**System modeling** is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).

**Models are used** during the requirements engineering process to help derive the requirements for a system, during the design process to describe the system to engineers implementing the system and after implementation to document the system’s structure and operation. You may develop models of both the existing system and the system to be developed.

**The most important aspect of a system model is that it leaves out detail.** A model is an abstraction of the system being studied rather than an alternative representation of that system. Ideally, a representation of a system should maintain all the information about the entity being represented. An abstraction deliberately simplifies and picks out the most salient characteristics.

---------------------------------------------------------------------------------------------------------------------------------------

You may develop *different models* to represent the system from different perspectives.

For example:

1. An external perspective, where you model the **context** or environment of the system.
2. An interaction perspective where you model the **interactions** between a system and its environment or between the components of a system.
3. A structural perspective, where you model the organization of a system or the **structure** of the data that is processed by the system.

4. A behavioral perspective, where you model the dynamic **behavior** of the system

and how it responds to events.

---------------------------------------------------------------------------------------------------------------------------------------

The detail and rigor of a model depends on how you intend to use it. There are three ways in which graphical models are commonly used:

1. As a means of facilitating discussion about an existing or proposed system.

2. As a way of documenting an existing system.

3. As a detailed system description that can be used to generate a system implementation.

In the first case, the purpose of the model is to stimulate the discussion amongst the software engineers involved in developing the system. The models may be incomplete (so long as they cover the key points of the discussion) and they may use the modeling notation informally.

When models are used as documentation, they do not have to be complete as you may only wish to develop models for some parts of a system. However, these models have to be correct—they should use the notation correctly and be an accurate description of the system.

In the third case, where models are used as part of a model-based development process, the system models have to be both complete and correct. The reason for this is that they are used as a basis for generating the source code of the system. Therefore, you have to be very careful not to confuse similar symbols, such as stick and block arrowheads, that have different meanings.

---------------------------------------------------------------------------------------------------------------------------------------

**Context models**

At an early stage in the specification of a system, you should decide on the system boundaries. This involves working with system stakeholders to decide what functionality should be included in the system and what is provided by the system’s environment. You may decide that automated support for some business processes should be implemented but others should be manual processes or supported by different systems. You should look at possible overlaps in functionality with existing

systems and decide where new functionality should be implemented. These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design.

---------------------------------------------------------------------------------------------------------------------------------------

**Interaction models**

All systems involve interaction of some kind. This can be user interaction, which involves user inputs and outputs, interaction between the system being developed and other systems or interaction between the components of the system. Modeling user interaction is important as it helps to identify user requirements. Modeling system to system interaction highlights the communication problems that may arise.

Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability. In this section, I cover two related approaches to interaction modeling:

1. **Use case modeling**, which is mostly used to model interactions between a system and external actors (users or other systems).

2. **Sequence diagrams**, which are used to model interactions between system components, although external agents may also be included.

---------------------------------------------------------------------------------------------------------------------------------------

**Structural models**

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be static models, which show the structure of the system design or dynamic models, which show the organization of the system when it is executing. These are not the same things—the dynamic organization of a system as a set of interacting threads may be very different from a static model of the system components.

Class diagrams

Class diagrams are used when developing an object-oriented system model to show

the classes in a system and the associations between these classes. Loosely, an object

class can be thought of as a general definition of one kind of system object. An association

is a link between classes that indicates that there is a relationship between these classes. Consequently, each class may have to have some knowledge of its associated class.

When you are developing models during the early stages of the software engineering

process, objects represent something in the real world, such as a patient, a prescription, a doctor, etc. As an implementation is developed, you usually need to define additional implementation objects that are used to provide the required system functionality. Here, I focus on the modeling of real-world objects as part of the requirements or early software design processes.

Generalization

Generalization is an everyday technique that we use to manage complexity. Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes. This allows us to infer that different members of these classes have some common characteristics (e.g., squirrels and rats are rodents). We can make general statements that apply to all class members (e.g., all rodents have teeth for gnawing).

Aggregation

Objects in the real world are often composed of different parts. For example, a study pack for a course may be composed of a book, PowerPoint slides, quizzes, and for further reading.

---------------------------------------------------------------------------------------------------------------------------------------

**Behavioral models**

Behavioral models are models of the dynamic behavior of the system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment. You can think of these stimuli as being of two types:

1. *Data* Some data arrives that has to be processed by the system.

2. *Events* Some event happens that triggers system processing. Events may have associated data but this is not always the case.

**Data-driven** models show the sequence of actions involved in processing input data and generating an associated output. They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system. That is, they show the entire sequence of actions that take place from an input being processed to the corresponding output, which is the system’s response.

Data-driven models were amongst the first graphical software models. In the 1970s, structured methods such as introduced **data-flow diagrams** (**DFD**s) as a way of illustrating the processing steps in a system. Data-flow models are useful because tracking and documenting how the data associated with a particular process moves through the system helps analysts and designers understand what is going on. Data-flow diagrams are simple and intuitive and it is usually possible to explain them to potential system users who can then participate in validating the model.

**Event-driven** modeling shows how a system responds to external and internal events. It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another. For example, a system controlling a valve may move from a state ‘Valve open’ to a state ‘Valve closed’ when an operator command (the stimulus) is received. This view of a system is particularly appropriate for real-time systems.

---------------------------------------------------------------------------------------------------------------------------------------

Model-driven engineering & Model-driven architecture

**Model-driven engineering** (**MDE**) is an approach to software development where models rather than programs are the principal outputs of the development process. The programs that execute on a hardware/software platform are then generated automatically from the models. Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms. Model-driven engineering has its roots in **model-driven architecture (MDA)** Model-driven engineering and model-driven architecture are often seen as the same thing. However, I think that MDE has a wider scope than MDA.

As I discuss later in this section, MDA focuses on the design and implementation stages of software development whereas MDE is concerned with all aspects of the software engineering process. Therefore, topics such as model-based requirements engineering, software processes for model-based development, and modelbased testing are part of MDE but not, currently, part of MDA.

Although MDA has been in use since 2001, model-based engineering is still at an

early stage of development and it is unclear whether or not it will have a significant

effect on software engineering practice. The main arguments for and against MDE are:

1. ***For MDE***Model-based engineering allows engineers to think about systems at a high level of abstraction, without concern for the details of their implementation. This reduces the likelihood of errors, speeds up the design and implementation process, and allows for the creation of reusable, platform-independent application models. By using powerful tools, system implementations can be generated for different platforms from the same model.

Therefore, to adapt the system to some new platform technology, it is only necessary to write a translator for that platform. When this is available, all platform-independent models can be rapidly rehosted on the new platform.

2. ***Against MDE***As I discussed earlier in this chapter, models are a good way of facilitating discussions about a software design. However, it does not always follow that the abstractions that are supported by the model are the right abstractions for implementation. So, you may create informal design models but then go on to implement the system using an off-the-shelf, configurable package. Furthermore, the arguments for platform independence are only valid for large long-lifetime systems where the platforms become obsolete during a system’s lifetime. However, for this class of systems, we know that implementation is not the major problem—requirements engineering, security and dependability, integration with legacy systems, and testing are more significant.