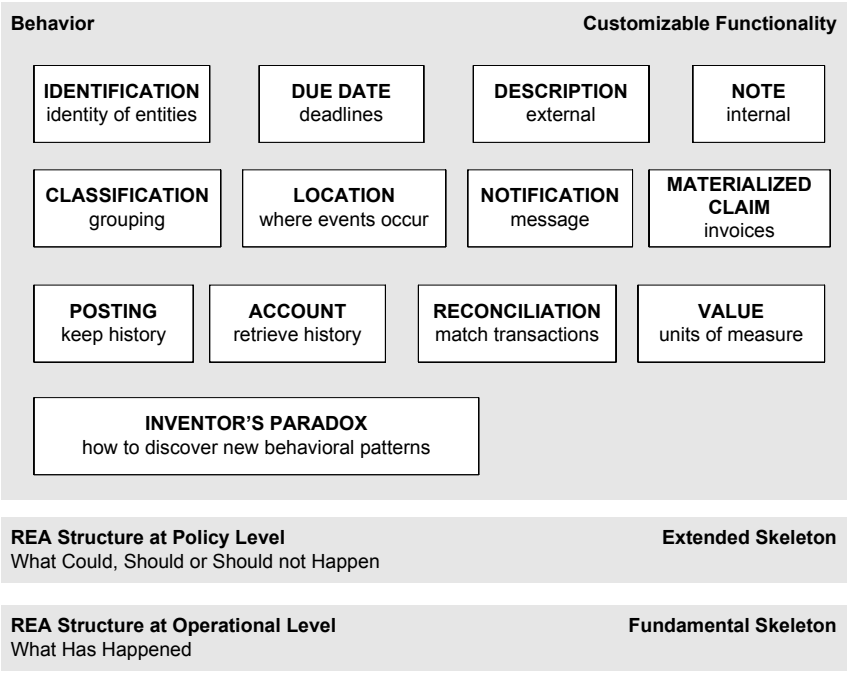
Part II Behavioral Patterns

The previous part, Structural Patterns, discussed the structure of a business application, which conforms to the laws of the business domain, consisting of REA entities and their relationships. To build a useful business application, this structure is only one of the things an application developer has to determine. Users of business applications usually require additional functionality, such as serial numbers, accounts, price calculations, and conversions between units of measure. This functionality is essential in some applications, but it might not be required in others. All depends on the users of a business application, actual configuration of an application, and the common practices in their businesses.

In this part, Behavioral Patterns, we describe how the REA model can be extended to support specific functionality that originates in user requirements.



4 Cross-Cutting Concerns

4.1 Behavior May Not Be Localizable Into REA Entities

Units of functionality that extend the REA model are usually not localizable into a single REA entity. An example is illustrated in Fig. 105. This example shows the economic resource *Vehicle*, which belongs to the *Vehicle Category*, and is used in the economic events *Trip*.

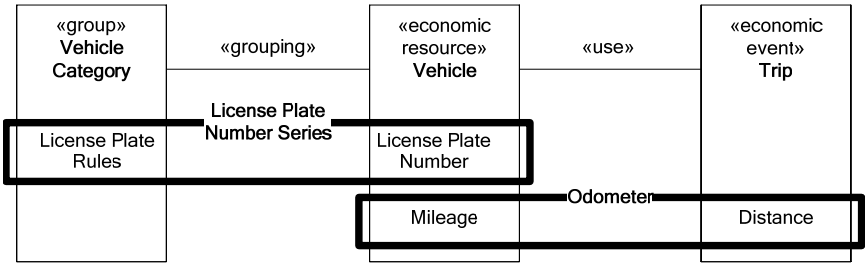


Fig. 105. Behavioral patterns often crosscut REA entities

A *License Plate Number* of a vehicle is an attribute of the economic resource *Vehicle*. The *License Plate Number* is usually not a random number. It is constructed using a *License Plate Rules*, which is a property of *Vehicle Category* (for example, numbers of police cars, military cars, and diplomatic cars are constructed using different rules than numbers of other cars). The property *License Plate Rules* contains rules specifying the uniqueness of the *License Plate Number*, its format, its dependency on previous numbers or other attributes, and so on. Therefore, the unit of functionality of a *License Plate Number Series* is present on two REA entities, the resource and the resource group, and the number is constructed by mutual collaboration between the part that resides on the resource and the part that resides on the group.

Likewise, a *Mileage* of a *Vehicle* is calculated as the aggregated number of the trip *Distances* the vehicle traveled. As *Trip* is an economic event, the *Odometer* is a unit of functionality present on two REA entities, the economic resource and the economic event.

It is still useful to think about a *License Plate Number Series,* and about an *Odometer* as single units of functionality, but these units span several REA entities.

We will use aspect-oriented programming as a conceptual framework and a convention of thought for modeling the crosscutting modules of functionality. Aspect-oriented programming is one of the mechanisms for describing the crosscutting features and manipulating them as modular units. Aspect-oriented programming is based on the ideas of Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, (Kiczales 1996). This group at the Palo Alto Research Center, a subsidiary of Xerox Corporation, developed a general purpose aspect-oriented language called AspectJ, an extension of the Java programming language with aspect-oriented features. Many other research centers have developed other aspect-oriented languages, both general purpose and specific to a certain domain.

At the end of Part II of this book we illustrate two ways of implementing the behavioral patterns, one in C# code, and the other using a model framework. Nevertheless, the behavioral patterns, as described in this book, can be also used without a specific implementation in mind, or implemented in another way.

To stay independent of any particular implementation, we call *Aspects* the crosscutting units of functionality, such as License Plate Number Series, and *Aspect Elements* the units that are present on REA entities, such as License Plate Rule, License Plate Number, Mileage, and Distance.

4.2 Framework-Based Approach

Aspect-oriented languages that are not framework-based, such as AspectJ, express the structure of a software application in the form of code in the programming language, and the crosscutting concerns, or aspects, are also expressed as code. During compilation, both the application code and the aspect code are combined together in a process called weaving; see Fig. 106.

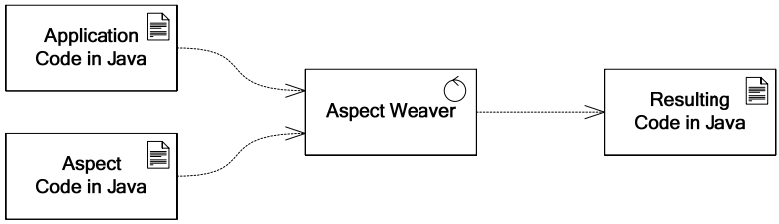


Fig. 106. Aspect-oriented programming at the code level (not framework-based)

Keeping in mind requirements such as extensibility and configurability, a disadvantage of such programming languages is that the code has to be weaved (which means recompiled) every time the functionality of an application (expressed as application code or aspect code) changes. The consequence is that upgrading an application is complicated and expensive.

Furthermore, since some or all functionality of an object is provided in the aspect code, it is impossible for the weaver to guarantee a system-wide quality for an application, because the weaver has no way of knowing what the aspect code does.

To satisfy the requirements for extensibility, configurability, and upgradeability, we use the framework approach to model and implement the aspects. Every aspect is represented at two levels of abstraction, the *Aspect type level* and the *Application model level*; see Fig. 107.

We will use a simplified version of the *IDENTIFICATION PATTERN* as an example. The *IDENTIFICATION PATTERN* encapsulates business logic for providing identity to REA entities, such as serial numbers and names. Details about the identification pattern are described in the *IDENTIFICATION PATTERN* chapter.

The *Aspect Type* level specifies the types of the aspects, and metadata that can be applied to the aspects in the application model. This level encapsulates the business logic of the aspect, and specifies the configuration properties, which can be set by application developers. In the example illustrated in Fig. 107, the *Identification Aspect* consists of two element types, *Identifier Type* and *Identifier Setup Type*. The cardinality of the composition indicates that instances of these types (i.e., *Identifier* and *Identifier Setup*) can be configured in the application model several times. These two elements are related by a one-to-many relationship, which indicates that for each configured identification aspect in the application model, for one *Identifier* there can be exactly one *Identifier Setup*, and for one *Identifier Setup* there can be one or more *Identifiers*.

The *Application Model* level specifies the runtime attributes that can be set by the users of business applications or automatically by the system. The application model level also specifies which aspects are configured on which REA entities; in Fig. 107 the REA entities are shown by dashed lines, to indicate that they are not part of the aspect. An REA entity of a type group can contain zero or more *Identifier Setup* aspect elements, and any REA entity can contain zero or more *Identifier* elements. For each *Identifier* instance at runtime, there is exactly one *Identifier Setup* instance; for each *Identifier Setup* instance at runtime, there can be zero or more *Identifier* instances.

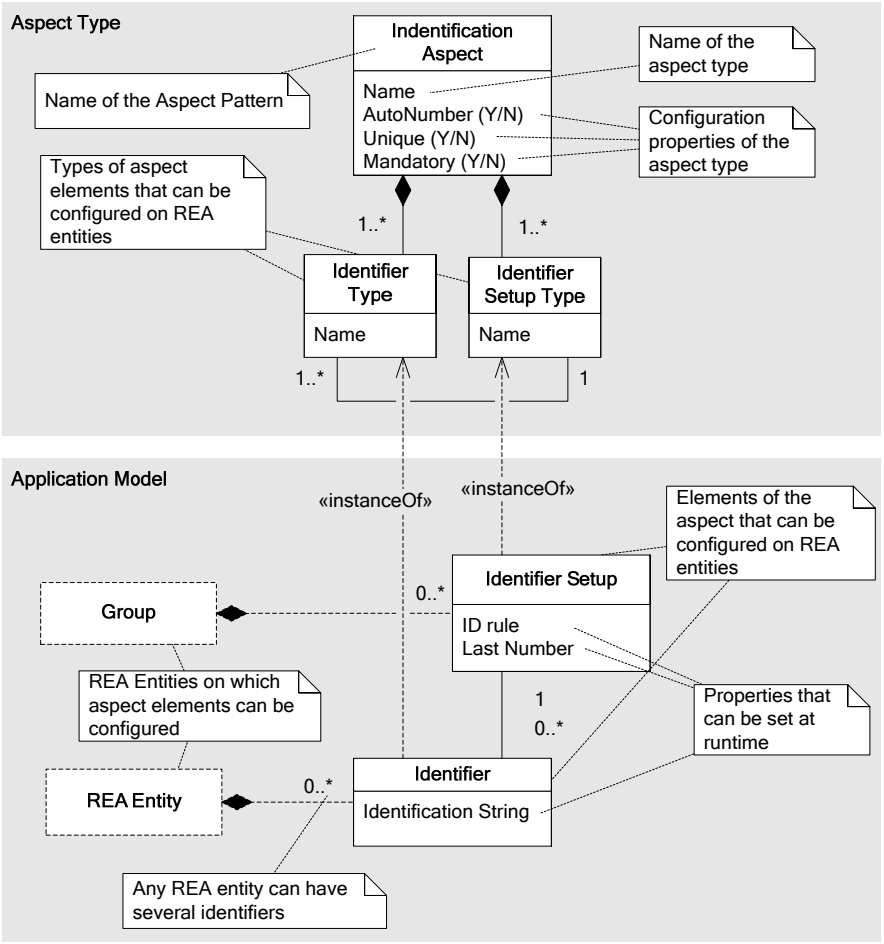


Fig. 107. Aspect pattern in the framework approach

In the examples illustrating application models with aspects, we use the notation in Fig. 108. The *Aspect Elements* are shown as rectangles with thick line; their runtime properties are shown similarly, as UML attributes. Properties of the aspect element types (i.e. the properties whose values are set at the aspect type level) are shown in the name compartment of the aspect. Values of the properties of the aspect type are shown as text close to the line connecting the aspect elements, similarly as UML attributes; for example, 'Mandatory = yes'.

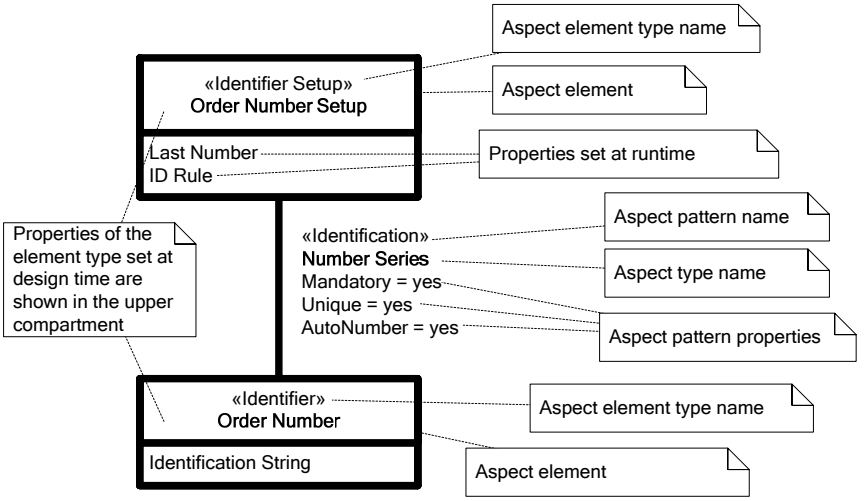


Fig. 108. Notation used for aspects in application models

Model in Fig. 109 illustrates a fragment of an REA model with two REA entities, a contract *Sales Order* and a group *Orders*. Application developers would like to implement sales order number on the *Sales Order* entity. They decide to configure the identification aspect on the *Sales Order*. The result is illustrated in Fig. 110.

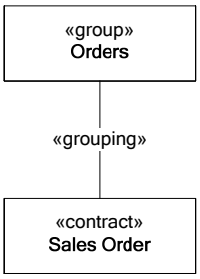


Fig. 109. Fragment of an REA model without aspects

The *Identification Aspect Pattern* has the name *Number Series*. The configuration parameters *Mandatory*, *Unique*, and *AutoNumber* are all set to *yes*. The *Identifier Setup* element is called *Order Number Setup*, and the *Identifier* is called *Order Number*.

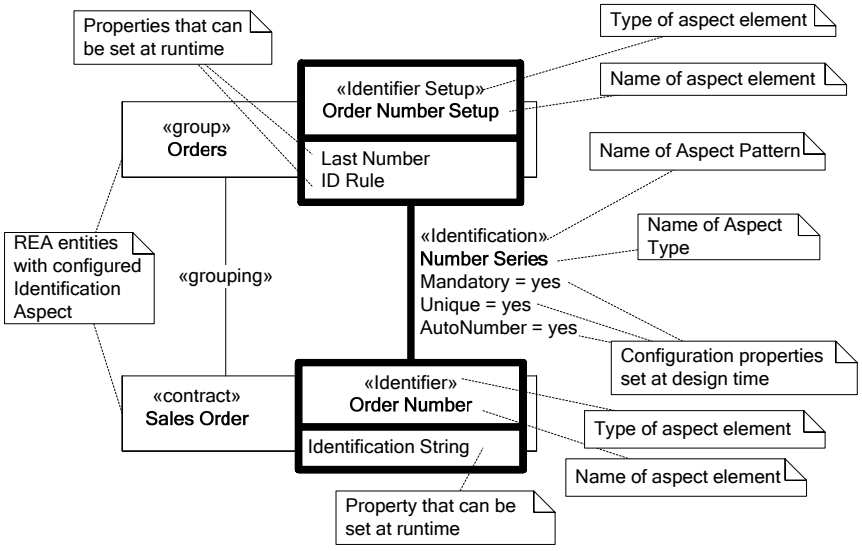


Fig. 110. Fragment of an REA model with identification aspect

Advantage of a system with explicitly modeled aspect types is that software business applications are much easier to configure, customize and upgrade than if the aspects were to be represented only as code in a programming language.

Configuration of software business applications using the aspect patterns is basically reduced to creating an REA model, setting the configuration parameters of aspects, and specifying which aspects are present on which objects. This can be done without writing any code in a programming language.

Software applications are easy to customize, as the customization task basically comprises setting up the configuration parameters of the aspects.

Furthermore, software applications are easy to upgrade, because all application logic is encapsulated in the elements at the aspect type level, and it can be extended independently of the configured application model. The upgrade of the software application basically means replacing the elements at the aspect type level with elements with upgraded functionality. The

framework developer designs the interface (the configuration properties and the corresponding behavior) that the elements at the aspect type level expose. If the upgraded elements support the old interface, the software applications can be upgraded without reweaving or recompiling the application.

Even if the upgraded elements are not backwards compatible (backward compatibility is considered anti-pattern by some practitioners), it is possible to write an upgrade script that modifies the configured applications to support the upgraded elements.

Quality of the software applications is easier to control, as all functionality of business applications is encapsulated in a framework, and is therefore tested by framework developers. The framework developers, who provide the elements at the aspect type level, have full control over what application developers may do with their aspect elements. In other words, providing application developers a domain-specific modeling language reduces the number of errors the application developers can make, compared to the situation in which the application developers write code in a general programming language.

4.3 There Is No Complete List of Behavioral Patterns

While with the structural patterns our aim was to find the minimal, yet complete set of abstractions covering the business domain, this is not possible with behavioral patterns. Users of business applications will always need new features, and behavioral patterns provide a mechanism to add new features to a business application without changing its fundamental structure.

There are behavioral patterns waiting to be discovered. This section describes the patterns we came across in building our business solutions, but it is not a complete list of all patterns that might be needed in any line of business. As the REA structural patterns define more or less a complete set of concepts, if application developers identify user requirements for new functionality, they would likely be either new behavioral patterns which crosscut the REA entities or features in a domain other than the business domain.

http://www.springer.com/978-3-540-30154-7

Model-Driven Design Using Business Patterns
Hruby, P.
2006, XVI, 368 p., Hardcover
ISBN: 978-3-540-30154-7