Chapter 12

DOMAIN MODEL: ADDING ATTRIBUTES

Any sufficiently advanced bug is indistinguishable from a feature.

—Rich Kulawiec

Objectives

- Identify attributes in a domain model.
- Distinguish between correct and incorrect attributes.

Introduction

It is useful to identify those attributes of conceptual classes that are needed to satisfy the information requirements of the current scenarios under development. This chapter explores the identification of suitable attributes, and adds attributes to the domain model for the NextGen domain model.

12.1 Attributes

An attribute is a logical data value of an object.

Include the following attributes in a domain model: Those for which 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) normally includes a date and time, and management wants to know the dates and times of sales for a variety of reasons. Consequently, the *Sale* conceptual class needs a *date* and *time* attribute.

12.2 UML Attribute Notation

Attributes are shown in the second compartment of the class box (see Figure 12.1). Their type may optionally be shown.

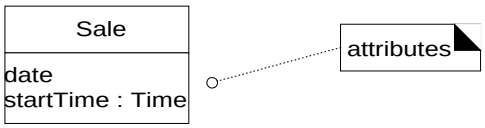

Figure 12.1 Class and attributes.

12.3 Valid Attribute Types

There are some things that should not be represented as attributes, but rather as associations. This section explores valid attributes.

Keep Attributes Simple

Intuitively, most simple attribute types are what are often thought of as primitive data types, such as numbers. The type of an attribute should not normally be a complex domain concept, such as a *Sale or Airport.* For example, the following *currentRegister* attribute in the *Cashier* class in Figure 12.2 is undesirable because its type is meant to be a *Register,* which is not a simple attribute 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..

The attributes in a domain model should preferably be simple attributes or data types.
Very common attribute data types include: *Boolean, Date, Number, String (Text), Time*
Other common types include: *Address, Color, Geometries (Point, Rectangle), Phone Number, Social Security Number, Universal Product Code (UPC), SKU, ZIP or postal codes, enumerated types*

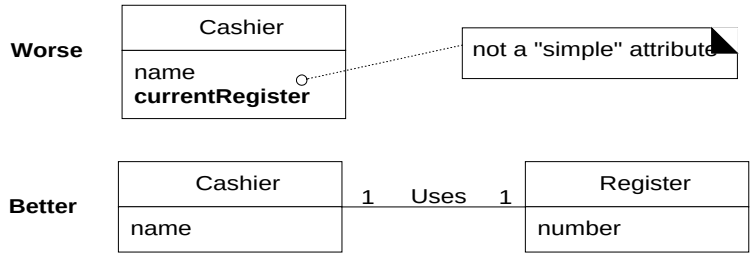

Figure 12.2 Relate with associations, not attributes.

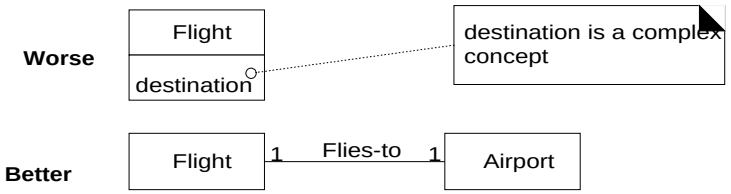

Figure 12.3 Avoid representing complex domain concepts as attributes; use associations.

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 12.3.

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

Conceptual vs. Implementation Perspectives: What About Attributes in Code?

The restriction that attributes in the domain model be only of simple data types does *not* imply that C++ or Java attributes (data members, instance fields) must only be of simple, primitive data types. The domain model focuses on pure conceptual statements about a problem domain, not software components.

Later, during design and implementation work, it will be seen that the associations between objects expressed in the domain model will often be implemented as attributes that reference other complex software objects. However, this is but one of a number of possible design solutions to implement an association, and so the decision should be deferred during domain modeling.

Data Types

Attributes should generally be **data types.** 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]. For example, it is not (usually) meaningful to distinguish between:

- Separate instances of the *Number* 5.
- Separate instances of the *String* 'cat'.
- Separate instances of *PhoneNumber* that contain the same number.
- Separate instances *of Address* that contain the same address.

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

In terms of software, there are few situations where one would compare the memory addresses of instances of *Number, String, PhoneNumber,* or *Address;* only value-based comparisons are relevant. By contrast, it is conceivable to compare the memory addresses *of Person* instances, and to distinguish them, even if they had the same attribute values, because their unique identity is important.

Thus, all primitive types (number, string) are UML data types, but not all data types are primitives. For example, *PhoneNumber* is a non-primitive data type.

These data type values are also known as **value objects.**

The notion of data types can get subtle. As a rule of thumb, stick to the basic test of "simple" attribute types: Make it an attribute if it is naturally thought of as number, string, boolean, date, or time (and so on); otherwise, represent it as a separate conceptual class.

> If in doubt, define something as a separate conceptual class rather than as an attribute.

12.4 Non-primitive Data Type Classes

The type of an attribute may be expressed as a non-primitive class in its own right in a domain model. For example, in the POS system there is an item identifier. It is typically viewed as just a number. So should it be represented as a non-primitive class? Apply this guideline:

Represent what may initially be considered a primitive data type (such as a number or string) as a non-primitive class if:

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

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 non-primitive *ItemID* class, because it satisfies many of the guidelines above.
- The *price* and *amount* attributes should be non-primitive *Quantity* or *Money* classes because they are quantities in a unit of currency.
- The *address* attribute should be a non-primitive *Address* class because it has separate sections.

The classes *ItemID, Address,* and *Quantity* are data types (unique identity of instances is not meaningful) but they are worth considering as separate classes because of their qualities.

Where to Illustrate Data Type Classes?

Should the *ItemID* class be shown as a separate conceptual 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 important), it may be shown in the attribute compartment of the class box, as shown in Figure 12.4. But since it is a non-primitive class, with its own attributes and associations, it may be interesting to show it as a conceptual class in its own box. There is no correct answer; it depends on how the domain model is being used as a tool of communication, and the significance of the concept in the domain.

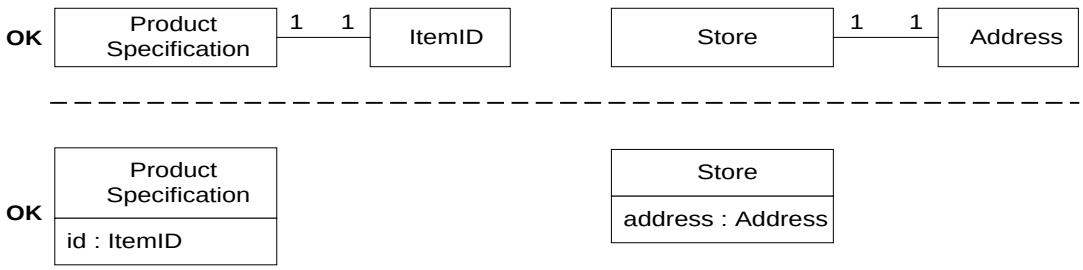

Figure 12.4 If the attribute class is a data type, it may be shown in the attribute box.

> A domain model is a tool of communication; choices about what is shown should be made with that consideration in mind.

12.5 Design Creep: No Attributes as 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 12.5 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, in order to avoid **design creep.**

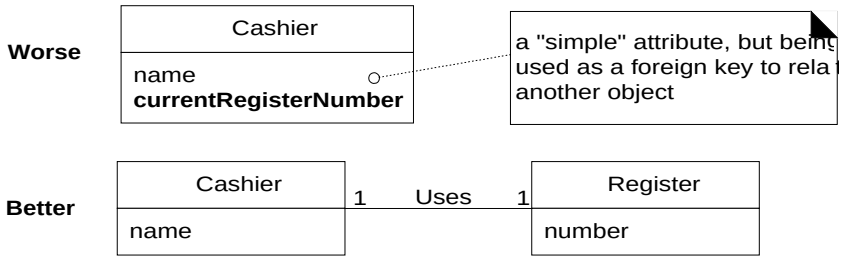

Figure 12.5 Do not use attributes as foreign keys.

12.6 Modeling Attribute Quantities and Units

Most numeric quantities should not be represented as plain numbers. Consider price or velocity. These are quantities with associated units, and it is common to require knowing the unit, and to support conversions. The NextGen POS software is for an international market and needs to support prices in multiple currencies. In the general case, the solution is to represent *Quantity* as a distinct conceptual class, with an associated *Unit* [Fowler96]. Since quantities are considered data types (unique identity of instances is not important), it is acceptable to collapse their illustration into the attribute section of the class box (see Figure 12.6). 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.

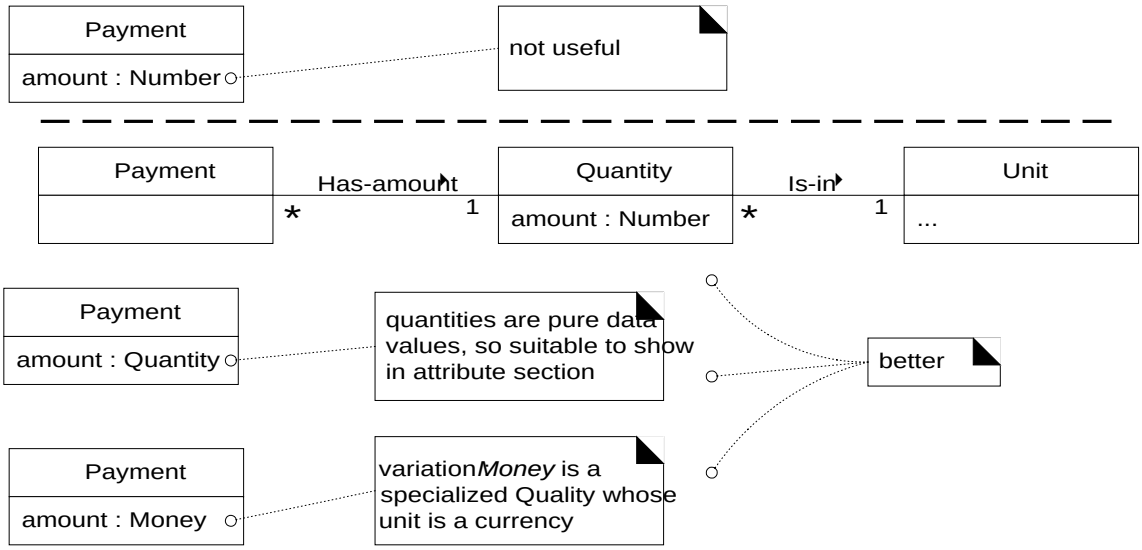

Figure 12.6 Modeling quantities.

12.7 Attributes in the NextGen Domain Model

The attributes chosen reflect the requirements for this iteration—the Process *Sale* scenarios of this iteration.

Payment	*amount*—To determine if sufficient payment was provided, and to calculate change, an amount (also known as "amount tendered") must be captured.
Product-Specification	*description*—To show the description on a display or receipt.
	id—To look up a *ProductSpecification,* given an entered itemID, it is necessary to relate them to a *id.*
	price—To calculate the sales total, and show the line item price.
Sale	*date, time*—A receipt is a paper report of a sale. It normally shows date and time of sale.
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.

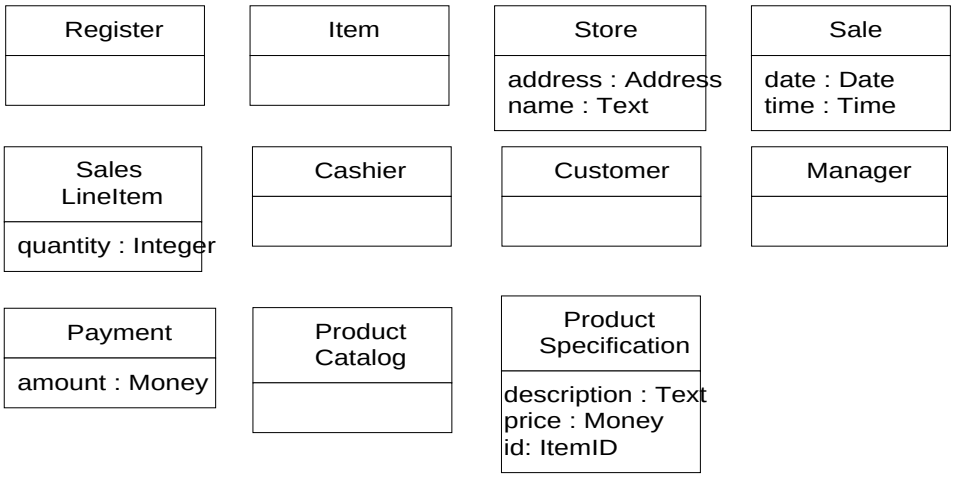

Figure 12.7 Domain model showing attributes.

12.8 Multiplicity From SalesLineItem to Item

It is possible for a cashier to 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 12.8). However, the quantity can be calculated from the actual multiplicity value of the relationship, so it may be characterized as a **derived attribute**—one that may be derived from other information. In the UML, a derived attribute is indicated with a "/" symbol.

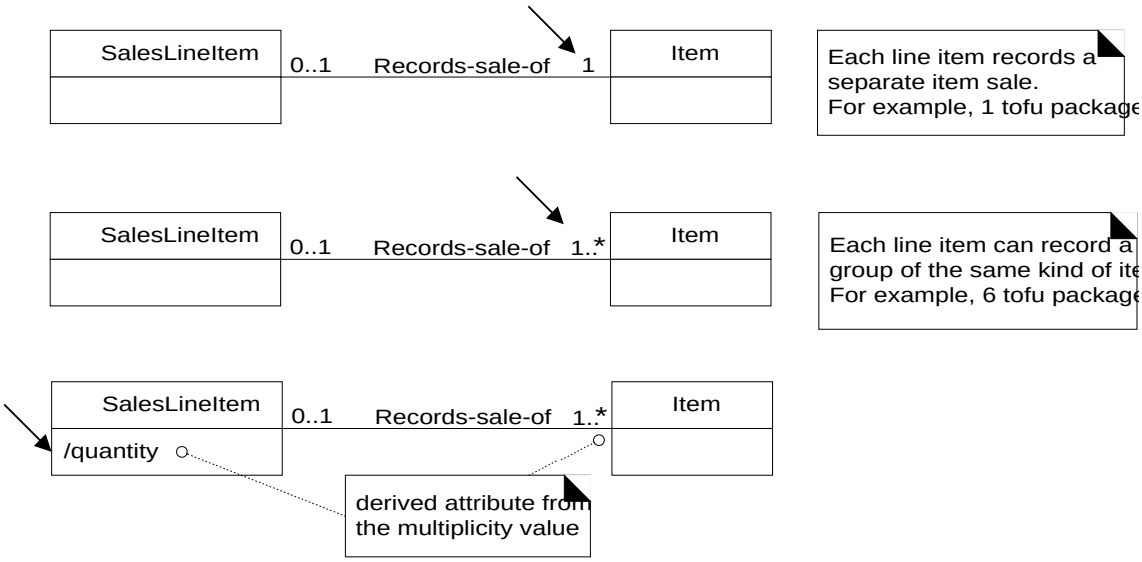

Figure 12.8 Recording the quantity of items sold in a line item.

12.9 Domain Model Conclusion

Combining the conceptual classes, associations, and attributes discovered in the previous investigation yields the model illustrated in Figure 12.9.

A relatively useful domain model for the domain of the POS application has been created. There is no such thing as a single correct model. All models are approximations of the domain we are attempting to understand. A good 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.

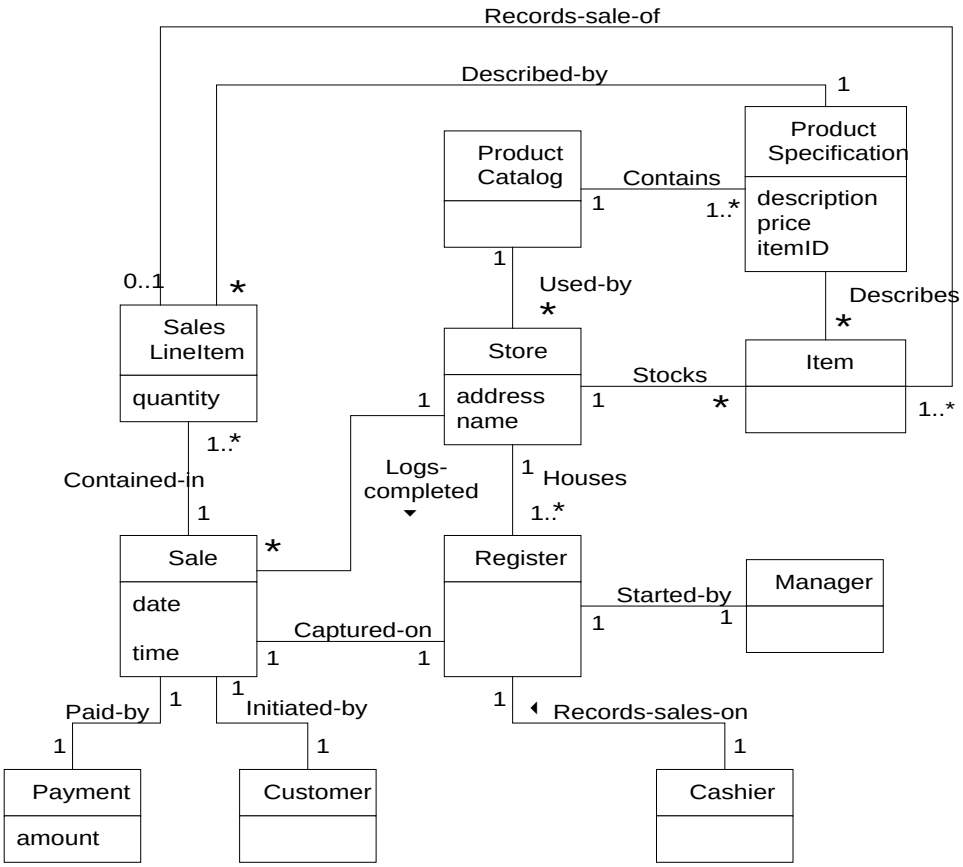

Figure 12.9 A partial domain model.