System Model

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system [SOMMERVILLE2010]. 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). However, it is also possible to develop formal (mathematical) models of a system, usually as a detailed system specification.

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:

1. Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.

2. Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation. In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

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 but unfortunately, the real world (also knows as the universe of discourse) is utterly complex so weed to simplify. An abstraction consciously simplifies and picks out the most evident characteristics.

You may develop different models to represent the system from different perspectives [SOMMERVILLE2010]. 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.

These perspectives have much in common with Krutchen's 4 + 1 view of system architecture, where he suggests that you should document a system's architecture and organization from different perspectives.

When developing system models, you can often be flexible in the way that the graphical notation is used. You do not always need to stick rigidly to the details of a notation. 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. This is how models are normally used in so-called 'agile modeling'.

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.

[SOMMERVILLE2010] identifies four important types of system models, namely, context models, interaction models, structural and behavioral models are introduced.

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.

In some cases, the boundary between a system and its environment is relatively clear. For example, where an automated system is replacing an existing manual or computerized system, the environment of the new system is usually the same as the existing system's environment. In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.

Once some decisions on the boundaries of the system have been made, part of the analysis activity is the definition of that context and the dependencies that a system has on its environment. Normally, producing a simple architectural model is the first step in this activity.

Our case study does not deal with all aspects of passing an ordered item to its customer. The proposed system deals with neither the details of payment (e.g., for allowing payment with credit card, the issuing bank's system needs to be accessed) nor the shipment of the purchased product. It is outside the responsibilities of the system to maintain up-to-date information on the stock items. This information is contained in an inventory database that can be updated if new items arrive to stock separately from the online store application. Context models are often depicted as box and line diagrams since such simple diagrams are enough to put the system into context with other systems. The context model for our case study is shown in the following figure.

**Figure 3.1. Context model of the online store.**
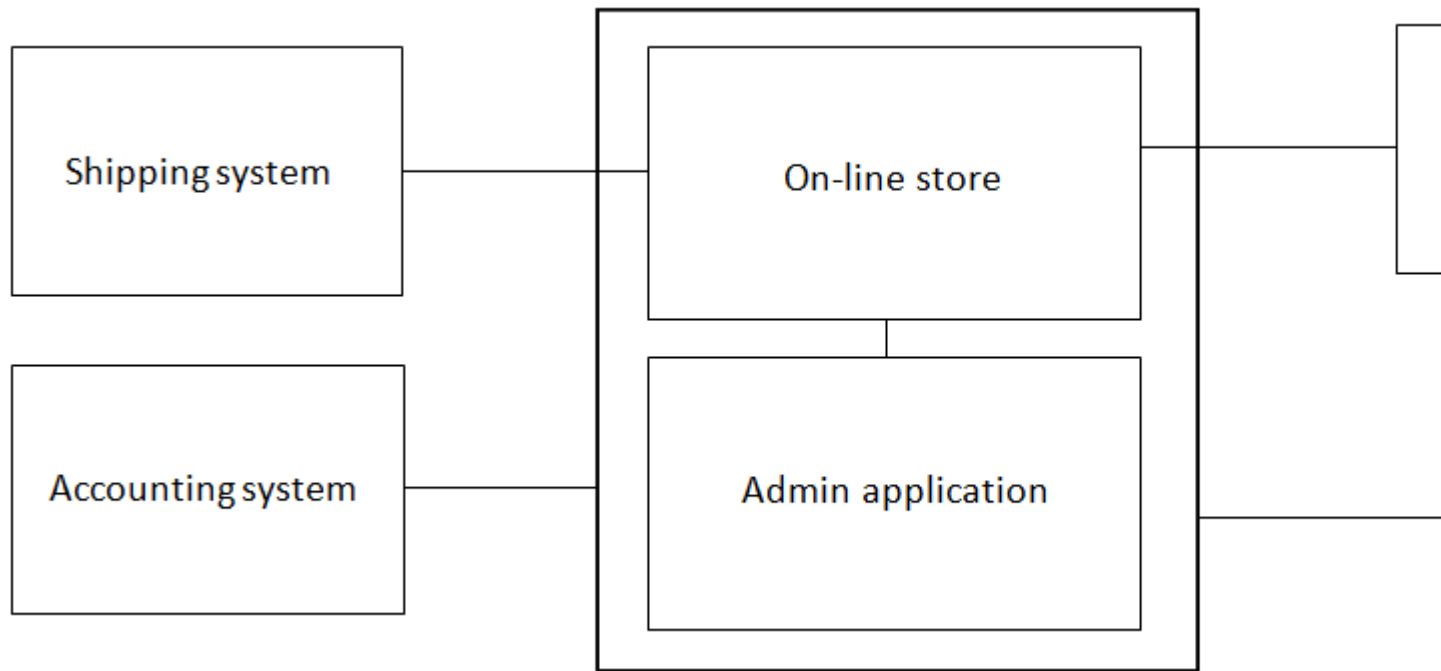
Interaction models

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.

Sommerville covers two related (complementary) 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.

3. **Figure 3.2. Use cases describing interactions to Bank and Shipping system (external systems).**
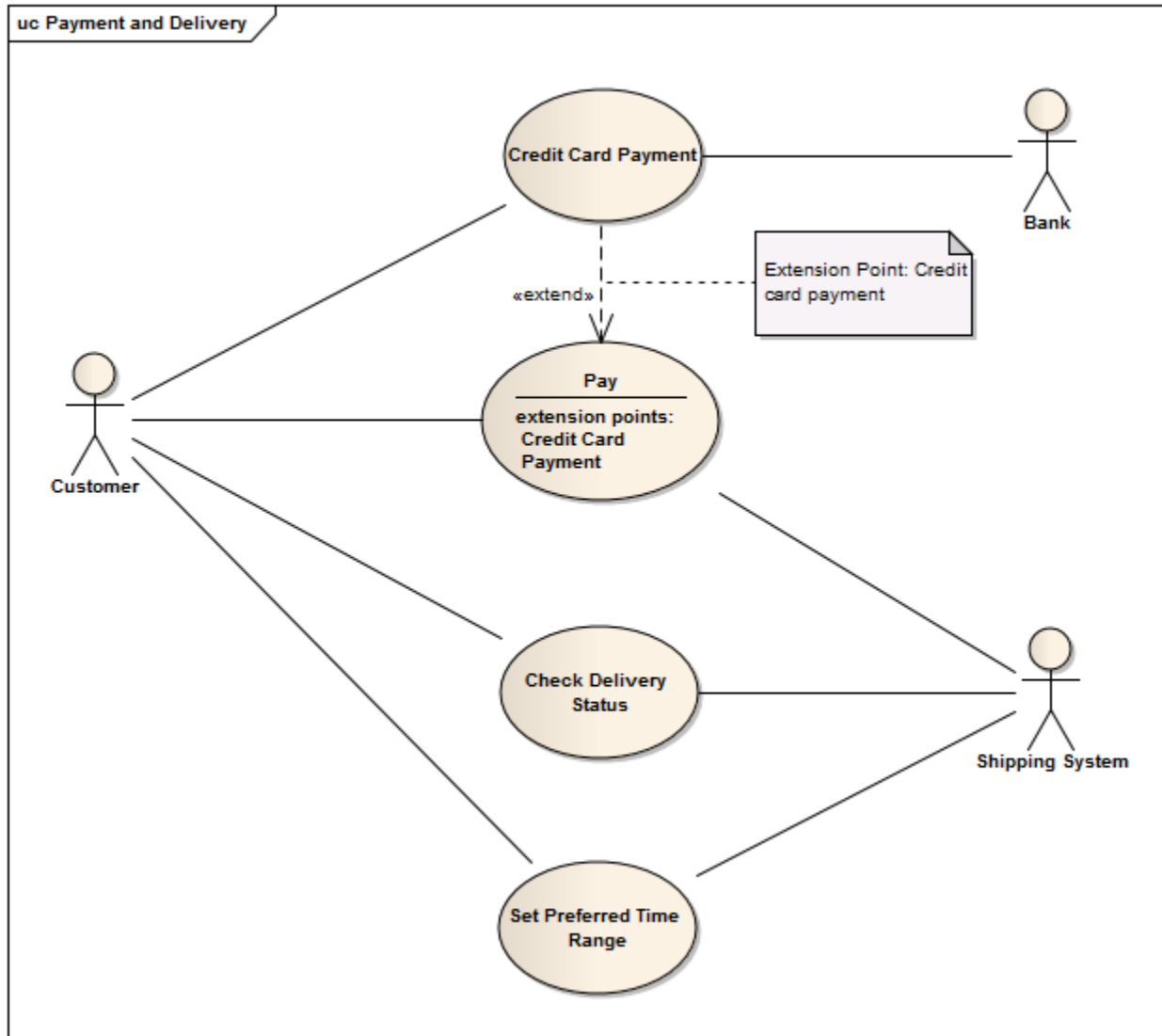
Structural models

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.

You create structural models of a system when you are discussing and designing the system architecture. Architectural design is a particularly important topic in software engineering and UML component, package, and deployment diagrams may all be used when presenting architectural models. Deeper details on architectural modeling will be covered in the next chapter. Here we focus on the creation of the analysis-level class diagrams only that are useful for better understanding of the problem domain.

Domains represent the different subject matters that we need to understand to build a system. A **domain** is an autonomous, real, hypothetical or abstract world inhabited by a set of conceptual entities that behave accordingto characteristic rules and policies. (Definition from [MELLOR2002].) We abstract like things and and call them **classes**. In forming such abstractions, we ignore most of the things as our aim is to leave out details and concentrate only on the important aspects. The remaining things are are grouped according to some perceptions about what is means to be "like".

Important

A domain model is a representation of real-world conceptual classes, not of software components. It is not a set of diagrams describing software classes, or software objects with responsibilities. A domain model serves as a visual dictionary of abstractions.

UML uses class diagrams at all abstraction levels to describe the static structure of a system. Starting from domain classes and abstract concepts through design patterns till the constructs of programming language(s) used for the implementation, we use classes. However, you should not think that the same name means the same construct: a UML class and a Java class, for example, might be quite similar and very different, as well. This is because UML uses 4 interpretations when talking about classes:

Table 3.1. UML interpretations of classes

Class as a **concept**	Classes can be used to document and define concepts. We use classes to describe (more) abstract concepts based on some existing concepts.
Class as a **data type**	Classes can be considered as data types. Object instances are considered to be the values of the data type so a class is defined by its internal structure (intension).
Class as a **set of objects**	Classes can be seen as a set of similar objects that belong to the same class (extension).
Class as **implementation**	In object-oriented programming languages classes play only the role of an implementation that is shared among the instances.

Class diagrams appear in different contexts. It is very convenient to classify them based on the stages of the software engineering lifecycle where they are used.

Analysis-level classes are the elements of the problem domain. They help us to understand and document the problem space.

Design-level classes are about how to transform domain structures into technical structures so we work in the solution space instead of the problem space. Design-level classes combine domain motivations with technical aspects. Some of the classes are closer to the domain elements while others are closer to the elements of the solution.

Implementation-level classes show how a solution is given. These classes map directly to the corresponding construct of the implementation language (Java, C# or C++, etc.).

First we need to establish an analysis-level class diagram as this is waht is needed to describe the problem domain itself. To achieve this, an abstraction process is needed: we need to find out

what elements (constructs) of the real world are important for our **universe of discourse**, how can the concepts of the problem domain be revealed.

We choose attributes that support the ideas of likeness we have in mind when abstracting the class. Relationships exist between the things of the domain that are abstracted to associations between classes. The result of this abstraction process is an analysis-leve class diagram that will serve as a good source for further refinements (to evolve it into design-level and probably implementation-level diagrams).

UML 2 Class diagrams [UML-DIAGRAMS.ORG]

The class diagram shows the basic building blocks of any object-orientated system. Class diagrams depict a static view of the model, or part of the model, describing what attributes and behavior it has rather than detailing the methods for achieving operations. Class diagrams are most useful in illustrating relationships between classes and interfaces. Generalizations, aggregations, and associations are all valuable in reflecting inheritance, composition or usage, and connections respectively.

- A **class** is a classifier which describes a set of objects that share the same

  o features

  o constraints

  o semantics (meaning).

Identification of domain classes

The domain model illustrates conceptual classes or vocabulary in the domain. Informally speaking, a conceptual class is an idea, thing, or object. More formally, a conceptual class may be considered in terms of its symbol, intension, and extension.

- **Symbol**: words or images representing a conceptual class.

- **Intension**: the definition of a conceptual class.

- **Extension**: the set of examples to which the conceptual class applies.

Consider the conceptual class for a product as an example. Its symbol is `Product`, its intension is that is represents something that can be sold or bought which has a name and a unit price. The `Product`'s extension is the set of all products.

Important

It is better to overspecify a domain model with lots of fine-grained conceptual classes than to underspecify it. Do not think that a domain model is better if it has fewer conceptual classes; quite the opposite tends to be true.

It is common to miss conceptual classes during the initial identification step, and to discover them later during the consideration of attributes or associations, or during design work. That's why this work should be performed in iterations. When a new conceptual class is found, the domain model should be augmented. Relational database design principles are probably not the best to follow since when modeling the domain, even attributeless classes might be important to include as they describe a purely behavioral role in the domain instead of playing an informational role.

A common technique used for class identification is to look for the noun phrases in the overall project description. In our case study, we had nouns like customer, order, payment, category, book, DVD, product, name, address, etc. These nouns will be potential classes and attributes.

In order to decide whether a given noun describes a class or an attibute we can apply the following guideline: if it has an identity in the domain then it is a class, otherwise it is an attribute. In this latter case, of course, we need to find its class, as well. That is the reason why we execute this process in iterations. First, we only concentrate on classes, then in a new iteration we assign attributes to classes. The third phase should be finding additional attributes that did not come up in the textual description.

By following these guidelines, we can find a set of initial classes that belong to the domain.

Figure 3.34. Initial domain classes
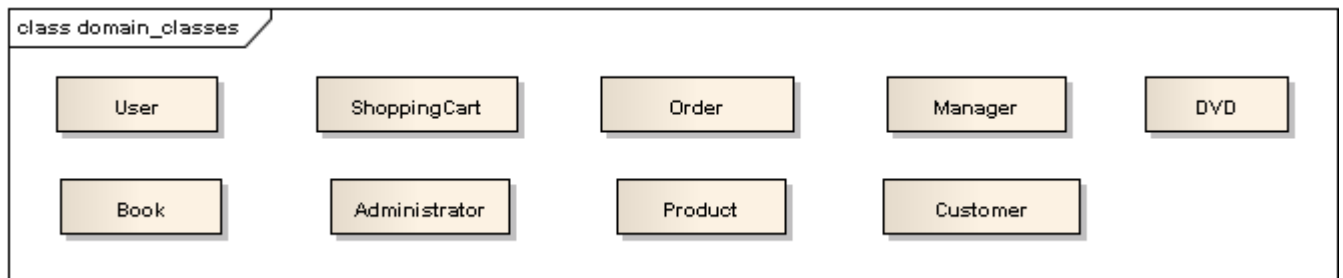
Note

Of course, the actual domain highly influences such decisions. Would this system be used for the registry of real estates, address of a property might become a first-class citizen of this system and should have been modeled as a class instead of an attribute.

Name can be associated with users, customers, manages, administrators but produts might also have a name. Category is something that allows grouping of various products. Address is belonging to humans (a home address, for example), however, an order can also have associated addresses in multiple roles: a billing address and a shipping address are both addresses. From the domain's perspective these belong to the order not the customer since without an order it would be pointless to talk about a customer's billing or shipping address. That's why it is not considered as a attrribute of the customer but something that describes an order.

After that, we need to identify additional attributes of our domain classes. Even if they were not mentioned in the description, customer's phone number (that can be used to contact the customer), the order's date or the actual price of a product are examples of such descriptive information that can easily be identified when investigating the domain deeper. Even we might think that orders have an attribute called customer which represents the person fro whom this order has been created.

Important

When modeling an application, you will always have lots of options. Many of those might result in a good design (even if it is very hard to tell what **good** is). However, you should **always** be aware of the consequences of your decisions. They will form constraints on subsequent design activities.
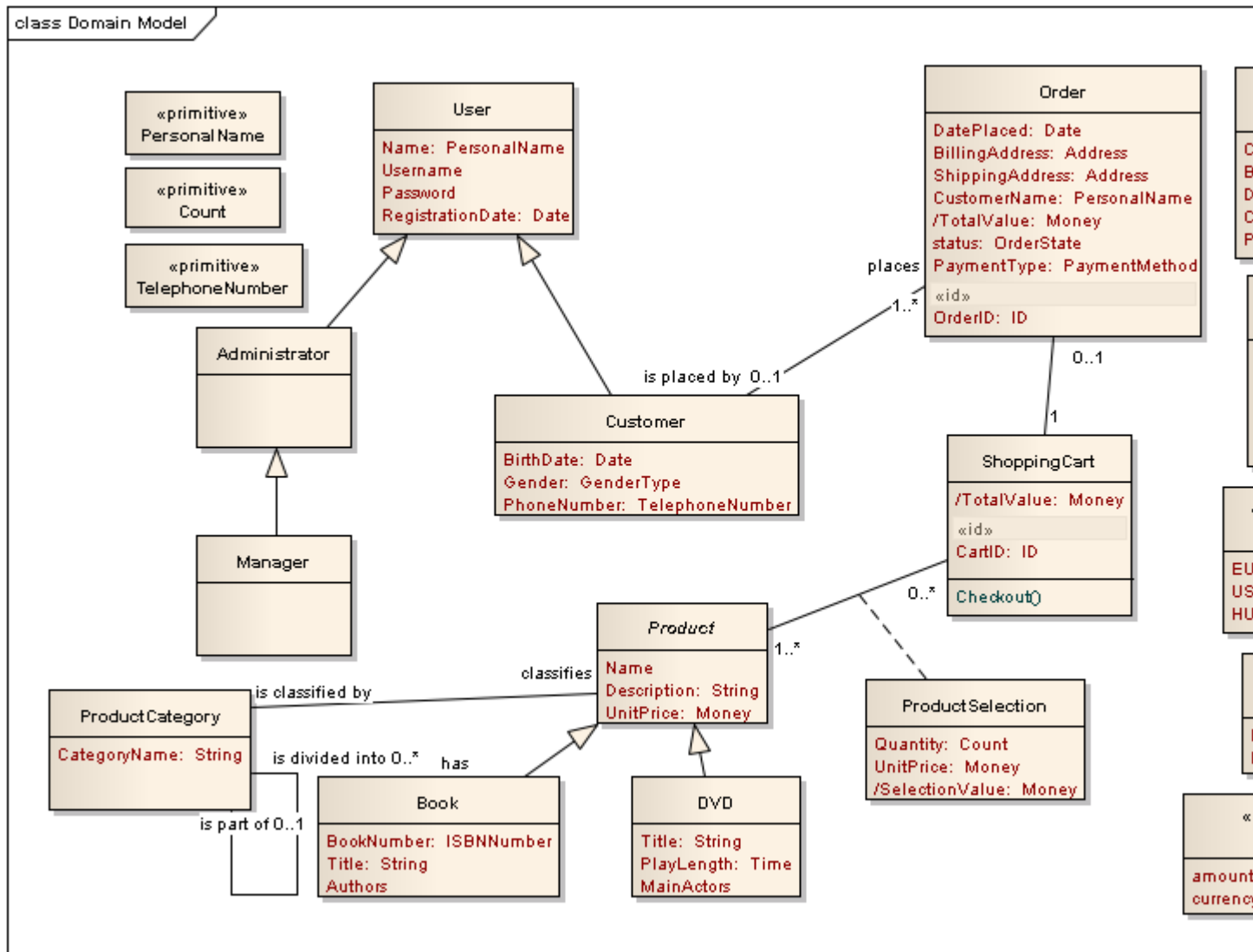
Refining the model

Even if enough care is taken when designing our domain models, there might be some concepts that were missing. When they are found (regardless which stage of the development cycle we are in) the model should be revised. In many cases, this is not about forgotten concepts but either the deeper understanding results in the need for some claases, attrubutes or relationships or we left out some concepts on purpose for subsequent iterations.

The latter has happened with our initial model. There we omitted details on type of attributes, however, UML can use stereotypes primitive and dataType for defining special classes that represent data types. Primitive types are those for which identity is not used in checking equality. PersonalName, TelephoneNumber and Count are examples of primitive types. Of course, one might think that simply using Strings instead of first two or integer instead of Count would be enough since thi is how we will implement them finally. However, it would be a bad idea to make such a decision now, at analysis level. Of course, not all of the strings would qualify as a name of a person or as a phone number. They must follow some patterns that are known from the domain.Using String in place of TelephoneNumber would mean that we do not want to check the format of phone numbers for sure, just as we do not do any checks on other textual objects (like on product description).

We can also apply analysis patterns. Pattern "***P of EAA 488***" descibes the Money pattern to represent monetary values. This pattern was described by Martin Fowler in page 488 of his excellent book entitled Patterns of Enterprise Application Architecture [FOWLER2002] (a similar pattern, Quantity, has been described earlier in [FOWLER1996]). This pattern advocates the coupling of an amount and a currency type together. For the sake of completeness, the primitive types can be seen as a manifestation of Value object pattern ("***P of EAA 486***").

We can also notice that the class Order did not contain any information on the status of the order (which needs to be traceable according to the reuirements) or the payment type how the order is paid. Most systems offer a set of possible order states and payment methods that are allowed. Therefore we can create few enumerations that list those acceptable values. These are illustrated in Figure 3.37, "Revised domain model with supporting data types".

Figure 3.37. Revised domain model with supporting data types

class Domain Model

«primitive»
PersonalName

«primitive»
Count

«primitive»
TelephoneNumber

User
Name: PersonalName
Username
Password
RegistrationDate: Date

Administrator

Manager

Customer
BirthDate: Date
Gender: GenderType
PhoneNumber: TelephoneNumber

is placed by 0..1

Order
DatePlaced: Date
BillingAddress: Address
ShippingAddress: Address
CustomerName: PersonalName
/TotalValue: Money
status: OrderState
PaymentType: PaymentMethod
«id»
OrderID: ID

places 1..*

0..1

1

ShoppingCart
/TotalValue: Money
«id»
CartID: ID
Checkout()

0..*

Product
Name
Description: String
UnitPrice: Money

classifies 1..*

is classified by

ProductCategory
CategoryName: String

is divided into 0..*   has

is part of 0..1

Book
BookNumber: ISBNNumber
Title: String
Authors

DVD
Title: String
PlayLength: Time
MainActors

ProductSelection
Quantity: Count
UnitPrice: Money
/SelectionValue: Money

EU
US
HU

amount
currency

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.

Many business systems are data processing systems that are primarily driven by data. They are controlled by the data input to the system with relatively little external event processing. Their processing involves a sequence of actions on that data and the generation of an output. For example, our bookstore system will accept information about orders made by a customer,

calculate the costs of these orders, and using another system, it will generate an invoice to be sent to that customer.

Many business systems are data processing systems that are primarily driven by data. They are controlled by the data input to the system with relatively little external event processing. Their processing involves a sequence of actions on that data and the generation of an output. For example, our bookstore system will accept information about orders made by a customer, calculate the costs of these orders, and using another system, it will generate an invoice to be sent to that customer.

Data-driven modeling

Data-driven models show the sequence of actions involved in processing input data and generating the associated output. This is very useful during the analysis stage since they show end-to-end processing in a system which means that they show the entire action sequence of how input data become output data. In ither words, it shows the response of the system to particular input.

In UML, activity and sequence diagrams can be used to describe such data flows. Note that these are the same diagram types we used for interaction modeling but now the emphasis is put on the processing itself, not on the objects that will participate in processing (interactions). That is why acivity diagrams are better used for that purpose since the lifelines of sequence diagrams depict objects and actors therefore some attention must be paid to responsibility allocation when using sequence diagrams.

The basic processes of the online store (starting from the insertion of a new product through browsing and selecting products to buy till tracking the order's delivery and provide feedback) is shown on the following activity diagram.

Event-driven modeling

Event-driven modeling shows how a system responds to external and internal events (stimuli). It is based on the assumption that a system has a finite number of states and that an event (stimulus) may cause a transition from one state to another.

The UML supports event-based modeling using *state machine diagrams*

UML 2 State machine diagrams

A state machine diagram models the behaviour of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events. It contains the following elements:

- **State.** A state is denoted by a round-cornered rectangle with the name of the state written inside it. There are two special states:

  o **Initial state.** The initial state is denoted by a filled black circle and may be labeled with a name.

  o **Final state.** The final state is denoted by a circle with a dot inside and may also be labeled with a name.

- **Transitions.** Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below. *Trigger* is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. *Guard* is a condition which must be true in order for the trigger to cause the transition. *Effect* is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

- **State action.** State actions describe effects associated with a state. A state action is an activity label/behavior expression pair. The activity label identifies the circumstances under which the behavior will be invoked. There are three reserved activity labels:

  o entry: the behavior is performed upon entry to the state,

  o do: ongoing behavior, performed as long as the element is in the state,

  o exit: a behavior that is performed upon exit from the state.

- **Self-transition.** A state can have a transition that returns to itself. This is most useful when an effect is associated with the transition.

- **Entry point.** If we do not enter the machine at the normal initial state, we can have additional entry points.

- **Exit point.** In a similar manner to entry points, it is possible to have named alternative exit points.

Besides these main element, it is important to note that UML state machine diagrams support the notion of superstates that encapsulate a number of separate states. This superstate can be used as a single state on a higher-level model but is then expanded to show more details.

The following state machine diagram shows the internal states and transitions of a shopping cart.

Figure 3.39. State machine diagram of the class ShoppingCart
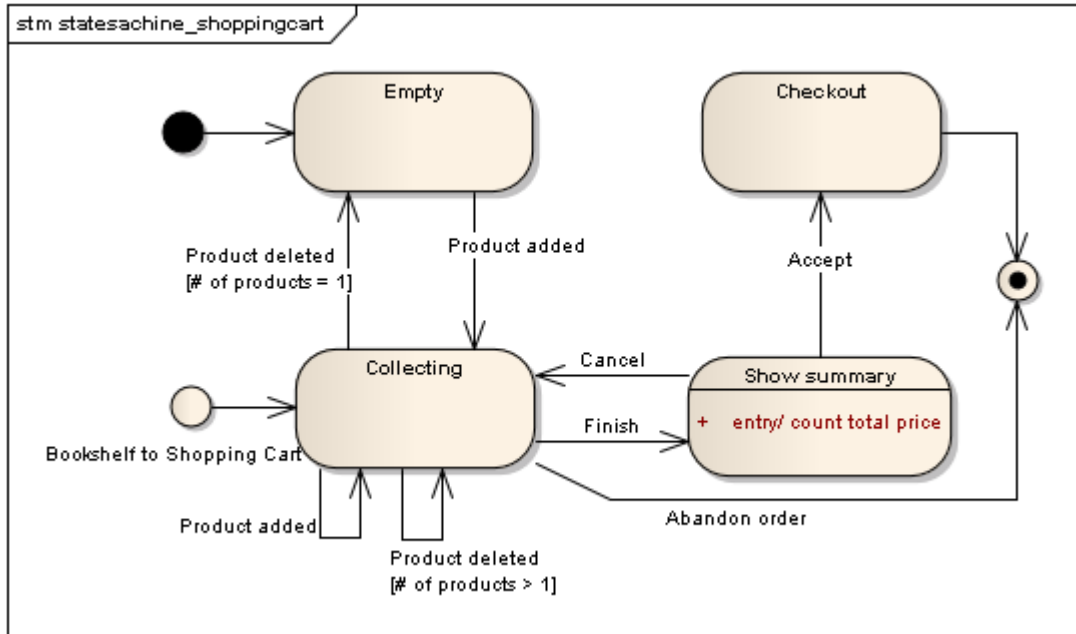
When the shopping cart is initiated, it will be in an Empty state. Whenever products are added, a transition to Collecting state is performed. When the shopping cart is in the Collecting state and a Product deleted stimulus happens then based on the guards (if the number of products in cart is equal to 1 or greater) a transition to Empty state might happen.

One might argue that Empty and Collecting are very similar that it does not worth separating them. This can be a valid point, however, the reason why we separated is that when converting a bookshelf's contents (like a wishlist) to shopping cart contents will not allow having an empty cart so the inclusion of the entry point resulted in the separation.

Note

When the cart is in the Empty state, it cannot receive a Product deleted stimuli (there is no such transition starting from Empty) therefore this model disallows deleting products from an empty cart (of course, it is impossible). This is a second reason why states Empty and Collecting has been separated.

The state "Show summary" contains an effect that is executed upon entering the state: total price should be counted as it should be presented to the customer.

The UML notation lets you indicate the activity that takes place in a state. In a detailed system specification you have to provide more detail about both the stimuli and the system states. These are shown in the following tables with a tabular description of each state and how the stimuli that force state transitions are generated.

Table 3.4. States of a shopping cart

| State	Description
Empty	The shopping cart does not contain any products.
Collecting	The cart contains some products.

13

Show summary	Overview of the shopping cart contents are shown.
Checkout	Cart contents are ordered.

Table 3.5. Stimuli of a shopping cart

Stimulus	Description
Product added	The user pressed the Add product button.
Product deleted	The user selected a product and pressed the Remove product button.
Finish	The user pressed the Checkout button.
Cancel	The user pressed the Continue shopping button.
Accept	The user pressed the Accept button so the contents of the cart are ordered.
Abandon order	The user pressed the Abandon order button.

A less detailed state machine diagram showing a possible set of order statuses is shown below:

Figure 3.40. State machine Order
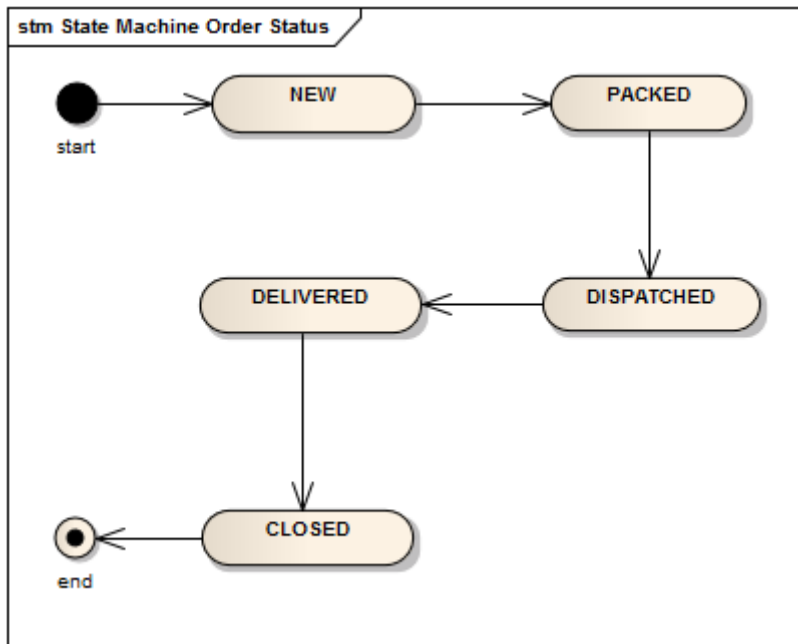
Robustness analysis

Rosenberg and Stephen [ROSENBERG2007] introduced robustness analysis as a way for filling the gap between analysis (the *what*) and design (the *how*). From that point of view robustness analysis is a preliminary design when designers make assumptions on the design and start thinking of the possible technical solutions.

For supporting robustness analysis, they use robustness diagrams. This is a nonstandard diagram type in the manner that it is not described by the UML specification, however, it uses UML concepts. It is a specialized communication diagram that uses stereotyped objects. These stereotypes are defined in UML:

- **«boundary».** The *interface* between the system and the outside world. Boundary objects are typically screens or web pages (i.e., the presentation layer that the actor interacts with).

- **«entity».** Entity objects are usually objects from the domain model.

- **«control».** Control objects are the "glue" between boundary and entity objects.

A robustness diagram is somewhat of a hybrid between a class diagram and an activity diagram. It visually represents a use case's behavior, showing both participating classes and software behavior. Nevertheless, it does not describe which class is responsible for which parts of the behavior. A robustness diagram is probably easier to read than an activity diagram since objects talks to each other. This flow of action is represented by a line between the two objects that are talking to each other.
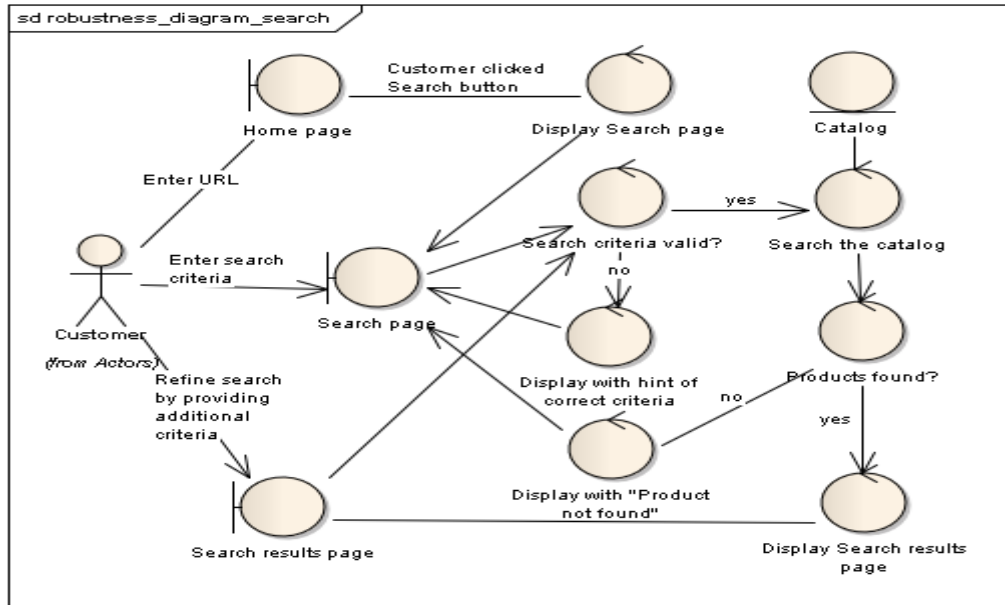
It is a specialized communication diagram since not all object can talk to any other object. It is useful to think of boundary objects and entity objects as being nouns, and controllers as being verbs. Then the following rules apply in robustness diagrams:

- Nouns can talk to verbs (and vice versa).

- Nouns can't talk to other nouns.

- Verbs can talk to other verbs.

These rules help to enforce a *noun-verb-noun* pattern in the text of use cases. This is useful as sequence diagrams have the very same nature: the objects are the nouns, and the messages that go between them are the verbs. So by following that pattern it will be easier to do the detailed design task later. Robustness analysis provides a sanity check for our use cases.

By revisiting our use cases, the first thing that we can find out that browsing and searching products need to be separated into two use cases since they have different operations. Here we only focus on searching which is the more complex of them.

Figure 3.42. Robustness diagram for Search product

sd robustness_diagram_search

Based on the robustness diagram we can rephrase the use case as follows:

Use case ID	UC001A
Use case name	**Search Products**
Actors	Customer
Description	Search for products based on some criteria.
Trigger	The customer wants to browse among products or the customer would like to search for certain products.
Precondition	Customer starts a web browser.
Postcondition	Search results meeting the criteria are displayed.
Normal flow	1. Customer visits application home page. 2. Customer clicks Search button. 3. Search page is displayed by the system. 4. Customer enters search criteria. 5. The system validates the criteria provided. 6. The system looks up the catalog to find the products that meet the criteria. 7. Search results page is displayed with the products fulfilling the criteria.
Alternative flows	Refine Search Results

Use case ID	UC001A
Use case name	**Search Products**
	The following steps are added: 8. Customer refines search results by providing additional criteria. 9. Steps 5–7 are re-executed.
Exceptions	In Step 5, if search criteria is invalid or even missing then Step 3 (display search page) will be executed along with some hints on valid criteria. In Step 6, if no products meet the criteria then Step 3 (display search page) will be executed along with providing the error message "Product not found".
Includes	None
Notes and issues	None

Based on that we can also introduce a new class to our domain model: Catalog. The Catalog class is an abstraction of a container of all products. This is not to be confused with the inventory that has been mentioned earlier. The inventory contains information about the actually available pieces of products (those that are stocked) while catalog contains product metadata about each existing product (regardless of that there are any pieces of them on stock). Inventory was concerned as implemented by an external application that provides an interface to retrieve information on products in warehouses, however, catalog should be an integral part of this application since the whole lifecycle of a product is covered by our system's functionalities.

We have also created a roustness diagram for one of the most important use cases of the system, Place order.

Figure 3.43. Robustness diagram for Place order