Chapter
9

DOMAIN MODELS

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.
- Create an initial domain model.
- Model appropriate attributes and associations.

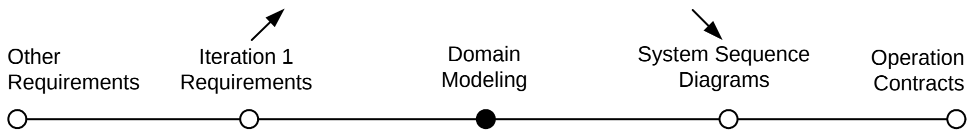
Introduction

more advanced domain modeling p. 507

A domain model is the most important—and classic—model in OO *analysis*.[1] It illustrates noteworthy concepts in a domain. It can act as a source of inspiration for designing some software objects and will be an input to several artifacts explored in the case studies. This chapter also shows the value of OOA/D knowledge over UML notation; the basic notation is trivial, but there are subtle modeling guidelines for a useful model—expertise can take weeks or months. This chapter explores basic skills in creating domain models.

What's Next?

Having scoped the work for iteration-1, this chapter explores a partial domain model. The next examines the specific operations upon the system that are implied in the use case scenarios under design for this iteration.



1. Use cases are an important requirements analysis artifact, but are not *object*-oriented. They emphasize an activity view.

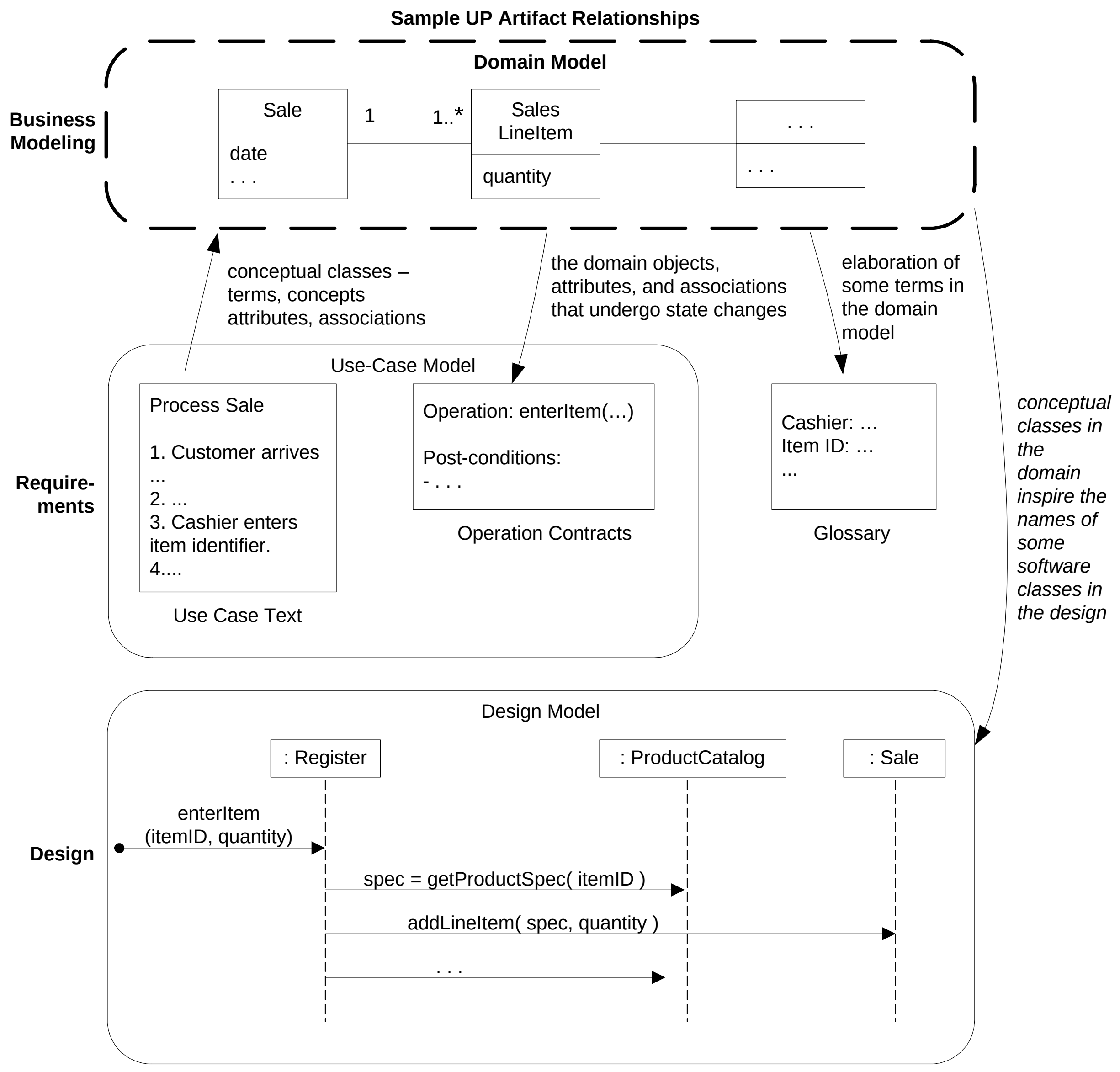

Sample UP Artifact Relationships
Domain Model
Business Modeling
Sale
date
. . .
1
1..*
Sales LineItem
quantity
. . .
. . .
conceptual classes – terms, concepts attributes, associations
the domain objects, attributes, and associations that undergo state changes
elaboration of some terms in the domain model
conceptual classes in the domain inspire the names of some software classes in the design
Use-Case Model
Require-ments
Process Sale
1. Customer arrives
...
2. ...
3. Cashier enters item identifier.
4....
Use Case Text
Operation: enterItem(…)
Post-conditions:
- . . .
Operation Contracts
Cashier: …
Item ID: …
...
Glossary
Design Model
Design
: Register
: ProductCatalog
: Sale
enterItem (itemID, quantity)
spec = getProductSpec(itemID)
addLineItem(spec, quantity)
. . .

Figure 9.1 Sample UP artifact influence.

As with all things in an agile modeling and UP spirit, a domain model is optional. UP artifact influence emphasizing a domain model is shown in Figure 9.1. *Bounded by* the use case scenarios under development for the current iteration, the domain model can be evolved to show related noteworthy concepts. The related use case concepts and insight of experts will be input to its creation. The model can in turn influence operation contracts, a glossary, and the Design Model, especially the software objects in the **domain layer** of the Design Model.

domain layer p. 136

9.1 Example

Figure 9.2 shows a partial domain model drawn with UML **class diagram** notation. It illustrates that the **conceptual classes** of *Payment* and *Sale* are significant in this domain, that a *Payment* is related to a *Sale* in a way that is meaningful to note, and that a *Sale* has a date and time, information attributes we care about.

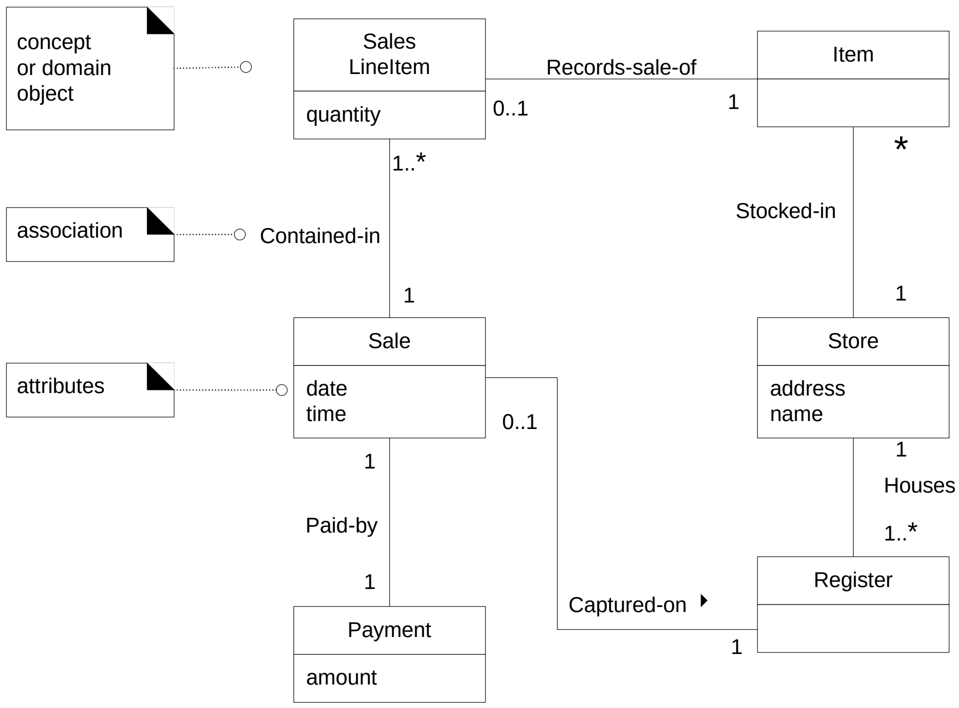


Figure 9.2 Partial domain model—a visual dictionary.

conceptual perspective p. 12

Applying the UML class diagram notation for a domain model yields a *conceptual perspective* model.

Identifying a rich set of conceptual classes is at the heart of OO analysis. If it is done with skill and *short* time investment (say, no more than a few hours in each early iteration), it usually pays off during design, when it supports better understanding and communication.

Guideline

Avoid a waterfall-mindset big-modeling effort to make a thorough or "correct" domain model—it won't ever be either, and such over-modeling efforts lead to *analysis paralysis*, with little or no return on the investment.

9.2 What is a Domain Model?

The quintessential *object*-oriented analysis step is the decomposition of a domain into noteworthy concepts or objects.

A **domain model** is a *visual* representation of conceptual classes or real-situation objects in a domain [MO95, Fowler96]. Domain models have also been called **conceptual models** (the term used in the first edition of this book), **domain object models**, and **analysis object models**.[2]

Definition

In the UP, the term "Domain Model" means a representation of real-situation conceptual classes, not of software objects. The term does *not* mean a set of diagrams describing software classes, the domain layer of a software architecture, or software objects with responsibilities.

The UP defines the Domain Model[3] as one of the artifacts that may be created in the Business Modeling discipline. More precisely, the UP Domain Model is a specialization of the UP **Business Object Model** (BOM) "focusing on explaining 'things' and products important to a business domain" [RUP]. That is, a Domain Model focuses on one domain, such as POS related things. The more broad BOM, not covered in this introductory text and not something I encourage creating (because it can lead to too much up-front modeling), is an expanded, often very large and difficult to create, multi-domain model that covers the *entire* business and all its sub-domains.

Applying UML notation, a domain model is illustrated with a set of **class diagrams** in which no operations (method signatures) are defined. It provides a *conceptual perspective*. It may show:

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

Definition: Why Call a Domain Model a "Visual Dictionary"?

Please reflect on Figure 9.2 for a moment. See how it visualizes and relates words or concepts in the domain. It also shows an *abstraction* of the conceptual

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" or terms is used to emphasize it as an official model name defined in the UP, versus the general well-known concept of "domain models."

classes, because there are many other things one could communicate about registers, sales, and so forth.

The information it illustrates (using UML notation) could alternatively have been expressed in plain text (in the UP Glossary). But it's easy to understand the terms and especially their relationships in a visual language, since our brains are good at understanding visual elements and line connections.

Therefore, the domain model is a *visual dictionary* of the noteworthy abstractions, domain vocabulary, and information content of the domain.

Definition: Is a Domain Model a Picture of Software Business Objects?

A UP Domain Model, as shown in Figure 9.3, is a visualization of things in a real-situation domain of interest, *not* of software objects such as Java or C# classes, or software objects with responsibilities (see Figure 9.4). 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]

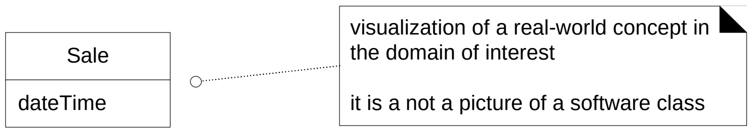

Sale
dateTime
visualization of a real-world concept in the domain of interest
it is a not a picture of a software class

Figure 9.3 A domain model shows real-situation conceptual classes, not software classes.

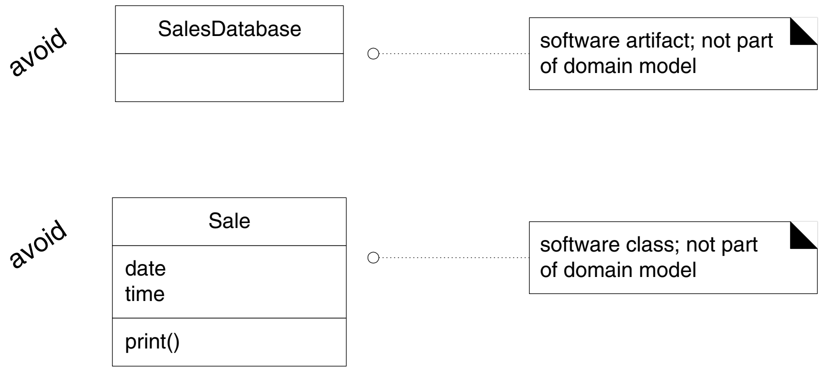

avoid
SalesDatabase
software artifact; not part of domain model
avoid
Sale
date
time
print()
software class; not part of domain model

Figure 9.4 A domain model does not show software artifacts or classes.

4. In object modeling, we usually speak of responsibilities related to software objects. And methods are purely a software concept. But, the domain model describes real-situation concepts, not software objects. Considering object responsibilities during *design* work is very important; it is just not part of this model.

Definition: What are Two Traditional Meanings of "Domain Model"?

In the UP and thus this chapter, "Domain Model" is a conceptual perspective of objects in a real situation of the world, not a software perspective. But the term is overloaded; it also has been used (especially in the Smalltalk community where I did most of my early OO development work in the 1980s) to mean "the domain layer of software objects." That is, the layer of software objects below the presentation or UI layer that is composed of **domain objects**—software objects that represent things in the problem domain space with related "business logic" or "domain logic" methods. For example, a *Board* software class with a *getSquare* method.

Which definition is correct? Well, all of them! The term has long established uses in different communities to mean different things.

I've seen lots of confusion generated by people using the term in different ways, without explaining which meaning they intend, and without recognizing that others may be using it differently.

In this book, I'll usually write **domain layer** to indicate the second software-oriented meaning of domain model, as that's quite common.

Definition: What are 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 9.5).

- **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 (English) 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 sale instances in the universe.

Definition: Are Domain and Data Models the Same Thing?

A domain model is not a **data model** (which by definition shows persistent data to be stored somewhere), so do not exclude a class simply because the requirements don't 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. For example, it's valid to have attributeless conceptual classes, or conceptual classes that have a purely behavioral role in the domain instead of an information role.

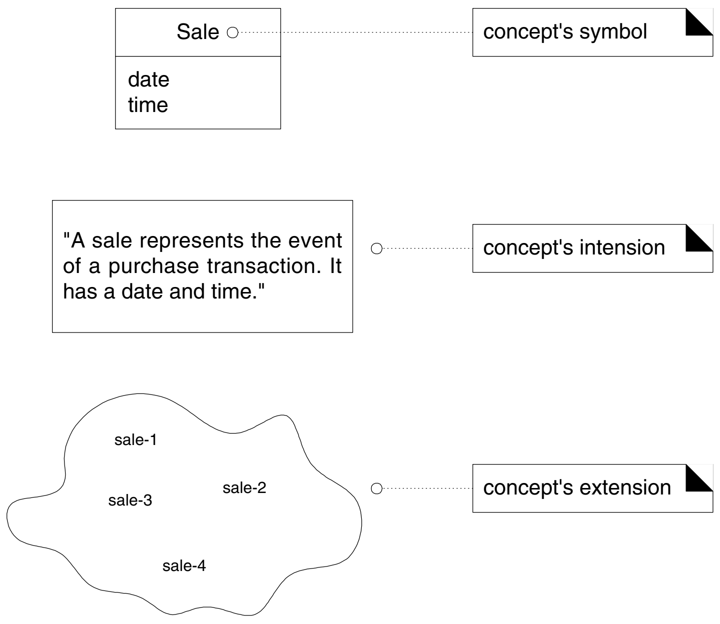


Figure 9.5 A conceptual class has a symbol, intension, and extension.

9.3 Motivation: Why Create a Domain Model?

I'll share a story that I've experienced many times in OO consulting and coaching. In the early 1990s I was working with a group developing a funeral services business system in Smalltalk, in Vancouver (you should see the domain model!). Now, I knew almost nothing about this business, so one reason to create a domain model was so that I could start to understand their key concepts and vocabulary.

domain layer p. 206

We also wanted to create a **domain layer** of Smalltalk objects representing business objects and logic. So, we spent perhaps one hour sketching a UML-ish (actually OMT-ish, whose notation inspired UML) domain model, not worrying about software, but simply identifying the key terms. Then, those terms we sketched in the domain model, such as *Service* (like flowers in the funeral room, or playing "You Can't Always Get What You Want"), were also used as the names of key software classes in our domain layer implemented in Smalltalk.

This similarity of naming between the domain model and the domain layer (a real "service" and a Smalltalk *Service*) supported a lower gap between the software representation and our mental model of the domain.

Motivation: Lower Representational Gap with OO Modeling

This is a key idea in OO: Use software class names in the domain layer inspired from names in the domain model, with objects having domain-familiar information and responsibilities. Figure 9.6 illustrates the idea. This supports a **low representational gap** between our mental and software models. And that's not just a philosophical nicety—it has a practical time-and-money impact. For example, here's a source-code payroll program written in 1953:

1000010101000111101010101010001010101010101111010101 …

As computer science people, we know it runs, but the gap between this software representation and our mental model of the payroll domain is huge; that profoundly affects comprehension (and modification) of the software. OO modeling can lower that gap.

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 objects have in supporting ease of change and extension, and managing and hiding complexity.

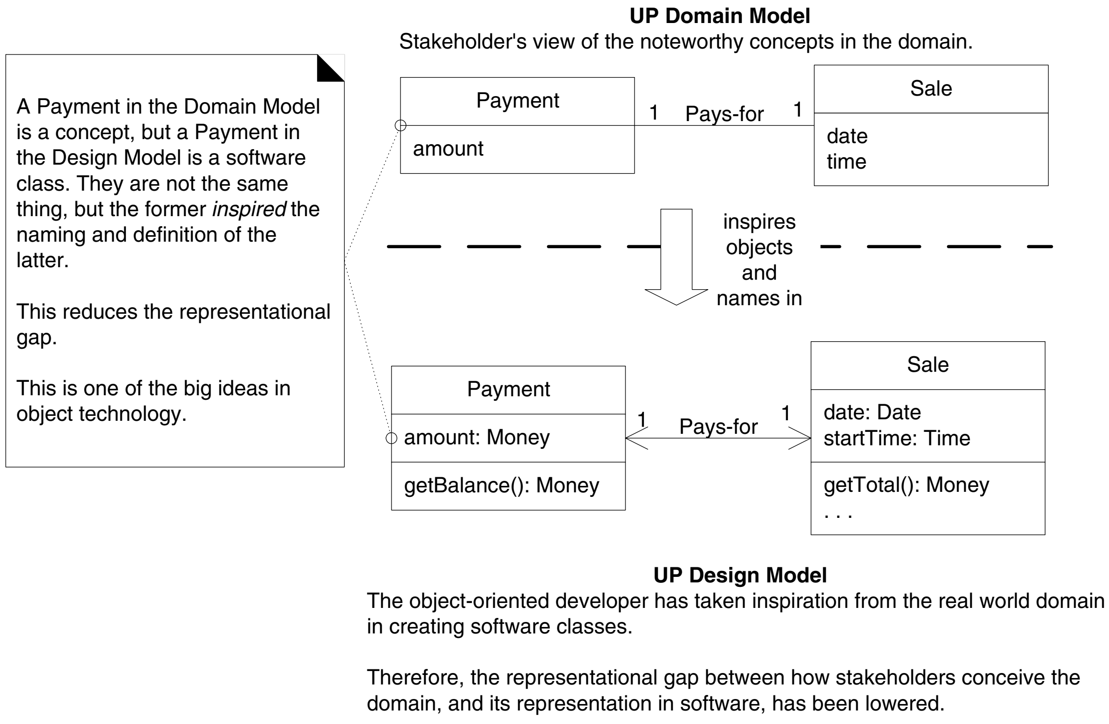

UP Domain Model
Stakeholder's view of the noteworthy concepts in the domain.
A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.
This reduces the representational gap.
This is one of the big ideas in object technology.
Payment
amount
1
Pays-for
1
Sale
date
time
inspires objects and names in
Payment
amount: Money
getBalance(): Money
1
Pays-for
1
Sale
date: Date
startTime: Time
getTotal(): Money
. . .
UP Design Model
The object-oriented developer has taken inspiration from the real world domain in creating software classes.
Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Figure 9.6 Lower representational gap with OO modeling.

9.4 Guideline: How to Create a Domain Model?

Bounded by the current iteration requirements under design:

1. Find the conceptual classes (see a following guideline).
2. Draw them as classes in a UML class diagram.
3. Add associations and attributes. See p. 149 and p. 158.

9.5 Guideline: How to Find Conceptual Classes?

Since a domain model shows conceptual classes, a central question is: How do I find them?

What are Three Strategies to Find Conceptual Classes?

1. Reuse or modify existing models. This is the first, best, and usually easiest approach, and where I will start if I can. There are published, well-crafted domain models and data models (which can be modified into domain models) for many common domains, such as inventory, finance, health, and so forth. Example books that I'll turn to include *Analysis Patterns* by Martin Fowler, *Data Model Patterns* by David Hay, and the *Data Model Resource Boo*k (volumes 1 and 2) by Len Silverston.
2. Use a category list.
3. Identify noun phrases.

Reusing existing models is excellent, but outside our scope. The second method, using a category list, is also useful.

Method 2: Use a Category List

We can kick-start the creation of a domain model by making a list of candidate conceptual classes. Table 9.1 contains many common categories that are usually worth considering, with an emphasis on business information system needs. The guidelines also suggest some priorities in the analysis. Examples are drawn from the 1) POS, 2) Monopoly, and 3) airline reservation domains.

Conceptual Class Category	Examples
business transactions *Guideline*: These are critical (they involve money), so start with transactions.	*Sale, Payment* *Reservation*
transaction line items *Guideline*: Transactions often come with related line items, so consider these next.	*SalesLineItem*
product or service related to a transaction or transaction line item *Guideline*: Transactions are *for* something (a product or service). Consider these next.	*Item* *Flight, Seat, Meal*
where is the transaction recorded? *Guideline*: Important.	*Register, Ledger* *FlightManifest*
roles of people or organizations related to the transaction; actors in the use case *Guideline*: We usually need to know about the parties involved in a transaction.	*Cashier, Customer, Store* *MonopolyPlayer* *Passenger, Airline*
place of transaction; place of service	*Store* *Airport, Plane, Seat*
noteworthy events, often with a time or place we need to remember	*Sale, Payment* *MonopolyGame* *Flight*
physical objects *Guideline*: This is especially relevant when creating device-control software, or simulations.	*Item, Register* *Board, Piece, Die* *Airplane*
descriptions of things *Guideline*: See p. 147 for discussion.	*ProductDescription* *FlightDescription*

Conceptual Class Category	Examples
catalogs *Guideline*: Descriptions are often in a catalog.	*ProductCatalog* *FlightCatalog*
containers of things (physical or information)	*Store, Bin* *Board* *Airplane*
things in a container	*Item* *Square (in a Board)* *Passenger*
other collaborating systems	*CreditAuthorizationSystem* *AirTrafficControl*
records of finance, work, contracts, legal matters	*Receipt, Ledger* *MaintenanceLog*
financial instruments	*Cash, Check, LineOfCredit* *TicketCredit*
schedules, manuals, documents that are regularly referred to in order to perform work	*DailyPriceChangeList* *RepairSchedule*

Table 9.1 Conceptual Class Category List.

Method 3: 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.5

Guideline

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

5. Linguistic analysis has become more sophisticated; it also goes by the name **natural language modeling**. See [Moreno97] for example.

Nevertheless, linguistic analysis 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? Some are in the use cases. Others are in other documents, or the minds of experts. In any event, use cases are one rich source to mine for 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 simply attributes of conceptual classes. See p. 160 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.

9.6 Example: Find and Draw Conceptual Classes

Case Study: POS Domain

iteration-1 requirements p. 124

From the category list and noun phrase analysis, a list is generated of candidate conceptual classes for the domain. Since this is a business information system, I'll focus first on the category list guidelines that emphasize business transactions and their relationship with other things. The list is constrained to the requirements and simplifications currently under consideration for iteration-1, the basic cash-only scenario of *Process Sale*.

Sale	*Cashier*
CashPayment	*Customer*
SalesLineItem	*Store*
Item	*ProductDescription*
Register	*ProductCatalog*
Ledger	

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, different modelers will produce similar lists.

In practice, I don't create a text list first, but immediately draw a UML class diagram of the conceptual classes as we uncover them. See Figure 9.7.

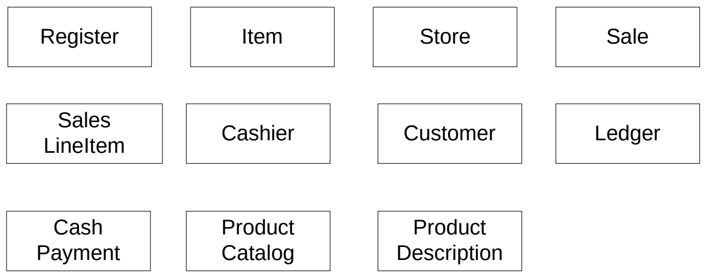


Figure 9.7 Initial POS domain model.

Adding the associations and attributes is covered in later sections.

Case Study: Monopoly Domain

iteration-1 requirements p. 124

From the Category List and noun phrase analysis, I generate a list of candidate conceptual classes for the iteration-1 simplified scenario of *Play a Monopoly Game* (see Figure 9.8). Since this is a simulation, I emphasize the noteworthy tangible, physical objects in the domain.

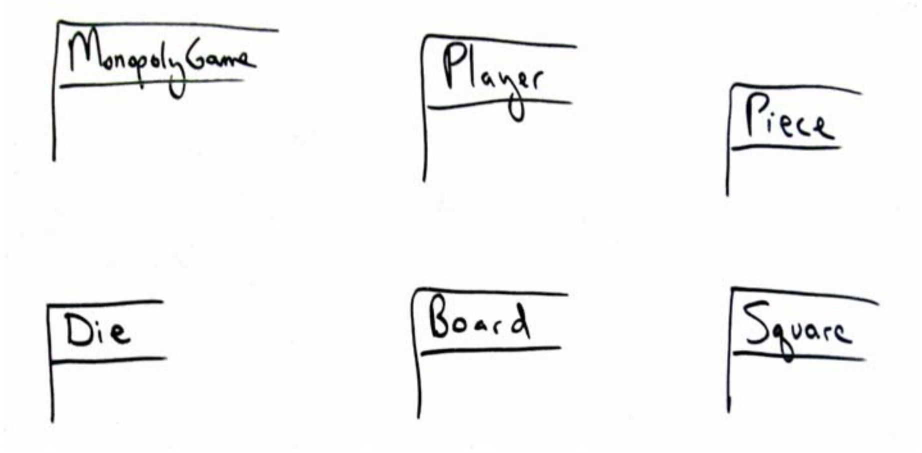


Figure 9.8 Initial Monopoly domain model.

9.7 Guideline: Agile Modeling—Sketching a Class Diagram

Notice the sketching style in the UML class diagram of Figure 9.8—keeping the bottom and right sides of the class boxes open. This makes it easier to grow the classes as we discover new elements. And although I've grouped the class boxes for compactness in this book diagram, on a whiteboard I'll spread them out.

9.8 Guideline: Agile Modeling—Maintain the Model in a Tool?

It's normal to miss significant conceptual classes during early domain modeling, and to discover them later during design sketching or programming. If you are taking an agile modeling approach, the purpose of creating a domain model is to quickly understand and communicate a rough approximation of the key concepts. Perfection is not the goal, and agile models are usually discarded shortly after creation (although if you've used a whiteboard, I recommend taking a digital snapshot). From this viewpoint, there is no motivation to maintain or update the model. But that doesn't mean it's wrong to update the model.

If someone wants the model maintained and updated with new discoveries, that's a good reason to redraw the whiteboard sketch within a UML CASE tool, or to originally do the drawing with a tool and a computer projector (for others to see the diagram easily). But, ask yourself: Who is going to use the updated model, and why? If there isn't a practical reason, don't bother. Often, the evolving *domain layer* of the software hints at most of the noteworthy terms, and a long-life OO analysis domain model doesn't add value.

9.9 Guideline: Report Objects—Include 'Receipt' in the Model?

Receipt is a noteworthy term in the POS domain. But perhaps it's only a *report* of a sale and payment, and thus duplicate information. Should it be in the domain model?

Here are some factors to consider:

- In general, showing a report of other information in a domain model is not useful since all its information is derived or duplicated from other sources. This is a reason to exclude it.
- On the other hand, it has a special role in terms of the business rules: It usually confers the right to the bearer of the (paper) 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, we would be justified to include it.

9.10 Guideline: Think Like a Mapmaker; Use Domain Terms

The mapmaker strategy applies to both maps and domain models.

Guideline

Make a domain model in the spirit of how a cartographer or mapmaker works:

- Use the existing names in the territory. For example, if developing a model for a library, name the customer a "Borrower" or "Patron"—the terms used by the library staff.
- Exclude irrelevant or out-of-scope features. For example, in the Monopoly domain model for iteration-1, cards (such as the "Get out of Jail Free" card) are not used, so don't show a *Card* in the model this iteration.
- Do not add things that are not there.

The principle is similar to the *Use the Domain Vocabulary* strategy [Coad95].

9.11 Guideline: How to Model 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. Yet it is still possible to create a domain model in these domains. It requires a high degree of abstraction, stepping back from familiar non-OO designs, and listening carefully to the core vocabulary and concepts that domain experts use.

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

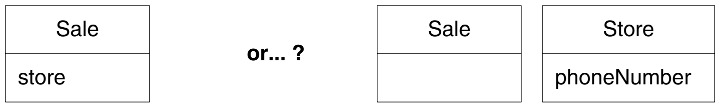
9.12 Guideline: A Common Mistake with Attributes vs. Classes

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

Guideline

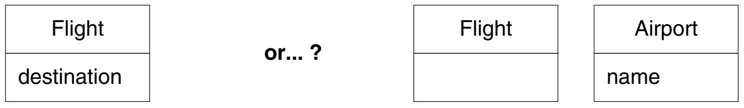
If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.

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 that occupies space. Therefore, *Store* should be a conceptual class.

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.

9.13 Guideline: When to Model with 'Description' Classes?

A **description class** contains information that describes something else. For example, a *ProductDescription* that records the price, picture, and text description of an *Item*. This was first named the *Item-Descriptor* pattern in [Coad92].

Motivation: Why Use 'Description' Classes?

The following discussion may at first seem related to a rare, highly specialized issue. However, it turns out that the need for description classes is common in many domain models.

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 one 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.

Here are some related problems: The model, if implemented in software similar to the domain model, has duplicate data, is space-inefficient, and error-prone (due to replicated information) because the description, price, and itemID are duplicated for every *Item* instance of the same product.

The preceding problem illustrates the need for objects that are *descriptions* (sometimes called *specifications*) of other things. To solve the *Item* problem, what is needed is a *ProductDescription* class that records information about items. A *ProductDescription* does not represent an *Item*, it represents a description of information *about* items. See Figure 9.9.

A particular *Item* may have a serial number; it represents a physical instance. A *ProductDescription* wouldn't have a serial number.

Switching from a conceptual to a software perspective, note that even if all inventoried items are sold and their corresponding *Item* software instances are deleted, the *ProductDescription* still remains.

The need for description classes is common in sales, product, and service domains. It is also common in manufacturing, which requires a *description* of a manufactured thing that is distinct from the thing itself.

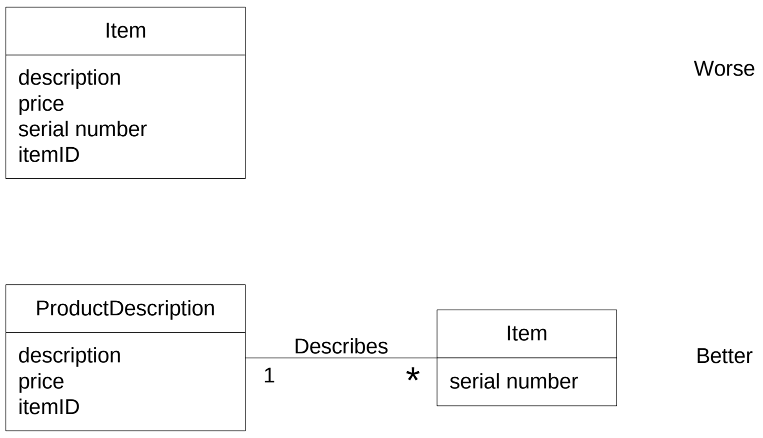


Figure 9.9 Descriptions about other things. The * means a multiplicity of "many." It indicates that one *ProductDescription* may describe many (*) *Items*.

Guideline: When Are Description Classes Useful?

Guideline

Add a description class (for example, *ProductDescription*) 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, but was incorrectly associated with the deleted thing.
- It reduces redundant or duplicated information.

Example: Descriptions in the Airline Domain

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.

The problem can be solved, both from a purely conceptual perspective in a domain model and from a software perspective in the software designs, with a *FlightDescription* that describes a flight and its route, even when a particular flight is not scheduled (see Figure 9.10).

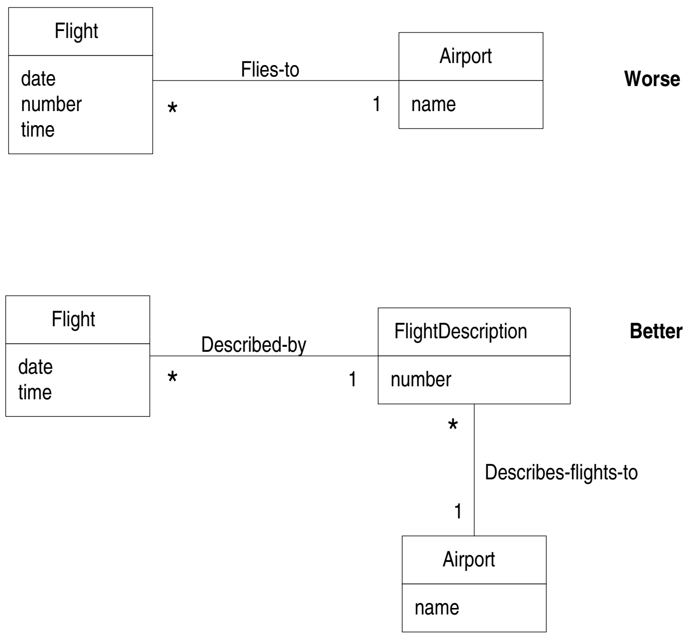


Figure 9.10 Descriptions about other things.

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, 2047 at $55 per month”). Marketing needs to define and record this service plan or *MobileCommunicationsPackageDescription* before any are sold.

9.14 Associations

It’s useful to find and show associations that are needed to satisfy the information requirements of the current scenarios under development, and which aid in understanding the domain.

An **association** is a relationship between classes (more precisely, instances of those classes) that indicates some meaningful and interesting connection (see Figure 9.11).

In the UML, associations are defined as "the semantic relationship between two or more classifiers that involve connections among their instances."

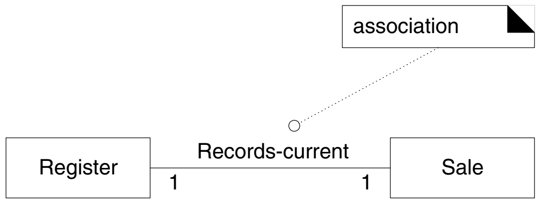


Figure 9.11 Associations.

Guideline: When to Show an Association?

Associations worth noting usually imply knowledge of a relationship that needs to be preserved for some duration—it could be milliseconds or years, depending on context. *In other words, between what objects do we need some* ***memory*** *of a relationship*?

For example, do we need to *remember* what *SalesLineItem* instances are associated with a *Sale* instance? Definitely, otherwise it would not be possible to reconstruct a sale, print a receipt, or calculate a sale total.

And we need to remember completed *Sales* in a *Ledger*, for accounting and legal purposes.

Because the domain model is a conceptual perspective, these statements about the need to remember refer to a need in a real situation of the world, not a software need, although during implementation many of the same needs will arise.

In the monopoly domain, we need to remember what *Square* a *Piece* (or *Player*) is on—the game doesn't work if that isn't remembered. Likewise, we need to remember what *Piece* is owned by a particular *Player*. We need to remember what *Squares* are part of a particular *Board*.

But on the other hand, there is no need to remember that the *Die* (or the plural, "dice") total indicates the *Square* to move to. It's true, but we don't need to have an ongoing memory of that fact, after the move has been made. Likewise, a *Cashier* may look up *ProductDescriptions*, but there is no need to remember the fact of a particular *Cashier* looking up particular *ProductDescriptions*.

Guideline

Consider including the following associations in a domain model:

- Associations for which knowledge of the relationship needs to be preserved for some duration ("need-to-remember" associations).
- Associations derived from the Common Associations List.

Guideline: Why Should We Avoid Adding Many Associations?

We need to avoid adding too many associations to a domain model. Digging back into our discrete mathematics studies, you may recall that in a graph with n nodes, there can be (n·(n-1))/2 associations to other nodes—a potentially very large number. A domain model with 20 classes could have 190 associations lines! Many lines on the diagram will obscure it with "visual noise." Therefore, be parsimonious about adding association lines. Use the criterion guidelines suggested in this chapter, and focus on "need-to-remember" associations.

Perspectives: Will the Associations Be Implemented In Software?

During domain modeling, an association is *not* a statement about data flows, database foreign key relationships, instance variables, or object connections in a software solution; it is a statement that a relationship is meaningful in a purely conceptual perspective—in the real domain.

That said, many of these relationships *will* be implemented in software as paths of navigation and visibility (both in the Design Model and Data Model). But the domain model is not a data model; associations are added to highlight our rough understanding of noteworthy relationships, not to document object or data structures.

Applying UML: Association Notation

An association is represented as a line between classes with a capitalized association name. See Figure 9.12.

The ends of an association may contain a multiplicity expression indicating the numerical relationship between instances of the classes.

The association is inherently bidirectional, meaning that from instances of either class, logical traversal to the other is possible. This traversal is purely abstract; it is *not* a statement about connections between software entities.

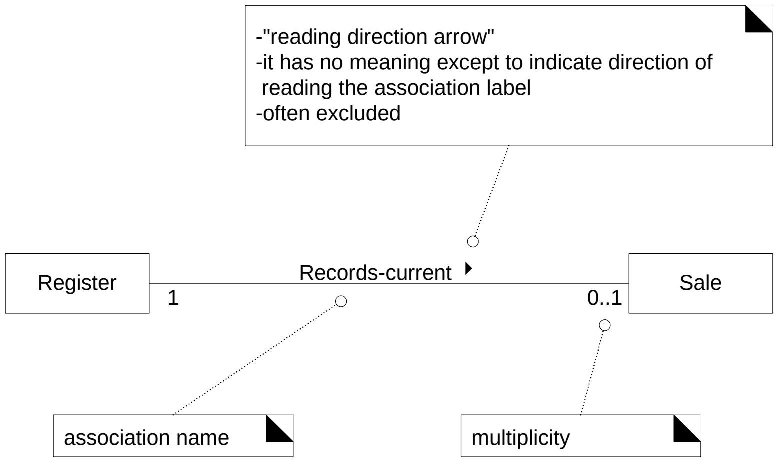

-"reading direction arrow"
-it has no meaning except to indicate direction of reading the association label
-often excluded
Register
Records-current ▸
Sale
1
0..1
association name
multiplicity

Figure 9.12 The UML notation for associations.

An optional "reading direction arrow" indicates the direction to read the association name; it does not indicate direction of visibility or navigation. If the arrow is not present, the convention is to read the association from left to right or top to bottom, although the UML does not make this a rule (see Figure 9.12).

Caution

The reading direction arrow has no meaning in terms of the model; it is only an aid to the reader of the diagram.

Guideline: How to Name an Association in UML?

Guideline

Name an association based on a *ClassName-VerbPhrase-ClassName* format where the verb phrase creates a sequence that is readable and meaningful.

Simple association names such as "Has" or "Uses" are usually poor, as they seldom enhance our understanding of the domain.

For example,

- *Sale Paid-by CashPayment*
  - bad example (doesn't enhance meaning): *Sale Uses CashPayment*
- *Player Is-on Square*
  - bad example (doesn't enhance meaning): *Player Has Square*

Association names should start with a capital letter, since an association represents a classifier of links between instances; in the UML, classifiers should start with a capital letter. Two common and equally legal formats for a compound association name are:

- *Records-current*
- *RecordsCurrent*

Applying UML: Roles

Each end of an association is called a **role**. Roles may optionally have:

- multiplicity expression
- name
- navigability

Multiplicity is examined next.

Applying UML: Multiplicity

Multiplicity defines how many instances of a class *A* can be associated with one instance of a class *B* (see Figure 9.13).

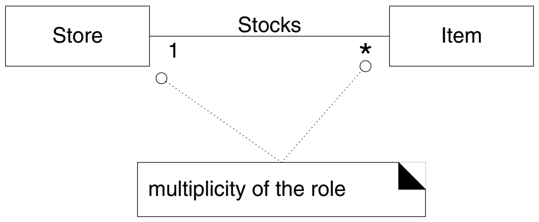


Figure 9.13 Multiplicity on an association.

For example, a single instance of a *Store* can be associated with "many" (zero or more, indicated by the *) *Item* instances.

Some examples of multiplicity expressions are shown in Figure 9.14.

The multiplicity value communicates how many instances can be validly associated with another, at a particular moment, rather than over a span of time. For example, it is possible that a used car could be repeatedly sold back to used car dealers over time. But at any particular moment, the car is only *Stocked-by* one dealer. The car is not *Stocked-by* many dealers at any particular moment. Similarly, in countries with monogamy laws, a person can be *Married-to* only one other person at any particular moment, even though over a span of time, that same person may be married to many persons.

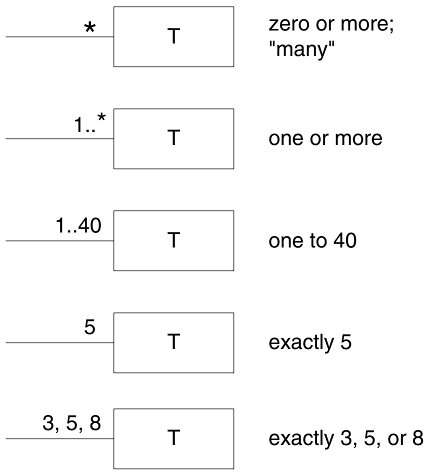
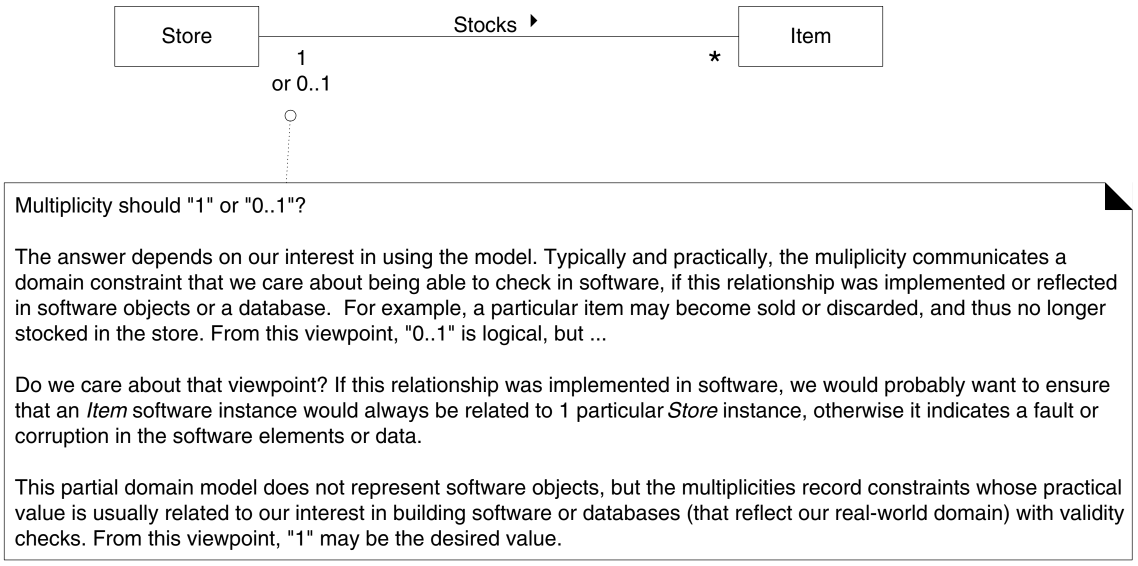


Figure 9.14 Multiplicity values.

The multiplicity value is dependent on our interest as a modeler and software developer, because it communicates a domain constraint that will be (or could be) reflected in software. See Figure 9.15 for an example and explanation.



Multiplicity should "1" or "0..1"?

The answer depends on our interest in using the model. Typically and practically, the muliplicity communicates a domain constraint that we care about being able to check in software, if this relationship was implemented or reflected in software objects or a database. For example, a particular item may become sold or discarded, and thus no longer stocked in the store. From this viewpoint, "0..1" is logical, but ...

Do we care about that viewpoint? If this relationship was implemented in software, we would probably want to ensure that an *Item* software instance would always be related to 1 particular *Store* instance, otherwise it indicates a fault or corruption in the software elements or data.

This partial domain model does not represent software objects, but the multiplicities record constraints whose practical value is usually related to our interest in building software or databases (that reflect our real-world domain) with validity checks. From this viewpoint, "1" may be the desired value.

Figure 9.15 Multiplicity is context dependent.

Rumbaugh gives another example of *Person* and *Company* in the *Works-for* association [Rumbaugh91]. Indicating if a *Person* instance works for one or many *Company* instances is dependent on the context of the model; the tax department is interested in *many*; a union probably only *one*. The choice usually depends on why we are building the software.

Applying UML: Multiple Associations Between Two Classes

Two classes may have multiple associations between them in a UML class diagram; this is not uncommon. There is no outstanding example in the POS or Monopoly case study, but an example from the domain of the airline is the relationships between a *Flight* (or perhaps more precisely, a *FlightLeg*) and an *Airport* (see Figure 9.16); the flying-to and flying-from associations are distinctly different relationships, which should be shown separately.

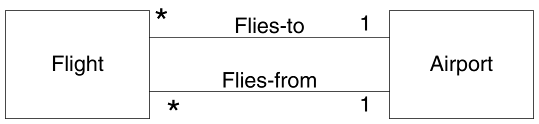


Figure 9.16 Multiple associations.

Guideline: How to Find Associations with a Common Associations List

Start the addition of associations by using the list in Table 9.2. It contains common categories that are worth considering, especially for business information systems. Examples are drawn from the 1) POS, 2) Monopoly, and 3) airline reservation domains.

Category	Examples
A is a transaction related to another transaction B	*CashPayment—Sale* *Cancellation—Reservation*
A is a line item of a transaction B	*SalesLineItem—Sale*
A is a product or service for a transaction (or line item) B	*Item—SalesLineItem (or Sale)* *Flight—Reservation*
A is a role related to a transaction B	*Customer—Payment* *Passenger—Ticket*
A is a physical or logical part of B	*Drawer—Register* *Square—Board* *Seat—Airplane*

Category	Examples
A is physically or logically contained in/on B	*Register—Store, Item—Shelf* *Square—Board* *Passenger—Airplane*
A is a description for B	*ProductDescription—Item* *FlightDescription—Flight*
A is known/logged/recorded/reported/ captured in B	*Sale—Register* *Piece—Square* *Reservation—FlightManifest*
A is a member of B	*Cashier—Store* *Player—MonopolyGame* *Pilot—Airline*
A is an organizational subunit of B	*Department—Store* *Maintenance—Airline*
A uses or manages or owns B	*Cashier—Register* *Player—Piece* *Pilot—Airplane*
A is next to B	*SalesLineItem—SalesLineItem* *Square—Square* *City—City*

Table 9.2 Common Associations List.

9.15 Example: Associations in the Domain Models

Case Study: NextGen POS

The domain model in Figure 9.17 shows a set of conceptual classes and associations that are candidates for our POS domain model. The associations are primarily derived from the "need-to-remember" criteria of this iteration requirements, and the Common Association List. Reading the list and mapping the examples to the diagram should explain the choices. For example:

- **Transactions related to another transaction**—*Sale Paid-by CashPayment*.
- **Line items of a transaction**—*Sale Contains SalesLineItem*.

- **Product for a transaction (or line item)—***SalesLineItem Records-sale-of Item*.

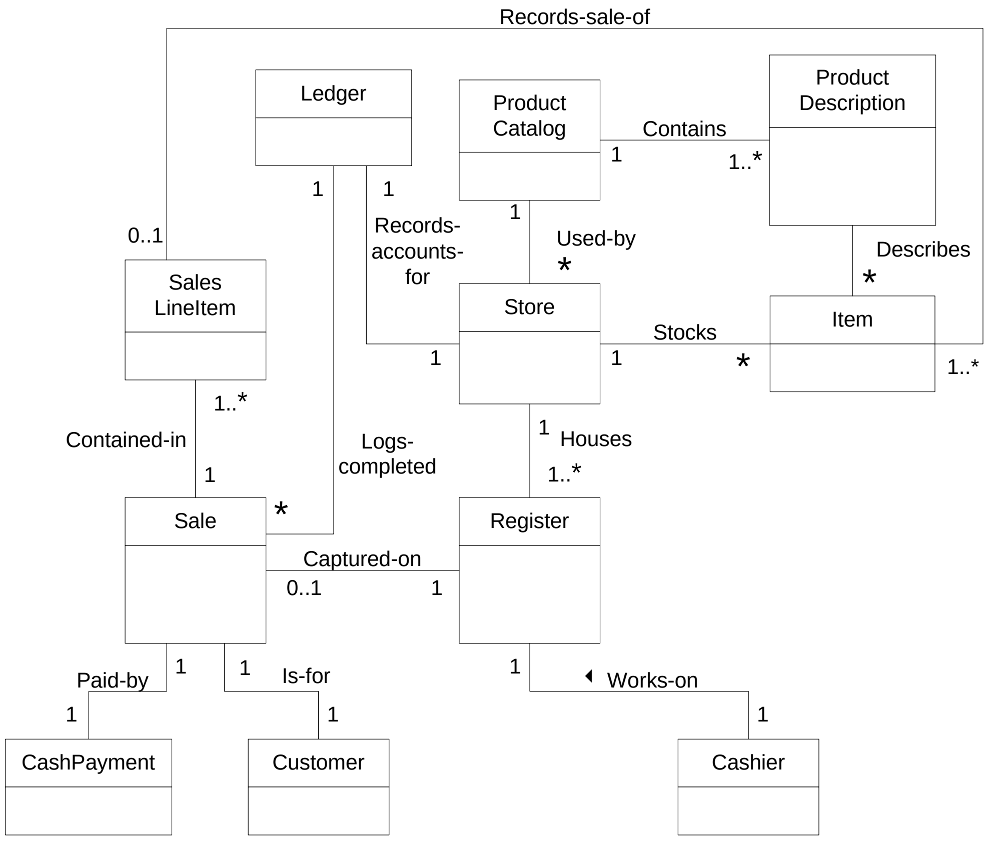


Figure 9.17 NextGen POS partial domain model.

Case Study: Monopoly

See Figure 9.18. Again, the associations are primarily derived from the "need-to-remember" criteria of this iteration requirements, and the Common Association List. For example:

- **A is contained in or on B—***Board Contains Square*.
- **A owns B—***Players Owns Piece*.
- **A is known in/on B—***Piece Is-on Square*.
- **A is member of B—***Player Member-of (or Plays) MonopolyGame*.

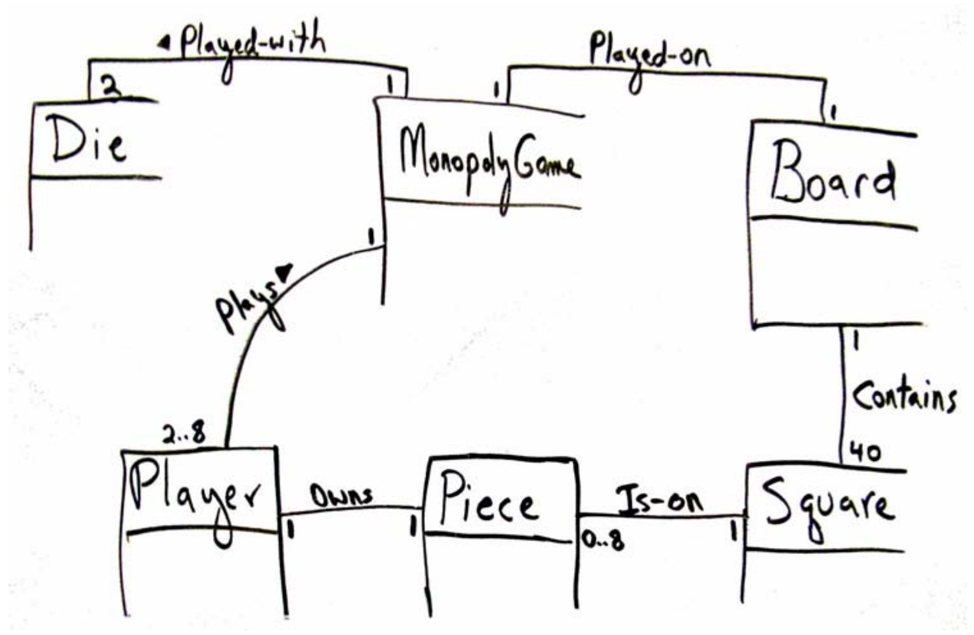


Figure 9.18 Monopoly partial domain model.

9.16 Attributes

It is useful to identify those attributes of conceptual classes that are needed to satisfy the information requirements of the current scenarios under development. An **attribute** is a logical data value of an object.

Guideline: When to Show Attributes?

Include attributes that the requirements (for example, use cases) suggest or imply a need to remember information.

For example, a receipt (which reports the information of a sale) in the *Process Sale* use case normally includes a date and time, the store name and address, and the cashier ID, among many other things.

Therefore,

- *Sale* needs a *dateTime* attribute.
- *Store* needs a *name* and *address*.
- *Cashier* needs an *ID*.

Applying UML: Attribute Notation

Attributes are shown in the second compartment of the class box (see Figure 9.19). Their type and other information may optionally be shown.

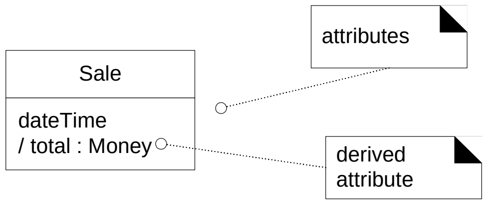


Figure 9.19 Class and attributes.

More Notation

detailed UML class diagram notation p. 249, and also on the back inside cover of the book

The full syntax for an attribute in the UML is:

visibility name : type multiplicity = default {property-string}

Some common examples are shown in Figure 9.20.

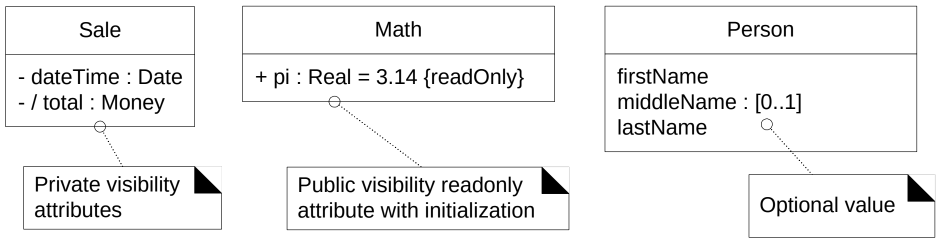

Sale
- dateTime : Date
- / total : Money
Private visibility attributes
Math
+ pi : Real = 3.14 {readOnly}
Public visibility readonly attribute with initialization
Person
firstName
middleName : [0..1]
lastName
Optional value

Figure 9.20 Attribute notation in UML.

As a convention, most modelers will assume attributes have private visibility (-) unless shown otherwise, so I don't usually draw an explicit visibility symbol.

{readOnly} is probably the most common property string for attributes.

Multiplicity can be used to indicate the optional presence of a value, or the number of objects that can fill a (collection) attribute. For example, many domains require that a first and last name be known for a person, but that a middle name is optional. The expression *middleName : [0..1]* indicates an optional value—0 or 1 values are present.

Guideline: Where to Record Attribute Requirements?

Notice that, subtly, *middleName : [0..1]* is a requirement or domain rule, embedded in the domain model. Although this is just a conceptual-perspective domain model, it probably implies that the software perspective should allow a missing value for *middleName* in the UI, the objects, and the database. Some modellers accept leaving such specifications only in the domain model, but I find this error-prone and scattered, as people tend to not look at the domain model in detail, or for requirements guidance. Nor do they usually maintain the domain model.

Instead, I suggest placing all such attribute requirements in the UP Glossary, which serves as a data dictionary. Perhaps I've spent an hour sketching a domain model with a domain expert; afterwards, I can spend 15 minutes looking through it and transferring implied attribute requirements into the Glossary.

Another alternative is to use a tool that integrates UML models with a data dictionary; then all attributes will automatically show up as dictionary elements.

Derived Attributes

The *total* attribute in the *Sale* can be calculated or derived from the information in the *SalesLineItems*. When we want to communicate that 1) this is a noteworthy attribute, but 2) it is derivable, we use the UML convention: a / symbol before the attribute name.

As another example, a cashier can receive a group of like items (for example, six tofu packages), enter the *itemID* once, and then enter a quantity (for example, six). Consequently, an individual *SalesLineItem* can be associated with more than one instance of an item.

The quantity that is entered by the cashier may be recorded as an attribute of the *SalesLineItem* (Figure 9.21). However, the quantity can be calculated from the actual multiplicity value of the association, so it may be characterized as a derived attribute—one that may be derived from other information.

Guideline: What are Suitable Attribute Types?

Focus on Data Type Attributes in the Domain Model

Informally, most attribute types should be what are often thought of as "primitive" data types, such as numbers and booleans. The type of an attribute should *not* normally be a complex domain concept, such as a *Sale* or *Airport*.

For example, the *currentRegister* attribute in the *Cashier* class in Figure 9.22 is undesirable because its type is meant to be a *Register*, which is not a simple data type (such as *Number* or *String*). The most useful way to express that a *Cashier* uses a *Register* is with an association, not with an attribute.

Guideline

The attributes in a domain model should preferably be **data types**. Very common data types include: *Boolean, Date (or DateTime), Number, Character, String (Text), Time.*

Other common types include: *Address, Color, Geometrics (Point, Rectangle), Phone Number, Social Security Number, Universal Product Code (UPC), SKU, ZIP or postal codes, enumerated types*

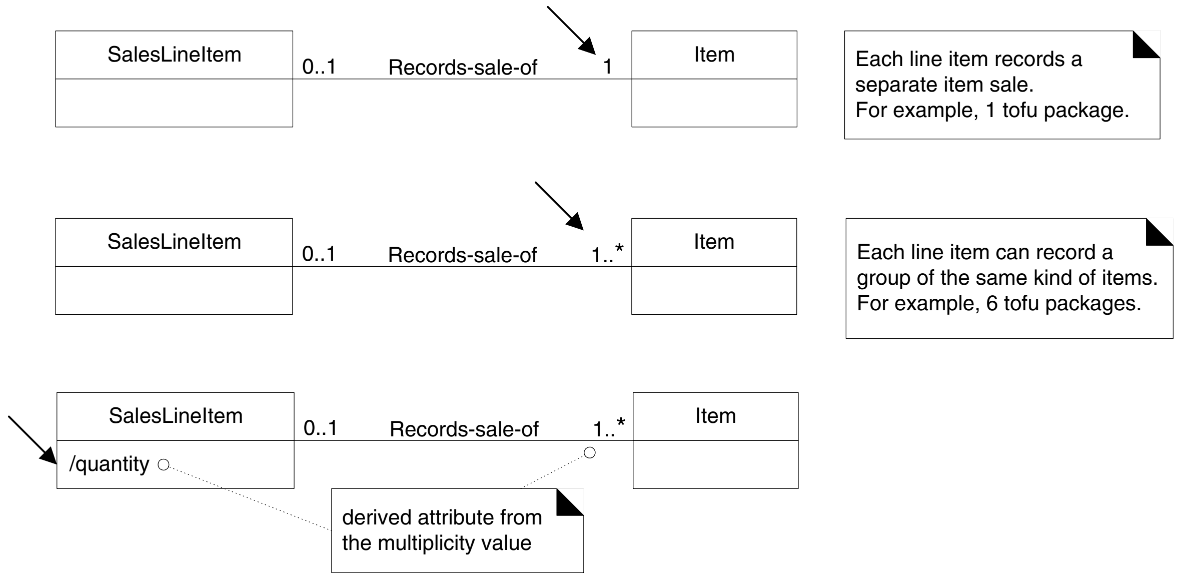

SalesLineItem
0..1
Records-sale-of
1
Item
Each line item records a separate item sale. For example, 1 tofu package.
SalesLineItem
0..1
Records-sale-of
1..*
Item
Each line item can record a group of the same kind of items. For example, 6 tofu packages.
SalesLineItem
/quantity
0..1
Records-sale-of
1..*
Item
derived attribute from the multiplicity value

Figure 9.21 Recording the quantity of items sold in a line item.

To repeat an earlier example, a common confusion is modeling a complex domain concept as an attribute. To illustrate, a destination airport is not really a string; it is a complex thing that occupies many square kilometers of space. Therefore, *Flight* should be related to *Airport* via an association, not with an attribute, as shown in Figure 9.23.

Guideline

Relate conceptual classes with an association, not with an attribute.

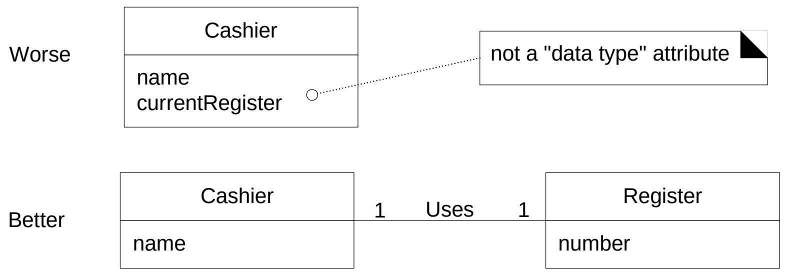


Figure 9.22 Relate with associations, not attributes.

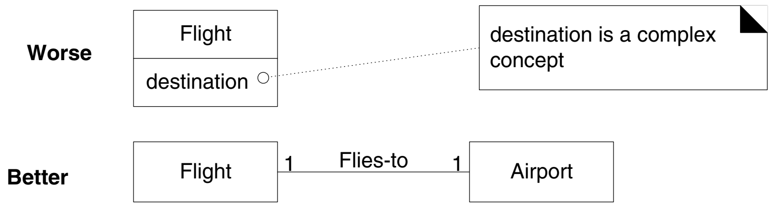


Figure 9.23 Don't show complex concepts as attributes; use associations.

Data Types

As said, attributes in the domain model should generally be **data types**; informally these are "primitive" types such as number, boolean, character, string, and enumerations (such as Size = {small, large}). More precisely, this is a UML term that implies a set of values for which unique identity is not meaningful (in the context of our model or system) [RJB99]. Said another way, equality tests are *not* based on identity, but instead on value.[6] For example, it is not (usually) meaningful to distinguish between:

- Separate instances of the *Integer* 5.
- Separate instances of the *String* 'cat'.
- Separate instance of the *Date* "Nov. 13, 1990".

By contrast, it *is* meaningful to distinguish (by object identity) between two separate *Person* instances whose names are both "Jill Smith" because the two instances can represent separate individuals with the same name.

Also, data type values are usually immutable. For example, the instance '5' of *Integer* is immutable; the instance "Nov. 13, 1990" of *Date* is probably immutable. On the other hand, a *Person* instance may have its *lastName* changed for various reasons.

6. In Java, for example, a value test is done with the *equals* method, and an identity test with the == operator.

From a software perspective, there are few situations where one would compare the memory addresses (identity) of instances of *Integer* or *Date*; only value-based comparisons are relevant. On the other hand, the memory addresses of *Person* instances could conceivably be compared and distinguished, even if they had the same attribute values, because their unique identity is important.

Some OO and UML modeling books also speak of **value objects**, which are very similar to data types, but with minor variations. However, I found the distinctions rather fuzzy and subtle, and don't stress it.

Perspectives: What About Attributes in Code?

The recommendation that attributes in the domain model be mainly data types does *not* imply that C# or Java attributes must only be of simple, primitive data types. The domain model is a conceptual perspective, not a software one. In the Design Model, attributes may be of any type.

Guideline: When to Define New Data Type Classes?

In the NextGen POS system an *itemID* attribute is needed; it is probably an attribute of an *Item* or *ProductDescription*. Casually, it seems like just a number or perhaps a string. For example, *itemID : Integer* or *itemID : String*.

But it is more than that (item identifiers have subparts), and in fact it is useful to have a class named *ItemID* (or *ItemIdentifier*) in the domain model, and designate the type of the attribute as such. For example, *itemID : ItemIdentifier*.

Table 9.3 provides guidelines when it's useful to model with data types.

Applying these guidelines to the POS domain model attributes yields the following analysis:

- The item identifier is an abstraction of various common coding schemes, including UPC-A, UPC-E, and the family of EAN schemes. These numeric coding schemes have subparts identifying the manufacturer, product, country (for EAN), and a check-sum digit for validation. Therefore, there should be a data type *ItemID* class, because it satisfies many of the guidelines above.
- The *price* and *amount* attributes should be a data type *Money* class because they are quantities in a unit of currency.
- The *address* attribute should be a data type *Address* class because it has separate sections.

Guideline

Represent what may initially be considered a number or string as a new data type class in the domain model if:

- It is composed of separate sections.
  - phone number, name of person
- There are operations associated with it, such as parsing or validation.
  - social security number
- It has other attributes.
  - promotional price could have a start (effective) date and end date
- It is a quantity with a unit.
  - payment amount has a unit of currency
- It is an abstraction of one or more types with some of these qualities.
  - item identifier in the sales domain is a generalization of types such as Universal Product Code (UPC) and European Article Number (EAN)

Table 9.3 Guidelines for modeling data types.

Applying UML: Where to Illustrate These Data Type Classes?

Should the *ItemID* class be shown as a separate class in a domain model? It depends on what you want to emphasize in the diagram. Since *ItemID* is a **data type** (unique identity of instances is not used for equality testing), it may be shown only in the attribute compartment of the class box, as shown in Figure 9.24. On the other hand, if *ItemID* is a new type with its own attributes and associations, showing it as a conceptual class in its own box may be informative. There is no correct answer; resolution depends on how the domain model is being used as a tool of communication, and the significance of the concept in the domain.

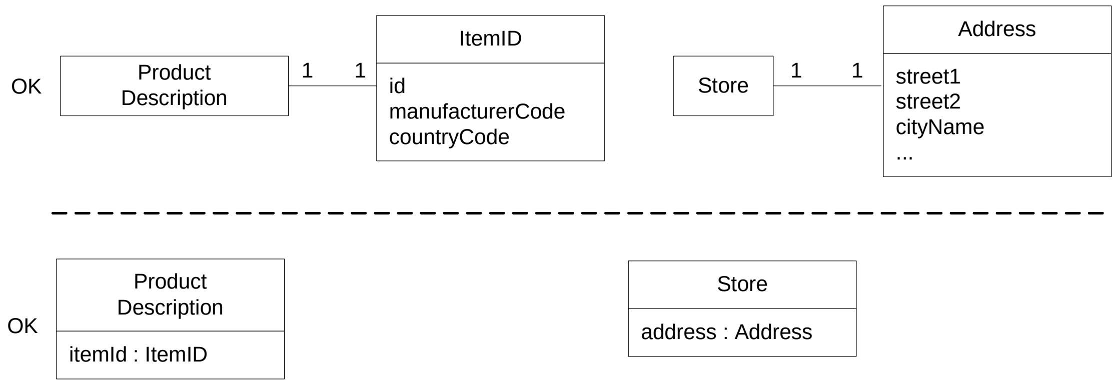


Figure 9.24 Two ways to indicate a data type property of an object.

Guideline: No Attributes Representing Foreign Keys

Attributes should *not* be used to relate conceptual classes in the domain model. The most common violation of this principle is to add a kind of **foreign key attribute**, as is typically done in relational database designs, in order to associate two types. For example, in Figure 9.25 the *currentRegisterNumber* attribute in the *Cashier* class is undesirable because its purpose is to relate the *Cashier* to a *Register* object. The better way to express that a *Cashier* uses a *Register* is with an association, not with a foreign key attribute. Once again, relate types with an association, not with an attribute.

There are many ways to relate objects—foreign keys being one—and we will defer how to implement the relation until design to avoid **design creep**.

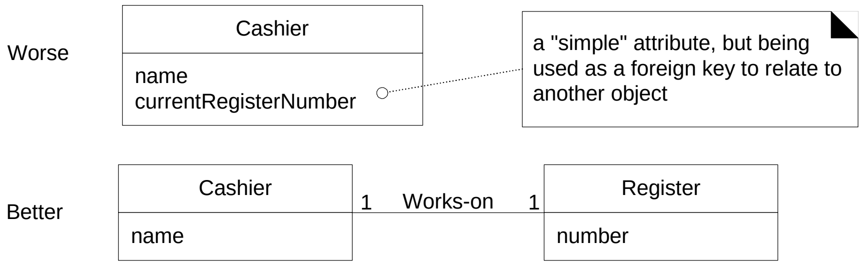

Worse
Cashier
name
currentRegisterNumber
a "simple" attribute, but being used as a foreign key to relate to another object
Better
Cashier
name
1
Works-on
1
Register
number

Figure 9.25 Do not use attributes as foreign keys.

Guideline: Modeling Quantities and Units

Most numeric quantities should *not* be represented as plain numbers. Consider price or weight. Saying “the price was 13” or “the weight was 37” doesn’t say much. Euros? Kilograms?

These are quantities with associated units, and it is common to require knowledge of the unit to support conversions. The NextGen POS software is for an international market and needs to support prices in multiple currencies. The domain model (and the software) should model quantities skillfully.

In the general case, the solution is to represent *Quantity* as a distinct class, with an associated *Unit* [Fowler96]. It is also common to show *Quantity* specializations. *Money* is a kind of quantity whose units are currencies. *Weight* is a quantity with units such as kilograms or pounds. See Figure 9.26.

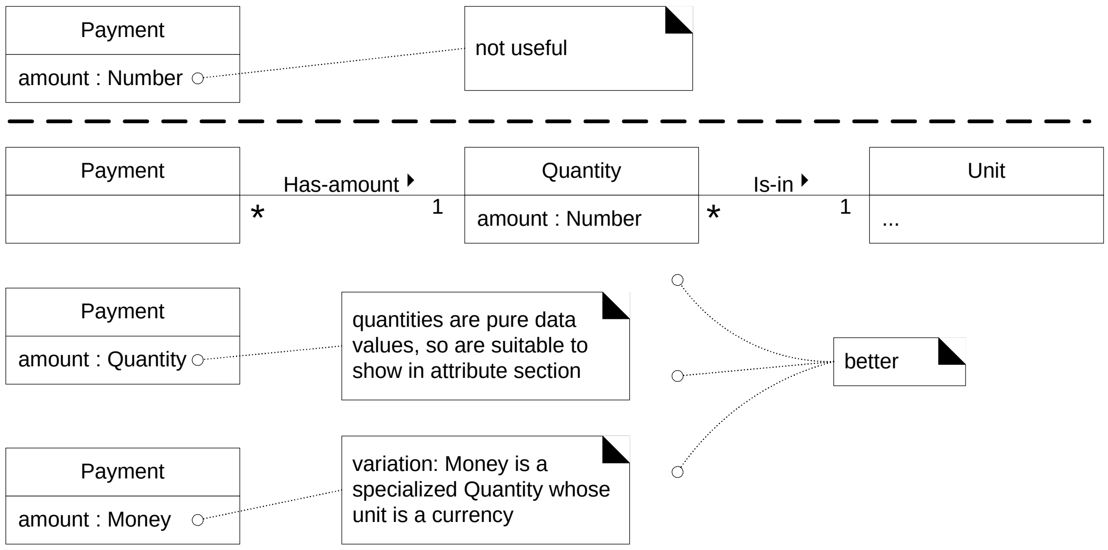

Payment
amount : Number
not useful
Payment
Has-amount ▸
*
1
Quantity
amount : Number
Is-in ▸
*
1
Unit
...
Payment
amount : Quantity
quantities are pure data values, so are suitable to show in attribute section
better
Payment
amount : Money
variation: Money is a specialized Quantity whose unit is a currency

Figure 9.26 Modeling quantities.

9.17 Example: Attributes in the Domain Models

Case Study: NextGen POS

See Figure 9.27. The attributes chosen reflect the information requirements for this iteration—the *Process Sale* cash-only scenarios of this iteration. For example:

CashPayment	*amountTendered*—To determine if sufficient payment was provided, and to calculate change, an amount (also known as "amount tendered") must be captured.
Product-Description	*description*—To show the description on a display or receipt.
	itemId—To look up a *ProductDescription*.
	price—To calculate the sales total, and show the line item price.

Sale	*dateTime*—A receipt normally shows date and time of sale, and this is useful for sales analysis.
SalesLineItem	*quantity*—To record the quantity entered, when there is more than one item in a line item sale (for example, *five* packages of tofu).
Store	*address, name*—The receipt requires the name and address of the store.

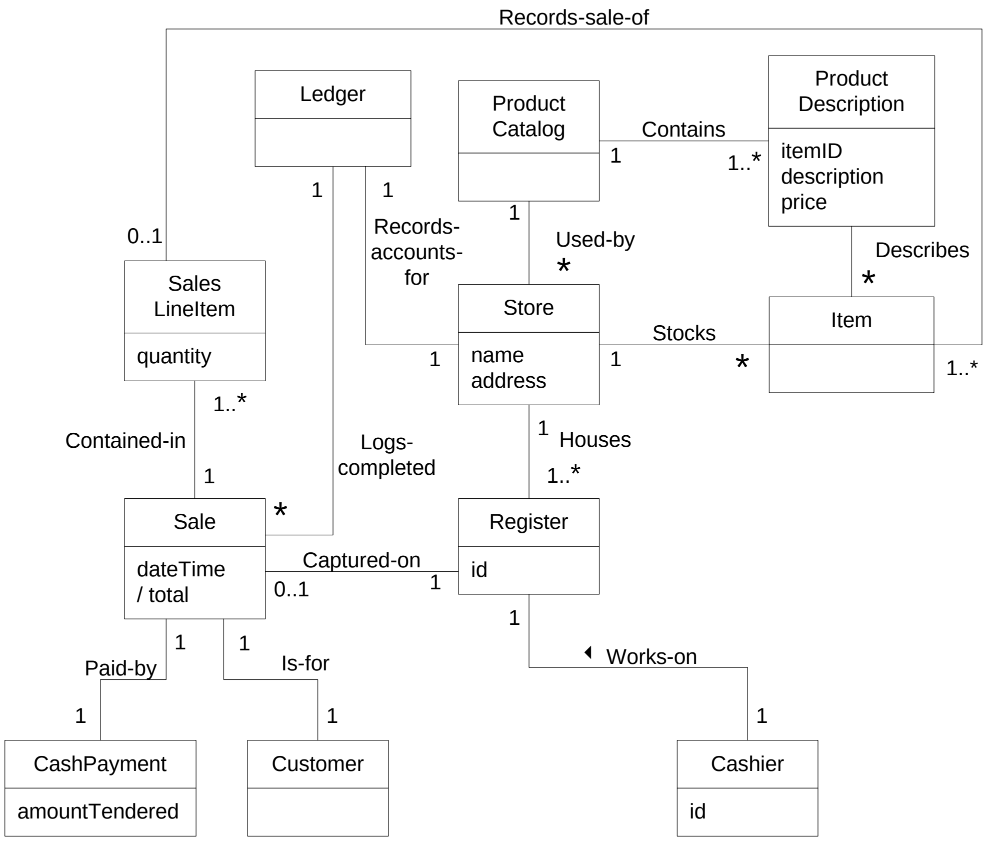


Figure 9.27 NextGen POS partial domain model.

Case Study: Monopoly

See Figure 9.28. The attributes chosen reflect the information requirements for this iteration—the simplified *Play Monopoly Game* scenario of this iteration. For example:

Die	*faceValue*—After rolling the dice, needed to calculate the distance of a move.
Square	*name*—To print the desired trace output.

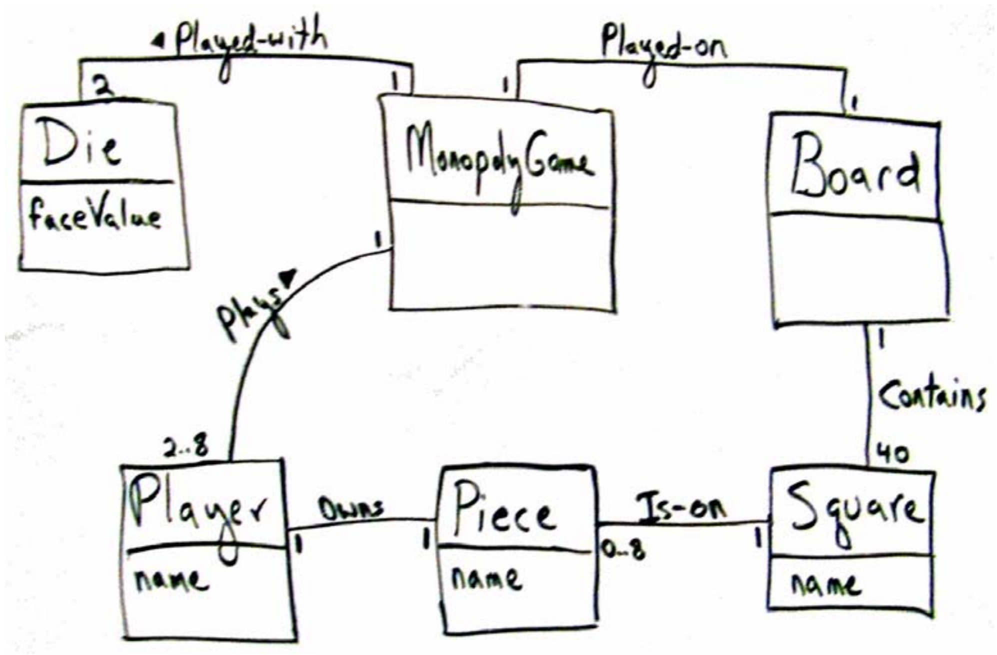


Figure 9.28 Monopoly partial domain model.

9.18 Conclusion: Is the Domain Model Correct?

There is no such thing as a single correct domain model. All models are approximations of the domain we are attempting to understand; the domain model is primarily a tool of understanding and communication among a particular group. A useful domain model captures the essential abstractions and information required to understand the domain in the context of the current requirements, and aids people in understanding the domain—its concepts, terminology, and relationships.

9.19 Process: Iterative and Evolutionary Domain Modeling

Although paradoxically a significant number of pages were devoted to explaining domain modeling, in experienced hands the development of a (partial, evolutionary) model in each iteration may take only 30 minutes. This is further shortened by the use of predefined analysis patterns.

In iterative development, we incrementally evolve a domain model over several iterations. In each, the domain model is limited to the prior and current scenarios under consideration, rather than expanding to a "big bang" waterfall-style model that early on attempts to capture all possible conceptual classes and relationships. For example, this POS 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.

And to reiterate advice from the start of this chapter:

Guideline

Avoid a waterfall-mindset big-modeling effort to make a thorough or "correct" domain model—it won't ever be either, and such over-modeling efforts lead to *analysis paralysis*, with little or no return on the investment.

Limit domain modeling to no more than a few hours per iteration.

Domain Models Within the UP

elaboration phase p. 33

As suggested in the example of Table 9.4, the UP Domain Model is usually both started and completed in the elaboration phase.

Discipline	**Artifact**	**Incep.**	**Elab.**	**Const.**	**Trans.**
	Iteration→	I1	E1..En	C1..Cn	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	

Table 9.4 Sample UP artifacts and timing. s - start; r - refine

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.

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.

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 the many other definitions of a BOM—is a kind of enterprise model that describes 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]

9.20 Recommended Resources

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 much 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.