Chapter 10

DOMAIN MODEL-VISUALIZING CONCEPTS

It's all very well in practice, but it will never work in theory.

—anonymous management maxim

Objectives

Identify conceptual classes related to the current iteration requirements.

Create an initial domain model.

Distinguish between correct and incorrect attributes.

Add *specification* conceptual classes, when appropriate.

Compare and contrast conceptual and implementation views.

Introduction

A domain model is widely used as a source of inspiration for designing software objects, and will be a required input to several subsequent artifacts discussed in this book. Therefore, it is important to read this chapter if the subject of domain modeling is unfamiliar.

A domain model illustrates meaningful (to the modelers) conceptual classes in a problem domain; it is the most important artifact to create during object-oriented analysis.[1] This chapter explores introductory skills in creating domain

1. Use cases are an important requirements analysis artifact, but are not object-oriented. They emphasize a process view of the domain.

models. The following two chapters expand on domain modeling skills—adding attributes and associations.

Identifying a rich set of objects or conceptual classes is at the heart of object-oriented analysis, and well worth the effort in terms of payoff during the design and implementation work.

The identification of conceptual classes is part of an investigation of the problem domain. The UML contains notation in the form of class diagrams to illustrate domain models.

Key Idea
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.

10.1 Domain Models

The quintessential object-oriented step in analysis or investigation is the decomposition of a domain of interest into individual conceptual classes or objects—the things we are aware of. A **domain model** is a *visual* representation of conceptual classes or real-world objects in a domain of interest [MO95, Fowler96]. They have also been called **conceptual models** (the term used in the first edition of this book), **domain object models, and analysis object models.**[2]

The UP defines a Domain Model[3] as one of the artifacts that may be created in the Business Modeling discipline.

Using UML notation, a domain model is illustrated with a set of **class diagrams** in which no operations are defined. It may show:

- domain objects or conceptual classes
- associations between conceptual classes
- attributes of conceptual classes

For example, Figure 10.1 shows a partial domain model. It illustrates that the conceptual class *of Payment* and *Sale* are significant in this domain, that a *Pay-*

2. They are also related to conceptual entity relationship models, which are capable of showing purely conceptual views of domains, but that have been widely re-interpreted as data models for database design. Domain models are not data models.
3. Capitalization of Domain Model is used when I wish to emphasize it as an official model defined in the UP, vs. the general well-known concept of "domain models."

merit is related to a *Sale* in a way that is meaningful to note, and that a *Sale* has a date and time. The details of the notation are not important at this time.

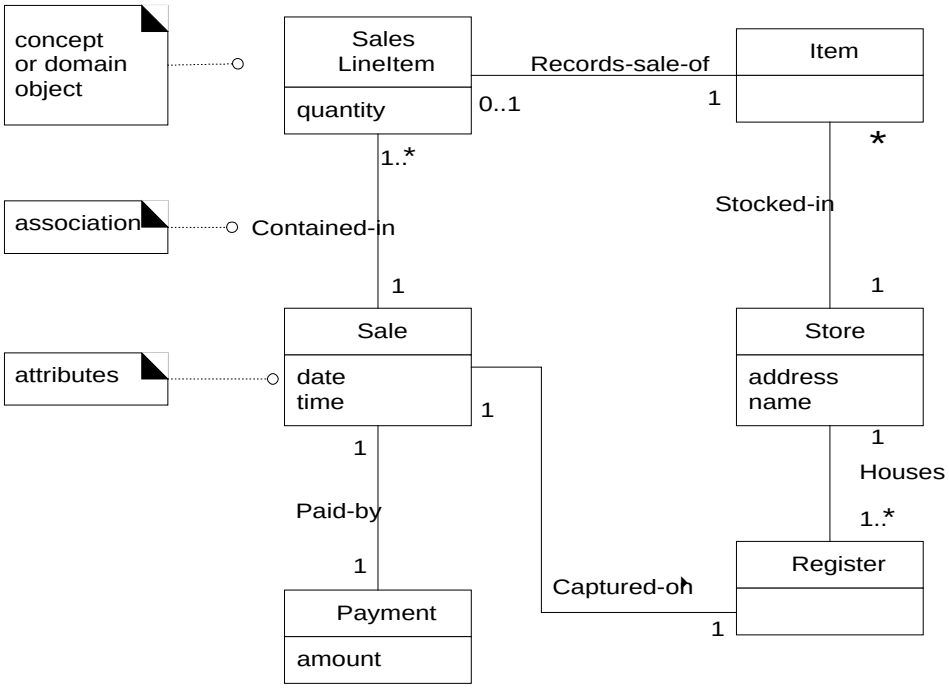


Figure 10.1 Partial domain model—a visual dictionary. The numbers at each end of the line indicate multiplicity, which is described in a subsequent chapter.

Key Idea: Domain Model—A Visual Dictionary of Abstractions

Please reflect on Figure 10.1 for a moment. It visualizes and relates some words or conceptual classes in the domain. It also depicts an *abstraction* of the conceptual classes, because there are many things one could communicate about registers, sales, and so forth. The model displays a partial view, or abstraction, and ignores uninteresting (to the modelers) details.

The information it illustrates (using UML notation) could alternatively have been conveyed in prose, in statements in the Glossary or elsewhere. But it is easy to comprehend the discrete elements and their relationships in this visual language, since a significant percentage of the brain participates in visual processing—it is a human strength.

Thus, the domain model may be considered a *visual dictionary* of the noteworthy abstractions, domain vocabulary, and information content of the domain.

Domain Models Are not Models of Software Components

A domain model, as shown in Figure 10.2, is a visualization of things in the real-world domain of interest, *not* of software components such as a Java or C++ class (see Figure 10.3), or software objects with responsibilities. Therefore, the following elements are not suitable in a domain model:

- Software artifacts, such as a window or a database, unless the domain being modeled is of software concepts, such as a model of graphical user interfaces.
- Responsibilities or methods.[4]

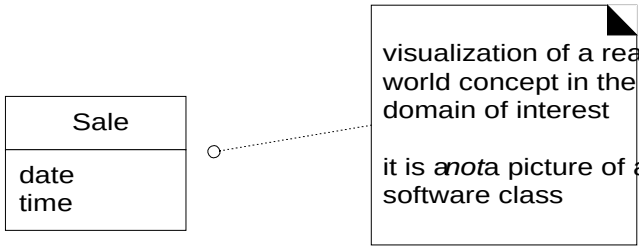


Figure 10.2 A domain model shows real-world conceptual classes, not software classes.

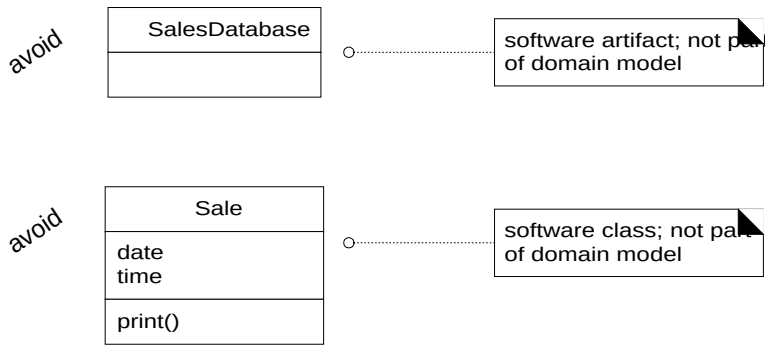


Figure 10.3 A domain model does not show software artifacts or classes.

4. In object modeling, we usually speak of responsibilities related to software components. And methods are purely a software concept. But, the domain model describes real-world concepts, not software components. Considering object responsibilities during *design* work is very important; it is just not part of this model. One valid case in which responsibilities may be shown in a domain model is if it includes human worker roles (such as Cashier), and the modeler wishes to record the responsibilities of these human workers.

Conceptual Classes

The domain model illustrates conceptual classes or vocabulary in the domain. Informally, 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 [MO95] (see Figure 10.4).

- **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.

For example, consider the conceptual class for the event of a purchase transaction. I may choose to name it by the symbol *Sale*. The intension of a *Sale* may state that it "represents the event of a purchase transaction, and has a date and time." The extension *of Sale* is all the examples of sales; in other words, the set of all sales.

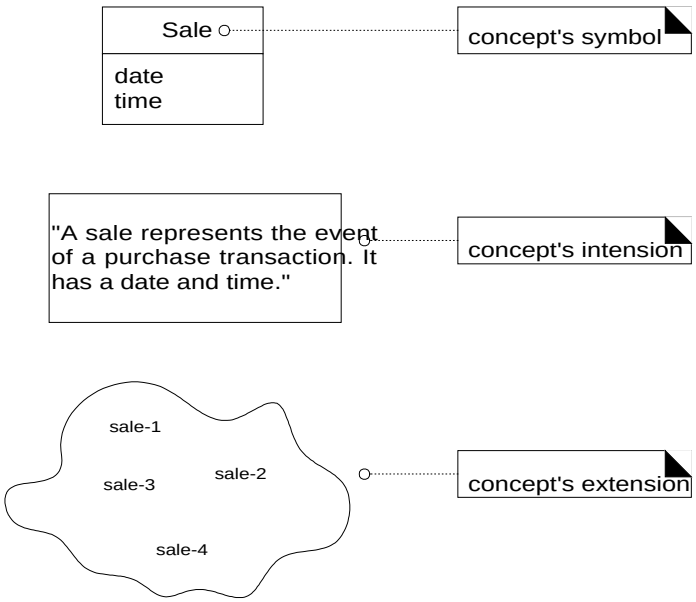


Figure 10.4 A conceptual class has a symbol, intension, and extension.

When creating a domain model, it is usually the symbol and intensional view of a conceptual class that are of most practical interest.

Domain Models and Decomposition

Software problems can be complex; decomposition—divide-and-conquer—is a common strategy to deal with this complexity by division of the problem space into comprehensible units. In **structured analysis,** the dimension of decomposition is by processes or *functions.* However, in object-oriented analysis, the dimension of decomposition is fundamentally by things or entities in the domain.

> A central distinction between object-oriented and structured analysis is: division by conceptual classes (objects) rather than division by functions.

Therefore, a primary analysis task is to identify different concepts in the problem domain and document the results in a domain model.

Conceptual Classes in the Sale Domain

For example, in the real-world domain of sales in a store, there are the conceptual classes *of Store, Register,* and *Sale.* Therefore, our domain model, shown in Figure 10.5, may include *Store, Register,* and *Sale.*

Figure 10.5 Partial domain model in the domain of the store.

10.2 Conceptual Class Identification

Our goal is to create a domain model of interesting or meaningful conceptual classes in the domain of interest (sales). In this case, that means concepts related to the use case *Process Sale.*

In iterative development, one incrementally builds a domain model over several iterations in the elaboration phase. In each, the domain model is limited to the prior and current scenarios under consideration, rather than a "big bang" model which early on attempts to capture all possible conceptual classes and relationships. For example, this iteration is limited to a simplified cash-only *Process Sale* scenario; therefore, a partial domain model will be created to reflect just that—not more.

The central task is therefore to identify conceptual classes related to the scenarios under design.

The following is a useful guideline in identifying conceptual classes:

> 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. When found, they may be added to the domain model.

Do not exclude a conceptual class simply because the requirements do not indicate any obvious need to remember information about it (a criterion common in data modeling for relational database design, but not relevant to domain modeling), or because the conceptual class has no attributes.

It is valid to have attributeless conceptual classes, or conceptual classes which have a purely behavioral role in the domain instead of an information role.

Strategies to Identify Conceptual Classes

Two techniques are presented in the following sections:

1. Use a conceptual class category list.
2. Identify noun phrases.

Another excellent technique for domain modeling is the use of **analysis patterns,** which are existing partial domain models created by experts, using published resources such as *Analysis Patterns* [Fowler96] and *Data Model Patterns* [Hay96].

Use a Conceptual Class Category List

Start the creation of a domain model by making a list of candidate conceptual classes. Table 10.1 contains many common categories that are usually worth considering, though not in any particular order of importance. Examples are drawn from the store and airline reservation domains.

Conceptual Class Category	Examples
physical or tangible objects	*Register* *Airplane*
specifications, designs, or descriptions of things	*ProductSpecification* *FlightDescription*
places	*Store* *Airport*
transactions	*Sale, Payment* *Reservation*
transaction line items	*SalesLineItem*
roles of people	*Cashier* *Pilot*
containers of other things	*Store, Bin* *Airplane*
things in a container	*Item* *Passenger*
other computer or electro-mechanical systems external to the system	*CreditPaymentAuthorizationSystem* *AirTrafficControl*
abstract noun concepts	*Hunger* *Acrophobia*
organizations	*SalesDepartment* *ObjectAirline*
events	*Sale, Payment, Meeting* *Flight, Crash, Landing*
processes (often *not* represented as a concept, but may be)	*SellingAProduct* *BookingASeat*
rules and policies	*RefundPolicy* *CancellationPolicy*
catalogs	*ProductCatalog* *PartsCatalog*

Conceptual Class Category	Examples
records of finance, work, contracts, legal matters	*Receipt, Ledger, EmploymentContract MaintenanceLog*
financial instruments and services	*LineOfCredit Stock*
manuals, documents, reference papers, books	*DailyPriceChangeList RepairManual*

Table 10.1 Conceptual Class Category List.

Finding Conceptual Classes with Noun Phrase Identification

Another useful technique (because of its simplicity) suggested in [Abbot83] is linguistic analysis: identify the nouns and noun phrases in textual descriptions of a domain, and consider them as candidate conceptual classes or attributes.

> Care must be applied with this method; a mechanical noun-to-class mapping isn't possible, and words in natural languages are ambiguous.

Nevertheless, it is another source of inspiration. The fully dressed use cases are an excellent description to draw from for this analysis. For example, the current scenario of the *Process Sale* use case can be used.

Main Success Scenario (or Basic Flow):

1. **Customer** arrives at a **POS checkout** with **goods** and/or **services** to purchase.
2. **Cashier** starts a new **sale.**
3. **Cashier** enters **item identifier.**
4. System records **sale line item** and presents **item description, price,** and running **total.** Price calculated from a set of price rules.

Cashier repeats steps 2-3 until indicates done.

5. System presents total with **taxes** calculated.
6. Cashier tells Customer the total, and asks for **payment.**
7. Customer pays and System handles payment.
8. System logs the completed **sale** and sends sale and payment information to the external **Accounting** (for accounting and **commissions)** and **Inventory** systems (to update inventory).
9. System presents **receipt.**
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

7a. Paying by cash:

1. Cashier enters the cash **amount tendered.**

2. System presents the **balance due,** and releases the **cash drawer.**
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

The domain model is a visualization of noteworthy domain concepts and vocabulary. Where are those terms found? In the use cases. Thus, they are a rich source to mine via noun phrase identification.

Some of these noun phrases are candidate conceptual classes, some may refer to conceptual classes that are ignored in this iteration (for example, "Accounting" and "commissions"), and some may be attributes of conceptual classes. Please see the subsequent section and chapter on attributes for advice on distinguishing between the two.

A weakness of this approach is the imprecision of natural language; different noun phrases may represent the same conceptual class or attribute, among other ambiguities. Nevertheless, it is recommended in combination with the *Conceptual Class Category List* technique.

10.3 Candidate Conceptual Classes for the Sales Domain

From the Conceptual Class Category List and noun phrase analysis, a list is generated of candidate conceptual classes for the domain. The list is constrained to the requirements and simplifications currently under consideration—the simplified scenario *of Process Sale.*

Register	*ProductSpecification*
Item	*SalesLineItem*
Store	*Cashier*
Sale	*Customer*
Payment	*Manager*
ProductCatalog	

There is no such thing as a "correct" list. It is a somewhat arbitrary collection of abstractions and domain vocabulary that the modelers consider noteworthy. Nevertheless, by following the identification strategies, similar lists will be produced by different modelers.

Report Objects—Include Receipt in the Model?

A receipt is a record of a sale and payment and a relatively prominent conceptual class in the domain, so should it be shown in the model?

Here are some factors to consider:

- A receipt is a report of a sale. In general, showing a report of other information in a domain model is not useful since all its information is derived from other sources; it duplicates information found elsewhere. This is one reason to exclude it.
- A receipt has a special role in terms of the business rules: it usually confers the right to the bearer of the receipt to return bought items. This is a reason to show it in the model.

Since item returns are not being considered in this iteration, *Receipt* will be excluded. During the iteration that tackles the *Handle Returns* use case, it would be justified to include it.

10.4 Domain Modeling Guidelines

How to Make a Domain Model

Apply the following steps to create a domain model:

1. List the candidate conceptual classes using the Conceptual Class Category List and noun phrase identification techniques related to the current requirements under consideration.
2. Draw them in a domain model.
3. Add the associations necessary to record relationships for which there is a need to preserve some memory (discussed in a subsequent chapter).
4. Add the attributes necessary to fulfill the information requirements (discussed in a subsequent chapter).

An adjunct useful method is to learn and copy analysis patterns, which are discussed in a later chapter.

On Naming and Modeling Things: The Mapmaker

The mapmaker strategy applies to both maps and domain models.

> Make a domain model in the spirit of how a cartographer or mapmaker works:
>
> - Use the existing names in the territory.
> - Exclude irrelevant features.
> - Do not add things that are not there.

A domain model is a kind of map of concepts or things in a domain. This spirit emphasizes the analytical role of a domain model, and suggests the following:

- A mapmaker uses the names of the territory—they do not change the names of cities on a map. For a domain model, this means *use the vocabulary of the domain when naming conceptual classes and attributes.* For example, if developing a model for a library, name the customer a *"Borrower"* or *"Patron"*—the terms used by the library staff.
- A mapmaker deletes things from a map if they are not considered relevant to the purpose of the map; for example, topography or populations need not be shown. Similarly, a domain model may exclude conceptual classes in the problem domain not pertinent to the requirements. For example, we may exclude *Pen* and *PaperBag* from our domain model (for the current set of requirements) since they do not have any obvious noteworthy role.
- A mapmaker does not show things that are not there, such as a mountain that does not exist. Similarly, the domain model should exclude things *not* in the problem domain under consideration.

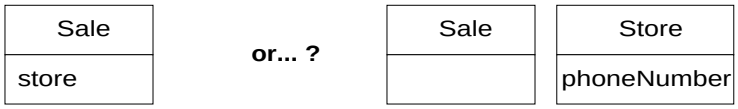
The principle is also named the *Use the Domain Vocabulary* strategy [Coad95].

A Common Mistake in Identifying Conceptual Classes

Perhaps the most common mistake when creating a domain model is to represent something as an attribute when it should have been a concept. A rule of thumb to help prevent this mistake is:

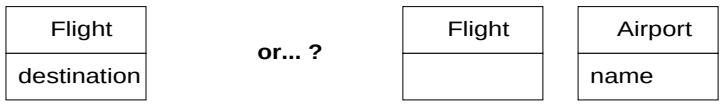
> If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.

As an example, should *store* be an attribute of *Sale,* or a separate conceptual class *Store?*



In the real world, a store is not considered a number or text—the term suggests a legal entity, an organization, and something occupies space. Therefore, *Store* should be a concept.

As another example, consider the domain of airline reservations. Should *destination* be an attribute *of Flight,* or a separate conceptual class *Airport*?



In the real world, a destination airport is not considered a number or text—it is a massive thing that occupies space. Therefore, *Airport* should be a concept.

If in doubt, make it a separate concept. Attributes should be fairly rare in a domain model.

10.5 Resolving Similar Conceptual Classes—Register vs. "POST"

POST stands for point-of-sale terminal. In computerese, a terminal is any end-point device in a system, such as a client PC, a wireless networked PDA, and so forth. In earlier times, long before POSTs, a store maintained a *register*—a book that logged sales and payments. Eventually, this was automated in a mechanical "cash register." Today, a POST fulfills the role of the register (see Figure 10.6).

A register is a thing that records sales and payments, but so is a POST. However, the term *register* seems somewhat more abstract and less implementation oriented than *POST.* So, in the domain model, should the symbol *Register* be used instead *of POST?*

First, as a rule of thumb, a domain model is not absolutely correct or wrong, but more or less useful; it is a tool of communication.

By the mapmaker principle, *"POST"* is a term familiar in the territory, so it is a useful symbol from the point of view of familiarity and communication. By the goal of creating models that represent abstractions and are implementation independent, *Register* is appealing and useful.[5] *Register* may be fairly considered to represent both the conceptual class of a place to register sales, and/or an abstraction of various kinds of terminals, such as a POST.

Both choices have merit; *Register* has been chosen in this case study somewhat arbitrarily, but *POST* would also have been understandable to the stakeholders.

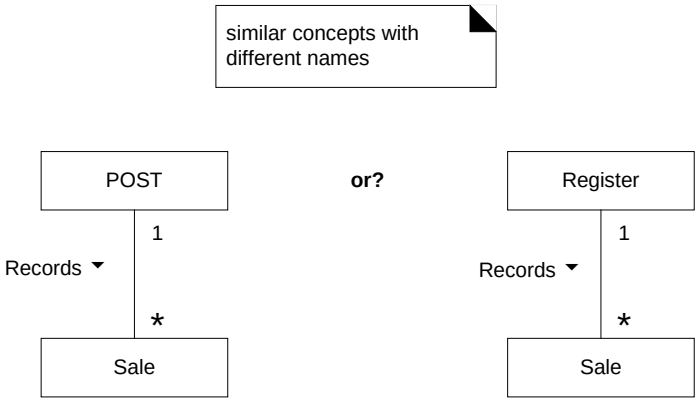


Figure 10.6 POST and register are similar conceptual classes.

10.6 Modeling the *Unreal* World

Some software systems are for domains that find very little analogy in natural or business domains; software for telecommunications is an example. It is still possible to create a domain model in these domains, but it requires a high degree of abstraction and stepping back from familiar designs.

For example, here are some candidate conceptual classes related to a telecommunication switch: *Message, Connection, Port, Dialog, Route, Protocol.*

10.7 Specification or Description Conceptual Classes

The following discussion may at first seem related to a rare, highly specialized issue. However, it turns out that the need for specification conceptual classes (as will be defined) is common in many domain models. Thus, it is emphasized.

5. Note that in earlier times a *register* was just one possible implementation of how to record sales. The term has acquired a generalized meaning over time.

Assume the following:

- An *Item* instance represents a physical item in a store; as such, it may even have a serial number.
- An *Item* has a description, price, and itemID, which are not recorded anywhere else.
- Everyone working in the store has amnesia.
- Every time a real physical item is sold, a corresponding software instance of *Item* is deleted from "software land."

With these assumptions, what happens in the following scenario?

There is strong demand for the popular new vegetarian burger—ObjectBurger. The store sells out, implying that all *Item* instances of ObjectBurgers are deleted from computer memory.

Now, here is the heart of the problem: If someone asks, "How much do ObjectBurgers cost?", no one can answer, because the memory of their price was attached to inventoried instances, which were deleted as they were sold.

Notice also that the current model, if implemented in software *as* described, has duplicate data and is space-inefficient because the description, price, and itemID are duplicated for every *Item* instance of the same product.

The Need for Specification or Description Conceptual Classes

The preceding problem illustrates the need for a concept of objects that are specifications or descriptions of other things. To solve the *Item* problem, what *is* needed is a *ProductSpecification* (or *ItemSpecification, ProductDescription, ...*) conceptual class that records information about items. A *ProductSpecification* does not represent an *Item*, it represents a description of information *about* items. Note that even if all inventoried items are sold and their corresponding *Item* software instances are deleted, the *ProductSpecifications* still remain.

Description or specification objects are strongly related to the things they describe. In a domain model, it is common to state that an *XSpecification Describes an X* (see Figure 10.7).

The need for specification conceptual classes is common in sales and product domains. It is also common in manufacturing, where a *description* of a manufactured thing is required that is distinct from the thing itself. Time and space have been taken in motivating specification conceptual classes because they are very common; it is not a rare modeling concept.

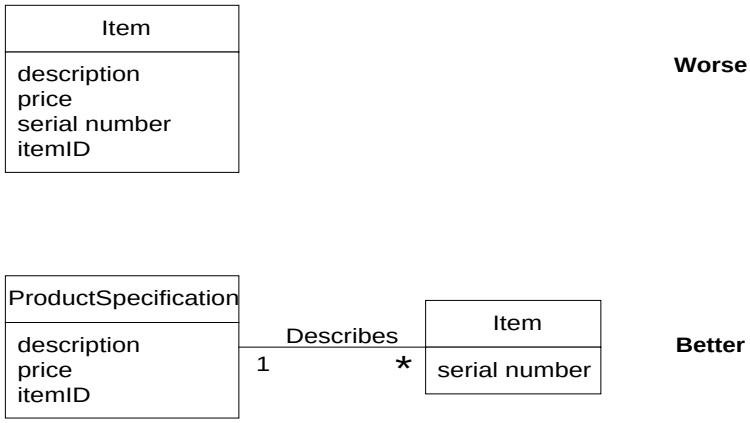


Figure 10.7 Specifications or descriptions about other things. The "*" means a multiplicity of "many." It indicates that one *ProductSpecification* may describe many (*) *Items.*

When Are Specification Conceptual Classes Required?

The following guideline suggests when to use specifications:

Add a specification or description conceptual class (for example, *ProductSpecification)* when:

- There needs to be a description about an item or service, independent of the current existence of any examples of those items or services.
- Deleting instances of things they describe (for example, *Item)* results in a loss of information that needs to be maintained, due to the incorrect association of information with the deleted thing.
- It reduces redundant or duplicated information.

Another Specification Example

As another example, consider an airline company that suffers a fatal crash of one of its planes. Assume that all the flights are cancelled for six months pending completion of an investigation. Also assume that when flights are cancelled, their corresponding *Flight* software objects are deleted from computer memory. Therefore, after the crash, all *Flight* software objects are deleted.

If the only record of what airport a flight goes to is in the *Flight* software instances, which represent specific flights for a particular date and time, then there is no longer a record of what flight routes the airline has.

To solve this problem, a *FlightDescription* (or *FlightSpecification)* is required that describes a flight and its route, even when a particular flight is not scheduled (see Figure 10.8).

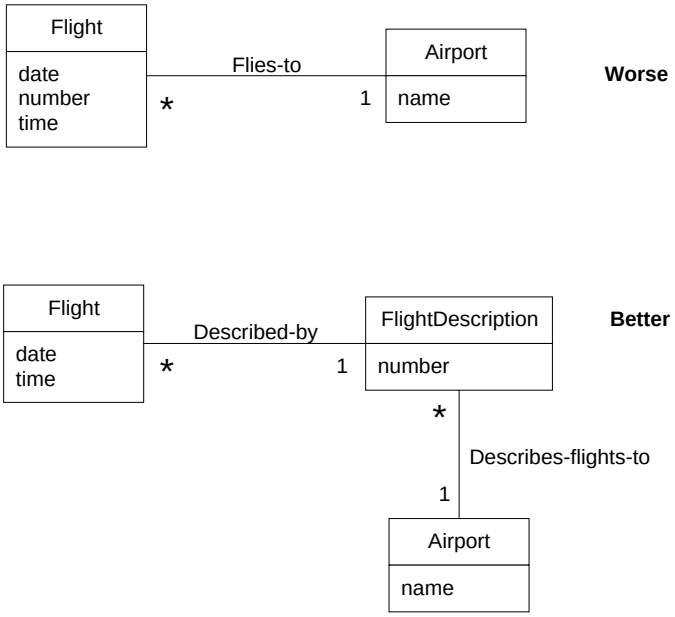


Figure 10.8 Specifications about other things.

Descriptions of Services

Note that the prior example is about a service (a flight) rather than a good (such as a veggieburger). Descriptions of services or service plans are commonly needed.

As another example, a mobile phone company sells packages such as "bronze," "gold," and so forth. It is necessary to have the concept of a description of the package (a kind of service plan describing rates per minute, wireless Internet content, the cost, and so forth) separate from the concept of an actual sold package (such as "gold package sold to Craig Larman on Jan 1, 2002 at $55 per month"). Marketing needs to define and record this service plan or *MobileCommunicationsPackageDescription* before any are sold.

10.8 UML Notation, Models, and Methods: Multiple Perspectives

The UP defines something called a Domain Model, which is illustrated with UML notation. However, there is no term "Domain Model" to be found in the official UML documentation. This points to an important insight:

> The UML simply describes raw diagram types, such as class diagrams and sequence diagrams. It does not superimpose a method or modeling perspective on these. Rather, a process (such as the UP) applies raw UML in the context of methodologist-defined models.

For example, raw UML class diagramming notation can be used to create pictures of domain conceptual classes (a domain model), software classes, relational database tables, and so forth.

Thus, do not confuse the basic UML diagram notation with its application to visualizing various kinds of models defined by methodologists (see Figure 10.9). This point applies not only to UML class diagrams, but to most UML notation.

As another example of raw diagrams being interpreted differently in different models, UML sequence diagrams can be used to illustrate messaging between software objects (as in the UP Design Model), or interaction between people and parties in the real world (as in the UP Business Object Model).

This insight was emphasized in the Syntropy object-oriented method [CD94], and reiterated by Martin Fowler in *UML Distilled* [FSOO]. That is, the same diagramming notation may be used for three perspectives and types of models:

1. **Essential or conceptual perspective**—the diagrams are interpreted as describing things in the real world or domain of interest.
2. **Specification perspective**—the diagrams (using the same notation as for essential models) are interpreted as describing software abstractions or components with specifications and interfaces, but no commitment to a particular implementation (for example, not specifically a class in C# or Java).
3. **Implementation perspective**—the diagrams (using the same notation as for essential models) are interpreted as describing software implementations in a particular technology and language (such as Java).

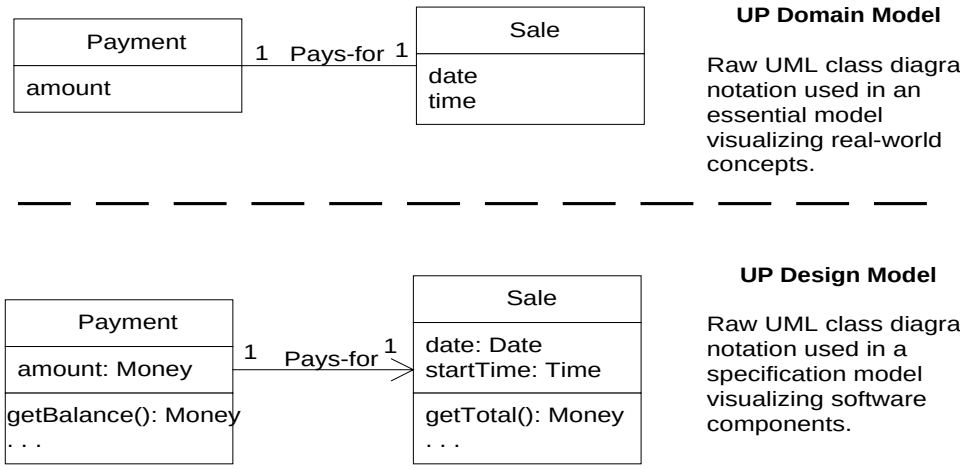


Figure 10.9 Raw UML notation is applied in different perspectives and models defined by a process or method.

Superimposing Terminology: UML vs. Methods

In the raw UML, the rectangular boxes shown in Figure 10.9 are called **classes,** but note that in the UML, this term encompasses a variety of phenomenon—physical things, software things, events, and so forth.[6] A process or method will superimpose alternative terminology on top of the UML. For example, in the UP, when the UML boxes are drawn in the Domain Model, they may be called **domain concepts** or **conceptual classes;** the Domain Model offers a conceptual perspective. In the UP, when UML boxes are drawn in the Design Model, they are officially called **design classes;** the Design Model offers a specification or implementation perspective, as desired by the modeler.

Regardless of the definition, the bottom line is that it is useful to distinguish between the perspective of an analyst looking at real-world concepts such as a sale (a conceptual perspective), and software designers specifying software components such as a *Sale* software class (a specification or implementation perspective).

The UML can be used to illustrate both perspectives with very similar notation and terminology, so it is important to bear in mind which perspective is being taken.

A UML class is a special case of the very general UML model element **classifier**—something with structural features and/or behavior, including classes, actors, interfaces, and use cases.

To keep things clear, this book will use class-related terms as follows, which is consistent with the UML and the UP:

- Conceptual class — real-world concept or thing. A conceptual or essential perspective. The UP Domain Model contains conceptual classes.
- Software class — a class representing a specification or implementation perspective of a software component, regardless of the process or method.
- Design class — a member of the UP Design Model. It is a synonym for software class, but for some reason I wish to emphasize that it is a class in the Design Model. The UP allows a design class to be either a specification or implementation perspective, as desired by the modeler.
- Implementation class — a class implemented in an object-oriented language such as Java.
- Class — as in the UML, the general term representing either a real-world thing (a conceptual class) or software thing (a software class).

10.9 Lowering the Representational Gap

Please consider Figure 10.10. Why do books and educators discussing object design common only show the use of software classes whose names reflect domain vocabulary? Why choose a software class name such as *Sale,* and what does a *Sale* do?

Simply, choosing names that reflect the domain vocabulary *(Sale)* enhances quick comprehension and provides a clue as to what to expect from the chunk of code in a *Sale* software class. We have a mental or domain model of the domain in question (for example, a store selling things). In the real world, we know that a sale has a date. Consequently, if we create a Java class named *Sale,* and give it the responsibility of knowing about a real sale and its date, then the Java class *Sale* somewhat corresponds to our mental or domain model of the real domain; that is, it appeals to our "intuitions" of the domain.

The Domain Model provides a visual dictionary of the domain vocabulary and concepts from which to draw inspiration for the naming of some things in the software design.

This relates to the issue of **representational gap** or semantic gap—the gap between our mental model of the domain and its representation in software.

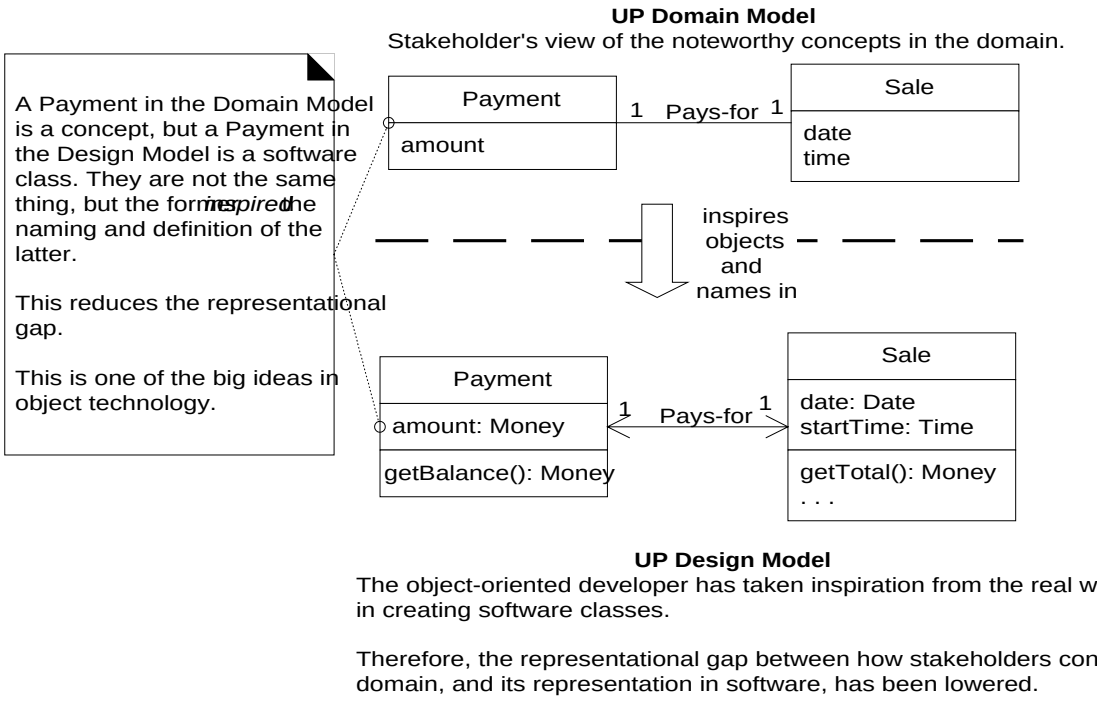


Figure 10.10 In object design and programming it is common to create software classes whose names and information is inspired from the real world domain.

At one extreme, we could directly program the NextGen POS application in raw binary code to invoke the processor instruction set. We understand that the gap in representations is huge, and there will be a real cost—albeit hard to quantify—in software with such a large representational gap because it is hard to comprehend or relate to the problem domain. Closer to the other end of the spectrum are object technologies that allow us to chunk code into classes whose names reflect the kind of chunking we perceive in the domain. In the real world we perceive a "chunk" (or event) called a sale, so in software land we have a software class called *Sale.* This closer one-to-one mapping between the domain vocabulary and our software vocabulary and its chunking reduces the representational gap. This speeds comprehension of existing code (because it works in ways we expect, knowing the domain) and suggests "natural" ways to extend the code in ways that similarly correspond to the domain, or appeal to our intuitions of the domain. Put simply, the software model reminds us of the conceptual or mental model, and works in predictable ways.

There is a practical advantage to software models that reduce the representational gap. Most software engineers know this is true, even if it is hard to quantify. Indeed, a proof of this is that Java obfuscators make source code hard to practically reverse-engineer from bytecode by changing the names of Java

classes and methods so they are unintelligible, and thus no longer appeal to our intuitions of the domain, even though the control and data structures are unchanged.

Of course, object technology is also of value because it can support the design of elegant, loosely coupled systems that scale and extend easily, as will be explored in the remainder of the book. A lowered representational gap is useful, but arguably secondary to the advantage of objects to support ease of change and extension, and their support to manage and hide complexity.

10.10 Example: The NextGen POS Domain Model

The list of conceptual classes generated for the NextGen POS domain may be represented graphically (see Figure 10.11) to show the start of the Domain Model.

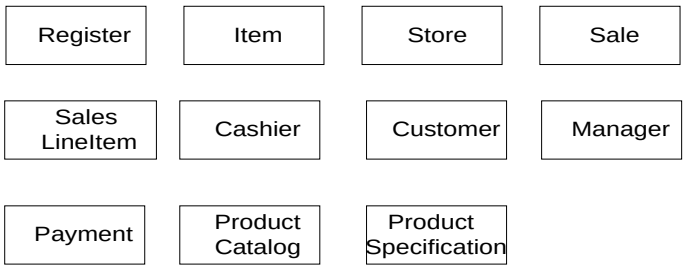


Figure 10.11 Initial Domain Model.

Consideration of attributes and associations for the Domain Model will be deferred to subsequent chapters.

10.11 Domain Models Within the UP

As suggested in the example of Table 10.2, a Domain Model is usually both started and completed in elaboration.

Inception

Domain models are not strongly motivated in inception, since inception's purpose is not to do a serious investigation, but rather to decide if the project is worth deeper investigation in an elaboration phase.

Discipline	Artifact Iteration→	Incep. Il	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	*Domain Model*		s		
Requirements	Use-Case Model (SSDs)	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 10.2 Sample UP artifacts and timing, s - start; r - refine

Elaboration

The Domain Model is primarily created during elaboration iterations, when the need is highest to understand the noteworthy concepts and map some to software classes during design work.

Although ironically a significant number of pages will be devoted to explaining domain object modeling, in experienced hands the development of a (partial, incrementally growing) domain model in each iteration should only take a few hours. This is further shortened by the use of predefined analysis patterns.

The UP Business Object Model vs. Domain Model

The UP Domain Model is an official variation of the less common UP Business Object Model (BOM). The UP BOM—not to be confused with how other people or methods may define a BOM, which is a widely used term with different meanings—is a kind of enterprise model used to describe the entire business. It may be used when doing business process engineering or reengineering, independent of any one software application (such as the NextGen POS). To quote:

> [The UP BOM] serves as an abstraction of how business workers and business entities need to be related and how they need to collaborate in order to perform the business. [RUP]

The BOM is represented with several different diagrams (class, activity, and sequence) that illustrate how the entire enterprise runs (or should run). It is most useful if doing enterprise-wide business process engineering, but that is a less common activity than creating a single software application.

Consequently, the UP defines the Domain Model as the more commonly created subset artifact or specialization of the BOM. To quote:

> You can choose to develop an "incomplete" business object model, focusing on explaining "things" and products important to a domain. ... This is often referred to as a domain model. [RUP]

10.12 Further Readings

Odell's *Object-Oriented Methods: A Foundation* provides a solid introduction to conceptual domain modeling. Cook and Daniel's *Designing Object Systems* is also useful.

Fowler's *Analysis Patterns* offers worthwhile patterns in domain models, and is definitely recommended. Another good book that describes patterns in domain models is Hay's *Data Model Patterns: Conventions of Thought.* Advice from data modeling experts who understand the distinction between pure conceptual models and database schema models can be very useful for domain object modeling.

Java Modeling in Color with UML [CDL99] has more relevant domain modeling advice than the title suggests. The authors identify common patterns in related types and their associations; the color aspect is really a visualization of the common categories of these types, such as *descriptions* (blue), *roles* (yellow), and *moment-intervals* (pink). Color is used to aid in seeing the patterns.

Since the original work by Abbot, linguistic analysis has acquired more sophisticated techniques for object-oriented analysis, generally called natural language modeling, or a variant. See [Moreno97] as an example.

10.13 UP Artifacts

Artifact influence emphasizing the Domain Model is shown in Figure 10.12.

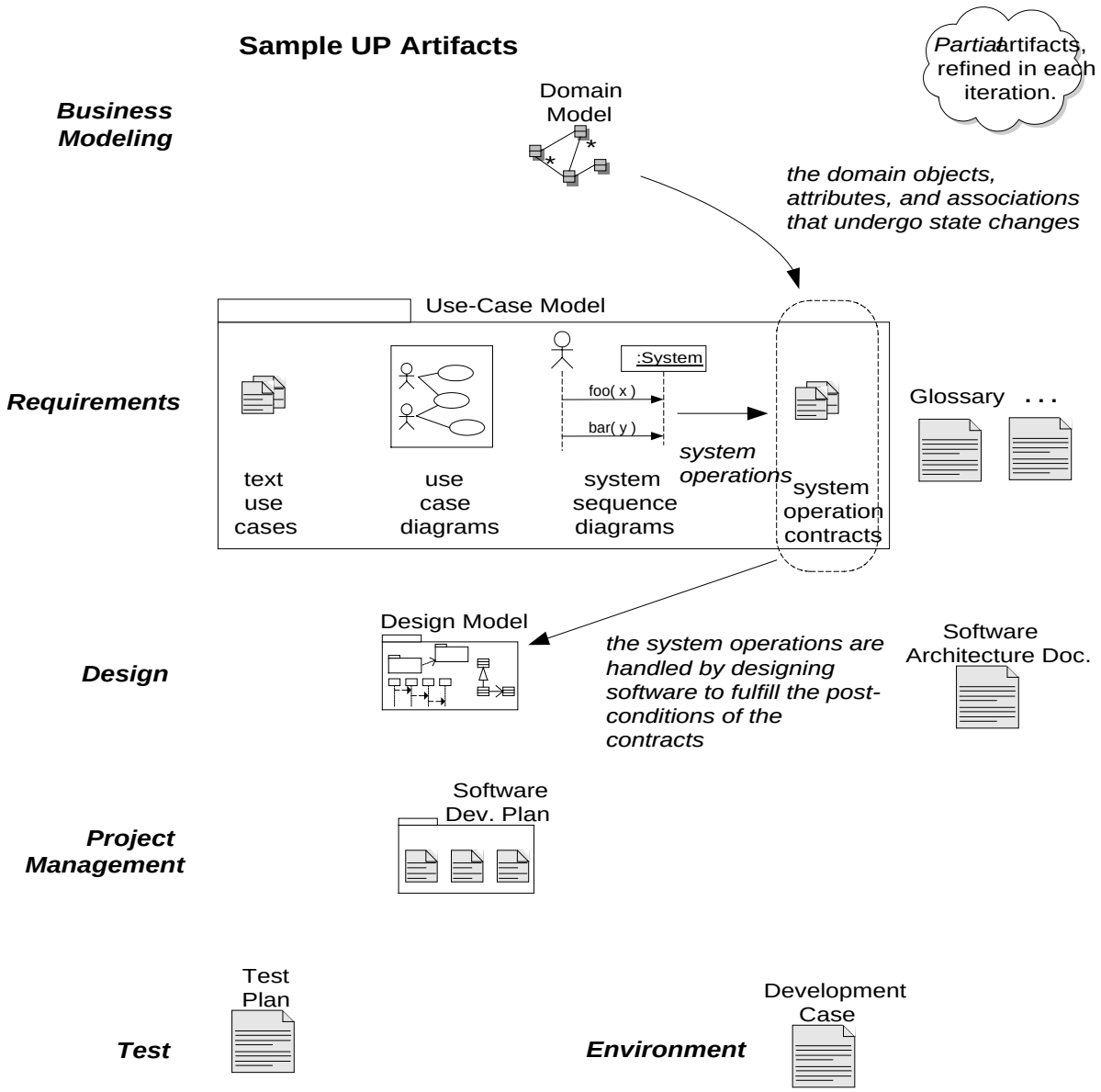


Figure 10.12 Sample UP artifact influence.