Proceedings of the First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software

April 23, 2002

Held in conjunction with the First International Conference on Aspect-Oriented Software Development (AOSD 2002)

Enschede, The Netherlands

The Department of Computer Science
UNIVERSITY OF BRITISH COLUMBIA
201-2366 Main Mall
Vancouver, B.C.
V6T 1Z4

TR-2002-02

Yvonne Coady (Ed.)

First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software

April 23, 2002

A one-day workshop held in conjunction with the
First International Conference on Aspect-Oriented Software Development (AOSD 2002)
April 22-26, 2002, Enschede, The Netherlands

Aspect-oriented programming, component models, and design patterns are modern and actively evolving techniques to improving the modularization of complex software. In particular, these techniques hold great promise for the development of "systems infrastructure" software, e.g., application servers, middleware, virtual machines, compilers, operating systems, and other software that provides general services for higher-level applications. The developers of infrastructure software are currently faced with increasing demands from application programmers needing higher-level support for application development. Meeting these demands requires careful use of software modularization techniques, since infrastructural concerns are notoriously hard to modularize.

Aspects, components, and patterns provide very different means to deal with infrastructure software, but despite their differences, they have much in common. For instance, component models try to free the developer from the need to deal directly with services like security or transactions. These are primary examples of crosscutting concerns, and modularizing such concerns are the main target of aspect-oriented languages. Similarly, design patterns like Visitor and Interceptor facilitate the clean modularization of otherwise tangled concerns.

This workshop aims to provide a highly interactive forum for researchers and developers to discuss the application of and relationships between aspects, components, and patterns within modern infrastructure software. The goal is to put aspects, components, and patterns into a common reference frame and to build connections between the software engineering and systems communities.

Acknowledgments

Many thanks to Lodewijk Bergmans, the Workshop Chair at AOSD '02, Richard van de Stadt, the author of CyberChair, and Therapon Skotiniotis, for external reviewing.

Table of Contents

Programming OS Schedulers with Domain-Specific Languages and Aspects: New Approaches for OS Kernel Engineering*

Luciano Porto Barreto
COMPOSE group, http://compose.labri.fr/
INRIA/LaBRI/Enseirb, 33405 Talence Cedex, France
Luciano.Barreto@labri.fr

Rémi Douence, Gilles Muller, Mario Südholt
École des Mines de Nantes
4, rue Alfred Kastler, La chantrerie, Nantes
{douence,gmuller,sudholt}@emn.fr

Abstract

There is a continuous demand for new scheduling policies to address specific requirements of modern OSes. However, the implementation of such policies within an existing OS kernel raises many problems, mainly because optimizations within schedulers hinder code maintenance and implementation of existing schedulers is spread over the kernel.

In this paper we motivate that schedulers form an aspect within OS kernels. We show how the DSL of the Bossa system for the definition of scheduling policies and its runtime support can be integrated with a framework for Aspect-Oriented Programming, Event-based AOP. Finally, we discuss the generalization of AOP-based techniques to other OS kernel modules.

1 Introduction

Over the recent years, there has been a continuous demand for new scheduling policies to address specific requirements of modern OSes and emerging applications. Examples include policies for multimedia and real-time applications [2, 3, 6, 10, 14] and energy-based policies so as to increase the mission time of portable devices [8, 12, 11].

While the need for new scheduling policies is well recognized, their implementation within an existing OS kernel raises many problems. Based on an analysis of several OS kernels such as RT-Linux, Linux and BSD, we have identified the following difficulties in integrating a new scheduling policy:

- **Massive optimizations hinder code maintenance**.

  Because schedulers are executed very frequently, they should be highly optimized so as to not degrade overall system performance. In fact, critical parts of schedulers are often written in assembly code and meticulously structured to exploit specific features of the target architecture. As a consequence, the scheduling policy is mixed with low-level optimizations, thus obfuscating the implementation and complicating the development of new policies. Additionally, architecture-dependent optimizations that were initially valid can be useless or even degrade performance on the next of generation processors.

- **Implementation of existing schedulers is spread over the kernel**. A scheduler is often tied to multiple kernel mechanisms such as process synchronization, system calls, and device drivers. As such, it is common to find scheduling-related code spread over different parts of the kernel [13]. Only few experts are able to fully understand how the scheduler really works, even in well-documented OSes such as Linux and BSD.

These problems discourage real experimentation on OSes and restrain a wide dissemination of research results. In industry, the situation is even worse since developers have tight time-to-market constraints. Therefore, developers have little time to devote to risky tasks such as implementing a new scheduling policy.

Such an engineering nightmare calls for the use of new methodologies that can improve the implementation of OS kernels. In this paper we discuss the use of two approaches, Domain-Specific Languages (DSLs) and Aspect-Oriented Programming (AOP) that are promising in engineering operating system kernels.

A DSL is a high-level language providing constructs appropriate to a particular class of problems. The use of such a language simplifies programming, because solutions can be expressed in a way that is natural to the domain and because low-level optimiza-

*This work has been partially funded by the EU project "EasyComp" (www.easycomp.org), no. IST-1999014191

tions and domain expertise are captured in the language implementation rather than being coded explicitly by the programmer [9]. Recently, Barreto and Muller have presented Bossa, a DSL for programming schedulers [1]. This language simplifies scheduler programming and allows the verification of critical safety properties of a scheduler at compile time

Aspect-Oriented Programming [7] has recently been introduced to address problems involving code tangling, i.e. the implementation of concepts which cannot be encapsulated using a given programming paradigm and are scattered all over a program. Technically, AOP is aiming at language support for the elimination of tangling and is looking for corresponding translation techniques, commonly called code weaving. Recently, Coady *et al.* have demonstrated the benefit of aspects in OS implementation by adapting the cache behavior in the BSD file systems [4]. We discuss how Bossa's approach to scheduler definition can be integrated into an aspect-oriented approach.

The rest of the paper is organized as follows: Section 2 presents Bossa and the benefits of using a DSL for programming schedulers. Section 3 discusses that scheduling can advantageously be treated as an aspect and how this can be done. Section 4 presents a perspective: how to generalize such techniques to an AOP-based OS kernel.

2 Bossa: a DSL for programming schedulers

Bossa has been designed to address two goals: (i) to evolve the scheduler into a modular kernel component; this is achieved by reengineering the kernel around an event-based run-time system, (ii) to ease the development of scheduling policies and to make possible the verification of important safety properties that are specific to the domain of scheduling; this is achieved by the Bossa DSL. We now highlight the main characteristics of Bossa [1]. [1]

2.1 The Bossa run-time system

Evolving the scheduler into a modular component requires substantial kernel reengineering. First, code fragments related to scheduling, i.e. *scheduling points*, that are initially spread over the kernel must be carefully identified. Then, scheduling points are replaced by a corresponding event notification to the Bossa Run-Time System (RTS). For that, the RTS provides a set of events to identify scheduling points (see Table 1). There are events for signaling process creation, process termination, process blocking and unblocking. Events are organized as a hierarchy that indicates the subsystem or the driver that is the source of the event. This permits an event handler to treat either a generic case (i.e., all blocking events) or specific instances (i.e., blocking events from the `scsi disk` driver). Finally, an event handler has to be written for each event to be considered. All event handlers are centralized in a single module, i.e. the scheduling policy. As a result, changing or evolving the scheduling policy amounts to modification of only one module.

Table 1: Bossa events

Event	Generated by
`processBlock.*`	I/O calls, drivers
`processUnblock.*`	drivers, time service
`processYield`	`sched_yield()` primitive
`clockTick`	Clock interrupt handler
`processNew`	fork() sys. call: clone(), exec*()
`processEnd`	exit(), kill()
`Schedule`	Bossa run-time system

The Bossa RTS has been implemented within the Linux kernel. Initial performance evaluations show that Bossa induces an overhead of below 5%. We are currently experimenting with other schedulers such as a variant of BSD.

2.2 The Bossa DSL

The Bossa language provides high level abstractions such as process attributes, process states, process lists and events. A Bossa scheduler contains three parts: process state and attribute declaration, event handler definition, and an interface for interaction with processes managed by the scheduler. The Bossa compiler translates a Bossa scheduler into a C module that can be linked with the kernel and the RTS.

We now introduce the main features of the Bossa DSL by presenting code excerpts of the Bossa implementation of a simple priority-based scheduler.

Declarations

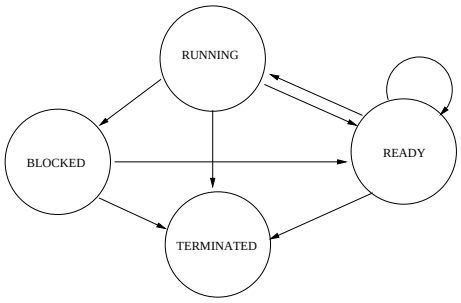
A scheduler defines a `process` type that contains attributes to support policy execution. In the priority-based scheduler, `process` only contains an integer that defines the process priority:

```
process =   {  int priority;  };
```

A process managed by a Bossa scheduler is associated with a state that describes its current activity. A process can be either in a `RUNNING`, `READY`

[1] Policy examples and a complete definition of Bossa is available at `http://compose.labri.fr/prototypes/bossa`.

or BLOCKED state. Additionally, a TERMINATED pseudo state is associated with a process that finishes. Only specific state transitions are allowed as illustrated by the following automaton:



To elect or preempt a process, it is necessary to be able to compare two processes according to a scheduling-specific relation. In Bossa, the ordering of processes is specified by the ordering_criteria declaration. The excerpt below specifies that the priority-based scheduler always selects the process with the greater priority attribute.

```
ordering_criteria = {highest priority};
```

When designing a policy, it is necessary to define support for storing processes. For that need, Bossa provides process queues and process variables. Queues and variables are always associated with a state; several queues and variables may be associated with a single state. Indeed, queues and variables refine the basic state abstraction for a specific policy.

The excerpt below specifies that there is only one process in the RUNNING state (i.e., running on the processor), that there is a list of processes associated with the READY state which is sorted according to the previous ordering_criteria, that there is a fifo list of processes associated with the BLOCKED state, and a process in TERMINATED. (Note that no storeing support is accotiated with TERMINATED.)

```
states = {
  RUNNING running : process;
  READY ready     : sorted queue;
  BLOCKED blocked : fifo queue;
  TERMINATED terminated;
};
```

Event handlers

The behavior of a scheduler is determined by the events it subscribes to. For each event, the policy must define an event handler that specifies the actions to be executed when the event occurs. An event handler basically updates process attributes and performs state changes (which are denoted by the move operator =>).

The following code excerpt specifies the behavior of the scheduler when a process unblocks. First, the scheduler moves the process for which the event was generated to the ready state. If the process that unblocks (e.target) has a greater priority than the running process using the > operator, the scheduler preempts the running process (by moving it back to the ready state.

```
On processUnblock.*  {
  e.target => ready;
  if ((!empty(running)) &&
      (e.target > running))
  {
   running => ready;
  }
}
```

Verifications and benefits

The immediate benefit of Bossa is that the programmer no longer has to deal with low-level implementation details such as manipulating pointers and lists, which may possibly crash the kernel.

Additionally, several domain-specific properties are enforced by the Bossa compiler. In particular, we are interested in properties that can avoid hazards that may lead the system into a dangerous state.

- Exactly one queue in the READY state has to be sorted by the ordering criteria of the policy ; only processes selected from this queue can be given the processor (i.e., moved to the process variable associated with RUNNING).
- When assigning a process to a variable, the variable must be empty. This constraint ensures that no process reference is lost due a wrong manipulation.
- Only transitions between process states valid w.r.t. the previously introduced automaton are accepted. By analyzing the manipulation of processes states with respect to the automaton specification, the compiler is able to detect unsafe state transitions. For example, moving a process from READY to BLOCKED is clearly incorrect.
- There is no event omission in a scheduler specification. By analyzing the specification of a process scheduler, we are able to identify whether the scheduler treats all necessary process events exported/implemented by the OS kernel.

3 Revisiting Bossa as an aspect

When analyzing the Bossa architecture, one can observe that the RTS has been designed to solve a crosscut problem. In this section we provide evidence that scheduling inherently leads to code tangling with OS

kernels structured into subsystems (e.g. synchronization, drivers). As a result, scheduler implementation could benefit from the use of aspect-oriented techniques.

3.1 Scheduling as an OS aspect

A scheduler depends on a variety of OS kernel mechanisms. Scheduling policies must refer to many different OS subsystems and their implementation is spread over a large part of the OS kernel implementation.

For instance, in priority-based schedulers, priorities may be updated in different parts of the kernel (e.g., when a process blocks waiting for I/O or periodically at a clock tick). Other schedulers may rely on additional information from OS subsystems. For example, in order to avoid starvation in high priority processes, the scheduler can implement a priority inheritance policy by assigning the priority of a high priority process that blocks in the kernel to the process that holds the resource[2]. This requires the scheduler to access the synchronization and file system subsystems.

Another important characteristic of a scheduler is to define in which situations it should preempt the running process so as to select another process. Preemption is usually performed when the running process finishes its execution (normally, killed by another process or due to an exception), voluntarily yields the CPU or is blocked in the kernel. One typical preemption point in a priority-based scheduler is the arrival of a high-priority process in the ready queue. The arrival of such a process can have different causes (e.g., immediately after the creation of a process or when a process becomes runnable due to a timer expiration). In time-sharing schedulers, preemption occurs when a process expires its CPU quantum. These different preemption points are related to different kernel mechanisms (e.g., system timers, process creation and destruction primitives).

The dissemination of scheduling-related code shown by the previous examples provides strong evidence that the scheduler crosscuts the kernel: a scheduling policy thus forms an OS aspect.

The Bossa approach to programming schedulers may seem quite different from AOP. Indeed, AOP is frequently assimilated to macro processing (e.g., "insert a method call at the beginning of every method that returns an `Int`") or reflection (e.g., "intercept calls to any method of class `Foo` in order to increment its second argument"). Bossa's approach of decoupling scheduling issues from the OS kernel by means of an event bus does not really fit either technique. In the following we motivate that it can be advantageously integrated in the so-called Event-based model of AOP [5].

[2]Note that another strategy is to terminate the low priority process that holds the resource.

3.2 Bossa and Event-based AOP

AOP frameworks should provide aspect languages for the concise definition of aspects and appropriate weaving technology. On the language level, crosscutting is one of the key notions of AOP. Crosscuts denote different program points where aspects modify the execution of the underlying program — most frequently by inserting new functionality. As motivated above, execution points where processes are created, preempted, destroyed or their priorities changed are examples of such points in OS code for scheduling purposes. Furthermore, at these points, information must be transferred between the OS subsystems and the scheduler: e.g., when a process terminates, the scheduler must be called with the identity of the terminating process.

These examples highlight an essential feature for an AOP framework: a mechanism which abstracts programs (code or executions) to points of interests, i.e., points where information is needed from and where new behavior is to be inserted.

Bossa can be integrated smoothly into an AOP framework recently proposed by Douence and Südholt, Event-based AOP [5]. EAOP uses events and relations between events for crosscut definition and (conceptually) relies on execution monitors to weave aspects into the base program. AOP's framework characteristics are materialized in EAOP as follows:

- The points of interest of a program execution are defined in terms of events emitted during program execution.
- Points of interest are denoted by patterns of events to be matched.
- Once a pattern has been matched, the base program execution is suspended, an action is executed and the base program is resumed.

EAOP is a very general yet operational model for AOP. It offers a natural abstraction in terms of events, enables the explicit definition of complex crosscuts by means of event complex patterns and accommodates very general actions. Moreover, it allows dynamic weaving: an aspect can be plugged (i.e., woven) at run-time.

Bossa can be clearly seen as an instance of this approach to AOP. Bossa's scheduling-specific events can be interpreted as EAOP events. After emission, Bossa events are stored and processed at some later execution point, which corresponds to event matching in EAOP. Finally, event processing in Bossa consists in executing actions that implement a scheduling policy.

4 Toward an AOP-based OS kernel

Conceptually, the implementation of EAOP relies on an event-based infrastructure and a monitor (for action execution). Bossa fits this implementation model since its implementation uses an event bus for event generation/notification and a scheduler module for event processing.

The Bossa DSL and its event model blend quite well with EAOP which can be seen as a systematic framework for expression of scheduling policies. We believe that AOP is more generally promising as a engineering approach for the implementation of scheduling policies.

Indeed, the techniques outlined in this paper could be applied to other kernel subsystems such as networking, disk scheduling and memory management. These subsystems rely on strategies that could be expressed using DSLs and implemented using AOP.

The resulting aspects could then serve as building blocks for a complete AOP-based kernel framework to allow the implementation of both configurable and reliable OSes. This ambitious goal offers many research opportunities. We detail here three of them.

First, each kernel subsystem should be studied on order to design a DSL to program strategies relevant to this subsystem. This work requires to study tradeoffs between expressiveness and safety. For instance, Bossa is very expressive and still supports static verification of the correctness of scheduling properties.

Second, it can be tedious to modify kernel subsystems in order to generate events. Systematic means to define and generate events should be provided. Bossa's current implementation is ad hoc: event generation has been inserted manually at the "right" places in the kernel code. A more comprehensive approach to kernel engineering could provide events related to the implementation such as "the method `foo` is called" or "the variable `bar` is assigned." Such events could be used to define higher-level events such as "the cache becomes invalid" or "a packet has been lost." It is important to note that application domains can impose constraints on the implementation. For example, in the case of scheduling instructions that implement the event bus must be carefully placed in the kernel to avoid synchronization problems.

Third, for the sake of efficiency, optimization should be studied. Partial evaluation is a good candidate. In some case, it could suppress the monitor by inlining event-related code of the monitor in the kernel.

References

[1] L. P. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *10th International Conference on Real-Time Systems (RTS'2002)*, Paris, France, March 26–28 2002. To appear.

[2] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. Move-to-rear list scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of ACM Multimedia, Seattle, Washington*, November 1997.

[3] H. Chu and K. Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems (ICMCS'99)*, Florence, Italy, June 1999.

[4] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European Software Engineering conference*, pages 88–98, Vienna, Austria, September 2001.

[5] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*. Springer Verlag, September 2001.

[6] K. Duda and D. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 261–276, December 1999.

[7] G. Kiczales, J. Lamping, A. Menhdhekar, et al. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.

[8] J. Lorch and A. Smith. Scheduling techniques for reducing processor energy use in MacOS. *Wireless Networks*, 3(5):311–324, October 1997.

[9] G. Muller, C. Consel, R. Marlet, L. P. Barreto, F. Mérillon, and L. Réveillère. Towards robust oses for appliances: A new approach based on domain-specific languages. In *ACM SIGOPS European Workshop 2000 (EW'2000)*, October 2000.

[10] J. Nieh and M. S. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 184–197, October 1997.

[11] P. Pillai and K. G. Shin. Real-Time dynamic voltage scaling for Low-Power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, pages 89–102, Banff, Canada, October 21–24 2001.

[12] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings*

of the 36th ACM/IEEE conference on Design Automation Conference (DAC'99), pages 134–139, New Orleans, USA, June 1999.

[13] D. A. Solomon. *Inside Windows NT*. Microsoft Press, 1998.

[14] D. K. Y. Yau and S. S. Lam. Adaptive rate-controlled scheduling for multimedia applications. *IEEE ACM Transactions on Networking*, 5(4):475–488, August 1997.

An Aspect-Oriented Framework for Schema Evolution in Object-Oriented Databases[1]

Robin Green
Computing Dept.,
Lancaster University
United Kingdom, LA1 4YR
+44 01524 593541
r.d.green@lancaster.ac.uk

Awais Rashid
Computing Dept.,
Lancaster University
United Kingdom, LA1 4YR
+44 01524 592647
marash@comp.lancs.ac.uk

ABSTRACT

Persistent objects in an object database need to be adapted, either by physical conversion or wrapping, when the schema is changed to fix bugs or meet new requirements. Object database schema evolution introduces a number of concerns into the system, such as adaptation rules, the choice between conversion or wrapping, and backward compatibility. Our research aims to allow strategies for addressing such concerns to be dynamically replaced or altered for an existing, running database. An early prototype evolution framework has been developed as an interpreter for a custom object-oriented language, written in AspectJ. This position paper discusses some areas where aspects have been used to separate concerns, and suggests other concerns in the framework which are likely to benefit from an aspect-oriented approach. The concerns discussed include: selective lazy evaluation, contracts, metacrosscutting, and the maintenance of custom version-specific extents.

1. INTRODUCTION

This paper discusses the use of aspect-oriented programming in a prototype framework for schema evolution in object-oriented databases. Since the framework needs to be highly configurable, for reasons outlined below, and since some of the concerns involved are crosscutting, this problem domain is a clear candidate for the use of AOP. The framework has been partially implemented in AspectJ 1.0.3. This paper first introduces the problems of schema evolution in object databases, then discusses aspects currently implemented in our schema evolution framework, and finally concludes with an examination of some other concerns that will be investigated as candidates for AOP as the implementation progresses.

2. BACKGROUND

Just as with relational databases, the schema for an existing, populated object database is subject to modification to fix mistakes or meet new requirements. Two key issues can be identified in schema evolution:

1. Existing objects need to be adapted in some way to conform to the new schema, so that they have the expected fields and methods. This can either be performed:
   - by the use of *transparent view wrappers*, which act as if they were instances of the corresponding class from the new schema;
   - or by physically converting the object into an instance of the new class [8], which entails dynamic reclassification.
2. It may be necessary for old applications to continue to access the database as if it still conformed to an older schema – that is, backward compatibility may be required.

If the schema evolution support in a particular OODBMS is not flexible enough for a desired change, the developer is forced to perform a complete "dump and reload". This entails copying all the data in the database to an intermediate location, then recreating the database with the new schema, and finally copying all the data back into the new database, making any structural and data modifications as necessary. This is a time-consuming and ad-hoc process for the developer, and could be very wasteful in terms of time and disk space – rendering it unacceptable for some systems.

However, even if the schema evolution facilities of the OODBMS allow the schema to modified in the desired manner, they may require the database to be taken offline while a full database conversion to the new schema takes place. Alterna-tively, they may allow the system to stay running, but not allow the new schema to be used until a complete background conversion of the database has taken place. On the other hand, if objects are lazily converted, this could impose an undesirable performance degradation on large database operations. Thus, it is arguable that for some decisions about schema evolution approaches – such as the decision as to whether to use immediate conversion, lazy conversion, a hybrid approach, or simulating conversion with views – no single approach serves the require-ments of all database applications in a satisfactory manner.

Another example of such a decision is whether to store multiple versions of a schema in the database – and if so, whether to store only the differences between schema versions (at some granularity) or whether to store each schema version in full.

Moreover, the most suitable evolution approaches to use for a given application may *themselves* change, as and when the application scales up or has to deal with new requirements [10].

Our research therefore involves constructing a schema evolution framework for object-oriented databases which is flexible enough to allow, not only different approaches to schema evolution to be configured, but also the approaches in use to be changed for an existing database (in some cases, at runtime). In order to make the framework easier to understand, configure, and extend, it is desirable to separate out these implementation decisions from each other and from the main bodies of the OODBMS and the runtime environment. This work is grounded in our earlier work

1 This work is supported by UK Engineering and Physical Sciences Research Council Grant GR/R08612.

Because our framework requires a versioned type system at the application programming level for evolution purposes, and since such a type system is not available in mature object-oriented languages such as Java[1], we chose to implement a new language. This has the benefit of greater flexibility for implementing features such as version conversion. The new language is called *Vejal* and – although it is XML-based for convenience reasons – borrows significantly from Java, AspectJ and Eiffel [7]. Applications – and application-specific aspects to convert between different versions of schemas – are written in Vejal. The framework itself, including generic schema evolution strategies, is currently implemented as a Vejal interpreter, written in AspectJ (considered here to be a superset of the Java language).

3. SELECTIVE LAZY EVALUATION

Lazy evaluation is a technique from functional programming, in which expressions are only evaluated when their values are required, and compound values such as lists may be evaluated gradually as needed. In a "pure" functional language (i.e. a fully referentially-transparent language) such as Haskell [4], the interpreter or compiler can transparently use lazy evaluation for any expression. However, in an imperative, object-oriented language such as Java, it would clearly be unsafe to lazily evaluate *arbitrary* expressions, since expressions could have undesired side effects if executed out-of-order.

Selective lazy evaluation may be defined as the process of deferring the evaluation of a particular programmer-specified expression until its value is needed. In our current framework, such a deferral is necessary or useful for two key purposes:

i) *Kind resolution:* Vejal types and classes (collectively known as *kinds*) are by default stored persistently in an *unresolved* form, which means that:

   – kind references within them are unversioned, and

   – parameterised kinds are stored as *templates*, rather than reduced to a collection of unparameterised kinds by parameter substitution.

   However, for performance reasons, it is essential to resolve every kind at or before the time that it is first used by the interpreter – otherwise the kind would have to be re-resolved every time it was used, which would be disastrous in loops. Moreover, rather than reading in and resolving an entire kind graph at once, it is more efficient to only resolve kinds on demand – similarly to the way in which Java virtual machines typically only load classes as needed. This is a less obvious form of lazy evaluation, which can bring significant benefits in terms of faster restart times for systems in development or systems being upgraded.

   Each Vejal database application is bound to a specific schema version which specifies precisely which class versions to use for that application. Thus, the behaviour of an application will be unaffected by whether kinds are resolved early or late.

ii) Vejal object resolution: For implementation reasons, inside the Vejal interpreter, Java representations of Vejal objects are typically cloned upon being read from disk. Again, it should not be necessary to read in and perform a deep clone of an entire Vejal object graph in order to access just one object.

Design patterns for selective lazy evaluation in imperative OO languages already exist (e.g. [9] and Virtual Proxy in [5]). However, in our evolution framework we have employed an aspect-oriented approach which we call *encapsulated reassignment* [3]. A sufficiently broad wildcarded pointcut designator is used to track, at runtime, all fields that the proxy object (also known as a *thunk*) is assigned to, as shown in this example:

```
aspect SpecificTracker {

 /* Assume the field's type will be SysTypeRef or some
subtype thereof. */

 after (ReassigningTypeRef ref, Object parent):

 set (SysTypeRef+ Object+.*)

 && args (ref)      // Right hand side of assignment

 && target (parent)

    // Object that the field being assigned to belongs to

 && !within (SpecificTracker)

     // exclude code within this aspect

  { ... }

 ...

}
```

The thunk implements all the methods that the type of the expression specifies, and forwards all appropriate messages to the actual evaluated object. However, when a message is sent to the thunk which requires it to evaluate the corresponding deferred expression, after evaluation all known references to the thunk are replaced with references to the evaluated object. This means that future access to the redirected references will be more efficient, since there will be no need for a double dispatch, or a check to see whether the deferred expression has been evaluated yet. Other references, such as local variables (which are not trackable in AspectJ 1.0.3 pointcuts), or fields not addressed by the set-tracking pointcut mentioned above, will still point to the thunk, but messages will be forwarded to the evaluated object. A more detailed description of the encapsulated reassignment approach is given in [3].

4. VERSIONING MODES AND DYNAMIC ASPECTS

One of the dimensions of configuration supported by our schema evolution framework is the versioning mode axis, which currently consists of a one-version mode, an N-version mode, and a mode to transition between them. The one-version mode is predicated on the assumption that only one schema version exists in the database, which allows a number of optimisations to be enabled. However, for any schema evolution to take place, in the current prototype the N-version mode must be entered, because schema evolution requires the existence of an old schema version and a new schema version. Hybrid modes are also planned.

1 Nor are versioned types - to our knowledge - available in *any* other existing programming language. Explicit versioning of types is distinct from, and more powerful than, versioned assemblies in C#.

The versioning modes are implemented as an aspect hierarchy, inheriting from the abstract aspect VersioningMode which contains some shared functionality. However, the bulk of the functionality in all of the versioning mode aspects is currently located in ordinary methods, rather than advice. This is because the methods involved, such as createClassRef and typeCheckAll, are invoked by callers for which their functionality is central, rather than a peripheral concern. It would be unnecessarily complex and would serve no real purpose to create artificial join points to allow the direction of invocation of these methods to be reversed by AspectJ with advices. Adopters of AOP should carefully consider whether a configurable concern really benefits from being implemented with advices rather than methods.

However, there are a few advices which are part of versioning modes, such as a "postLookup" advice which ensures that *persistent root* objects read from the database are adapted as necessary to the current schema version in use (other objects are handled by the lazy object cloning mechanism). "postLookup" is a good example of an advice which is not *by itself* crosscutting, since it only advises one method, but which still usefully separates a peripheral and *configuration-specific* concern from the core functionality of – in this case – a lookup method. However, the "postLookup" advice forms part of an aspect addressing a crosscutting concern, so it certainly qualifies as aspect-oriented programming.

Versioning-mode-specific advice always begins with a check that the versioning mode aspect to which the advice belongs is in fact enabled. This is in effect a *metacrosscutting concern* – a concern which crosscuts all the advices in an aspect. Basic metacrosscutting facilities are provided in AspectJ 1.0.3 with clauses such as **perthis** and **percflow** which can be applied to entire aspects, and which are implicitly ANDed to the pointcut designators of every advice in that aspect. However, none of these clauses strongly facilitate programmatic disabling and re-enabling of aspects – which is a crucial concern for implementing "dynamic aspects". The current alternatives are either to scatter redundant **if** statements through the advice, or to turn each advice into a stub "trampoline" into an individual aspect method, and then advise all such aspect methods using a wildcarded pointcut.

A more convenient way to enable and disable aspects would be to have an optional **when** clause in the aspect header, specifying a boolean condition that has to hold for the advice to be activated – similar to the **if** PCD, but applying to the whole aspect. (Advice that should run irrespective of whether the **when** condition is satisfied, such as system initialisation advice, could simply be moved into a static inner aspect or a separate privileged aspect.)

This would deal with one particular class of metacrosscutting concerns – enabling and disabling dynamic aspects. Other metacrosscutting concerns – such as synchronizing every advice in an aspect – might be dealt with by introducing a new primitive PCD for advice execution. It would be strictly speaking unnecessary to have a hierarchy of aspects, meta-aspects, meta-meta-aspects etc., because aspects can already operate on themselves. However, it might nevertheless be a better separation of concerns to separate base advice from meta-advice in this way.

5. CONTRACTS AS ASPECTS

In a complex software system, such as a highly configurable schema evolution framework, it is helpful to make the *intent* of code clear by abstracting away unnecessary details, and this is one of the key goals of AOP. Clarifying the intent of code and division of responsibilities in a system also supports reliability – which is very important for a piece of core system infrastructure such as a database evolution framework. A complementary approach to the same age-old intent problem is Design by Contract (DbC) [7], in which the behaviour of a class is semi-formally specified with preconditions and postconditions for methods and constructors, and a class invariant. [6] uses aspects to separate out runtime checks for preconditions, postconditions and invariants from a class, so that they can be selectively or fully disabled for performance reasons. However, there are other reasons for using aspects here. Firstly, there are simplicity and safety advantages compared to e.g. using **try...catch...finally** to implement reliable postconditions. The second reason, strict substitutability, points towards more rigourous guidelines for using AspectJ to check contracts at runtime.

Applied consistently, Design by Contract implies that a class should always be *strictly* substitutable wherever it is type-substitutable at all – in other words, if a Person variable can hold either an Employee or a Customer object, then both the Employee class and the Customer class should conform to the Person contract, as well as their own contracts. (It should be noted that this strict substitutability view of inheritance can cause problems with other uses of inheritance which are arguably still quite valid [13]; however, these problems are beyond the scope of this paper, and are touched on to some extent in [3].)

We first assume that contracts, apart from their invariants, apply to methods irrespective of whether they are called from the same class or not. (In practice, contract-checking would sometimes have to be excluded in cases where a method was called from the contract-checking aspect, in order to avoid indefinite recursion, but we ignore this here for the sake of simplicity.) Strict substitutability then implies that, for a method m on a type T with argument types {A1, A2, ...} and return type R:

i) The postcondition check should normally be implemented as an advice approximately equivalent to the following (context-yielding pointcut designators such as **this** and **args** may of course be added):

   after () **returning**: **execution** (R T+.m (A1, A2, ...))

   { ... }

   returning must be used because postconditions are not required to hold when a method exits abnormally – and it would be extremely misleading, not to mention incorrect, to ignore exceptions thrown by the method and throw a "postcondition check failed" instead!

   T+, indicating "T and all its subtypes", is used because the contract of a method on a type should apply to all its subtypes. The use of T+ ensures that erroneous code will be caught if and when it breaks the strict substitutability principle at runtime (assuming that the postconditions being checked are sufficiently detailed). Additionally, consistent use of this idiom allows postcondition checking to be implemented incrementally, because all the supertype postconditions, if any, will always be checked before a method returns control to its caller. As postconditions in Design by Contract should always be side-effect-free (though AspectJ cannot guarantee this), the order of checking should be irrelevant.

However, it is generally important for postcondition-checking advice *not* to assume the corresponding precondition. This is because strict substitutability allows preconditions to be strictly weakened in subtypes. So, for example, if a method with argument x has a precondition x>=0 && x<array.length, then strictly speaking an unsafe postcondition check such as array[x]!=null should be replaced with the safe equivalent x>=0 && x<array.length && array[x]!=null (In some cases, however, adhering to this rule would be too pedantic because of the very low likelihood of the precondition being weakened by a subclass.)

Arguably, it would be incorrect to simply substitute **call** for **execution** in the above advice, without any added restrictions. Suppose that T has a supertype S which declares a method with the same signature as m, but with a strictly weaker postcondition. Then the advice above with **call** substituted for **execution** would *not* be activated for code such as:

```
S var = new T ();
var.m (...);
```

since S, the declared type of var, is not a subtype of T.

It could be argued that this is not strictly speaking a failure to check a postcondition, but is rather a type error in the client code. If the client code wanted to guarantee that the postcondition of T.m would be fulfilled, it should have declared var to be of type T, or cast it to type T. However, this is not the case, for two reasons:

- Perhaps client code should not in general assume that a non-null value of an expression statically-typed to S will necessarily adhere to the contract of T; perhaps instead it should make that assumption explicit with a cast. However, the developer is entitled to rely upon a subtly different assumption at *all* times: namely, the universal conditional that *if* an object is of type T, then it will adhere to the contract of T.
- Similarly, if the class T fails to adhere to its contract at any time, that is unequivocally a bug, and should be detected by a postcondition check if such checking is enabled – regardless of in what manner the method was invoked. In particular, in AspectJ 1.0.3, the **execution** PCD (pointcut designator) matches method executions even when they are invoked by code outside the compilation unit, including java.lang.reflect.Method.invoke, unlike the **call** PCD.

ii) The precondition check for m should normally be implemented as something similar to:

```
before (): call (R (T || T1 || T2 ||...).m (A1, A2, ...)) { ... }
```

where {T1, T2, ...} are optional and are all those types, if any, which have *identical* preconditions for that method signature. It is *not* in general appropriate to use the unrestricted form T+. This is because in general subclasses should be allowed to make preconditions strictly weaker for methods which override or implement other methods, and such an unrestricted advice in effect states that subclasses will not do so. Furthermore, for a similar reason, it is *essential* not to use T+ here if third parties without access to the source code might subclass T in future, since it is difficult to override a call advice in AspectJ 1.0.3 without also overriding the destination of the call.

We also employ the assumption that when a message is sent to the value of an expression statically-typed to T, the relevant precondition in T should always be adhered to, irrespective of the runtime type of the value. The rationale for this assumption that precondition selection should depend on the static type of an expression is almost a mirror-image of the argument above that postcondition selection should depend on the *runtime* type of an object. In both cases, the conclusion is that the strictest relevant condition should be checked. For the precondition, that suggests using a **call** PCD, in most cases. Exceptions to this principle would be cases where a **call** PCD would not capture all calls of interest – either for implementation reasons, or because it is desired to check **super** calls, which **call** does not match in AspectJ.

6. FUTURE WORK

6.1 Dynamic Reassignment

The encapsulated reassignment approach can be seen as a special case of *dynamic reassignment* – tracking all references to an object and then switching them all to point to a different object at the same time. This is not a new idea, since it is supported by the **become** primitive in Smalltalk. However, aspect-orientation now allows adding this feature (or at least an approximation of it) to a language with no native "become" primitive or similar.

Dynamic reassignment could be useful for purposes other than lazy evaluation, such as simulating dynamic reclassification in languages which do not directly support it. (Again, this is one of the uses of the **become** primitive in Smalltalk – it can be used to extend an object with a new instance variable.) Dynamic reclassification can be (crudely) simulated with explicit proxy objects, but in some cases it might be more efficient to dispense with proxies and point directly to "real" objects, while using the dynamic reassignment approach to reclassify objects. For this to work, however, it would be essential – not merely useful as in encapsulated reassignment – for the aspect language involved to support pointcut designators referring to local variables and parameters.

However, this approach to dynamic reclassification would still be vulnerable to some of the criticisms levelled at the proxy approach, such as the well-known object identity problem: reclassification produces not the same object, as desired, but a different one – which is detectable with methods such as java.lang.System.identityHashCode().

6.2 Version-specific Extents

Extents are simply collections of all the persistent instances of a given class in a database. They make it easy to run SQL-like queries such as "Select * from Employees". However, the object data standard ODMG 3.0 [1] does not fully define, nor require, extents. Also, some OODBMSs (e.g. Ozone) do not have any explicit support for extents.

For the purpose of physically converting all persistent objects that currently belong to an older schema into instances of a corresponding class in a new schema, it would be useful to have extents specific to particular class versions to speed up the process of finding the objects that still need to be converted.

However, this performance gain needs to be balanced against the time and space costs of maintaining version-specific extents for the rest of the time.

Our preliminary investigations suggest that implementing version-specific extents in our current framework would involve a high degree of crosscutting code which could usefully be localised using aspects. As well as standard extent maintenance tasks such as deleting an object from its extent when the Database.delete method is called on it, and tracking which objects have been added to and removed from [2] the database at the end of each transaction, there is also the need to move objects between extents when they are dynamically reclassified. Typically this would be converting between class versions, but this could possibly be extended to arbitrary reclassification.

6.3 Version Conversion Aspects

In Vejal we are planning to allow the programmer to specify arbitrarily complex transformations between class versions, in the form of *version conversion aspects*. These are essentially transparent view wrappers, written by the application programmer to present e.g. a Person[1] as a Person[2], which are invoked by the runtime environment automatically whenever an object needs to be adapted to a different class version. Crucially, they work by transforming data at the field level, and do not attempt to emulate methods (and nor do they require the application-specific evolution code to emulate methods) – the "real" methods are always used from the Vejal class version required by the application. Although this means a version conversion aspect breaks the encapsulation of the destination class version, this is arguably a good trade-off, because the alternative of emulating method behaviour leaves more room for error, and converting an object between class versions often requires knowledge of implementation details. There are no language restrictions on changes that can be made between one class version and the next – in particular, methods can be added, deleted and rewritten.

Vejal version conversion aspects are intended to support either views or conversions with exactly the same aspect. Thus, the real adaptation approach in use is abstracted out. If the system is configured to use views for a particular class, the version conversion aspect will just be used as-is; if not, the runtime environment will "scan through" the aspect to physically convert the object to the new class. In either case, "hidden fields" will be used if required, to store data from previous schemas that is invisible now but may become visible upon another adaptation [8]. Thus no data is lost due to destructive conversions – unless a previous schema is itself deleted.

In this way, version conversion aspects can be specified once for each pair of source and destination class versions, independently of whether a view technique or a physical conversion technique is being used to adapt objects.

Version conversion aspects arguably meet both criteria set out in [2] for a technique to be aspect-oriented: quantification and obliviousness. However, this is not the only candidate definition of AOP – and there exist systems such as metaobject protocols which effectively offer quantification and obvliousness, but are not necessarily aspect-oriented. Also, the planned join point model for version conversion aspects is currently much simpler than that of AspectJ's: simply matching on any Vejal objects read from the database which need adapting for the current schema, and belong to particular specified class versions.

However, one way in which more powerful join point models might be useful for version conversion aspects is to select different conversions depending on the aggregation context. For example, in a schema evolution operation on an engineering database, one might wish to specify that a Pipe object should be converted to an ActivePipe object if it represents a pipe that is currently part of a physical structure, or a StockPipe if it is just a spare part. A context PCD for version conversion aspects would offer an alternative to scattering if statements around the conversion aspects for the relevant parent classes. The interpreter, compiler and/or runtime environment would be responsible for validating the type-safety of conversions.

2 If an extent uses ordinary references, it is impossible for an object in that extent to become no longer reachable (except by an explicit delete invocation). However, extents may instead use weak references, which do not prevent the garbage collection of the objects they point to.

REFERENCES

[1] Cattell, R.G.G., *ed. The Object Data Standard: ODMG 3.0.* Morgan Kaufman, 1999.

[2] Filman, R.E. and Friedman, D.P. "Aspect-Oriented Programming is Quantification and Obliviousness". *Workshop on Advanced Separation of Concerns, OOPSLA 2000,* Minneapolis, 2000.

[3] Green, R.D. and Rashid, A. "Aspect-Oriented Selective Lazy Evaluation." Submitted to *IFIP Working Conference on Generic Programming*; under review.

[4] Jones, S.P. and Hughes, J., *eds. Haskell 98: A Non-strict, Purely Functional Language. http://haskell.org/definition/*

[5] Larman, C. *Applying UML and Patterns*. PrenticeHall, 1998.

[6] Lippert, M. and Lopez, C.V. "A Study on Exception Detection and Handling Using Aspect-Oriented Programming". Xeroc PARC Technical Report P9910229CSL-99-1, Dec. 1999.

[7] Meyer, B. *Object-Oriented Software Construction , 2nd. ed.* PrenticeHall, 1997.

[8] Monk, S. *A Model for Schema Evolution in OO Database Systems.* PhD, Lancaster University, 1993.

[9] Nguyen, D. and Wong, S. Design Patterns for Lazy Evaluation. *Proceedings of the 31st Technical Symposium on Computer Science Education* (*SIGCSE '00*), ACM, pp.21-25. 2000.

[10] Rashid, A., Sawyer, P. and Pulvermueller, E. "A Flexible Approach for Instance Adaptation during Class Versioning". *ECOOP 2000 Symposium on Objects and Databases,* pp.101-113, Springer-Verlag LNCS 1944, 2000.

[11] Rashid, A. and Sawyer, P. "Aspect-Orientation and Database Systems: An Effective Customisation Approach". *IEE Proceedings - Software*, **148**(5), pp.156-164, IEE 2001.

[12] Rashid, A. "A Hybrid Approach to Separation of Concerns: The Story of SADES." *3rd International Conference on Meta-Level Architectures and Separation of Concerns* , pp.231-249, Springer-Verlag LNCS 2192, 2001.

[13] Taivalsaari, A. "On the Notion of Inheritance". *ACM Computing Surveys* **28**(3), 1996.

Non-intrusive constraint solver enhancements*

Rémi Douence and Narendra Jussien
École des Mines de Nantes
La Chantrerie – 4, rue Alfred Kastler
BP 20722
F-44307 Nantes Cedex 3, France
{douence,jussien}@emn.fr

ABSTRACT

Constraint solvers are useful tools that provide solutions to very complex problems. These infrastructure software rely on simple mechanisms, however their actual implementation can be quite complex. A good knowledge of their inner mechanisms is required to introduce enhancements which crosscut basic algorithms and structures. In this paper, we advocate non-intrusive constraint solver enhancements. First, a minimal solver is introduced. Second, different enhancements are implemented with the help of aspect oriented programming.

1. INTRODUCTION

Constraint solvers are useful tools that provide solutions to very complex problems. These infrastructure software rely mainly on two simple mechanisms: variable enumeration and constraint propagation. However, modern solvers (Ilog Solver [5], Chip from Cosytec [1], gnuProlog from INRIA [2], `choco` [9]) integrate many optimizations and their actual implementations can be quite complex.

Most of these systems are monolithic: a good knowledge of their implementation is required in order to introduce enhancements. The others are libraries: a good programmer is required in order to build a solver including enhancements. In both cases, the solver enhancements crosscut the basic algorithms and structures.

In this paper, we advocate non-intrusive constraint solver enhancements. First, a minimal solver is implemented (described in section 2). Second, different enhancements (here explanation and dynamic backtracking capabilities, see section 3) of this minimal solver are implemented with the help of aspect oriented programming [8] (AOP from here on).

2. A MINIMAL SOLVER

We have implemented a minimal solver in Java: `Cacao`[1]. It only deals with discrete domains and binary constraints. It is based on two simple mechanisms: first, an enumeration heuristic chooses one possible value for a variable, and second, the propagation algorithm propagates consequences of this choice. Propagation enforces arc-consistency [10] by removing from variable domains values that will never appear in a solution (taking into account the already made choices). For instance, let us consider the following problem with three variables: $X = \{X1, X2, X3\}$, $Y = \{Y1, Y2, Y3\}$, $Z = \{Z1, Z2, Z3\}$ and two relations: $X\text{Diff}Y = \{(X1, Y2), (X1, Y3), (X2, Y1), (X2, Y3), (X3, Y1), (X3, Y2)\}$ and $Y\text{Eq}Z = \{(Y1, Z1), (Y2, Z2), (Y3, Z3)\}$.

When enumeration assigns a value to X (e.g.$X = \{X1\}$)[2], then propagation computes consequences by suppressing non compatible values: $Y = \{Y2, Y3\}$ because of X and $X\text{Diff}Y$ and $Z = \{Z2, Z3\}$ because of Y and $Y\text{Equal}Z$.

This solver is simple but not simplistic: it illustrates classical concerns in constraint programming. `Cacao` has a very basic but widely used behavior: a queue of relations (constraints) is used to propagate decisions through the constraint network.

The main loop (see Figure 1) makes decisions *i.e.* extends the current partial assignment. That process ends when a solution is obtained (no remaining unassigned variable) or if the lack of solution has been proved. If a contradiction occurs, a Java exception is thrown and caught within the loop where a function `error` is called in order to print a message and halts the solver. Notice that our minimal solver has a greedy algorithm and cannot undo bad choices: this is unlike classical constraint solver. Nevertheless, as we will see in the following such a backtracking behavior will be introduced in the minimal solver thanks to AOP.

The different notions of constraint solving appear naturally in our simple implementation. For instance, we defined the classes `Value`, `Variable`, `Relation`, `Problem`, and the methods `Relation.add(Value, Value)` to actually build a `Relation`, `Relation.revise()` to locally propagate some domain modifications, `Problem.propagate()` to perform the overall propagation of a decision. The two last methods enforce arc-consistency by suppressing values from the variables. It is explicitly handled through a `Removal` class.

3. ASPECTS FOR EXPLANATION-BASED CONSTRAINT PROGRAMMING

We now define a few aspects with the help of AspectJ[3] [7] in order to introduce *explanations* and *dynamic backtracking*

*This work has been partially funded by the EU projet *Easy-Comp* (www.easycomp.org), no. IST-1999014191

[1]All code described in this paper is available on request to the authors. We plan to make it available to the public audience on the web.

[2]Note that this assignment can be expressed as an extra relation $X\text{Assign}X1 = \{(X1, X1)\}$

[3]`aspectj.org`

```
void run() {
    // finished is true if a solution has been found
    // or no solution can be found
    boolean finished = false;
    boolean feasible = true;
    try {
        // initial propagation
        relationQueue = originalRelations.copy();
        propagate();
        while (! finished) {
        try {
            // make some new decisions
            extend();
            propagate();
        }
        catch (Exception e) {
            // handling contradiction
            error();
        }
        }
        // a solution was found
        feasible = true;
        System.out.println("A solution\n" + this);
     }
     catch (Exception e) { // error itself threw an exception
        // there is no possible solution
        finished = true;
        feasible = false;
        System.out.println("No solution");
     }
}
```

Figure 1: Main loop of Cacao

in the minimal solver presented above. These enhancements are non-invasive since the minimal solver is never modified. We focus first on explanations.

3.1 Computing explanations

Explanations computation in constraint solving can be viewed as a dynamic dependency analysis. It returns the set of variables assignments that conjunctly leads to a value suppression. For instance, in the example at the beginning of Section 2, removals of $Y1$ and $Z1$ are both explained by the assignment relation $X\text{Assign}X1$.

However, the result is generally much less trivial (propagation chains may get significantly longer, several propagation chains may interleave, ...). Explanation generation is decomposed into two steps (*i.e.* aspects): *build list of supports*[4] *for values* and *compute explanations*.

3.1.1 An aspect for storing information

First, the list of supports of a value is required in order to gather explanations of this value removal. Unfortunately our minimal solver does not maintain such a list. So, we define a new class `Support` (basically a pair relation-value) and an aspect `AspectBuildSupports`. In Figure 2, this aspect introduces a new field `supports` in the class `Value`. This extra field is initialized with an empty set of supports for each value. In the minimal solver, the method `add` of the class `Relation` extends a relation definition with a pair of values; the second value supports the first one. So, the crosscut named `inRelation` captures these method calls and the corresponding advice (at the bottom of the figure) updates the set of supports of the first value every time the method `add` is called.

At this point, we did not change a line of the minimal solver, however, with the help of AspectJ and `AspectBuildSupports`, once the problem is built, each `Value` instance contains a list of its supports.

```
class Support {
    Relation relation;
    Value value;

    Support(Relation relation, Value value) {
        this.relation = relation;
        this.value = value;
    }
}

aspect AspectBuildSupports {
    Set Value.supports = new Set(); // introductions

    // build direct access to in relation values
    pointcut inRelation(Relation relation, Value value1,
                        Value value2)
      :
        target(relation) &&
        args(value1, value2) &&
        call(Relation Relation.add(Value, Value));

    before(Relation relation, Value value1, Value value2)
      : inRelation(relation, value1, value2) {
        value1.supports.add(new Support(relation, value2));
    }
}
```

Figure 2: An aspect for supports building

3.1.2 An aspect for computing information

The second step towards explanation generation requires the introduction of an extra field `explanation` in every value as specified at the beginning of the aspect `AspectExplanation` in Figure 3. We also introduce an extra method `isDecision` in the class `Relation` in order to identify decision constraints (they are binary constraints that deal with the same variable twice). Finally, the aspect defines a crosscut named `removal` in order to detect the relation and the value involved every time a value is suppressed. In the minimal solver, when a value is removed (*i.e.* a `Removal` instance is created) the relation being revised is not known: all we know is there is a call to `Relation.revise` in progress in the control stack. So, the crosscut definition uses the AspectJ construction `cflow` (for control flow) which allows us to remember the last revised relation (*i.e.* pending call to `revise`) when a value is removed. When a value is removed, the advice add the last revised relation (if it is a decision) to the explanation of this value removal. Then, it enumerates[5] the support of this value, and it gathers the explanation of these supports removal.

At this point, we still did not change a line of the minimal solver, however every time a value is removed, its field `explanation` contains contains the set of relations that explain this suppression.

3.2 Using explanations

[4] In a constraint involving the variables X and Y the value Yi is a support for value Xj if the constraint holds when $X = \{Xj\}$ and $Y = \{Yi\}$. When a value has no support for a given constraint, it can be removed from its domain: no solution will exist if that value is assigned to its variable.

[5] In this paper, for the sake of conciseness, we use a pseudo javacode and note enumeration loops as `forall`.

```
aspect AspectExplanation {
    // introductions
    Set Value.explanation = new Set();
    boolean Relation.isDecision() {
       return var1.equals(var2);
    }
    pointcut removal(Relation r, Variable variable,
                     Value value)
      :
          cflow(target(r) && call(Removal revise()))
        && args(variable, value)
        && call(Removal.new(Variable, Value));

    before(Relation relation, Variable variable, Value value):
      removal(relation, variable, value) {
        // a set of decision constraints is now attached to
        // its value
        if (relation.isDecision())
            value.explanation.add(relation);
        // and the explanations too
        Enumeration enum = value.supports.elements();
        forall support in value.supports
            if (support.relation == relation)
                value.explanation.union(support.value
                          .explanation);
        }
    }
}
```

Figure 3: An aspect for explanation generation

We now focus on backtracking in order to generate a solution with the help of aspects.

3.2.1 An aspect for redefining parts of the solver

In the minimal solver, when a domain becomes empty, an exception is thrown and the method `error` is called in order to print an error message and stop. In order to implement backtracking, our aspect must replace the original method `error` by another one that actually undoes decisions (restore with the help of explanations suppressed values), repairs the state of the solver (in order to get a consistent state) and resumes its execution. In Figure 4, the aspect `AspectBacktrack` defines a crosscut `callError` to denote the method call to `error` and the associated advice replaces (*i.e.* keyword `around`) this method call by another one to `undoAndRepair`.

This long and complex method implements dynamic backtracking with the help of explanations as detailed in [6]. *This method could not be defined without aspects.* Indeed, it has to access and modify the state of the solver but some pieces of information are not available in the minimal solver. We list here these different pieces of information used in the `undoAndRepair` method and the corresponding aspects that make them available. These aspects definitions can be found in [3].

3.2.2 Aspects for accessing the state of the solver

First, the method `undoAndRepair` must be able to access the solver state. So, the aspect `AspectBacktrack` in Figure 4 must be *extended* with a variable `problem` initialized with the reference of the instance of the problem to be solved. This way, the `undoAndRepair` method can access for instance the `relationQueue` or call the method `propagate`.

Second, when a domain becomes empty, it is mandatory to know the last modified variable in order to study its contra-

```
aspect AspectBacktrack {
    pointcut callError():
        call(void Problem.error());

    void around() throws Exception: callError() {
        undoAndRepair();
    }

    void undoAndRepair() throws Exception {
        Set contradictionExplanation = new Set();
        forall value in problem.lastModifiedVariable.originalDomain
            contradictionExplanation.union(value.explanation);
        if (contradictionExplanation.isEmpty()) { // no solution
            throw new Exception();
        } else {
            select most recent decisionToUndo
                   from contradictionExplanation;

            // remove this constraint from the problem
            problem.originalRelations.remove(decisionToUndo);
            // remove past effects
            forall variable in problem.variables
                forall value in variable.originalDomain
                    if (value.explanation.member(decisionToUndo)) {
                        // restore the value back in the domain
                        variable.domain.add(value);
                        // empty the explanation
                        value.explanation = new Set();
                        // prepare re-propagation
                        problem.relationQueue.union(
                           problem.relations(variable));
                    }
                }
            }
            try {
                problem.propagate();
            } catch (Exception e) {
            undoAndRepair();
        }
        contradictionExplanation.remove(decisionToUndo);
        boolean decisionsStillValid = all decisions in
                                  contradictionExplanation valid;
        if (decisionsStillValid) {
            try {
                // remove a value (equivalent to a non-decision
                // relation)
                Value value = (Value)(((Pair)(
                        decisionToUndo.pairs.get())).fst);
                decisionToUndo.var1.domain.remove(value);
                // set explanation
                value.explanation = contradictionExplanation;
                // prepare propagation
                problem.relationQueue.union(
                    problem.relations(
                        problem.lastModifiedVariable));
                problem.propagate();
            } catch (Exception e) {
                undoAndRepair();
            }
        }
    }
}
```

Figure 4: An aspect for backtracking

diction explanation. In the minimal solver, the last modified variable is not known when `error` is called. So, an aspect `AspectLastModifiedVariable` must be defined in order to keep track of this variable identity.

Third, the original domain of the last modified variable must be enumerated. This extra information is introduced in the minimal solver with the help of an aspect `AspectOriginalDomain`. This aspect stores all the values added in a variable (thus defining its domain) at creation time.

Finally, for the sake of completeness (see [4]), the method `undoAndRepair` must select *the most recent* decision to undo. To this end, an aspect `AspectTimeStamp` introduces a time stamp (actually a simple number rather an actual time) in every relation at creation time.

3.3 Discussion

In this section, we presented several aspects for solver enhancements that exemplify different kinds of aspects. Aspects can be used to compute extra information (e.g., `AspectBuildSupports`), modify existing behavior (e.g., `AspectBacktrack`) or store information (e.g., `AspectOriginalDomain`). For the sake of clarity, we tried to keep these aspects as small as possible.

For instance, we split explanation computation into two aspects: `AspectBuildSupport` and `AspectExplanation`. Both advices are required to generate explanations. Actually, a third advice is needed in order to take into account the symmetric nature of relations (each relation of the problem is represented by an instance of `Relation` *and* an instance of `InverseRelation`). The complete explanation aspect gathers the three parts.

4. CONCLUSION AND FUTURE WORK

In this paper, we have demonstrated how a minimal constraint solver could be non intrusively enhanced with the help of AOP in order to implement explanations and dynamic backtracking.

The minimal solver architecture, described in Section 2, must be known by the aspect programmer. For instance, in order to build the lists of supports, he must know a relation is built by repeated calls to the method `add`. In the other hand, he does not need to know the detailed implementation of the minimal solver. For example, the propagation could use a stack instead of a queue (or the `Set` of values could be replaced by `List` of values in order to make wiser choice at enumeration time) and our aspects would still be valid. This knowledge remains to be precisely characterized in order to formally define validity of aspects for the sake of correctness and reuse. Such information could also help to compose aspects (e.g. `AspectBuildSupport` and `AspectExplanation` can be composed sequentially, because every solvers have two distinct phases: build the problem, *then* solve the problem).

Future works also include practical experiments with existing solvers. We have developed our minimal solver with no assumption about its future enhancements. Then, we have defined aspects without modifying the minimal solver. We believe existing solvers could be enhanced the same way without modification. Indeed, existing solvers do not include explanation management, but they basically implement the same main loop as `Cacao`. So, it is possible to adapt our aspects to other solvers. This should be practically studied. We should also study aspect based enhancements of more complex solvers that deal, for example, with global constraints. These studies should compare our aspect based approach with alternative solutions such as inheritance, mixin or design patterns in the same context (no assumption about future enhancements and no modification of the solver).

Finally, we plan to study how aspects could be used to declaratively express strategies in order to guide the solver. For instance, *at enumeration time, select the variable but* `V1` *with the smallest domain*, or *at propagation time, use depth first propagation when the variable* `V2` *is involved and use breadth first propagation when the relation* `R1` *is involved* could be expressed as aspects by accessing the state of the solver and redefining parts of it. We believe AOP is a very promising track in order to make constraint solvers programmable, hence more efficient.

5. REFERENCES

[1] Abderrahmane Aggoun, M. Dincbas, A. Herold, H. Simonis, and P. Van Hentenryck. The CHIP System. Technical Report TR-LP-24, ECRC, Munich, Germany, June 1987.

[2] D. Diaz and P. Codognet. The GNU prolog system and its implementation. In *ACM Symposium on Applied Computing*, Villa Olmo, Como, Italy, 2000.

[3] Rémi Douence and Narendra Jussien. Non-intrusive constraint solver enhancements. Research Report 02-2-INFO, École des Mines de Nantes, Nantes, France, 2002.

[4] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[5] Ilog. Solver reference manual version, 1993.

[6] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP*, pages 327–353, 2001.

[8] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.

[9] François Laburthe. Choco: implementing a cp kernel. In *CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, Singapore, September 2000.

[10] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

H&V Consistency Checking for Software Health Monitoring

Naghmeh Ghafari, Alexander Lau, Barry Pekilis, James Thai, Rudolph Seviora
Bell Canada Software Reliability Laboratory
University of Waterloo
Waterloo, Ontario, Canada
+1 (519) 888-4567 x2850

{nghafari, alexlau, bpekilis, jthai, seviora}@swen.uwaterloo.ca

ABSTRACT

The capability to provide an indication of the internal well-being or health of an operational software system would be very valuable in a number of situations. This paper considers a way of adding such capability to existing Java programs by taking advantage of AspectJ, an aspect-oriented programming language. It introduces an approach for detecting internal state corruption by using health indicators which perform state consistency checks. An example is presented and experience obtained from the AspectJ implementation of a number of state consistency health indicators is summarized.

1. INTRODUCTION

Experience shows that external failures of software systems are often preceded by deterioration in their internal execution state. From an operational perspective, the capability to provide an indication of the well-being or health of a system's internal state and its evolution would be very valuable. For example, external indication of a growing internal impairment to the correct operation of software (e.g. growing corruption of the internal state of key program entities) would alert system operators and provide an opportunity for corrective action to be taken before a major operational disruption occurs. *Software health monitoring* is an approach that strives for early detection of internal errors in software systems [1]. Figure 1 shows an architecture for software health monitoring. A *software health index* is computed automatically from the data collected by *software health indicators*. Each indicator designed to specifically monitor a particular facet of execution.

This paper presents an approach for detecting the extent of corruption of internal program state by the means of state consistency health indicators. The indicators are implemented in AspectJ [2]. The aim is to assess if the states of directly or indirectly communicating objects [3] are consistent with each other. To achieve this goal, state information of selected communicating objects is collected and its consistency is checked. The consistency checks considered in this paper take advantage of aspect-oriented programming to attach pieces of sensor code to existing code to collect the state information. The focus is on systems with a layered internal structure whose objects have finite state machine behaviour, and whose Java sources are available. The approach is illustrated in an example taken from an embedded real-time uni-process system.

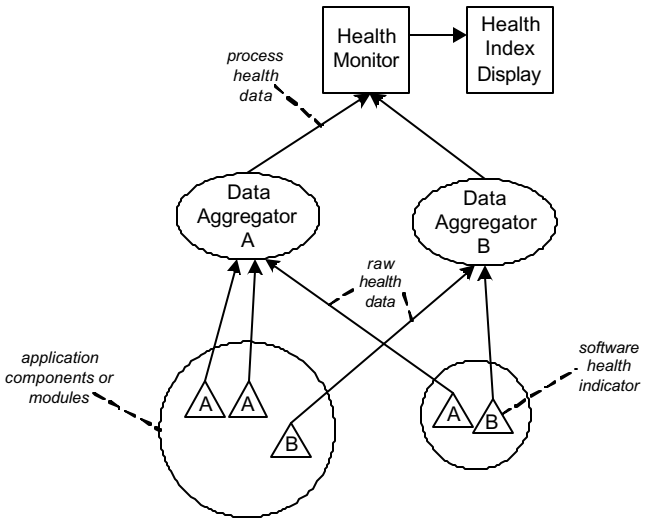


Figure 1. Software Health Monitoring Architecture

2. STATE CONSISTENCY CHECKING

The software system is assumed to have a layered design with top layer entities having behaviour described by finite state machine [4]. The lower layers present the available resources and interface with the underlying hardware (Figure 2). The design can be represented in UML or similar formalism. Each layer is built in terms of the layers below it and provides a basis for implementation of the ones above it. Classes in each layer can be independent, but there is usually some correspondence between the classes in dif-

ferent layers. A vertical client-supplier association exists between upper layers (users of services) and lower layers (providers of services). The association between the classes in the same layer is a horizontal peer-to-peer association.

An executing program moves form one global state to another under the influence of inputs that are external to the program. The program state is composed of the states of individual program objects or entities. Inconsistency between the states of these entities is indicative of the corruption of the program state. Inconsistency arises whenever the legal relationship between states of entities, as defined in the design specification, is violated. Consistency checks may not catch all forms of corruption.

In a large software system, there are many pairs of communicating objects with different consistency relationships, both direct and indirect. However, only a limited number of consistency checks can realistically be carried out. This contribution will discuss several heuristics for definition and placement of such checks.

Two major types of consistency relationships are singled out: horizontal and vertical.

Horizontal State Consistency: The objects within the same layer are often loosely coupled. The consistency between objects within the same layer is called horizontal state consistency.

Vertical State Consistency: The consistency between the states of upper layer objects with their related lower layer objects is the subject of vertical state consistency checks. Since the lower layers provide service or more concrete functionality to the higher layers, the state of lower layer objects should reflect the state of upper layer objects.

Heuristics

The heuristics for definition and placement of consistency checks suggest relative rank of particular checks. Some heuristics are primarily horizontal, while others vertical.

The main heuristic rule is based on the importance of classes. It suggests that the top layer objects and their associations have a higher rank as candidates for consistency checks. Appearing earlier on in development, they are more important to the operation of the system. The consistency checks derived from this heuristic are primarily horizontal – the horizontal associations between top layer objects are primary candidates. A similar heuristic might refer to the number of object instances of a particular class.

The importance of class associations forms another heuristic. Certain relationships may be more important or more mission critical for the functioning of the system. Related heuristics are based on the number of associations (number of methods invoked), association traversal frequency (frequency of methods invoked), association multiplicity (a large number of instances of one class associated with another class), or likelihood of failure (for each association traversal).

One heuristic primarily for vertical consistency checking is based on the presence of boundaries such as technology (software/hardware interface), network (local/remote calls), and reuse (using a component developed elsewhere). The presence of such boundaries may help in identifying entities for vertical consistency checking.

Lastly, the holding time of associations or entity states can be considered, with longer holding times targeted. The number of resources in-use in a given state is another similar heuristic, with states using a greater number of resources being better targets, since there is a positive correlation that they are performing the most amount of work.

One can also consider a consistency check over an indirect association between one end and the other of a chain separated by more than one association.

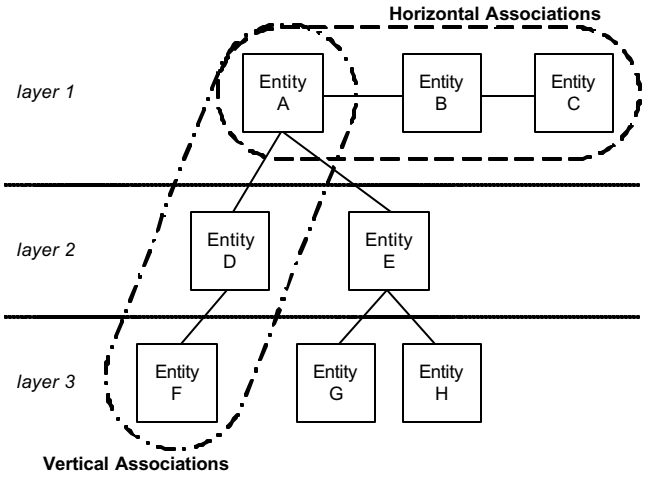


Figure 2. Horizontal and Vertical Associations

3. IMPLEMENTATION EXAMPLE

Several state consistency checks were implemented for a small, 60-phoneline telephone exchange (PBX) control program coded in Java (3000 LOC). A partial class model for the control program is shown in Figure 3. The model is layered and the top layer classes have finite state machine behaviour. The classes in the top layer were identified in the early software design phases; lower layer classes appeared later in the design cycle.

The heuristics discussed suggests a number of possible consistency checks. Three primary candidates are presented below. They focus on the top-most class – the PhoneHandler class. Other consistency checks were implemented but are not presented here, [5]. See Table 1 for the PhoneConsistencyAspect.java aspect.

First considering horizontal state consistency, the PhoneHandler class has a peer association with itself. The first consistency

check determines whether during a phone conversation, the two parties are associated with each other.

Second, the vertical traversal frequency of the association between the PhoneHandler class and the TouchToneReceiver class is comparatively high. The second consistency check determines whether the states of the two class instances are consistent with this association.

The third example is based on the association between the PhoneHandler class and the switching network. It is a vertical relationship crossing through the hardware boundary. This consistency check determines whether the correct tone generator is applied to the PhoneHandler.

The three consistency checks were implemented within a single aspect (120 LOC). (Other consistency checks were implemented separately.) Advice was attached to the enterState() method of the PhoneHandler class, which, after successfully entering a state, would perform the three consistency checks. For simplicity, the discovery of inconsistency is reported via an error message and a RuntimeException. JDK v1.3.1 SE and AspectJ v1.0.3 were used.

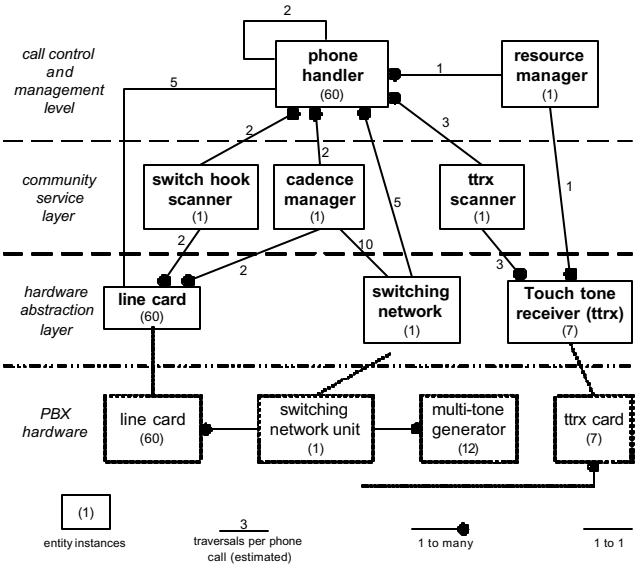


Figure 3. Class Diagram of PBX Control Program

4. OBSERVATIONS

Generally it was straightforward to implement the health indicators in AspectJ. The join points were easily located based on the stable state transitions. Multiple consistency checks were implemented together in one aspect. Because all consistency checking code, as aspects, was decoupled from the functionality, it can be disabled easily by not including the file.

Some facets of aspect implementation ran against basic software engineering principles. An example of this was the roundabout way in which internal state of objects was extracted by aspects. Certain classes didn't provide mechanisms to read back certain fields. It did not seem appropriate to use introduction to insert 'Get' methods into those classes because they would be separated from their corresponding 'Set' methods originally provided. Instead, these 'Get' methods were included in the functional program itself even though the original program didn't use them.

It seemed advantageous that for the PhoneHandler class, the horizontal and its corresponding vertical checks be placed together in the same aspect since they dealt with the same upper layer object. Rather than identifying all the locations for each consistency check, it was easy to have one single advice, which, after each successful stable state transition, performs the appropriate consistency checks depending on the PhoneHandler state.

When implementing the third consistency check for the tone generators, it was apparent that sometimes the identity (which instance) of the object on the other side of an association wasn't readily known. In this case, some traversal of the switching network was necessary to determine what was physically connected at the other end. If more than one check relied on this sort of traversal, it would be beneficial to traverse the association once and perform all the consistency checks together. This type of information digging is more complicated than simpler 'Get' methods. It is yet to be determined whether it is more suitable to include it in the aspect or in the original classes.

Some relationships are persistent, such as the one-to-one relationship between each pair PhoneHandler and LineCard instances. Other relationships change, such as when TouchToneReceivers are reassigned to different PhoneHandlers. One possible new heuristic would be based on knowing the persistency. It may help in determining which associations to check, or how often they should be checked.

In the initial implementations, consistency checks were triggered on every state change. This has the potential for scalability problems, which could be resolved by running the checks on sampling basis. This was shown to work by implementing the aspect to use the PBX's scheduler to execute the advice periodically.

The PhoneHandler consistency checks were relatively small, with the aspect totalling 100 LOC. Runtime overhead was 26 µs per PhoneConsistencyAspect invocation on a 300 MHz Sun UltraSPARC 10. Containing three consistency checks, this is similar to the ~5 µs per advice measured earlier in [5].

5. CONCLUSION

The paper presented an approach to performing state consistency checks and detecting internal state corruption of an executing software system. The paper outlined a categorization of and a number of heuristics for selecting consistency checks and their placement.

The consistency checks derived could be coded directly as part of the system code. However, the aspect-oriented implementation

provides a cleaner separation while significantly reducing the likelihood of inserting new faults into the system code.

6. ACKNOWLEDGMENTS

This work was supported by the Ontario Ministry of Energy, Science, and Technology, Singapore-Ontario Joint Research Programme project UW2827601MEST.

7. REFERENCES

[1] J. Thai, B. Pekilis, A. Lau, R. Seviora. Aspect-oriented Implementation of Software Health Indicators. Proc. Asia Pacific Software Engineering Conference (APSEC 2001). IEEE CS Press, 2001. 96-104.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold. An Overview of AspectJ. Proc. European Conference for Object-Oriented Programming (ECOOP 2001). Springer-Verlag, 2001.

[3] A. M. Davis. Software Requirements: Objects, Functions, and States. Prentice-Hall, 1993.

[4] I. Jacobson, G. Booch, J. Rumbaugh. The Unified Software Development Process. Addison Wesley, 1999.

[5] J. Thai, B. Pekilis, A. Lau, R. Seviora. Detection of Errors Using Aspect-Oriented State Consistency Checks. Supp. Proc. International Symposium on Software Reliability Engineering (ISSRE 2001). IEEE CS Press, 2001. 29-30.

Table 1. PhoneConsistencyAspect.java

```
/*
* File:              PhoneConsistencyAspect.java
* Description:       Perform three consistency checks on PhoneHandler.class
* Project:           PBX Control Project
* Author:            Alex Lau (2001)
*/

package bsr.pbx;
import bsr.pbx.*;
import bsr.pbx.resources.TTRXClass;
import bsr.pbx.resources.LineCardClass;

privileged aspect PhoneConsistencyAspect
{
    after(PhoneClass p) returning (boolean success): execution(boolean PhoneClass.enterState(String)) && target(p)
    {
        if(success)                                                 // Check after successful entry into stable state
            checkPhoneConsistency(p);
    }

    void checkPhoneConsistency(PhoneClass p)
    {
        boolean inconsistent = false;

                                                                    // FIRST CONSISTENCY CHECK -- HORIZONTAL
        if(p.state == PhoneClass.OHT)                               // Offhook talking
        {
            if(p.otherParty == null)                                // but associated with null other party
                inconsistent = true;
            else if(p.otherParty.otherParty != p)                   // but my other party is not associated with me
                inconsistent = true;
        }

                                                                    // SECOND CONSISTENCY CHECK -- VERTICAL
        if(p.state == PhoneClass.OHW)                               // Offhook waiting for digit
        {
            if(p.touchToneReceiver == null)                         // but associated with null touch-tone receiver
                inconsistent = true;
            else if(p.touchToneReceiver.notifyObject != p)          // but touch-tone receiver not associated with me
                inconsistent = true;
        }
        else if(p.touchToneReceiver != null)                        // else shouldn't have a touch-tone receiver
            inconsistent = true;

                                                                    // THIRD CONSISTENCY CHECK -- VERTICAL
        int tone = getTone(p.getLineCard());                        // Determine the tone generated connected to me
        if(p.state == PhoneClass.OHW && p.phoneNumberDialled.length() == 0)
        {                                                           // Offhook waiting for 1st digit, no dial tone
            if(tone != Managers.ToneGeneratorManagerClass.DIAL_TONE_CARD
               && !(tone >= Hardware.FIRST_TTRX && tone <= Hardware.LAST_TTRX))
```

```
                inconsistent = true;
        }
        else if(p.state == PhoneClass.OHT)                          // Offhook talking, but some tone is applied
        {
            if(tone != -1)
                inconsistent = true;
        }
        else if(p.state == PhoneClass.OHRT)                         // Offhook ring tone, but ring tone not applied
        {
            if(tone != Managers.ToneGeneratorManagerClass.RING_TONE_CARD)
                inconsistent = true;
        }
        else if(p.state == PhoneClass.OHSBT)                        // Offhook slow busy tone, but not applied
        {
            if(tone != Managers.ToneGeneratorManagerClass.SLOW_BUSY_CARD)
                inconsistent = true;
        }
        else if(p.state == PhoneClass.OHFBT)                        // Offhook fast busy tone, but not applied
        {
            if(tone != Managers.ToneGeneratorManagerClass.FAST_BUSY_CARD)
                inconsistent = true;
        }
        else if(p.state == PhoneClass.OffHI)                        // Offhook idle, but idle tone not applied
        {
            if(tone != Managers.ToneGeneratorManagerClass.IDLE_TONE_CARD)
                inconsistent = true;
        }

        if(inconsistent)
            throw new RuntimeException("Phone [" + p.getLineCard().getShelf() + "-" + p.getLineCard().getCard()
                                       + "] inconsistent");
    }

    int getTone(LineCardClass lc)                                   // Get the tone that's connected to our linecard
    {
        boolean inconsistent = false;                               // Tone undetermined *or* not applied
        int destShelf = lc.getShelf();                              // Destination is my own linecard
        int destCard = lc.getCard();
        int slot, sourceShelf, sourceCard = -1;                     // Source is what's connected to me

        if(lc.getIdle())                                            // Idle tone connected
            return Managers.ToneGeneratorManagerClass.IDLE_TONE_CARD;

        slot = SchedulerClass.getInstance().managers.switchManager.outputTimeSwitch[destShelf]
               .getOutputTimeSwitchSlot(destCard);
        if(slot < Hardware.FIRST_USED_SLOT || slot > Hardware.LAST_USED_SLOT)  // Slot out of range
            inconsistent = true;
        else
        {
            sourceShelf = SchedulerClass.getInstance().managers.switchManager.spaceSwitch
                          .getSwitchSource(slot, destShelf);
            if(sourceShelf != Hardware.SERVICE_SHELF)
                inconsistent = true;                                // Source shelf not SERVICE_SHELF
            else
            {
                sourceCard = SchedulerClass.getInstance().managers.switchManager.inputTimeSwitch[sourceShelf]
                             .getInputTimeSwitchCard(slot);
                if(slot <= Hardware.ZERO_CHANNEL || slot >= Hardware.CTRL_CHANNEL)
                    inconsistent = true;                            // Source card out of range
            }
        }

        if(inconsistent)
            return -1;
        return sourceCard;
    }
}
```

Aspect Oriented Logging in a Real-World System

Sabine Canditt
Siemens AG, CT SE 1
Otto-Hahn-Ring 6
81730 Munich
+49 (89) 636-46752
sabine.canditt@mchp.siemens.de

Manfed Gunter
Siemens AG
+49 (9131) 84-5837
manfred.gunter@siemens.com

ABSTRACT
In this position paper, we present the concept of using AspectJ for system logging in a large scale distributed system developed at Siemens. First we will give a short description of the project itsself with focus on the logging concept. Then we will state the arguments that led to the decision to use AspectJ [1]. A coded example will show the essentials of the logging aspects. Finally, we will mention open issues and problems to be solved.

Keywords
Logging, AspectJ, log4j

1. INTRODUCTION
The complexity of modern software systems introduces challenging requirements for testing, deployment and maintenance. Logging is a mechanism to gain information about a running system and is therefore an important part of a system's infrastructure. In order to generate log information, probes that intercept and store the execution flow must be introduced into the System Under Test. This step is called instrumentation. It can be done manually or automatically with tool support. As logging is a typical "crosscutting concern", Aspect Oriented Programming seems an appropriate approach to solve the instrumentation task. With AspectJ, a compiler to support instrumentation is available for Java programs.

This position paper is a snapshot of the current state of the project, where we have just started to develop and apply aspect oriented logging. At the time of the workshop we hope that we will be able to provide answers to the open question.

2. THE SYSTEM
The system of interest is a Data Management System (henceforward referred to as 'DM') that is designed to handle different kinds of data such as images, documents and multimedia. Challenging requirements must be met such as access by dissimilar clients and protocols. The system will be deployed on multiple platform hardware. Figure 1 shows a typical configuration with:

- Web Server: allows DM access via the web
- Application Server: contains the core application logic
- File Server: stores binary data such as images, audio and video files

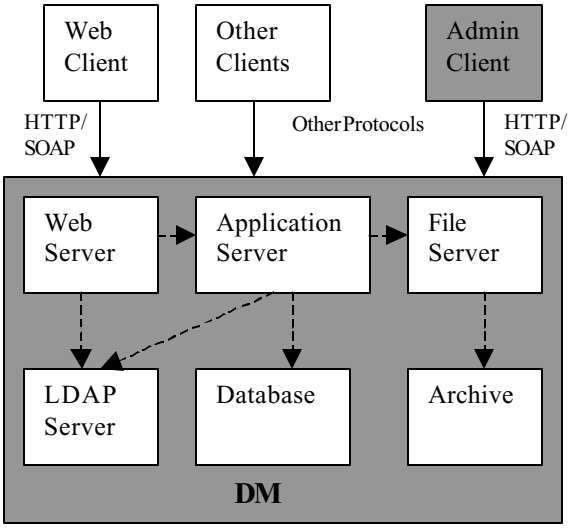


Figure 1: High level architecture of the DM

- LDAP Server: provides user-related data, e.g. role information for a certain user
- Database Server: provides various data as well as the location of the binary data
- Archive: stores image data on a permanent basis
- Admin client: allows administrators to administer and configure the DM

Each of the platforms host dedicated software building blocks (e.g. application, storage/persistence), as well as commonly used software building blocks (e.g. communication, infrastructure). The software is 100% pure Java.

3. THE LOGGING CONCEPT
3.1 Introduction
Logging is an essential concept to guarantee the serviceability of a DM. The administration and service guidelines require that each message must contain information about date, time, source (component, subsystem), severity (level of concern) and facility (stakeholder).

The DM logging framework is part of a software building block dedicated to infrastructure that is available on each platform. There are basically four different types of logging:

- Error: reports incorrect behavior
- Trace: reports the normal flow of program execution. It can be used to localize the reason for errors and also to prove correct behavior
- Audit
- Statistics

The DM logging framework consists of four major parts:

- Message: contains clases for the representation of the logged messages and their attributes. A message is composed of a message id and dynamic extended attributes for the message text. Further information (severity, facility, message text itsself) is encapsulated in the message id and obtained at runtime (see 3.2)
- Exception: contains exception classes that integrate error handling with the message concept
- Logger: contains classes to log messages and exceptions
- Log4J [2]: a logging framework from the Apache Jakarta project

Logging information is used by different stakeholders during different phases in the system' s lifecycle:

- for the developer in the implementation and (unit) test phases as well as for the integrator in the integration, integration test and system test phase (stakeholder *Developer*)
- for the administrator in the operation phase (stakeholder *Service*)
- and the customer/user in the operation phase (stakeholder *Customer*).

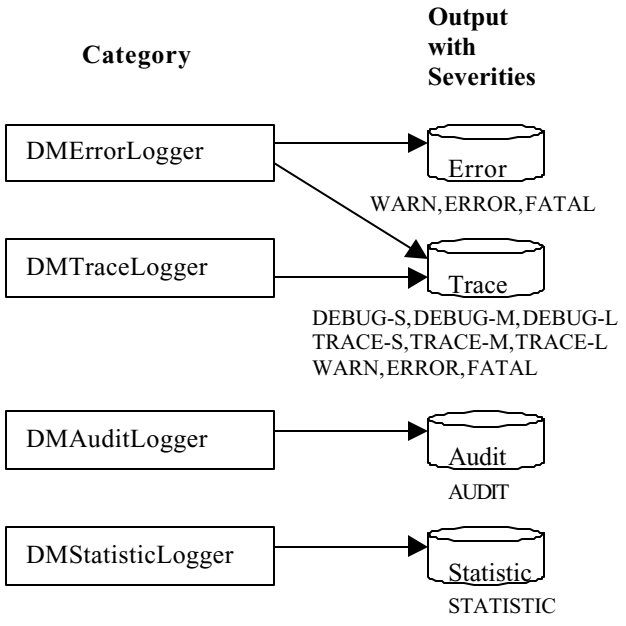


Figure 2: Loggers with outputs and available severities

It must be possible to provide logging information of the different types at different levels of granularity with relevant information appropriate for the different stakeholders. To this end, each log message contains the two elements:

- Severity: identifies the level of concern (e.g. ERROR, INFO..., see Figure 2)
- Facility: identifies the stakeholder of the message (Developer, Service, or Customer)

Log4J is an open source logging framework for Java that is now part of the Apache Jakarta project. It is designed to be fast in order to have a minimal impact on the system performance. It allows the developer to control which log statements are output with arbitrary granularity (configurable at runtime using external configuration files) and is therefore suitable to support the different levels of logging for the different stakeholders. For this reason, it was chosen as a basis the DM logging framework.

Log4J provides so-called *Categories* that provide basic logging methods (like info or error) with assigned *Priorities* (e.g if c is a Category instance then c.info(" ") is a logging request with Priority INFO). Furthermore, each Category may be attached to one or more output destinations (so-called *Appenders)* like console, files, sockets, etc. An Appender can be associated with an output format (*Layout*). Log4j also renders information about date, time and the message source.

The DM logging framework provides Logger classes similar to Categories for Error, Trace, Audit and Statistic log types which may be obtained for each class using LoggerFactory. DM severities are mapped to log4j Priorities, whereas DM facilities are simple Strings within the logging message. Figure 2 shows the DM Logger classes with available severities. It visualizes the call of a log method to the DMErrorLogger, which produces an output to both the error and the trace output, whereas the DMTraceLogger only feeds into the trace output. Note that the current physical representation of the outputs are local files, but the final decision (local files/central files/database) is postponed until more details are known about the frequency and characteristics of the log messages. This is feasable thanks to the flexible Appender concept.

3.2 Example

The following coded example logs a message with id TS_DM_MESSAGE1 to a traceLogger which is obtained via a LoggerFactory for the application class MyDMClass.

```
DMTraceLogger traceLogger=
  DMLoggerFactory.getTraceLogger(MyDMClass.class);
  traceLogger.log(new DMMessage(
    Tracing.class, // yields package name for ids
    DMMessageIds.TS_DM_MESSAGE1));//id
```

Severity and facility (and other elements) of the Message are contained in DMMessageIds.TS_DM_MESSAGE1:

```
public static final DMMessageId TS_DM_MESSAGE1=
  new ImsMessageId(
    "TS_DM_MESSAGE1",
    DMSeverity.TRACE_S,
    DMFacility.DEVELOPER);
```

The resulting output looks as follows:

```
03/19/200211:52:10.213[139.23.189.611154715079
0][MyDMClass.TS_DM_MESSAGE1][Developer]TRACE_S
– Example message text
```

The output message contains both severity and facility information and may be filtered according to the stakeholder's needs. The date, time and source information has been supplied by log4j.

3.3 Trace point definition

While the DM logging framework provides the infrastructure and API necessary to produce logging information, one important task remains to be done: the definition of the trace points. We do not want to provide overall logging, for example, of all method calls (or all remote method calls) but rather carefully select those points for logging that mark a relevant progress in the program's execution. This is a challenging tasks which involves all stakeholders. The goal is to provide the stakeholders with information he/she can effectively work with and to avoid overwhelming him/her with confusing details.

View	Description	Severity	Facility
External	Interfaces between clients and DM	TRACE_S	Customer
Node	Interfaces between DM hardware platforms	TRACE_M	Service
Component	Interfaces between DM software building blocks	TRACE_L	Service
Low Level	Within a SW building block	DEBUG_S, DEBUG_M, DEBUG_L	Developer

Table 1: DM Views

To take the stakeholders' interests into account, we differentiate between four different system views (see Table 1). The process of creating these diagrams is to go through the use cases and to identify the points of emphatic progress, zooming stepwise into the system. On each level (view) it must be shown how an external interaction with the SDM system resumes inside the system, which nodes and components are affected and how they interact with each other. In this context, error and trace messages have to be defined.

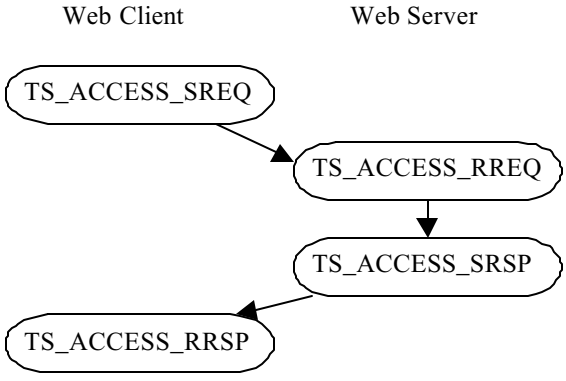


Figure 3: Activity diagram for trace point documentation

UML activity diagrams are used to document the trace points. Each activity represents a trace point, swim lanes are used to assign them to the respective software unit. Figure 3 shows an example for the External View (TS_ denotes the severity TRACE_S). In addition to the activity diagrams, a detailed description of the trace point position has to be defined (e.g. MyDMClass#myMethod() signals execution commencing).

The advantage of this approach is that all the message that are generated during run-time fit into the use case-based model of the system that was communicated with the stakeholders in the design phase. For verification, the traces can be compared with the activity diagrams:

Momentarily, the definition of the trace points is still underway (only approx. 10 trace points have been completely defined!) going hand in hand with the incremental software development process. That means that there is only incomplete knowledge about the amount and characteristics of the tracing information.

4. WHY USE (OR NOT USE) ASPECTJ?

After the identification of the trace points, the source code has to be instrumented accordingly. Instead of applying an error-prone "copy/paste" manual approach, we decided to use AspectJ. Tracing and logging is a common example for AspectJ. The arguments for this approach are as follows:

- The instrumentation is implemented in Aspects, i.e. it is separated from the application source code. Therefore the instrumentation can easily be maintained and checked for consistency with the trace point definition.
- The AspectJ compiler automatically inserts instrumentation calls at many locations of the source code and thus rendering the work of manual instrumentation unnecessary (especially on the client side of a method call). This is valuable if the amount of trace points is high. However, it turns out that this is not very relevant in the DM case (due to different tracepoint characteristics, see below).
- Instrumentation may be laid in the hands of one developer. This makes it much easier to fulfill instrumentation guidelines consistently; Not every developer has to deal with the instrumentation concept and API.

However, the DM tracing concept requires selective logging whereas the typical AspectJ tutorial example follows an approach of "complete coverage". For example: The following "tutorial" aspect traces the begin and the end of the execution of each method in each class of mypackage. It uses the AspectJconstructs *pointcut* (to define the trace point) and *advice* (to define what happens at the pointcut, i.e. the the tracing calls). The traced methods may be restricted using wildcard constructs, but the advice (tracing of the method name) is always the same.

```
aspect Tracing {
  pointcut trace():
    within(mypackage.*)&&
    execution(**(..));

  // advice: trace begin of execution
```

```
  before() : trace() {
    TraceSupport.traceEntry(
    thisJoinPoint.getSignature());
  }

  // advice: trace begin of execution
  after() : trace() {
    TraceSupport.traceExit(
    thisJoinPoint.getSignature());
  }
```

In the DM, not only the method name is traced, but other types of information, mainly the message Id (containing information about severity, facility etc.) and some parameters (not necessarily call parameters of the method). Also, the trace points are not as simple as in the example above: they may be set not only at begin or end of method execution but e.g. at the beginning of a catch block. I.e. the tracing advices have to be treated in a more individual manner, in the extreme case even with one dedicated advice per trace point. An automatic approach like AspectJ is more valuable if the trace points are frequently similar in nature. The decision as to whether the AspectJ approach is worth the effort at all (taking into account the problems and open issues raised below) can only be reached when more is known about the amount and characteristics of the trace points.

Note that AspectJ is not used to support different levels of instrumentation for the different stakeholders. There is only one version of instrumented code; selective logging is done via the configuration possibilities of log4j.

5. IMPLEMENTATION OF THE LOGGING ASPECT

Each application class has static Logger classes for the categories Trace, Error, Statistics and Audit. The example shows the category Trace only.

```
aspect Logging {
  // Trace logger for MyDMClass
  private static TraceLogger
    MyDMClass.traceLogger=
    DMLoggerFactory.getTraceLogger(
      MyDMClass.class);
...
  // log method1 execution
  before () :
    execution(*MyDMClass.method1(..){
      MyDMClass.traceLogger.log(
        newMessage(Tracing.class,
          DMMessageIds.TS_METHOD1_EXEC));//id
    }
}

  // log method2 execution (with param)
  before(String param) :
    execution(*MyDMClass.method2(..)&&
    args(param) {
      MyDMClass.traceLogger.log(
        newMessage(Tracing.class,
          DMMessageIds.TS_METHOD2_EXEC,//id
          param));
    }
}
```

The example shows that even the tracing of the execution begin of two method requires separate advices (due to different message ids and parameters). There are more complicated situations that necessitate some refactoring in the application code to simplify the application of the tracing aspects. Example: trace point TS_METHOD3_TRY marks the successful execution of a try block:

```
void method3() {
  try {
    method1();
    method2();
    //trace TS_METHOD3_TRY here
  } catch (Exception1 e) {
  } catch (Exception2 e) {
  }
}
```

There is no pointcut to capture the join point of "not handling an exception". The best way to achieve this functionality is to encapsulate the methods within the try block in a newMethod whose end of execution may be captured easily:

```
void method3() {
  try {
    newMethod();
  } catch (Exception1 e) {
  } catch (Exception2 e) {
  }
}


void methods() throws Exception1, Exception2 {
  method1();
  method2();
  //trace TS_METHOD3_TRY here
}
```

The necessity to refactor the application code is an argument against the AspectJ approach. However, the amount of refactoring is only visible when all the trace points are defined.

6. OPEN ISSUES AND PROBLEMS TO BE SOLVED

To summarize our experiences so far, the following problems and questions arise:

- Characteristics of trace points:
  The trace points for the DM system are very different: they require create tracing messages with a dedicated message id and a varying number of parameters. The target position in the application code is different (begin/end of method call/execution, but also begin/end of catch/try blocks..). A

separate advice has to be written for each trace point. The argument that the work of manual instrumentation is alleviated does not hold in this case.

- Documentation:
  UML is used to document the DM software. Aspects and crosscutting concerns in general are not easily captured by UML diagrams. While activity diagrams are an appropriate way to document the trace points, additional information has to be given to identify the exact position (method name, etc.).

- Debugging:
  The development team is used to work with the JBuilder IDE including the debugger. While AspectJ provides a nice extension for JBuilder to visualize the aspect code and the relations between application and aspect code, running the aspected software under debugger control doesn't seem to be possible. It is impossible to set breakpoints and step through the code, neither for the application code nor for the aspect code. This is a clear disadvantage of the AspectJ approach. The only readily discernable possibility for dealing with this problem is to apply the instrumentation aspects after the software has been developed and unit tested, i.e. when the main debugging work has already been done.

- Integration into the build process:
  The instrumentation aspects will belong to an extra package. This package has to be treated differently from the application packages (the AspectJ compiler has to be used instead of the regular Java compiler; class files are produced by the compilation not only for this but also for other packages). The build process has to be adapted accordingly.

- Test:
  As with any other software, the instrumentation has to be tested. One important test is to ensure that the instrumentation does not change the functional system behavior (naturally, instrumentation always changes the runtime behavior and thus the performance). As JUnit tests are implemented for the application packages, these tests will be run both on the original and the aspected classes. I.e. there will be two versions of the software, one with and one without instrumentation. This approach also alleviates the debugging problem mentioned above as the uninstrumented software is always available for debugging. However, since the unit tests do not aim at testing the instrumentation, it is more or less a matter of chance whether and in which order they produce the tracing output. As the main goal of the AspectJ approach is to modularize tracing, it would be desirable to modularize testing also, i.e. to have dedicated unit tests for the instrumentation only. This is difficult to achieve as the instrumented classes are embedded in their environment and may not be run separately. Providing the necessary environment for the instrumentation tests alone would require enormous overhead.

  Instead of instrumentation unit testing, the complete use cases that lead to the definition of trace points will be executed to ensure that the logging output actually contains all the defined trace points with all attributes set correctly. The logging output (textual, stored in the file) then would have to be compared, possible manually, with the trace point definition (which are captured in activity diagrams). Some automatic support would be helpful here. Another approach is to rely at least partially on static testing (i.e. a code review that compares the instrumentation aspect code with the trace point definition). The intermediate source code, obtained with the –preprocess option of the AspectJ compiler, may be reviewed to verify that the tracing calls have been inserted at the intended positions.

  Once the first set of trace files has been verified, regression tests will be much easier by simply comparing different version of trace files.

- Synchronization with application code:
  When defining the trace points, the methods that have to be traced (eventually including the complete signature) are captured in instrumentation aspects. But what if the names of methods/classes, quantity, or types of call parameters change during the development process? The aspect has to be adapted accordingly, otherwise the logging is never executed. There is no way for the AspectJ compiler to verify that the method names and signatures in the pointcut declarations really match with those in the application software. Here regression testing may give the crucial hints: if the trace file produced by the current software version does not contain all the entries that have been produced by an older version it is likely that some names have changed and the instrumentation aspects need to be adapted.

7. SUMMARY

Tracing and logging has frequently been mentioned as a typical task for AOP, and examples can be found in AspectJ tutorials. In these examples we often find an overall tracing approach that does not carefully select trace points nor differentiate between different levels and types. We evaluate the usage of a combination of AspectJ and a logging framework based on log4j to provide purely relevant information for different stakeholders in a large-scale real world distributed system. This position paper describes the advantages of this appoach as compared to a manual solution but also states problems and pertinent issues. It may seem that the problems, though soluble, outweigh the advantages under certain circumstances. A final decision can be made only with a more intimate knowledge about the characteristics of the trace points.

8. REFERENCES

[1] http://www.aspectj.org

[2] http://jakarta.apache.org/log4j/docs/index.html

Orthogonal Persistence using

Aspect Oriented Programming

Koenraad Vandenborre

Muna Matar

Ghislain Hoffman

Inno.com cva

Belgium

INTEC

Ghent University

Belgium

koenraad.vandenborre@inno.com , muna.matar@intec.rug.ac.be , ghislain.hoffman@rug.ac.be

Abstract

This paper describes a novel approach towards the decoupling of persistence issues from a class library. It first describes the need for persistence and then how persistence issues get fully orthogonalised from the class library by using the Aspect Oriented Paradigm. It is completed by an example, using Java and AspectJ, to illustrate the sketched methodology.

Introduction

In software engineering applied to business systems, there clearly is an evolution from writing proprietary middleware code towards using middleware services. We for instance mention the efforts taken in the J2EE environment concerning persistence, transaction management… This evolution enables developers to separate the writing of business logic from the writing of middleware services.

This evolution in software engineering enforces and gets enforced by an evolution in the business paradigm many organisations nowadays are confronted with. This evolution, as well encountered inside as outside the walls of the organisation, drives organisations to migrate from a silo based towards a service based business.

In silo based environments there often is a culture of data replication resulting in many data sources, containing inconsistent and redundant data, a situation that becomes intolerable in a service based environment.

So, from a software engineering point of view we don't want to bother the developer with persistence issues and from a business point of view, unambiguous persistence is a major requirement. These considerations lead to the conclusion that from both the engineering and the business point of view there's a rationale to look for an effective persistence model.

Scope

In software engineering persisting an entity means extending its lifetime beyond the lifetime of the application that created it, so that the entity can be used later on in the same application, or in other applications. In achieving this goal software engineers are confronted with a myriad of challenges:

- The entity can be saved in a relational database, stored in an XML repository, put in a spreadsheet or it can be decided not to store the entity by itself but instead recalculate it from other persisted entities.
- Furthermore, the functionality to deal with the persisted entities - select, update, create, delete – can be written using 4GL, stored procedures, JDBC, entity EJB's, dedicated data access objects behind a Session

Façade, through a proprietary API of an EIS….

- How the entity should be stored and retrieved from its persistence medium can be influenced by the fact whether it's used in batch or on-line processing
- It could be necessary to replicate a data source if using the original data source would impact the performance of the systems already running on the original data source...
- On top of that, the entities we use in applications and store in data sources are merely models of real life entities. It could turn out that these entities must be remodelled after a certain period of time resulting in different versions
- Many classes in an OO-model need persistence, which results in scattering of the persistence code through the class library. This decreases the maintainability of the code and the reusability of the classes as they contain persistence related code that is not necessarily needed or wanted in another system or business domain.

This paper doesn't pretend to solve all persistence issues but sketches a methodology to decouple persistence related issues from Java classes.

Much work has been done to address the persistence problem, we mention for instance [2], [3], [8]. As far as we know however, all those approaches were restricted to one programming language and/or lost much of their intrinsic value due to the restrictions of the used programming language and the adhered paradigm. Up to the emergence of the aspect-oriented paradigm, there hasn't been a clean, language-independent meta-description of the problem.

The aspect-oriented paradigm on the other hand finds its reason of existence in capturing issues that crosscut a certain class library. This leads to our statement that persistence is an issue that can be captured and described using the aspect-oriented paradigm. Furthermore we state that using the aspect-oriented paradigm, persistence can be fully orthogonalised from a class system or business model. In doing so we introduce a novel approach towards the problem.

In [8], a methodology is developed to build a framework that has the ability, through combination of introspection and the use of JavaDoc tags – to overcome Java's lack of declarativity for persistence -, to build and maintain knowledge how to make objects of certain classes persistent. Therefore, it would be a realistic approach to restrict our selves to describe how to prepare classes for persistence. However, for the sake of proof and simplicity, the example implementation in this paper doesn't use the framework but instead gives a very rough, per class persistence implementation.

The rest of the paper focuses on how to achieve the orthogonalisation between persistence and a class as an abstraction of a real life entity. To prove our statements a programming language has to be chosen. The object-oriented language used is Java, the aspect oriented one aspectJ, developed at Parc Xerox.

First the general methodology, starting from a business model, is sketched and thereafter it is applied to a simple example.

Designing the business model

Suppose we have to model a simple invoicing system. As a first step the problem domain must be analysed and a business model must be designed. In this business model some classes must be made persistent. At this phase however, we don't want to be bothered with persistence related issues, we just want to design our classes as abstractions of real life entities. At a later stage it will be decided what classes must be made persistent and how this must happen. The business model for the simple invoicing system is the one depicted in Figure 1.

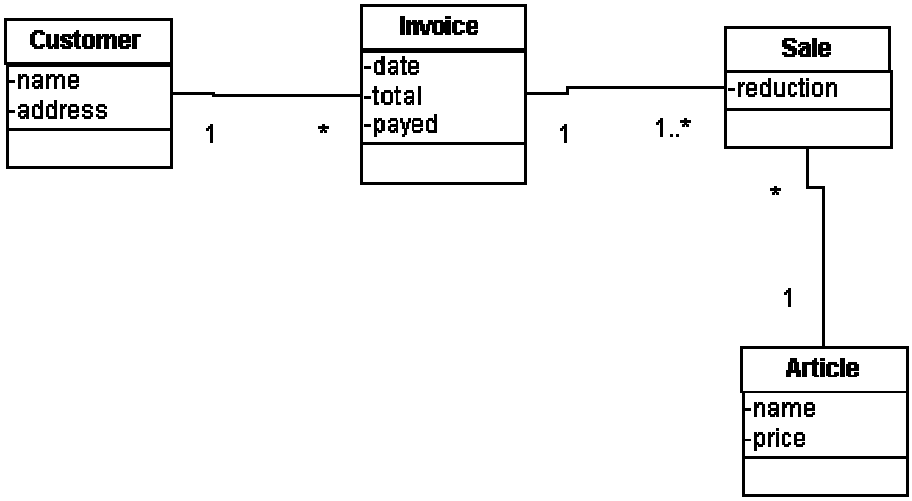


Figure 1: Simple Invoicing System

It is possible to implement this model at this stage and to test it, by providing it with dummy data.

Persistence and the business model

When considering object persistence, two main issues arise. First, every object to persist must have the appropriate functionality to be stored in and retrieved from a data source. Secondly, query functionality is needed. If not, we can save our selves a lot of trouble by just serialising the objects. This query functionality could for instance be: find all customers that have invoices that should have been payed last month. The question arises if this functionality should be part of one of the classes involved in the association. We believe not. This functionality results from the relation between Customer and Invoice for a specific business domain. Another business domain might impose the same relation between Customer and Invoice – a one to many relation - but demand other functionality from it. We believe this functionality should be put in a dedicated object that models the relation between Customer and Invoice. By using this approach, the Customer and Invoice classes become more reusable over different business domains. The needed query functionality for the relation class then becomes just a persistence issue for this class.

During implementation, if only pure Java is used, there exist two major approaches to indicate on the class level that a class must be made persistent. The first is to extend the involved classes from a base class, say PObject, which introduces the needed attributes and methods. Drawbacks in using this method are:

1. Java's only "extend" relationship is consumed for implementation purposes, not for design purposes.
2. Some attributes must be declared protected in order to manipulate them in derived classes, which introduces weaker encapsulation.

Another way around would be to let every class that has to be made persistent, implement an interface, say Persistent. Drawbacks here are:

1. We cannot introduce non-static, non-final attributes in the classes implementing the interface.
2. There is no way of introducing some standard behaviour for the methods of the interface.

Whatever approach is chosen, extending from a base class or implementing an interface, the programmers implementing the class involved are concerned with the class as an abstraction and with persistence aspects, be it overriding methods from the ancestor class PObject or implementing methods from the interface Persistent. Furthermore, persistence issues are spread over all the classes that are to be made persistent, which causes them to ripple throughout the entire code, a maintenance nightmare.

The key problem is – as stated above - that persistence crosscuts the entire object model in a way that cannot be cleanly expressed by just using the Java implementation of the OO-paradigm.

Introducing persistence aspects

The aspectJ language provides us with two techniques to capture crosscutting concerns. The first one - known as introduction – provides the ability to introduce attributes, methods and constructors in existing classes. The second one - known as advice - provides the ability to execute extra code at certain points in time defined by what in aspectJ is called pointcuts. It is the combination of both which will allow full orthogonalisation.

Using introduction one can introduce in existing classes the extra features needed to obtain a persistent class. The two items to decide on are: what and where to introduce.

To solve the question where to introduce, we can make every class that has to be made persistent implement an empty interface Persistent, very much like the standard Java interface Serializable. This makes that the class can be considered being of the type Persistent.

What to introduce can for instance be an attribute objectIdentifier and the methods read(), write(), update() and delete() - to read, write, update or delete an object from or to a persistence medium. The introduced methods can contain implementation code. This however is not such a good idea because for every class whose objects must be made persistent, one must clearly define how to make them persistent. The writing of a generic write() and read() would, if not impossible, at least cause lots of trouble. We can, however, introduce the methods read() and write() as being (nearly) empty, solving the "what" question, and consider the second technique provided by aspectJ.

Taking into account the second technique - advice - it's possible to execute extra code at certain points in time. A set of points in time could for instance be the invocation of a method write(). Whenever a write() is called on an object which instantiates a class which implements the interface Persistent, extra code is executed. The aspectJ language provides us with the possibility to know, when a write() is executed and from which object it originates, allowing us to react appropriately.

The application of these novel techniques will be illustrated in the following paragraphs.

An example

Let's reconsider the UML diagram of Figure 1. The business model has been constructed, the classes have been developed using pure Java, and at this point it is decided that the classes Customer and Invoice must be made persistent. To that extent we construct the aspect PersistentIntroductor that will introduce the necessary persistence related attributes and methods that are common to both classes. Therefore, to be able to treat the classes as being of the same type Persistent, the aspect declares that each class should implement the empty interface Persistent, which is also written at that point, or reused. The aspect further introduces

- An object identifier
- A method remaining private to the aspect to retrieve the object identifier. This restricts calls to this method, and thus knowledge about persistence, to the aspect. It's not a part of the interface of the objects that will be used in the applications
- Generic methods to write, update and delete persistent objects, each returning Booleans to indicate if the operation succeeded
- A generic method read() returning a Vector, because a read operation can return multiple objects, customers not necessarily have unique names

```
public aspect PersistentIntroductor

{
  declare parents : Invoice implements Persistent;
  declare parents : Customer implements Persistent;


  private Long Persistent.oID = new Long(Math.round(Math.random() * 1000000));


  private Long Persistent.getOID()
  {return oID;}
  public Boolean Persistent.write(Persistent p)
```

```
  {return new Boolean(false);}
  public Vector Persistent.read(Long i)
  {return new Vector();}

  public Boolean Persistent.update(Long i)
  {return new Boolean(false);}
  public Boolean Persistent.delete(Long i)
  {return new Boolean(false);}
}
```

At this stage, however we have only introduced rather generic methods that don't execute persistence code. To address this problem, we construct other aspects. For brevity, this is illustrated for the aspect PInvoice working on the Invoice class. This aspect,

- Is privileged to be able to access the getOID() method, and to have access to the private attributes of the object, not having getXXX() methods, in order to write them to the database.
- Defines the pointcuts, the object involved must be of type Invoice and are exposed in the context
- Is responsible for the database connection
- On each pointcut there's an advice after returning from the methods denoted in the pointcut. This advice has access to the return value of the original method

```
public privileged aspect PInvoice
{
  pointcut reader(Invoice p) : target(p) && call(public Vector read(..));
  pointcut writer(Invoice p) : target(p) && call(public Boolean write(..));
  pointcut updater(Invoice p) : target(p) && call(public Boolean update(..));
  pointcut deleter(Invoice p) : target(p) && call(public Boolean delete(..));
  private Connection con = null;
  private void setConnection()
  {/*connects to database*/}
  after(Invoice p) returning (Vector v) : reader(p)
  {
       /*retrieves the argument of the method the advice is advicing on, gets a
       connection to the database, builds a PreparedStatement to read the invoice from
       the invoice table and the associated customer from the customer table, builds an
       Invoice Object and puts this object in the vector v returned by the original
       method being the subject of this advice*/
  }
  after(Invoice p) returning (Boolean success) : writer(p)
  {
       /*gets a connection to the database, builds a PreparedStatement to write the
       invoice object to the appropriate tables and returns true on success. This return
       value becomes the return value of the original method being the subject of this
       advice*/
  }
  /*analogous after advices for updater and deleter*/
}
```

Conclusions

1. All persistence issues - attributes and methods - can be removed from the business classes to a separate place: an aspect. In the coding example this was done for the attribute oID and the methods getOID(), read(), write(), update() and delete().
2. Business classes are better suited for reuse, because they're closer to just being a design abstraction, uncluttered with persistence issues. Furthermore the business classes become independent of the data source used because all persistence related code resides in the aspects.
3. The business model can be tested to a certain extent before any persistence feature is introduced. Even if it's not yet decided what persistence mechanism will be used, the business model can be implemented.
4. Adjusting the business model to make extra classes persistent takes an aspect source code operation and a recompilation.
5. The persistence interface of persistent classes is very simple and very logical to application programmers making use of the business model. They just have to call the read(), write()… methods.
6. The Java Virtual Machine doesn't need to be changed, because the aspectJ compiler ajc generates intermediate java code that gets compiled with the regular java compiler.
7. It can be argued that a technique like introduction breaks encapsulation. We do insert new methods and attributes in existing classes. However this is done in a clean and controllable way and stays at the level of implementation of the business model, not at the level of applications built on top of the business model.

Future work

In this paper we sketched a novel approach to decouple persistence related issues from a business model. The coding effort is rough and ad hoc. We certainly want to investigate the possibility of merging this approach with the framework from [8]. This would allow separation of the preparing of classes to be made persistent through AOP, from the actual repository that contains and maintains knowledge of how to make classes persistent.

Persistence is not the only middleware service. Transaction management and security for instance are also candidates for "aspectisation". We guess there's a lot of work to do in not only writing those aspects but certainly in the co-operation of different aspects.

References

[1] AspectJ™ , Xerox Corporation, Palo Alto : http://aspectj.org

[2] Peter M. Heinckiens : Building Scalable Database Applications, Object Oriented Design, Architectures and Implementations, The Addison Wesley Object Technology Series, 1998

[3] Atkinson : The Pjama project, University of Glasgow, Department of Computing Science : http://www.dcs.gla.ac.uk/pjava

[4] Scott W Ambler : Mapping objects to relational databases http://www.ambysoft.com/mappingObjects.pdf

[5] H.Ossher and P.Tarr : Multidimensional separation of concerns and the Hyperspace Approach, IBM T.J. Watson Research Center : http://www.research.ibm.com/hyperspace/Papers/sac2000.pdf

[6] http://www.research.ibm.com/journals/sj/361/srinivasan.html

[7] JavaBlend 2.0 tutorial from Sun Microsystems

[8] Muna Matar : A methodology for object persistence in Java based on a declarative strategy, PhD thesis Ghent University, faculty of applied sciences, Department of Information Technology, 2001

The relevance of AOP to an Applications Programmer in an EJB environment

Howard Kim and Siobhán Clarke

Department of Computer Science, Trinity College Dublin, Ireland
howard.kim@cs.tcd.ie

Position Paper for the 1st International Conference on Aspect-Oriented Software Development Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)

Abstract. Many of the examples that are used to demonstrate the value of aspect-oriented programming are based on crosscutting concerns such as distribution support, remote access of objects, and synchronisation. Enterprise Java Beans (EJB), a standard component model for component transaction monitors (CTMs), provides inbuilt support for these concerns, thereby reducing the need for the applications programmer to be concerned about them. Does this make aspect-oriented programming irrelevant in an EJB environment? In this paper, we describe a distributed system developed using EJB where crosscutting concerns were managed by the EJB environment. The system is an eVoting system that allows students to vote online in Student Union elections in Trinity College

Introduction

Most software systems consist of several concerns. Typical examples of concerns are: *logging, transaction integrity, persistence, authentication, security, and performance*. Many of these concerns do not affect one implementation module of the system but affect multiple modules; these are known as crosscutting concerns. With EJB the container handles support for *security, performance and container managed persistence (CMP)*. These sound a lot like the crosscutting concerns that are used to motivate the need for aspect-oriented programming (AOP) kinds of techniques. This led us to ponder the need for aspect-oriented programming in component-based development environments such as EJB.

In this paper, we examine the crosscutting concerns that were evident in an EJB implementation of an eVoting software system. In this system students should be allowed to authenticate themselves using a given student I.D and password and then be allowed to vote online in the given election. The system has a requirement that communication between client and server is secure and how a client voted is not determinable by examining communication channels or log files, this has the implication that when a vote is entered into the database it is not logged. Security is therefore a key concern in the system. Other concerns we discuss here are persistence and transactions.

In the background of this paper we discuss the J2EE platform. We then describe how EJB were useful in the development process and what problems they solved. We also relate our development with an emphasis on how aspects may be used in conjunction with EJB to solve the problems of crosscutting concerns. We conclude that EJB does solve some of the problems dealt with by aspects but there may exist the need for aspects in a distributed environment depending on the requirements of the system.

Background

J2EE application servers usually support three different types of security: *authentication*, *access control* and *secure communication*.

1. Authentication confirms the identity of a particular user and allows them access to certain resources in the system.
2. Access control applies administrator defined explicit policies that regulate what a user can do in the system. Policies are particular business goals/objectives that the application server must understand before it can determine the right of access to a resource. The EJB deployment descriptor allows an administrator to identify particular groups of users and permit access to the resource.
3. At present no specification exists for the secure communication between EJBs. The most popular solution used in web-based applications is Secure Socket Layer (SSL).

The EJB architecture deals with crosscutting concerns such as: security, administration, performance and container managed persistence (CMP). An applications programmer can if they so wish handle the code of persistence this is called bean-managed persistence (BMP)[7]. With CMP a developer can concentrate on developing the business logic and the container would manage the named crosscutting concerns. With EJB XML deployment descriptor it allows the applications programmer to associate container services with beans; a bean deployer or developer would mark beans as being persistent, transactional and would set the security policy. Fig 1 shows how an applications programmer would define persistence and security in an EJB environment.

```
<entity>
        <ejb-name>ISS</ejb-name>
        <home>ie.tcd.server.authentication.AuthenticateHome</home>
        <remote>ie.tcd.server.authentication.Authenticate</remote>
        <ejb-class>ie.tcd.server.authentication.AuthenticateBean</ejb-
class>
        <persistence-type>Container</persistence-type>
        <prim-key-class>java.lang.Integer</prim-key-class>
        <reentrant>False</reentrant>
……      //CMP Fields
</entity>
<security-role>
     <role-name>everyone</role-name>
</security-role>
<container-transaction>
     <method>
        <ejb-name>ISS</ejb-name>
        <method-name>*</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
</container-transaction>
```

Fig. 1 XML Deployment Descriptor

The SU eVoting System

For the development of the eVoting system, Session and Entity Beans (CMP) were needed. The system is based on the Model View Controller paradigm [1] and is also loosely based on the J2EE Front Controller pattern [2]. Fig 2 shows the architecture of the system. In the client side of the application a student enters their login details in an applet; this applet then generates a public/private key pair that is used to encrypt data sent to the server. Once the server authenticates that the user is valid, a special token or ticket is generated by the server and sent back to the client. The client can then use this ticket to vote in the given election by again encrypting their vote and sending it to the ballot server that validates if the ticket is a valid.

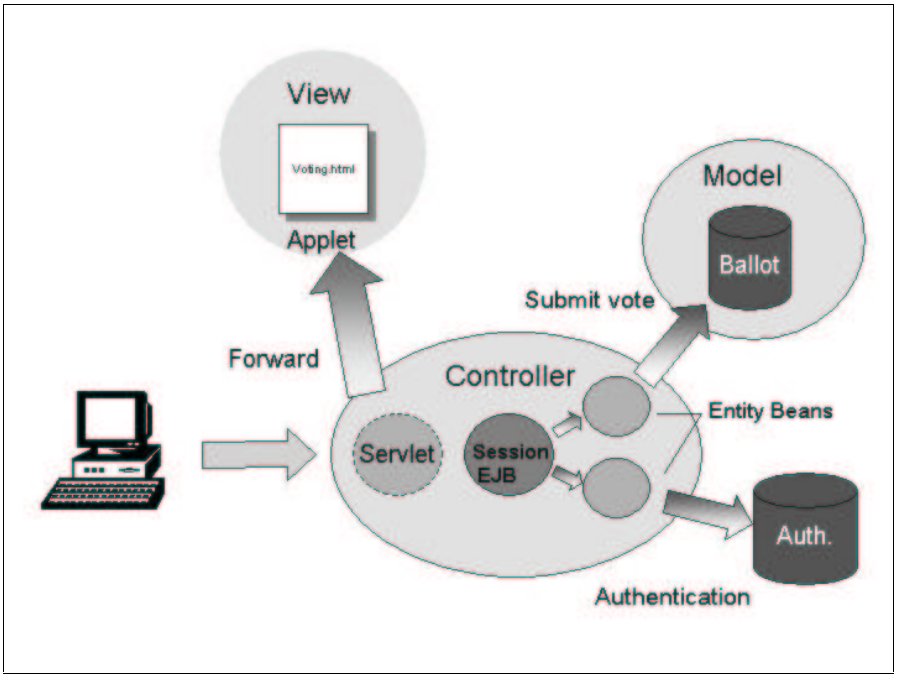


Fig. 2: Architectural Diagram of the SU eVoting system

We can view the system as a combination of multiple concerns; the concerns being:

1. Authentication of a given user.
2. Secure communication between client and server
3. Persistence storage of a user vote
4. Transactional vote updates
5. Secured access to the voting database.

These were probably the most important concerns of the system. Once a vote is entered into the database this vote must be persistent otherwise the loss of vote would mean a re-election being called. It was also imperative that no unauthorised access be allowed to the

vote database as you may guess the validity of the election depends on this concern. Finally authentication of users is a major concern of the system.

A more in-depth look at the concerns in the system revealed that all are crosscutting concerns. But the code remained modular because the container handled some these concerns[†] as Table 1 displays.

Concern Name	Crosscutting in a non-EJB environment	Solved by
Authentication of users	Yes	Using a combination of a Session and an Entity Bean solved this problem. The code is modularised into two components and not replicated. Fig 3 shows how authentication was handled for the system. No crosscutting remained for applications programmer.
Secure communication	Yes	By using Public Key Infrastructure (PKI), the communication channels between client and server remained secure. The code usage is scattered between client and server but this is necessary due to the nature of PKI. The code is not replicated and remained in a single class. No crosscutting remained for applications programmer.
Persistence storage of votes	Yes	The vote Entity bean handled persistent storage of votes. No crosscutting remained for applications programmer.
Transactional vote updates	Yes	Again the vote Entity bean/XML Deployment Descriptor handled transactional issues; by using the deployment descriptor beans were marked as being transactional. No crosscutting remained for applications programmer.
Secure access to database	Yes	The security policy for the system was set in the deployment descriptor at deployment. No crosscutting remained for applications programmer.

Table 1. Concerns in the system and how they were solved

[†] It should be noted that although these concerns resulted in modular code from an applications programmer point of view, the only reason is because the EJB container handles all these issues internally; such as *object locking, persistence management and security management*.

By using the EJB technology in the application many of the concerns involved when programming distributed systems were taken away such as transactions and server-side security. In Fig 1 we stated which methods should be marked as transactional and define the security policy (who is allowed access these methods). In this declarative programming model the applications programmer who creates the bean need not be the same person as the deployer who states that bean is transactional or set the security rights. As Giese [3] describes with EJB there is a pre-defined component lifecycle that ensures a well-defined interface, but this also has the property that instead of specific modules realising a particular aspect, the application server provides a predefined list.

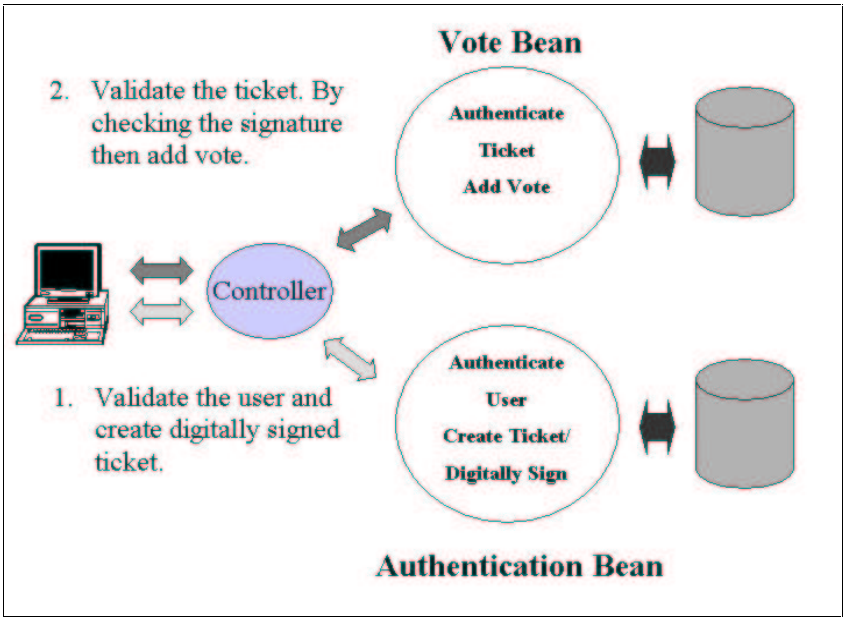


Fig 3. Process of validating user & vote

Although 'aspects' or 'crosscutting concerns', from the applications programmers view; were hard to find in this system we believe that they could be useful when an applications programmer needs transaction/database logging or in BMP.

Logging

When logging in an EJB environment the EJB specification suggests that `java.io.*` classes are not used within the bean (this is to increase portability). The two main products available for logging are JLog [4] and Log4j [5]. These have been used for logging within beans. Other solutions may involve writing log data through the network rather than to disk.

BMP

With BMP all persistence logic issues are left to the applications programmer. The applications programmer must write the persistence handling code into the bean class implementation. The structure of the database and how the bean class's fields map to the database must be known by the applications programming. Monson-Haefel [7] states that

application programmers can use BMP to develop custom beans for their business systems. Further research is necessary to see if AOP would help with logging or BMP in an EJB environment.

Discussions and Conclusions

It would be a surprising if there existed a large enterprise system that had no crosscutting concerns. Which is probably why a lot of techniques have been developed to modularise the implementation of systems with crosscutting concerns; these techniques include mixin-classes, design patterns and domain-specific solutions [6] and of course AOP. EJB is an example of a domain-independent server-side component model.

The project architecture ensured modularity even though as discussed some modules cross cut the application. By using EJB it ensured that the application could be used with any J2EE compliant server. The main difficulties with the J2EE architecture is that there is a steep learning curve in development, but this is out weighed by the advantages it has to offer; reliability, robustness, scalability and the enterprise computing power.

So what is the relevance to the applications programmer? The EJB specification requires the container to encapsulate crosscutting concerns such as transactions and persistence, but these are a fixed set of services that cannot be modified. EJB technology modularises crosscutting concerns by using design patterns (interceptor) but when the interceptor technique is not available or insufficient the code can easily become tangled, as in the case of BMP. AOP suggests using language mechanisms.

Our on going research is in the area of AOP and our next research is an evaluation of AOP in the Microsoft .NET environment.

Acknowledgements
Many thanks to the anonymous reviewers and to Roman Pichler for all their excellent comments.

References

1. S. Burbeck, Applications Programming in Smalltalk-80™: How to use.the Model-View-Control (MVC), 1999. http://st-www.cs.uiuc.edu/users/smarch/
2. Sun Microsystems, Sun Java Centre J2EE Patterns, 2002. http://developer.java.sun.com/developer/technicalArticles/J2EE/ patterns/
3. H. Giese. Towards Ruling Component-Based Distributed Systems with Role-Based Modelling and Cross-Cutting Aspects, University of Paderborn.
4. JLog IBM implementation of Java Logging tool http://www.alphaworks.ibm.com/tech/loggingtoolkit4j
5. Log4j Java Logger www.log4j.org
6. R. Laddad, I want my AOP (Part 1) Separate software concerns with aspect-oriented programming. http://www.javaworld.com 2002
7. R. Monson-Haefel, Enterprise JavaBeans 2nd Edition, O'Reilly publications 2000.

Using design patterns to improve aspect reusability and dynamics

Andrey Nechypurenko
Siemens AG, CT SE2
Otto-Hahn-Ring 6
Munich, 81739, Germany

andrey.nechypurenko@mchp.siemens.de

ABSTRACT

After the first implementations of AOP languages allow developers to exercise in applying the idea of concern separation in OOP, it becomes clear that despite opening new possibilities for developers, aspects are still software entities with all related old problems like reusability, customizability and effectiveness. In addition, the need to be able to dynamically switch aspects on and off has also been realized.

This paper provides contribution to research in the field of theoretical background for effective aspect implementation and introduces the design pattern-based Detector framework as a way to improve aspect reusability and to add the possibility to dynamically add/remove aspect-related functionality in the applications.

This paper also motivates aspect type classification as control-flow and state- triggered and proposes a way to deal with both types in a similar way by separating aspectual condition detection and handling using Observer-based Detector framework.

1. PROBLEM STATEMENT

One size does not fit all. If you are a software engineer who should create a library which will be used by other developers to build higher level libraries or applications, sooner or later you will realize that there is a set of incompatible (preemptive) requirements you should satisfy. There are two possible ways in this situation: a) to analyze the possible usage scenarios and optimize the library for the most likely use-cases; b) provide customization possibilities to let the library be tuned for concrete needs.

If you chose the first approach, 90% of your customers *probably* will be happy but the rest 10% will be forced to either re-implement required functionality or to code "between the lines" to achieve their goals.

With the second approach, you will need to define the set of interfaces to your system to provide the way to customize the system by substituting existing (default) functionality with custom implementation. This approach is well described in [Kiczales92] and the idea of "*dual-interface*" system was presented as a possible solution where the "primary" interface is the business functionality exposed by the system and the "secondary" is the interfaces which provide different application customization possibilities.

In this paper I would like to address the problem of how the set of interfaces for application customization ("*secondary"* interfaces) should look like. The problem stems from the fact that with general-purpose libraries it is impossible to predict what kind of functionality will need to be customized. In extreme case, to provide highest degree of flexibility, all tasks (like memory allocation, error handling, synchronization, etc.) should be done indirectly using some kind of delegation to the substitutable implementation. The Strategy design pattern [GoF] is a possible way to implement such a delegation. But in practice, it is impossible to predict all situations where such flexibility will be necessary. It means that there will always be a risk that the one who will use your library will need to tune something that was not foreseen in the implementation.

To solve this problem, at least three tasks should be accomplished: a) define the functionality to be customized which is possibly distributed over the whole application, b) figure out how this definition should looks like in order to be reusable, robust, introduce minimum overhead, etc. and c) substitute existing implementation with the custom version.

AOP is the modern way to separate the functionality which crosscut the application, localize the implementation of such crosscutting concerns and weave the custom implementation back to the application. So the implementation part in this paper will rely on currently available AOP support provided by the AspectJ language and would concentrate on the second part - how the "secondary" interfaces should look like.

Currently available AOP languages like AspectJ provide linguistic means to localize crosscutting concern (aspect) related code in a single logical unit (aspect definition), and to define points (join points) in the dynamic call graph of a running program where aspect-related functionality should be inserted and executed. Despite obvious advantages, there are still problems left that need to be addressed by software architects who are trying to apply the idea of separation of concerns for developing next-generation software systems. This paper focuses on the three following problems:

- *Aspect reusability problem* - how to avoid application specific code in aspect implementation.
- *Aspect dynamics problem* - how to make it possible to switch on/off aspect-related functionality.

- *Aspect uniformity problem* - there are at least two major aspect categories: *control-flow-* and *state-triggered* aspects. Despite different nature, it is desirable to handle both categories in the similar way.

Depending on the AOP support provided by the respective aspect language, the problems mentioned above could be even more complicated in case no access to the source code is available – it is not possible to add/remove already available compiled aspect implementation to existing compiled application or dynamically switch aspect-related functionality on/off.

To illustrate the problems mentioned above, the sample application called *bean* which is distributed as a part of AspectJ will be analyzed. This application will be also modified to demonstrate the advantages of the proposal.

1.1 Reusability Problem

The reusability problem stems from the fact that assumptions and expectations about properties like method signatures or names of methods and variables are frequently encoded directly in the aspects. Consider the following advice definition of an aspect that adds JavaBean property notification mechanism support to a Point class.

```
/** Advice to get the property change event
  * fired when the setters are called. It's
  * an around advice because one needs the old
  * value of the property.
  */
void around(Point p): setter(p) {
String propertyName =
thisJoinPointStaticPart.getSignature().getName().su
bstring("set".length());
  int oldX = p.getX();
  int oldY = p.getY();
  proceed(p);
  if (propertyName.equals("X")) {
    firePropertyChange(
      p, propertyName, oldX, p.getX());
  } else {
    firePropertyChange(
      p, propertyName, oldY, p.getY());
  }
}
```

Figure 1. Original advice definition

This advice declaration makes assumptions about the existence of X and Y properties (marked by bold font) and corresponding getter/setter methods. Such an assumption leads to the impossibility to use the complete aspect for classes with other attributes because the advice mentioned above will fail to call corresponding notification methods.

1.2 Dynamics Problem

To illustrate this problem, please consider a network application with ability to detect intrusions and react on such a situation by switching to SSL protocol to transmit information over the network.

SSL, as any other encryption introduces calculation overhead, which is not desirable when application, performs in secure environment (for example intranet). But if the environment state changed and is considered as insecure (intrusion attempt detected), additional encryption should be turned on. This behavior could be treated as security-related aspect of the application and illustrates the need to be able to turn certain functionality on/off depending on current application and execution environment state.

With most available AOP languages, after an aspect is weaved with application code, it is impossible to turn weaved functionality on or off.

1.3 Aspect Uniformity Problem

Executing some piece of code before or after certain method calls can be considered as a typical example of *control-flow-triggered aspect* because the call to a particular method is considered as a condition to trigger aspect functionality.

But there could be other conditions where it is also necessary to execute some specific logic. I would call such conditions as *state-triggered aspects*. The primary difference between these two aspect types is that in the last case execution of aspect code is not related to call graph of a running program but triggered by special events generated as a reaction to the execution environment properties.

The security aspect example mentioned in the previous section could be treated as state-triggered (not control-flow-triggered as in *bean* example) because the execution of this aspect is triggered by some kind of external event and will lead to the application state change. But this state (secure state) could be not initially foreseen and will be introduced later for example as a reaction to the new requirements. But if the available functionality is enough to properly react on this new state changes (for example if there is already a method in the application to switch SSL encoding on/off) what we need is to call this functionality during state change.

Network bandwidth, network packets latency, processor loading, amount of available memory and free space on hard disk could be considered as another examples of such properties. Reaction on these properties changes is typically spread across the whole application and could be treated as crosscutting concern. It is a design challenge to provide infrastructure where such a type of aspects could be well localized and handled the same way as control-flow-triggered aspects.

1.4 Paper organization

The idea of how to resolve the problems mentioned above is based on the analysis of different roles of developers in software projects where AOP is used, and different aspect types which can be found in most applications.

The remainder of this paper is organized as follows: Section 2 introduces different roles of developers in software project where AOP is used; Section 3 introduces the key ideas how to improve aspect reusability and how to add dynamics in the meaning of the possibility to programmatically plug and unplug aspect-related functionality to the application code; Section 4 compares described approaches with related work; Section 5 summarize the open issues; and Section 6 presents concluding remarks.

2. DEVELOPER ROLES IN AOP PROJECT

In order to solve the introduced problems effectively it is necessary to understand the roles the developer can play in a project where AOP is used. Understanding of such roles could help to elaborate a solution with minimized code dependencies and as a result improve development and testing parallelism.

1. *Business logic implementer* – this is the one whose main assignment is to implement the business functionality expected from the application or library.

2. *Aspect architect* – this is the person who is responsible for defining application structure in such a form which will make it possible later on to insert different aspect related functionality. This role is not obvious and requires additional explanation. Consider the bank application which transfers money from account A to account B. The pseudo-code for transfer as a transaction-enabled method could looks like following:

```
/** Method to transfer money
  */
void transferMoney(
  Account source,
  Account destination,
  float amount) {

  transaction.begin();

  try {
    source.withdraw(amount);
    destination.credit(amount);
  }
  catch(OperationFailure x) {
    // rollback and report an error
    transaction.rollback();
    Logger.error(
      "Transfer: " + source + destination);
    return;
  }
  catch(Throwable x) {
    // any other possible problems
    transaction.rollback();
    Logger.error(x.getMessage());
    return;
  }

  transaction.commit();
}
```

Figure 2. Transaction-aware transfer method

If now we assume that transaction processing is a crosscutting concern and will be encapsulated in aspect definition, the business logic developer could interpret it as "*…just forget about transactions, the weaver will insert everything necessary in the right place*" and implement the same code as following:

```
/** Method to transfer money
  */
void transferMoney(
  Account source,
  Account destination,
  float amount) {

  try {
    source.withdraw(amount);
    destination.credit(amount);
  }
  catch(Throwable x) {
    // any other possible problems
    Logger.error(x.getMessage());
    return;
  }
}
```

Figure 3. Transaction-unaware transfer method

Such an implementation does not leave the chance (or better to say makes it rather difficult) to enable transactional behavior defined as aspects because it is not necessary to rollback transaction in any places in code where a Throwable exception is catched. It is possible to try to make such an insertion context specific using function names as criteria. But you will also need to wrap the same method with begin/commit calls. This is another join point type, so you got the maintenance problem – set of transactional methods should be synchronized with criterion definition to insert rollback call in catch clause. Instead, the following code does not contain transaction specific code but essentially simplify the task of adding transaction processing using weaver:

```
/** Method to transfer money
  */
void transferMoney(
  Account source,
  Account destination,
  float amount) throws OperationFailure {

  try {
    source.withdraw(amount);
    destination.credit(amount);
  }
  catch(Throwable x) {
    // any other possible problems
    Logger.error(x.getMessage());
    if(x instanceof OperationFailure) {
      throw (OperationFailure)x;
    }
  }
}
```

Figure 4. Redesigned transaction-unaware transfer method

With this implementation it will be possible to write "around" advice which wraps the original call with begin/commit calls and invokes rollback in case of OperationFailure exception.

It means that *aspect related functionality could not be added to any code.* The code *should be prepared to be aspectised* and this is exactly the *task of Aspect Architect* to define the rules a) how to remove crosscutting concerns from the business logic and b) how to write the code, which let aspects be weaved easily.

3. *Aspect implementer* – this is a developer who writes aspect related code based on the conventions and rules defined by Aspect Architect assuming that business logic follows these conventions.
4. *Application assembler* – this is the developer who is responsible for defining which aspects are need to be inserted into the business logic to satisfy application requirements.

3. DETECTOR-BASED FRAMEWORK

As a way to solve the problem described in the Problem statement section and taking into account different aspect types and developer roles mentioned above the use of Observer [GoF] design pattern in combination with Component Configurator design pattern [POSA2] could be considered.

The intent of Observer design pattern is to "*Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*" [GoF]. It is possible to use this functionality to notify all registered observers to let them execute some logic

and/or influence the call graph in case of system enters the certain state (for example, low network bandwidth available) or perform certain action (for example, allocating memory)[1].

The main idea is to encapsulate conditions which could lead to execution of aspect-related code using *Detectors* which detect particular conditions (code- or environment related) and *fire events* to let application react on certain conditions by executing *preconfigured code* which represents crosscutting concern. The following picture represents the general structure of the Detectors framework.

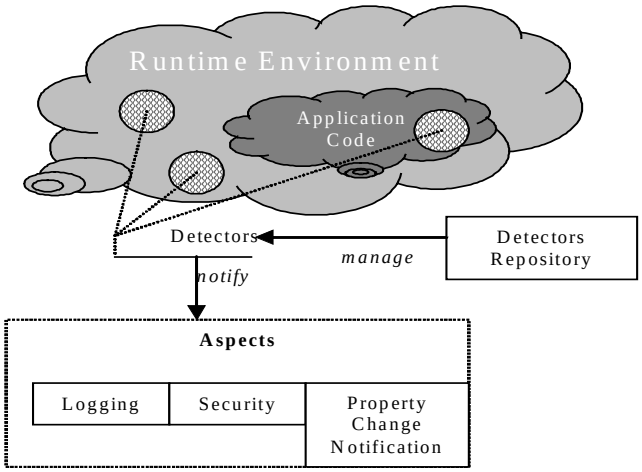


Figure 5. Framework structure

The goal of this structure is to *allow processing of different conditions* by inserting Detectors in the places where such a conditions could become true. A Detector itself does not contain any processing code but plays the *Observable* role and just provides the information that something happens and lets aspect-implementation play the *Observer* role and handle such an event using provided *context information*. Weaver could perform the Detector insertion task for code-related aspects. All the other Detectors types (for example to capture execution environment-related conditions, or composite detectors) should be configured into the Detector Repository by application developer.

To describe the framework in more details, let me revisit the *bean* example mentioned above applying proposed application structure.

3.1 Modified Bean Example

To modify the *bean* example we need to make the following steps.

1. Decide what kind of events we would like to detect using Detectors.
2. Figure out how to insert these Detectors. Manually or using AOP language
3. Implement the aspect as a listener for events fired by Detectors.
4. Instantiate Detectors and register them in the Repository.
5. Subscribe listeners for corresponding events.

The following subsections elaborate on each of these steps.

3.1.1 Event Types We Would Like To Detect

To support property change notification mechanism we need to execute our specific code before and after each property modification call (setter method) with our code. It means that we need to detect the *before method call condition* and *after method call condition*. Or using another words – we need to insert Detector before and after each setter call.

3.1.2 How to Insert Detectors

Based on the condition types described in previous section we could say that our events are *code-related* and the best way to insert such a Detector is to define an *around advice* and let weaver insert this definitions in proper places. The following is the possible definitions of such an advice.

```
/** Advice to get the property change event fired
  * when the setters are called. It's an around
  * advice because you need the old value of the
  * property.
*/
void around(Point p): setter(p) {
  // Obtaining detector instance from the
  // Repository
  if(aroundDetector_ == null) {
    aroundDetector_ =
    (AroundDetector)DetectorRepository.instance().
      get("around_setters");
  }
  if(aroundDetector_ != null) {
    // Before condition detection
    aroundDetector_.before(
      thisJoinPointStaticPart, p, this);
  }
  // Original method call
  proceed(p);
  if(aroundDetector_ != null) {
    // After condition detection
    aroundDetector_.after(
      thisJoinPointStaticPart, p, this);
  }
}
```

Figure 6. Advice definition for Detector insertion

This definition contains four steps.

1. Obtaining a Detector instance from the Repository using the "around_setter" string as a key for requests. Using a repository here allows different implementations of the Detector class itself. It is also possible that an "around_setter" Detector does not exists at all. In such a case no aspect code will be executed at all and the code will behave as without aspects at all.
2. Because in this example the Detector itself is a passive object we need to let the Detector "detect" the *before* condition by calling the corresponding method. This method contains the code, which is responsible for notifying all registered listeners.
3. Call original setter method using the *proceed()* keyword supported by AspectJ to let the original setter method be executed.
4. This step is similar to step 2 but lets Detector detect *after* condition.

[1] The Interceptor pattern [POSA2] could be considered as an alternative. Please see the Open Issues section for discussion on this topic.

Note, how the implementation specific code is removed from the advice declaration using detectors. Now this declaration could be considered as reusable because there is no assumptions about interface structure and property names in the advice declaration. This code just defines the place where the condition detection event will be fired.

3.1.3 Detector Implementation

The Detector is the Observable object and is responsible for maintaining observers list and notifying them.

```
public class AroundDetector
{
  public void addAroundListener(AroundListener l)
  {…}
  public void removeAfterListener(AfterListener l)
  {….}
  public void before(JoinPoint.StaticPart jpsp,
    Object target_object,
    Object advice) {…}
  public void after(JoinPoint.StaticPart jpsp,
    Object target_object,
    Object advice) {…}
  protected void notifyAfterListeners(
    AfterNotification n) {…}
}
```

Figure 7. Detector implementation

3.1.4 Implement the Aspect as a Detector Listener

The relationships between *detector* and *aspect implementation* are represented by means of the *Observer* pattern.

```
public class AroundSetterListener implements
  BeforeListener, AfterListener
{
  public void notifyBefore(BeforeNotification n){
    this.p = (Point)n.getTarget();
    this.propertyName =
      n.getJPStaticPart().getSignature().
        getName().substring("set".length());
    this.oldX = this.p.getX();
    this.oldY = this.p.getY();
  }

  public void notifyAfter(AfterNotification n) {
    if(this.propertyName.equals("X")) {
      ((BoundPoint)(n.getAspect())).
        firePropertyChange(
          this.p, this.propertyName,
          this.oldX, this.p.getX());
    } else {
      ((BoundPoint)(n.getAspect())).
        firePropertyChange(
          this.p, this.propertyName,
          this.oldY, this.p.getY());
    }
  }
  private String propertyName;
  private int oldX;
  private int oldY;
  private Point p;
}
```

Figure 8. Aspect as a Detector Listener

Now the code, which was initially in the advice declaration (see Figure 1), has been moved to the listener implementation.

3.1.5 Instantiate Detectors and Register them in the Repository

After running the weaver our initial code will be *Detector-enabled*. It means that we provide the infrastructure for detecting conditions of interest. It is like installing the communication channels for information distribution. But in addition, we need to provide information suppliers and consumers. In our case detectors are suppliers and aspect implementation as listeners are consumers. This task could be well formalized using the *Component Configurator* design pattern where detectors play the component role and the Detector Repository plays the Component Repository role.

Using this idea, two steps should be done to *equip the code with Detectors*: 1) instantiate the detector and 2) register the detector in the repository. Actually, if the Detector repository is implemented as a *Singleton*, the registration task could be handled in the Detector base class. In such a case the only thing that should be done is just a detector instantiation.

3.1.6 Subscribe Listeners for Corresponding events

The code, equipped with detectors will not expose any aspect – relevant behavior. We need to register our aspect implementations as listeners for certain detectors. The following code demonstrates how to make this step.

```
AroundSetterListener my_listener =
  new AroundSetterListener ();
AroundDetector around_detector =
   (AroundDetector)DetectorRepository.instance().
    get("around_setter");
if(around_detector == null ||
  !(around_detector instanceof AroundDetector)) {
    System.out.println(
      "Requested Detector not found");
    System.exit(1);
}
around_detector.addListener(my_listener);
```

Figure 9. Registering aspect implementation

This code fragment contains the following steps.

1. Instantiating the concrete listener implementation.
2. Obtaining a corresponding detector instance from the repository.
3. Subscribe the listener to the event produced by the obtained detector.

3.2 State-triggered Aspects

The previous sections describe in details how to insert code related aspects and let them detect the conditions of interest.

According to the classification presented in section 1.3 there is a second group of aspects – state-triggered. Using the *detector-listeners* paradigm, we could hide the different nature of aspects behind detectors. It means that the only difference in environment related case would be where and how corresponding detectors will evaluate available environment properties and fire events.

There are two possibilities – active and passive condition evaluations.

3.2.1 Active Condition Evaluation

In this case, there should be dedicated execution thread within the application. This thread periodically calls corresponding detection methods of detectors to let them evaluate available properties and decide whether to fire events (call registered listeners) or not.

3.2.2 Passive Condition Evaluation

In this case, execution of evaluation methods within detectors should be done using one of the application threads. Such behavior could be achieved for example by inserting evaluation calls before and/or after each or dedicated set of application business methods.

3.3 Tasks and Roles

As an advantage of proposed approach consider the following analysis of developer roles and corresponding responsibilities.

The tasks described above correspond to the different roles identified in section 3. The following table summarizes role/task relationships.

Table 1. Role/task relationships summary

Role	Task
Business logic implementer	Implement business logic
Aspect architect	Design code to make it possible later to introduce aspect-related behavior
Aspect implementer	Provide aspect implementation
Application assembler	Create application by putting together business logic and required aspects

This table demonstrates that proposed pattern provides clear separation of development roles. Such a separation could increase development parallelism. Such role/task definition could also be used as a hint for assigning code artifacts owners and project directory structure.

3.4 Performance Impact

It is obvious that introducing additional calls (for example before and after business method call) has negative impact on performance. But in the case of an empty listener list or even absence of detectors at all, the performance impact could be treated as really small if the business logic is complex. In case of trivial or really fast logic implemented by business methods, performance impact could be considerable comparing with overall time required for method execution.

4. RELATED WORK

The idea to use some kind of objects whose responsibility is to detect conditions of interest and let react on particular state in the execution environment is not new. The QuO framework [QuO] uses *SystemCondition* objects to provide interfaces to resources, mechanisms, objects, and ORBs in the system that need to be measured and controlled by QuO *Contracts*. However, the QuO framework is concentrated on QoS related aspects and does not provide easy possibility to seamlessly integrate source code related aspects into the infrastructure defined by the framework. QualProbes [QualProbes] is another framework which introduces not only the ways to catch particular events and/or state changes using Probes but also automates the adaptation algorithm by defining target condition and the ways to influent the system to reach this desired condition. This framework is also concentrated on QoS aspects as the QuO framework and does not cover execution flow related aspects.

In [Aspectual], the authors present the idea of aspect definition and implementation separation using a new term - *Connector*. This approach is similar to the idea presented in this paper. Detectors could be treated as a kind of connectors which just provide the link between the place where condition of interest was registered and code which handle this particular condition. [Aspectual] paper does not cover the state-triggered aspects though proposed approach could probably be used to cover this aspect type also.

5. OPEN ISSUES

As an alternative to the Observer pattern based framework, the Interceptor design pattern [POSA2] could also be considered. More concretely – in case of control-flow-triggered aspects, Detectors could be inserted “between” the method calls and “intercept” the invocation. But this is the simplest case. The proposed framework also makes it possible to install the aggregated detectors which observe the more primitive ones and correlate fired events. In this scenario, detectors looks rather like observables then interceptors. A state-triggered aspect is another case where detectors are more observables as interceptors. Especially if active condition evaluation is used (there is a dedicated execution thread to evaluate conditions in the detectors), each detector could block on detection call or itself get notification from another objects used to detect condition of interest.

But I am not quite sure that combination of Interceptor and Strategy pattern could not resolve the problems mentioned in this paper. I am going to evaluate this approach also.

6. CONCLUSIONS

This paper proposes the way to define application customization (“secondary”) interface as a Detector-based framework. Using such framework could also improve aspect reusability and introduce dynamics by interpreting the cases when aspectual functionality should be invoked as conditions detected by special classes called Detectors and processed by classes listening for notifications from Detectors. Such separation simplify the task of application fine tuning and allows changes to be introduces later in development time or even after deployment using reconfiguration possibility provided by Component Configurator pattern.

7. ACKNOWLEDGMENTS

I would like to say thank you to Klaus Ostermann who introduced the AOP to me and spent a lot of his time discussing the topics presented in this paper with. I am also appreciated hard but very helpful criticism and valuable paper improvement suggestions I have got from Frank Buschman. I am also would like to say thank you to Roman Pichler for reviewing early versions of this paper.

8. REFERENCES

[Kiczales92] Kiczales G. Towards a New Model of Abstraction in Software Engineering. In proceedings of the international workshop on new models for software architecture, November 4 – 7, 1992, Tokyo.

[POSA2] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2, Wiley & Sons, New York, NY, 2000.

[GoF] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[QuO] Schantz RE, Loyall JP, Atighetchi M, Pal PP. Packaging Quality of Service Control Behaviors for Reuse. ISORC 2002, The 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing, April 29 - May 1, 2002, Washington.

[QualProbes] Li B, Nahrstedt K, QualProbes: Middleware QoS Profiling Services for Configuring Adaptive Applications.

[Aspectual] Lieberherr K., Lorenz D., Mezini M., Programming with Aspectual Compone

Aspect-Oriented Programming for .NET

Mario Schüpany, Christa Schwanninger, Egon Wuchner
Siemens AG, CT SE2
Otto-Hahn-Ring 6
81739 Munich, Germany
Mario.Schuepany@fhs-hagenberg.ac.at
{Christa.Schwanninger, Egon.Wuchner}@mchp.siemens.de

Abstract

In the first part of this paper discusses the relationship between aspects, component models and patterns. The focus is the applicability of the technologies to handle infrastructure services, a topic relevant for all of them. Infrastructure services contained in component model platforms are among the most popular examples for crosscutting concerns in AOP. But the most intuitive crosscutting concerns may not always be the most rewarding to separate. To find out how developers use AOP in their day-to-day work we have to get more experience with AOP languages and systems. We have to extend the user community.

The second part of the paper describes an AOP system for Mircosoft's .NET development platform. The system combines ideas from several existing AOP languages like AspectJ™ [AspectJ], Hyper/J™ [Hyper/J] and Minos [Mezini et.al.] and makes use of special features in the .NET infrastructure.

1 Introduction

The phenomenon formerly called the software crisis describes the fact, that the size and complexity of software increases to the same extent software engineers think of better ways how to produce software. Some of the more recent approaches to cope with software complexity are aspects, component models and patterns. The question how these three paradigms relate to each other is very relevant to find the next steps in the race between ever increasing requirements for software systems and software development (r)evolution.

2 Aspects, Component Models and Patterns

The call for papers asks potential participants to think about the relations between aspects, component models, and patterns. These three concepts differ significantly regarding their generality and potential use beyond modularizing infrastructure services.
Component models like J2EE [Sun] or CORBA Component Model [CCM] are the most specific approach of these three. They find their predecessors in middleware that liberates the developer from caring for implementation of communication means and OS specific services. Component models are a consequent successor of the middleware idea concerning their goal to encapsulate infrastructure services in a container. This of course can't be done without standardizing the way in which business logic and infrastructure services deal with each other. The developers are forced to operate within these restrictions to be able to use the services [Pichler et. al.].

Patterns are the broadest concept of these three, component models, patterns and aspects. A pattern is a solution to a recurring problem in a context, and is always based on the experience of the experts in a field. A written pattern representation describes the pattern that emerged in at least three different places independently and proofed to be general enough to solve a whole family of problems and concrete enough to tell the user how to solve her problem.
Patterns exist in every domain, they are known for long in sociology, then in (building) architecture [Alexander1977], and for about 10 years we have pattern literature on conducting, teaching and organizing software development.
Thinking about the relationship between component models and patterns we conclude that there are design patterns that can be used to support separation of infrastructure concerns from business code in a flexible and extensible way. They are found in implementations of component models (Examples: Interceptor, Command, Proxy, Strategy, Chain of Responsibility, Factory Method). But design patterns are in no way restricted to such problem domains. Patterns are already documented for nearly every computer science niche in arbitrary levels of detail, while component models make use of patterns, but are restricted to solving these problems: relieving the programmer from caring for infrastructure services and fostering reuse by standardizing the interaction between components and their container.

Aspect Oriented Programming (AOP) [Kiczales et.al.] is a concept developed to support separation of crosscutting concerns. Before talking more about the relevance of AOP for implementing infrastructure services lets speculate about the relationship between AOP and patterns. There are some patterns that deal with problems that arise because of lack of means to separate concerns properly. Some of this patterns can also be implemented with AOP, sometimes the solution is

easier to understand and more elegant than the object-oriented solution described in the respective patterns. The downside of some "classic" OO design pattern implementations is that the code for the pattern has to be "entangled" with the business code upfront and thus makes it harder to understand the business code. Using AOP to implement the pattern helps in such cases.
But more often than not AOP is then just another implementation of such a pattern. E.g. when we take the well-known Observer [GOF], which was shown to be "implementable" with means of AOP languages easily [Noda2001]. But the pattern is still there; the AOP implementation is just another instantiation of the idea behind the pattern. The pattern describes the deeper concept. Once it helped to understand the problem properly it can be implemented with a variety of different languages and programming paradigms. On the other hand, some OO patterns are simply obsolete using AOP languages or technologies. But the pattern concept is not obsolete with AOP, it is more general than AOP, which is a new programming paradigm. Like patterns for OO evolved, patterns for AOP will evolve as soon as it is broadly used. Despite being orthogonal both approaches touch each other permanently.

The relationship between component models and AOP is discussed thoroughly in [Pichler et.al.]. The authors conclude, that both technologies are useful for separating infrastructure concerns from business code but both also have major disadvantages. Current component models require that the developer of business code follows certain design rules without being able to check them at compile time and they are not tailorable enough. The major AOP languages on the other hand lead to strong coupling of aspect code to business code and provide no runtime control over aspect lifetime.

2.1 AOP and Infrastructure Services

But lets get back to AOP and infrastructure services. Since AOP is made for separating crosscutting concerns, using it for separating infrastructure services from business code seems quite natural. In fact services like security, transactions, synchronization, persistence and distribution are amongst the most popular examples for crosscutting concerns next to tracing and debugging [AspectJ FAQ] [Elrad et.al.]. But the user community of dedicated AOP languages is still small, and quite a few of the identified "typical aspects" were found by deliberately thinking about what a crosscutting concern could possibly be. Many of them didn't emerge from a large community of developers working on their day-to-day problems. Elisa Baniassad conducted a small exploratory study [Baniassad] aiming to find what typical crosscutting concerns developers would like to see separated in big software systems when they are forced to perform major change tasks. The assumption was, that the change tasks themselves would reveal the crosscutting concern the developer would rather have factored out, e.g. when changing something in the notification policy the developer would like to see all code concerning notification in an aspect module. Surprisingly this wasn't the case. The developers (not familiar with AOP concepts) easily found all the places concerning their change task and had a pretty good plan what to do in which order. But every developer in the study encountered difficulties when the code to be changed intersected with some code belonging to a completely different crosscutting concern, that wasn't subject to the change task but was influenced by the same code (e.g. computation assumptions built into data structures, hardware platform dependencies, user interface consistency, resource speed and sometimes undecipherable obstacle). Obviously crosscutting concerns occurring in real-world problems tend to be less intuitive than we thought [Baniassad]. What if the easy to identify crosscutting concern are not really worth to be separated because they don't cause problems in their "entangled" form anyway? Infrastructure concerns are easy to identify (despite not always that easy to separate). But there are those less easy to identify, really badly entangled concerns that cause really hard problems when trying to change code.

What do we learn from this study? A relatively small number of researchers and early adopters can't do what a big group of software developers concerned with real project work can do: find out how to deal with this very powerful new way of decomposing software, a way that suits their needs and will be successful at the end. We believe, that identifying and isolating crosscutting concerns is a difficult task. Infrastructure services like tracing, security and persistence are quite independent building blocks of an application and do not require a lot of interaction with other code. Since it is easy to identify them, is not surprising that component models mostly deal with these types of crosscutting concerns. But more complex software systems have a relatively high degree of interdependent concerns. Trying to factor out such a concern into a reusable crosscutting building block requires some mechanisms of interaction with other concerns without breaking encapsulation. Handling the dependencies and the invocation order of crosscutting concerns seems to be a further challenge for AOP.

We have to extend the scope of AOP from people who are curious, take risk, are easily excited by new technology and willing to learn, towards the people who are under pressure to deliver software, maybe conservative, staying on the "save side", not easy to affect with a hype, but who are also incredibly valuable for conducting successful projects because they comprise a whole lot of experience. What we want them is to gain experience with AOP to find out, what crosscutting concerns should be separated to increase the quality of software. We want them to develop the AOP patterns.

This approach seems reasonable if we think of the first steps in object-oriented programming. It was easy to convince developers at that time that a window or a button can be an object. GUI was suited for OO, but what about compilers, banking applications and communication protocols? It took the industry quite a while to learn and honor the potential of OO, and it will take some time to learn what AOP is all about.

To foster the growth of the AOP community we work on an AOP environment for Microsoft's .NET platform. The reason for this is not that we think .NET is any better than Java, but because a lot of AOP tools are based on Java already. Tools to use AOP on .NET, which provides a number of languages helps all the developers fluent in these languages to join the user community.

There are a number of good languages, like [AspectJ] and [Hyper/J] that are ready to be used by the developers in their daily work and lots of ideas for new languages and means to incorporate the new paradigm into our daily work. In fact these groups work hard to enlarge their user communities [Kiczales et.al. 97] [Tarr et.al.]. The environment we work on takes elements from these languages and combines them with the means .NET offers. We are not the first to realize AOP on .NET. Ulrich Eisenecker and Daniel Weber implemented several prototypes for making use of AOP in C# [Eisenecker et.al.]). They, too, inspired our work. There is another approach from Dharma Shukla et. al. [ShFeSe2002]. They use COM+ functionality to implement AOP. This approach contains interesting ideas but is not flexible enough for our needs. It requires the developer of the base code to a priori tag the code with attributes wherever aspect code might be applied to later.

We deliberately don't consider all AOP approaches that are implemented within other programming paradigms in this paper. There are some really remarkable approaches to handle crosscutting concerns in a modular and flexible way without requiring AOP languages usually by providing OO framework approaches [Akkawi et.al.] [Teichert]. But these approaches often require in depth knowledge about the design of these frameworks and their use. We assume that this stands in the way when trying to propagate them to a broader community.

3 AOP# with .NET

In section 2.1 we motivated why we care for an AOP implementation building on .NET. In the second half of this position paper we describe the system, which only recently moved from the conception to the implementation phase. We call it AOP# in accordance with the new language C# for .NET. A prototype should be ready by June 2002.

The forces imposed by the wish to make it easy to use an AOP environment drive the requirements, which are

- no language extension; the user neither has to learn a new language, no non-standard compiler is needed nor legacy software is invalidated
- complete separation of business and aspect code; to foster reuse of both and make independent reasoning about only one concern at a time possible
- easy mechanism to join aspect and business code, preferably by simply configuring aspects into business code like configuring infrastructure services into a J2EE application.

A special thing we borrowed from Minos [Mezini et.al.] is the ability to switch aspects on and off during runtime. This allows to experiment with a new idea: the aspect context of an application can change during runtime and every context change can alter the code of "aspectized" methods. This concept is called "aspectual polymorphism", in analogy to OO polymorphism. Aspectual polymorphism means, that the set of aspects (the context) that affect an object at runtime is seen as part of its type and can change due to context switches. Every time a method of the object is executed this runtime type is used to decide which aspectual code is executed in addition to/ instead of the original method.

3.1 Overview of AOP# with .NET

The following section describes very briefly an AOP solution, which uses some of the language independent features of .NET, thus fostering the use of any language available within .NET. Every executable program in the proposed AOP solution consists of three parts: The *application assembly(ies)* holds the application or business code, henceforth called "core code", the *aspect assembly*, where the aspect code resides and a coordinating component, the *AOPEnvironment*, provided by the AOP system.
The information on how the aspects interfere with the core code is specified in an XML file, called *connector*. The connector is used by the *AOPEnvironment* to decide at run-time which aspect method has to be called before/after which core code method. Figure 1 shows these three parts and their relationship.

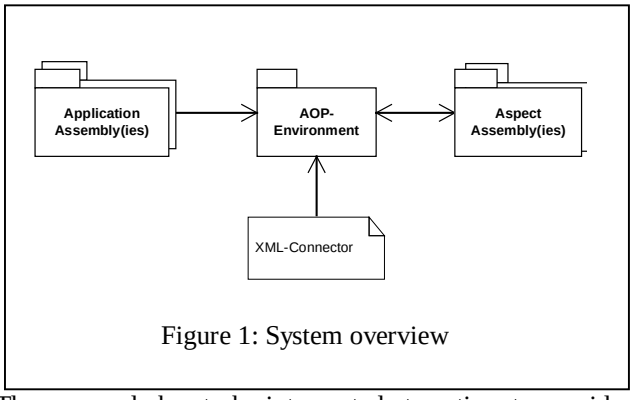


Figure 1: System overview

The core code has to be intercepted at runtime to provide an entry point for the *AOPEnvironment*, which then triggers the

execution of aspect code. This interception code is inserted at the loading time of the core application methods by using the Profiling and Metadata-API of .NET. One restriction at this point is that the Profiling API can only handle managed code. Code that can be compiled to Microsoft's intermediate language (IL) is called managed code in .NET, because .NET only has complete control over such code. Memory allocation commonly used in C++ for example is a language feature that can not be mapped to IL, thus portions of code that contain such constructs is unmanaged code.

3.2 What an aspect looks like in AOP# for .NET

An aspect in AOP# for .NET is an ordinary .NET class, which is declared abstract and derived from the base class `Aspect`. Aspect classes are organized in assemblies like other ordinary .NET libraries. No language extensions are necessary to specify aspect code.
The implementation of an aspect consists of

- the (abstract) declaration of an interface expected from any core class the aspect later should be weaved in and
- the implementation of the aspect's methods that correspond to "around" advices in AspectJ.

The expected interface of an aspect consists of abstract methods that can be separated in two categories marked by the attributes "*IsRequired*" and "*IsExtended*".
Both types of methods and their signatures represent a kind of contract between a core code class and an aspect class. Concrete aspect method implementations use the expected interface and the core code has to supply this interface. Calling an expected method within the aspect implementation results in a call to the respective core method at runtime.

"*IsExtended"*' abstract methods are a substitute of the core methods that should be extended/surrounded with some additional aspect code. An "*IsExtend"* method signature requires a concrete '*Extends_**' method implemented in the aspect. This "*Extends_**" method "aspectizes" the core method its "*IsExtended*" equivalent stands for. Any runtime call to the core method results in invoking the corresponding "*Extends_**" method in the aspect code first. The aspect code then usually calls the core method using the signature of the abstract "*IsExtended*" method, which results in a call to the concrete core method at runtime.

"IsRequired" methods of the aspect aim at interacting with 'aspectized' core classes without breaking encapsulation. Calling such a method inside a concrete aspect method results in a call to the equivalent core method. This allows the aspect code to retrieve information from the core objects without knowing the core classes concrete implementation at development time, thus increasing the flexibility of the aspect implementation.

The challenge for the developer is to provide some interaction mechanisms between aspect and core code, which is hopefully possible by using '*IsRequired*' methods.

The example in figure 2 shows an (very simplistic) aspect *BoundsChecker* that should work on graphical elements of a core system. Its goal is to 'aspectize' any move operation of a graphical element in horizontal dimension in order to prevent it from exceeding the canvas' bounds, the maximal value the x coordinate should have is MAX_X The aspect guarantees that the value of the x coordinate always is small enough that the value added in a shift operation never exceeds MAX_X. The concrete "*Extends_ShiftX*" operation surrounds the core methods by calling "*ShiftX*" within its implementation.
The core code is an implementation of graphical elements not shown here. An XML-connecter specifies the relationship between aspect and core code. This relationship consists of the mapping between the "*IsRequired*" and "*IsExtended*" method names of the aspect to the concrete method names of an application and the mapping of method parameters.

Sometimes it is useful to feed an "*IsExtended*" method with values for improving the aspect functionality. (cf. Pointcut Parameters in AspectJ [Kiczales et.al. 01]) In figure 2 the method "*ShiftX*" takes an argument *xValue*, which refers to an argument of the same type of the "aspectized" method at runtime. The parameter is used for a bounds check inside the aspect code. It gets added to a value retrieved by calling the "*IsRequired"* method "*GetX()"*.
The implementation of "*Extends_ShiftX"* also shows that code can be inserted before and/or after the call to the core code method. So the behaviour is like an around advice in AspectJ [Kiczales et.al. 01] and should provide greatest flexibility.

```
public abstract class BoundsChecker : Aspect
{
    //required application-code interface
    private const int MAX_X=100;
    [IsRequired]
    public abstract int GetX();
    [IsRequired]
    public abstract void SetX(int x);
    [IsExtended]
    public abstract void ShiftX(int shiftFor);

    //implementation of aspect-functionality
    public void Extends_ShiftX(int shiftFor)
    {
      //before
      bool isOutOfBounds;
      if((GetX() + shiftFor > MAX_X ){
          SetX(MAX_X - shiftFor);
      }
      ShiftX(xValue); //call to core code-method
      //after
      //...
    }
}
```

Figure 2: Aspect implementation in C#

The *BoundsChecker* aspect expects that the core application class provides at least three methods, one returning an integer, and two returning void and having at least one parameter of type integer. The latter will be the target of the "*Extends_ShiftX*" method of the aspect. Of course the developer who specifies the connection between core code and aspect must know the semantic of the expected methods to be able to provide a useful mapping.

To sum up *"IsRequired"* and *"IsExtended"* methods deal with different directions of the aspect and core code interaction. Each *"IsExtended"* method represents a core method whose functionality is decorated by additional concerns. In fact the Decorator pattern helps to conceive the control-flow when invoking a core method, which results in calling its relevant aspect methods first. "IsExtended" methods result in a call into the aspect, *"IsRequired"* method calls within aspect methods on the other hand temporarily redirect the control-flow back to core code (See [Herrmann et.al.])

3.3 Connector

The connector describes in XML how aspect and core code methods are mapped on each other. On the one hand there are the implemented aspect methods, which require certain methods in the core code. On the other hand there is the core code that doesn't know anything about aspects. The connector bridges the gap between these two.
It describes which aspect method should act on which core code method. As already mentioned the aspect methods with the "*IsExtended*" attribute are allowed to have parameters. The aspect parameters can be a subset of the core code method parameters and the mapping between core code method and aspect method parameters is also described in the connector. Each aspect method parameter needs one corresponding core method parameter, and the type of the aspect method parameter must either match or be a super type of the corresponding core method parameter. This improves the reusability of aspects and limits at the same time the dependency between aspect and core code to a minimum.

3.4 Runtime behavior of aspects

Aspects can come in two flavours; they either can be instantiated for every object of a class they are connected or weaved to, or they can be singletons. The difference is that the aspect's member variables either are created per object or, as the name singleton indicates, only once for the whole core application.

One of the new concepts mentioned earlier is the enabling and disabling of aspects at runtime, thus changing the behaviour of methods due to the context or set of aspects that apply to them at every specific moment during execution. Switching on and off isn't useful for every kind of aspect. For example a security aspect might apply to the whole core application runtime without the possibility to be switched on and off. Therefore such an aspect can be defined as context independent. They are like static variables, created at program start-up and destroyed at program shutdown. The *AOPEnvironment* offers an interface for enabling and disabling aspects. The core code can make use of this interface. All the aspect managing stuff can also be put into a context independent aspect that gets "instantiated" automatically, so that the core code isn't aware of AOP stuff at all. This approach reflects the nature of aspect management being a crosscutting concern itself. Further programming work and experience, which combines OO development and AOP mechanisms should decide on the usability of these concepts.

3.5 Developing in AOP# for .NET

It's now time to get the whole picture by describing how applications are developed in AOP# and what happens at runtime. The core code can be implemented without being aware of the factored out crosscutting concerns. Similarly the aspects needn't be aware of the business code they later will be applied to. The mapping between those two is done in the connector description.

From a programmer's point of view the work is now done. But the implemented aspects are still abstract classes. They are missing code that connects them with the *AOPEnvironment*. A tool, which parses the XML connector file and the aspect code, generates concrete C# classes. So the *AOPEnvironment* is able to call aspect methods and can get information from the aspects back.
A second tool, which uses the profiling functionality from .NET and the IL generation feature, generates the entry hooks into the *AOPEnvironment* for all "aspectized" core code methods every time a method is loaded for the first time. The profiling API of .NET is able to intercept the just-in-time compilation event before a core code method is compiled into native code. With the metadata API it is possible to insert the necessary hook into the *AOPEnvironment*. "Hooked" core methods yield control to the *AOPEnvironment*, which is able to decide if an aspect should be processed or not. This decision depends on the information in the XML-connector and the set of active aspects. After processing the aspect methods and the core method, the control-flow returns to the core application and the game can start again. From a pattern point of view the resulting connection of aspect and core code consisting of a "hook" and the invocation of the AOPEnvironment conforms to the Interceptor pattern.

3.6 Made for Change

The design of the AOP system for .NET is kept lightweight and its components as independent of each other as possible.

They should be easy to change and evolve according to the feedback of the developers who use them. But this lightweight approach makes it hard to provide tools that are properly integrated into the .NET environment, which again will unfortunately lead to lower acceptance among developers. The discrepancy between keeping the mechanism easy to change during an exploration state on the one hand and involving many developers to get the experience in which direction the system should be evolved on the other can't be solved easily.

3.7 Summary

A first prototype of the AOP system for .NET is just under development, but since all relevant concepts are taken from existing successful approaches we expect it to be quite useful from the beginning.

The reasoning and implementation of the AOP system for .NET lead to several statements we mentioned in this paper. First it gives evidence to our statement that AOP systems often relate to patterns and their proposed solutions. Knowing the underlying patterns (e.g. the Interceptor or Decorator pattern) helps to understand the AOP system and to design aspects according to the AOP system concept.
Secondly the AOP system for .NET tries to deal with major challenges of Aspect Oriented Programming like the interaction of aspect methods with core code and aspectual polymorphism.

But the main goal is to help widen the user community by inviting the traditional C++ or Visual Basic developers to contribute to aspect oriented development. Additionally we hope to gain some experience with the rather new concept of "aspectual polymorphism", which origins from and is explored for Java through the work of Klaus Ostermann and Mira Mezin in Minos [Mezini et.al.].

4 Acknowledgements

We thank all researchers who explored the idea of cross cutting concerns to a state we industrial researchers can build on, especially the group around AspectJ, Hyper/J and Minos and our colleagues in Siemens who gave valuable feedback.

5 References

[Akkawi et.al.] Faisal Akkawi, Atef Bader, Tzilla Elrad. Dynamic Weaving for Building ReconfigurableSoftware Systems. Workshop on Advanced Separation of Concerns at OOPSLA2001

[Alexander1977] Christopher Alexander. A Pattern Language. Towns Buildings Construction. Oxford University Press, New York, 1977

[AspectJ] AspectJ™ web site: www.aspectj.org

[AspectJ FAQ] http://aspectj.org/doc/dist/faq.html

[Baniassad] E. Baniassad , G. Murphy, Christa Schwanninger and Michael Kisrcer. Managing Crosscutting Concerns During Software Evolution Tasks: An Inquisitive Study. Submitted to the 2nd

[CCM] Interim FTF Report of the Components December 2000 Finalization Task Force to the Platform Technical Committee of the Object Management Group November 3, 2001 Document Number:ptc/2001-10-26

[Eisenecker et.al.] U.Eisenecker, D. Weber. Aspektorientiere Programmierung. Talk at Advanced Developers Conference in Hannover, Germany. 2001

[Elrad et.al.] T. Elrad, R. Filman, A. Bader. Aspect-Oriented Praogramming. Communications of the ACM, Vol. 44, No. 10. October 2001.

[GOF] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, 1995

[Herrmann et.al.] S. Herrrmann, M. Mezini. Aspect-Oriented Software Development with Aspectual Collaborations. Submission to ECOOP 2002.

[Hyper/J] Hyper/J™ web site: www.research.ibm.com/hyperspace/HyperJ/HyperJ.html

[Kiczales et.al 97.] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda and C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect Oriented Programming.* In *Proc. of European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science* Vol. 1241, pp. 220-242, 1997.

[Kiczales et.al. 01] G. Kiczales. E.Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. An overview of aspectj. In Proc. Of 15th. ECOOP, LNCS 2072, pages 327-353, Springer-Verlag, 2001

[Mezini et.al.] Mira Mezini, Klaus Ostermann, Object Creation Aspects with Flexible Aspect Deployment.

[Noda2001] N. Noda, T.Kishi, Implementing Design Patterns Using Advanced Separation of Concerns,.Workshop on Advanced Separation of Concerns at OOPSLA2001

[Pichler et.al.] R. Pichler, K. Ostermann, M. Mezini. On Aspectualizing Component Models

[ShFeSe2002] Dharma Shukla, Simon Fell and Chris Sells. Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse. MSDN article March 2002 – Vol. 17 No 3

[Sun] Sun Microsystems. Enterprise JavaBeans Specification, Version 2.0. 2001

[Tarr et.al.] P.Tarr, H. Ossher, W.Harrison and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In proceedings of the 21st International Conference on Software Engineering (ICSE'99), 107-119, May 1999.

[Teichert]G. Teichert-Matuschek. Thinking in Aspects – Aspektorientierte Entwicklung ohne aspektorientierte Sprachen. OOP 2002

Promoting Component Reuse by Integrating Aspects and Contracts in an Architecture Model

Patrice Gahide, Noury Bouraqadi
Ecole des Mines de Douai
Dept. G.I.P.
941 rue Charles Bourseul - B.P. 838
59508 Douai Cedex - France
{gahide,bouraqadi}@ensm-douai.fr

Laurence Duchien
USTL-LIFL
Bâtiment M3
59655 Villeneuve d'Ascq Cedex - France
Laurence.Duchien@lifl.fr

ABSTRACT

In the last years, Component-Based Software Engineering (CBSE) has emerged in an attempt to establish engineering principles for building software with reusable parts. Contract-based approaches represent a good way to extend specifications of both components and interactions among components. Likewise, Aspect-Oriented Programming (AOP) promotes software reuse by achieving separation of the different concerns that appear in a system. However, few efforts have been made to merge AOP with CBSE and contract approaches. In this paper, we propose to integrate contracts and aspects within an architecture model based on the terminology used in Architecture Description Languages (ADLs).

Keywords

Reuse, Component, AOP, Contract, Architecture Model

1. INTRODUCTION

With its use for e-business, Component-Based Software Engineering (CBSE) tends to become standard to build large distributed applications. Various component-based implementation frameworks appear, including EJB, COM and CCM. CBSE promises to be a way to produce high quality software fast, with less effort, by composing highly-reusable components. Particular languages, called Architecture Description Languages (ADLs) [11], emerge to help software architects building a system out of multiple components, by providing models, notations and tools to describe components and their interactions. However, there is still a gap between the ideal vision of "plug-and-play" components and the actual achievements in this domain. Attempts at developing and using Commercial Off-The-Shelf (COTS) components, although successful in well-defined application domains, show some limits: composition of COTS components provided by different parties is far from being as trivial as connecting a keyboard to a computer. The question that arises is: what should an architecture model provide to promote reusable component design and composition? To answer this question, we first need to identify barriers to reuse in existing models.

The first limitation is due to the traditional "black-box" approach: components designed within the black-box abstraction only expose a functional interface (i.e. provided services). Yet the component composition problem highlights the need to clearly express all component properties and relationships between components and the underlying structure, and among components. Functional interfaces are not suitable for that purpose. The CBSE community widely supports this statement, and the question of identifying the best way to address this issue led to the concept of *contract* [5, 3]. However, no consensus has been reached with regard to the way contracts have to be specified and used in the composition process.

Another issue is the difficulty to achieve separation of concerns [6]. A whole system reveals global non-functional properties that tend to "cut across" components (the system functional parts). Examples of non-functional properties include distribution, synchronisation, persistence and scheduling. Those are usually hard-coded in each component, resulting in the so called "code-tangling problem" [8]. Therefore, a software architect has no way to adapt components according to the non-functional requirements of the application. As a result, the potential of composition exposed by these components is significantly reduced. Aspect-oriented programming (AOP) [8] is a paradigm that aims at achieving this separation of concerns in a clean way.

We claim that the aspect paradigm, coupled with the concept of contract, could be of great help in designing and composing reusable components. Our approach provides a way to contractually specify and to adapt a component in accordance with non-functional aspects. This dismisses the classic vision of the component as a black-box entity interacting only through its conventional interface. In that manner we join the Open Implementation approach introduced by Kiczales [9], which defines an additional interface to a module, allowing one to change its non-functional behaviour. We argue that this idea of customization should even be extended to all entities that compose a component-based architecture.

In this paper, we first outline existing work related to concepts exposed above. Next, we present our architecture model for aspect integration. Finally, we conclude giving some perspectives and guidelines for future improvement of this model.

2. RELATED WORK

2.1 Contracts

As we saw before, the concept of contract is bound to play an important part when trying to address the requirements of component documentation, co-dependency and composition. Beugnard et al. [3] introduce component contracts as a means to describe a component at four hierarchical levels: syntactic, behavioral, synchronisation, and Quality of Service (QoS). By analogy with the AOP terminology, the syntactical and behavioral levels express the functional properties of the component. The former corresponds to the interface specification, which we know to be inappropriate to describe non-functional aspects. The latter deals with the effects of operation calls, through the use of invariants, pre- and post-conditions [13]. The two higher property levels attempt to cover non-functional parts. The third level concerns synchronisation protocols in a concurrent context. Finally, the fourth level describes QoS, such as maximum response delay or precision. Note that there is neither mention of how to describe such contracts, nor of the way the different levels interact with each other.

Andrade and Fiadeiro propose *coordination contracts* [2] that define a set of constraints and rules on the interactions between components. Such contracts involve a predefined number of participants (i.e. components). Each coordination rule describes a behaviour that is superposed on a particular interaction. Coordination contracts are exclusively defined at the inter-components connection level. Therefore, adaptation is achieved by changing the coordination contract without having to adapt the components. The authors suggest a design pattern for implementing coordination contracts. In this pattern, participants are reduced to black-boxes, and thus are not entitled to facilitate any composition process initiated by a third party.

Another contract-based approach to the composition issue is the one proposed by Wydaeghe [16], which is also a role-based approach. Components come with a documentation describing scenarios of interaction for the supported functionalities. Each scenario is defined by a special kind of Message Sequence Charts (MSC), an extension of UML sequence charts, that uses a small set of primitives instead of API calls to model the component's behaviour. MSC are also used to model composition patterns (role interactions). Composition patterns and component documentation provide a way to perform compatibility checking between a role and a given component at the time of the composition phase.

2.2 Architecture Description Languages

Architecture Description Languages (ADLs) [11] specify common notations for formally representing software architectures. The notion of contract is merged into an ADL. For instance, in ADLs, a component specification typically includes the "provides" and "requires" clauses (i.e. the provided and required services). To cover different paradigms of interconnection, ADLs come up with the connector abstraction [12], thus allowing different types of connectors (e.g. pipes, RPC, events...) to coexist in an architecture. In this context, a *configuration* consists of a set of *components* and *connectors* linked together in a well-specified way. This clear separation between components, connectors and configurations is a basic concept shared by all ADLs. However, despite a large popularity among the research community, no ADL has yet reached common practice. Moreover, no ADL exposes particular facilities to achieve AOP goals.

2.3 Separation of concerns

Various approaches have been proposed for a few years supporting separation of concerns. Although each of these techniques addresses the problem in accordance with different views, they converge at an abstract level towards a common goal: the modularization of different concerns of a program that cannot easily be modularized using traditional decomposition units (procedures, classes...). The different approaches thus differ in how they define additional concerns, and how they support their composition with the functional program units. Program transformation techniques such as AspectJ [7] allow programmers to define aspects as first-class entities and to weave them with the functional units. Adaptive Programming is a subset of Aspect-Oriented Programming that promotes the decoupling of algorithms from class hierarchies [10]. The composition filter approach [1] attempts to model various aspects of a program as a set of filters applied to messages exchanged by objects. Other projects, such as JAC [14], propose a framework to support dynamic aspect application to objects.

All of the above techniques are in constant evolution. None of them currently integrates direct support for component technology, especially as far as the component composition problem is concerned. A notable exception is Enterprise Java Beans (EJB) [4], the component framework from Sun Microsystems. EJB model allows some level of separation of concerns: application functionalities are defined using *beans* (components), while some non-functional properties are handled by beans containers. However, EJB is too rigid and too restricted to allow adaptation and extendable use of aspects.

3. TOWARDS AN ARCHITECTURE MODEL INTEGRATING ASPECTS

We now present the definition of our architecture model that integrates aspects and contracts. For our purpose, we adopt a terminology similar to ADLs, that makes a distinction between *components*, *connectors* and *configurations*. This leads us to set out our own definitions of these terms. Concerning the term "component", for example, there is no shortage of definitions of it in the literature. This is not surprising: different approaches to analyze and solve problems invariably lead to different definitions of the constituent parts of solutions (components). Even within the component-based software engineering community there is considerable variation in definitions of component, although those variations are usually quite subtle. However, the CBSE community seems to gather around the definition proposed by Szypersky [15], and our own definition is based on this one. Figure 1 presents a view of our architecture model. The remains of this section outline the

elements which constitute this model and the way they interact.

> A ***component*** is a *contractually specified* unit of composition, that exposes a set of *provided services interfaces*, may have *required services*, includes *entry points*, and is subject to *composition* by third parties.

The *provided services* correspond to the component functional part. In order to provide these services, a component may require services from other components. As we saw before, a syntactic description of services is essential but not sufficient to provide software architects with the knowledge needed to build component-based systems successfully. Consequently, a component must include a contract that specifies extended informations such as memory requirements, response times and behavioural description. In short, the contractual specification must describe the conditions that must hold for that component to work properly. It can be seen as a pattern of interaction between the component and its environment, but rooted on that component. The contract specification should be written in a formal (or semi-formal) way, so that validity of a given composition can be checked automatically.

The component *entry points* allow to adapt the non-functional properties of the component. For example, if the software architect wants a component to store its data on a permanent medium, she/he can specify it via the entry points. Clearly, the change has a local effect on the internal component behaviour, but it could also affect the external view of the component, namely its contract. In the previous example, persistence application could have some repercussions on the component contract in at least two ways. On the one hand, fixed or average response times increase. On the other hand, this component has more storage requirements. The former concerns clients of that component while the latter affects the deployment strategy.

This breaks the traditional view of the contract as a rigid specification. Components come with a base contract which is exposed to refinements as soon as modifications on the component's non-functional behaviour are applied. This is another argument in favour of a formal specification of component contracts: this is a necessary condition to achieve automatic contract transformation.

We carry on with the definition of a *connector* in our architecture model:

> A ***connector*** is an entity that *connects* two or more components together in a *contractually specified* way, and that includes *entry points*.

Connectors provide a convenient way to separate issues concerning component behaviour from inter-components interactions. For example, a connector can isolate the particular communication mechanism used to transfer messages among components (e.g. sockets, RPC...). Just like a component, a connector exposes its contract. A connector contract specifies QoS properties, such as transfer time or bandwidth, but could also include semantics of interaction among components. We do not exclude the possibility for a connector to link more than two components together. The connector contract could for instance specify the routage policy used to determine which component receives a message, when two or more components linked to the connector provide the same service.

Moreover, entry points to the connector are provided to enable behavioral adaptation: changing the routage policy in the above example is done by acting on the entry points of the connector. Again, connector adaptation leads to contract modification.

We have not defined the entities that act on components and connectors entry points yet. As one could guess, these entities correspond to aspects:

> An ***aspect*** is the definition of a *transverse property* (usually non-functional) and can *act on the entry points* of components and connectors.

For example, the entity that acts on components to include some persistence properties is obviously a persistence aspect. Defining the routage policy at the connector level is one of the responsibilities of the load balancing aspect. However, we point up the fact that the so called "coordination contract" (in the sense of Andrade et al. [2]) is a particular aspect in the context of architecture description. Interactions defined by this aspect are part of the functional properties of the overall system. Therefore, coordination can be seen as a *transverse functional aspect*. Our model allows the software architect to specify the coordination dimension as well as component assembly, through the definition of an overall coordination contract. Such a contract could be expressed with a dedicated language. Aspects and coordination contract rules and constraints are spread among components and connectors at assembly time by acting on entry points. Since the coordination contract is a functional part, aspects should be allowed to act on it just as on components or connectors.

The final element of our architecture model is the configuration:

> A ***configuration*** is a *set of components* bound with *connectors*, submitted to a *coordination contract* and possibly other *aspects*. A configuration is *self-sufficient* and provides an *interface* for the services it offers. It may include *entry points*.

The configuration clearly reveals the binding of components through connectors in a single overall entity. A configuration is *self-sufficient*, i.e. at this level it is a ready-to-use system through the *services interface* (see figure 1). Therefore, we choose to place the coordination contract and aspects inside the configuration. Those are defined as first-class entities, contributing to maintain separation of concerns during the configuration life cycle. This is because we don't exclude the possibility to reuse a configuration in a new composition process. To enable this feature, a configuration may include entry points: they allow to act on the configuration local aspects. Such actions should involve addition, replacement, and possibly adaptation of aspects. Besides, configuration reuse also requires specification of configuration contract. We end up with

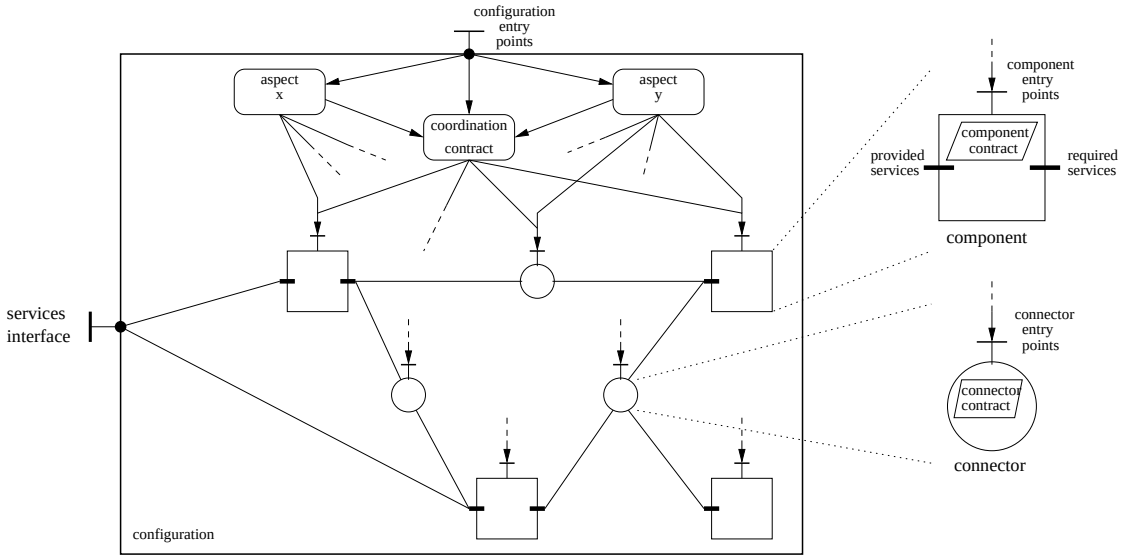


Figure 1: The architecture model

a configuration that gets closer to what a component is, and the underlying question emerges: can we consider a configuration as a component?

4. CONCLUSION AND FUTURE WORK

This model is definitely early work in progress. Our objective is to simply outline a possible direction for the solution of quite a complex question: how can the aspect-oriented approach be integrated, and how helpful can it be revealed in an architecture model. We believe that an incremental approach, where minimal, first-principles properties are first identified and then restrictions and properties are progressively added, is a good way to arrive at a clear, unambiguous foundation for an architecture model.

Our model points up two distinct senses of contract: one where the subject is the component or the connector (local contracts), and the other where the subject is the whole set of components and connectors (global coordination contract). Formal or semi-formal foundations are welcome to specify contracts. Non-functional aspects are plugged on the entry-points of components and connectors thus allowing non-functional behaviour and contract adaptation. The questions of how to represent these aspects and the way they act remain open issues yet. Besides, the possibility of configuration reuse must be examined deeper: if the configuration can become a reusable entity in our architecture model, then it becomes a component, and as such must expose a contract. Moreover, aspects and coordination contract adaptation through configuration entry points reveals the problem of inter-aspects relationships.

One possible track of research could be to examine the possibilities of integrating aspects as first-class entities in ADLs, and to consider each issue in this well-defined context. This constitutes the main part of our future research.

5. REFERENCES

[1] M. Aksit, L. Bergmans, and S. Vural. An Object-Oriented Language-Database Integration Model: the Composition-Filters Approach. In O. L. Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 372–395, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[2] L. F. Andrade and J. L. Fiadeiro. Contracts: Supporting Architecture-Based Evolution. submitted.

[3] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

[4] G. Blank and G. Vayngrib. Aspects of Enterprise Java Beans. *Proceedings of ECOOP'98 Aspect-Oriented Programming Workshop*, 1998.

[5] I. M. Holland. Specifying Reusable Components using Contracts. In O. L. Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 287–308, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[6] W. Hürsch and C. V. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, Massachusetts, 24 Feb. 1995.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda,

C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP'97*, LNCS 1241, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[9] G. Kiczales and X. Parc. Beyond the Black Box: Open Implementation. *IEEE Software*, 13(1):8–11, 1996.

[10] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-Oriented Programming with Aspectual Methods. Technical Report NU-CCS-2001-01, College of Computer Science, Northeastern University, Boston, MA, Feb. 2001.

[11] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Jan. 2000.

[12] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 178–187. ACM Press, June 2000.

[13] B. Meyer. Applying Design by Contract. *IEEE Computer (special issue on inheritance and classification)*, 25(10):40–52, 1992.

[14] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: a Flexible Solution for Aspect-Oriented Programming in Java. *Lecture Notes in Computer Science*, 2192, 2001.

[15] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[16] W. Vanderperren and B. Wydaeghe. Separating Concerns in a High-Level Component-Based Context. In *Proceedings of ETAPS'02 Workshop on Software Composition*, Apr. 2002.

Exploiting the possibilities of "weave-time" aspects in the creation of component-based ecological models.

Position Paper: First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software

Douglas R. Dechow
Computer Science Department
Oregon State University
101 Dearborn Hall
Corvallis, Oregon 97331
dechow@cs.orst.edu

Abstract

Current aspect weavers operate at compile time or runtime. This paper introduces a temporal position that occupies the space between the two: weave-time. Software applications that are dependent upon human interaction—such as simulation development environments—potentially offer system latencies that can be exploited for the weaving of aspects. By alleviating the modeler of such entangling concerns as logging/tracing, an aspect-oriented model development environment would allow modelers to focus more fully on the simulation that is at hand. This paper provides a preliminary description of an aspect-oriented infrastructure and its associated aspect weaver.

Keywords

Separation of concerns, common intermediate language (CIL), preprocessors, aspects, aspect-oriented programming, aspect weaver, compile time, runtime, weave-time, components, ecological modeling.

1. Rationale

A distinct problem associated with introducing a new software simulation system to the world of ecological modeling is that many would-be users are not proficient programmers. Mechanisms such as tracing and logging can be used to provide feedback to model developers. Additionally, the ability to monitor and evaluate the state variables as the simulation evolves would be useful. However, such mechanisms are not salient features of the ecological model itself. As such, they can be seen as orthogonal to the model and a potential source of tangled code. Using aspect-oriented approaches to automate the control of these concerns can remove this burden from the modeler while simultaneously making the features available to provide the necessary feedback.

ModCom is a component-based simulation tool for distributed ecological modeling [10]. ModCom is an object-oriented application framework (C++), and it is built on the Microsoft Component Object Model (COM). Although a visual development environment for ModCom is under development, current users of the framework must write their simulations in C++ or Visual Basic (VB developers are, potentially, a large part of ModCom's user base).

An aspect-oriented infrastructure to interoperate with ModCom is currently under development. The AO infrastructure will be built upon the .NET Common Language Runtime (CLR)[9]. ModCom will execute as a "C++ with managed extensions" [11] application.

Current aspect-weavers operate at compile time (AspectJ)[2] or runtime (AOP/ST, AspectS) [1][5]. The aspect weaver that is being developed for ModCom will operate in the region between the two.

In the strictest sense, the weaver will be a compile time weaver. However, by making use of multiple translation model and the runtime facilities (specifically the class loader) of the CLR, it will appear to the user as if the weaving of aspects is occurring at runtime. For the purposes of this paper, this intermediate time frame will be referred to as *weave-time*.

It should be noted that weave-time is mentioned in Boellert [3] but not defined. Its usage in this paper is somewhat different than what is implied in Boellert's paper [3].

2. Objectives

The goal of this project is to design and implement an aspect-oriented infrastructure that interoperates with the ModCom ecological simulation tool. The aspect-oriented infrastructure can then be used to instrument the simulations in order obtain feedback. This knowledge will be used to help inexperienced programmers to build simulations.

Additionally, the project will offer opportunities to evaluate potential aspect reuse techniques in the form of the tracing and logging aspect libraries. Given the inherent multi-language support that is present in the CLR, the development of multi-language aspect libraries will be examined.

Initially, the needs of the ModCom user base require that the OO CLR languages—C#, VB, VC++--will be targeted. Support for the non-OO CLR languages will be investigated as the project moves forward.

Finally, the visual tool—which is intended to be a fully integrated part of the infrastructure—will offer a novel method for evaluating aspect integration techniques.

3. Approach

Many of the architectural and implementation features of the project were predetermined by virtue of the fact that ModCom is an extant system. In light of this, building the aspect-oriented infrastructure on the .NET platform was the path of least resistance.

The core of an aspect-oriented infrastructure rests upon the system's ability to accomplish three tasks: "a join point model, a means of identifying join points, and a means of affecting implementation at join points."[7] The key features of the research project that provide this functionality are outlined below.

3.1 Aspect weaver

The aspect weaver will be a source-to-source preprocessor designed to manipulate the Common Intermediate Language (CIL) [4] of the .NET CLR. In the final paragraphs of section 1, it was mentioned that the CLR supports a multiple translation model. The high-level language compilers of the CLR all emit CIL. This is the first compilation/translation step. Next, the CIL is compiled

into native machine code by the CLR's JIT compiler. [12] It is during this second, runtime compilation that loading and linking take place.

Weave-time is the time frame that exists between the translation of a high-level language to CIL and the secondary JIT compilation of the CIL into machine code. Weave-time takes place under the direction of the user. Hence, the notion of weave-time can be seen to be operating apart from the loading and linking that takes place in the CLR's runtime engine.

Initially, a subset of the aspect model promulgated by AspectJ [7] will be targeted as the design point. As was mentioned above, the CLR supports a wide variety of object-oriented, functional, and logic-based programming languages. In addition to Microsoft's own .NET languages (C#, JScript, Visual Basic, and Visual C++), compilers that target the CLR exist for Eiffel, Cobol, Standard ML, Mercury, Scheme, Haskell, and Smalltalk [6]. Undoubtedly there are others.

Since all of these compilers emit the CIL that the aspect weaver is designed to manipulate, this will provide an intriguing opportunity to investigate the use of multiple languages (and language paradigms) in an aspect-oriented environment.

3.2 Aspect Libraries

An integral part of this project is identifying and developing the aspects themselves. Initially, this effort will focus on the creation of aspects that can aid the users of ModCom in developing simulations.

Eventually, aspect libraries for tracing and logging will be developed. The use of these aspects in other contexts will be investigated.

3.3 Aspect-oriented infrastructure

After the development of the aspect weaver and the aspects themselves, the final two pieces of the infrastructure are an architecture and a visual tool.

The architecture, or framework, will serve as a focus or collecting point for unifying all of the pieces of the infrastructure.

The visual aspect integration tool will be a modeler's primary means of interacting with the infrastructure. The tool will allow the users to select join points in their models and aspects from the libraries. The visual tool will be designed so that the user will have no explicit knowledge that they are weaving aspects into the model.

3.4 Weaving an aspect

Lastly, what follows is an event trace that provides a high-level description of the functioning of the aspect-oriented infrastructure during the weaving process.

1. The modeler uses the visual tool to select a join point/crosscut.
2. The modeler uses the visual tool to select an aspect for weaving.
3. The running simulation object(s) are serialized then unloaded.
4. ILDASM generates CIL source file from assembly.

5. IL aspect weaver weaves in aspectual concerns.
6. ILASM generates new assembly.
7. The CLR loads/verifies the new assembly.
8. The simulation object(s) are instantiated from their persistent forms.

Clearly, this is just first-cut approximation intended to prototype a functional system. In the future, transformation of the assemblies directly via the use of metadata and reflection (and possibly the class augmentation process) [8] offers a more efficient alternative to the use of ILDASM and ILASM.

4. Ongoing Work

The development of ModCom has been ongoing for approximately 24 months. It has matured to the point that its first general release is imminent.

This development of the aspect-oriented infrastructure that is proposed in this paper is in its preliminary phase. Prototyping of the aspect weaver and aspect libraries is beginning. Work on the overall system framework and the visual integration tool should be underway by June of this year.

After the system described in this paper is implemented, other potential areas of the ModCom framework will be evaluated. Another area of ModCom that could benefit from an aspect-oriented approach is persistence.

5. References

[1] K. Boellert. The AOP/ST homepage, http://www.theoinf.tu-ilmenau.de/~kaib/aop/.

[2] The AspectJ homepage, http://www.aspectj.org.

[3] K. Boellert. On Weaving Aspects. Position paper at the ECOOP '99 workshop on Aspect-Oriented Programming, 1999.

[4] J. Gough. *Compiling for the .NET Common Language Runtime*. Prentice-Hall, 2001.

[5] R. Hirschfeld. AspectS—AOP with Squeak. *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. 2001.

[6] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Conference on Programming Language Design and Implementation 2001*. ACM, Snowbird, Utah. 2001.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59-65, October 2001.

[8] S. Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.

[9] E. Meijer, and J. Gough. Technical Overview of the Common Language Runtime. Available at http://research.microsoft.com/~emeijer/.

[10] The ModCom homepage, http://biosys.bre.orst.edu/ModCom/.

[11] The .NET Common Language Runtime homepage, http://msdn.microsoft.com/net.

[12] J. Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.

Runtime Weaving of Aspects using Dynamic Code Instrumentation Technique for Building Adaptive Software Systems

Dangeti Thirunavukkarasu Jeyabal

Software Solutions Lab
Honeywell Laboratory
{srinivasarao.dangeti, thirunavukkarasu.ramasamy, jeyabala.murugan}@honeywell.com

ABSTRACT
Aspect Oriented Programming (AOP) methodology provides separation of crosscutting concerns across the multiple modules in the software system. These concerns can be weaved, into the application modules, either at compile time or at runtime of the application. Deferring the aspect weaving to runtime, provides greater flexibility in choosing the right kind of aspect to be weaved. Software Systems can exploit this flexibility in providing Intelligent Adaptiveness into the system.

In this paper, we try to describe how runtime weaving of aspects are possible, in non-interpreted languages, using the Dynamic Code Instrumentation Technique. We also discuss the benefits that can be reaped for building Intelligent Adaptive Systems.

1. INTRODUCTION

Separation of crosscutting concerns, using Aspect Oriented Programming (AOP) [1], in software systems gives a new dimension to the way we look and think of the systems we build. These concerns allow us to separate out the modules, which are fairly independent by themselves. Concern modules are developed independently and weaved into the system either at the build time or at the runtime.

Once concern modules are separated, architects can think of designing the core application modules in the initial phase and think of introducing these concern modules in the later part, seamlessly. Developers can concentrate on building different concern modules and leave the integration of the concerns with the application to the aspect weaver. These concern modules can even be reused across different applications.

Weaving the concern modules with the application modules at build time transforms it into a single monolithic application, where we have very less control on managing the concern modules. In addition to adding new concern modules at runtime, even altering and removing will be practically impossible. Runtime weaving gives the flexibility of managing the concerns at the runtime of the application. It overrides all the restrictions laid by build time weaving.

Most of the current implementations of AOP like AspectJ [2], AspectC++ [3] are targeted towards static weaving of the aspect modules into the application modules. These implementations may well suit applications that clearly know all the possible aspects before hand, and these aspects are fixed through out the application's life. These implementations make the following assumptions.

- All the aspects are identified at design time and there will not be any requirement for new aspects to be defined at runtime.
- The Aspect code need not be modified.
- Aspects need not be inserted and removed at runtime.

Very few implementations of AOP have provided runtime weaving, that too using only interpreted languages like Java. Interpreted languages transform the code into machine independent intermediate form (in case of Java it is byte code), before transforming to machine-level instructions. This intermediate code gives runtime weaving a greater flexibility in manipulating the application modules with concern modules. Even late loading or the loading on demand of classes provided by these interpreted languages in the form of class loaders helps in manipulating the code.

In non-interpreted languages the compilers convert the code into machine-dependent binaries at compile time. Binary image is loaded into the memory, once the application starts executing. The manipulation of this binary in memory poses greater challenges to the runtime weaver of concerns. Dynamic Instrumentation provides a technique to manipulate application code at runtime by working on the binary image [4]. It allows adding arbitrary snippets of code into different points in the application. With this technique, realizing runtime weaving of concerns is very much feasible. AOP with runtime weaving can be applied to a wide range of areas like intelligent adaptive systems, dynamic configuration of system, process monitoring and diagnosis, self-healing systems, evolutionary systems, etc. In the following sections, we shall discuss how adaptability can be brought into the system using runtime AOP.

Adaptive Software Systems senses the changes in the environment and from within, and reacts to these changes accordingly. These systems need to continuously monitor the changes in the surrounding environment and from within. Depending upon the sensed input they bring back change into the system in order to adapt to the changing conditions. The changes need to be brought smoothly without compromising on system behavior and system integrity. Real-Time and mission critical systems need to have dynamism built in as part of the system itself. Real-Time systems cannot be brought down for making minor changes into the system, so also with mission critical systems. Building dynamic adaptability into the system poses a greater challenge to researchers and developers.

The rest of the paper is divided as follows. In section 2 and 3 we will have a quick overview of Aspect Oriented Programming and Dynamic Code Instrumentation Technique respectively. In section 4 we will discuss the implications of runtime aspect weaving and see how dynamic instrumentation is used for aspect weaving. The role of Aspect Oriented Programming in building Intelligent Adaptive Systems is discussed in section 5. Section 6 summarizes the paper and gives the future work.

2. SEPARATION OF CONCERNS IN AOP

Software systems consist of several concerns that crosscut multiple modules. Typically, these concerns include logging, data persistence, security, transaction integrity, debugging, resource pooling, synchronization, etc. These concerns are termed as *crosscutting concerns* (also called system level concerns), as these crosscut different modules in the system. Current techniques, like Object Oriented Programming, do not provide an efficient way to represent these concerns, resulting in code tangling and code scattering [5].

Aspect Oriented Programming (AOP) methodology provides the capability to modularize these crosscutting concerns. This facilitates implementations that are easier to understand, design, implement and maintain, resulting in increased reusability, higher productivity, better traceability and flexibility.

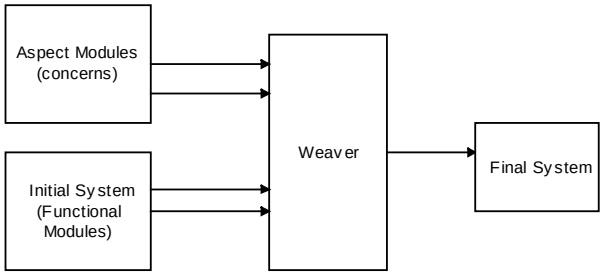


Figure 1. Aspect Oriented Programming Model

Crosscutting concerns can be identified from the system level requirements, which are independent of each other as well as the application level requirements. The identified crosscutting concerns are implemented in a loosely coupled modularized fashion; these concerns are then weaved into the system to form the final system. These concerns can be weaved into the system either at compile time or at runtime.

A typical development cycle of AOP involves identification of concerns in the systems, implementation of concerns and integration of these concerns into the system.

Different abstractions are provided for the developer by some of the implementations of AOP like AspectJ [2], AspectC++[3]. These abstractions are specified as join points, point cut, advice and aspect [6]. *Join points* are defined as points in the code where aspects can interfere. *Point cut* is a set of join points. *Advice* contains the code that should be run when the join points specified by the point cut are reached. *Aspect* combines all these constructs into one. Aspect languages are provided, which are basically extensions to current languages like Java and C++, to define the AOP constructs. Separate compilers are provided to compile the aspect code.

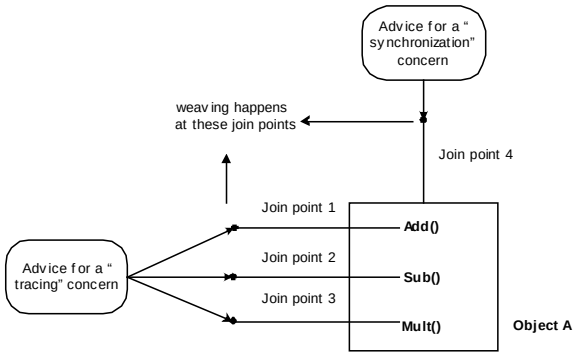


Figure 2. Aspect Oriented Programming Constructs

3. DYNAMIC CODE INSTRUMENTATION TECHNIQUE

Dynamic Code Instrumentation [4] provides a technique for the instrumentation of the snippet code into the application, when the application is executing, by working on the application's binary image. It doesn't require the application to re-compile, re-link or even re-execute unlike the other traditional code instrumentation techniques.

This technique works on the simple idea of code relocation. The instructions at the instrumentation point are relocated to another memory location called base-trampoline. The instrumentation point instructions in the application are replaced by instructions to branch to the base trampoline. In addition to the original instructions of the application, the base trampoline contains instructions to branch to the mini trampoline. Mini trampoline saves the appropriate machine state (such as the registers and condition codes) and contains the code for the snippet to be inserted. After executing the instrumented snippet it restores the machine state and branches back to the base trampoline. The base trampoline executes the relocated program instructions and branches to the application.

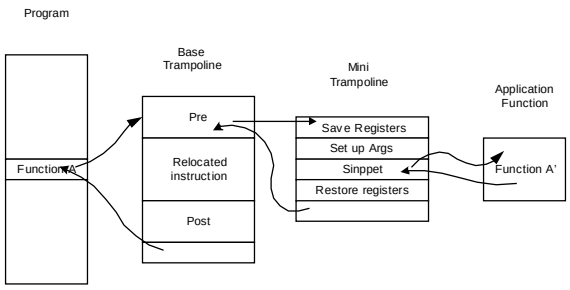


Figure 3. Dynamic Instrumentation Model [4]

DynInst [7, 8, 9] is a C++ class library built on Dynamic Code Instrumentation Technique. DynInst API allows creation of a new snippet of code and insertion of the same into a running program. It also allows alteration of the instrumentation and permits machine-independent binary instrumentation programs to be written.

DynInst API provides two primary abstractions as points and snippets. The location in the program where instrumentation can be done is called as point and the code to be inserted at this point is called snippet. Using the interface, the code snippet is defined as Abstract Syntax Tree and given to the library, specifying the point at which the code has to be instrumented. The DynInst is thus provided with details such as what to instrument and where to instrument. This enables the machine dependent code to be generated at run-time and weaved at the points specified in the binary image during its execution.

At present, the Dynamic Instrumentation Technique is primarily exploited in Paradyn, a project for building Parallel Performance Tool [10]. Paradyn allows collecting metric data from parallel processes running on multiple nodes across the network. The metric data to be collected can be specified at the runtime.

4. RUNTIME WEAVING OF CONCERNS USING DYNAMIC INSTRUMENTATION TECHNIQUE

Runtime weaving is comparatively difficult to achieve in non-interpreted languages. Dynamic instrumentation is one technique that looks at instrumenting the application binary at runtime. This technique can be used for runtime weaving of aspects.

Dynamic instrumentation provides abstractions called points and snippets. The aspect modules can be thought of as snippets and these can be inserted at different points. At present the DynInst API provides only three instrumenting points, namely function entry, function exit and function call.

The specific aspect to be instrumented need not be known to the application until the runtime. This allows the application developer to build a repository of aspects and weave them depending on certain criteria. Even the aspects can be dynamically generated and weaved. Aspect modules are compiled separately and only the code that has call statements to this aspect is given to the runtime weaver. This reduces the overhead in compiling on the weaver.

One of the overheads involved in using dynamic instrumentation is that it temporarily stops the process execution for the time of instrumentation. This may be a major drawback while building real time systems.

5. AOP FOR BUILDING INTELLIGENT ADAPTIVE SYSTEMS

Many Software Systems are designed, keeping static models in mind. When the system is deployed on the field, it faces many challenges either from the modules within the system or from the environment. Subsystems may not get expected services from other subsystems. For instance, a common persistence subsystem may not provide the required information within the specified time. The external environment may not provide the required resources for the system. The CPU or the memory may not be sufficient for the system to perform efficiently. The system has to survive in dynamic changing conditions without compromising on its intended goals. There needs to be adaptability built into the system.

Most of the internal challenges can be addressed by building adaptiveness into the system. When the system is designed some of the common modules can be separated and made independent of the application modules. In normal execution the subsystem gets the services from a particular subsystem. When it is not satisfied with the services it can choose to move to another subsystem which gives better service. Building some intelligence into the system can bring the process of choosing the best subsystem.

To summarize, an *Intelligent Adaptive System* [11] is one that responds to changes in the environment and from within, to achieve the intended goal.

5.1 Static Vs Dynamic Adaptation

Adaptation can be brought into the system in two ways either by static or by dynamic means. Static adaptation takes the liberty of bringing down the system to induce the change and also demands sufficient amount of downtime of the system. Dynamic adaptation works on the executing system and brings the change gracefully into the system.

Dynamic adaptation poses a greater challenge than static adaptation. Some of the challenges include designing of fairly independent subsystems, runtime assembly/disassembly of these subsystems into the system, maintaining structural integrity of the system, etc.

5.2 Concern Modules for Dynamic Adaptation

Most of the modules that need dynamic adaptation fall into the system level concerns. There are several reasons for which dynamic adaptation is required in such systems, either the concerns modules have to be added or removed at runtime, or the concerns need to be redefined. Repositories of concern modules are maintained and these are introduced into the system on demand.

Since, most of the required adaptation modules are captured as concern modules, it will be easy to build adaptation into the system. The runtime management of these concerns can be either automated or manually done.

Whenever adaptation needs to be brought into the system the user interacts with the system and specifies the necessary changes. These changes can be induced into the system either by adding new concerns, replacing or removing existing concerns. Automation of this process can turn the system into an intelligent self-adaptive system. Basically a self-adaptive system has a set of sensors to monitor different parameters

from within and outside the system. Depending on the monitored parameters it checks whether some change need to be brought into the system. This involves some level of reasoning and inferring mechanisms built into the system using knowledge base. Change into the system can be brought about by managing different concerns modules.

5.3 Sample

We have built a small prototype as a "Proof of Concept" to demonstrate the feasibility of runtime weaving using dynamic instrumentation. Here we present a simple sample that can be realized using this concept and map some of the presented concepts.

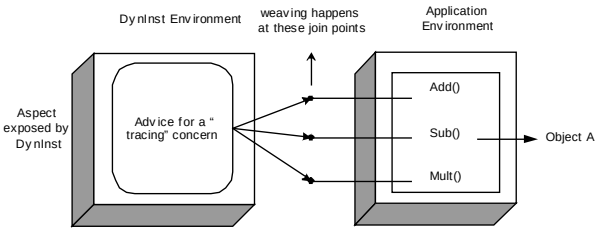


Figure 4. Sample Application

The sample application provides basic mathematical functions like addition, subtraction and multiplication. One of the requirements was to trace the function parameters before the function uses it. The tracing should be added to the functionality based on the need.

Tracing is a system level requirement that is independent of the application and is therefore considered as an aspect. Tracing code is instrumented at the join points through DynInst. This instrumentation is done at runtime.

6. SUMMARY AND FUTURE WORK

In this paper, we tried to present a case for runtime weaving in aspect oriented programming and show how adaptive systems can be benefited. We also presented how dynamic instrumentation technology will be useful for runtime weaving of these concerns.

Even though we didn't contribute in conceptualizing either the AOP or the dynamic instrumentation technique, we tried to see how they could be used together in building Intelligent Adaptive Systems. Most of our contribution in this area is limited to only building a prototype, and in realizing our thoughts and ideas in this direction. Other inherent challenges need to be explored in depth.

A full-fledged framework can be developed on top of this concept, providing some level of abstraction to the developer.

7. ACKNOWLEDGEMENTS

We would like to thank Bryan R Buck, Department of Computer Science, University of Maryland, for his help in understanding DynInst API library and running the samples. We would also like to thank Olaf Spinczyk, Otto-von-Guericke University Magdeburg, for his support in helping us understand AspectC++.

8. REFERENCES

[1] Aspect Oriented Software Development web site, http://www.aosd.net
[2] AspectJ website, http://www.aspectj.org
[3] AspectC++ website, http://www.aspectc.org
[4] J.K. Hollingsworth, B.P. Miller, and J.M. Cargille. "Dynamic Program Instrumentation for Scalable Performance Tools. *Scalable High Performance Computing Conference"*, Knoxville, TN (May 1994).
[5] I want my AOP! , Part 1, JavaWorld Article, Jan 18, 2002
[6] Olaf Spinczyk, Andreas Gal and Wolfgand Schroder-Preikshchat, *AspectC++: An Aspect-Oriented Extension to C++*.
[7] [Bryan Buck and J.K. Hollingsworth. "An API for Runtime Code Patching". *Journal of Supercomputing Applications*,2000.
[8] J. K. Hollingsworth and B. Buck, "*DyninstAPI Programmer's Guide*", http://www.dyninst.org/docs/dyninstProgGuide.v30.pdf, March 2001
[9] Dynamic Instrumentation website, http://www.dyninst.org
[10] Paradyn Parallel Performance Tool website, http://www.paradyn.org
[11] Application Fault Management and Diagnosis, Honeywell Labs Internal document.

Connecting Aspects in AspectJ: Strategies vs. Patterns

Stefan Hanenberg
University of Essen, Institute for Computer Science
Schützenbahn 70, D-45117 Essen, Germany
shanenbe@cs.uni-essen.de

Pascal Costanza
University of Bonn, Institute of Computer Science III
Römerstraße 164, D-53117 Bonn, Germany
costanza@cs.uni-bonn.de

ABSTRACT

Aspects in AspectJ can be connected to existing classes and applications in order to amend them with additional ancestors, methods and advice to existing methods. However, for concrete usage scenarios there are different options of how to use AspectJ's features, and these options deeply impact the opportunities for further evolution of both base classes and aspects.

The purpose of this paper is two-fold. First, it introduces *strategies* that describe these options and their specific tradeoffs. These strategies provide a common terminology and support developers in deciding which option to use in what situation. Second, their presentation obviously resembles the structure of well-known design patterns, but it is not clear to what extent they can rightfully be regarded as patterns themselves. This issue is discussed by giving two oppositional position statements.

1. INTRODUCTION

AspectJ developed at the Xerox Palo Alto Research Center is currently the most popular general purpose aspect language built on top of the programming language Java and offers additional composition mechanisms to modularize cross-cutting concerns. It supplies a class-like construct called *aspect* that permits to define code that cross-cuts a given application. Furthermore, it offers means to define *how* this code cross-cuts a given application. The usage of these language constructs has a direct impact on how reusable an aspect is and how easy it can be applied to new situations. So the developer has to be careful when designing aspects using those constructs because it might influence the evolution of the resulting software or the applicability of the developed aspects in an undesired way. Since the definition of how aspects cross-cut applications means to describe connections between aspects and applications, the main focus of developing aspects in AspectJ lies on these connections.

In this paper we describe strategies for connecting aspects to applications in AspectJ. These strategies are recurring in different contexts, so this collection of strategies can be regarded as a catalogue that gives developers an overview of techniques that are used often. Since they are presented in a form that resembles the structure of (design) patterns it seems reasonable to discuss the relationship between such strategies and patterns.

In section 2 and 3 we propose recurring strategies and exemplify their benefit in section 4. In section 5 and 6 we discuss in two oppositional statements the relationship between the proposed strategies and patterns. Finally, we summarize this paper.

2. STATIC CROSS-CUTTINGS

According to the AspectJ terminology, we use the term *static crosscutting* to describe crosscuttings that influence the interfaces of the involved types [2]. AspectJ provides a mechanism called *introduction* to achieve this kind of influence.

Direct Ancestor Introduction

It is often observable that different objects have common properties from a certain perspective. A perspective is a subjective view on the system, this means such mutuality is not intrinsic to those objects. From this perspective, all of those objects should be treated in the same way and therefore should be substitutable. In object-oriented programming substitutability is achieved by classification. Here classification does not occur because of intrinsic common properties of such objects, but because of an aspect specific view on the system. Therefore the classification is not part of the object definition, but part of an aspect definition. A *direct ancestor introduction* directly introduces an aspect-related, extrinsic ancestor to objects, i.e. the desired mutuality of objects is not intrinsic to those objects.

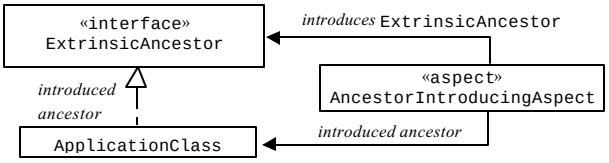


Figure 1: Direct Ancestor Introduction

The participants of this strategy are:

- *extrinsic ancestor*: the class, interface or aspect that contains the common properties.
- *ancestor-introducing aspect*: the aspect that defines the classification.

The consequences of using a *direct ancestor introduction* are:

- *extrinsic classification*: classification of objects is not only determined by the class definition, but also by the introducing aspect.
- *matching signatures*: when applying this strategy the developer must guarantee that the application related class is able to establish the introduced ancestor. For example, if the ancestor is an interface the developer has to guarantee that the class implements the methods of that interface.

AspectJ directly supports the direct ancestor introduction on the language level. This strategy just corresponds to the usual application of static cross-cutting for declaring an *implements* or *extends* relationship where the type pattern in the introduction directly corresponds to existing classes.

Direct Member Introduction

Sometimes it is desirable to add properties to selected objects because of a certain perspective. This means that from a certain perspective, different objects have common properties. A *direct member introduction* introduces extrinsic properties directly to objects without achieving substitutability of those objects.

The participant of this strategy is:

- *member-introducing aspect*: the aspect that contains the introduction. The introduction directly refers to the classes of those objects that should get the new members.

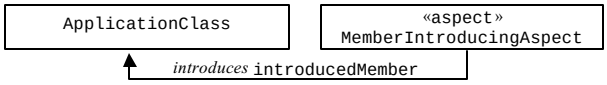


Figure 2: Direct Member Introduction

The consequences of using a direct member introduction are:

- *limited reusability*: the introductions themselves a hardly reusable since AspectJ does not permit to override introductions incrementally in an "object-oriented programming way". For example, if the developer decides later on that the introductions should be applied to further classes or interfaces, the aspect itself has to be modified.
- *members inherent to the aspect:* the introduced members are extrinsic to the objects. Therefore the developer has to guarantee that only clients that are aware of the introducing aspect can use them.
- *no substitutability:* although different classes get common properties their instances are still not substitutable.
- *member conflicts*: The developer has to guarantee that there are no conflicts between the extrinsic and intrinsic members. For example, no extrinsic member's identifier is allowed to be equal to an intrinsic member's identifier.

AspectJ directly supports direct member introductions on the language level.

Indirect Introduction

Sometimes it is necessary to apply several introductions to certain objects from different perspectives. This means that there are several extrinsic characteristics that originate from different perspectives and should be combined to be applied to certain objects later on. Although it is known which perspectives are to be combined the definition of the objects that they are to affect should be deferred. An *indirect introduction* collects several extrinsic properties from different perspectives within one unit and defers the binding to existing objects.

The participants of this strategy are:

- *introduction container*: the unit that is used as the target for the introductions. The container contains the property definitions and the ancestor relationships.
- *introduction loader*: the aspect that introduces properties and ancestors to the container.
- *container connector*: the aspects that connects the container to application classes.

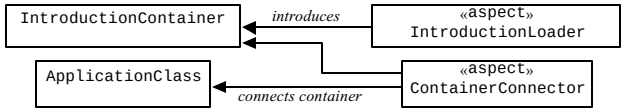


Figure 3: Indirect Introduction

The consequences of using an indirect member introduction are as follows:

- *reusable introductions*: the introductions are defined independent of the classes they influence and their application just consists of a container connection without the need to re-implement the introductions itself.
- *little aspect-related knowledge required at connection time:* the container connection does not need to know about introduction loaders.
- *member conflicts:* because the container connector does not know about the concrete introductions to the container there is some danger of possible conflicts between class members. The container connector cannot resolve conflicting introductions because the introducing aspects are transparent.

In AspectJ there are mainly two ways of implementing a indirect introduction. First, it is possible to introduce members and ancestors directly to an interface. In this case the ancestor introduction is limited. For example, it is not possible to introduce a class as an ancestor to an interface. Second, it is possible to introduce members and ancestors to each class that implements the container interface. Then the interface can be applied to classes by a direct ancestor introduction. The difference to the former implementation is that the container is not changed by the introduction loaders. Instead it is only used for identifying the classes that are to be affected by the introductions. Both approaches have in common that they make use of a direct ancestor introduction.

3. DYNAMIC CROSS-CUTTINGS

The previous sections describe strategies for adding attributes or ancestors that do not influence the behavior of applications. This can only be achieved by so called *dynamic cross-cuttings*. AspectJ provides two language constructs for dynamic cross-cutting: *advice* and *pointcuts*. Advice define the adapted behavior and pointcuts the places where advice crosscut existing structures. Like in the sections before, the names of the strategies are directly derived from the AspectJ terminology.

Direct Pointcut Connection

Sometimes is it desirable to adapt the existing behavior of certain objects well known to the developer. The behavior to be added is

extrinsic to such objects and it is not assumed that the behavior of any further objects not mentioned in this context should be amended in the same way. A *direct pointcut connection* directly influences the behavior of application objects.

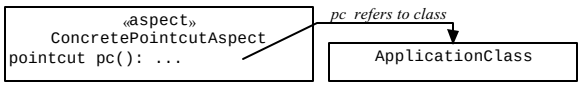


Figure 4: Direct Pointcut Connection

The strategy consists of the following participants:

- *concrete pointcut aspect*: the aspect that contains the behavior to be added and the definition of the situations when the additional behavior takes place.

The consequences of this strategy depend on its implementation:

- *no incremental modifications*: if the aspect itself includes concrete pointcuts (that are not inherited from a superaspect), there is no possibility to modify them incrementally (see [6] for a further discussion).

Direct pointcut connections are directly supported by AspectJ and correspond to the standard application of concrete pointcuts.

Indirect Pointcut Connection

An *indirect pointcut connection* defines a uniform way for adapting object behavior without naming the concrete objects.

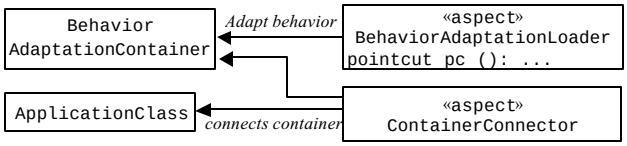


Figure 5: Indirect Pointcut Connection

The participants of this strategy are

- *behavior adaptation container*: the container that collects several behavior adaptations.
- *behavior adaptation loader*: the aspect that contains the new behavior and the description at what join points the new behavior should occur.
- *container connector*: the aspect that connects the behavior adaptation container to the application classes.

The consequences of using an indirect pointcut connections are:

- *typespecific cross-cuttings*: the dynamic cross-cutting code can be attached to arbitrary types. However, it is not possible to attain behavior-specific cross-cuttings.
- *few aspect-related knowledge required at connection time:* the pointcut definitions are transparent to the container connector. No information is needed about the behavior adaptation loaders to perform the connection.
- *aspect conflicts*: the developers that implement the behavior adaptation loaders must guarantee the consistency of the loaders. The container connector cannot detect or solve any consistency problems.

In AspectJ an indirect pointcut connection is achieved by defining (concrete) aspects with (concrete) pointcuts for a specific interface. Afterwards, this interface can directly be introduced to application classes.

Template Advice

A *template advice* separates the definition of behavior adaptation from the definition of how this behavior crosscuts a given structure. The crosscut is available as a hook for later specification, independent of the actual behavior. In that way, a template advice allows advice to be reused in different situations.

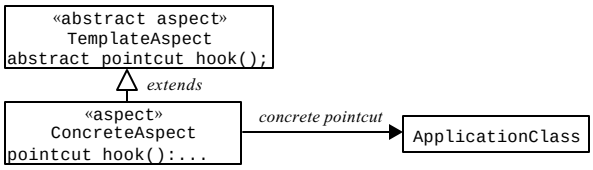


Figure 6: Template Advice

A template advice consists of:

- *template aspect*: the aspect that contains new behavior without specifying where this new behavior occurs. The behavior should take place at a certain hook pointcut.
- *concrete aspects*: the aspect that extends the template aspect and specifies the corresponding join points where the behavior should take place.

The consequences of applying a template aspect are:

- *pointcut independent aspect reuse*: it is possible to apply the behavior adaptation to situations that have not been foreseen at the time of aspect definition.
- *non-transparent pointcut*: in contrast to the indirect pointcut connection the developer responsible for connecting the dynamic cross-cutting code to an application has to know something (the hook pointcut) about the aspect to connect.

In a straight forward implementation of a template advice in AspectJ, the template aspect has to be abstract and the concrete aspect has to extend the concrete aspect. In [6], the application of the template advice in AspectJ is discussed in more detail.

Composite Pointcut

It is often observable that the way dynamic crosscutting occurs can be expressed by a combination of independently defined dynamic cross-cuttings. A *composite pointcut* separates a pointcut into two logically independent pointcuts.

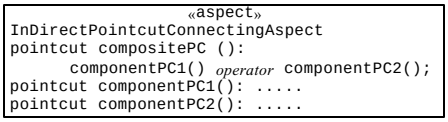


Figure 7: Composite Pointcut

A composite pointcut consists of the

- *component pointcuts*: the logically independent pointcuts.
- *composite pointcut*: the pointcut that combines several component pointcuts.

The consequences of using a composite pointcut are:

- *independent pointcut modification*: the logically independent component pointcuts can be modified without knowing the complete (composite) pointcut.
- *pointcut consistency*: the composite pointcut cannot guarantee the consistency of the pointcuts, so the developer must be aware of how to define the component pointcuts correctly.

In AspectJ a composite pointcut can be implemented by defining a pointcut that consists of a combination of pointcuts and does not use any pointcut designators on its own. Usually a composite pointcut is used in the context of a template advice.

4. EXAMPLE

In this section we analyze an implementation of the Observer pattern [5] in AspectJ similar to the one proposed in [2] by applying the strategies described above.

For example, we would like to provide graphical representations of application-specific objects that are automatically revised whenever the corresponding subjects changes. To support the Observer pattern, subjects must provide an interface for attaching and detaching observers. So the mechanism of member introduction in AspectJ can be used to equip classes with the (extrinsic) methods for attaching and detaching observers without actually changing the application's source code. The question is if a direct member or a indirect introduction should be used. Since the subject related methods can be used for a number of different classes (even though we are right now just interested in a few of them) an indirect introduction provides more flexibility. So we build the interface `Subject` (introduction container) that includes the methods `addObserver()` and `removeObserver()`. Moreover, we create an interface `Observer` that contains the method `update()` that should be invoked whenever a subject changes.

In order to allow an indirect introduction, we create the introduction loader `SubjectObserverProtocol` that introduces appropriate implementations to `Subject`:

```
aspect SubjectObserverProtocol {
    public Vector Subject.observers = new Vector();
    public void Subject.addObserver(Observer obs) {
        observers.addElement(obs);
        obs.setSubject(this);}
    public void Subject.removeObserver(Observer obs) {
        observers.removeElement(obs);}
}
```

Additionally, we are able to implement actions that should happen whenever a subject's state changes in this aspect: the `update()` method of every attached observer must be invoked. This code is part of the dynamic cross-cutting because it should be executed whenever the join points have been reached that immediately follow a change of the subject's state. However, it is hardly possible to define a consistent connection strategy for all possible subject classes in this case (cf. [3], [4], [6]). Therefore, we make use of the template advice that defers this decision. We regard it as a good idea to implement the advice in `SubjectObserver-Protocol` and thus we have to define the aspect abstract:

```
abstract aspect SubjectObserverProtocol {
  ...
  ... pointcut stateChanges(Subject s) ...
  after(Subject s): stateChanges(s) {
      for (int i = 0; i < s.observers.size(); i++)
        ((Observer) s.observers.elementAt(i)).update();
  }
}
```

We still have to decide how to implement the pointcut connection. Obviously, we are able to define that the observed target is of type `Subject`. However, we cannot decide what message receptions change a subject's state. Therefore the needed pointcut consists of a known part (target is of type `Subject`) and an unknown part. Therefore, we should use a composite pointcut.

```
abstract aspect SubjectObserverProtocol {
    ...
    abstract pointcut stateChanges();
    after(Subject s):
        target(Subject) && stateChanges(s) {...}
...}
```

The implementation in [2] does not use a *composite pointcut* and just uses an abstract pointcut. The result is that developers that want to apply the protocol have to guarantee that the pointcut parameter `s` refers to the right subject instance in their pointcut definition. Instead, the use of the composite pointcut already restricts developers to targets of type `Subject`, and therefore reduces errors when connecting the protocol to an application.

This example illustrates how the strategies for connecting aspects introduced above allow us to design a concrete high-level subject/observer protocol in AspectJ. Nevertheless, there are still some variation points of how the protocol can be applied to existing applications.

Whereas the usage of the indirect introduction needs `Subject` to be used as the extrinsic ancestor in a direct ancestor introduction, the actual implementation of the method `update()` in `Observer` is not fixed. It can either be added by a direct member introduction (the implementation in [2] uses this strategy), or by a simple Java implements relationship where the developer of the observer class is responsible for the definition. Furthermore, it is not prescribed how the concrete pointcut (`stateChanges()`) of `SubjectObserverProtocol` is connected to the application within the template advice. Usually, this is achieved by a direct pointcut connection.

5. Hanenberg: Strategies, no Patterns

In the previous sections, we have introduced recurring strategies that are used when developing AspectJ applications. I use the term strategy intentionally to delimit it from the term "pattern". In the following sections, I argue that these strategies are no patterns.

The main purpose of identifying these strategies was to find out what language features of AspectJ are usually used in what situations. Afterwards, we wanted to provide a catalogue of strategies that supports developers to decide what strategy to use in certain situations. Thereto, it is necessary to organize the strategies in a way that allows developers to easily identify them

and find out when and how to use them. Furthermore, developers must be aware of the consequences when using a certain strategy.

We organized the strategies in the following way. Every strategy has a *unique name*, a *description of its essence*, a *description of a situation* where it is typically used (skipped in this paper), a *description of the strategy's participants*, an *illustration of its form*, a *discussion of the consequences* of its application and a *discussion of how the strategy can be used* in AspectJ.

In this way, the strategies are organized similar to the GoF-design patterns [5]. Furthermore, it seems as if the benefit of the strategies is similar to that of design patterns: developers get a common vocabulary that eases their communication, and a catalogue that permits to decide when to use what strategies. In section 4 we have shown how those strategies can be applied in concrete scenarios. Nevertheless, there are differences between patterns and the strategies mentioned here.

The success of patterns is based upon a common understanding of object-oriented programming. All of the GoF design patterns are directly build on top of object-oriented constructs. Such a common understanding permits the problem and solution to be visualized effectively, by using standard object-oriented notations. Finally, the solution part of patterns can easily be understood by all object-oriented developers. Although the underlying programming languages may differ, developers are familiar with concepts like *object* or *message*. A similar situation is yet not given in the aspect-oriented community. Until now, there is no common understanding of aspect-oriented programming and therefore, no commonly accepted design notation.

The strategies have directly arisen from the usage of AspectJ, so they are the result of observing AspectJ code. This means that the strategies depend highly on the language. Although other aspect-oriented approaches like HyperJ permit to implement these strategies as well, the form of their implementation completely changes - aspects do not exists on the language level, and for example, no inheritance relationship between aspects can be used as is required in the template advice. As the form of the strategies differs between different aspect-oriented approaches, there is no reasonable usage of them. For example, a HyperJ programmer would not understand how the illustration of a strategy relates to the tool at hand.

Furthermore, it should be mentioned that the distinction between static and dynamic cross-cutting has directly arisen from AspectJ's terminology. However, there is a parallel between a template advice (dynamic crosscutting) and an indirect pointcut connection (static crosscutting). Both allow crosscutting code to be defined without specifying what locations the code should be woven into. So the distinction between static and dynamic crosscutting does not seem to be "natural". If we would only distinguish between crosscutting strategy (the way how something crosscuts various structures) and crosscutting code the only difference would be that code might affect code within a method (dynamic crosscutting) or code within a class (static crosscutting). There does not seem to be a strict necessity to provide different language features for both kinds of crosscutting.

Because of the language dependency it seems to be more appropriate to discuss the relationship between the strategies and *idioms* that are "low-level patterns specific to a programming language (..). which describe how to implement particular aspects of components or the relationships between them using the features of the given language." [1].

The second major reason why the strategies are no patterns is based on their "quality". The "quality without a name" describes the "life and wholeness" of a product. Patterns are constructs for generating this quality, so software generated by a suitable application of patterns should have the quality. However, our strategies were identified from an appropriate usage of the language constructs provided by AspectJ. In fact it cannot be definitively determined if the usage of those constructs has this quality because of the following reasons: there is not enough experience in the area of aspect-oriented programming to determine the quality of an (aspect-oriented) solution. It is not even determined, if the composition mechanism in AspectJ are good at all, even though they seem to solve known problems in object-oriented programming. To determine if some of these strategies are patterns, it is necessary to have a lot of experience in the area of aspect-oriented programming.

At least there seems to be a difference in the abstraction of the strategies in comparison to known patterns. The strategies concentrate on how to connect aspects with existing software - how extrinsic properties can be attached to objects. In this way, the problem space handled by those strategies seems to be directly derived from the typical problem of aspect-oriented programming on the implementation level.

So the overall impression is as follows. Although there are similarities between the strategies and patterns, they are not equal. I doubt that trying to bring the strategies to a corresponding pattern form would really result in new aspect-oriented patterns, since they are too dependent on the language AspectJ. Nevertheless, there seem to be strategies which are more interwoven with AspectJ (like composite pointcut) than others (like direct member introduction). From my point of view, it seems to be appropriate to compare new aspect-oriented technologies which appear from time to time with the strategies. This might improve a common understanding on aspect-oriented programming and improve the understanding on the mutuality of different aspect-oriented techniques.

6. Costanza: A First Step Towards AO Patterns

Before giving my position about the work introduced in this paper, I would like to recall the general idea of Patterns. At the present stage, there are mainly two views on what Patterns are all about. The one is to characterize Patterns as a literary form that is well-suited to communicate recurring problems and good solutions that resolve the forces of these problems. For this reason, a generally accepted "canonical form" has been established over the

past years. According to this form, each pattern consists of a name, a problem statement, a context, the forces that lie at the center of the problem, a solution, examples, the resulting context, a rationale (why the pattern works), related patterns and known uses [1]. Although the AspectJ strategies of this paper do not exactly match this form, they surely are very close. The only really missing elements are the forces, examples, the resulting context, related strategies and known uses. It is easily conceivable that examples can be drawn from introductory material, for example [2], and known uses from ongoing "real-world" projects that make use of AspectJ. The relation between certain strategies can already be seen to some extent in this paper – for example, the direct vs. indirect introduction are roughly used for the same purposes, with different tradeoffs (resulting contexts). It is equally conceivable to elaborate on the forces. An interesting case is the need to modularize crosscutting concerns which would be a force that all aspect-oriented strategies have in common. The fact that there might not be enough known uses for the strategies given here implies that they can only be regarded as "proto-patterns" [1], but on a more general level, from a "literary" point of view, they certainly qualify for being good examples of the pattern idea.

Another view on Patterns is the notion of achieving the so-called "Quality Without a Name" (QWAN). A "light-weight" paraphrase of this idea is the goal of making people feel more comfortable. For example, programs that employ object-oriented design patterns [5] make programmers more comfortable in changing their source code and, for example, adding new functionality. Again, the strategies of this paper qualify for having QWAN, at least in principle, because they also aim at easing the maintenance of software. Again, the lack of known uses, or other rationales, indicate that they can only be regarded as "proto-patterns" because their application in the "real world" might necessitate their modification in order to really achieve QWAN. However, this does not generally preclude their perception as patterns.

It is important to note that none of the views on Patterns presented here require them to be applied in object-oriented contexts only – the patterns from [5] just *happen* to be based on the building blocks of object-oriented programming, like composition, inheritance, overriding, and so on. However, many patterns also encompass other areas, for example other programming paradigms, as in hybrid languages like C++ and Lisp, up to methodological and organizational patterns that do not directly deal with programming at all [8]. So regardless of the view on patterns as a literary form or as a means to achieve QWAN, there is no reason at all to *not* apply them in an aspect-oriented setting. The only difference is that now aspect-oriented concepts are the building blocks, like pointcuts, introductions and advice.

So my conclusions are as follows. The strategies presented in this paper are a very valuable first step towards a catalogue of aspect-oriented patterns. Future steps include...

- ...elaboration of forces and known uses. These are the missing elements that probably require most of the work and investigation of existing projects.
- ...discovery of more advanced aspect-oriented patterns. The strategies of this paper are very elementary ingredients to aspect-oriented programming, but there will certainly arise more complex scenarios. For example, good candidates for solutions to be documented in pattern form are those that deal with feature interaction among different aspects.
- ...generalization of aspect-oriented patterns in order to be independent of a specific aspect-oriented approach. Apart from being more useful in different environments, such patterns would help to improve our understanding of the essence of the still emerging field of AOSD.

7. Summary

In this paper we have pointed out that the main focus of aspect-oriented software development lies in the connection between aspects and applications. We have described recurring strategies for connecting aspects and applications in AspectJ and we have illustrated how they can be used in a concrete example. Afterwards we have discussed to what extent these strategies can be regarded as patterns or not by giving two oppositional positions statements.

In conclusion, it is clear that developers who want to exploit the prominent features of aspect-oriented approaches need to gather good solutions and communicate them effectively. This paper provides strategies for AspectJ as good starting points and in doing so, hints to a feasible future practice of documentation for the aspect-oriented community, regardless of whether the proposed strategies will be perceived as patterns or not.

8. REFERENCES

[1] Appleton, B., Patterns and Software: Essential Concepts and Terminology, http://www.enteract.com/~bradapp/docs/patterns-intro.html

[2] AspectJ-Team, The AspectJ Programming Guide, http://aspectj.org/doc/dist/progguide/

[3] Brichau, J., De Meuter, W., De Volder, K., Jumping Aspects, Workshop on Aspects and Dimensions of Concerns, ECOOP, 2000.

[4] Costanza, P., Vanishing Aspects, Workshop on Advanced Separation of Concerns, OOPSLA, 2000.

[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns, Addison-Wesley, 1995.

[6] Hanenberg, S., Unland, R. Using And Reusing Aspects in AspectJ. Workshop on Advanced Separation of Concerns, OOPSLA, 2001.

[7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwing, J. Aspect-Oriented Programming. Proceedings of ECOOP, 1997.

[8] Rising, L., The Pattern Almanac 2000, Addison Wesley, 2000.

A pattern based approach to separate tangled concerns in component based development

Wim Vanderperren
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 62
wvdperre@vub.ac.be

ABSTRACT

This work builds on aspect-oriented software development ideas and our previous research where we lift the abstraction level of visual component based development. In component based development, the components are the natural unit of modularization. However, there will always be concerns that cannot be confined to one single component. We introduce composition adapters as a means to modularize crosscutting concerns in separate and reusable entities. A composition adapter describes an adaptation of the interaction protocol between a set of components. An important feature of a composition adapter is that the adaptations are described independent of a concrete API, making them highly reusable. Using composition adapters, we are able to weave crosscutting aspects in a component based application. The weaving algorithm uses automata theory to allow the state-based insertion of a composition adapter into the interaction protocol. This allows a seamless integration with our component based methodology. We embedded composition adapters and our algorithms into PacoSuite, a visual component composition tool that is used in our lab as a research vehicle. PacoSuite hides the underlying complexity to the component composer, rendering an easy to use visual component based development environment that includes now aspect separation features through composition adapters.

1. INTRODUCTION

Component based software development is considered a promising paradigm for curing the so-called software crisis [1]. The idea is that applications are created by composing reusable components. Hence both the software quality and the development speed improve substantially. At the System and Software Engineering Lab (SSEL) we have been doing research on a novel component based software development methodology for a couple of years. The major goal of our approach is to lift the abstraction level for component based software development. The success of design patterns [2] indicates that there exist collaboration patterns that are used frequently. Therefore, we introduce explicit and reusable composition patterns. A composition pattern is an abstract specification of an interaction between a number of roles. Our approach allows us to automatically verify whether a component is able to work as a role of a composition pattern prescribes. Moreover, we are able to generate glue-code that translates syntactical compatibilities between a number of components mostly automatically.

Another research direction that has received lots of attention in the last years is Aspect-Oriented Software Development (AOSD) [3]. Some aspects of an application cannot be cleanly modularized using current software engineering methodologies. Typical examples include logging or synchronization. The focus of AOSD research has been on separating crosscutting concerns in an object-oriented context. However, the same problem also applies to component based software development. To be able to separate crosscutting concerns in our component based context, we introduce the concept of a composition adapter. A composition adapter describes adaptations of the external behavior of a component independently of a specific API. When a composition adapter is applied on a composition of components, we are able to verify whether this makes sense. Moreover, we are able to automatically insert the adaptations described by the composition adapter into the composition pattern. These algorithms are based on finite automata theory.

The next section describes the context in which this research is conducted, namely our current component based approach. The documentation of components and composition patterns is explained in more detail. Section 3 introduces the composition adapter and shortly sketches the algorithms necessary to automatically insert a composition adapter into a composition pattern. Section 4 presents the tool support that implements these ideas. After a short discussion of related work, we state our conclusions and describe our future work.

2. RESEARCH CONTEXT

We mainly focus our component based research on lifting the abstraction level for component based development. We want to realize the plug and play idea of component based development. Therefore, we propose to document components with usage scenarios that specify how to use the component. A usage scenario is expressed by a special Message Sequence Chart (MSC) [4]. The main difference with a normal MSC is that the signals are taken from a limited set of pre-defined semantic primitives. Each of these signals is also mapped on the concrete API that performs them. So the documentation of a component is both abstract and concrete. Figure 1 illustrates a usage scenario of a generic TCP/IP network component. One participant of the usage scenario represents the component and the others represent the environment participants the component expects. In this case, there's only one environment participant, namely the NetworkUser participant. This usage scenario documents that the

network component either expects data to send over the network or submits events to the NetworkUser environment participant. The network component submits an event when it received data, when the connection is established or when it is disconnected.

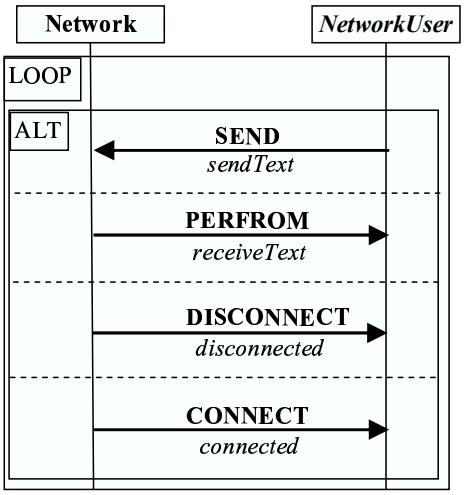


Figure 1: Usage scenario of Network component.

In addition, we introduce explicit and reusable composition patterns. A composition pattern is an abstract specification of the interaction between a number of roles and is also expressed by an MSC. The signals between the roles come from the same limited set of semantic primitives. This allows us to compare the signals in a usage scenario of a component with these in a composition pattern. Figure 2 illustrates a generic game composition pattern. This composition pattern specifies the interaction between three roles: the Network, GameGui and Checker roles. One of the applications of this game composition pattern is a distributed scrabble game. The checker role is than filled by a dictionary component that is used to verify the validity of a word. The GameGUI role is filled by a dedicated Scrabble user interface component. The network role can be filled by the network component of Figure 1.

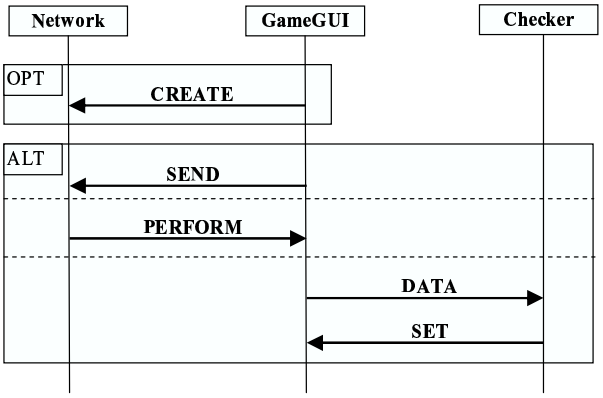


Figure 2: Generic game composition pattern.

The documentation of components and composition patterns allows us to automatically check compatibility of a component with a role. Glue-code that constraints the behavior of the components and that translates syntactical compatibilities is also generated automatically. These algorithms are based on finite automata theory. In this paper we do not go into the details of these algorithms. The interested reader is referred to [5,6,7].

3. COMPOSITION ADAPTERS

3.1 Introduction

Some aspects cannot be cleanly modularized using our current component based approach. Typical examples of such aspects are logging or synchronization. We encountered a more complicated aspect in the SEESCOA[1] research project. In this project we want to verify the quality of service of component based applications. More specifically, we would like to check both statically and dynamically whether a component based application satisfies certain timing constraints. Run-time checking of timing constraints turns out to be a crosscutting concern. If we want to check timing constraints dynamically using our current concepts, we have to alter every composition pattern individually in the same way. Of course, when the application goes into the production phase, we do not want to keep the dynamic timing aspect into the application. Consequently, we have to alter all the involved composition patterns again to remove the timing aspect. To solve this problem, we introduce the concept of a composition adapter.

3.2 Documentation

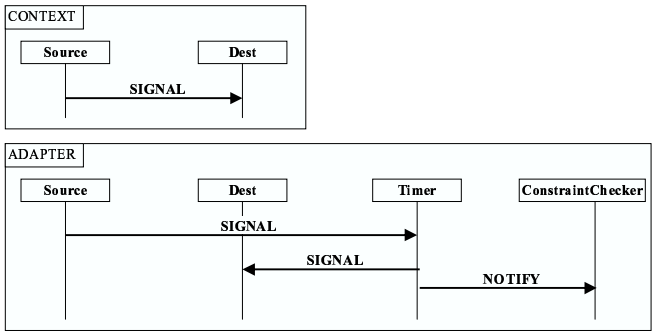


Figure 3: Dynamic timing checker composition adapter.

A composition adapter is able to describe adaptations of the external behavior of a component independently of a specific API. A composition adapter is also documented by a special kind of MSC and consists of two parts: a context part and an adapter part. The composition adapter we use to modularize the timing aspect is depicted in Figure 3. The context part describes the behavior that will be adapted. This can be as simple as one signal send as in Figure 3, but can very well be a full protocol. The adapter part describes the adaptation itself. In the case of the dynamic timing composition adapter every signal between the source and destination role will be re-routed through a Timer role. The Timer role is responsible for taking a timestamp and notifies the ConstraintChecker role. The ConstraintChecker role has a small database of timing constraints and verifies whether every signal it

[1] The SEESCOA (Software Engineering for Embedded Systems using a Component-Oriented Approach) IWT project is funded by the Flemish government. For more information see: http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/SEESCOA/

is notified of does not violate a constraint. To minimize the disruption of the system, the component that will be mapped on the ConstraintChecker role could do the verification process offline and/or run on a different CPU.

3.3 Applying a composition adapter

When the component composer applies a composition adapter onto an existing composition pattern, the context roles of the composition adapter have to be mapped onto roles of the composition pattern. For example, suppose we want to time the communication between the GameGUI and Checker roles of the composition pattern in Figure 2. Then we would have to map the Source role of the timing composition adapter of Figure 3 onto the GameGUI role of the composition pattern. Likewise, the Dest role has to be mapped onto the Checker role. The result will be that the DATA signal is not send directly to the Checker/Dest role but is first send to the Timer role. After sending the DATA signal to the Checker/Dest role, the ConstraintChecker role is notified.

Inserting a composition adapter seems obvious from the example explained above. In this example, merely syntactically scanning the affected composition pattern would do the job. However, when the context part of the composition pattern specifies a full protocol, a more involved algorithm is needed. Therefore, we developed an algorithm in three steps based on finite automata theory. In this paper, the algorithm is only shortly sketched. A more elaborate explication of the algorithm can be found in [8]. The algorithm does not work directly on MSC's but on Deterministic Finite Automata (DFA). The transformation of an MSC to a DFA is a standard process and described in literature [9]. The first step is a verification phase. This means searching all paths in the affected composition pattern that correspond to the context part of the composition adapter. If there are no matching paths, the application of this composition adapter makes no sense. In the second step, we insert the adapter part of the composition adapter in the composition pattern at the paths that match with the context part. The last phase consists of removing all paths that match with the context part. To this end, we calculate the difference automaton between the automaton resulting from the previous phase and a special version of the context part.

4. TOOL SUPPORT

The work described in this paper has been implemented in a prototype tool called PacoSuite. PacoSuite is entirely written in JAVA and consists of two applications, PacoDoc and PacoWire. PacoDoc is a graphical editor that allows drawing, loading and saving of component documentation, composition patterns and composition adapters. The PacoWire tool is our actual component composition tool and implements the algorithms we developed in our work [5,6,7,8]. It uses a pallet of components, composition patterns and composition adapters. The tool allows dragging a component on a role of a composition pattern. The drag is refused when the component does not match with the selected role and optionally mismatch feedback is given to the user. A composition adapter can be visually applied on a composition pattern. The tool checks whether the application of the composition adapter makes sense and automatically inserts the composition adapter into the composition pattern. When all the component roles are filled, the composition is checked as a whole and glue-code is generated. Figure 4 shows some screenshots of our tool.

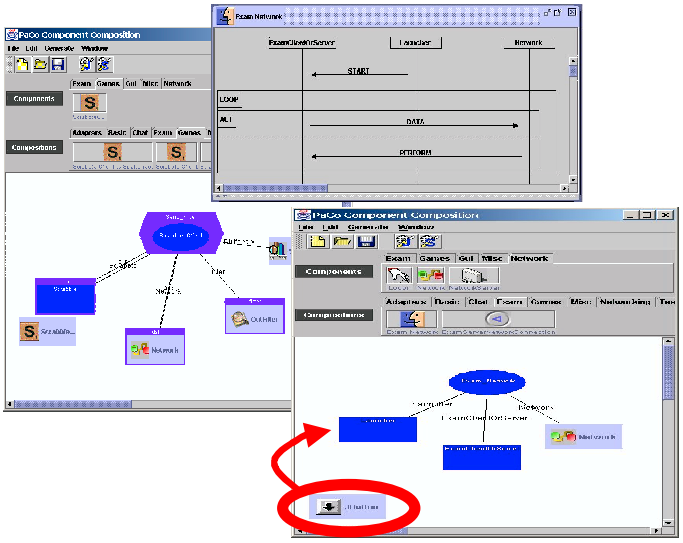


Figure 4: Screenshots of PacoSuite. At the top-right a screenshot of PacoDoc, our documentation tool is shown. At the lower-right, our actual component composition tool called PacoWire is shown. In this screenshot, the component composer is about to map a component on a role of the composition pattern. The leftmost shot shows a composition adapter that is applied on a composition pattern.

5. RELATED WORK

Although combining AOSD ideas with component based development is a rather new research direction, some approaches are already emerging. An interesting approach is event based AOSD [10]. Similar to the composition adapter approach they allow specifying an aspect on a full protocol of events.

The aspectual component approach [11] proposes a new component model to be able to specify crosscutting concerns. The aspects are weaved into the components using binary code adaptation techniques. The aspectual component approach improves on the composition adapter idea because aspects that alter the internals of a component can be specified. On the other hand, it is impossible to directly recuperate it in our component-based context. Because we do not want to lower the abstraction level, we have to come up with a (preferable graphical) notation of what the consequence of the adaptations on the exterior behavior of the altered components will be. This extra information is needed to allow automatic compatibility checking and glue-code generation.

Filman [12] proposes dynamic injectors to introduce aspects into a given component configuration. He incorporates dynamic injectors into OIF (Object Infrastructure Framework), a CORBA centered aspect-oriented system for distributed applications. The dynamic injector approach is very similar to our composition adapter idea because both approaches employ a wrapping and filtering technique to insert crosscutting concerns into a composition of components.

6. CONCLUSIONS AND FUTURE WORK

Using composition adapters, we are able to cleanly modularize crosscutting concerns in our component based context. Composition adapters can be verified and inserted automatically in a composition of components. We improve on current aspect-oriented approaches as the joinpoints where the composition adapter will be applied are specified by a full protocol instead of a mere set of methods. An important feature of a composition adapter is that the adaptations are described independent of a concrete API, making them reusable. Consequently composition adapters still preserve the high abstraction of our visual component composition methodology. However, this approach is only able to alter the exterior behavior of components by re-routing or ignoring their messages. As a consequence, concerns that require adaptations of the interior behavior of a component cannot be specified.

To be able to alter the internals of a component we have to use an aspect-oriented implementation language. There already exists a wealth of generic approaches to separate crosscutting concerns in an object-oriented context. Well known approaches include AspectJ [13], composition filters [14] and HyperJ [15]. However, most of these approaches are not very well suited to be used in a component based context for several reasons. First, components interact in a well-defined manner (e.g. JAVA Beans interact by posting events to interested listeners), so aspects should be able to declare joinpoints specific for the component model. For the JAVA beans component model, this means that it should be possible to declare joinpoints on events. Secondly, components come from different vendors and are not explicitly created to work with each other. In order to make the aspects reusable, the declaration of the aspect behavior has to be separated from the concrete interface of the base component. This means that it should be possible to declare abstract joinpoints in the aspect specification. At aspect weaving time, the abstract joinpints are connected to concrete joinpoints in the components. Finally, source code from third-party components is often not available, therefore source code weaving becomes unfeasible. In addition, source code weaving is also unsuited for enabling the dynamic weaving and unweaving of aspects.

To solve the problems described above, we envision a new aspect-oriented implementation language tailored for the component based field. The language will be able to specify joinpoints specific for the component model. Explicit and reusable connectors connect the abstract joinpoints in the aspect declaration to concrete joinpoints in the components. The aspects are weaved into the components using binary code adaptation techniques. We already conducted experiments in component adaptation for JAVA by directly acting on the byte code instead of the source code. This has resulted in a first prototype aspect-oriented implementation language. In a next phase, we plan to use this aspect-oriented programming language as an implementation for a composition adapter. In this way, we are able to specify concerns that alter the internals of a component at a component based design level. This enables a seamless integration with our current component based methodology.

7. ACKNOWLEDGMENTS

We owe our gratitude to Dr. Bart Wydaeghe who developed the component based methodology during his PhD research. We also want to thank him for his interesting feedback and participation in this research. In addition, we like to thank Prof. Dr. Viviane Jonckers for her invaluable help during our research and for proof reading this paper. Since october 2000 the author is supported by a doctoral scholarship from the Fund for Scientific Research (FWO or in flemish: "Fonds voor Wetenschappelijk Onderzoek").

8. REFERENCES

[1] Szyperski, C. (1997). *Component Software; beyond Object-Oriented Programming.* Addison-Wesley.

[2] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[3] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A. and Murphy, A. *Aspect-Oriented Programming.* In proceedings of the 19th International Conference on Software Engineering (ICSE), Boston, USA. ACM Press. May 1997.

[4] ITU-TS. ITU-TS Recommendation Z.120: *Message Sequence Chart (MSC).* ITU-TS, Geneva, September 1993.

[5] Wydaeghe, B. *PACOSUITE: Component Composition Based on Composition Patterns and Usage Scenarios.* PhD Thesis, available at: http://ssel.vub.ac.be/Members/BartWydaeghe/Phd/member_phd.htm

[6] Vanderperren, W. and Wydaeghe, B. *Towards a New Component Composition Process.* In Proceedings of ECBS 2001, April 2001.

[7] Wydaeghe, B. and Vandeperren, W. *Visual Component Composition Using Composition Patterns.* In Proceedings of Tools 2001, July 2001.

[8] Vanderperren, W. and Wydaeghe, B. *Separating concerns in a high-level component-based context.* EasyComp Workshop at ETAPS 2002, April 2002. To be published.

[9] Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Second ed. 2001.

[10] R. Douence, O. Motelet, M. Südholt *A formal definition of crosscuts.* Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns, LNCS.

[11] Lieberherr, K., Lorenz, D. and Mezini, M. *Programming with Aspectual Components.* Technical Report, NU-CCS-99-01, March 1999. Available at: http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html.

[12] Filman, R.E. *Applying Aspect-Oriented Programming to Intelligent Synthesis.* Workshop on Aspects and Dimensions of Concerns, 14th European Conference on Object-Oriented Programming, Cannes, France, June 2000.

[13] Kiczales G. et al. *An overview of AspectJ.* In Proceedings of the European Conference on Object-Oriented Programming, Budapest, Hungary, 18--22 June 2001.

[14] L. Bergmans and M. Aksit. *Composing Crosscutting Concerns Using Composition Filters.* Communications of the ACM, Vol. 44, No. 10, pp. 51-57, October 2001.

[15] H. Ossher and P. Tarr. *Multi-Dimensional Separation of Concerns and The Hyperspace Approach.* In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.

Run-time Support for Aspects in Distributed System Infrastructure

Eddy Truyen, Wouter Joosen, Pierre Verbaeten
DistriNet, Dept. Computer Science
K.U.Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
+32 (0) 16327602
{eddy, wouter, pv}@cs.kuleuven.ac.be

ABSTRACT
Adaptation of distributed system software to changes in the execution environment or user requirements by switching non-functional algorithms at run-time is powerful yet difficult to implement. Aspect-oriented programming is a necessary, but insufficient means to achieve this goal. This paper consists of two parts. First, we present what is in our opinion the best direction towards an AOP model that supports switching of aspectual algorithms at run-time. Second, we try to explain that an adequate AOP programming model is not enough to provide support for non-functional requirements. Architectural support is also important. In this regard, we explain how component frameworks and AOP can benefit from each other.

1 INTRODUCTION

The success of distributed system infrastructure such as middleware and application servers depends on their ability to *integrate support for non-functional requirements*. Furthermore, there is a growing need to build distributed systems infrastructure whose support for non-functional requirements can be *dynamically reconfigured* in order to adapt to *evolving context-specific needs* (application-specific requirements, preferences of end users, and characteristics of hardware platforms – see Figure 1). More importantly, the need for these reconfigurations occurs not only at load-time but also at run-time.

Traditional system infrastructure is however implemented as a black box. The implementation decisions taken by the system developers are locked within the black box and thus cannot be customized by application developers. If some of the implementation choices conflict with the needs of the application, there is only one option left: use another middleware platform that provides the appropriate support. This is bad, because it requires rewriting the code of the application and involves time-consuming learning of the new middleware platform.

We try to build system infrastructure that has an open implementation such that it can be dynamically customized to context-specific needs. However, the implementation of non-functional requirements is typically very hard to modularize, because they crosscut through multiple implementation units of as well the system infrastructure as the application itself. As such the programming of non-functional extensions would require invasive change into existing code and is complicated by the lack of locality. More importantly, this makes the run-time integration of non-functional extensions almost impossible. Therefore, a pre-requisite for making run-time integration of non-functional extensions possible is that software systems have a modular structure such that non-functional extensions *are compositional with this structure*. There exist many different ways to tackle this problem such as design patterns [4], architectural solutions [21], component-oriented programming [25], and domain-specific meta-object protocols [2]. With respect to this problem, we are however highly inspired by the aspect-oriented software development research that works on providing analysis, design, linguistic, and run-time mechanisms for capturing such concerns as modular units that would otherwise 'crosscut' [9] through multiple parts of the design and implementation artifacts.

Based on this view, we present the customization process of a system infrastructure as a *dynamic and selective combination of aspects* to a 'stable' core functionality. The core is stable in the sense that it is invariant for all applications and computing environments. For example, the core of an object request broker consists of the basic representation of objects and the communication of requests.

Each aspect implements a specific algorithm for a specific non-functional requirement. The semantics of customization process described by the phrase "dynamic and selective combination of aspects" consists of three parts.

- *dynamic* in the sense that aspects must be pluggable and unpluggable from the core system at run-time. For each non-functional requirement, there may

exist multiple alternative algorithms. Which algorithm is the best to use cannot be determined in advance, but depends on context-specific properties (end user preferences, characteristics of the computing environment, …). For example, when a drop of communication resources in the networking environment occurs, all aspects in the system may have to be switched to a variant algorithm that has a cheaper usage of the network. This kind of decisions can clearly only be taken at run-time.

- *selective* because we have to choose between alternative non-functional aspects; a decision which is context-dependent.
- *combination*, because multiple aspects may have to be applied at the same time. Aspects may be developed independent from each other by different people (see also section 3).

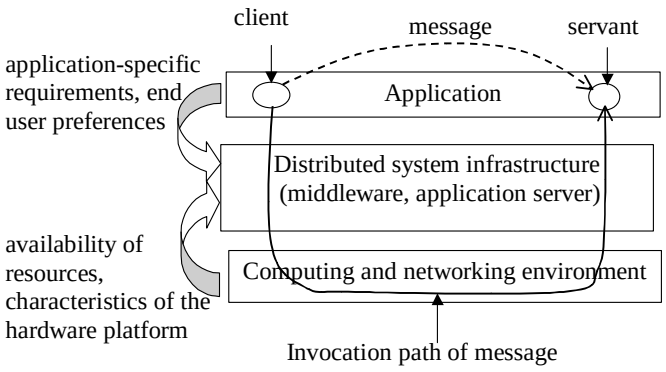


Figure 1 Terminology

2 REQUIREMENTS FOR A DYNAMIC ASPECT-ORIENTED PROGRAMMING MODEL

Customization of software to changes in the execution environment or user requirements by switching algorithms at run-time is powerful yet difficult to implement, especially in distributed systems. In the following sections we present our ideas about what's in our opinion the best direction towards an AOP model that supports such a dynamic switching of aspects.

Per-collaboration switching

Due to the crosscutting nature of non-functional aspects, the dynamic switching of non-functional aspects typically requires run-time adaptations all over the distributed system. As such a coordination mechanism is needed that involves doing the adaptation in a coordinated manner so that the integrity of the system is preserved while the adaptation is in progress. Two types of coordination may be needed: *inter-component coordination* that coordinates adaptation across the layers of the core system infrastructure on a given host, and *inter-host coordination* that coordinates adaptation across hosts in a distributed system [3].

We understand this problem better by looking at the messages circulating in a distributed system: a *client* sends a request to a *servant* by means of a *message* (see Figure 1). The underlying middleware core infrastructure performs some client-side processing and server-side processing along the *invocation path* of the message. Additional non-functional aspects may be applied along certain intercession points on this invocation path of the message. It is clear that once an algorithm is selected for processing a message, this decision must be propagated along the invocation path of the message, such that the same algorithm is applied consistently at the client and server-side, e.g. compression and decompression of the message data. So when the client-side middleware decides to switch from algorithm for compressing a specific message, the server-side must switch correspondingly to the new algorithm for decompressing that message. At the same time, however, it may be that messages arriving from other client hosts must be decompressed by yet another algorithm. We can only coordinate this in a good way by performing a dynamic and selective combination of aspects *on a per message basis*.

The need for consistent selection of algorithms may even be on a per collaboration basis (a graph of messages initiated by a client request) between multiple component instances of the application. The servant typically sends some new messages to other components when processing a client request. The point is that the non-functional aspects that were initially applied to the original client request may have to be applied for the entire collaboration. For example, when a client wants to perform some data-centric operation in a transactional manner, the collaboration that is triggered by this client request must also be executed in a distributed transaction.

Preserving, per-instance switching

The dynamic switching of aspects should also operate *at the instance level*. This is because approaches that operate at the class-level have a very restricted applicability at run-time, especially in the case of unanticipated adaptation (a notable exception to this, however, is the Rondo model [18]).

Instance-level approaches can be further classified as either *replacing* or *preserving*, depending on whether they replace an existing instance by its adapted version or let both be used simultaneously [10].

To make instance replacement safe, there must be a clear separation between the interfaces and the implementation of instances. As such, this will naturally lead to the use of components [25]. There are many reasons however why component instance replacement does not work well for the dynamic switching of algorithms in distributed systems:

- Component instance replacement does in general work not well when instances encapsulate state. This

is a problem, since middleware components often contain state that is very difficult to hand over between components due to data incompatibilities.

- Furthermore, middleware typically executes in a multi-threaded environment. Since, transfer of execution state between the old and new component instance is not possible in general, the execution of the original component must be suspended first, which is not desirable for distributed applications that tolerate no small disruption of service, e.g. audio/video streaming applications
- Component instance replacement is difficult to coordinate in a distributed system. For example, when a component must be replaced, there may still be messages circulating somewhere in the distributed system, that have to be processed by the original component. As such switching to the new component must be delayed until all these messages are processed. For instance, this problem occurs when replacing a data compression component while there are still messages in transmission over the network that were compressed with the original data compression component. To illustrate that this is a very difficult problem: In [3], one proposes a very complicated adaptation protocol that allows component instance replacements to be made in a coordinated manner across hosts.
- Finally for particular kinds of system infrastructure such as application servers, joint use of the original and the new component is often required. This is because these kinds of systems are at the same time used by multiple client applications that may want different adaptations to be applied to the system infrastructure. As such, an adaptation, performed on behalf of a specific client, might not be desirable for another client. As such the first client must be able to use the adapted component instance exclusively in its own context, while the other client still must be able to uses the original version of the component.

When component instances cannot be directly replaced from a running system, we are faced with the problem to change their behavior solely by adding more components. This leads us to the use of wrappers [4]. The wrapper-based approach enables unanticipated, run-time, instance-level adaptation, joint use of different versions of a component and easy modeling of components that present different interfaces to different clients [10]. In addition, wrappers support non-invasive integration of aspects to a minimal core component instance. Code needed for implementing an aspect can be completely encapsulated into the wrapper. Joint use of different combinations of aspects can be supported by disjunctive wrapper chains.

However, the usefulness of the wrapper approach in class-based programming languages is limited by the underlying object model. The most important problem is the lack of a common self or the so called object schizophrenia problem [25]. Existing works [10,13] have tackled the object schizophrenia problem that appears at the specialization interface of objects by introducing delegation (aka object-based inheritance) to class-based programming languages. In the terminology of [13], delegation of is a combination of acquisition and method overriding with transparent redirection.

However, in the presence of dynamic and selective combination of aspects another object identity problem is much more severe: the object identity problem that is apparent from outside the core object at its client interface: a wrapper cannot be transparently interposed between the core object and the client, because the wrapper has a separate object identity of its own [5]. This problem substantially aggravates with joint use of different wrappers (disjunctive wrappers) around the core object. Outside objects must maintain the references to these different wrappers, since the outside objects must *select*, for every message, through which wrappers the message should go through. Maintaining this indirection is tedious and hugely impacts the scalability and maintainability of the system [27]

The Lasagne model

We have already presented elsewhere [27] the Lasagne model that defines a component model for dynamic and context-specific combination of aspects on per collaboration basis. We will give only a short overview of Lasagne, discussing its main features. We implement each aspect as a set of wrappers that work together (e.g. at multiple core components, at client and server side, etc.) to implement an algorithm for a non-functional requirement. Lasagne uses a *dynamic wrapper model* [26] to support a selective combination of aspects on a per collaboration basis, while taking into account context-sensitive needs. A *composition policy* specifies the subset of aspects that must be applied for a specific collaboration. To make context-sensitive customization possible, the composition policy is controlled by *interceptors*, that are placed in the application and computing environment and allow expressing that an extension should be executed when a programmer-defined expression over the current context is true.

Lasagne effectively addresses the object schizophrenia problem apparent at the client interface. Furthermore, with respect to the specialization interface, Lasagne supports acquisition and method overriding, and optionally transparent redirection. Transparent redirection is actually not always a desired property. Generally speaking, transparent redirection makes object-based composition as fragile as inheritance in the sense that inheritance breaks the encapsulation of components [25, 23]. Therefore Lasagne allows the programmer to decide on a per component basis whether transparent redirection must be

turned on/off by playing with the *local composition policy* of that component. The local composition policy governs how the wrappers around the core component must be composed. In general, having control on the local composition policy allows dealing with so-called feature interaction problems when combining multiple aspects. (Some good examples to illustrate this have been given by Renaud Pawlak in [15] who developed the JAC framework that implements the dynamic wrapper model of Lasagne in Java). In fact, the breaching of encapsulation due to overriding with transparent redirection can be seen as a specific type of feature interaction problem.

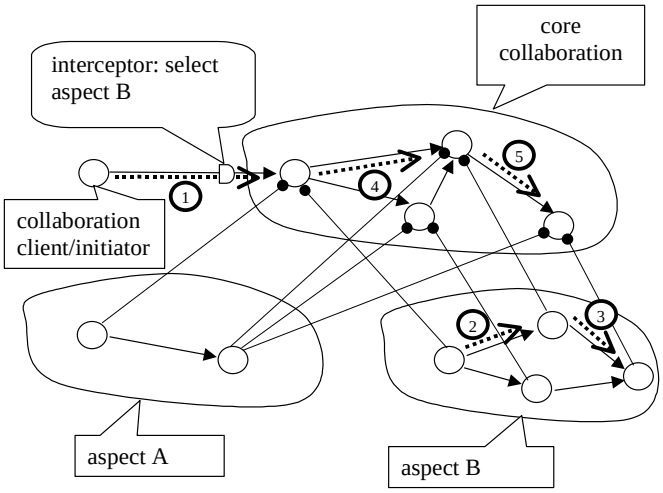


Figure 2 Per collaboration, per instance switching

Figure 2 shows a naive and simple representation of how our system works. All the plain arrows are constructed at the initialization time of the application. The plain arrows represent references that can be set while the core component instances are constructed. The arrows with round heads represent the wrapping link between a core component instance and a wrapper that wraps this component instance. These wrapping links are set at the weaving time by a particular part of the system called a deployer (or weaver to stay consistent to the AOP terminology).

One can see on the figure that the second and third interactions are made within the B aspect. Thus, the core collaboration that is $1 \rightarrow 4 \rightarrow 5$ is in fact refined, for this collaboration, by aspect B to produce the final collaboration $1 \rightarrow 2^{(B)} \rightarrow 3^{(B)} \rightarrow 4 \rightarrow 5$. The B aspect is activated for this collaboration since the interceptor that has been added to the application/computing environment selects the B aspect (otherwise the B aspect would have not been activated).

In our system, the aspect selection is context-sensitive. It means that the interceptor could have selected a different aspect depending on the current needs of the context. This feature allows the system to easily perform client-specific customizations and adapt to changing circumstances in the computing environment. For example, within another collaboration triggered by another client request, aspect A could have been activated instead of aspect B.

Dynamic combination of aspects on a per collaboration-basis introduces a big run-time performance overhead. As such, this can only be applied at the coarse-grained architectural level of a system (see section 3 about this). At a finer-grained scale, dynamic selection of aspects on a per message basis causes too much performance overhead to be acceptable in certain application domains. In that case, less dynamic wrapping models such as dynamic composition of aspects at object construction time [12] must be used, hoping that the inter-host coordination problems can be solved in another way.

3 COMPONENT FRAMEWORKS AND AOSD COMPLEMENT EACH OTHER

The good and the bad of prefabricated architectural knowledge

Adequate programming support is not sufficient to provide dynamic support for non-functional requirements in middleware. Architectural support is also important. For example, to support transactions, there must be a transaction manager readily available somewhere in the system. This is important architectural knowledge that should be prefabricated such that it can be reused for the implementation of different transaction algorithms.

Component framework technology [25, 17] supports this idea by providing the system developer with a semi-complete architecture that is tailored for a specific application domain or family of applications, incorporating support for only those non-functional requirements that are relevant for that specific application domain. Certain well-defined parts in this architecture, called *variation points* (also known as hot spots) [7], are left unspecified because they would expose important implementation details that would vary among particular, fully executable implementations. A *system implementation* provides the missing details by plugging an application-specific part into the generic architecture, filling up the variation points. Component frameworks are also often referred to as black-box frameworks that accept *"plug-in" components* [25] which correspond to the application-specific part provided by the system implementation. Each plug-in component must conform to the *size and kind* of the variation points. Component frameworks have indeed more and more been put forward as the key to realize dynamic configurable middleware and distributed applications [2,6,8,11,16,20].

There is however an anomaly with component frameworks that causes a lot of problems for customization. We observe that the architecture of a system is heavily characterized by the way components of the system interact with each other. A similar observation has been made a long time ago in the domain of software architecture [19]. Here a system is also factored into a number of components and connectors that encapsulate the interactions between these components. As a consequence, the key to component framework design is that component frameworks are able to effectively control the *collaborations* between the components within that system, i.e. they are able to determine how components interact with each other. However, the component framework 'fixes' the collaborations, while it leaves open the implementation of the components that are to participate in these collaborations. Unfortunately, this opposes the ability to customize system implementations. The cause of this problem is that components are normally not the most variable elements of a software architecture – the interactions between components are [1]. A customization nearly always involves *refining the interaction behavior* of core components, thus impacting the architecture of the system. An interaction refinement may require that additional component instances with unanticipated interfaces are introduced into the system, may require adding new interfaces to existing core components, or specializing the interaction behavior of existing components. However, it is exactly the interactions between components that are fixed by the component framework! To solve this anomaly, [1] proposes the concept of stratified architectures. This concept presents the architecture of a system as consisting of multiple strata. These strata are not layers in the normal sense (like for example in the ISO OSI model), but may actually contain the same object (component instance) as another stratum but with a wider interface reflecting the effects of an interaction refinement. As such, an object (thus a component) is able to morph its interface over the different strata at run-time. The result is that one can both have prefabricated architectural knowledge (in the form of an architectural stratum) and the possibility to tweak component interactions (by reconfiguring the composition of architectural strata).

In traditional object-oriented programming languages, there is no practical implementation of stratified architectures, due to the inability of an individual object or class to morph itself over the different strata. However, mixin layers [22] (and probably a lot of other existing AOSD methods) can be used to implement this concept. Stratified architectures can also be implemented by means of Lasagne, with the important difference that architectural strata can be composed at run-time. Variation points in the component framework are simply reified as light-weight meta-level objects that implement the Lasagne dynamic wrapper model (for more details see [26]). Lasagne wrappers support as well extending the interfaces of plugged-in components, as refining the interaction behavior of plugged-in components. Lasagne wrappers are in this way used for dynamically plugging and unplugging of architectural strata.

Component frameworks support independent extensibility

A second important contribution of component frameworks is support for independent extensibility. A system is called independent extensible if it can cope with the late addition of components without requiring a global integrity check [24]. In [28] one explains that "components can be developed by different people in complete ignorance of each other and these components will be combined later with other components developed in parallel. Individual components must be designed to allow for composition with other, unknown components. The only way to achieve this is to set up design rules for component developers in advance. These design rules are specifications of future components. They will lead to certain *common abstractions*. It is the purpose of component frameworks to implement these abstractions and to enforce obedience to the specification, as far as possible. This is particular important as far as global security is concerned. In theory, a comprehensive documentation of design rules to be obeyed by components would be sufficient to define these common abstractions. Such an approach does however not provide any safety. A component framework's designer should strive for abstractions that enforce the necessary behavioral constraints, or that at least allow detecting violations against them. The only practical tool currently available to implement these abstractions and to protect them against violations is information hiding behind interfaces."

The notion of independent extensibility is not well supported by current AOSD technologies [14] and this translates itself into so called feature interaction problems when combining multiple aspects. We believe however that the above idea of 'common abstractions' also provides a good starting point to tackle these problems.

In the context of wrapper-based approaches, the idea of common abstractions is at least related with the type-safe delegation principle formulated by [10] that is based on the existence of a *common parent type* shared between the wrapped component and the wrapper.

With Lasagne we go a step further: the reified variation points do not only define the common abstractions to be obeyed by the wrappers, but they can also implement a tailored local composition policy to solve feature interaction problems when combining multiple aspects.

4 ACKNOWLEDGMENTS

This research is supported by a grant from the Flemish Institute for the advancement of scientific-technological research in the industry (IWT). Thanks to Renaud Pawlak

for making Figure 2.

5 REFERENCES

1. C. Atkinson, T. Kühne, C. Bunse, “Dimensions of Component Based Development”, in Proceedings of the 4th International Worshop on Component-Oriented Programming.
2. G.S. Blair, G. Coulson, P. Robin, M. Papathomas, "An Architecture for Next Generation Middleware", Proc. IFIP International Conference on Distributed Systems.
3. Wen-Ke Chen, M.A. Hiltunen, R. D. Schlichting, “Constructing Adaptive Sofware in Distributed Systems”, in Proceedings of International Conference on Distributed Computing (ICDCS’2001), 2001, pp. 635-643.
4. E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns, Elements of Reusable Object-Oriented Software”, Addison-Wesley, ISBN 0201633612.
5. U. Hölzle, "Integrating Independently-Developed Components in Object-Oriented Languages, in Proceedings of ECOOP' 93, 1993, Springer-Verlag LNCS.
6. J. Hummes and B. Merialdo, “Design of extensible component-based groupware”, in Computer Supported Cooperative Work - An International Journal, 1998.
7. I. Jacobsen, M. Griss, P. Jonsson, “Software Reuse; Architecture, Process and Organization for Business Success”, Addison Wesley, 1997, ISBN 0-201-92476-5.
8. B. N. Jørgensen, E. Truyen, F. Matthijs, W. Joosen, “Customization of Object Request Brokers by Application Specific Policies”, in Proceedings of the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware2000), Springer-Verlag, 2000.
9. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwan, “Aspect-Oriented Programming”, in Proceedings of ECOOP’97, June 1997, Springer-Verlag LNCS 1241, pp. 220-242.
10. G. Kniesel, “Type-Safe Delegation for Run-Time Component Adaptation”, In Proceedings of ECOOP’99, June 1999.
11. F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, R. H. Campbell, “Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB”, in Proceedings of the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing (Middleware2000).
12. K. Ostermann, “Implementing Reusable Collaborations with Delegation Layers”, First Workshop on Language Mechanisms for Programming Software Components at OOPSLA 2001,
13. K. Ostermann, M. Mezini, “Object-Oriented Composition Untangled”, in Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’2001), 2001, pp. 283-299.
14. K. Ostermann, G. Kniesel, “Independent Extensibility – an open challenge for AspectJ and Hyper/J”, position paper for the ECOOP’2000 Workshop on Aspects and Dimension of Concerns, C. V. Lopes (ed.), 2000.
15. R. Pawlak, L. Seinturier, L. Duchien, G. Florin, “ Dynamic wrappers: handling the composition issue with JAC”, in Proceedings of TOOLS USA’2001.
16. F. Matthijs, “Component Framework Technology for Protocol Stacks”, Phd. Thesis, K.U.Leuven, ISBN 90-5682-224-1
17. T. D. Meijler, O. Nierstrasz, “Beyond Objects: Components”, in Cooperative Information Systems: Current Trends and Directions, M.P. Papazoglou, G. Schlageter (Ed.), Academic Press, November 1997, pp. 49-78.
18. M. Mezini, “Dynamic Object Evolution without Name Collisions”, in Proceedings of the ECOOP’97 Conference.
19. M. Shaw and D. Garlan, “Software Architecture”, Prentice-Hall, 1996, ISBN 0-13-182957-2
20. D. C. Schmidt and C. Cleeland, “Applying Patterns to Develop Extensible ORB Middleware”, Design Patterns in Communications, (Linda Rising, ed.), Cambridge University Press, 2000.
21. Douglas C. Schmidt, Michael Stal, Hans Rohert, and Frank Buschmann, Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, John Wiley and Sons, 2000.
22. Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”, in Proceedings of the ECOOP’98 Conference, July 1998, Springer-Verlag LNCS 1445, pp. 550-570.
23. P. Steyaert, and W. De Meuter, “A Marriage of Class-Based and Object-Based Inheritance Without Unwanted Children”, in Proceedings of ECOOP’95, 1995, Springer-Verlag, LNCS 952, pp 127-145.
24. C. Szyperski, “Independent Extensible Systems: Software Engineering Potential and Challenge”, in Proceedings of the 19th Australasian Computer Science Conference, 1996.
25. C. Szyperski, “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley, 1998, ISBN 0-201-17888-5.
26. E. Truyen, B. N. Jørgensen, W. Joosen, "Customization of Component-Based Object Request Brokers through Dynamic Configuration", in Proceedings of TOOLS Europe'2000, IEEE press, 2000.
27. E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten and B. N. Jørgensen, "Dynamic and Selective Combination of Extensions in Component-based Applications", in Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001), 2001.
28. W. Weck, “Independently Extensible Component Frameworks”, in Special Issues in Object-Oriented Programming, M. Mühlhäuser (ed.), dpunkt Verlag, 1997, pp. 177-183.

Security and Aspects: A Metaobject Protocol Viewpoint

Ian S. Welch
Department of Computing
University of Newcastle upon Tyne
i.s.welch@ncl.ac.uk

Robert J. Stroud
Department of Computing
University of Newcastle upon Tyne
r.j.stroud@ncl.ac.uk

ABSTRACT

In this paper we reflect upon the results of experiments that have attempted to use Metaobject Protocols to implement security as a crosscutting concern. As security is often cited as a crosscutting concern that could be implemented using Aspects we would like to point the way to some of the requirements that should be met by any aspect language used to implement security as a crosscutting concern.

1. INTRODUCTION

There is a mature body work on applying metaobject protocols to address security requirements. What can the Aspect community learn from this work, and what might it have to offer?

The motivation for using a metaobject protocol approach is that it allows security to be treated as a cross-cutting concern. This makes it possible to increase the flexibility of the enforcement mechanisms provided by the system infrastructure. Metaobject protocol researchers have had to consider problems such as how to ensure that their metaobject protocols provide complete mediation between subjects and objects as well as being tamperproof and verifiable. This work provides a viewpoint on some of the requirements that would have to be satisfied by an aspect language that was used to implement security enforcement.

How metaobject protocols have been used to treat security as a crosscutting concern is explained in section 2. We discuss what can be learnt from the experiments in 3. The advantages of an aspects approach is discussed in section 4. Future work such as carrying out a case study on a third-party application, and implementing an aspect language using an existing metaobject protocol are discussed in section 5.

2. BACKGROUND

A reference monitor is an example of an EM class mechanism [10]. EM class mechanisms enforce security policies by monitoring a target system and terminating any execution that is about to violate the security policy being enforced. As the enforcement decision is made on the basis of observing a single execution step this is limited to access control policies. Access control policies restrict what operations can be performed by programs on objects [7, 2].

Traditionally, reference monitors have been implemented as part of operating systems and have mediated access to operating system resources such as files or networking resources. However, there has been increasing interest in implementing reference monitors at the application level (see [11] for a good review). This has been achieved by using techniques such as program analysis and program rewriting. These techniques reduce performance costs as the enforcement code executes in the same address space as the program and so context switching is not required. Also access control policies can be formulated in terms of program abstractions as enforcement is applied at the interface between the program and the user. Another benefit is that since the reference monitor implementation is no longer hard coded within the operating system then the reference monitor implementation can be tailored to the types of policies it will enforce.

Advanced Separation of Concerns (ASoC) technologies provide a generic way to implement program rewriting for the implementation of security. In particular metaobject protocols provide a way to abstract away from the nitty-gritty of program rewriting and focus on changing behaviour rather than implementations. The enforcement code implemented by program rewriting can be seen as a crosscutting concern that can be separated out into aspects or metaobjects. In fact, prior to and concurrently with work on program rewriting there have been a number of experiments where metaobject protocols provide the technology to implement security as a crosscutting concern [6, 3, 8, 9, 1, 13, 5, 4]. A metaobject protocol allows the programmer to adjust the language semantics and implementation. Generally, each object has a metaobject that defines language semantics for that object. For example, a metaobject may define method execution for base level methods of the object. Changing the metaobject implementation results in changes to method execution for all methods of the object bound to the metaobject. The metaobject implementation may check whether method execution is allowed or denied by applying a security policy. Effectively, the metaobject can be seen to be as a fine-grained reference monitor.

ACP4IS '02 Enschede, The Netherlands

3. LESSONS FROM MOPS

Irrespective of whether the enforcement of a security policy is carried out by a reference monitor embedded in the infrastructure or a metaobject we can say that correct enforcement requires that the implementation of the reference monitor is tamperproof[1], always invoked, and (ideally) small enough to be subject to verification through analysis and testing. Note that we consider the implementation of the reference monitor we consider the trusted computing base which is the reference monitor plus the services it depends upon for correct operation (TCB) [12]. We use these requirements to structure our review of metaobject protocol experiments and what steps designers had to take to satisfy them.

3.1 Complete mediation

Correct enforcement of a security policy depends upon all accesses to the object being controlled by the reference monitor. In the context of a metaobject protocol this means that the metaobject must allow redefinition of all programming language semantics that allow access to an object's state or methods.

Experiments where only distributed object security is considered bring remote method invocation under the control of the metaobject protocol [3, 1]. The problem with this approach is that it is potentially vulnerable to another program on the same server using local method invocations to bypass the reference monitor.

Experiments with host-based object security such as [6, 5, 4] have only focused on controlling local method execution. Again this does not prevent other access routes to the object being exploited by a malicious attacker. Some researchers [8, 9, 13] have gone further and considered controlling state access, method sending, and exceptions.

Another aspect of complete mediation is protection of the object from attacks that break its encapsulation. Such an attack would result in the reference monitor being bypassed and the security policy not being enforced. As this attack takes place below the metaobject protocol abstraction it must be prevented through careful design of the metaobject protocol implementation itself, and careful choice of underlying runtime. Our own work [13] achieves *non-bypassability* through the rewriting of compiled code at loadtime and by relying upon Java's inability to forge pointers. Although we cannot claim this is totally non-bypassable we argue it is harder to bypass than other techniques that rely upon proxies to implement wrappers for existing classes.

Any aspect language that is used to implement security must provide the ability to advise on accesses to object state and methods, and non-bypassability should be implemented through a combination of clever design of the aspect language weaver and careful choice of underlying runtime.

3.2 Tamperproofness

Should the metaobject implementation or the metaobject protocol implementation be tampered with then there is no guarantee of correct enforcement of the desired security policy. Therefore both implementations should be protected against tampering.

The metaobject protocol implementation can be protected using operating system controls, for example only administrators have write access to the programs that the metaobject protocol implementation depends upon for their correct execution. The majority of the experiments took this approach. However, some experimental systems did go further. For example, Brilix [6] which uses a virtual machine to implement a metaobject protocol goes through a secure bootstrapping process at startup. Only the core bootstrapping code and a public key issued by the implementor is trusted. As the virtual machine implementation is digitally signed by the implementer using a secret private key before distribution then the integrity of the implementation can be checked using the public key at runtime.

Prior the execution there must also be assurance that the metaobject implementations have not been tampered with. Again operating system controls can be applied. However, Kava [14] and Brilix go further. As above code signing is used to check the integrity of code, in this case the integrity of the metaobject implementations.

An aspects implementation of security as a crosscutting concern will rely upon operating system controls to protect the aspect weaver implementation and the aspect implementations. For greater confidence then use of code signing techniques for both implementations would be advisable.

3.3 Verification

The smaller the reference monitor and TCB are then the more practicable it is to verify their correctness through analysis and testing. An advantage of a metaobject protocol approach is that since they have a standard and very general interface that it is possible to test them independently of the base level application. A disadvantage of the metaobject protocol approach is the size of the TCB. There are two reasons for a large TCB. First, the metaobjects themselves are coded using a high level language – usually the same language that is used to code the base level objects. Therefore the compiler must be included in the TCB. Second, the metaobject infrastructure that ensures that the metaobject is bound to a base level object may be very complex. In the experiments some relied upon specialised virtual machines, and others upon source preprocessors or binary rewriting tools. Clearly the virtual machines are complex and require a good deal of effort to verify. The source preprocessors and binary rewriting tools are simpler but in most cases are written using a general purpose high level language which again means that the language compiler becomes part of the TCB.

An aspects implementation should try and ensure that the aspects are independently testable and the aspects language and therefore the TCB is kept small. Ideally the aspects language should not be written in a general purpose high level language as this makes formal verification more difficult.

[1] In practice, as tamper-resistant as possible.

4. ASPECTS VS. MOPS?

Aspect languages offer two advantages over a metaobject approach. First, it may be possible to use a domain specific language for expressing security policies. Second, they offer weaving languages that are richer than metaobject protocol binding specifications.

4.1 Domain specific languages

The metaobject protocol experiments all used the same language for the base level and meta level. This meant that security policies were encoded using a high level language. This has the advantages that application abstractions (i.e. actual application classes) can be referred to directly in security policies and that programmers worked with a language they were already familiar with. On the other hand, more declarative languages are arguably more expressive, closer to the security designer's mental model, and make it easier to express the policy that the user wants without getting bogged down in implementation detail.

4.2 Weaving languages

Kava offers a declarative binding specification language. Generally the more complex the binding specification language then the more portable the metaobjects as they do not have to introspect on the base level in order to determine their context, for example our binding language removes the need for a metaobject to check which base level method it has intercepted at runtime. Also the more complex the binding language the greater confidence that all operations of interest will be brought under the control of the meta level. However, although our binding language is complex compared to other metaobject protocols it does not have the richness of expressivity of pointcuts in languages such as AspectJ. These include more sophisticated data flow analysis that could be very useful in identifying security policy relevant operations.

5. DISCUSSION

Experiments with metaobject protocols have provided results that can be applied to aspects. This is useful because in order to convince the security community of the worth of implementing security as a crosscutting concern it is essential to address complete mediation, tamperproofness and verification. The work done to date provides some ideas on addressing these problems. However, verification remains an open area. Verification may be made more tractable through the use of domain specific languages and the use of simpler languages for the implementation of weavers. Also, it is possible that work on formal models of aspects could provide a formal underpinning for verification.

An area for future work is to carry out a case study with a third party application. This is important because the experiments referred to in this paper tend to be small scale and to be applied to applications that have been designed by the experimenters.

A more speculative area for future work is to implementing an aspects language using a metaobject protocol. This would have the advantage of taking a design that addresses some of the requirements introduced in section 2 and would allow us to gain some of the advantages associated with aspect languages covered in section 4. In addition, the use of a metaobject protocol such as Kava would provide an aspect language with native support for loadtime aspects.

Acknowledgements

This work has been supported by the UK Defence Evaluation Research Agency, grant number CSM/547/UA and also the ESPIRIT LTR project MAFTIA.

6. REFERENCES

[1] M. Ancona, W. Cazzola, and E. B. Fernandez. Reflective Authorization Systems: Possibilities, Benefits and Drawbacks. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume LNCS 1606, pages 35–49. Springer-Verlag, 1999.

[2] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), October 1972.

[3] M. Benantar, B. Blakley, and A. J. Nadain. Approach to Object Security in Distributed SOM. *IBM Systems Journal*, 35(2), 1996.

[4] D. Caromel, F. Huet, and J. Vayssiére. A Simple Security-Aware MOP for Java. In *Metalevel Architectures and Separation of Crosscutting Concerns, Third International Conference, REFLECTION 2001*, volume LNCS 2192, pages 118–125, Kyoto, Japan, 2001. Springer-Verlag.

[5] D. Caromel and J. Vayssiere. Reflections on mops, components and java security. In *ECOOP2001 – Object-Oriented Programming, 15th European Conference*, volume LNCS 2072, pages 256–274, Budapest, Hungary, 2001. Springer-Verlag.

[6] W. E. K. Hermann Härtig, Oliver Kowalski. The BirliX Security Architecture. *Journal of Computer Security*, 2(1):5–21, 1993.

[7] B. Lampson. Protection. In *5th Princeton Conf. on Information Sciences and Systems*, Princeton, 1971. Reprinted in ACM Operating Systems Rev. 8, 1 (Jan. 1974), pp 18-24.

[8] T. Riechmann and F. J. Hauck. Meta objects for access control: extending capability-based security. In *New Security Paradigms Workshop*, pages 17–22, Langdale, Cumbria, United Kingdom, 1997. ACM.

[9] T. Riechmann and F. J. Hauck. Meta objects for access control: a formal model for role-based principals. In *New Security Paradigms Workshop*, pages 30–38, Charlottesville, VA USA, 1998. ACM.

[10] F. B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[11] F. B. Schneider, J. G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics*, pages 86–101, 2001.

[12] US Department of Defense. DoD Trusted Computer System Evaluation Criteria (The Orange Book). Technical Report DOD 5200.28-STD, US Department of Defense, 1985.

[13] I. Welch and R. Stroud. Using Reflection as a Mechanism for Enforcing Security Policies in Mobile Code. In *ESORICS'00 - 6th European Symposium on Research in Computer Security*, volume LNCS 1895, Toulouse, France, October 2000. Springer-Verlag.

[14] I. Welch and R. Stroud. Kava – Using Byte-Code Rewriting to Add Behavioral Reflection to Java. In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, pages 119–130, San Antonio, Texas, 2001.