

From Aspect-Oriented Model to Implementation

Watch Out for Impedance Mismatch

Johan Ovlinger
Northeastern University
johan@ccs.neu.edu

Abstract

An important part of both comprehensibility and traceability is that the implementation resemble the design. As the design is refined to a concrete implementation, it is important that concepts have a clear correspondence to implementation artifacts – even if this mapping is not one-to-one. This holds true for crosscutting – or aspect-oriented – behaviors as well. We illustrate by investigating the Observer pattern, modeled using subject-oriented extensions to UML, and implemented in AspectJ, identifying three mismatching properties between design and implementation. We conclude that these are due to *impedance mismatch* between the abstract design and the concrete language.

1 Modeling Aspects

Modeling is a key phase in software engineering, allowing the software production process to be represented at a variety of stages and levels of detail. At first glance, modeling should be a natural fit for aspect-oriented programming or multi-dimensional separation of concerns, as the act of modeling naturally represents the same program in a variety of views, much like modeling will present the same program from a variety of viewpoints. However, for a model to take advantage of the powers of AOP, aspects need to be captured as modeled entities in their own right.

The issue of modeling crosscutting concerns is covered by Clarke [Cla02]. She identifies the mismatch between how object-oriented design and the requirements are typically decomposed, and presents *subject-oriented designs* that aim to align more closely to the requirements. The crosscutting nature of such designs is captured using *composition relationships*, which specify how design models relate, illustrating both *merge* and *override* relationships.

Clarke and Walker [CW02] suggest mappings from subject-oriented designs and their composition relationships to AspectJ [KHH⁺01]. Their strategy allows such designs to be implemented using popular aspect-oriented languages, but it is unclear how easy

such an implementation will be to keep current with an evolving design.

An important part of both comprehensibility and traceability is that the implementation resemble the design. As the design is refined to a concrete implementation, it is important that concepts have a clear correspondence to implementation artifacts – even if it is not one-to-one. For example, an association between classes in UML can be bi-directional, unlike the references offered by programming languages, but it is not difficult to implement bi-directional association semantics using references.

The traceability requirement applies equally to the implementation of crosscutting behaviors. The expectation is that the increased expressiveness of aspect-oriented programming languages will allow implementations to more closely resemble designs.

We investigate how well this works on a well known example, expressed as a subject-oriented design and then implemented in AspectJ. We find that while AspectJ does a significantly better job of maintaining traceability than plain Java would, several mismatches between design and implementation remain. We believe that these mismatches are fundamental to the differing assumptions of the abstract design language and concrete implementation language we have chosen for the comparison. We call such differences *impedance mismatches*, and identify three such mismatches in the example. Lastly, we briefly illustrate how the situation would fare if the design were implemented in a language that more closely resembled the subject-oriented design language.

2 Design to Implementation

We'll illustrate our concerns by investigating the Observer pattern, modeled using the subject-oriented extensions to UML from [Cla02], and implemented in AspectJ. Figure 1 shows the subject-oriented design of the familiar Observer pattern, and how it is composed with a base application two times.

The Observer Pattern's design consists of two roles: Watched and Observer, which contain some behavior, and an association between the roles.¹ The roles' behavior contains both concrete and deferred methods. Watched provides a method `getObservers` that returns an array of the observers, while the Observer provides a similar getter for its end of the association. To keep the presentation concise, methods to manipulate the association have been omitted.

¹In the Observer Pattern, an Observer normally observes a Subject, but this is easily confused with the *subject* construct in subject-oriented design.

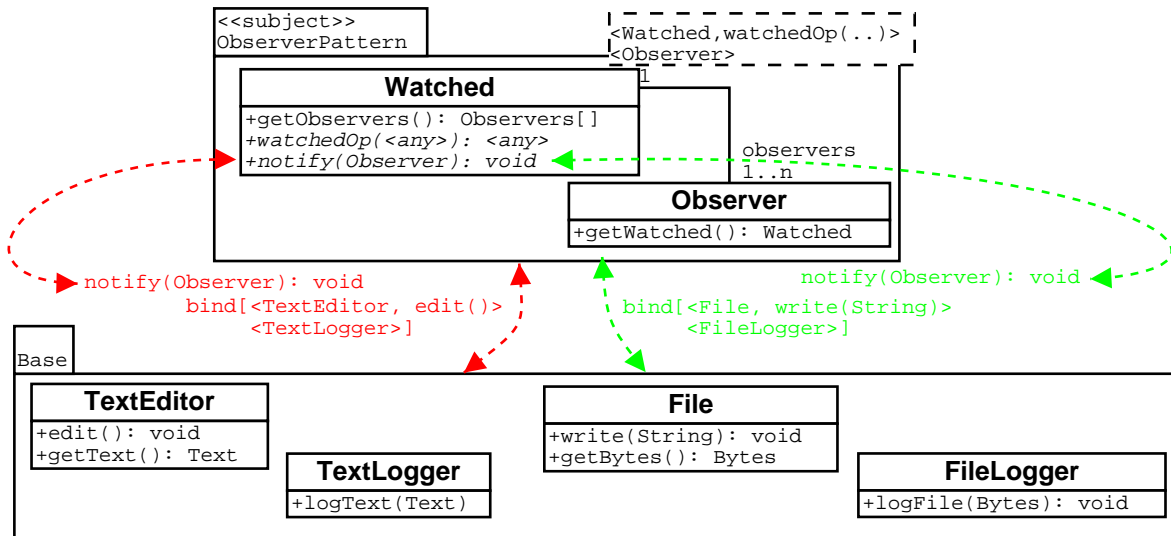


Figure 1. The Observer Pattern as a Subject-Oriented Design.

Watched has two deferred methods: these must be *provided* via composition, in a later design stage. The generic `watchedOp` is advice for the method whose invocation triggers the observer. Its implementation is specified as iterating over the array of observers and passing each `Observer` as an argument to the `notify` method. The last method, `notify` is declared as deferred on `Watched`, taking an `Observer` as argument, rather than the other way around.² Since `notify` will be invoked by the implemented methods in the `ObserverPattern` subject, its eventual implementation is constrained to have the same signature as the place-holder.

The `ObserverPattern` subject is composed twice with `Base`, once to observe `TextEditor` edits, and once to observe `File` writes. The arrows and `bind` relationships corresponding to each composition have been colored the same color: this should translate to different greys on a laser-printer. Both `TextEditor` and `File` have associated Loggers, which become the corresponding `Observers`. The `notify` methods are provided by `log` methods that are defined outside the subject. We signify that the actual implementations of these methods are provided by the composition by putting them outside the `Observer` or `Base` subjects.

It is unclear how subject-oriented design deals with compositions that are not purely crosscutting nor merging. We have attempted to model this in a similar fashion to the examples describing subject-oriented design: `bind` is used for crosscutting behavior, while a merge arrow indicates corresponding operations (`notify` in this case).

2.1 An AspectJ implementation

Clarke and Walker [CW01, CW02] suggest several mappings of subjects to AspectJ. We have chosen the one that was deemed most promising by the authors, using Java interfaces to implement subject roles, and modeling composition as the inheritance of abstract advice. A similar strategy is used by Hanneman and Kiczales

²This is for exposition: we wish to illustrate what happens when a reusable design defers a method that takes a role as an argument. Our justification is catering to flexibility: by taking the `Observer` as argument, double dispatch can be implemented.

[HK02] to investigate reusable implementations of design patterns using AspectJ. Figure 2 illustrates using this strategy to implement a reusable `Observer` pattern, while Figure 3 shows how to compose that implementation twice with `Base`. We have omitted showing `Base`, as it is assumed to be implemented in plain Java.

The following sections highlight the salient properties of the implementation that are not apparent in the design and vice-versa.

2.1.1 Symmetry

The design symmetry of `Base` and `ObserverPattern` is not reflected in the implementation: some roles become interfaces, other become classes, and the bi-directional arrows of the model have become directed “implements” arrows. This implementation strategy would not suffice if we wanted `Base` to advise some events in `ObserverPattern`.

Another reflection of the symmetry of the design being implemented in an asymmetric manner is that a composition should really produce a new result, whereas our implementation side-effects the `Base`. Symmetry is attractive, as this would allow us to compose complex aspects from smaller aspects, and to increase comprehensibility by reusing designs at multiple levels. While the design is symmetric, the implementation language constrains us to implement an asymmetric version.

We say that there is an *impedance mismatch* between the design language and the final implementation. As the design language explicitly allows hierarchical designs, the impedance mismatch goes from aesthetic to critical when we try to implement such designs using our chosen implementation strategy.

2.1.2 Invasiveness

The design has two correspondence relationships between `TextEditor` and `Watched`, and `File` and `Watched`. These suggest that composing `Observer` into `base` may introduce a spurious relationship between `TextEditor` and `File`. We borrow a term from Ernst [Ern00], and say that the composition is *invasive* if it introduces any relationships in the implementation.

```

abstract aspect ObserverPattern {
  abstract pointcut watchedOp(Watched s);
  before(Watched s): watchedOp(s) {
    Observer [] obs = s.getObservers ();
    for ( int i=0; i<obs.length; i++) s.notify (obs[i ]);
  }

  interface Watched { void notify (Observer o); }
  Observer [] Watched.getObservers () { ... }

  interface Observer { }
  Watched Observer.getWatched () { ... }
}

```

Figure 2. ObserverPattern Subject.

Indeed, our example implementation *is* invasive, as all Watched role players in the system (such as TextEditor and File) now share Watched as a common super-type. We can thus test at runtime using java's instanceof to deduce how the system was assembled, and more seriously, mistakenly pass one attachment's subjects to another's notify method, resulting in a dynamic casting error. For an example, see the badMeth code snippet in Figure 3. This method will compile without a complaint, yet trigger a dynamic error.

For small systems, or systems where a subject is reused only once, invasiveness is not really an issue. However, the systemic nature of the typing relationships means that two hierarchically composed subjects may both independently reuse the ObserverPattern subject internally. These hierarchical subjects may not generate casting errors when used independently, but care would be required when reusing both in the same application, so as not to make analogous errors to those of badMeth.

The fundamental problem is that invasiveness exposes an *unintentional* type equivalence to the programmer – one that may not even be apparent from the documentation, in the case of 3rd party subjects – thus weakening the static typing of the application. This exposure happens because the implementation strategy did not match the intended semantics of the design.

The type equivalence induces secondary interactions with method overriding and visibility. The approach will fail if either File or TextEditor contains a notify or getObservers method, and will always expose notify to external invocation. Experimentations shows that the implementation *is* able to prevent the getter methods from external invocation if there is no overriding method on concrete class, and if they are marked private to the aspect.

2.1.3 Attachment

Each (re)use of the ObserverPattern subject involves several correspondences: how the roles of ObserverPattern correspond to those of Base, how the deferred methods are mapped to concrete implementations, and what the visibility of concrete methods across the subjects should be. These details are particular to one such use scenario, and contain the complete details of the composition relationship. We refer to the composition, along with all relevant details, as the *attachment* of the two subjects.

Attachment is a key entity to model, as it encapsulates precisely how two (or more) subjects relate. Issues such as how multiple attachments of the same subject relate can only be discussed if the attachment is modeled as an entity. As an example of such rela-

```

aspect ObserveTE extends ObserverPattern {
  declare parents: TextEditor implements Watched;
  declare parents: TextLogger implements Observer;
  void TextEditor.notify (Observer obs) {
    TextLogger tl = (TextLogger) obs;
    tl.logText ( this.getText ());
  }
  pointcut watchedOp(Watched s): call (* edit(..))&& target (s);
}

aspect ObserverF extends ObserverPattern {
  declare parents: File implements Watched;
  declare parents: FileLogger implements Observer;
  void File.notify (Observer obs) {
    FileLogger fl = (FileLogger) obs;
    fl.logFile ( this.getBytes ());
  }
  pointcut watchedOp(Watched s): call (* write(..))&& target (s);
}

void badMeth(TextEditor te, File f) {
  te.notify ( new FileLogger ());
  f.notify ( new TextLogger ());
}

```

Figure 3. ObserverPattern's composition with Base.

tionships, consider the association between Watched and Observer: in our implementation, we wanted each attachment to introduce a private association. Other situations might be reversed: a logging subject may well want to have one output stream *shared* between all attachments. Note that these sharing relationships are not at the object level – these are at the design level. Sharing between attachments may well introduce typing issues that must be dealt with – for example, a shared observer association would interact poorly with the dynamic downcasting highlighted in Section 2.1.2 – but such issues can only be considered after the interacting entities have been identified.

There is an impedance mismatch here as well, but interestingly it is in the other direction, with the implementation possessing the desirable property, while the design does not. The implementation groups each attachment into textual units – implemented as concrete sub-aspects – that connect the reused subject to the base application. However, in the design we were forced to resort to colors to signify the grouping of the merge correspondence relationships.

3 Concrete Proposal

There are basically two ways to increase traceability and comprehensibility: either use a better implementation strategy that preserves more modeling entities, or change either of the modeling or implementation languages to be a closer fit. This section outlines how Aspectual Collaborations (ACs) [LLO01] can be used to both model and implement crosscutting subjects.

Aspectual Collaborations is a research system that investigates the combination of module systems with aspect-oriented features. By combining them, we are able to leverage powerful properties of module systems, such as encapsulation, composition, and separation, to control the power of aspectual features. This allows mature methodologies for modular decomposition to be applied to modeling crosscutting behaviors. This is a similar argument to that of Szyperski [Szy92], which argues that inheritance cannot substitute

```

collab ObserverPattern ;
participant Watched {
  aspectual RV watchedOp(JP jp) {{
    Observer [] obs = getObservers ();
    for ( int i=0; i<obs.length; i++) notify (obs[i ]);
    return jp.invoke ();
  }}
  expected void notify (Observer o);
  Observer [] getObservers () {{ ... }}
}
participant Observer {
  Watched getWatched () {{ ... }}
}

```

Figure 4. Aspectual Collaboration implementation

for modular import.

A prototype implementation of ACs is available from [Ov1], and a self hosting re-implementation is underway.³ Additionally, a simplified semantics for Aspectual Collaborations using Featherweight Java [IPW99] is being analyzed to formally prove the validity of the claims we make regarding AC's static and dynamic properties.

Figure 4 shows the Aspectual Collaboration implementing the separately compiled ObserverPattern subject, while figure 5 shows how its object-code is composed with that of Base. The code is similar in structure to that of figures 2 and 3, but with significant differences in the detailed semantics. We'll highlight the differences by describing the code and comparing it to the AspectJ implementation (names have been chosen to correspond, where appropriate).

The complete program consists of three collaborations, as earlier, we don't show Base, which is not substantially different from a pure Java implementation of the UML design. Like a Hyper/J HyperSlice [TOHS99], a collaboration is declaratively complete, and can be analyzed in isolation – including separate compilation. This holds true for crosscutting features as well.

Collaboration ObserverPattern implements the reusable subject design. Its two roles (called *participants* in our nomenclature) resemble classes more than interfaces. The notable difference from classes is that they contain both imported methods (*expected* methods) and crosscutting advice (*aspectual* methods) that can intercede in other methods' executions. Thus, our participants closely resemble the role models they implement. Since ACs are a research prototype, we have only implemented “around” advice, so the aspectual method `watchedOp` needs to call `proceed` (*invoke*) explicitly. The signature of `watchedOp` is used to isolate it from the signature of the methods it intercepts, allowing it to advise methods of any signature.

Figure 5 applies ObserverPattern to the Base, generating a *new* collaboration, ObservedBase. It is declared to extend Base – this declaration is desugared to importing and reexporting the contents of Base – and has ObserverPattern *attached* twice. Like the AspectJ implementation, the attachments specify both type mappings and

³The syntax presented here reflects the full design, rather than the somewhat less capable implementation. The differences are of the nature that the implementation expects a desugared input, while the presentation reflects the more aesthetic input syntax. The desugaring in all cases is straight-forward, and does not involve semantic transformations.

```

collab ObservedBase;
extends Base;
attach ObserverPattern {
  TextEditor += Watched {
    around edit do watchedOp;
    provide notify with
      void ntfy (Observer obs) {{
        TextLogger tl = (TextLogger) obs;
        tl.logText ( this .getText ();
      }}
  }
  TextLogger += Observer { }
}
attach ObserverPattern {
  File += Watched {
    around write do watchedOp;
    provide notify with
      void ntfy (Observer obs) {{
        FileLogger fl = (FileLogger) obs;
        f.logFile ( this .getBytes ();
      }}
  }
  FileLogger += Observer { }
}

```

Figure 5. Aspectual Collaboration composition with Base.

pointcuts. A key difference to AspectJ is that the type-mappings are scoped within one attachment and are not based on inheritance: after composition, File will not be a subtype of Watched, nor will File and TextEditor have a common supertype. Similarly, methods visibility is by default constrained to an attachment, unless explicitly exported into the public scope. The lack of such exports means that the composition of Observer will not augment the signature of File or TextEditor with any additional methods or fields: not only would `badMeth` of figure 3 result in a static type error, the method `notify` would not be visible.

A useful feature is that collaborations are symmetric. Base or ObserverPattern could both be composed from smaller collaborations. Should both of these happen to reuse the same constituent collaboration, that constituent would be completely undetectable in the result. A compositional property is that no new dynamic error locations are introduced, so that an aspectual method will either generate a static error if misused or be guaranteed not to fail at compile time. This feature is a result of treating the types in the signature of an aspectual method as existentially quantified, so that they are guaranteed to be unique and encapsulated within each attachment.

Aspectual Collaborations is a research system, and as such do not attempt to offer the convenience features or level of engineering that more mature systems such as AspectJ do. However, we have attempted to illustrate that the powers of a full-fledged module system interact very well with aspect-oriented programming, allowing a greater degree of design traceability by minimizing the impedance mismatch to the design language. This allows traceability of design features down to the implementation level. The encapsulation features inhibit the design from being exposed at the implementation level, which makes a large class of dynamic errors impossible. The same features additionally temper the come-from nature of aspects [Cla74], which otherwise imply the need for tool support or programmer omniscience to comprehend local behavior.

4 Conclusion

We have looked at an existing proposal for an Aspect-Oriented Design Model, and a sample implementation of multiple reuse scenario. We have identified three issues in our small example:

- The fundamental hindrance to traceability was the impedance mismatch between the design language and the implementation language. Both comprehensibility and traceability suffer as a consequence of such mismatches, as the analyst or programmer must manually translate from one world view to the other.
- We have identified a case where it is harmful for the design to be evident in the implementation: invasiveness. This suggests that there is a balance between traceability and safety that must be engineered carefully so as not to allow undue development hurdles while allowing the design to penetrate deeply into the implementation.
- To maximize comprehensibility and reuse, it is desirable to encapsulate the details of the reuse into an attachment entity. This was an interesting case as the implementation language allowed a more useful design representation than the modeling language. It is however also possible that we failed to use the modeling notation to its fullest capacity.

Finally, we have suggested that one way to minimize impedance mismatch is to bring model and implementation closer together, allowing design features to be reflected almost at a one-to-one level in the implementation. We illustrate that this is possible by presenting a system based on a strong module system foundation that controls systemic nature of aspects, aligning their crosscutting nature more closely with the subject-oriented design language. It is attractive to imagine a system combining AspectJ's power to implement collaborations with the static safety of ACs to control visibility of classes, pointcuts, and aspects during composition.

An uninvestigated approach would be to instead vary the model language to more closely match the implementation language. However, a contra-indication to that approach is that the design language used in this paper – subject-oriented design – and our suggested implementation language – aspectual collaborations – were developed independently, yet fit together very well. This suggests that this organization has merit both as a modeling approach and as an implementation language.

5 References

- [Cla74] R. Lawrence Clark. A linguistic contribution to goto-less programming. *Communications of the ACM*, 1974. Originally published in *Datamation*, 1973, available online: <http://www.fortranlib.com/gotoless.htm>.
- [Cla02] Siobhan Clarke. Extending standard uml with model composition semantics. *Science of Computer Programming*, 44:77–100, July 2002.
- [CW01] Siobhan Clarke and Robert J Walker. Separating crosscutting concerns across the lifecycle: From Composition Patterns to AspectJ and Hyper/J. Technical Report TCD-CS-2001-15 and UBC-CS-TR-2001-05, Trinity College, Dublin and University of British Columbia, May 2001.
- [CW02] Siobhan Clarke and Robert Walker. Towards a Standard Design Language for AOSD. In Gregor Kiczales, editor, *First International Conference on Aspect-Oriented Software Development*, pages 113–119, Enschede, The Netherlands, 2002. ACM Press.
- [Ern00] Erik Ernst. Syntax Based Modularization: Invasive or Not? In Peri Tarr, Lodewijk Bergmans, Martin Griss, and Harold Ossher, editors, *Position papers from the workshop on Advanced Separation of Concerns, OOPSLA'00*. Department of Computer Science, University of Twente, The Netherlands, 2000.
- [HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 161–173, Seattle, Washington, 2002. OOPSLA'02, ACM SIGPLAN Notices 37(10) October 2002.
- [IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 132–146, Vancouver, BC, Canada, October 18–22 1999. OOPSLA'99, ACM SIGPLAN Notices 34(10) October 1999. Full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3), May 2001, pp 396–450.
- [KHH⁺01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, Hungary, June 18–22 2001. ECOOP 2001, Springer Verlag.
- [LLO01] Karl Lieberherr, David H. Lorenz, and Johan Ovlinger. Aspectual collaborations for collaboration-oriented concerns. Technical Report NU-CCS-01-08, College of Computer Science, Northeastern University, Boston, MA 02115, November 2001.
- [Ovl] Johan Ovlinger. Aspectual collaboration compiler web site. <http://www.ccs.neu.edu/home/johan/-research/acc>.
- [Szy92] Clemens A. Szyperski. Import is not inheritance, why we need both. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming*, number 615 in *Lecture Notes in Computer Science*, pages 19–32, Utrecht, The Netherlands, June 1992. ECOOP'92, Springer Verlag.
- [TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In D. Garland and J. Kramer, editors, *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, Los Angeles, California, May 16–22 1999. ICSE 1999, IEEE Computer Society.