

Gradual Encapsulation

Stephan Herrmann, Technische Universität Berlin, Germany

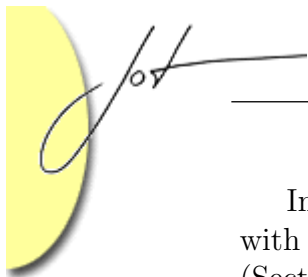
Strictly enforced encapsulation is a key concept for modular software designs. However, in many situations like unanticipated reuse, productivity can be raised by a more *flexible* approach to encapsulation. Also the discussion about the role of “obliviousness” in aspect-oriented programming is one instance of a general conflict between strictness and flexibility. Here, different technologies take different stands and the choice of a particular technology locks a project into prioritizing one side over the other.

In this paper we suggest that this choice should not be determined by technology but technology should support the co-existence of encapsulation and its inverse — decapsulation — within a single system. We postulate four principles that define a solution space, called “gradual encapsulation”, in which each project should find the best fitting balance between encapsulation and decapsulation with the option to shift this balance during the life time of a system. We use the programming language ObjectTeams/Java for illustrating how encapsulation and decapsulation can be supported by technology and how the four principles can be implemented on top of such technology.

1 INTRODUCTION

This paper is about the inherent conflict between encapsulation and “obliviousness” [7]. It bundles three strands of discussion, that were started independently: At SPLAT’06, Ossher [16] proposed “confirmed join points” as a basis for negotiating possible join points between a “class owner” and an “aspect owner”. At SPLAT’07, Leavens and Clifton [13] suggested that language engineers should consider more than one concern at a time and that they should use gradual scales for assessing a language with respect to a given concern. At ECOOP’07, Siek [17] demonstrated how typed and untyped parts of a program can safely exist side by side and how type analysis can cope with this combination of paradigms.

At SPLAT’08 [12] we started to connect these strands by arguing that the suggestion by Leavens and Clifton should be taken one step further, as to defer compromises from the point of language design down to the point of language usage. Borrowing from Siek’s notion of “gradual typing”, we initially proposed “gradual encapsulation”, where indeed the conflict between encapsulation and obliviousness remains unsettled in the language. We mentioned, that the approach of confirmed join points enables a project to negotiate the conflict of encapsulation vs. obliviousness in a case based manner. From these ingredients we ventured to say that language designers should in general be more modest in a way of fixing fewer decisions in the language and passing more options down to the users of the language.



In this paper we expand the ideas from [12], focusing on the role of encapsulation with a specific view on how it inherently conflicts with aspect-oriented programming (Sect. 2). We elaborate the concept of gradual encapsulation, which connects language design issues (Sect. 3) to necessary organizational considerations (Sect. 4). This approach reconciles previously incompatible requirements, supporting new, manageable ways even for the difficult scenario of aspect-base co-evolution. We will present, how the concept can be implemented in a practical setting, using the language ObjectTeams/Java for illustration, and drawing from experience of using ObjectTeams/Java for adapting a large, complex, real-world system (Sect. 5). We will conclude with a discussion about costs and benefits plus a summary (Sect. 6).

2 ON THE ROLE OF ENCAPSULATION

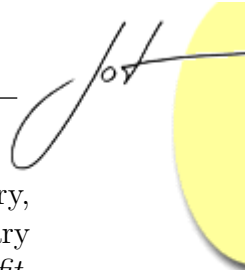
Among all principles in software engineering, *encapsulation* stands out to the degree that hardly any software engineer seems to question the primacy of encapsulation as *the* pre-requisite to a good modular system structure, which in turn is a pre-requisite to many software engineering properties like comprehensibility and maintainability. Therefore, programming languages are commonly considered “good for software engineering” when they help to strictly enforce encapsulation, and “bad for software engineering” if they provide means for bypassing any module boundaries. In this mind set, the world of software is like a huge piece of Lego artwork, where *anything* can be composed from neatly delineated building blocks, plugged together at simple, predefined plug-points.

Complexity rules out Lego-like encapsulation

Today’s software business follows rules that are different from the Lego-metaphor: systems are more complex than what can be expressed with Lego pieces. By their very nature, Lego pieces are not good for capturing *crosscutting concerns* and all discussions about “obliviousness” as a possible foundation of aspect-oriented programming hint at experience from practical development, where ignoring module boundaries may have a value — be it a double-edged value.

Another source of complexity is actually imposed by the *organizational context* of software development. Software is not always developed as a one-shot production of a monolithic system, but often bits and pieces are gathered from existing libraries, frameworks and previous applications, whose origins are arbitrarily remote and scattered, i.e., this is code written by someone else.

In the early days of software reuse, developers were perhaps happy when they found any library that came close to what they needed, and would adjust the design of their application to the given facts of that library. Today’s clients (and developers) seem to be more demanding: they would insist on exactly what features they had in mind for their application. Competition does not allow to adjust the product



under development to the libraries that are found for building it. To the contrary, reusing a library should only help, not constrain the development. If the library (or other piece of reusable software) does not perfectly fit, it *should be made fit*. In software development with such *opportunistic reuse*, adaptation of given parts is daily business. Such adaptation rarely fits into the neatly declared interfaces of a Lego piece. Rather, these adaptations may require adjustments at the inside of other people's components.

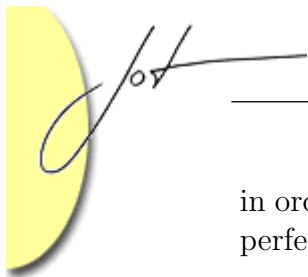
Purists who object to this role of opportunistic adaptation, should carefully weigh the alternatives. Wallnau et al [18] draw a splendid picture of what happens, when systems are to be built from black box COTS components with full obedience to encapsulation: They observed *discontinuities* in the design space of systems built this way, which means: arbitrarily small changes in the system's requirements can *not* always be dealt with by small changes in the design, but by a certain small change, an entire design would just break down and had to be replaced with a completely new design, simply because a third-party component would not allow a required alteration of its behavior. Such a switch from one design to another is a worst case scenario, incurring uncontrollable costs for only small improvements, thus easily causing complete project failure. From this we learn that even if software reuse worked well up-to a certain point in time, by simply plugging together existing black box components, as the system evolves we may come to a point where we are faced with only two alternatives: start over (with an alternative design, perhaps using a completely different set of reusable assets) or perform the adaptations needed to reconcile the existing design with the new requirement.

Technical options regarding encapsulation

If we put aside legal considerations of changing third-party software, we find that both "worlds" are actually supported by some existing technologies:

- Most class-based, statically typed, object-oriented programming languages support *encapsulation* and their compilers can be used to check and *enforce* that module boundaries are respected. Component frameworks add further levels of blackness of their components.
- Other programming languages may not have any notion of visibility by which class features are divided into public and private parts, *or* languages may support a meta-object protocol by which all encapsulation can be bypassed, *or* a language may support AOP, etc.

Leavens and Clifton [13] criticize that language design is often discussed in black and white, in a boolean judgment about perhaps only one language property that for a given discussion is considered paramount. Their contribution is twofold: (1) They encourage language designers to evaluate how well a language supports a given concern on a *gradual scale* rather than a boolean one. (2) They encourage language designers to consider multiple concerns — even conflicting ones — simultaneously,



in order to develop compromises that have much higher values than a language that perfectly supports one concern but completely fails on an other.

Applied to the conflict between encapsulation and adaptation, this could be read as a plea for a language that supports – say – “90% encapsulation” and “10% adaptation” (using a fictitious scale à la [13]). At a first look such a language could indeed be a suitable platform for projects where encapsulation is the default and some adaptations are also possible. The problem is: at the start of a project it is impossible to know which level and kind of adaptation may become necessary later-on. Yet, the choice of the programming language must be settled at the start.

Now, what happens if a project started in a perfect Lego world with a “100%-encapsulation” language? It will never benefit from existing technology for adaptation, because it is locked into a language that prescribes the absence of violations of the pre-defined module structure.

Opposite, if a project started with a focus on flexibility in order to glue together a set of noncompliant components, what if the design should be consolidated later, in order to more and more respect module boundaries, perhaps before shipping a first public release? If they chose a technology that only favors flexibility, this project will have very difficult times in enforcing policies which no compiler actually checks.

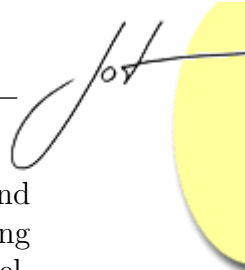
Finally, when calling for (perhaps densely) populating the language space between “100% encapsulation” and “0% encapsulation” we would end up with an extreme fragmentation between users of far too many languages, with no chance of reuse due to ubiquitous language barriers.

Debugging the debate

This dilemma may sound ridiculous, so what’s wrong in this discussion? Encapsulation is good for many software engineering properties. Reuse is an economical must. The need for adaptation directly follows from conflicts between reuse and deviating requirements. What can language design do to solve the dilemma?

After mentioning that some complexity in this situation is not caused by technology but by the organizational context of software development, it is only a small step to understanding that technology (in the form of language design) is not the right level for solving the dilemma. The mentioned phenomenon of *design space discontinuities* does not arise from poor language design. Even if we would put together all known mechanisms for defining interfaces and encapsulation at any level of precision (think of friends in C++, class-by-class export in Eiffel, etc.), we simply cannot always refactor an existing design to accommodate new requirements because we may not be *allowed* to refactor those pieces we acquired from third parties, nor can we change the past to produce a “perfect design” in the first place.

In real world projects many interface conflicts are conflicts between different stakeholders working at different points in time, producing different pieces of software, and having different goals. These are the real barriers we have to consider, of



which a given degree of encapsulation may only be a technical manifestation. And yes, we do have to consider legal issues when adapting third-party software. Along these lines, one might even claim that AOP doesn't yet have a sound business model, if liabilities of an aspect programmer are not legally defined.

It is our belief that the task of the language designer is not in *deciding* about the degree of encapsulation supported by the language, but in passing such options down to the developers using the language. These developers, embedded in their respective organizational contexts, are the only ones who can effectively balance the specific forces in a given project.

This paper defines *gradual encapsulation* as an approach where technology supports both encapsulation and its absence (or violations of module boundaries). Gradual encapsulation shall also mean, that a project can explicitly choose and establish its specific compromise between the extremes. Gradual encapsulation shall finally mean a method, guided by four principles, by which the newly gained powers can be mastered.

3 BALANCING ENCAPSULATION AND FLEXIBILITY

In order to avoid locking a project into either the safety provided by encapsulation *or* flexibility of obliviousness with a lack of support for the respective other goal, we develop the concept of gradual encapsulation. We use as our starting point the situation of Java-like languages, i.e., languages with static typing and strict enforcement of a predefined encapsulation policy. In order to re-establish some degree of flexibility we add explicit support for intentionally breaking encapsulation, which we call *decapsulation*.

Decapsulation

In order to be very explicit, we coined the term *decapsulation* to refer to any mechanism that intentionally breaks encapsulation. Since encapsulation will always remain a major goal, we want “decapsulation” to be perceived as a potentially dangerous mechanism. This should motivate developers to keep the extent of decapsulation at a minimum.¹

Decapsulation means any kind of communication in a software system which crosses an encapsulation boundary along paths that have not been specified (cf. Aldrich's principle of “communication integrity” [1]). Fundamentally this communication can happen in two directions: from a client *to* hidden details of the encapsulated module or *from* within the encapsulated module to the client.

¹ Unlike [13] we still think that “discouraging” certain uses of a language could be a useful means for guiding developers towards good designs, such that overriding a particular compiler warning, e.g., should not be taken too lightly, but require a conscious decision.

- In **API decapsulation** a client uses regular object-oriented concepts for accessing elements (classes, methods, fields) in a base module with the only difference that access restrictions as defined by the providing module are intentionally disregarded. This kind of decapsulation enables a client to *modify the interface* of a third-party component by using a wider interface than it was offered by the component.
- In **join point decapsulation** the client component does not call methods of the supplier component, but rather instructs the supplier to call methods of the client at specific join points. This kind of decapsulation enables a client to *modify the behavior* of a third-party component. Although carefully designed aspects (“harmless advice” [4]) can indeed leave the base program intact, the mere potential of an aspect to indeed modify the base code’s behavior should be taken seriously. Thus we propose to consider any aspect as breaching encapsulation *unless* it is proven to be harmless.

By this distinction we suggest an understanding of encapsulation — and thus decapsulation — that is not so much focused on visibility (“is module A allowed to *see* member *m* of module B?”) but rather on selectively permitting *communication* between elements of different modules. Thus, each instance of decapsulation adds one — previously undeclared — channel of communication to the architecture (cf. [1]).

At the technical levels both kinds of decapsulation may require to change the code of the base component. In Java-based languages this can be performed by byte code transformations that can be executed at either compile-time or load-time or run-time. More specifically, in AOP-languages these transformations are the job of the aspect weaver.

Confirmed Join Points

Standard aspect-oriented programming languages tend to come with support for both encapsulation *and* join point decapsulation. Thus it shouldn’t surprise that this conflict has started a heated debate, usually connected to the concept of “obliviousness”. While most contributions to this discussion try to solve the issue on a purely technical level, they miss an essential point: they don’t sufficiently question the motivation for wanting either encapsulation or decapsulation. Thus, the desire for either strictness or flexibility is usually taken for granted, almost in the style of a dogma.

It is the contribution of Ossher [16] to point out “*that the balance of control and flexibility is largely a social issue among the developers responsible for different parts of the software, such as class and aspects owners*”. Only when considering the different stakeholders involved, it is possible to find suitable compromises. Ossher’s concept of *confirmed join points* is based on the two roles of *class owner* and *aspect owner*, where the class owner is by nature conservative, striving to protect his code, whereas the aspect owner is progressive in terms of desiring to adapt existing base



code by applying aspects to it. Ossher suggests the confirmation of join points as an additional, annotation-like part of the base software. Join point confirmation asserts that class and aspect owners have come to an agreement about the suitability of specific join points.

From the approach of confirmed join points we take inspiration for defining gradual encapsulation. Gradual encapsulation takes on the perspective of the stakeholders involved, considering also a third role: the application user. Furthermore, gradual encapsulation broadens the scope to also include API decapsulation and defines four principles that form a conceptual framework for how technology and organizational processes could be harmonized.

4 PRINCIPLES FOR GRADUAL ENCAPSULATION

We agree with Leavens and Clifton[13] regarding the necessity to consider various concerns in language design as candidates for compromises. Yet, we uphold that encapsulation plays a unique role, because encapsulation is able to directly *reflect* the barriers in the organization domain (i.e., barriers/conflicts between stakeholders) *into* the software proper. By this role new means for elaborating project specific compromises are even more important than compromises regarding other concerns.

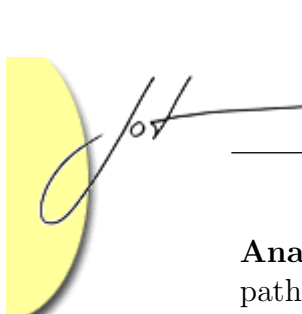
In order to establish decapsulation as a manageable option that adds some needed flexibility to the development process, we propose the following four principles to which any mechanism for decapsulation should adhere:

Analyzability: It should be very easy to find out which parts of a system are affected by decapsulation and which parts are not. Such analysis should be possible for the architecture-levels of components and classes and also for individual elements of control flows.

Negotiation: Decapsulation must be subject to negotiation between all stakeholders involved. Each stakeholder must be able to specify his or her assumptions and should state the guarantees he is willing to give if those assumptions are met. Based on the statements of other stakeholders each stakeholder must be able to judge the risks he is willing to accept.

Enforcement: Whichever level of encapsulation has been chosen, perhaps as the result of some negotiation, it must be possible to enforce adherence to the architecture. While dynamic checks at runtime are ultimately needed to create trust, as much as possible should be statically checked during development.

Migration: For each system using decapsulation a gradual path must exist along which the system can be migrated to an equivalent system not using decapsulation. The same should be true for the opposite direction. By requesting the path to be gradual the migration can be performed systematically while avoiding unnecessary risks during migration. Naturally, the consensus of all three stakeholders may be required to actually perform such migration.



Analyzability The principle of analyzability mandates to mark any additional paths of communication within a system that are not explicitly allowed by the interfaces of the components involved. It should be noted that such paths of communication consist of connections between individual *pairs* of components: For API decapsulation a specific client component may interact with non-public elements of a specific supplier component. At the same time other clients should still respect the public API of the supplier component. When considering join point decapsulation one specific aspect component receives the control from its specific base component, with no other components being affected. Such bilateral access permissions are difficult to achieve in Java-extensions, since in Java visibility is not defined between individual pairs of classes but exports are defined in a much more coarse grained style.

When we define analyzability based on the communication paths that are possible in a system this is still open to refinement in two directions: (1) If static analysis of an aspect can show that the aspect will never influence the base behavior (an “Observer” in the terminology of [3]), this aspect may not need to be considered in the subsequent steps “negotiation” and “enforcement” (except for issues of software evolution). (2) Our interface level analysis could be extended with a detailed feature interaction analysis in order to obtain a much better understanding of the *behavioral* impact of an aspect. Both kinds of supplementary analysis require at least the level of analyzability defined above and are certainly beneficial if available.²

One purpose of the principle of analyzability is to clearly delineate areas of flexibility from areas of safe encapsulation. Thus, a purpose of any decapsulation annotation must be to allow for the reverse conclusion: all parts of the system that are not marked as applying decapsulation are guaranteed to follow the discipline of encapsulation.

Given that a component may define its public interface and given that other parts of the system may declare to use decapsulation to access hidden details, will this not create an infinite regression of measures and counter-measures? Shouldn’t the language also support annotations to prohibit any decapsulation of specific parts? Will those prohibited areas not create the need to define some super-decapsulation that may even override those prohibition annotations, etc. ad infinitum?

These questions cannot be solved at the technical level because no fixpoint could be identified if the powers of encapsulation and decapsulation were exactly symmetric. The good news is that a plain technical solution for this issue is not needed, moreover, it is not the purpose of technology to settle this issue.

Negotiation Zooming out of the system and considering also the stakeholders involved, we identify three parties: a base-owner, an adaptation developer and the

² Note that this statement also applies to ordinary API usage: the understanding of *any* module connection benefits from the mentioned analyses.



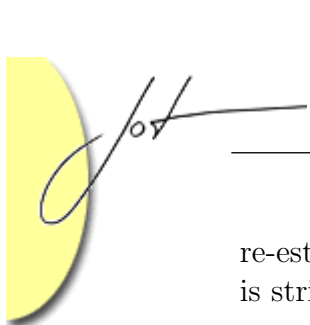
user of the composed system³. The base owner provides some reusable component and wants to protect this component from unsuitable uses. The adaptation developer wants to use the base component in a system where the base component needs to be adapted to specific requirements. The user finally decides which system he or she wants to install and run.

Basically, the base owner has a conservative position, because he would be unable to guarantee correct functioning of the base component if arbitrary adaptations were allowed. On the other hand the base owner might be interested in broadening the set of potential uses for his base component, because this may lead to higher sales figures. The adaptation developer strives for re-using the base component even if it does not perfectly match all of his requirements. All mentioned motivations for decapsulation apply to the adaptation developer because he wants to combine the power of evolving the system to any desirable direction with the reduced efforts of buying an existing base component instead of building it from scratch. Yet, the adaptation developer should not fully ignore encapsulation: It is him or her who must guarantee proper functioning of the whole system, even having limited means to investigate the base component. Thus the adaptation developer will want to re-use the guarantees given by the base owner, which, however, are only valid if the base component's encapsulation is not breached.

In the end it is the user who decides whom to trust. The user may for himself balance risks against actual costs. As a foundation for this decision the user needs to exactly know about the state of negotiation between the base owner and the adaptation developer. In a perfect world, the user will find out that any interactions programmed by the adaptation developer are actually acknowledged by the base owner. By this acknowledgment the base owner will testify that his guarantees are still valid in this particular adaptation scenario. If the consent of the base owner cannot be shown, running the application will imply a higher risk, because parts of the program will not be covered by the guarantees of either developer, unless the adaptation developer promises on his own, that the adapted base component will not misbehave. If a risk remains that is not tolerable, the user may have to pay the adaptation developer more money in order to create a substitute for the base component, whose public API will perfectly serve the application without any need for decapsulation.

Enforcement The issue of enforcement relates to analyzability as discussed above but reverses the obligations. A system with weak encapsulation can still satisfy the principle of analyzability if all decapsulation happening within the system can be discovered by looking at the code or by using additional tools. Any stakeholder may investigate the state of the system regarding encapsulation but the initiative must come from a stakeholder. Conversely, the principle of enforcement disallows any decapsulation *unless* explicitly allowed as the result of negotiation. Thus, enforcement

³ Conventional uses of the base component which adhere to all rules of encapsulation or not considered here, as they cause no conflict.



re-establishes encapsulation as the default and any attempt to apply decapsulation is strictly checked against the approved policy, not tolerating any violations.

Analyzability and enforcement emphasize the two perspectives of development and execution: Development requires some freedom to create the best suitable architecture. In a component-based system development includes system assembly and thus may also involve an end-user composing a system from individual parts. In all development activities analyzability is an important guide to achieve a good architecture. When a system is executed the architecture is fixed, flexibility is no longer needed, but now safety comes into focus. It is a prerequisite to any safety analysis that no interaction between components can ever happen that has not been explicitly specified.

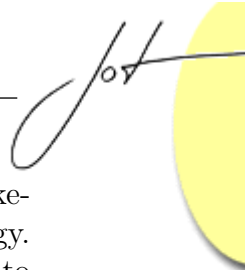
Based on enforcement, design by contract can be applied also in the context of gradual encapsulation. With enforcement any failure at run-time can be ascribed to the exact component that did not fulfill its contract. This is the basis for contractual issues of licensing, since component vendors can be guarded against unchecked decapsulation.

Migration Since one of the motivations for introducing decapsulation was the observation of discontinuous design spaces in the presence of third-party components, we must show how decapsulation helps to re-establish continuity of the design space. This issue introduces two questions:

1. Can encapsulated components be migrated to a state of enhanced flexibility?
2. Can a system using decapsulation be consolidated in order to re-establish strict encapsulation?

Question (1) is answered by the very concept of decapsulation. Any solution to (2) will rely on the principle of analyzability. Only if the locations of decapsulation are known precisely, it is feasible to address and restructure those locations in order to avoid decapsulation. If decapsulation was motivated by rapid development, each single occurrence of decapsulation must be investigated in order to check whether a simple extension to an existing interface suffices to avoid decapsulation.

If decapsulation was motivated by the need to adapt a component for which one does not own the code, consolidating the application will require additional negotiations among base owner and adaptation developer. In this scenario negotiation could mean to change the public API of the base component so that decapsulation is no longer needed. The new API will then be available to all clients. On the other hand, several reasons exist why an architecture using an aspect (and thus decapsulation) might be superior over an architecture with an extended base API. In that case, negotiation should simply result in the permanent confirmation of a set of join points with the guarantee that these join points will also be available in future versions.



Either way the main difficulty lies in achieving a consensus between both stakeholders, and difficulties shouldn't be aggravated by artificial barriers of technology. To the contrary, the principle of analyzability provides all necessary information to aid these negotiations.

The four principles presented above define what we call gradual encapsulation: Encapsulation that can be applied as a strict discipline or can be ignored and where different shades between *all* and *nothing* are well supported, too. Selection of the policy to be applied for a given project should not be dictated by technology but rather be subject to decisions of the project management.

5 IMPLEMENTING GRADUAL ENCAPSULATION

Adopting the concept of gradual encapsulation has both a technical and a social part. In this section we focus on how a *programming language* (and perhaps further tooling/infrastructure) can and should support gradual encapsulation. Given that both extremes — strict encapsulation and full, unrestricted flexibility — are already supported by individual technologies, the goal is not in supporting architectures that until now were impossible. The goal is in providing a well integrated set of conceptual tools that empower each software project to choose its specific balance of strictness and flexibility in a way that this balance can be adjusted still while the project is running.

Clearly, encapsulation and decapsulation are mechanisms to be provided by the programming language. Regarding encapsulation we see an acceptable starting point in the visibility restrictions as they are implemented in Java-like programming languages. Even stronger support for encapsulation can be achieved when also borrowing from component technology or from some kind of alias control. Decapsulation, on the other hand, is inherently supported by most aspect-oriented programming languages. More specifically, join point decapsulation seems to be a necessity for AOP, whereas API decapsulation is a convenient extension but has rarely been discussed in the aspect-oriented community.

The contribution in this section is in demonstrating how those existing extremes can be combined in a single, coherent programming language without a premature bias to either side. After discussing support for encapsulation and decapsulation we will present a Join Point Access Controller as a checker for join point confirmations. Finally we will iterate through the four principles discussing how far technology can go in supporting these. Throughout this section we will use the programming language ObjectTeams/Java for illustration (OT/J, [9]), which already incorporates the mechanisms proposed in this section. We intentionally refrain from using concrete OT/J source code examples, because only the fact that certain bindings are explicit in the source code is relevant, not the exact syntax. The interested reader can find various entry points regarding OT/J on our web page [15].

Encapsulation

When looking at aspect-oriented programming languages based on an object-oriented host language like Java, the degree of encapsulation mandated by gradual encapsulation is already provided by that host language. Naturally, also aspects should be protected by encapsulation, which is generally achieved by making aspect a class-like construct. All Java-based aspect languages basically inherited the standard, class-based styles of encapsulation from their host language Java.

In order to understand how OT/J strengthens encapsulation, it suffices to consider the two kinds of classes introduced by OT/J: teams and roles, plus the underlying type-theory of family polymorphism[6]. An aspect in OT/J is written as a team class containing a set of role classes. Both team and role classes are much more normal classes than an aspect in AspectJ. This lets them participate in arbitrary interactions and relations including instantiation, inheritance and nesting. In AspectJ various constraints like, e.g., “a nested aspect must be `static`” prevent aspects from living up the full power of classes, a regression which is carefully avoided in OT/J.

By applying family polymorphism[6] a team *instance* actually protects its contained role *instances*, which is stronger than Java’s static visibilities. In [11] we have presented how this protection can be taken even further towards strong confinement and representation encapsulation. Since teams can be nested, this strong form of encapsulation scales well even for large designs.

In [10] we have presented OT/Equinox, the integration of OT/J with the component framework Equinox (Eclipse’s implementation of the OSGi standard). Using OT/Equinox, developers can exploit the encapsulation support from OT/J and Equinox together. In [10] we have discussed in which way support for encapsulation in OT/Equinox is stronger than AspectJ-based extensions of given component models. The key point here is an explicit aspect binding declaration between a team class and a base bundle. We will discuss below, how these declarations contribute to the analyzability of a system.

Decapsulation

Join point decapsulation is inherent to most aspect languages. However, join point decapsulation in AspectJ is fully unconstrained, without even an option to re-establish encapsulation thus failing our goal to avoid a premature bias towards strictness nor flexibility. This is where approaches like open modules [2] and XPI [8] try to help, by making join points parts of the explicit module interface. We will discuss these approaches as we go through the four principles.

In order to understand the mechanisms for decapsulation in OT/J, it suffices to consider three kinds of bindings called *playedBy*, *callin* and *callout*. A `playedBy` declaration links a role class to its player (the “base class”). This declaration entitles



the role to establish bindings at the level of methods and fields. In OT/J, join point decapsulation is given by **callin** method bindings. Generally, also OT/J puts no restrictions on join point decapsulation. Only in one specific situation — a **replace** callin binding to an invisible base method — will the compiler issue a warning, but only, because within the bound role method a *base call* (corresponds to **proceed** in AspectJ) would cause API decapsulation⁴, a matter which goes undetected in AspectJ. Instead of an in-language solution for restricting join point decapsulation we propose to adopt the concept of *confirmed join points* [16] as an essential foundation for supporting the four principles (see the next sub-section).

AspectJ features the keyword **privileged**, by which an aspect receives permission for completely unrestricted **API decapsulation**. For a privileged aspect all gates are widely opened for accessing arbitrary invisible elements within the entire system and the language provides no means for declaring a more focused channel of communication from the aspect to selected elements of the base system. Again, with this all-or-nothing policy the **privileged** keyword contradicts the very concept of gradual encapsulation.

OT/J supports API decapsulation at two levels: (1) a **playedBy** declaration may mention a base class which according to the normal rules would be invisible (“base class decapsulation”); (2) a **callout** method binding may refer to an invisible base method (or field) thus making it accessible as a member of the role using implicit forwarding.⁵ By default the compiler reports a warning for any API decapsulation. In strong contrast to AspectJ’s **privileged** keyword, API decapsulation in OT/J is bounded in two ways: (1) A role class has privileged access only to its associated instance of the declared base class and only to those methods bound by callout. (2) Any element made accessible using API decapsulation can only be accessed along the specific channel of playedBy and callout, i.e., any direct access within the body of any method must still adhere to the normal rules of visibility.

Summarizing the different take at API decapsulation, AspectJ has a single switch by which any statement within a privileged aspect can access any element of the base system, whereas OT/J upholds the principle that only declared channels can be used for communication. PlayedBy and callout only add new means for actually declaring such channels.

For the full flexibility of decapsulation, it is important that also elements of an aspect can be accessed in much the same way as regular classes, objects, methods etc. In OT/J this is given by treating teams and roles as mostly normal classes/objects and by avoiding the unfortunate invention of “advice”, which is not amenable to overriding and only supports very restricted forms of interception via aspects-of-aspects. As a result, an aspect in OT/J can equally be the subject or the object in decapsulation.

⁴ This can be interpreted as: Every **replace** callin binding implicitly includes an opposite *callout* binding, binding the base call to the appropriate base method (see “*callout*” below).

⁵ Similar considerations apply to the implicit callout bindings for base calls mentioned above.

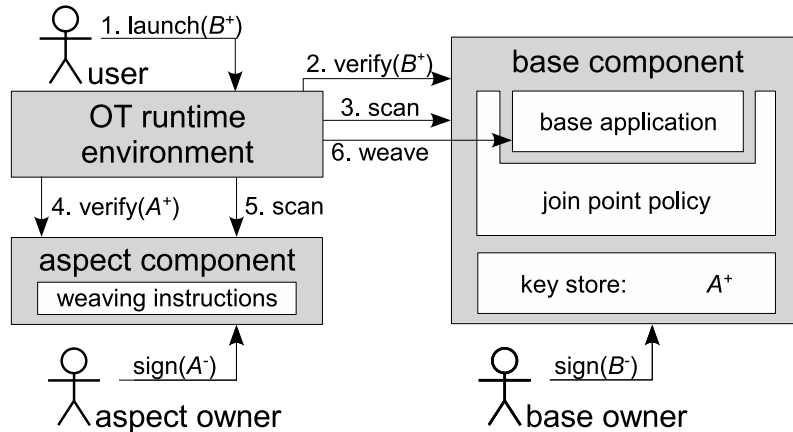


Figure 1: Security architecture for the join point access controller

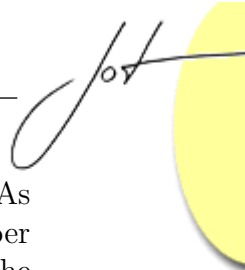
Checking Join Point Confirmations

Given that OT/J supports both encapsulation and decapsulation, we needed a third, unbiased element that controls where and when encapsulation is enforced, or decapsulation admitted. For this purpose we adopted and extended the concept of confirmed join points. Widiker [19] developed a *Join Point Access Controller* (JPAC) for OT/J that monitors whether all occurrences of decapsulation have been confirmed by the base owner. In contrast to the annotation-based style of Ossher's original proposal [16], the JPAC uses a separate join point policy file per base component, containing confirmations (and denials) for join points of the associated base component. This way, the join point policy can be changed without touching the code of the base component; the component only has to be re-packaged with a new policy file, thus lowering the burden on configuration management.

In contrast to [16] and despite its name given in [19], the join point policy not only addresses join point decapsulation but also API decapsulation.

The JPAC is implemented using the security architecture depicted in Fig. 1. The right hand side of this figure symbolizes that the actual base application is shielded by the join point policy that controls all requested decapsulation. For securing the base component against tampering with the join point policy the component is signed using the base owner's private key (B^-). Also the aspect component is signed by its owner's private key (A^-), in order to assert this owner's identity. The Object Teams Runtime Environment (OTRE), which hosts the JPAC, now checks consistency of the composed system using the following steps:

1. The JPAC receives the public key of the base owner (B^+) as an argument passed by the user, whereby the user specifies who he assumes has provided the base application.
2. The base component is verified using the given key (B^+) against the component's signature.



3. Upon success, the base component is scanned for its join point policy. As a result of this scan, the JPAC receives lists of join point confirmations per aspect component. Each list is annotated with the public key (A^+) of the accepted aspect owner.
4. The JPAC now verifies the aspect component using this key (A^+).
5. If the aspect component could be verified successfully it is then scanned for weaving instructions regarding the base component.⁶
6. The OTRE finally performs the actual weaving, but only as permitted by the specified join point policy, which has been scanned in step 3.

The security architecture ensures the following guarantees:

- The base component and its contained join point policy are original versions as signed by the base owner.
- If the join point policy mentions any specific aspect owner, the aspect component must be an original version as signed by the aspect owner, which must be provably the same owner as mentioned in the join point policy.
- Any decapsulation on behalf of a given aspect component must be confirmed by the base owner for the corresponding aspect owner.

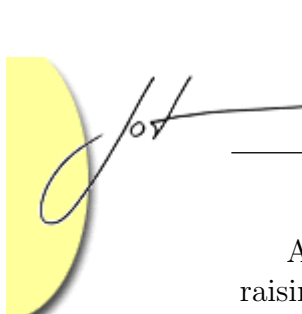
These guarantees are the pre-requisite for any blame assignment in the presence of decapsulation. In particular, if a composed system fails any of the security checks, all liabilities of the base and aspect owner may be considered as void. More details about the JPAC will be discussed as we go.

Principles

We will now discuss how OT/J's technologies for encapsulation and decapsulation plus the join point access controller fit under the principles of gradual encapsulation.

Analyzability When analyzing decapsulation in a given OT/J program, the central elements to look for are all the `playedBy` declarations in the system. Only a `playedBy` declaration may apply base class decapsulation and only under the umbrella of an existing `playedBy` declaration, callin and callout method bindings can cause join point decapsulation and API decapsulation, respectively. In other words, `playedBy` declarations plus callin and callout method bindings are all that needs to be analyzed in order to find out about the state of affairs regarding decapsulation in a given software.

⁶ This assumes that any decapsulation corresponds with a weaving instruction. While this is obvious for join point decapsulation, also API decapsulation requires some weaving/transformation of the byte code, because otherwise the JVM's byte code verifier would not tolerate decapsulation.



Also the design of the OT/Equinox technology [10] puts special emphasis on raising visibility of decapsulation. In OT/Equinox the top-level declaration to consider is a so-called `aspectBinding` which is declared between a team and a base bundle. Only within the scope defined by an `aspectBinding` can roles (contained in the given team) be played by base classes (contained in the given base bundle). Within the team, all designated base classes must be imported with the modifier `base`, to ensure that these classes are only mentioned in right-hand side positions of `playedBy`, `callin` and `callout`. This avoids that a decapsulated name escapes the control of these three kinds of bindings. Because the `aspectBinding` declaration is part of the top-level application architecture, it is easy to zoom in and out the architecture and at each level make precise statements about the extent of decapsulation.

Negotiation In OT/J negotiation regarding decapsulation follows the concept of confirmed join points. The central artifact here is the join point policy mentioned above. In this approach, an aspect developer creates a join point policy file specifying the locations in base components where aspects (wish to) apply decapsulation. This *request policy* is sent to the base owner. When the base owner confirms the requested join points, he or she creates a new package (jar) of the base component, which now contains the confirmed policy. This package is signed by the base owner. If join points are confirmed for a single aspect owner, the join point policy refers to that aspect owner by his public key to ensure that only the approved aspects can utilize the confirmed join points, thus ruling out any attempts to piggyback on a foreign join point confirmation.

This communication could possibly go back and forth several times until a join point policy has been found to which both sides can agree. The relevant steps of creating and confirming a join point policy are aided by a small interactive tool (in [19] called the “packaging tool”, as it also helps to create the final signed packaging).

One might want to compare our join point policy to the `pointcut` interfaces in the open modules approach [2] or XPI [8]. Firstly, both approaches completely lack any support for controlling API decapsulation. Secondly, they are both still confined to the perspective of the core software development, rather than admitting the importance of organizational issues. This perspective manifests in the concept of making `pointcut` interfaces explicit *within* the software. In our approach a join point policy is a separate artifact living in a separate dimension of development. Aspect owners can simply ignore this dimension while prototyping their aspects, rather than being forced to program against a `pointcut` interface of some kind. Circumstances permitting, a preliminary join point policy can certainly help to direct the development towards a design where the final negotiation should be really easy, because the base owner’s interests have already been considered during aspect development. But in our approach technology imposes no limitations on what happens when. Similarly, when a base owner wants to ship the result of negotiation, he or she need not touch any source code, nor re-compile/build the component. Re-packaging existing binaries with a new policy is all that is needed.



From this we conclude that both mentioned approaches require significantly more (and earlier) involvement from the base owner, who in the confirmed join point approach only has to react to (and negotiate about) specific join point requests by the aspect owner, with no impact on the base implementation. This difference shouldn't be taken lightly, because in real world settings all should be done to disburden the base owner, who naturally has a lesser interest in any aspect bindings as compared to the aspect owner or even the user.

Enforcement A basic level of enforcement is given by the compiler: If, e.g., a project decides to not accept API decapsulation at all, a single switch in the compiler's configuration will turn all diagnostics concerning API decapsulation into errors, thus making violating source code uncompileable. At the default level, any decapsulation is reported as warnings. Here the developer has the option to either live with the warning or – if the compiler output should be tidied up – to add `@SuppressWarnings` annotations to the incriminated binding. Either way, the fact of decapsulation is visible, either as output from the tool or as annotations in the source code.

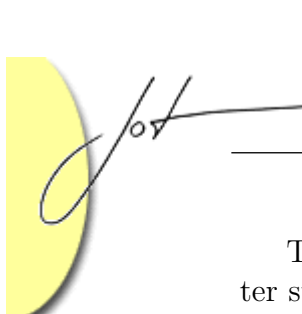
For more fine-grained control of decapsulation it is the join point access controller (JPAC) [19] which connects the social protocol of negotiation to a security protocol for enforcing the results of negotiation. The JPAC enforces that an application adheres at run-time to the policies defined in the negotiation phase. The signed base package including the policy file from the negotiation phase is installed at the user's site. As described above, the weaver checks (a) that the base and aspect components are correctly signed by their respective owners to ensure that neither the code nor the join point policy have been tampered with and (b) that the aspects weave into base components only at confirmed join points.

Some levels of tools support exist for creating and modifying policy files and for running applications at various levels of protection, like “development” or “secure”.

Migration Ideally, migration could be seen as moving from one confirmed state to another confirmed state (with more or less decapsulation — with equal or more functionality). However, in practical development it is important to also support a gray area, where confirmation is not (yet) given but functionality can already be integrated and tested, perhaps already shipped to courageous users.

At the language level this corresponds to a state where an OT/J program is still allowed to produce compiler warnings regarding unconfirmed decapsulation.

The JPAC can be configured to run, e.g., in “development mode” where decapsulation is allowed for all join points that are not explicitly denied. Before sending a request policy to the base owner, an aspect developer should run the JPAC in “request test mode”, to find out, whether the request policy is sufficient. This mode isn't strictly necessary, because the required set of join points can be calculated statically, but it may help to further the confidence in a given architecture even before join points are actually confirmed.



These considerations should make clear that gradual encapsulation provides better support for “opportunistic” software reuse and adaptation, than the mentioned approaches of re-establishing encapsulation in the presence of aspects [2, 8]. The safety achievable by these approaches can equally be established if a phase of opportunistic exploration is followed by a successful phase of negotiations producing an agreed and enforced join point policy. Note, that the base owner is free to put any knowledge about potential pointcuts into his module. So in exposing specific events of his module he may actually follow the proposed method of either mentioned approach. But in our case this is an option among others, not a requirement.

We are convinced that open modules and XPI are interesting *points* in the field between strictly encapsulated, hierarchical structures at the one extreme and fully unrestricted capabilities of – say – a privileged aspect in AspectJ at the other end of the spectrum. But both approaches are locked to a specific compromise, with all its advantages regarding join point decapsulation and its ignorance towards API decapsulation. The problem we see with open modules as well as XPI is their lacking support for dynamic project situations, where rules may need to be changed while the show is running. The problem is that both approaches postulate new sets of rules which are then carved in stone with no provision for exceptions.⁷ As a result from this attitude these approaches don’t provide suitable *paths for migration*.

On the practical side we have experience in migrating components that have been black boxes before, towards a more flexible integration. The Object Teams Development Tooling (OTDT) itself uses the OT/Equinox technology to adapt existing Eclipse plug-ins like JDT and PDE [10]. Moving from encapsulation towards decapsulation is directly supported by the technology.

During this development and before the OT/Equinox technology was ready we also had phases of *copy-and-paste reuse*, simply because we needed to make adaptations that were not possible by normal means. Expectedly, this copy-and-paste reuse was immediately successful but in the long run caused significant problems during maintenance. In this situation we migrated parts of the system from copy-and-paste to using OT/Equinox aspects. The copy-and-paste solution, which had adhered to the letter of encapsulation could do so only by actually abandoning one encapsulation boundary: due to inevitable ripple effects we had to adopt large parts of the base code, with no easy way for merging with new upstream versions. By migrating to an OT/Equinox solution we could re-establish the encapsulation boundary with only a narrow channel added by decapsulation.

During two cycles of migrating to a new version of Eclipse, this new architecture has proven much more robust and amenable to parallel evolution of base and aspect. This indicates that a narrow interface using decapsulation may in some cases actually produce the most robust architecture possible under the given organizational constraints.

⁷ By contrast, our approach contains a hint of self-irony, whereby it questions its own rules. One might speculate, that such self-irony is a great enabler of learning. In this light we actually favor to see our approach as the beginning of something, rather than a wrap-up of previous research.



In other cases OT/Equinox aspects have been used to prototype changes that later have been adopted by the Eclipse developers. This direction benefits from several properties of OT/J: Analyzability provides the means for isolating aspect-induced changes in order to refactor the changes to using only object-oriented mechanisms. Also, the similarity between the role-based relationship and regular inheritance helps to migrate some role-based implementations to inheritance-based implementation. This, of course, applies to only a subset of OT/J programs.

Evaluating migration based on negotiated join point policies remains a task for future research, where collaboration with external partners is sought.

6 CONCLUDING DISCUSSION

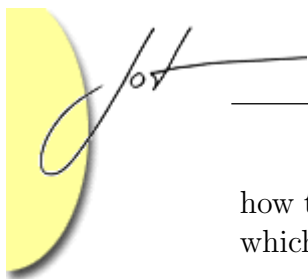
Costs vs. benefits Introducing join point policies as additional artifacts to be maintained raises the question how this scales for large projects involving many parties with even more developers, and evolving over time. For a base owner, confirming a join point has a similar impact as publishing an API: both acts impose restrictions on future evolution. A significant advantage of a join point policy is the fact, that a specific client (aspect owner) can be named, so that re-negotiating certain join points becomes feasible, which is not the case for globally published API.

In order to carefully compare different options it is necessary to be explicit about the alternatives. In many cases disallowing any decapsulation would either mean to adjust the requirements to the provided functionality of existing software assets, *or* to refrain from reusing a specific software asset and building a better fitting one from scratch. From an economical point of view both these alternatives may not be acceptable.

If adaptations indeed have to be performed, one could also require the base owner to add explicit variation points to their software. In Equinox terminology this could mean to add a new extension point. If a client requests rather few extension points this might in rare situations even be tractable. A larger number of client-requested extension points will usually not be realistic, because each additional extension point (a) pollutes the source code with stuff that provides no functionality for the large majority of uses and (b) may cause runtime overheads that affect all users of this software. Compared to equipping the base program with new variation points, adding an entry to a join point policy requires tremendously less efforts, both initially and during evolution. This approach incurs no runtime overhead on the normal user.

A final alternative would be to allow uncontrolled adaptations by aspects. As discussed above this would result in contractually unclear situations, where any previous liability may become void and users would be left running the composed software totally at their own risk. In many projects these considerations render such uses of aspects unacceptable.

We conclude that *especially* in larger settings involving several parties we see no alternative to gradual encapsulation that comes close to the benefits of gradual encapsulation without incurring significant new problems. It remains to be seen



how the burdens of negotiation and maintenance will be split between stakeholders, which may likely include new kinds of payments for new kinds of services.

Work on encapsulation Obviously, literature about encapsulation is abundant and impossible to discuss here in full. Many publications have proposed individual points in the design space. It seems that join point decapsulation can only be discussed in the context of aspect-oriented programming, which, however, is not fully true, because any mechanism for implicit invocation may produce similar situations. Surprisingly, aspect-oriented literature has hardly produced any discussion about API decapsulation.

The OCaml module system [14] could be seen as an existing example of API decapsulation. In OCaml signatures (interfaces) can be bound to structures (the implementation) after the fact, thus exposing elements of a structure just as needed. At a closer look, this is better described as optional “after-the-fact encapsulation”, because binding a signature can only hide existing members and never expose previously hidden ones. It is not clear how this would scale up in a commercial setting for enforcing encapsulation in a system composed from third-party modules.

The Eclipse community has also identified a need for more specific interface definitions and introduced new annotation types to be checked by supplementary API tooling [5]. E.g., the `@noextend` annotation disallows extending a given class across component boundaries thus restricting the scope of the fragile base-class problem and ruling out any attempt to bypass visibility restrictions by using inheritance. Still these annotations cannot grant access to individual client modules, only.

One thing is more important than the individual critique of all those different proposals for supporting encapsulation: each approach we have seen so far only describes one individual point in the design space, but what is needed is a conceptual tool for positioning a concrete project on its *specific* sweet-spot in this design space and to empower it to *move* to a different point when the forces determining the project change.

Gradual encapsulation This paper presented a new approach for developing and evolving modular designs in full awareness of the organizational context and its constraints. The approach itself can be adopted gradually.

Using just about any aspect language, a given base system can be decapsulated so that regular aspects apply unrestricted join point decapsulation and privileged aspects additionally apply unrestricted API decapsulation. If the absence of decapsulation shall be enforced, one should refrain from using such languages.

With the support of obligatory declarations, as it has been shown for OT/J, visibility of decapsulation can be significantly improved. At this level, already the compiler “knows” about the double-edged nature of decapsulation and the capability of decapsulation can be controlled by corresponding compiler options. Based on the visibility granted, experienced developers can already utilize most benefits of gradual encapsulation.



With the additional support of join point policies the technology can be fully embedded into the organizational context. As a last step, using a join point access controller the results of negotiation can be technically enforced at runtime. At this level of adoption, gradual encapsulation indeed implies a new business model for building software using third-party components. This step might feel like an undue burden, but we strongly believe that AOP on the long run has no other choice but making its business model explicit. If AOP indeed involves any kind of decapsulation, the business model has to account for this.

The technology we have developed for OT/J, including OT/Equinox and the JPAC, provides end-to-end support for gradual encapsulation. We have practical experience with the unilateral application of OT/J and OT/Equinox for adapting Eclipse in unanticipated ways. Here, technology significantly helped to achieve and maintain a design with comparatively few and narrow channels of decapsulation implementing a large number of adaptations that without decapsulation would be close to impossible.

Technology, however, can only provide a platform for gradual encapsulation. In the end development communities will evolve the actual rules for negotiating interfaces to their needs, but we are convinced that new rules are indeed needed and that gradual encapsulation has the potential for supporting new kinds of software reuse and cross-organizational cooperation among developers.

REFERENCES

- [1] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *Proc. of ICSE 2002*. ACM, May 2002.
- [2] Jonathan Aldrich. Open modules: Reconciling extensibility and information hiding. In *Proc. of the SPLAT workshop at AOSD'04*, March 2004.
- [3] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Proc. of the FOAL workshop at AOSD'02*, Enschede, The Netherlands, March 2002.
- [4] D. S. Dantas and D. Walker. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 383–396, New York, NY, USA, 2006. ACM.
- [5] Eclipse API tools. Homepage: <http://www.eclipse.org/pde/pde-api-tools>, Description: http://wiki.eclipse.org/PDE/API_Tools.
- [6] Erik Ernst. Family polymorphism. In *Proc. of ECOOP'01*, number 2072 in LNCS, pages 303–326. Springer Verlag, 2001.
- [7] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Proc. of the Workshop on Advanced Separation of Concerns at OOPSLA 2000*, October 2000.

- [8] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.
- [9] S. Herrmann, C. Hundt, and M. Mosconi. ObjectTeams/Java Language Definition version 1.0 (OTJLD). Technical Report 2007/03, Technische Universität Berlin, 2007.
- [10] S. Herrmann and M. Mosconi. Integrating Object Teams and OSGi: Joint efforts for superior modularity. In *Journal of Object Technology*, 6(9), 2007.
- [11] Stephan Herrmann. Confinement and representation encapsulation in Object Teams. Technical Report 2004/06, Technical University Berlin, 2004.
- [12] Stephan Herrmann. Balancing language concerns – who decides? In *Proc. of the SPLAT workshop at AOSD’08*, Bruxelles, Belgium, March 2008.
- [13] G.T. Leavens and C. Clifton. Multiple concerns in aspect-oriented language design: a language engineering approach to balancing benefits, with examples. In *Proc. of the SPLAT workshop at AOSD’07*. ACM, 2007.
- [14] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
- [15] ObjectTeams/Java web page. <http://trac.ObjectTeams.org/ot/wiki/OTJ>.
- [16] Harold Ossher. Confirmed join points. In *Proc. of the SPLAT workshop at AOSD’06*, Bonn, Germany, March 2006.
- [17] J. Siek and W. Taha. Gradual typing for objects. In Erik Ernst, editor, *Proc. ECOOP 2007*, volume 4609 of *LNCS*, pages 2–27. Springer, 2007.
- [18] K. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. SEI Series in Software Engineering. Addison-Wesley, 2001.
- [19] Jürgen Widiker. Policy-basierte Zugriffskontrolle für Joinpoints in der aspektorientierten Sprache ObjectTeams/Java. Diploma thesis (in German), Technische Universität Berlin, 2007.

ABOUT THE AUTHOR



Stephan Herrmann received his Ph.D. at Technische Universität Berlin in 2002 for his work on applying new techniques for separation of concerns to the development of a multi-view software engineering environment. Since then his focus is on developing the language ObjectTeams/Java and its tools. He has taught ObjectTeams/Java both at university and in tutorials at international conferences.