- plan for the first iteration
- candidate tools list

On to Elaboration

Elaboration is the initial series of iterations during which, on a normal project:

- the core, risky software architecture is programmed and tested
- the majority of requirements are discovered and stabilized
- the major risks are mitigated or retired

Elaboration is the initial series of iterations during which the team does serious investigation, implements (programs and tests) the core architecture, clarifies most requirements, and tackles the high-risk issues. In the UP, "risk" includes business value. Therefore, early work may include implementing scenarios that are deemed important, but are not especially technically risky.

During this phase, one is not creating throw-away prototypes; rather, the code and design are production-quality portions of the final system. In some UP descriptions, the potentially misunderstood term "**architectural prototype**" is used to describe the partial system. This is not meant to be a prototype in the sense of a discardable experiment; in the UP, it means a production subset of the final system. More commonly it is called the **executable architecture** or **architectural baseline**.

> Elaboration in one sentence:
>
> *Build the core architecture, resolve the high-risk elements, define most requirements, and estimate the overall schedule and resources.*

Some key ideas and best practices will manifest in elaboration:

- do short timeboxed risk-driven iterations
- start programming early
- adaptively design, implement, and test the core and risky parts of the architecture
- test early, often, realistically
- adapt based on feedback from tests, users, developers
- write most of the use cases and other requirements in detail, through a series of workshops, once per elaboration iteration

What Artifacts May Start in Elaboration?

Table 3.1 lists *sample* artifacts that may be *started* in elaboration, and indicates the issues they address. Subsequent chapters will examine some of these in

greater detail, especially the Domain Model and Design Model. For brevity, the table excludes artifacts that may have begun in inception; it introduces artifacts that are more likely to start in elaboration. Note these will not be completed in one iteration; rather, they will be refined over a series of iterations.

Artifact	Comment
Domain Model	This is a visualization of the domain concepts; it is similar to a static information model of the domain entities.
Design Model	This is the set of diagrams that describes the logical design. This includes software class diagrams, object interaction diagrams, package diagrams, and so forth.
Software Architecture Document	A learning aid that summarizes the key architectural issues and their resolution in the design. It is a summary of the outstanding design ideas and their motivation in the system.
Data Model	This includes the database schemas, and the mapping strategies between object and non-object representations.
Use-Case Storyboards, UI Prototypes	A description of the user interface, paths of navigation, usability models, and so forth.

Table 3.1 Sample elaboration artifacts, excluding those started in inception.

You Know You Didn't Understand Elaboration When...

- It is more than "a few" months long for most projects.
- It only has one iteration (with rare exceptions for well-understood problems).
- Most requirements were defined before elaboration.
- The risky elements and core architecture are not being tackled.
- It does not result in an *executable* architecture; there is no production-code programming.
- It is considered primarily a requirements or design phase, preceding an implementation phase in construction.
- There is an attempt to do a full and careful design before programming.
- There is minimal feedback and adaptation; users are not continually engaged in evaluation and feedback.
- There is no early and realistic testing.
- The architecture is speculatively finalized before programming.

UNIT - 2
STATIC MODELING

Elaboration:

In Elaboration, we build the core architecture, resolve the high-risk elements, define most requirements and estimate the overall schedule and resources.

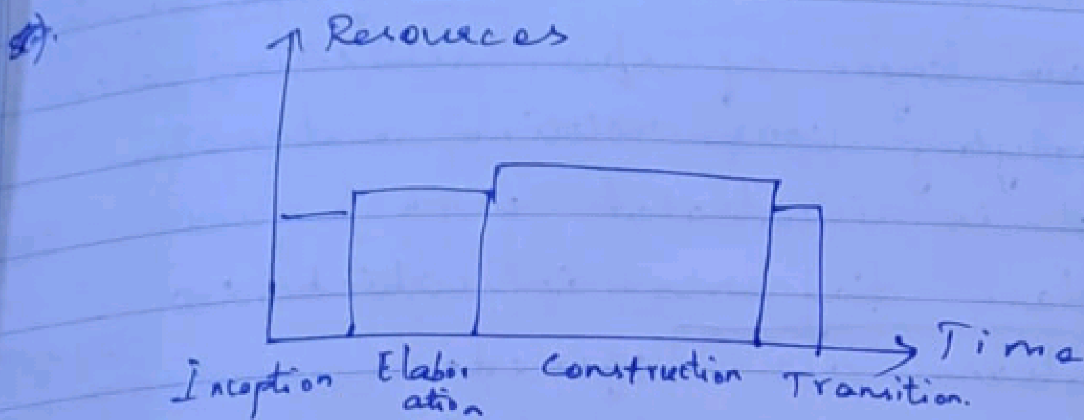
Guidelines to be followed in elaboration part are:-

1) do short timeboxed risk-driven iterations.
2) start programming early
3) adaptively design, implement and test the core risky parts of architecture.
4) test early, often realistically.

Outcomes:-

1) A use-case model in which use-cases and the actors have been identified and most of uc descriptions are developed.
2) Description of s/w archi. in s/w system dev. process.
3) Business case and risk list which are revised.
4) A development plan for overall project.

5).



→ primary goal of this phase is to address known risk factors and to establish validate sys. architecture.
→ Common processes include in This phase is creation of use case diagram, Conceptu diagram (class dig. with basic notations) & package dig. (architecture).

3.6 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.

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."

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 3.3 for a moment. See how it visualizes and relates words or concepts in the domain. It also shows an *abstraction* of the conceptual 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 3.4, 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 3.5). 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]

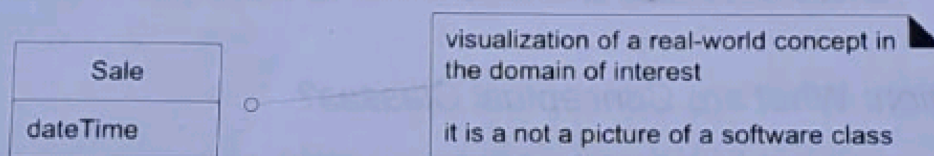


Figure 3.4 A domain model shows real-situation conceptual classes, not software classes.

---

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.

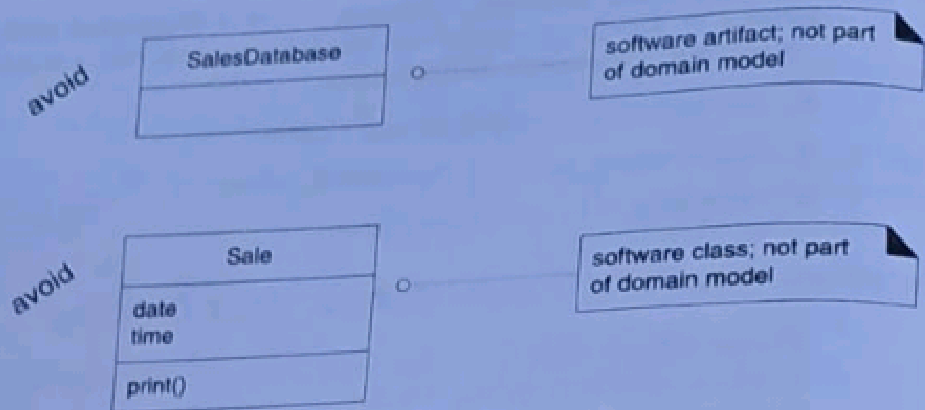


Figure 3.5 A domain model does not show software artifacts or classes.

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 3.6).

- **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.

3.7

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.

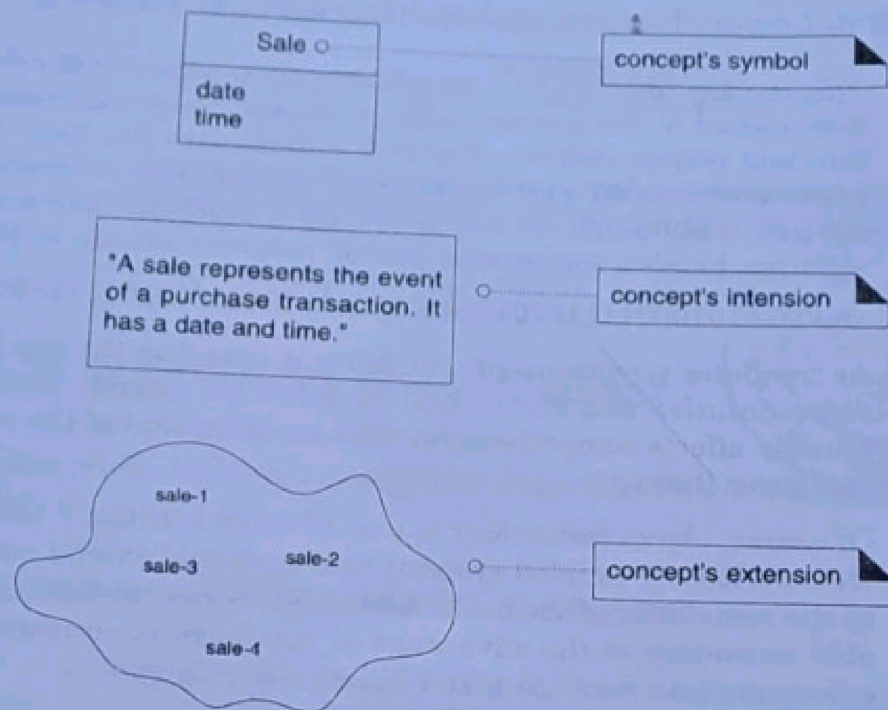


Figure 3.6 A conceptual class has a symbol, intension, and extension.

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. 152

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 3.7 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.

3

Guideline: How to Create a Domain Model?

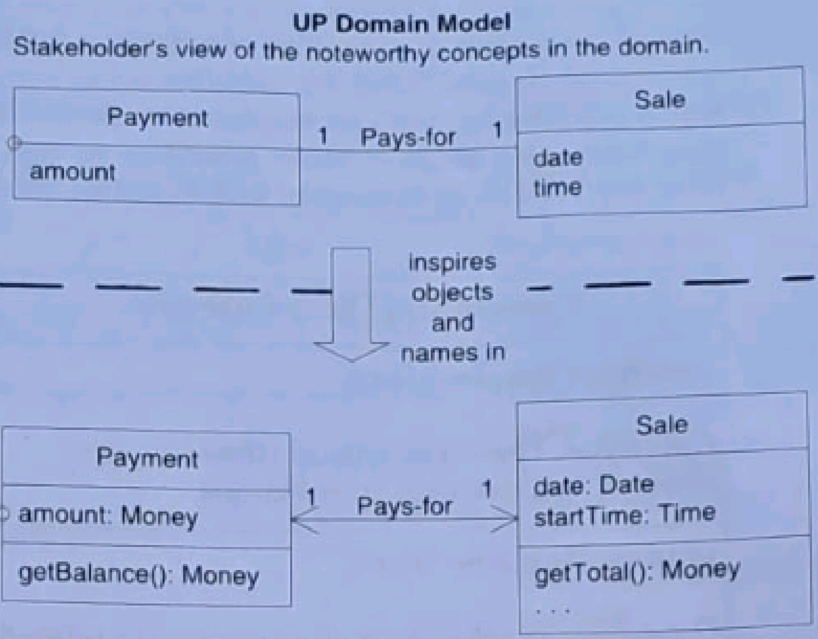
Bounded by the current iteration requirements under design:

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. 111 and p. 120.

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.

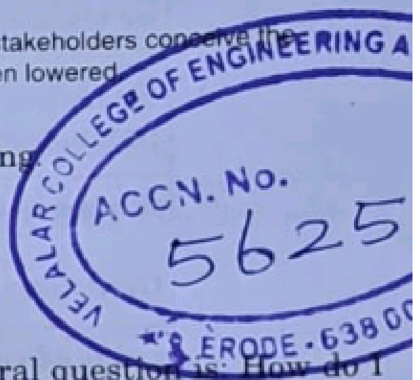
This is one of the big ideas in object technology.



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 3.7 Lower representational gap with OO modeling.



VELALAR COLLEGE OF ENGINEERING A...
ACCN. No. 5625
ERODE - 638 0..

9 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 Book* (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 3.2 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*

Conceptual Class Category	Examples
descriptions of things *Guideline*: See p. 109 for discussion.	*ProductDescription* *FlightDescription*
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 3.2 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.

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. 122 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.

degree of abstraction, stepping back from familiar non-OO designs, and listening carefully to the core vocabulary and concepts that domain experts use.

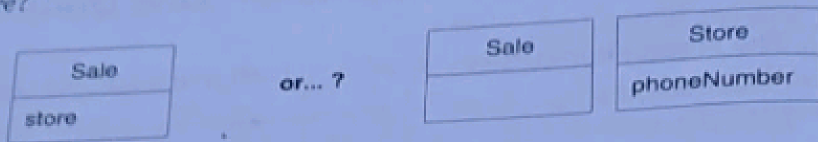
For example, here are candidate conceptual classes related to the domain of a telecommunication switch: *Message*, *Connection*, *Port*, *Dialog*, *Route*, *Protocol*.

3.16 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:

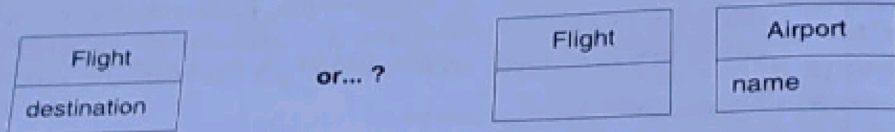
> *Guideline*
>
> If we do not think of some conceptual class X as a number or text in the real world, X is probably a conceptual class, not an attribute.

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.

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 3.10.

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.

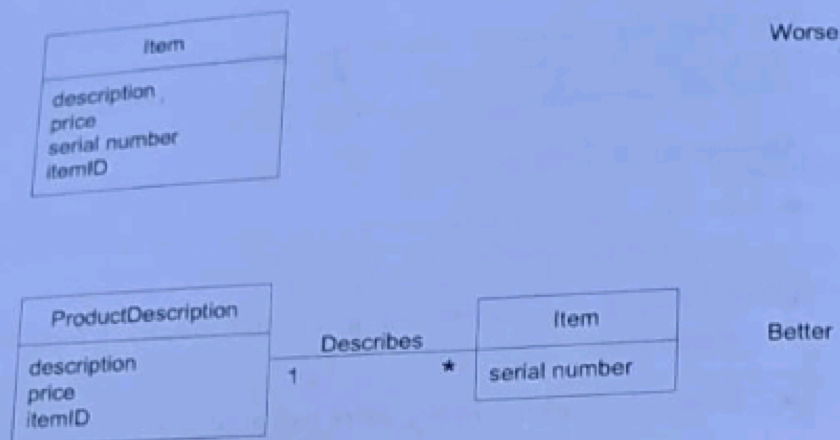


Figure 3.10 Descriptions about other things. The * means a multiplicity of "many." It indicates that one *ProductDescription* may describe many (*) *Items*.

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

3.18

FlightDescription that describes a flight and its route, even when a particular flight is not scheduled (see Figure 3.11).

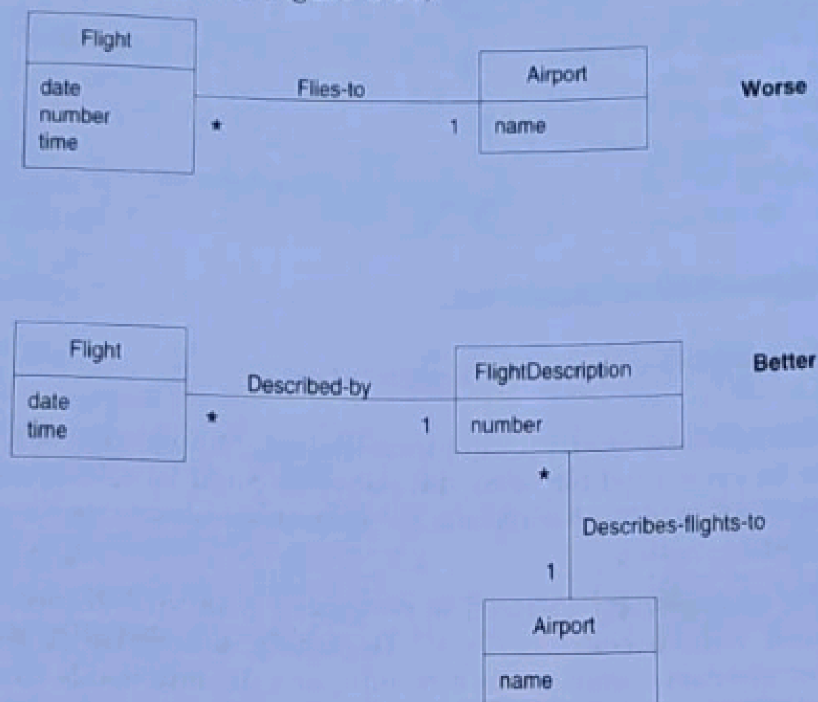


Figure 3.11 Descriptions about other things.

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.

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 3.12).

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

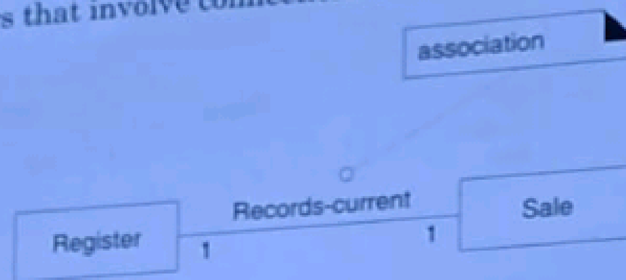


Figure 3.12 Associations.

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 3.13.

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.

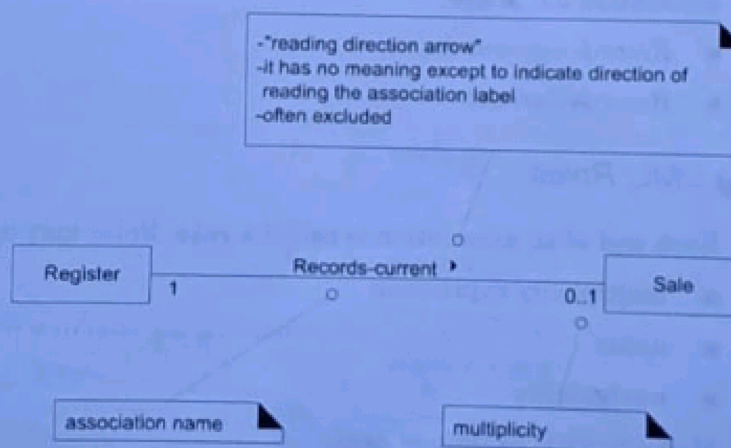


Figure 3.13 The UML notation for associations.

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 3.13).

> *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 3.14).

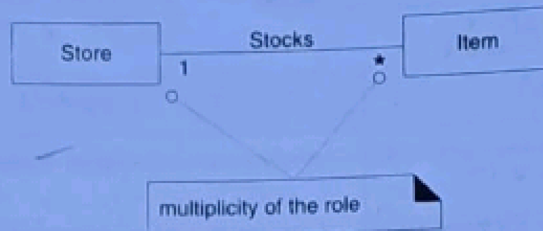


Figure 3.14 Multiplicity on an association.

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 3.15.

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.

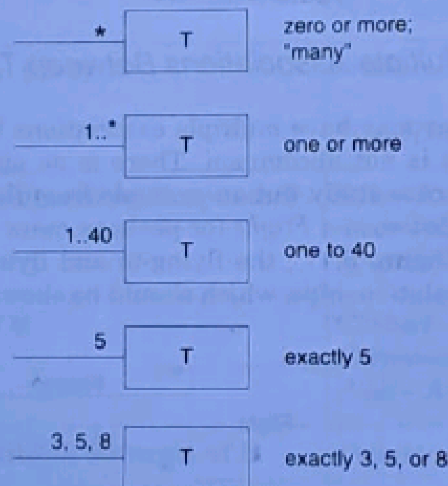


Figure 3.15 Multiplicity values.

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 3.16 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 3.16 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 3.17); the flying-to and flying-from associations are distinctly different relationships, which should be shown separately.

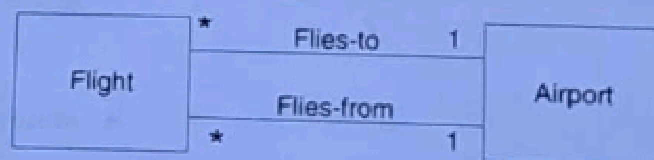


Figure 3.17 Multiple associations.

Guideline: How to Find Associations with a Common Associations List

Start the addition of associations by using the list in Table 3.3. 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*
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*