Package Diagram

Related terms:

Unified Modeling Language, Sequence Diagram, Activity Diagram, Block Definition Diagram, Internal Block Diagram, Model Organization, Package Hierarchy, Packageable Element, Requirement Diagram

View all Topics

Design

Carol Britton, Jill Doake, in A Student Guide to Object-Oriented Development, 2005

Dependencies.

Package diagrams allow us also to specify dependencies between packages. A dependency exists between packages if a change in one can affect the other. If a change in package A can affect package B, then package B depends on package A. A dependency is modelled by a dashed arrow going from the dependent package to the one it depends on, see Figure 9.2.

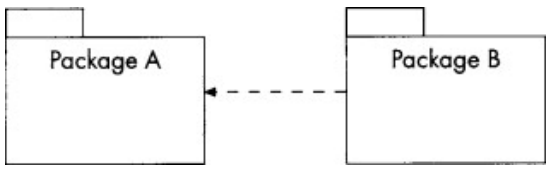

Figure 9.2. Packages and dependencies

A dependency exists between two packages, A and B, if a dependency exists between any class in package A and any class in package B. A dependency exists between two classes if, for example, they have a client–server relationship. In a client–server relationship, the client (i.e. the dependant) will be affected by a change to the server's interface. For example, let us suppose that the server changes an operation's signature from findBike(bike#) to findBicycle(bike#), if the client then sends a message that matched the old interface, e.g. bike[36].findBike(), it won't work.

In the example of the Wheels subsystems described above, several classes in the 'Hire Bike' package send messages to the Bike class, i.e. they use the services of the Bike

class. The Bike class is defined in the 'Manage Data' package, therefore the 'Hire Bike' package depends on 'Manage Data', see Figure 9.3.

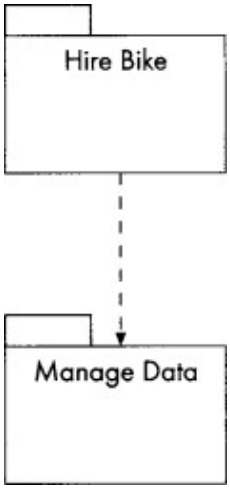

Figure 9.3. Packages and dependencies

> Read full chapter

Organizing the Model with Packages

Sanford Friedenthal, ... Rick Steiner, in A Practical Guide to SysML (Second Edition), 2012

6.3 Defining Packages Using a Package Diagram

SysML models are organized into a hierarchical tree of packages that are much like folders in a Windows directory structure. Packages are used to partition elements of the model into coherent units that can be subject to access control, model navigation, configuration management, and other considerations. The most significant kinds of packages used to organize models in SysML are models, packages, model libraries, and views.

A **package** is a container for other model elements. Any model element is contained in exactly one container, and when that container is deleted or copied, the model element it contains are deleted or copied along with it. This pattern of containment means that any SysML model is a tree hierarchy of model elements.

Model elements that can be contained in packages are called packageable elements and include blocks, activities, and value types, among others. Packages are themselves packageable elements, which allows packages to be hierarchically nested. The containment rules and other related characteristics, such as naming, of other kinds of packageable elements are described in the relevant chapters.

A **model** in SysML is a top-level package in a nested package hierarchy. In a package hierarchy, models may contain other models, packages, and views. The choice of model content and detail—for example, whether to have a hierarchy of models—is dependent on the methodology used. Typically, however, a model is understood to represent a complete description of a system or domain of interest for some purpose, as described in Chapter 2.

A model has a single primary hierarchy containing all elements, whose organizing principle is based on what is most suitable to meet the needs of the project. Views, which are described in Section 6.9, can be used to provide additional perspectives on the model using alternative organizing principles.

Often a package is constructed with the intent that its contents will be reused in many models. SysML contains the concept of a **model library**—a package that is designated to contain reusable elements. A model library is depicted as a package symbol with the keyword «modelLibrary» above the package name as shown in Figure 6.2 for *Components* and *Standard Definitions.* See Chapter 15, Section 15.2 for more details on model libraries.

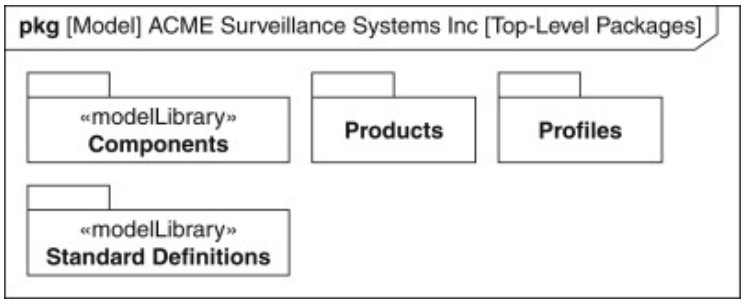

FIGURE 6.2. Package diagram for the surveillance system model.

The diagram content area of a package diagram shows packages and other packageable elements within the package represented by the frame. Packages are displayed using a folder symbol, where the package name and keywords can appear in the tab or the body of the symbol.

If a model appears on a package diagram, which may happen when there is a hierarchy of models, the standard folder symbol includes a triangle in the top right corner of the symbol's body.

The package diagram in Figure 6.2 shows the top-level packages within the corporate model of *ACME Surveillance Systems Inc.*, as specified in the diagram header. The user-defined diagram name for this diagram is *Top-Level Packages,* indicating that the purpose of this diagram is to show the top level of the model's package structure. In this example, the model contains separate package hierarchies for

- Standard off-the-shelf components
-

- Standard engineering definitions such as SI units—from the French *Système International d'Unités* (also known as International System of Units)
- The company's products

Any specific extensions required to support domain-specific notations and concepts (extensions to SysML, called profiles, are described in detail in Chapter 15)

Each package should contain packageable elements specific to the purpose of the package. These elements can then be represented as needed on different SysML diagrams including structure, behavior, parametric, and requirement diagrams, as described in later chapters in this part of the book.

> Read full chapter

Organizing the Model with Packages

Sanford Friedenthal, ... Rick Steiner, in Practical Guide to SysML, 2008

5.3 Defining Packages Using a Package Diagram

SysML models are organized into a hierarchical tree of packages that are much like folders in a Windows directory structure. Packages are used to partition elements of the model into coherent units that can be subject to access control, model navigation, configuration management, and other considerations. The most significant types of packages used to organize models in SysML are models, packages, model libraries, and views.

A **package** is a container for other model elements. Any model element is contained in exactly one container, and when that container is deleted or copied, the contained model element is deleted or copied along with it. This pattern of containment means that any SysML model is a tree hierarchy of model elements.

Model elements that can be contained in packages are called packageable elements and include blocks, activities, and value types, among others. Packages are themselves packageable elements, which allows packages to be hierarchically nested. The containment rules and other related characteristics of other packageable elements are described in the relevant chapters.

A **model** in SysML is a top-level package in a nested package hierarchy. In a package hierarchy, models may contain other models, packages, and views. The choice of model content and detail—for example, whether to have a hierarchy of models—is dependent on the methodology used. Typically, however, a model is understood to

represent a complete description of a system or subject area of interest for some purpose, as described in Chapter 2.

A model has a single primary hierarchy containing all elements whose organizing principle is based on what is most suitable to meet the needs of the project. Views, which are described in Section 5.9, can be used to provide additional perspectives on the model using alternative organizing principles.

Often a package is constructed with the intent that it will be reused in many models. SysML contains the concept of a **model library**—a package that is designated to contain reusable elements. A model library is depicted as a package symbol with the keyword «modelLibrary» above the package name as shown in Figure 5.2 for *Components* and *Standard Definitions*. See Chapter 14 for more details on model libraries.

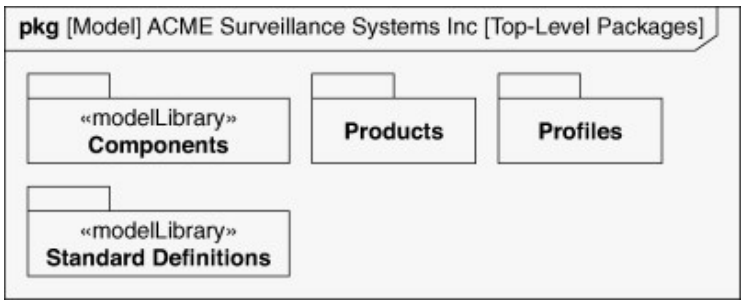

Figure 5.2. Package diagram for the surveillance system model.

Relationships, such as dependency and import relationships, can be established between packages and between the packageable elements within those packages. These relationships are described in Sections 5.7 and 5.8.

The diagram content area of a package diagram shows packages and other packageable elements within the package represented by the frame. Packages are displayed using a folder symbol, where the package name and keywords can appear in the tab or the body of the symbol. Where a model appears on a package diagram, which may happen where there is a hierarchy of models, the standard folder symbol includes a triangle in the top right corner of the symbol's body.

The package diagram in Figure 5.2 shows the top-level packages within the corporate model of *ACME Surveillance Systems Inc.*, as specified in the frame label of the diagram. The user-defined diagram name for this diagram is *Top-Level Packages*, indicating that the purpose of this diagram is to show the top level of the model's package structure. In this example, the model contains separate package hierarchies for

- Standard off-the-shelf components
- Standard engineering definitions such as SI units—from the French *Système International d'Unités* (also known as International System of Units)

- The company's products
- Any specific extensions required to support more domain-specific notations and concepts (extensions to SysML, called profiles, are described in detail in Chapter 14)

Each package should contain packageable elements specific to the purpose of the package. These elements can then be represented as needed on different SysML diagrams including structure, behavior, and requirement diagrams, as described in later chapters in this part of the book.

> Read full chapter

Organizing the Model with Packages

Sanford Friedenthal, ... Rick Steiner, in A Practical Guide to SysML (Third Edition), 2015

6.3 Defining Packages Using a Package Diagram

SysML models are organized into a hierarchical tree of packages that are much like folders in a computer directory structure. Packages are used to partition elements of the model into coherent units that can be subject to access control, model navigation, configuration management, and other considerations.

A **package** is a container for other model elements. It has a name and an optional **URI**, which uniquely identifies the package as a web-accessible resource, and is thus useful when packages are used widely within or between organizations. Any model element is contained in exactly one container, and when that container is deleted or copied, the model element it contains is deleted or copied along with it. This pattern of containment means that any SysML model is a tree hierarchy of model elements.

Model elements that can be contained in packages are called packageable elements and include blocks, activities, and value types, among others. Packages are themselves packageable elements, which allows packages to be hierarchically nested. The containment rules and other related characteristics of other kinds of packageable elements are described in the relevant chapters.

A **model** in SysML is a top-level package in a nested package hierarchy. In a package hierarchy, models may contain other models and packages. The choice of model content and detail—for example, whether to have a hierarchy of models—is dependent on the method used. Typically, however, a model is understood to represent

a complete description of a system or domain of interest for some purpose, as described in Chapter 2.

A model has a single primary hierarchy containing all elements. Its organizing principle is based on what is most suitable to meet the needs of the project.

Often a package is constructed with the intent that its contents will be reused in many models. SysML contains the concept of a **model library**—a package that is designated to contain reusable elements. A model library is depicted as a package symbol with the keyword «modelLibrary» above the package name, as shown in Figure 6.2 for *Components* and *Standard Definitions*. See Chapter 15, Section 15.3 for more details on model libraries.

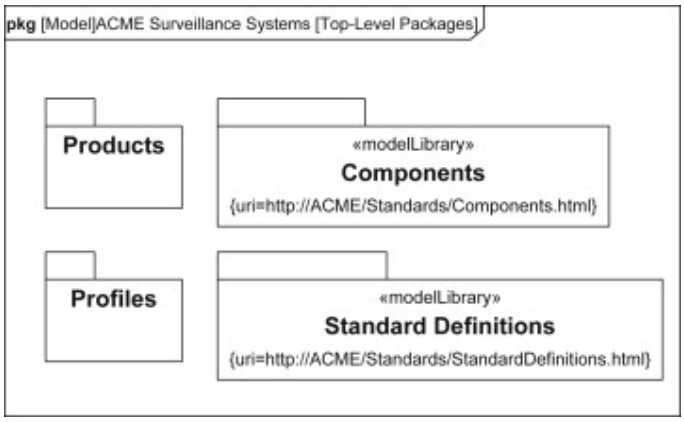

FIGURE 6.2. Package diagram for the surveillance system model.

The diagram content area of a package diagram shows packages and other packageable elements within the package designated by the frame. Packages are displayed using a folder symbol, where the package name and keywords can appear in the tab or the body of the symbol. The URI, if specified, appears in braces after the package name.

If a model appears on a package diagram, which may happen when there is a hierarchy of models, the standard folder symbol includes a triangle in the top right corner of the symbol's body.

The package diagram in Figure 6.2 shows the top-level packages within the corporate model of *ACME Surveillance Systems*, as specified in the diagram header. The user-defined diagram name for this diagram is *Top-Level Packages,* indicating that the purpose of this diagram is to show the top level of the model's package structure. In this example, the model contains separate package hierarchies for:

- The company's products;
- Standard off-the-shelf components;
- Standard engineering definitions such as SI units—from the French *Système International d'Unités* (also known as International System of Units); and

- Any specific extensions required to support domain-specific notations and concepts (extensions to SysML, called profiles, are described in detail in Chapter 15).

The *Components* and *Standard Definitions* packages both have URIs because they are widely used within *ACME Surveillance Systems* and therefore need to be uniquely identified and web accessible across company projects.

Each package should contain packageable elements consistent with the model organization approach. These elements can then be represented as needed on different SysML diagrams including structure, behavior, parametric, and requirement diagrams, as described in Chapter 3, Section 3.2 and in more detail in later chapters.

> Read full chapter

Packages and Namespaces

Michael Jesse Chonoles, in OCUP Certification Guide, 2018

8.1.2 Diagrams of Packages

In the above examples, we have illustrated stand-alone Packages. In UML, it is also possible to draw a diagram that depicts the same situation. We show the diagram approach in Fig. 8.5.

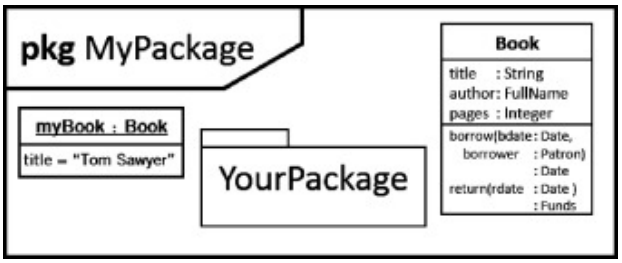

Figure 8.5. Package as Diagram.

This figure is a Package Diagram (because it shows Packages in the diagram) that shows the contents of the Package MyPackage. If we did not show the YourPackage member in the diagram, we might classify the figure as a Class Diagram3 that depicts the content of MyPackage Package. Remember that the diagram header "**pkg** MyPackage" indicates that the diagram is a Package Namespace whose name is MyPackage. The type of diagram is mainly determined by the preponderance of Elements.

> Read full chapter

Structuring Logical Layout of Software Design

Janis Osis, Uldis Donins, in Topological UML Modeling, 2017

10.2 Designing Packages

Initially packages are added to package diagram as subsystems from topological use case diagram which gets developed within Topological UML modeling behavior analysis and design activity. The contents of packages are added from the topological class diagram accordingly to the use cases in each system and the mappings between functional features and use cases. Thus, each package gets a set of classes that are responsible for particular subsystem. If needed the initial packages can be split up by grouping classes by their responsibilities. The output of this activity is package diagram structured according to subsystems and responsibilities of classes.

The developed package diagram in the context of enterprise data synchronization system development case study is given in Fig. 10.2, where one package added as topological use case diagram of enterprise data synchronization system (see Fig. 7.3) contains only one subsystem *Scheduler*. The graphical representation used in Fig. 10.2 shows package *Scheduler* without revealing its internal details.

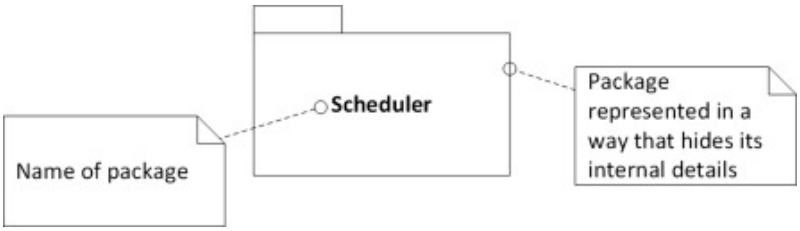

Figure 10.2. Initial package diagram of enterprise data synchronization system.

Another way of representing package is by revealing its details. According to UML, a package can contain any element, i.e., classes, interfaces, components, nodes, use cases, diagrams, and other packages grouped into it. Every element that is included in the package is defined within it. If we destroy the package, all the elements within it are destroyed as well. Fig. 10.3 shows package *Scheduler* revealing its internal details—classes and interfaces. The classes and interfaces are added by following mappings between topological use case diagram and topological class diagram. In the context of enterprise data synchronization system development case study all the classes and interfaces are added from topological class diagram developed during structure analysis and design activity (see Fig. 8.12) to the package *Scheduler*.

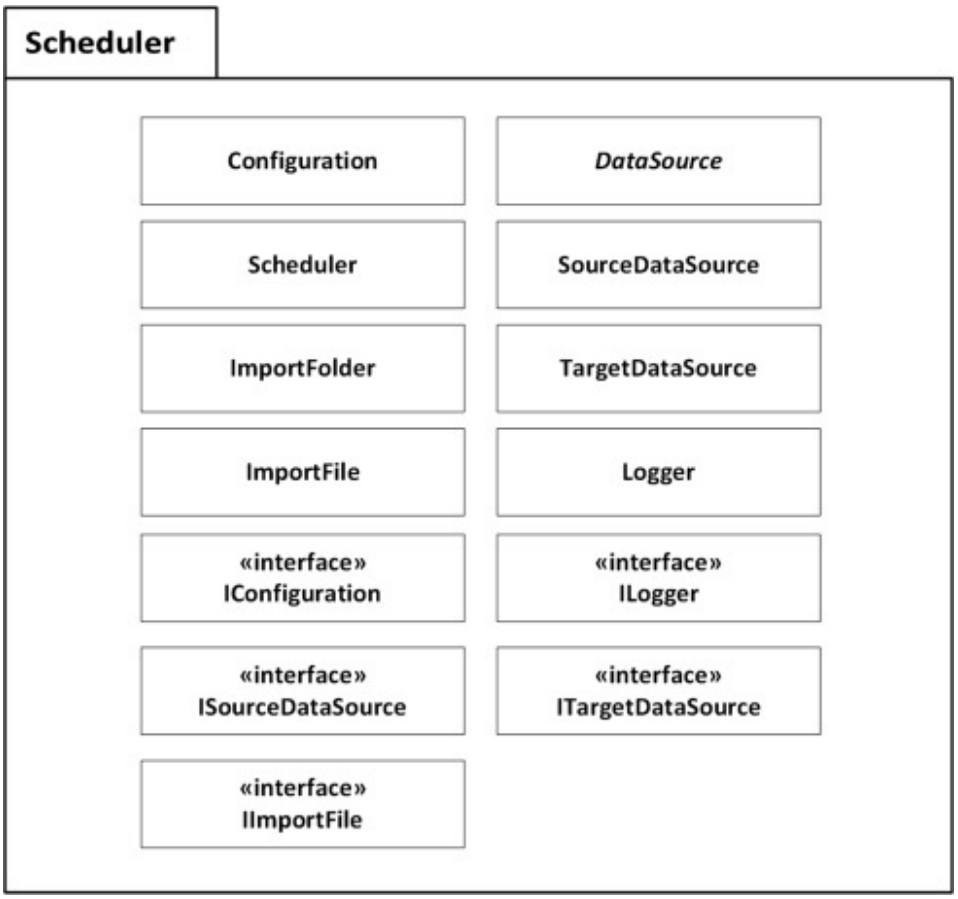

Figure 10.3. Package diagram showing internal details.

As package *Scheduler* contains classes and interfaces, we can make groupings of similar elements by adding additional packages, e.g., by grouping all interfaces in a special package with name *Interfaces* (see Fig. 10.4).

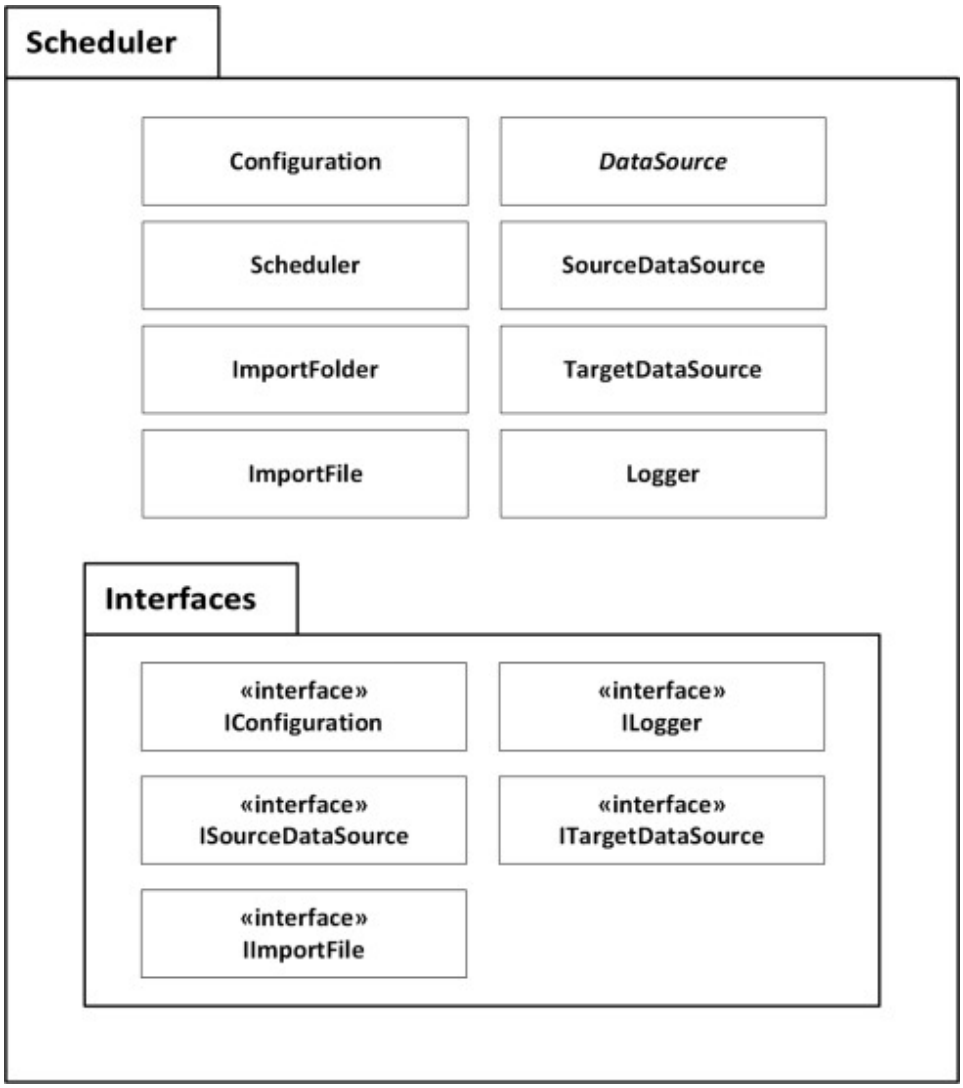

Figure 10.4. Package diagram with additional package for interfaces.

> Read full chapter

Logic synthesis in a nutshell

Jie-Hong (Roland) Jiang, Srinivas Devadas, in Electronic Design Automation, 2009

6.2.6 Boolean reasoning engines

Among the introduced data structures, BDD packages and SAT solvers are the most widely used Boolean reasoning engines. They are extensively used in various *symbolic*, or called *implicit*, algorithms, such as image computation, don't care computation, state reachability analysis, and so on. Any Boolean reasoning engine can be more or less used in developing symbolic algorithms. In the sequel when a computational task is expressed in terms of a QBF, we should be aware that its computation is already achievable by Boolean manipulation using a BDD package.

Although BDD-based algorithms and symbolic algorithms were once almost synonymous in the 1990s, recently other data structures were developed as alternatives to BDDs. Due to the capacity limit of BDDs, more and more symbolic algorithms are based on other data structures. Notably, Boolean reasoning engines using SAT and AIGs, for instance, are gaining in popularity in hardware synthesis and verification. Moreover, hybrid Boolean reasoning engines combining complementary data structures may become important tools. In fact, combinational equivalence checking of multi-million gate designs has been demonstrated in an industrial setting through such hybrid solvers combining BDD and AIG [Kuehlmann 1997].

> Read full chapter

Getting Started with SysML

Sanford Friedenthal, ... Rick Steiner, in A Practical Guide to SysML (Third Edition), 2015

3.7 Questions

1. What are five aspects of a system that SysML can represent?
2. What is a package diagram used for?
3. What is a requirement diagram used for?
4. What is an activity diagram used for?
5. What is the block definition diagram used for?
6. What is an internal block diagram used for?
7. What is a parametric diagram used for?
8. What are some of the common elements of the user interface of a typical SysML modeling tool?
9. Which part of the user interface presents a hierarchical view of the model elements contained in the model?
10. What is the purpose of applying an MBSE method?
11. What are the primary activities of the simplified MBSE method?

Discussion Topics

What are some factors that contribute to the challenges of learning SysML and MBSE, and how do they relate to the general challenges of learning systems engineering?

> Read full chapter

Adjusting Unified Modeling Language

Janis Osis, Uldis Donins, in Topological UML Modeling, 2017

3.5 Profile Specification Template

As shows the specification of OMG SysML [80] and SoaML [76] the best practice for UML profile specification is to use the same structure as used for UML specification, thus if the reader is familiar with UML specification it is easier to read and understand the specification of specific UML profile. UML specification is created by keeping in mind following aspects [77]: correctness, precision, conciseness, consistency, and understandability.

The profile specification should start with profile diagram showing the referenced metamodel and how the profile extends it. After profile diagram, one or more package diagrams should be provided showing the packages of which the profile consists. UML elements within its metamodel and specification also are grouped into packages. At this point it is advised to reuse the package specification style used in UML specification. Each package and each class in the UML specification (both infrastructure [77] and superstructure [78] specifications) has following structure:

- *Package*—this clause provides information for each package and each class in the UML metamodel or profile. Each package specification contains one or more of the following subclauses (*Classes*, *Diagrams*, and *Instance model*-).•*Classes*—contains a list of the classes specifying all the constructs defined in package. This subclause begins with one diagram or several diagrams depicting the abstract syntax of the constructs (i.e., the classes and their relationships) in the package, together with some of the well-formedness requirements (multiplicity and ordering). Then follows a specification of each class in alphabetic order. Each class specification has following subclauses:•-*Description*—an informal definition of the metaclass specifying the construct in UML.•*Attributes*—list of all attributes for metaclass. Attributes are given together with a short explanation.•*Associations*—list of all member ends of associations connected to this class (associations are listed in the same way as attributes).•*Constraints*—well-formedness rules of the metaclass. These rules specify constraints over attributes and associations defined in the metamodel.

Mostly they are defined by using OCL expressions together with an informal explanation of the expression.•*Additional operations (optional)*—contains any additional operations on the class which are needed for the OCL expressions.-•*Semantics*—the meaning of a well-formed construct is defined using natural language (can include formal definition of construct's semantics).•*Semantic variation points (optional)*—objective of a semantic variation point is to enable specialization of that part of UML for a particular situation or domain.•*Notation*—presents the notation of the construct (i.e., class).•*Presentation options (optional)*—if there are different ways to show the construct, these ways are described in this subclause.•*Style guidelines (optional)*—describes non-normative conventions that are used in representing some part of a model.•*Examples (optional)*—examples of how the construct is to be depicted.•*Rationale (optional)*—if there is a reason why a construct is defined like it is, or why its notation is defined as it is, then it is given in this subclause.

- *Diagrams*—this subclause is included into specification to describe specific kind of diagram, if this diagram uses the constructs that are defined in this package.

Instance model—shows an example of applying constructs defined in this package.

> Read full chapter

Addressing Usability Requirements in Mobile Software Development

Rafael Capilla, ... Hui Lin, in Relating System Quality and Software Architecture, 2014

12.5.2 Impact on the software architecture

This section describes the changes we performed on the software architecture of the M-ticket system to support the usability requirements and how these impacted the existing functionality. Figure 12.5 shows the new package diagram of the modified software architecture. The three layers of the design are as follows: (a) *the presentation layer* containing the entry screen to the Android applications, (b) *the* business logic layer of the M-ticket application containing the functionality of the app and the usability mechanisms introduced, and (c) *the middleware and data access layer* supporting the connection to the GPS and images captured by the phone that are sent to the Web server database.

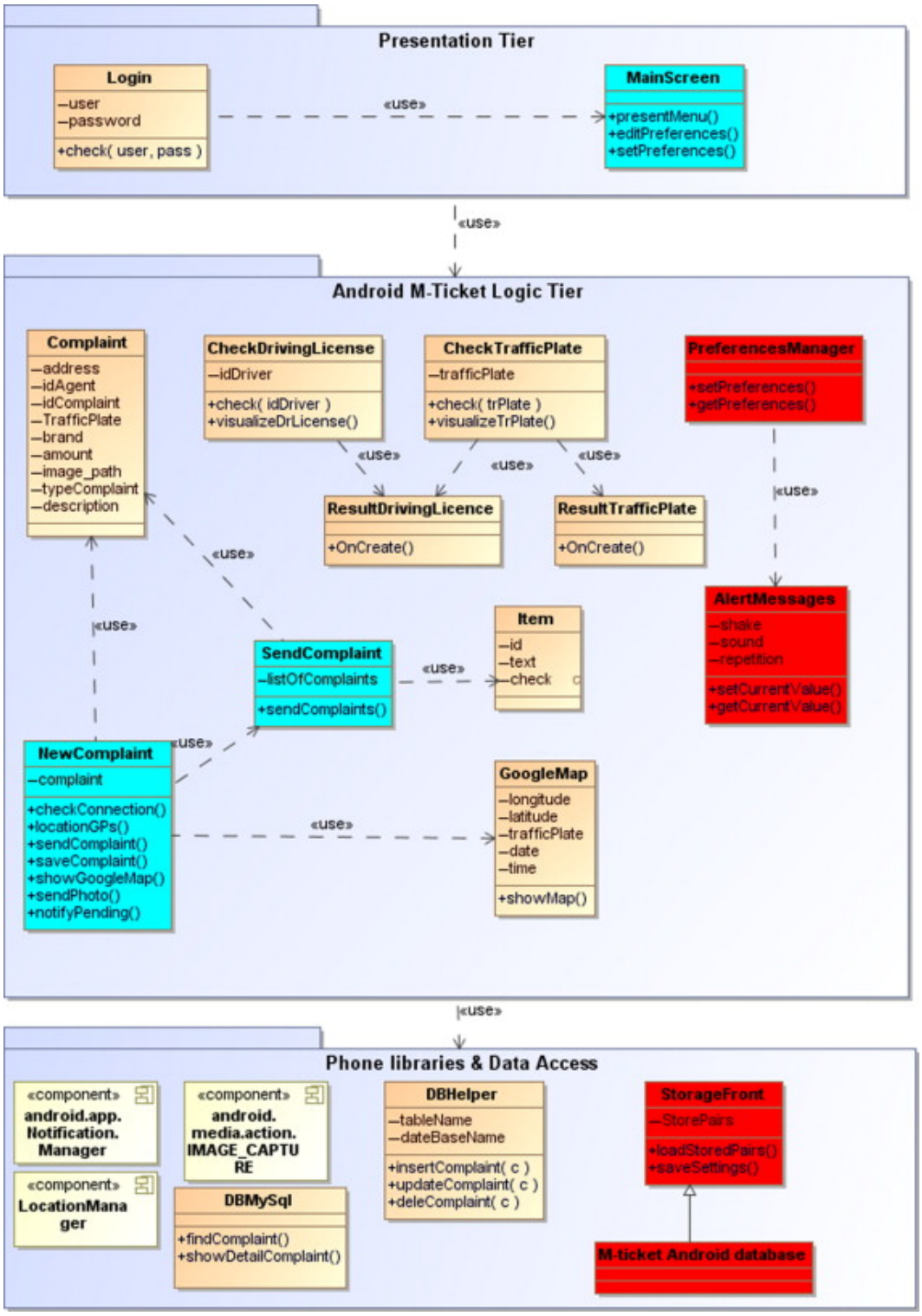

Figure 12.5. Modified software architecture of the M-ticket application including the classes for the two usability mechanisms.

The three-layered architecture of the M-ticket application depicts the new classes (red color) introduced in the design and the classes that changed (blue color). It also handles the two usability mechanisms (i.e., SSF and User Preferences) introduced in the system, which we explain below:

System status feedback: As we can see in Figure 12.5, in the Android application we modified two of the existing classes (NewComplaint, SendComplaint) in application logic tier in order to support the SSF mechanism. The NewComplaint class allows the police officer to create a new traffic ticket using the Complaint class shown in Figure 12.5. The functionality of the M-ticket app also implements the location of the vehicle using the GPS of the mobile phone, then it sends the form, the location, and a picture of the vehicle to a remote server. The changes introduced by the "Status Feedback" mechanism affect the notifications sent to the policeman using the mobile phone. In this way, we used the android.app.NotificationManager Android library to warn the user about events that may happen, such as a complaint already sent or no GPS signal. In addition, the changes to the SendComplaint class refer to the list of pending complaints stored in the mobile phone before they are sent to the server. In those cases of no pending complaints stored in the phone, the notifications will be removed.

User Preferences: Changing the user preference of the alert messages supported by the status feedback mechanism led us to introduce new classes (PreferenceManager, AlertMessages, StorageFront, and M-ticket Android database classes in Figure 12.5) in the architecture. However, this usability mechanism affected the functionality of all layers in the architecture. In the presentation layer, the MainScreen class, which acts as entry point of the mobile application once the user has logged onto the system, was modified to incorporate specific methods to set and edit the preferences of the alert messages. In the logic tier, we added two new classes, PreferenceManager and AlertMessages, which handle the specific preferences (i.e., shake, sound, and repetition) of each alert message. Finally, the implementation of the classes supporting this usability mechanism require a new class, StorageFront, located in the data access layer to store the user preferences, As we can see, there is another class in that layer, M-ticket Android database, which represents where the user preferences are stored. We added this class for the sake of clarity for designers, but in our system the storage of the user preferences data is located in a specific database of the M-ticket application.

In order to provide a better understanding of the classes we added and changed in the original architecture of the M-ticket app when usability was introduced, we describe in Tables 12.3 and 12.4 the association between the generic components of each usability mechanism and the classes that implement such functionality in our system accordingly to the architecture of Figure 12.5.

Table 12.3. Mapping Between the Classes of the SSF Usability Mechanism and Those Implemented in the Architecture of the M-ticket Application

Usability Mechanism	Generic Component	Classes in the M-ticket Architecture	M-ticket Architectural Responsibility
System Status Feedback	Display Status	New Complaint	

		This class displays the status to the user
Status Manager	N/A	We don't need this functionality as we only support one type of status
StatusConcrete Status	New Complaint	This class checks if there are pending complaints stored in the phone and updates the status when the complaints are sent to the server
Domain	NotificationManager(Android)	This library performs low-level operations when the status changes and assigns an ID for the status. In case of a loss in the connection between the phone and the server, the *New Complaint* class will inform the *Notification Manager* with an ID, which will be used by the *Send Complaint* class when the notification need to be removed

Table 12.4. Mapping Between the Classes of the User Preferences Usability Mechanism and Those Implemented in the Architecture of the M-ticket Application

Usability Mechanism	**Generic Component**	**Classes in the M-ticket Architecture**	**M-ticket Architectural Responsibility**
User Preferences	User	Main Screen	Users can configure the options of their alert messages using the Main Screen interface
Preferences Manager	Preferences Manager		It handles the preferences set by the user
Preference	Alert Messages		The alert message is the configurable preference supported by M-ticket
Group	N/A		Not supported
Settings Manager	N/A		Not supported
Setting	N/A		Not supported
Storage Front	Storage Front		It acts as an intermediate class to store the preferences edited by the user
Mobile Phone Database	M-ticket database		This class relates the Storage Front with the access to the M-ticket database where preferences are stored

The mapping between the generic components of each usability mechanism and the concrete classes in the M-ticket application described in Tables 12.3 and 12.4 guide software designers to introduce the concrete architectural responsibilities. Hence, software architects can use these mappings to determine the concrete responsibilities of new and existing classes in their application for supporting a particular usability mechanism. In our example, only one class is assigned to one component, but in more complex mechanisms, several classes can be assigned

to the same component. However, we do not suggest guidelines for coding the usability mechanisms because these may depend on the current functionality of the application and the code in which the usability feature will be added.

> Read full chapter