Dynamic Aspects as Observers

Vaidas Gasiunas, Mira Mezini, Klaus Ostermann, Ivica Aracic
Software Technology Group
Darmstadt University of Technology, Germany
{gasiunas, mezini, ostermann, aracic}@informatik.tu-darmstadt.de

ABSTRACT

Modularization of software is driven by the principle of separation of concerns, which often does not correspond to the direction of calls between modules. Therefore observation techniques play important role for avoiding cyclic dependencies between modules. Object-oriented observation mechanisms require too much design and implementation effort and still do not provide good modularity properties. These problems are solved by aspect-oriented languages, which enable clean separation of component observation concern from its main functionality. The paper demonstrates how the static aspect-oriented solution can be improved further using flexible aspect instantiation and deployment, which is provided by CaesarJ. The presented techniques remove the gap between pointcuts and observer objects, enable observation of remote objects and flexible control over observation scope.

1. INTRODUCTION

In properly modularized systems dependencies between modules are acyclic, because cyclic dependencies bring a lot of disadvantages [17]. If modules stand in a cyclic dependency, none of these modules can be understood in isolation, none of them can be reused without the other. A change in one of the modules can potentially destabilize all the other. It is not possible to distribute and deploy them in separation. Having in mind all these disadvantages of cyclic dependencies, it is not a surprise why software design is so much about breaking them. In the further discussion if module A depends on module B, we will call B as a component and A as its client.

In traditional programming languages, dependencies between modules are determined by the direction of routine calls. However, possibility to modularize systems exclusively in the direction of the control flow of the program is too limited, because there are other criteria for modularization, which often crosscut the control flow. There is a need to have a call, which direction opposite to a dependency between modules, i.e. call from component to client. Such calls are known as callbacks. Procedural languages realized them directly by callback routines.

Object-oriented programming languages achieve a similar effect by means of interfaces and inheritance. The component accepts certain interface and client implements it by a specific class. In such solution callback routines are methods of objects. This gives certain advantages. Firstly interfaces can define a group of callback types, and therefore it is easier to manage them as each callback type separately. Secondly, callback routines most often need some specific data from client side: parameters and references to relevant client objects, however they run in a static context and can collect only information from global variables. In object-orientation callback routines always run it a context of object, which provide them with necessary client specific data. Such solution is easier to manage and also type safe.

Nevertheless, all object-oriented callback patterns, such as Observer, Command [8], Events [2] and other Publisher-Subscriber interactions [7] still require special preparations for callbacks on the component side. Only aspect-oriented languages make software modularization completely independent from the call directions and enable clean separation of components from their observation concern. However, in order to replace object-oriented patterns, aspects must be more flexible. They must be more tightly related with other application objects. It must be possible to control their lifecycle and application scope at runtime.

The contibution of the paper is twofold. Firstly, it explains how dynamic aspects could be applied to solve observation problems in a better way than using traditional object-oriented approaches or static aspects. Secondly, the paper presents the flexible aspect deployment mechanism in CaesarJ, which enables application of aspects on various dynamic scopes ranging from individual objects to remote processes as well as provides possibility to control this scope at runtime.

The paper is organized as following. Sec. 2 outlines the problems with object-oriented observation patterns. In Sec. 3 we explain how AspectJ[14] aspects solve the stated problems. Sec. 4 shows how dynamic aspect features in CaesarJ can further improve the AspectJ solution by removing the gap between pointcuts and observers, enabling observation of remote objects and flexible control over observation scope.

Sec. 5 explains implementation of aspect support in CaesarJ. Sec. 6 discusses related work and and Sec. 7 gives a summary.

2. OBSERVATION PROBLEMS

Since component functionality does not depend on the functionality of its clients, usually components are designed and implemented before their concrete clients are known. It is also often the case that new clients appear during system evolution. In object-oriented applications we have to build-in the notification mechanisms in advance, if we do not want to change our component with emergence of new clients.

To implement a notification mechanism we have to design event types, observer interfaces, implement their registration mechanism. At all places were event occurs we have to notify all registered observers, collect information about the event and pass it to observers. As we can see such kind of functionality requires a significant implementation effort.

In order to reduce implementation effort and complexity overhead, we try to foresee what kind of component state changes and other types of events can be of interest to its potential clients. Even more difficult is to decide what kind of information about the event is required. In case of data observation, designer must balance between the tradeoffs of "push" and "pull" models [8]: between the efficiency and the simplicity of the notification mechanism and the efficiency of the observer update implementation. However it is very difficult to reason about the influence of one or another design decision on the efficiency of observers, without knowing the concrete observers.

Need for notification is often client specific. Therefore, if we somehow manage to define notification mechanism, which fulfills the needs of all potential clients, most probably it will be too complicated for individual clients. They will get notifications about events in which they are not actually interested in. They will receive extensive information about an event, but use only a small part of it or won't use it at all. It also can be that some of the events, which we considered as useful when designing our component will be not interesting for clients at all.

After several development iterations, numerous refactorings we will probably find a balanced notification mechanism for the component. We will have a lot of types of events, observer interfaces; the component will be full of notification code, the purpose of which won't be possible to understand without knowing the specific needs of its clients. The notifications will be tangled within the main component functionality. Notification functionality is often crosscutting. The same types of events happen at several places of the component. So we will have to repeat notification code and it will be difficult to locate all notifications of a certain type of event.

2.1 Example

For illustration of observation problems, let's consider a simplified version of a project management system (PMS). The core model (Fig. 1) of the system defines a project as a set of tasks, assigned to certain users. Some of the tasks would be related to output documents. There are various relations between tasks and various user roles in projects.

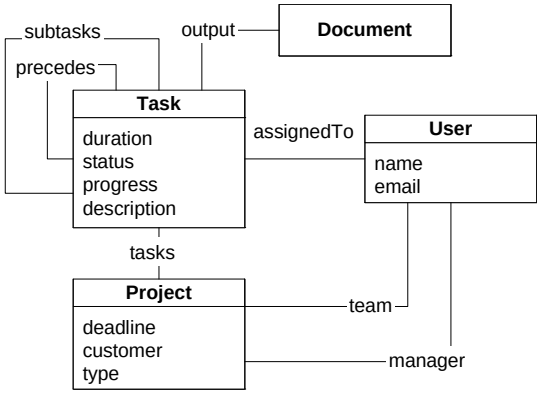

Figure 1: Project Model

The component, implemeting the project model, will be used for a lot of other functionality of the PMS (Fig. 1), such as for example, project tracking, resource management, reporting, billing, and so on. Concrete project management needs and practices vary from company to company, from project to project, therefore the project model component should be kept completely independent of its clients.

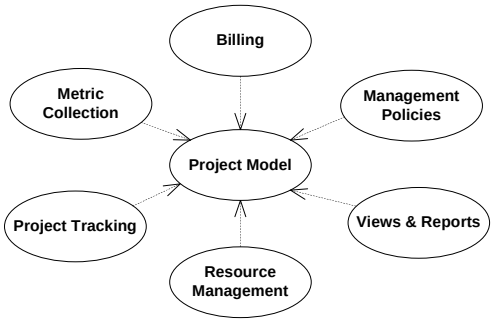

Figure 2: Sample Clients of Project Model

As we don't want to make any assumptions about project model clients, we may decide to implement as rich an observation mechanism as possible. So we define observer interfaces for each class in the model (List. 1) and include methods for all kind of events. Then we also need to implement observer registration and notification in the model classes.

```
interface TaskListener {
    void taskDurationChanged(Task t,
            double oldDur, double newDur);
    void taskProgressChanged(Task t,
            double oldPr, double newPr);
    void taskStateChanged(Task t, int oldSt, int newSt);
    void taskOutputChanged(Task t,
            Document oldDoc, Document newDoc);
    ...
}
interface ProjectListener {
    void taskAdded(Project p, Task t);
```

```
    void taskRemoved(Project p, Task t);
    void projManagerChanged(Project p,
            User oldMgr, User newMgr);
    ...
}
...
```

Listing 1: Interfaces for Observing Project Model

Despite of all this effort, the observation system will be far from ideal. For example, project tracking classes (List. 2) will only be interested in the schedule and task progress changes. Nevertheless it will receive all events, defined in the observer interfaces.

```
class CriticalChainTracking
    implements TaskListener, ProjectListener {
    double usedBuffer;
    Task chainTasks[];
    void taskProgressChanged(Task t,
            double oldPr, double newPr) {
        if (taskInChain(t)) {
            usedBuffer += oldPr - newPr;
        }
    }
    void taskOutputChanged(Task t,
        Document oldDoc, Document newDoc) {
        /* not interested in the event */
    }
    void taskAdded(Project p, Task t) {
        recalculateChain(t);
        t.registerListener(this);
    }
    ...
}
```

Listing 2: Tracking Critical Chain

A view, displaying project tasks (List. 3) may be interested in most of changes in the project and its tasks, but it does not need to distinguish between different types of changes, because it reacts to them in the same way. The rich observer interfaces are just unnecessary complexity for this client.

```
class ProjTasksView extends TableView
    implements TaskListener, ProjectListener {
    Project proj;
    void taskProgressChanged(Task t,
        double oldPr, double newPr) {
        repaint();
    }
    void taskOutputChanged(Task t,
        Document oldDoc, Document newDoc) {
        repaint();
    }
    void taskAdded(Project p, Task t) {
        repaint();
        t.registerListener(this);
    }
    ...
}
```

Listing 3: Project Tasks View

3. OBSERVING WITH STATIC ASPECTS

Aspects give a fundamentally new freedom for modularizing our systems by providing powerful mechanism to implement backward links from component to client. The advanced pointcut languages can intercept almost all points in the control flow of the component, at which the events of interest can occur. They also allow to collect various kind of information about the event: the affected object, its state, method parameter values and even some meta-information.

An important advantage is that all this can be done outside the component. This lets us to change the way how we design interaction between a component and its clients. It is not anymore the responsibility of the model to notify all its clients, but the responsibility of a client to collect all the events it needs. This gives a major advantage with respect to our stated problems.

For example, we would define separate aspects for different kinds of observation of project model. An aspect for *ProjectTaskView* update (List. 4) will observe all events about project and task changes. The aspect for critical chain tracking (List. 5) will observe only the changes in project schedule and task progress.

```
aspect ProjTasksViewUpdate { ...
    pointcut taskChanged(Tasks task) :
        execution(void Task+.set*(..)) && this(task);

    after(Task task) : taskChanged(task) {
        updateAllViews(task.getProject());
    }

    pointcut projectChanged(Project proj) :
        execution(void Project+.set*(..)) &&
        execution(void Project+.add*(..)) &&
        execution(void Project+.remove*(..)) &&
        this(proj);

    after(Project proj) : projectChanged(proj) {
        updateAllViews(proj);
    }
}
```

Listing 4: Project Tasks View Update with AspectJ

```
aspect CriticalChainTracking { ...
    pointcut progressChanging(Task task, int newPrg) :
        execution(void Task+.setProgress(int)) &&
        this(task) && args(newPrg);

    before(Task task, int newPrg) :
        progressChanging(task, newPrg) {
        updateChainBuffers(task.getProject(), task, newPrg);
    }

    pointcut scheduleChanged(Project proj) :
        execution(void Project+.addTask(..)) &&
        execution(void Project+.removeTask(..)) &&
        this(proj);

    after(Project proj) : scheduleChanged(proj) {
        recalculateChains(proj);
    }
}
```

Listing 5: Tracking Crical Chain with AspectJ

The possibility to add new ways of observation later on solves the preplanning problem. We can develop the component without thinking about the ways it can be observed. We don't need to decide what kind of state changes and method calls are of interest to its clients, what kind of additional information about the events should be supplied. Components are always open for observation with aspects, and the aspects can collect the information of interest themselves. We also avoid unnecessary implementation effort. The observation code is added only when it is really needed.

Aspects enable defining client specific ways of observation. We can define as many ways to observe the same model as we need without changing a single line the model implementation. Each client can intercept only the events, which it really needs, and collect only the relevant information. It is completely unaware of the ways other clients observe the component.

With aspects we can achieve better modularization. The component module needs to implement only the functionality, which is conceptually inherent to the component, and does not need contain any client specific code. Then we don't need to know about the clients of the component in order to understand it. The way a client observes a component is rather a part of the client's logic; therefore, we increase logical cohesiveness by including the implementation of observation to the client module. If we remove the client, its observation logic is also removed, because it does not serve any purpose to the component or to other clients of it.

Pointcut languages can also express the notification more concisely. Instead of inserting the same notification functionality in all places where an event occurs, a pointcut can specify all these places by a single expression and the corresponding advice would implement the notification functionality for this in one place.

4. OBSERVING WITH DYNAMIC ASPECTS

4.1 Gap between Pointcuts and Observers

The problem of aspects in AspectJ[15] is that they cannot be flexibly instantiated and therefore cannot be used instead of observer objects. For example, an observer implementing a view may be created, when user clicks on a certain toolbar button, and closed, when user clicks on the close button of the view. In a lot of situations we may need to create multiple instances of the same type of a view. We may need that these instances exist at the same time and observe the same events. Possibilities to control the lifecycle of the aspect instances in AspectJ is too limited for this example.

AspectJ aspects cannot be instantiated explicitly using *new()* keyword. By default all aspects are singletons and contain one instance, which exists all the time. Additionally, per control flow aspect instantation in AspectJ provides possibility to have different instances of the aspect in different joinpoints of the control flow. There are also two per object aspect instantiation methods: either per this or per target objects of the joinpoints of a certain pointcut.

All these instantiation methods are not sufficient. Singleton aspects do not support multiple instances at all. Per control flow aspect does not support existance of multiple instances at the same time. Per object instantiation method is the most flexible, because it lets to relate the lifecycle of an instance of the aspect with a certain object of a class. However, such aspect instance can intercept only the joinpoints of that object. For example, if we instantiate aspect per each view object, it can intercept only the joinpoints of the view, but not the joinpoints of the model, which it needs to observe.

It means that aspect instances cannot be the final observers, and an infrastructural code must be written to link aspect instances with the observer instances and delegate them the notifications about the intercepted events. For example, method *updateAllViews* in List. 4, must ensure that messages are further delegated to views, which observe the project, where the change happened. Therefore the aspect must contain knowledge about views that observe a certain project. The same problem is relevant to project tracking aspect of List. 5. The aspect must contain information about the critical chains of all projects, which use this tracking method. It means, that the aspects must contain registration mechanism for the final observers, and delegate notifications to them as for example in List. 6.

```
aspect ProjTasksViewUpdate { ...
   HashMap projTaskViews;
   void registerView(Project proj, ProjTasksView view) {
      List views = (List)projTaskViews.get(proj);
      if (views == null) {
         views = new LinkedList();
         projTaskViews.put(s, views);
      }
      views.add(view);
   }
   void unregisterView(Project proj, ProjTasksView view) {
      /* ... */
   }
   void updateAllViews(Project proj) {
      List views = (List)projTaskViews.get(proj);
      if (views != null) {
         Iterator iter = views.iterator();
         while (iter.hasNext()) {
            ((ProjTasksView)iter.next()).repaint();
      }
   }
}
```

Listing 6: Notifying Observers in AspectJ

The proposed reusable Observer pattern implementations with AspectJ [10, 19] cover only a specific case, when all events are handled uniformly and no context information from joinpoints is required to the observer objects. For example it is not the case for critical chain tracking, where are at least two types of events, which are handled differently.

4.2 Aspect Objects

The problem of mismatch between pointcuts and observers is solved in Caesar by unifying aspects and classes. Pointcuts and pieces of advice can be included to any class declaration. Aspects in this case are simply objects, which have pointcuts and advice. Considering aspects as objects has several implications. First of all, such aspects can be freely instantiated at any time in the application. There can be multiple instances of the same aspect type with different runtime state. Secondly, the pointcuts and advice are owned by objects. They operate in the context of their owner object, so they can access its state and call its methods.

Objects, supplied with pointcuts and advice, can themselves define the necessary observation. For example, we can implement the update of *ProjTaskView* by supplying it with corresponding pointcuts and pieces of advice (List. 7).

```
cclass ProjTasksView extends TableView { ...
   Project proj;

   pointcut taskChanged(Tasks task) :
      execution(void Task+.set*(..)) && this(task);
```

```
    after(Task task) : taskChanged(task) {
        if (this.proj == task.getProject()) {
            repaint();
        }
    }

    pointcut projectChanged(Project proj) :
        execution(void Project+.set*(..)) &&
        execution(void Project+.add*(..)) &&
        execution(void Project+.remove*(..)) &&
        this(proj);

    after(Project proj) : projectChanged(proj) {
        if (this.proj == proj) {
            repaint();
        }
    }
}
```

Listing 7: Project Tasks View in Caesar

In this way, we remove the gap between the pointcuts, defining observation, and the actual observer objects. The pointcuts immediately trigger the advices of the object. The advices can additionally check the relevance of the event using dynamic state of the object and immediately execute appropriate functionality to react to the event. Such solution preserves the encapsulation, because pointcut and advice being a part of the object can safely access its internal state. Pointcuts and advice can be viewed as implementation details of an object, used to observe other objects, processes and systems. The observation logic is encapsulated. A client can create an object and use it without knowledge of the structure which it observes. The client is freed from the duty to register the object to the appropriate notification services or to pass everything what the object needs from the context. The object can use pointcuts and advice to collect all context information it needs.

4.3 Aspect Deployment

Differently from AspectJ[14] aspects, Caesar aspect objects must be deployed to activate their pointcuts and updeployed to deactivate them. In this way, we get flexible control over aspects. Aspects are instantiated and deployed only when they are needed, and undeployed when they are not needed anymore. The same aspect object can be deployed and undeployed several times, so its state is preserved between different periods of activation. For example, in List. 8, when a view is created, it activates its pointcuts. When a user hides the view, the pointcuts are deactivated, while the object still exists. When the user decides to switch on the view again, its pointcuts can be again activated.

```
cclass ProjTasksView extends TableView { ...
    void init() { ...
        DeploySupport.deployLocal(this);
    }
    void hide() { ...
        DeploySupport.undeployLocal(this);
    }
    void show() { ...
        DeploySupport.deployLocal(this);
    }
    void close() { ...
        DeploySupport.undeployLocal(this);
    }
}
```

Listing 8: Deploying Aspect Objects

It may seem that in a lot of cases, it would be more convenient when aspects are implicitly deployed when the aspect instance is created and updeployed when it is not referenced anymore. However it is not sufficient, because an aspect object should not be destroyed when it is not referenced. It can be still reachable through a pointcut. There are aspects that just observe certain events and react to them. Therefore, it is necessary to stop this observation explicitly by undeploying such aspects.

4.4 Remote Deployment

In multithreaded and distributed applications, the scope of the aspect activation cannot be fully determined by the points in the control flow where aspect is deployed and undeployed. Certain aspects may need to observe only the events in a single thread. For exampleb a progress bar or a trace may need to observe only a single process running on a certain thread. Other aspects may need to observe events on all threads. For example, a view must be updated after any change in the model, independent of thread that caused it. Therefore, Caesar provides different aspect deployment strategies. An aspect object can be deployed on the current thread, on the entire JVM process, on a remote process or on a synchronous control flow, which crosses process boundaries.

The project management system normaly runs in a distributed multiuser environment. The project model will reside on an application server, while various views and notifications will operate on the client machines. In such a case, local aspect deployment, as shown in List. 8 is not appropriate. Project task view must be instead depoyed on the remote process, where the project model objects are created. At first, we must enable aspect deployment in the server process using the RMI address, which identifies the server as shown in List. 9. In this way ,we instruct to create and publish the object, which accepts aspect deployment requests. Then, on the client side we use the same RMI adress to deploy aspect objects on the remote server (List. 10).

```
cclass ProjectServer {
    static void main(String args[]) {
        ...
        CaesarHost host = new CaesarHost(
                "rmi://mycompany.net/projectserver/");
        host.activateAspectDeployment();
        /* create and publish remote objects */
        ProjectList projList = loadProjectList();
        host.publish(projList, "ProjectList");
        ...
    }
    ...
}
```

Listing 9: Enabling Aspect Deployment on a Server

```
cclass ProjectClient {
    static CaesarHost host = new CaesarHost(
            "rmi://mycompany.net/projectserver/");
    static CaesarHost getHost() {
        return host;
    }
    ...
}

cclass ProjTasksView extends TableView { ...
    void init() { ...
        ProjectClient.getHost().deployAspect(this);
    }
```

```
    void hide() {  ...
        ProjectClient.getHost().undeployAspect(this);
    }
    ...
}
```

Listing 10: Deploying Project Tasks View on Remote Host

Remote aspect deployment in Caesar is running over Java RMI. RMI uses generated stub classes for transparent communication with remote objects. We have developed a specialized RMI compiler, which generates stubs for Caesar classes. Classes, which are used or deployed remotely must be prepared by this tool. Differently from standard Java RMI, the Caesar RMI compiler does not require specially prepared remote interfaces. The stub can be generated for any Caesar class. This makes remoting in Caesar more transparent and easier to use.

4.5 Controlling Observation Scope

Standard aspect deployment strategies can limit the scope of observation to certain hosts or threads. These scopes are well suited to implement non-functional requirements or to trace certain processes, but they are too broad for most common observer aspects. For example, classical object-oriented observers register to individual subject objects. For example, the project task view in List. 3 registers to one project and all its tasks. On the contrary, the static aspect-oriented solution intercepts joinpoints in all project and task objects and only the advice filters out the relevant events.

In fact, the project task view as well as other project observers are only interested in changes in only one project. Therefore, we must limit observation scope to one project and its dependent objects. The observation scope can be limited by defining a custom deployment strategy. The CaesarJ runtime library provides a predefined class *PerObject-Deployer*, which implements object-based scoping. The class implements aspect deployment on objects, which identify the scope. The subclasses of this class need only to determine concrete object scopes by calling *setScopeObject()* method when the scope changes. For example, project scope could be implemented by a class in List. 11. The class intercepts all jointpoints on projects and their dependent objects and marks the currently active project.

```
public cclass PerProjectDeployer extends PerObjectDeployer {
    before(Project proj): execution(* Project.*(..)) && this(proj) {
        setScopeObject(proj);
    }
    before(Task task): execution(* Task.*(..)) && this(task) {
        setScopeObject(task.getProject());
    }
    private static PerProjectDeployer singleton = null;
    public static PerProjectDeployer instance() {
        if (singleton == null) {
            singleton = new PerProjectDeployer();
            DeploySupport.deployLocal(singleton);
        }
        return singleton;
    }
}
```

Listing 11: Per Project Deployment Strategy

We can use the singleton instance of *PerObjectDeployer* to deploy it on the projects, created in the local JVM process. However, views run on the client side and needs to be deployed on remote project objects. We can achieve this by publishing the *PerProjectDeployer* instance for remote access and use it on the client side. Listing 12 demonstrates deployment of *ProjTasksView* on a remote project object.

```
cclass ProjectServer {
    static void main(String args[]) {
        ...
        CaesarHost host = new CaesarHost(
                "rmi://mycompany.net/projectserver/");
        ...
        host.publish(PerProjectDeployer.instance(),
                        "PerProjectDeployer");
        ...
    }
    ...
}

cclass ProjectClient {
    static CaesarHost host = new CaesarHost(
            "rmi://mycompany.net/projectserver/");
    static PerProjectDeployer getPerProjectDeployer() {
        return (PerProjectDeployer)host
                        .resolve("PerProjectDeployer");
    }
    ...
}

cclass ProjTasksView extends TableView { ...
    Project proj;
    void init() {  ...
        ProjectClient.getPerProjectDeployer()
                        .deployAspect(this, proj);
    }
    void hide() {  ...
        ProjectClient.getPerProjectDeployer()
                        .undeployAspect(this, proj);
    }
    ...
}
```

Listing 12: Deploying on Remote Project

Such specialized deployment strategies have two advantages. Firstly, they provide a more efficient solution, because the observation scope of aspect object is limited to only relevant joinpoints. This is especially important for distributed applications, where each advice activation causes a remote call. In our example, the reduction of remote calls is proportional to the number of projects in the system.

```
cclass ProjTasksView extends TableView { ...
    pointcut dataChanged() :
        execution(void Project+.set*(..)) &&
        execution(void Project+.add*(..)) &&
        execution(void Project+.remove*(..)) &&
        execution(void Task+.set*(..));

    after() : dataChanged() {
        repaint();
    }
}
```

Listing 13: Simplified Project Tasks View

The second advantage is that custom deployment strategies reduce the complexity of aspect classes, because they don't need to filter out the relevant events anymore. For example, the observation in *ProjTasksView* is reduced to a single joinpoint and a trivial advice (List. 13). The deployment strategy class is reusable and can be used for deployment of

any other aspect objects, which require observation in the scope of a single project.

5. IMPLEMENTATION

Aspectual objects and flexible aspect deployment are features of CaesarJ programming language, which also implements virtual classes [16] and family polymorphism [6]. All these features together enable integration of reusable components into existing applications [18] and flexible variability management [20].

CaesarJ supports AspectJ[14] style pointcuts and advice. They are woven statically at compile time using AspectJ weaver. Aspect deployment framework is built on top of static aspects. For each Caesar class, containing pointcuts and advice, a registry class is generated, which is actually a conventional AspectJ aspect. The pointcuts from the Caesar class are moved to the registry class, while the advices are transformed to simple class methods. The registry class also provides methods for registration of the corresponding Caesar class objects.

Since registry classes are AspectJ aspects, we use AspectJ weaver to weave them into application code. So the joinpoints of the woven code contains calls to the advices of the registry classes. The advices of a registry class on their turn delegate the call to the registered aspect objects. In this way an advice call can be distributed to any number of objects. For more detailed information about implementation of aspect registries refer to [9].

Implementation of aspect deployment strategies is based on manipulations with aspect registration. Each type of deployment defines a special collection class that determines, which of the deployed objects must receive advice call in the current execution context. Simple deployment strategy notifies all deployed objects, while thread based deployment strategy notifies only the objects, which are deployed on the current thread. Each deployment strategy also define a deployer class, which instantiate the appropriate collection and sets up it in a registry class, when an aspect is deployed using this deployment method. There is also a special collection class which can aggregate over other collections and in this way enables simultaneous usage of multiple deployment strategies for the same aspect class.

All deployment strategies are defined outside Caesar compiler in the runtime library. The compiler is only responsible for generation of the registry classes and class methods, which ensure that an aspect object is also deployed to the registries of its superclasses. The classes in the runtime libraries can be further extended for new types of deployment strategies. In List. 11 we have seen how custom per-object deployment strategies can be defined.

The current implementation provides moderate performance characteristics. Weaver inserts advice calls only at joinpoints, which are referenced by the aspects in the application. If no aspect is deployed at a joinpoint, this causes one redundant static method call and one field check for the null value. The collections of aspect deployment strategies use hash tables to determine the aspect objects, which must be notified at the joinpoint. In this way unnecessary iterations are avoided. We believe that dynamic aspect deployment can be implemented even more efficiently using run-time weaving techniques [3].

6. RELATED WORK

A reusable Observer design pattern implementation is provided by Event library [2], which is based on Eiffel agents [5]. The library reduces implementation effort of registration and notification infrastructure. However other problems remain, for example the library still requires preplanning of event types and event triggering code is still tangled in the component implementation. New event types cannot be added without modifying the component class.

In [10] Hanneman and Kiczales propose reusable implementation of Observer design pattern with AspectJ. The implementation enables clean separation of observation concern from the observable subject. The problems with this solution are analized in [19]. The paper also presents a more reusable and conceptually clean solution of the Observer design pattern using Caesar. The proposed solution also allows multiple instantiation of Observer pattern on the same classes for different kinds of observation. The reusable implementations of Observer consider only the classical variant of the pattern, where an observer is interested a monolithic subject object, does not distinguish between different types of changes and does not require more information about the change.

Previous work on Caesar also identified the need of dynamic aspect deployment to enable variation of aspect functionality, depending on run-time settings [19] or the owner object [20]. Early implementations of Caesar [9] were limited to thread based deployment of aspects. More complicated deployment was implemented by special static aspects, responsible for deployment of other aspects, as demonstrated in [20]. A similar design pattern is also used for activation of call-ins in Object Teams [11]. However, custom deployment strategies is a better solution, because these strategies are more compact and reusable, the deployment scope is defined independently from concrete aspects.

Remote pointcuts introduced in DJcutter [21] allow definition of aspects, which refer to joinpoints on remote processes. All aspects run on a special aspect server and intercept joinpoints in all or some of the registered hosts. The solution is similar to remote aspect deployment in Caesar and provides more rich implementation of crosscutting features, such as cflow cross process boundaries or remote intertype declarations. DJcutter also employs load-time weaving. Load-time weaving enables deployment on hosts, which are unaware of aspects. Hovewer, DJcutter is not suitable for implementation of observer aspects, because aspects are singletons and they are running in one process and cannot be dynamically controlled.

CaesarJ and most of dynamic aspect activation approaches, such as AOP[4], JAC[22], PROSE[23], JBoss AOP[13] and AspectWerkz[1], require one or another form pre-runtime class preparation for weaving. The classes are either prepared at compile time, load time or just-in-time compilation. There are two possibilities for pre-runtime class preparation: either to insert hooks at all join points of a loaded class or

to limit to a fixed set of known join points. While the first option causes significant performance overhead, the second option (also used in Caesar) assumes initial knowledge about aspects, which will be activated. Load time weaving is better than compile time weaving, because it enables independent distribution and deployment of different software packages.

Dynamic aspect deployment can be more efficiently implemented on the systems supporting real run-time weaving, such as Steamloom [3] and AspectS [12]. Only the Steamloom is flexible enough for the needs of aspect deployment in CaesarJ, because it supports thread local aspects as well as aspect deployment on individual objects.

7. SUMMARY

In this paper we explained how aspects can provide clean separation between components and their observers. Aspect-oriented solution also frees us from preplanning problem and supports various ways to observe the same component. The aspect objects, introduced by Caesar, remove the gap between pointcuts and stateful observer objects. Treating aspects as objects also enable flexible control over their lifecycle and their remote usage. Flexible deployment strategies can also solve the observation scope problem. The paper demonstrate that dynamic aspects can be used not only for implementation of global crosscutting concerns, but also can be used to improve object-oriented solutions for functionality on various dynamic scope, ranging from individual objects to hosts in distributed systems.

8. REFERENCES

[1] Aspectwerkz home page. http://aspectwerkz.codehaus.org/.

[2] V. Arslan, P. Nienaltowski, and K. Arnout. Event library: an object-oriented library for event-driven design. In *Proceedings of Joint Modular Languages Conference'03*, 2003.

[3] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92. ACM Press, 2004.

[4] R. Douence and M. Sudholt. A model and a tool for event-based aspect-oriented programming. Technical Report Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.

[5] P. Dubois, M. Howard, B. Meyer, M. Schweitzer, , and E. Stapf. From calls to agents. *Journal of Object-Oriented Programming*, 12(6), June 1999.

[6] E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *Proceedings ECOOP 2001*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[7] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. Dynamic component adaptation. Technical Report Technical Report 200104, EPFL, 2001.

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

[9] J. Hallpap. Towards caesar: Dynamic deployment and aspectual polymorphism. Master's thesis, Department of Computer Science, Darmstadt University of Technology, 2003.

[10] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of the 17th ACM SIGPLAN conference on OOPSLA*, pages 161–173. ACM Press, 2002.

[11] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Proceedings of Net.ObjectDays*, Erfurt, Germany, 2002.

[12] R. Hirschfeld. Aspects - aspect-oriented programming with squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232. Springer-Verlag, 2003.

[13] JBoss Inc. JBoss aop beta3. http://www.jboss.org, 2004.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP '01*, 2001.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–355. Springer, 2001.

[16] O. L. Madsen and B. Mller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89. ACM SIGPLAN Notices 24(10)*, pages 397–406, 1989.

[17] R. C. Martin. Granularity. *C++ Report*, 8(11), 1996. www.objectmentor.com/publications/granularity.pdf.

[18] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA '02, ACM SIGPLAN Notices 37(11)*, pages 52–67, 2002.

[19] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–99. ACM Press, 2003.

[20] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *Foundations of Software Engineering (FSE-12), ACM SIGSOFT*, 2004.

[21] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut: a language construct for distributed aop. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 7–15. ACM Press, 2004.

[22] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Jac: A flexible solution for aspect-oriented programming in java. In *Proceedings of the third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 1–24. Springer, 2001.

[23] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for java. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109. ACM Press, 2003.