, when some combination of optimizations may actually interfere with each other, or to determine when the lifetime of an optimization has essentially expired due to changes in the external environment. We further believe that this kind of language could be adapted to be semantically viable at the level of aspects associated with the integration of business processes. Process modeling tools such as Protos [12] may give us some insights as to what these languages may require.

5 Conclusion

Healthcare devices and features they provide are the core building blocks of electronic healthcare system with the number of features growing rapidly. These clearly defined blocks provide a modular approach to the healthcare system, encapsulating each concern and facilitating change, maintenance and reuse. But the responsibility of integration and coordination of these devices cuts across the system and does not fall cleanly and separately into the legacy structure. This crosscutting structure is difficult to modularize with current approaches, but it is exactly the way in which AOSD looks to promote and support modular reasoning and implementations. AOSD cuts across the core hierarchical structural boundaries by specifying the uniform integration points across each feature.

Evolution of legacy entities in general is hard because traditional structural organization is not agile enough to change both statically and dynamically without breaking. AOSD supports this concept by providing a centralized locus of control of the explicitly defined elements of interaction.

The concept of modularity extends beyond the software design and implementation and has effects in business, society and multiple facets of human organization. Baldwin *et al.* identify the role of modularity in business organization and product development [2]. Baldwin goes on to recognize that this modularity does not necessarily align with the hierarchical organization structure we are accustomed to:

> "We would also say: do not be dogmatic about product and process boundaries. A process can be a module, and, if it is, the process can be a product. In fact, product definitions are endogenous in a modular system."

The examples provided in this paper, at three levels of integration, illustrate the ways in which the consideration and management of crosscutting structure in a modular way is beneficial. We believe, the consideration of crosscutting modularity at the level of business process, and ultimately the design and implementation of these complex healthcare systems is necessary to alleviate the cyclic relationship between evolving business processes and evolving technology. Specifically, as the opportunity to change interaction strategies is supported by the structural integrity of aspect-oriented development we can provide more configurable business processes, design and systems.

6 References

[1] W.P.M.v.d. Aalst and Hee, K.M.v. 2002. Workflow Management: Models, Methods, and Systems. Cambridge: MIT Press.

[2] Carliss Y. Baldwin and Kim B. Clark (2000) *Design Rules, Volume 1, The Power of Modularity*, MIT Press, Cambridge MA.

[3] Danilo Beuche, Antonio Augusto Frohlich, Reinhard Meyer, Holger Papajewski, Friedrich Schon, Wolfgang Schroder-Preikschat and Olaf Spinczyk and Ute Spinczyk (2000) On Architecture Transparency in Operating Systems. In *sigopsew00*, acm, Kolding Denmark.

[4] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord and Judith Stafford, *Documenting Software Architectures: Views and Beyond*, ISBN: 0201703726, 2002.

[5] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger and Ulrich W. Eisenecker (2002) Generative Programming for Embedded Software: An Industrial Experience Report. In *gpce02*, Springer.

[6] DTrace, *Solaris Dynamic Tracing Guide*, http://www.sun.com/bigadmin/content/dtrace/

[7] eCos, http://ecos.sourceware.org.

[8] Matti Hiltunen, Fancois Taiani, Richard Schlichting. Reflections on Aspects and Configurable Protocols, In *Proc. International Conference on Aspect-Oriented Software Development* (AOSD), 2006.

[9] Norm Hutchison, L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. On Software Engineering*, 1991.

[10] JFluid, http://profiler beans.org/index.html.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier and John Irwin, Aspect-Oriented Programming, European Conference on Object-Oriented Programming (ECOOP), 1997.

[12] Pallas Athena (1997). Protos User Manual. Plasmolen, the Netherlands: Pallas Athena BV.

[13] Jamal Siadat, Robert Walker, Cameron Kiddle. Optimization Aspects in Network Simulation. In *Proc. International Conference on Aspect-Oriented Software Development* (AOSD), 2006.

[14] TinyOS, www.tinyos.net.