

# Balancing Language Concerns: Who Decides?

Stephan Herrmann  
Technische Universität Berlin  
stephan@cs.tu-berlin.de

## ABSTRACT

Language design focusing on a single goal like, e.g., obliviousness, may easily interfere with other desirable properties. As proposed by Leavens and Clifton, a language engineering approach should apply a number of continuous scales for assessing a language's support for the multiple concerns to be considered. This leads to languages that implement a compromise between different concerns in order to balance benefits and drawbacks. In order for such a compromise to fit for all software development projects language engineers would need to be omniscient. Therefore, I propose a more modest approach, where language engineers only set the stage for negotiations between the actual stakeholder of development. Using encapsulation as an example concern, I sketch the concept of "gradual encapsulation" where each project at each point in time may choose the appropriate balance between safety and flexibility.

## 1. MULTI-CONCERN LANGUAGE ENGINEERING

In some respect this paper is a response to the SPLAT'07 paper by Leavens and Clifton[8]. I highly appreciate their statement, as it helps to connect the academic discussion about language design to the needs in real world projects. They propose a shift from language design to language engineering in terms of acknowledging that useful languages do not result from pursuing one or two individual design goals, but the consideration of many concerns is needed, some of which may be in conflict with each other.<sup>1</sup>

Thus instead of reaching a single goal, a good language engineer will balance multiple concerns. A language engineer does not ask "have I achieved goal X" but rather, "*to what degree* does the language support concern X". A language engineer must make compromises in order to balance various benefits (and drawbacks).

<sup>1</sup>In [6] I have discussed the (im)possibility to design a language based on purely orthogonal features.

While I couldn't agree more on the importance of considering multiple concerns and striving for good compromises, I strongly believe that these points should be taken one step further: Why should the language engineer define such a compromise? If a language engineer were to optimize the benefits along several concerns, how could he or she determine an optimum? Can the fitness of a language be measured outside the context of any projects that use the language to produce software? I don't believe that one should expect such wisdom from any language engineer as would be needed for optimizing their product (a programming language) once and for all, independently of the final context of use of this product. Or would, at the other extreme, each project require its own language engineer to first produce the specifically balanced language for this particular project? Obviously, this approach would be too costly.

To overcome this dilemma, I suggest to consult another SPLAT paper: Ossher's contribution to SPLAT'06[10]. His concept of confirmed join points shifts the issue of balancing between obliviousness and encapsulation from language design to those stakeholders who actually relate to the final software to be produced: As a key concept that paper introduces explicit negotiation between an *aspect owner* and a *base owner*, which should result in explicit confirmation of specific join points (with other join points remaining hidden or denied). The novelty in this approach lies in the shift from a purely conceptual or technical matter, to a social or organizational issue, and thereby from a universal issue, solved once and for all by an omniscient language designer/engineer, to an issue that can be addressed individually in the exact context where a specific compromise can be adequately evaluated.

In this position paper, I take as a representative for potentially conflicting goals the matters of encapsulation vs. flexibility. I propose to equip the programming language and its supporting technology with means to support strict encapsulation *as well as* powerful means for flexibly adapting existing programs. I will sketch how both concerns can indeed be addressed by a single language (Sect. 2).

Next (Sect. 3) I will sketch four principles that are geared to managing the newly gained powers in a structured way. With the technology and the principles in place, I propose to think of encapsulation as a concern to be evaluated on a continuous scale (in line with [8]), which leads to the term "gradual encapsulation". For long-term maintenance and

evolution of software I furthermore suggest that a software project should be able to adjust the balance between encapsulation and flexibility “on stage”, i.e., during the life-time of existing pieces of software. Gradual encapsulation helps to eliminate bad surprises of the form that small changes in the requirements produce huge efforts of changing the software. Such discontinuities have been observed especially in the field of COTS software [12], and generally occur whenever an initial software design turns out to be inappropriate for changes to be made at some later point.

AOP already avoids some discontinuities in so far as it untangles some design decisions. AOP, however, can add higher-level discontinuities, because the question whether or not obliviousness and the breaking of encapsulation are allowed in a particular project can typically not be answered on a gradually scale. Once a project has decided in this matter it is usually locked in this decision without the option to slightly adjust the balance.

## 2. 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*.

### 2.1 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 high 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.<sup>2</sup>

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.

- 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.

<sup>2</sup>Unlike [8] I 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 **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” [2]) 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 I propose to consider any aspect as breaching encapsulation *unless* it is proven to be harmless.

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.

If decapsulation would vitiate any attempt of modular reasoning about the system, we would see good reasons to completely ban decapsulation. Thus, decapsulation is only tolerable if it becomes manageable. In the following we define four principles that are designed to providing all necessary means for effectively controlling decapsulation while still admitting the desired flexibility.

## 3. PRINCIPLES FOR GRADUAL ENCAPSULATION

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 stakeholder 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. Enforcement should be supported in all phases, ranging from static analysis during development up-to dynamic checks at run-time.

**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.

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 user. 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 he or she 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 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.

## 4. GRADUAL ENCAPSULATION IN OBJECT TEAMS

In order to illustrate the concepts introduced above, I will sketch how gradual encapsulation is supported by the programming language ObjectTeams/Java (OT/J, [9, 4]) and its tools.

### 4.1 Encapsulation

OT/J supports the normal styles of encapsulation in Java, plus some more advanced options: by using family polymorphism[3] a team *instance* actually protects its contained role *instances* which is stronger than Java's static visibilities. In

[7] I 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.

Finally, by the integration of OT/J with the component framework Equinox (Eclipse's implementation of the OSGi standard) [5], developers can exploit the encapsulation support from OT/J and Equinox together.

### 4.2 Decapsulation

In OT/J, join point decapsulation is supported by "callin" method bindings. API decapsulation applies to base classes bound to a role using "playedBy" and to base methods accessed via "callout" method bindings. 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 regular access within the body of any method must still adhere to the normal rules of visibility.

### 4.3 Principles

*Analyzability.* By limiting decapsulation to playedBy and callin/callout method bindings, the state of affairs regarding decapsulation can very easily be analyzed for a given software. Furthermore, the design of the OT/Equinox technology [5] puts special emphasis on raising visibility of decapsulation. In OT/Equinox even the architectural view shows all (potential) locations of decapsulation. Thus, in a given OT/Equinox architecture, it is easily possible to zoom in and out the architecture and at each level make precise statements about the extent of decapsulation.

*Negotiation.* In order to support negotiation, we have implemented tool support for a slight variant of Ossher's proposal of confirmed join points [13]. The contribution of that development is in connecting the social protocol of negotiation to security protocols for enforcement.

In this model, an aspect developer creates a join point policy file specifying the locations in base components where aspects (wish to) apply decapsulation.<sup>3</sup> 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.

<sup>3</sup>Despite the name "join point policy" this file includes specification of API decapsulation, too.

**Enforcement.** A basic level of enforcement is given by the compiler: If, e.g., a project decides to not accept API decapsulation, a single switch in the compiler's configuration will turn all diagnostics concerning API decapsulation into errors, thus making violating source code uncompileable.

The Join Point Access Controller developed in [13] 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. 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.** We have concrete 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 [5]. Moving from encapsulation towards decapsulation is directly supported by the technology.

In some cases this technology has also been used to prototype changes that later have been adopted by the Eclipse developers. This direction benefits from several properties of OT/J: Analyzability as discussed above 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.

The different execution modes of the join point access controller support the migration between different levels of enforcement. Regarding concurrent evolution of base and aspect components we have some experience from migrating the OTDT from one Eclipse version to the next, which shows that the dependencies introduced by decapsulation are far better manageable than any other technique we could find for achieving the same functionality.

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

## 5. 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. 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. E.g., in Equinox terminology this could mean to add a new extension point. If a client requests one or two 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. In comparison, 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 use the composed software at their own risk. In many projects these considerations render such uses of aspects unacceptable.

I conclude that *especially* in larger settings involving several parties I see no alternative to gradual encapsulation that comes close to the benefits of gradual encapsulation without incurring significant new problems.

## 6. CONCLUDING DISCUSSION

I have sketched an approach by which language engineers may refrain from taking premature decision, in favor of passing options down to the stakeholders in individual projects. In this respect I follow the suggestion from [8] to not think of particular *goals* to be achieved (all-or-nothing) during language design, but rather of *concerns* to be addressed and balanced.

However, while doing so, I have identified a higher level goal: re-establishing continuity where technology tends to introduce accidental chasms. This means that the choice of a particular language/technology should not lock a project into a pre-defined policy (e.g., regarding obliviousness vs. encapsulation), but small changes on the requirements side should *always* require only small changes in the implementation. It is important that this should not only affect the concrete design but even more so the general policies and rules underlying the design.

The notion of gradual encapsulation was also inspired by the concept of gradual typing [11]. Both approaches share the intention to bridge the gap between different paradigms of software development. [11] introduce support for hosting statically typed and dynamically typed pieces of software within the same program using even the same programming language. It remains to be shown, what other tradeoffs in language engineering could also be passed as options to the actual developers using the particular language. The extreme position — admitting the full power of meta programming — obviously conflicts with the goal of predictability at any chosen level. For some concerns it is already practiced to keep the language specification rather abstract (like saying: their must be *some kind of* garbage collector) while still allowing different concrete implementations (of garbage collectors) with significantly different behaviours to be plugged into the language runtime.

In order to support gradual transition also for beginners learning a new language, it is important that the technology should be configurable very easily to either extreme. For an OT/J project it suffices to switch the warning level for decapsulation to either “Error” (i.e., no decapsulation is possible in the entire project) or to “Ignore” (i.e., decapsulation can be used without any additional efforts). More fine grained checking and negotiation is only necessary when actually required by otherwise conflicting demands. This means that even the transition towards gradual encapsulation can be taken gradually: you may start a project with a simplistic policy and gradually introduce more detailed specification and negotiation to growing parts of the projects.

## 7. REFERENCES

- [1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *Proc. of ICSE 2002*. ACM, May 2002.
- [2] Daniel S. Dantas and David 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.
- [3] Erik Ernst. Family polymorphism. In *Proc. of ECOOP'01*, LNCS 2072, Springer Verlag, 2001.
- [4] S. Herrmann and C. Hundt and M. Mosconi. ObjectTeams/Java Language Definition current version (OTJLD). <http://www.ObjectTeams.org/def/>, 2002–2008.
- [5] S. Herrmann and M. Mosconi. Integrating Object Teams and OSGi: Joint efforts for superior modularity. In *Proc. of TOOLS Europe 2007, also in: Journal of Object Technology*, 6(9), 2007, 2007.
- [6] Stephan Herrmann. Orthogonality in language design – why and how to fake it. In *Workshop on Object-oriented Language Engineering for the Post-Java Era, at ECOOP*, Darmstadt, 2003.
- [7] Stephan Herrmann. Confinement and representation encapsulation in Object Teams. Technical Report 2004/06, Technical University Berlin, 2004.
- [8] Gary T. Leavens and Curtis 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*, page 6, New York, USA, 2007. ACM.
- [9] Object Teams home page. <http://www.ObjectTeams.org>.
- [10] Harold Ossher. Confirmed join points. In *Proc. of the SPLAT workshop at AOSD'06*, Bonn, Germany, March 2006.
- [11] Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *Proc. ECOOP 2007*, volume 4609 of LNCS, pages 2–27. Springer, 2007.
- [12] K. Wallnau, S. Hissam, and R. Seacord. *Building Systems from Commercial Components*. SEI Series in Software Engineering. Addison-Wesley, 2001.
- [13] Jürgen Widiker. Policy-basierte Zugriffskontrolle für Joinpoints in der aspektorientierten Sprache ObjectTeams/Java. Diploma thesis (in German), Technische Universität Berlin, 2007.