Towards Patterns for Concern-Oriented Software Architecture

Mohamed Mancona Kandé[1,2] and Valentin Crettaz[1,2]
[1]Department of Management of Technology and Entrepreneurship
Swiss Federal Institute of Technology Lausanne, CH - 1015 Lausanne, Switzerland
E-mail: {Mohamed.Kande,Valentin.Crettaz}@epfl.ch

[2]Condris Technologies, PSE C, CH - 1015 Lausanne, Switzerland
E-mail: {Mohamed.Kande,Valentin.Crettaz}@condris.com

Abstract

This paper discusses the need of patterns for concern-oriented software architectures. It proposes a new pattern that serves as a basis for understanding, using and discussing concern-oriented patterns.

1 Introduction

The number of modern software engineering practices centered on the use of patterns is increasing. Patterns have been adopted in object-oriented analysis [9] and design [11], in advanced software development processes [16], in software architecture practices [4][5][21] and domain-specific architectures [8], and in component-based enterprise applications and middleware [2][6][10].

These approaches provide common support for selecting patterns from existing catalogues based on the forces at hand; but, they also usually require that the software be designed and implemented from the very beginning with the right modularization. However, future evolution of software is not predictable: even software implementations based on patterns must evolve when requirements change; they must be remodularized to comply with requirements evolution. Unfortunately, current pattern-oriented approaches lack support for identifying the *forces* in the solution schemas. Forces should *drive* the use of a given pattern, but they cannot be localized within the generic solution provided by that pattern.

We believe that without support for localizing forces in solution schemas, instantiating and reusing patterns will remain difficult; remodularizing pattern instantiations is even worse when the concerns related to individual forces crosscut various components and interconnections. However, to help improve the situation, we propose a concern-oriented approach to software architecture [18].

To offer a basis for understanding, using and discussing concern-oriented patterns, this paper introduces a new pattern for concern-oriented software architecture which enables remodularizations of existing and new software-intensive systems on-demand. The paper is structured as follows: Section 2 introduces a new notation for describing concern-oriented software architecture patterns. Section 3 presents a concern-oriented software architecture pattern, called On-Demand Remodularization. Section 4 describes the related work and section 5 concludes the paper.

2 Notation

The notation used to describe the new pattern is summarized in figure 1. With this notation, we propose another perspective on an AssociationRole, which is different from the standard UML definition. Essentially, our proposal consists of two key notions: perspectival associations and model slices [18]. The notion of *perspectival association* aims at focusing on interactions (i.e., what happens between instances of different Classifiers independently of the instances themselves) rather than on computation (i.e., what happens within an object). Thus, different perspectives on associations allow one to understand and express various aspects of an interaction among components. The notion of *model slice* focuses on the modularization of concern reification within software. For instance, different units (e.g., perspectival associations, methods and classes) that together reify a stockholder's concern into software can be described as a “perspectival module” represented by a model slice. Moreover, figure 1 shown different kinds of relationships: model slice to model slice (*refinement* and *application*) and unit to unit (*introduction* and *attachment*).

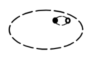
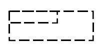
ODR Pattern Elements		Visual Representations	Description of Key Characteristics
Classifier Role			✓ Element of standard UML representing a participant of an interaction
Perspectival Association (PA)			✓ Mechanism for representing interaction concerns among a group of collaborating parts ✓ Consists of a collection of connection points, perspectival behavior, and Introductions ✓ Can be related with another PA only in precedence or inheritance relationships ✓ Cannot be understood in isolation (from all perspectives); requires reference to a base model
Connection point		<<connectpoint>> ●	✓ Locus for composing join points to which perspectival behavior can be added. ✓ Exposes its context in terms of parameters that can be used in the body of the associated perspectival behavior
Relationships	Attachment	– – – – –	✓ Attaches a role to a connection point
	Introduction	<<introduces>> – – – –>	✓ Supports structural amendments ✓ Supports the declaration of supplementary behavioral features ✓ Enables modification of an existing hierarchy.
	Refinement Application	<<refines>> – – – –> <<applies>> – – – –>	✓ Specify composition rules among model slices (see details of the pattern structure)
Model slice			✓ Mechanism for representing the reification of an individual concern ✓ Can be understood and reused in isolation, without reference to a base model ✓ Can be composed with others, or be a composition itself

Fig. 1. An Overview of the Notation used for Describing the Pattern

3 The New Pattern

In what follows, we describe the details of the new On-Demand Remodularization pattern using the pattern description schema proposed in [5][21].

The *On-Demand Remodularization* pattern provides the ability to remodularize both existing and new software systems along multiple kinds of concerns in a non-invasive way, and with no detriment of existing concern reifications.

3.1 Also Known As

ODR Pattern [18]

3.2 Example

Consider the development of an application that uses various components provided by independent software vendors (ISV). A development problem that motivates the use of this pattern consists of enabling such components to exchange data without providing themselves the data transfer capabilities.

For instance, the system to be built has five components of two different types—four *traffic lights* and a *timer*; thus, the requirements of the system are quite simple:

- The timer component is responsible for triggering an event at regular time intervals
- A traffic light component should always switch on the same light as its opposite peer
- A traffic light component should never show the same light as its direct neighbors
- A traffic light component should not maintain any knowledge of the state of its peers

The primary dimension of interest to address this problem is the *usability* of the application components. The main concern in that dimension is the *transfer of data*. A goal and an aspect of the problem which together characterize this concern are described as follows:

- **Goal**: improve usability by facilitating information exchange between two components
- **Aspect of the problem**: enable data transfer from one component to another in a non-invasive way

To solve the above problem, new functionalities need to be added to the application in order to support data exchange among the components. The ODR pattern allows one to achieve this in a non-invasive way.

3.3 Context

Any situations, in which we need to modify, add or remove functionalities to an existing system, transparently and in a non-invasive way.

3.4 Problem

Complex software systems often need to be modified to satisfy ever-evolving requirements coming from users, customers, changing technologies and environments. Addressing such modifications can be very difficult and expensive to achieve. We need new mechanisms for managing changes in complex systems.

The dimensions or driving forces relevant for solving this problem are: modifiability, adaptability, modularization, integrability, usability, reusability, understandability, and decoupling.

3.5 Solution

The solution proposed by the ODR pattern provides the ability to remodularize a software-intensive system according to various dimensions, non-invasively, and without eliminating concern reification based on prior decompositions.

The ODR pattern provides three important elements that work together to help remodularize an existing system into a more coherent, maintainable one. Therefore, the ODR pattern:

- Provides a mechanism for managing the interaction between participating components

- Enables particular system components to play the roles defined by the interaction protocol
- Allows one to customize a generic behavior for specific components types

3.6 Structure

The ODR pattern is structured into three different model slices, *connector*, *enabler* and *customizer*, and the relationships between these model slices. Figure 2 shows the overall structure of the ODR pattern.

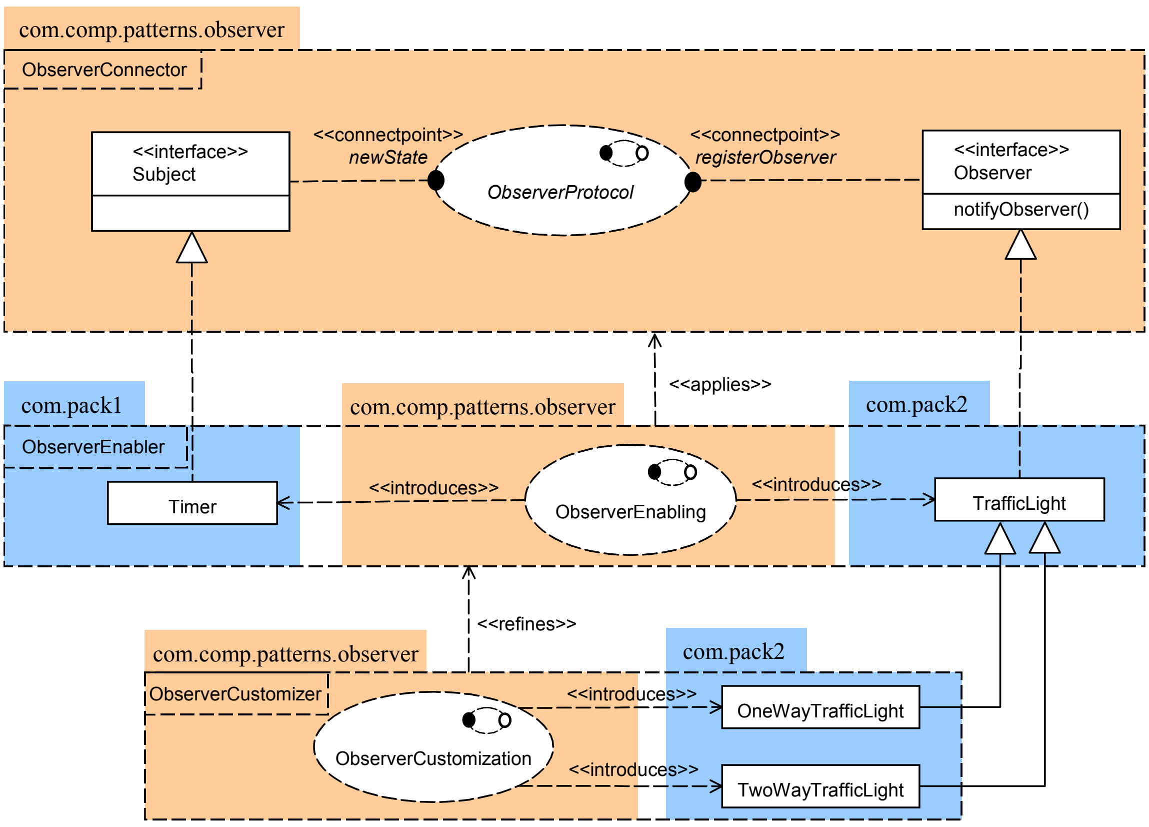


Fig. 2. Structure of the On-Demand Remodularization Pattern

A brief description of the model slices and their relationships is given in the following:

- The **connector** model slice defines the protocol and manages the interaction occurring among the participating roles.
- The **enabler** model slice provides mechanisms for enabling concrete components of the system to play the roles defined by the protocol of interaction in order to allow the components to participate in the interaction.
- The **customizer** model slice provides mechanisms for allowing individual components (children), which specialize existing components of the system (parents), to customize the generic behavior they inherit.
- The **relationships** between model slices essentially describe how to combine the model slices to make their units work together.

3.7 The Connector Model Slice

The structure of the connector model slice (named e.g., ObserverConnector) is described as a composition of a perspectival association and one or more interface type declarations:

- The *protocol* perspectival association (e.g., ObserverProtocol) represents a new mechanism for realizing communication protocols and managing the interaction among the collaborating roles (previously defined as interface types). Its interface consists of two connection points (e.g., newState and registerObserver). Each connection point identifies a specific point in the system at which the behavior of the participating roles should be glued with the interaction behavior. The dashed line represents the attachment between a connection point and a role that manifests this gluing.
- The interface types represent the roles to be fulfilled by the interacting components. In general, the behavior they declare must be provided by components playing the given roles.
- The connector model slice may also contain any other utility types, such as classes, that would help the perspectival association perform its job.

3.8 Enabler Model Slice

The structure of the enabler model slice (e.g., ObserverEnabler) consists of an enabling perspectival association and components of the system. The perspectival association and the components do not have to be defined in the same package; thus, in this case, the model slice crosscuts the boundaries of several packages (e.g., com.pack1, com.pack2 and com.comp.patterns.observer).

The key elements of the enabler model slice can be described as follows:

- The *enabling* perspectival association (e.g., ObserverEnabling) provides a mechanism for remodularizing a system by binding existing components to the roles defined in the connector model slice. The perspectival association provides support for introducing an additional behavior into participating components.
- The components represent the individual parts of the system (e.g., Timer and TrafficLight) that need to play the roles defined in the connector model slice.

3.9 Customizer Model Slice

The customizer model slice (e.g., ObserverCustomizer) is responsible for customizing and fine-tuning the behavior of components whose parent has already been adapted in the enabler model slice. This model slice can be optional when the behavior of the parent component is suitable for its children and does not necessitate any amendments. One or more customizer model slices can be used to refine the same enabler model slice: this allows us to dynamically change the customization of the component interaction. Here again, the customizer model slice composes various model elements defined in different packages (e.g., com.comp.patterns.observer and com.pack2).

The customizer model slice defines one or more customization perspectival associations and one or more specializations of the components defined in the enabler model slice:

- The customization perspectival association (e.g. ObserverCustomizer) allows one to specialize the generic behavior of components defined in the enabler model slice.
- The customized components inherit from the existing components defined in the enabler slice.

3.10 Relationships

The model slices presented so far must be combined to provide the overall structure of the ODR pattern by using four different kinds of relationships: applies, refines, introduces and attaches.

Relationships among Model Slices

- applies: This relationship applies a model slice to one or more other model slices. It
    1. exposes the public units (i.e., the interface types and their behavioral elements, e.g., Subject, Observer and notifyObserver()) of the connector model slice to the enabler model slice.
    2. requires the enabling perspectival association to realize all the abstract connection points declared in the perspectival association modeling the protocol of interaction.

  Visually, *applies* is shown as a dashed arrow in figure 2. It defines a directed relationship, for instance, from an enabler model slice to a connector model slice. As shown in figure 2, it applies an interaction module (i.e., *connector*) to a number of independent components that need to be composed into a system (e.g., Timer and TrafficLight). The connector interconnects the *components* and mediate their interaction.
- refines: defines a directed relationship (shown as a dashed arrow in figure 2) between a customizer model slice and an enabler model slice. It provides a means to the customizer model slice to refine or modify the behavior introduced by the enabling perspectival association into the components defined in the enabler model slice (e.g., TrafficLight). The behavior refinement is performed by the customization perspectival association (e.g., ObserverCustomization) that introduces the new behavior into the components defined in the customizer model slice (e.g., OneWayTrafficLight and TwoWayTrafficLight).

Relationships among Units of Model Slices

- introduces: defines a directed relationship (shown as a dashed arrow in figure 2) from a perspectival association to components of the system. It permits a perspectival association to introduce new features pertaining to the interaction protocol into existing components (e.g., Timer and TrafficLight). An introduction allows the components of the system to participate in the interaction. This is achieved by assigning the roles (e.g., Subject and Observer) defined in the connector model slice to the components defined in the enabler model slice (e.g., Timer and TrafficLight).
- attaches: defines a binding relationship between a connection point (e.g., newState) and an interface type representing a role (e.g., Subject) in the connector model slice. It is shown as an unnamed dashed line in figure 2, which in contrast to the other relationships has no stereotype.

3.11 Dynamics

This section provides a set of scenarios describing the runtime behavior of the ODR pattern. By runtime behavior, we mean the behavior of the system that results from the weaving process. The weaving can be performed at different times, including, but not limited to, compile-time, loading time and run time. Each scenario is described in terms of two types of diagrams: a concern-oriented collaboration diagram and a concern-oriented sequence diagram.

Essentially, the dynamics consists of describing the interaction behavior of the perspectival association defined in the connector model slice. In this model slice, the perspectival association provides the interaction behavior required by the system to make the components communicate.

To explain the dynamics of the structure shown in figure 2, we consider two examples of scenarios starting when the registerObserver and newState connection points are reached.

Scenario Associated to registerObserver

Figure 3 shows an concern-oriented collaboration diagram that models the scenario describing the behavior of the system when the execution flow reaches the registerObserver connection point. When a component playing the Observer role is created (e.g., a TrafficLight instance), it is registered by the ObserverProtocol in order to be notified at a later time.

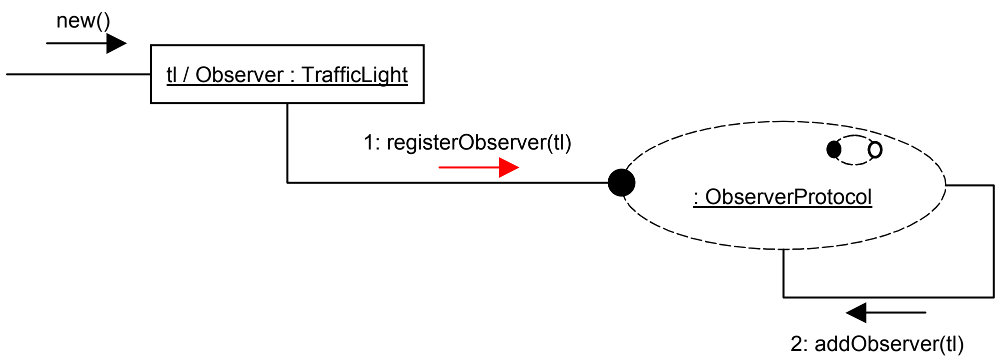


Fig. 3. Concern-Oriented Collaboration Diagram for the registerObserver Connection Point

Figure 4 illustrates an concern-oriented collaboration diagram that models the scenario describing the behavior of the system when the execution flow reaches the newState connection point. When a component playing the Subject role updates its state (e.g., tick() invoked on a Timer instance), the ObserverProtocol notifies all the registered Observers of the state change (e.g., by invoking notifyObserver() on each observer).

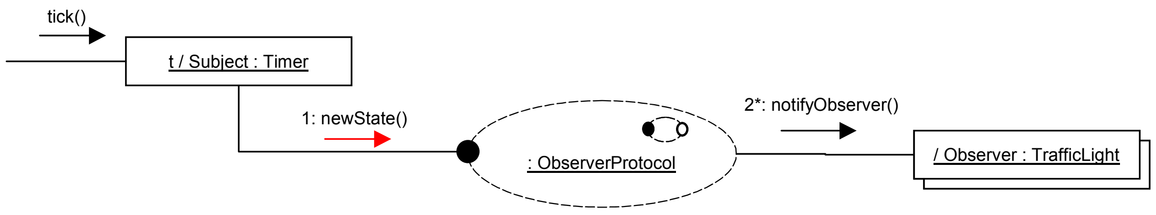


Fig. 4. Concern-Oriented Collaboration Diagram for the newState Connection Point

Figure 5 and figure 6 illustrate the concern-oriented sequence diagrams providing a different view of the above concern-oriented collaboration diagrams. In contrast to Figure 3 and figure 4, the sequence diagrams do not show the connection points. Instead, they graphically depict the execution times (shown by the vertical lines, called time lines) and the activations of instances (shown by the vertical activation boxes).

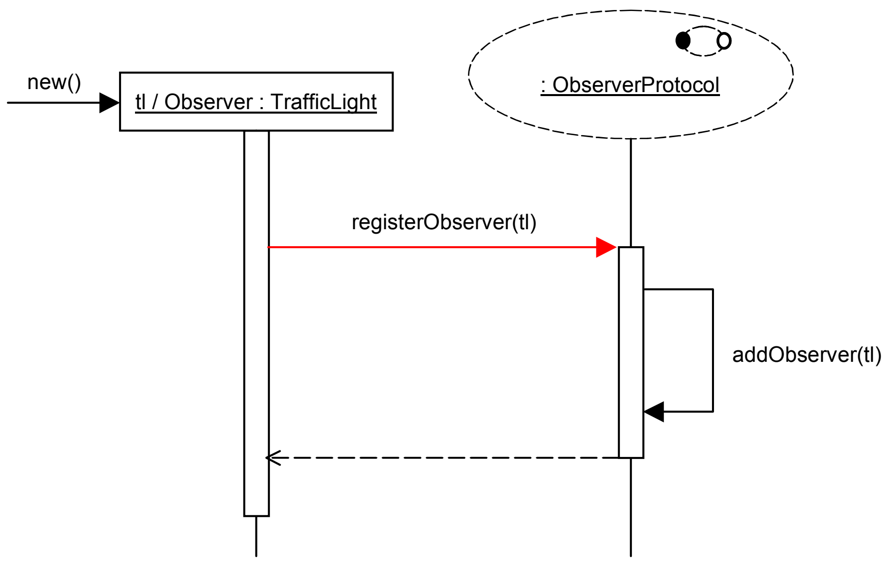


Fig. 5. Concern-Oriented Sequence Diagram for the registerObserver Connection Point

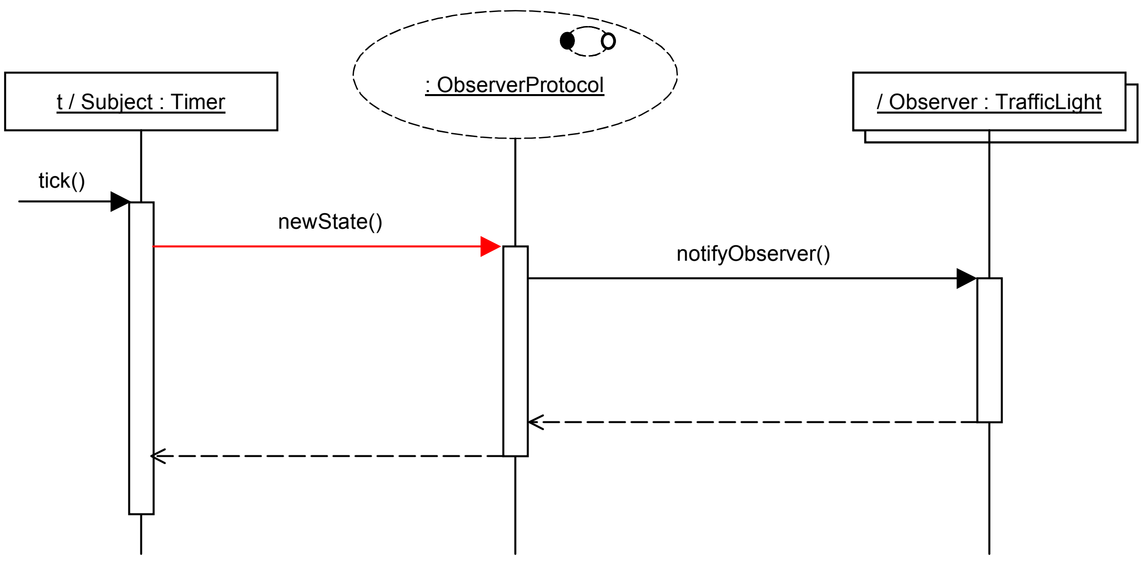


Fig. 6. Concern-Oriented Sequence Diagram for the newState Connection Point

3.12 Implementation

The implementation of the On-Demand Remodularization pattern can be realized in different ways using different aspect-oriented programming languages. In the following, we describe a sample realization of the ODR pattern by means of the Observer pattern [11] using the AspectJ programming language [3][12].

Figure 7 shows an AspectJ piece of code that implements the ObserverProtocol perspectival association as an aspect with the same name. The abstract pointcuts shown on lines 7 and 19 implement the connection points with the same names shown in Figure 2. The AspectJ advice (lines 10-16 and 21-24) realize the interaction behavior of the ObserverProtocol.

```
import java.util.*;

public abstract aspect ObserverProtocol {

  private List observers = new LinkedList();

  abstract pointcut newState();

  //On new state, propagate it to all registered observers
  after () : newState() {
    Iterator it = this.observers.iterator();
    while (it.hasNext()) {
      System.out.println("notifying observers...");
      ((Observer) it.next()).notifyObserver();
    }
  }

  //register the observer with this connector
  abstract pointcut registerObserver(Observer obs);

  after (Observer obs) : registerObserver(obs) {
    System.out.println("new observer...");
    this.observers.add(obs);
  }

  //introduce do-nothing methods into the observer role
  public void Observer.notifyObserver() {}
}
```

Fig. 7. An Implementation of the ObserverProtocol Perspectival Association

Figure 8 and Figure 9 present the interface types that provide the AspectJ implementations of the roles (Observer and Subject) to be played by the participating components.

```
// Observer role
public interface Observer {
    public void notifyObserver();
}
```

Fig. 8. An Implementation of the Observer Role

```
// Subject role
public interface Subject {
}
```

Fig. 9. An Implementation of the Subject Role

Figure 10 illustrates the implementation of the ObserverEnabling perspectival association. The ObserverEnabling aspect specializes the ObserverProtocol aspect; its pointcuts realize the abstract pointcuts of the ObserverProtocol aspect. The ObserverEnabling aspect implements the assignment of the Subject role to the Timer component, and the Observer role to the TrafficLight component. Furthermore, the ObserverEnabling aspect implements an additional behavior and introduces it into the TrafficLight component. This behavior consists of invoking the changeLight() method when the notifyObserver() method of the TrafficLight component is called.

```

public aspect ObserverEnabling extends ObserverProtocol {

  declare parents: Timer implements Subject;
  declare parents: TrafficLight implements Observer;

  pointcut newState():
    execution(void Timer.tick());

  pointcut registerObserver(Observer obs):
    this(obs) &&
    execution(TrafficLight.new(..));

  public void TrafficLight.notifyObserver() {
    this.changeLight();
  }
}
```

Fig. 10. An Implementation of the ObserverEnabling Perspectival Association

Figure 11 summarizes the mapping strategy for realizing the ODR pattern using the AspectJ programming language.

ODR Pattern Elements	AspectJ Language Elements	Comments
Role	interface	Representing a role using an interface is restrictive, since it only allows role to declare the obligations of a role in terms of operations. The prohibitions and the rights defined in the role cannot be expressed.
Perspectival Association	aspect	In contrast to the aspect construct of AspectJ, a perspectival association can be explicitly instantiated.
Connection point	pointcut	Each connection point can be realized by a set of join points
Relationship (introduces, applies, refines, attaches)	declares, extends, implements	The attachment between a role and a connection point is realized by a type pattern within a pointcut.
Model slice	-	May be implemented by many packages or by inserting custom tags in comments. The model slices do not directly correspond to programming-level units.

Fig. 11. Mapping the ODR Pattern Elements to AspectJ Constructs

3.13 Example Resolved

In this section, we apply the ODR pattern on a Drag'n'Drop architecture used in various development projects as illustrated in figure 12. This figure consists of the following model slices: DnDConnector, DnDEnabler, and DnDCustomizer.

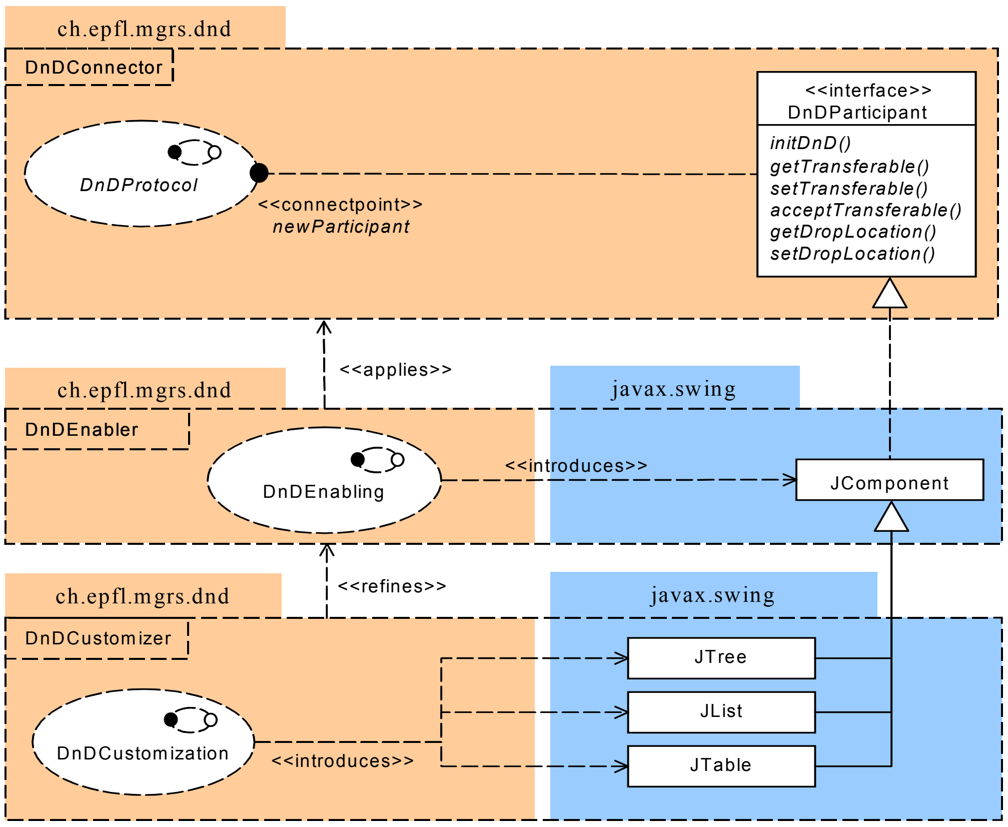


Fig. 12. An Application of the ODR pattern on the DnD Architecture

The DnDConnector model slice contains a perspectival association called DnDProtocol, and an interface type called DnDParticipant. DnDParticipant defines the behavior any component willing to participate in a Drag'n'Drop operation has to provide. A brief overview of each method declared in the DnDParticipant interface is given below:

- initDnD(): sets up a participating component for a future Drag'n'Drop operation
- getTransferable(): returns an object to be transferred when a Drag'n'Drop operation has been initiated on a participating component
- setTransferable(Object): provides the object that has been dropped on (transferred to) a participating component
- acceptTransferable(Flavor): returns a boolean indicating whether the participating component accepts the current object (denoted by its flavor or type) being transferred
- getDropLocation(): returns the exact point on the component where the transferred object has been dropped.
- setDropLocation(): sets the location where the transferred object has been dropped.

In addition, DnDProtocol defines a connection point that will detect when a new DnDParticipant is created within the system. When such a creation is underway, the perspectival association invokes initDnD() upon the participant in order to set it up for upcoming Drag'n'Drop operations.

The DnDEnabler model slice contains the DnDEnabling perspectival association whose primary goal is to allow any JComponent (of javax.swing) to participate in a Drag'n'Drop operation by giving to it the capability of playing the DnDParticipant role (defined in the DnDConnector model slice). Furthermore, DnDEnabling injects into JComponent behavior that shall be common to any JComponent (i.e., getDropLocation(), and setDropLocation()). All other methods need specific implementation that cannot be factored out into this model slice.

The task of the DnDCustomizer model slice is precisely to bridge that gap, and to introduce into subclasses of JComponent specific behavior for each of the remaining methods (i.e., initDnD(), getTransferable(), setTransferable(), and acceptTransferable()). The DnDCustomization perspectival association achieves this by inserting the implementation of an appropriate behavior, specified by the interface type DnDParticipant, into each of the participant components, such as JTree, JTable or JList. This is necessary because these components may realize the same role in different ways. For instance, a JList does not handle the drop of an object in the same way as a JTable or a JTree.

In summary, the DnDConnector model slice defines the interaction protocol between components engaged in a Drag'n'Drop operation (playing the DnDParticipant role). The DnDEnabler model slice applies the DnDParticipant role to components of the system. The DnDCustomizer model slice customizes the Drag'n'Drop behavior to each participating component. It is worth noting that this Drag'n'Drop ODR instance encloses all Drag'n'Drop concerns; there are no other places in the system where Drag'n'Drop concerns are present.

3.14 Variants

The Extensional ODR is a variant of the ODR pattern, which provides no protocol of interaction among the participating components. Its main purpose is to provide a mechanism for enhancing existing components by adding new functionalities. The interfaces defined in the connector model slice do not represent interaction roles. Instead, they stand for tag interfaces that can be filled with additional behavior.

3.15 Known Uses

The ODR pattern has been used successfully in several real world development projects by Condris Technologies and its partners.

3.16 See Also

Adapter, Mediator, Observer [11], Interceptor [21].

4 Related Work

This paper is generally related to a large amount of work on design and architectural patterns and to many efforts in the area of aspect-oriented modeling [1][7][22].

In particular, Tarr and Ossher first introduced the notion of on-demand remodularization [20] as a general-purpose mechanism of MDSOC (Multi-Dimensional Separation of Concerns) [23][24] that is required for encapsulating new concerns as they are identified throughout the software lifecycle. In practice, however, it is difficult to apply on-demand remodularization as a general mechanism for facilitating software evolution, reuse and integration *without* powerful tools supporting the remodularization process. This paper presents the ODR pattern as an "enabling tool" for on-demand remodularization. In contrast to MDSOC, this enabling tool represents a reusable architectural solution that can be instantiated repeatedly, in different contexts, and in a concern-oriented way.

Hannemann and Kiczales have proposed aspect-oriented implementations of the GOF patterns in [13]. Their work provides solutions for remodularizing pattern instantiations at code level using AspectJ. However, it does not support on-demand remodularization. Achieving remodularization in AspectJ somewhat limits reuse: the code must be rewritten whenever developers change the programming language. Instead, a model would be just simply reused. This paper provides a new pattern that enables remodularization on demand, while supporting design by concerns at both modeling and programming level.

Finally, the On-Demand Remodularization pattern presented in this paper refines and formalizes the ODR pattern first introduced in [18].

5 Summary

This position paper describes the advantages and limitations of existing pattern approaches. It introduces a new approach to describing and instantiating patterns at low-level design and software architecture level, while providing a foundation for using, creating and discussing concern-oriented patterns. The ODR pattern enables remodularizations of existing and new software-intensive systems on-demand, while allowing one to achieve architectural *design by concerns*. Finally, we hope that the new techniques proposed in this paper contribute to advancing the state of the art in both software architecture and aspect-oriented software development.

6 Acknowledgements

Work presented in this paper has been partially supported by the Soft[net] program of the Swiss Federal Office for Professional Education and Technology, SNS 6452.1.

7 References

[1] O. Aldawud, T. Elrad and A. Bader. *A UML Profile for Aspect-Oriented Software Development*. Workshop on Aspect-Oriented Modeling with UML, AOSD'2003, Boston, USA. (http://lglwww.epfl.ch/workshops/aosd2003/).

[2] D. Alur, et al. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR; 2nd edition (2003).

[3] AspectJ Website: http://www.eclipse.org/aspectj.

[4] L. Bass, P. Clements, R. Kazman: *Software Architecture in Practice*. Addison-Wesley (1998).

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons Ltd (1996).

[6] J. Carey, et al. *SanFrancisco Design Patterns: Blueprints for Business Software*. Addison-Wesley (2000).

[7] S. Clarke and R. J. Walker. *Composition Patterns: An Approach to Designing Reusable Aspects*. Proceedings of the International Conference on Software Engineering - ICSE'2001 (May 2001).

[8] P. Clements, L. Northrop. *Software Product Lines — Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley (2002).

[9] M. Fowler. *Anaylysis Patterns: Reusable Object Models*. Addison-Wesley (1997).

[10] M. Fowler, et al. *Patterns of Enterprise Application Architecture*. Addison-Wesley (2003).

[11] E. Gamma, R. Helm, J. Vlissides, R. Johnson: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley (1995).

[12] J. D. Gradecki, N. Lesiecki. *Mastering AspectJ - Aspect-Oriented Programming in Java*. John Wiley Publishing (2003)

[13] J. Hannemann and G. Kiczales. *Design Pattern Implementation in Java and AspectJ*. Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 161-173, November 2002.

[14] HyperJ Website: http://researchweb.watson.ibm.com/hyperspace/HyperJ/HyperJ.htm

[15] Institute of Electrical and Electronics Engineers (IEEE) Standards Board. *Recommended Practice for Architectural Description of Software-Intensive Systems (ANSI/IEEE-Std-1471)*. September 2000.

[16] I. Jacobson, G. Booch and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley (1999).

[17] G. Kiczales, E. Hilsdale, J. Hugunin, K. Kersten, J. Palm and W. G. Griswold. *An Overview of AspectJ*. in Proc. of ECOOP'01 (Budapest, Hungary, June 2001), LNCS 2072, 327-252.

[18] M. M. Kandé. *A Concern-Oriented Approach to Software Architecture*. Thèse n° 2796, 2003, EPFL, Lausanne, Switzerland. http://ahdoc.epfl.ch/EPFL/theses/2003/2796/EPFL_TH2796.pdf.

[19] OMG. *Unified Modeling Language Specification version 1.4*. February 2001. OMG document formal/10-09-67 available from http://www.omg.org/technology/documents/formal/uml.htm

[20] H. Ossher and P. Tarr. *On the Need for On-Demand Remodularization*. ECOOP 2000 Workshop on Aspects and Dimensions of Concerns.

[21] D. C. Schmidt, M. Stal, H. Rohnert and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*.Wiley and Sons, 2000.

[22] D. Stein, S. Hanenberg, and R. Unland. *A UML-based Aspect-Oriented Design Notation For AspectJ*. In Proceedings of the 1st International Conference on Aspect-Oriented Software Development - AOSD'2002 (April 2002).

[23] P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. Proceedings of the International Conference on Software Engineering - ICSE'99 (May 1999).

[24] P. Tarr and H. Ossher. *Multi-Dimensional Separation of Concerns and The Hyperspace Approach*. In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer (January 2000).