CORSO IS1 - 2002/03
Abstract Design Specification
Version 1.0

Gianna Reggio
DISI-Università di Genova, Italy
reggio@disi.unige.it

0.1 Introduction

Assume to have to develop a software system System.

The considered development method requires to produce the following models:

Problem Domain Model (PDM) a description of the domain of the problem that the System must solve, that is the part of the real world that concerns the System, or the part of the real world where the System will operate, but without any reference to the System itself. That may be summarized by the slogan "before the System". `[not considered for the MSMILE case for simplicity]`

Requirement Specification a model of the System making precise which problem it will solve (which are its goals), showing how it will be perceived by its users and which entities will have to cooperate with it to reach its scope.

Abstract Design Specification (or model-driven design) a description of the structure and of the behaviour of the System as **conceived** by the developers, but without making any assumption on the technologies that will be used for its realization.

Concrete Design Specification (or technology-driven design) a description of the structure and the behaviour of the System as it will be realized by the developers by using some particular technologies, which are explicitly mentioned; clearly, it must be a "specialization" of the Abstract Design Specification, produced in the previous phase.

Below, we list some examples of "technologies" to be used in a Specialized Design.

- Programming languages
- Middlewares (i.e., software encapsulation of software-hardware solutions for the coordination of the parts in a distributed system)
- Operating systems
- Hardware

Typically, a Specialized Design

- uses Java types (as Stream and Byte);
- uses only synchronous communications, or only event raising and detection to coordinate the behaviour of the active components of System, also when such choice is neither appropriate nor natural;
- proposes a fully centralized solution for a System that shows a very natural distributed architecture (due to constraints on the available hardware).

For example, after giving its Abstract Design Specification one can decide to realize System by using Java and the J2EE technology, or the Jini technology JavaSpaces, or by distributing it over Internet by coordinating the various parts by means of sockets.

Usually, the specification of a Specialized Design is a model written using a variant of UML, offering ways to denote the use of the chosen technologies. For example, it may offer some stereotypes for the datatypes of the chosen programming language or for the sockets.

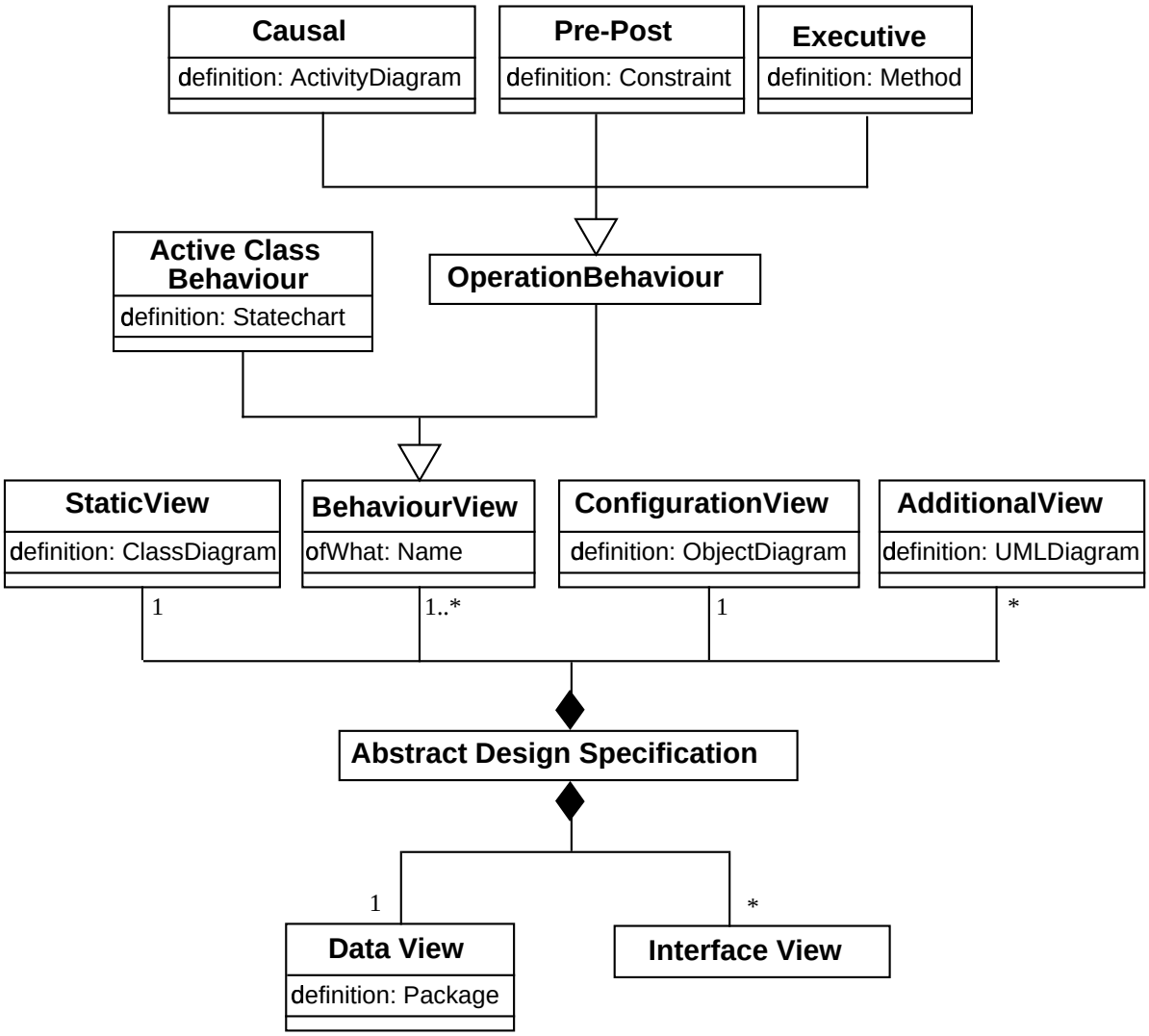

Figure 1: The Structure of an Abstract Design Specification

0.2 Abstract Design Specification

An Abstract Design Specification consists of different views of various kinds, which are listed below, of the designed System.

Data View describes the datatypes used by the entities composing the System.

Static View introduces the types of the entities composing the System.

Behaviour View describes the behaviour of a part of the System, which has been introduced by the Static View.

Configuration View describes the entities composing (a generic configuration of the) the System, by stating of which type and how many they are.

Additional View highlights some aspects concerning how some entities of the System interact among them to accomplish some particular task. These views are optional and intended just for documentation; they should not add any information not already present in the other views.

Interface View describes the interface of the System towards the external world (e.g., a GUI).

0.2.1 Data View

The Data View is a UML package including a class diagram, where all classes are datatypes (UML stereotype <<datatype>>) and where the only relationships among classes are specialization and aggregation/composition. A UML datatype is a classifier whose instances are pure values, i.e., they have no identity and their state cannot be changed; thus its operations are all pure functions.

The Data View defines all datatypes used by the entities composing the System.

In the case that we do not need to define any datatype, because the System entities use only the UML basic types, this view is trivial, and may be dropped.

0.2.2 Static View

The Static View is a UML class diagram importing the package Data View. It describes the types of the entities composing the System together with their interfaces. Technically, it consists of a class diagram where

- each class is depicted with its operations (its interface) and attributes (which must all be private, representing its encapsulated internal state) and has one of the following stereotypes

  <<**boundary**>> stereotype of active class. A boundary entity is a System entity that takes care of the interaction with some context entities.

  A boundary receives messages from the context entities, analyses them and afterwards either sends back an immediate answer (e.g., if the received message contains an error) or interacts with the executors and the stores as required by the message. Conversely, it forwards messages received by the executors to the context entities in the ways requested by their nature (for example, if the communication is unreliable, the boundary will take care of ensuring the reception of the message, e.g., by following the alternating bit protocol).

  <<**store**>> stereotype of passive class. A store is a System entity that contains persistent data (e.g., a database).

  <<**executor**>> stereotype of active class. An executor is a System entity that performs some core System activities as result of information received from the context thorough the boundaries; clearly during such activities it may collaborate with other executors, with stores, and with boundaries.

  <<**context**>> stereotype of active class. The elements of a <<context>> class are the context entities, which have been introduced in the Context View of the Requirement Specification. They cannot have attributes, and there should be one of them for each class of stereotype <<SU>> or <<SP>> in the Context View.[1]

- each association has the stereotype <<communication>>.

  A communication association from class C to class C' (visually depicted as an oriented arrow from C to C' without any name) means that an instance of class C interacts with some instances of class C' by calling their operations. The multiplicity assertions state with how many instances it will interact.

  The communication association should respect the following requirements

  - no communication association may leave a <<store>> class, because stores are fully passive and should be used only by the executors and the boundaries (if you need that a store communicates with another one, it just means that you have a structured store);
  - no communication association may go from an <<boundary>> class to another <<boundary>> (boundaries must have only a limited execution capability);
  - <<context>> classes may communicate only with <<boundary>> classes;

[1] Here we do not distinguish the context entities in service providers and service users, because this distinction is not relevant to design the system.

- any class in the diagram must be connected by a communication chain with at least a context class (otherwise it means that a part of the system is isolated and thus useless).

The Static View should contain at least one class with the stereotype <<context>>, whereas there are no constraints on the number of classes of the other kinds.

You should avoid to introduce trivial classes in the design of your System. A trivial <<boundary>> class is the one describing entities just passing data from <<context>> to either <<executor>> or <<store>> entities. In such case a <<context>> can communicate with entities that are not <<boundary>>.

A trivial <<executor>> entity is, e.g., the one executing very elementary operations over some data (e.g., a numerical computation), and a trivial <<store>> entity is, e.g., the one storing a private data (i.e., a data that is accessed by only one System entity).

0.2.3 Behaviour View

A Behaviour View describes the behaviour of a part of the System; precisely one of the following:

- an active class (either an <<boundary>> or an <<executor>>) appearing in the Static View; in this case its behaviour is defined by a statechart;
- an operation of a passive class (a <<store>>) appearing in the Static View; in this case its behaviour may be defined by
  - a UML method definition (that is a program written using the UML actions)[2];
  - a constraint of the form pre-post. This form may be used only if the algorithm for computing the specified operation is easy to find and does not require to interact with many other objects. It is ok for an operation finding the maximum in a set of numbers, not for an operation polloing all the connected clients;
  - an activity diagram. An activity diagram helps defining the behaviour of an operation at a rather abstract level; in particular the fork construct is used to define activities whose ordering is irrelevant.

It is mandatory to define the behaviour of any active class, except the <<context>> ones, and of any nontrivial operation of the passive classes (trivial operation are, for example, those getting and setting the attributes). The behaviour of the context entities cannot be defined, because they are not one of the aims of the design. Some information on their behaviour may be found in the Context View of the Requirement Specification.

0.2.4 Configuration View

A Configuration View is an object diagram whose objects are instances of the classes present in the Static View) describing a generic configuration of the System. The links of the communication associations show how such instances cooperate among them.

The multiplicity constraints on the context entities must be the same reported in the Context View of the Requirement Specification.

0.2.5 Additional View

An Additional View consists of a UML diagram (either sequence or collaboration or activity). It must be consistent w.r.t. the other views; this means precisely that the object roles in a sequence/collaboration are instances present in the Configuration View, the messages are operations described in the Static View and the used data follows the description given by the Data View; whereas for an activity diagram the actions and conditions must refer to the objects present in the Configuration View. Moreover, the described behaviour must be consistent with that presented by the various Use Case Behaviour View.

[2] Here we do not allow to use statements of some particular programming language, to guarantee the abstractness of the design.

The Additional Views are completely optional, in the sense that they do not influence the design of the System; their role is just to document some aspects of the System by presenting them in a more appealing way.

We think however, that trying to produce some Additional Views may help the developers to check the design or to understand the behaviour of some parts of the System.

Some suggested Additional Views are

- those corresponding to the various use case views presented in the Requirement Specification, as sequence diagrams for Use Case Interaction View, and activity diagrams for Use Case Causal View (see next section);
- those corresponding to additional Use Case Behaviour View for some operations;
- those introduced by the designer to help clarify the behaviour of the System in some cases before giving the related Behaviour Views. For example, when you define a protocol to regulate the interaction between two System entities, first give a Sequence Diagram showing an execution of such protocol, then define the behaviour of the involved entities.

0.2.6 Interface View

An Interface View is associated with a <<boundary>> class present in the Static View and describe the interface of the instances of such classe towards the context entities with whom interact. This view is optional; however it should be present for each boundary class interacting with human context entities.

There is no specific request on the format of an Interface View. Some possible ways to present an Interface View are

- a free visual presentation of a GUI,
- a UML presentation of a GUI using perhaps appropriate stereotypes, such as button, check box, and menú,
- a definition of some textual line commands.

The only relevant point is to establish the connection between the eleemnts of the interface and the operation of the boundary class and of the context entity communicating with it.

0.3 Checking the Quality of an Abstract Design Specification

Here we list some checks to perform on the produced Abstract Design Specification to detect problems.

Minimality The following checks avoid to define useless parts of the design.

- All datatypes defined in the Data View are used at least once, and all their operations are called at least once.
- All operations of a <<store>> /<<executor>> /<<boundary>> class are called at least once.
- All classes of the Static View have at least one instance in the Configuration View.

If one of the above checks fails, it is easy to solve the problem by removing some part of the design.

- All operations of a <<context>> are called at least once by an <<boundary>>.

In this case we can have also a design error (some requirement is not fully implemented) or some redundancy in the requirements.

Soundness

- The assumptions on the communications among the entities composing the System as stated by the <<communication>> association must be respected.

Correctness The following checks help see if all requirements have been correctly realized.

- The internal abstract state of System, as specified by the Internal View of the Requirement Specification, must be recoverable from the generic configuration shown by the Configuration View. This means that the state of each object and each link in the Internal View should correspond to a proper combination of states of the objects present in the Configuration View.
- For any operation in the interface of a context entity as presented in the Requirement Specification, either it is present in its interface in the corresponding class in the Abstract Design Specification or any of its calls is realized by a sequence of calls of other operations between the same context entity and some interfaces. Such sequence of calls should be presented by means of a Sequence Diagram.
- For any operation in the System class as presented in the Requirement Specification, either it corresponds to an operation of a <<boundary>> class or it is realized by a sequence of calls of other operations between a context entity and some boundaries. Such sequence of calls should be presented by means of a Sequence Diagram.
- Give an Additional View for each Use Case Interaction View or Use Case Causal View of the Requirement Specification. Check if they "correspond" taking advantage of the partial correspondences established by the above points.