ASPECTBOXES – CONTROLLING THE VISIBILITY OF ASPECTS

Alexandre Bergel[1] Robert Hirschfeld[2] Siobhán Clarke[1] Pascal Costanza[3]

[1] *Distributed Systems Group*
Trinity College Dublin, Ireland
Alexandre.Bergel@cs.tcd.ie
Siobhan.Clarke@cs.tcd.ie

[2] *Hasso-Plattner-Institut*
Universität Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

[3] *Programming Technology Lab*
Vrije Universiteit Brussel, Belgium
pascal.costanza@vub.ac.be

Keywords: Aspect-oriented programming, aspect composition, scoping change, aspects, classboxes, squeak

Abstract: Aspect composition is still a hot research topic where there is no consensus on how to express where and when aspects have to be composed into a base system. In this paper we present a modular construct for aspects, called *aspectboxes*, that enables aspects application to be limited to a well defined scope. An aspectbox encapsulates class and aspect definitions. Classes can be imported into an aspectbox defining a base system to which aspects may then be applied. Refinements and instrumentation defined by an aspect are visible *only* within this particular aspectbox leaving other parts of the system unaffected.

In Proceedings of the International Conference on Software and Data Technologies (ICSOFT 2006)

1 INTRODUCTION

Aspect-oriented programming (AOP) promises to improve the modularity of programs by providing a modularity construct called aspect to clearly and concisely capture the implementation of crosscutting behavior. An aspect instruments a base software system by inserting pieces of code called advices at locations designed by a set of pointcuts.

An important focus of current research in AOP is on aspect composition [Douence et al., 2004, Klaeren et al., 2000, Nagy et al., 2005, Brichau et al., 2002]. Ordering and nesting are commonly used when composing aspects and advices [Kiczales et al., 2001, Tanter, 2006]. Whereas most aspect languages provide means to compose aspects at a very fine grained level, experience has shown that ensuring a sound combination of aspects is a challenging and difficult task [Lopez-Herrejon et al., 2006]. First steps are already taken by AspectJ [Kiczales et al., 2001] by restricting pointcuts to a Java package or a class through the use of dedicated pointcuts primitives such as within and withincode primitive pointcuts.

If we regard an aspect as an extension to a base system, multiple extensions are difficult to manage and control, even if they are not interacting with each other. We believe that the reason for this is the lack of a proper scoping mechanism.

In this paper we define a new modular construct for an aspect language called an aspectbox. An *aspectbox* is a modular unit that may contain class and aspect definitions. Classes can be imported into an aspectbox and the aspect is then applied to the imported classes. Refinements originated from such aspects are visible *only* within the aspectbox that defines this aspect. Outside this aspectbox the base system behaves as if there were no aspect. Other parts outside a particular aspectbox remain unaffected.

In Section 2 we provide an example illustrating the issues when composing aspects. In Section 4 we describe the aspectboxes module system and its properties. In Section 5 we present our Squeak-based implementation of aspectboxes. Related work is discussed in Section 6. We conclude by summarizing the presented work in Section 7.

2 MOTIVATION

To motivate the need for limiting the scope of aspects, we use an example based on the design of a small four-wheel electric car, and its implementation based on a mainstream aspect language, AspectJ [Kiczales et al., 2001].

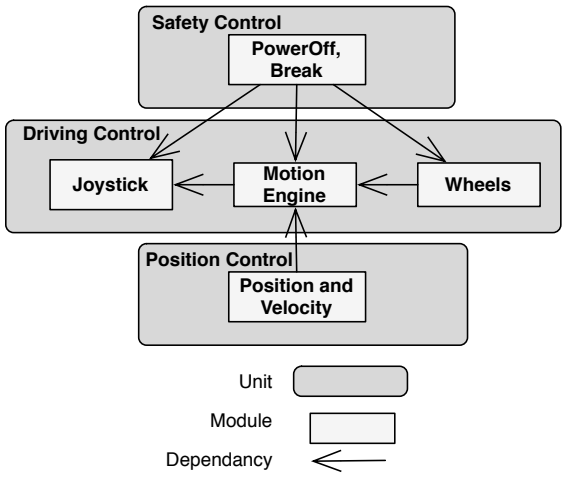

Figure 1: The three units and their modules that compose the CyCab electrical car.

The CyCab [Baille et al., 1999] is an electric four wheel car designed to transport up to two people. The mechanics is taken from a small electrical golf car frame. Functionalities implemented in a CyCab range from an autonomous driving facility (like a coach in a train) to ultrasonic sensors for collision avoidance. A CyCab is composed of three different units (*driving control*, *position control* and *safety control*). Each unit is composed of one or more modules. Figure 1 illustrates the architecture of a CyCab.

Driving Control. A CyCab is steered with a joystick emitting electrical pulses used by the motion engine to activate the four motored wheels. This feature is provided by three modules within the *driving control* unit. The *joystick* module emits signals that are captured by the *motion engine* module. This module controls the *wheels*.

Position Control. The *position control* unit computes the velocity and the location of the CyCab based on the acceleration given by the *motion engine* to the *wheels* and their angle between the car head.

Safety Control. The *safety control* unit verifies the interactions between the three modules of the *driving control* unit. For example, it asserts that pulses emitted by the joystick trigger the correct reaction in the engine and the wheels reflect the heading dictated by the joystick. In addition, in the event of failure the power is shut down and communication between the three modules is cut off.

3 EXAMPLE ANALYSIS

Behavior defined by the *safety control* unit crosscuts the whole *driving control* unit. For example, the impact of a power shut-down is that the *joystick*, the *motion engine* and *wheels* are disconnected. This can be easily captured in an aspect that adds behaviour to check the power status into each affected module, as implemented by the following AspectJ aspect:

```
aspect PowerOff {
  private boolean hasPower = ...;
  pointcut drivingControl():
    target(Joystick) && call(public * *(..)) ||
    target(Engine) && call(public * *(..)) ||
    target(Wheels) && call(public * *(..));
  void around(): drivingControl() {
    if (hasPower == true)
      proceed();
  }
  ...
}
```

The PowerOffAndBreak aspect inserts a check before all public methods of the classes Joystick, Engine and Wheels to proceed only if power is equal to true. This aspect is applied to the *driving control* unit and has to be composed with the PositionAndVelocity aspect defined by the *position control* unit:

```
aspect PositionAndVelocity {
  double speed;
  pointcut speedUp() : call (* Engine.accelerate());
  after(): speedUp() {
    //... Speed calculation
  }
  ...
}
```

PositionAndVelocity inserts a speed calculation functionality after the execution of the accelerate method. Defining the *position and velocity module* as an aspect has the benefit to leave the *driving control* unit free from referring to the speed and position computation. The two aspects PowerOffAndBreak and PositionAndVelocity are woven into the base system, the *driving control* unit, to form a deployable system. With current aspect languages such as AspectJ, *extensions defined by all aspects are automatically applied to all the modules in the system* (*i.e.,* the physical display screen, the electronic control unit in charge of the safety).

This facility is particularly dangerous regarding the implicit sharing of the control flow of the application. A failure raised by the PositionAndVelocity aspect may easily impact the PowerOffAndBreak aspect affecting the electronic control unit in charge of the safety.

Whereas most of current aspect languages offer sophisticated pointcut primitives to express location of join points, they do not provide a means to limit the impact of an aspect into a well-defined system area.

In the up coming section we define a module system for an aspect-oriented programming environment in which one or more aspect compositions are effective only in the context of a well-defined subset of the base system.

4 SCOPING ASPECTS WITH ASPECTBOXES

Most of today's aspect languages do not provide a way to limit the impact of an aspect within a delimited scope. In this section, we describe a module system for an aspect-oriented programming language that allows for controlling the visibility of a set of aspects relative to a well-defined system area.

4.1 Aspectboxes in a Nutshell

Aspectboxes is a namespace mechanism for aspects. An aspect lives in an aspectbox and the effects of this aspect is limited to the aspectbox in which it is defined and to other aspectboxes that rely on the base system extended by this aspect. An aspectbox can (i) define classes, (ii) import classes from another aspectbox and (iii) define aspects.

The import relationship is transitive: If an aspectbox AB2 imports a class C from another aspectbox AB1, then a third aspectbox AB3 can import C from AB2. From the point of view of the importing aspectbox AB3, there is no difference if the class is defined or imported in the provider aspectbox AB2. Because aspects cannot be reused across multiple base systems, aspects cannot be imported.

A pointcut definition contained in an aspect refers only to classes that are imported (*i.e.,* visible within the aspectbox that defines this aspect). An aspect in an aspectbox refines the behavior of the classes that are imported or defined, for instance by adding some code before and after some methods. The classes augmented with the aspect can also be imported from another aspectbox. From the point of view of an importing aspectbox, there is no distinction between classes defined within the aspectbox and those imported.

4.2 Namespace for Classes and Aspects

An aspectbox defines a namespace for class definitions, aspect definitions and aspect compositions.

Aspectbox as namespace for classes. The class Engine contained in the aspectbox DrivingControlAB[1] as illustrated in Figure 1 is defined as the following[2]:

```
(Aspectbox named: #DrivingControlAB)
  createClassNamed: #Engine
  instanceVariableNames: ''
```

The class Engine does not have any instance variables and two methods accelerateWheels: anAcceleration and setAnglewithHeading: anAngle are defined on it.

```
DrivingControlAB.Engine>>
      accelerateWheels: anAcceleration
  "accelerate the wheels with a given acceleration"
  ...

DrivingControlAB.Engine>>
      setAnglewithHeading: anAngle
  "set the heading of the car by setting
  appropriately the wheel angle"
  ...
```

An aspectbox acts as a code packaging mechanism and constrains aspect visibility. A class is *visible* within an aspectbox if this class is defined in or imported to this aspectbox. Any class visible within an aspectbox AB1 can be imported from AB1 by other aspectboxes. The aspectbox PositionControlAB imports the class Engine from DrivingControlAB

```
(Aspectbox named: #PositionControlAB)
  import: #Engine from: #DrivingControlAB
```

An instantiation of a class can occurs in any aspectbox as long as this class is visible in the aspectbox that contains the code performing the instantiation. Class instances (*i.e.,* objects) do not belong to an aspectbox.

Aspectbox as namespace for aspect definitions. The module *position and velocity* is implemented by the PositionAndVelocity aspect:

```
(Aspectbox named: #PositionControlAB)
  createAspectNamed: #PositionAndVelocity
  instanceVariableNames: 'heading velocity'
```

Because the aspect PositionAndVelocity has to be applied to the class Engine, this class has to be imported from the DrivingControlAB aspectbox. This aspect also defines advices to be applied to the methods accelerateWheels: anAcceleration and setAnglewithHeading: anAngle that compute the velocity and the heading, respectively, as illustrated in Figure 2.

Aspectbox as namespace for aspect compositions. An aspect, which is defined in an aspectbox, is applied to classes that are visible in this aspectbox (*i.e.,* classes that are imported or defined). The effect of

[1] We end the name of aspectboxes by AB to clearly make a distinction between them and regular class names

[2] Since our aspectboxes prototype is implemented in Squeak, we therefore use the Squeak syntax to describe them.

```
PositionControlAB. PositionAndVelocity>> adviceComputeVelocity
  ^AfterAdvice
    pointcut: (JoinPointDescriptor
                targetClass: Engine targetSelector: #accelerateWheels:)
    afterBlock: [:receiver :arguments :aspect |
                "computation of the velocity according to the speed of the wheels"
                velocity := ...]

PositionControlAB. PositionAndVelocity>> adviceComputeHeading
  ^AfterAdvice
    pointcut: (JoinPointDescriptor
                targetClass: Engine targetSelector: #setAnglewithHeading:)
    afterBlock: [:receiver :arguments :aspect |
                "computation of the heading according to the speed of the wheels"
                heading := ...]
```

Figure 2: The velocity and the heading are computed by two advices adviceComputeVelocity and adviceComputeHeading, respectively.

this aspect is limited to the aspectbox in which this aspect is defined. Outside this aspectbox, it is as if no aspect would have been applied to the base system.

The aspectbox SafetyControlAB defines the aspect PowerOff. This aspect has one advice, adviceDrivingControl that proceed a method call if the hasPower is true.

```
(Aspectbox named: #SafetyControlAB)
  createAspectNamed: #PowerOff
  instanceVariableNames: 'hasPower'.

SafetyControlAB.PowerOff>>
        adviceDrivingControl
  | joinpoints |
  joinpoints := JointPointDescriptor
          targetClasses: {Joystick . Engine . Wheels}.
  ^AroundAdvice
    pointcut: joinpoints
    aroundBlock: [:receiver :arguments :aspect |
        hasPower ifTrue: [ aspect proceed ]
```

Aspects PowerOff and PositionAndVelocity described above have a common pointcut: public method of the class Engine. Because these two aspects belongs to different aspectboxes (SafetyControlAB and PositionControlAB, respectively), they do not conflict with each other.

4.3 Executing Code in an Aspectbox

Triggering a program execution in an aspectbox is achieved by the method eval:.

```
(Aspectbox named: #SafetyControlAB) eval: [
  | app |
  app := SafetyApplication new.
  app run].
```

The code above instantiates the class SafetyApplication and invokes the method run. The code invoked by this method run will benefit from aspects defined in SafetyControlAB (*i.e.,* PowerOff). Similarly, an application invoked in the aspectbox PositionControlAB will benefit from PositionAndVelocity without being affected by SafetyControlAB.

4.4 Absolute Isolation of Aspects

It is widely accepted that encapsulating different functionalities of a system in distinct modular units aids their comprehensibility and maintainability [Parnas, 1972].

Figure 1 illustrates a modular architecture. Because it is closely linked to the physical and external physical mechanic events, the *driving control* unit needs special care and should not be altered by other units that are not necessary for its execution. Also, for safety reasons, the *position control* unit has to be built on top of the motion engine without affecting its execution. *Different concerns composed into a system have to be well modularized and isolated from the base system.*

The aspectboxes module system has the following properties:

- *Conflicts between aspects are avoided.* By living in different scopes, aspects are kept separated. Even if aspects defined in different aspectboxes have the same join points, there is no need to define precedence rules for composition ordering.
- *Minimal extension of the aspect language.* Combining the aspectboxes module system with AspectS [Hirschfeld, 2003] did not require any modification of the aspect language syntax. Static references contained in the definition of pointcuts are resolved using the classes visible in the aspectbox in which these pointcuts are defined in.

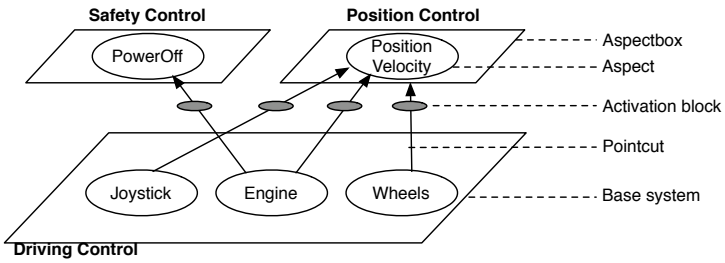

Figure 3: The PowerOff and PositionAndVelocity aspects hooked into the driving control module.

5 IMPLEMENTATION

A prototype of aspectboxes is implemented in Squeak. Figure 3 describes how the *safety control* and the *position control* are hooked into the *driving control* module.

AspectS. AspectS [Hirschfeld, 2003] is an approach to general-purpose aspect-oriented programming in the Squeak[3] Smalltalk environment [Ingalls et al., 1997]. It extends the Squeak metaobject protocol to accommodate the aspect modularity mechanism. In contrast to systems like AspectJ, weaving and unweaving in AspectS happens dynamically at runtime, on-demand, employing metaobject composition. Instead of introducing new language constructs, AspectS utilizes Squeak itself as its pointcut language. AspectS benefits from the expressiveness and uniformity of Squeak.

Activation blocks. AspectS uses Method Wrappers [Brant et al., 1998] to instrument both message sends and receptions. Such wrappers support execution of additional code before, after, around, or instead of an existing method. The core of the aspect activation mechanism is implemented in the isActive method of the class MethodWrapper. All additional code provided by a wrapper is to be activated only if all activation blocks associated with it evaluate to true. Activation blocks are treated as predicate methods, returning either true or false as the outcome of their execution.

Aspectboxes. The aspectboxes module system is fully integrated in the Squeak environment. When an aspect is woven, activation blocks are created and placed at join points shadows. When the control flow of the application reaches a join point, the isActive methods is executed in order to determine if this potential join point is within the scope of an aspectbox defining this aspect to yield activation or not (*i.e.,* if it is associated with the current control flow).

[3]Squeak is an open-source Smalltalk available from www.squeak.org

6 RELATED WORK

AspectJ. The pointcut language offered by AspectJ provides a mechanism to restrict a pointcut definition to a package or a class (*i.e.,* within and withincode pointcut primitives). The purpose of these constructs is to restrict the location of join points between a base system and an aspect, however advices hooked at those join points remain globally visible. Therefore, the restricting pointcut primitives of AspectJ do not help in scoping an aspect application.

CaesarJ. Aspects, packages and classes are unified in CaesarJ [Aracic et al., 2006] under a single notion, a cclass. Aspect deployment can either be global or thread local.

Aspectboxes promotes a syntactic scoping of aspects: an aspect is scoped to the aspectbox that defines it. In CaesarJ, an aspect is scoped to the thread it was installed in.

Classboxes. The Classbox module system allows a class to be extended by means of class member additions and redefinitions. These extensions are visible in a locally and well-delimited scope. Several versions of a same class can coexist at the same time in the same system. Each class version corresponds to a particular view of this class [Bergel et al., 2005].

Classboxes and aspectboxes have a common root which is the scoping mechanism for refinement. Whereas classboxes support structural refinement (*i.e.,* class members addition and redefinition), aspectboxes offer a scoping mechanism for behavioral refinement.

Context-aware aspects. Context awareness promotes software program behaviour to depend on "context". Context-aware aspects [Tanter et al., 2006]

offers language constructs to handle contexts. A context is defined by the programmer as a plain standard object. The pointcut language is extended with primitives such as inContext(c) and createdInContext(c) that restrict a pointcut expression to a particular context c and to objects that were created in a context c, respectively.

Whereas context-aware aspects trigger the activation of aspects based on some arbitrary context activation function, aspectboxes promote the concurrent applications of aspects by restricting them to different scope.

Context-oriented programming. ContextL [Costanza and Hirschfeld, 2005], a CLOS-based implementation for Context-Oriented Programming, provides dedicated programming language constructs to associate partial class and method definitions with layers. Layers activation and deactivation is driven by the control flow of a running program. When a layer is activated, the partial definitions become part of the program until this layer is deactivated.

Whereas scoping software system refinement is the common problem for context-oriented programming and aspectboxes, the approaches are different. A layer in ContextL encapsulate structural definitions, whereas aspectboxes encapsulate behavioral definitions.

AWED. Aspects with Explicit Distribution (AWED) [Navarro et al., 2006] is an approach for defining crosscutting behaviour on remote locations (*i.e.,* distributed applications). AWED is an aspect language supporting remote pointcuts, distributed advices and distributed aspects. A distributed aspect allows for state sharing and aspect instance to be distributed across multiple hosts.

7 CONCLUSION

Aspectboxes provide a new aspect modularity construct limiting the scope of aspect composition with a base software system. Modifications to the base system are visible only in the aspectbox the aspect is defined in. This allows one to deploy multiple concurrent modifications in the same base system, avoiding conflicting situations across aspectboxes.

In the work presented in this paper, an aspect cannot be imported from an aspectbox. The reason for this is that aspects are not generic (*i.e.,* cannot be applied to other base systems). As future work, we plan to refine the notion of import to enable reuse of aspects within multiple aspectboxes.

Our prototypical implementation is based on AspectS. It integrates the composition mechanisms of AspectS and Classboxes to achieve the desired composition and scoping behavior.

Acknowledgments. We gratefully acknowledge the financial support of the Science Foundation Ireland and Lero — the Irish Software Engineering Research Centre. We also like to thank Parinaz Davari and Daniel Rostrup for their valuable comments.

REFERENCES

Aracic, I., Gasiunas, V., Mezini, M., and Ostermann, K. (2006). An overview of caesarj. *Transactions on Aspect-Oriented Software Development*, 3880:135 – 173.

Baille, G., Garnier, P., Mathieu, H., and Pissard-Gibollet, R. (1999). Le cycab de l'inria rhône-alpes. Technical Report RT-0229, INRIA.

Bergel, A., Ducasse, S., and Nierstrasz, O. (2005). Classbox/J: Controlling the scope of change in Java. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, New York, NY, USA. ACM Press.

Brant, J., Foote, B., Johnson, R., and Roberts, D. (1998). Wrappers to the Rescue. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag. method wrappers.

Brichau, J., Mens, K., and Volder, K. D. (2002). Building composable aspect-specific languages with logic metaprogramming. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *LNCS*. Springer-Verlag.

Costanza, P. and Hirschfeld, R. (2005). Language constructs for context-oriented programming. In *Proceedings of the Dynamic Languages Symposium 2005*.

Douence, R., Fradet, P., and Südholt, M. (2004). Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150, New York, NY, USA. ACM Press.

Hirschfeld, R. (2003). AspectS – Aspect-Oriented Programming with Squeak. In Aksit, M., Mezini, M., and Unland, R., editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, number 2591 in LNCS, pages 216–232. Springer.

Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A. (1997). Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of As-

pectJ. In *Proceeding ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag.

Klaeren, H., Pulvermüller, E., Raschid, A., and Speck, A. (2000). Aspect composition applying the design by contract principle. In *Proceedings of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE 2000)*, volume 2177 of *LNCS*, pages 57–69. Springer-Verlag.

Lopez-Herrejon, R., Batory, D., and Lengauer, C. (2006). A disciplined approach to aspect composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA. ACM Press.

Nagy, I., Bergmans, L., and Aksit, M. (2005). Composing aspects at shared join points. In Robert Hirschfeld, Ryszard Kowalczyk, A. P. and Weske, M., editors, *Proceedings of International Conference NetObject-Days, NODe2005*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany. Gesellschaft für Informatik (GI).

Navarro, L. D. B., Südholt, M., Vanderperren, W., Fraine, B. D., and Suvée, D. (2006). Explicitly distributed AOP using AWED. In *Proceedings of the 5th Int. ACM Conf. on Aspect-Oriented Software Development (AOSD'06)*. ACM Press.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):1053–1058.

Tanter, É. (2006). Aspects of composition in the reflex aop kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, LNCS, pages 99–114, Vienna, Austria.

Tanter, É., Gybels, K., Denker, M., and Bergel, A. (2006). Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, LNCS, pages 229–244, Vienna, Austria.