DESIGN CLASS DIAGRAMS

1. Introduction

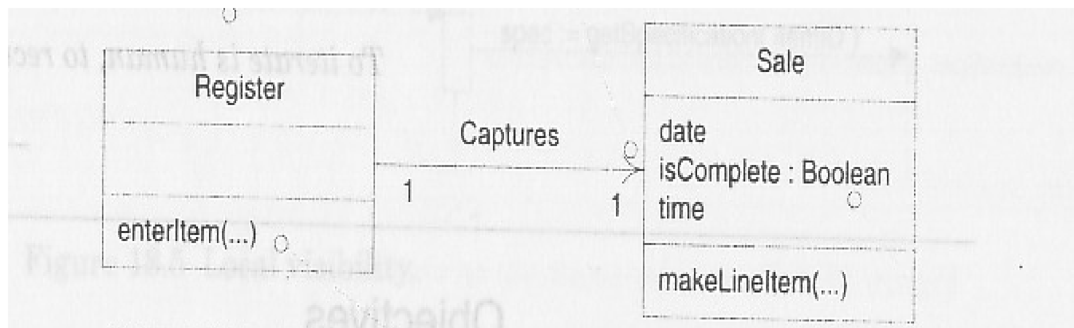
With the completion of interaction diagrams for use-case realizations, it is possible to identify the specification for the software classes (and interfaces) that participate in the software solution, and annotate them with design details, such as methods.

1.1. When to Create DCDs

DCDs are usually created in parallel with interaction diagrams. Many classes, method names and relationships may be sketched out very early in design by applying responsibility assignment patterns, prior to the drawing of interaction diagrams. It is possible and desirable to do a little interaction diagramming, then update the DCDs, then extend the interaction diagrams some more, and so on.

1.2. Example DCD

Here is an example of DCD with Register and Sale. The diagram consists of the methods of each class, attribute type information, and attribute visibility and navigation between objects along with the basic associations and attributes, which are created during Domain Modeling.



1.3. Terminology related to DCD

A design class diagram (DOD) illustrates the specifications for software classes and interfaces (for example, Java interfaces) in an application. Typical information includes;

- classes, associations and attributes
- interfaces, with their operations and constants
- methods
- attribute type information
- navigability
- dependencies

In contrast to conceptual classes in the Domain Model, design classes in the DCDs show definitions for software classes rather than real-world concepts.

2. Steps in creating DCD

- Identify software classes

? Illustrate them by Class Diagrams

? Add Method names

? Add Type information

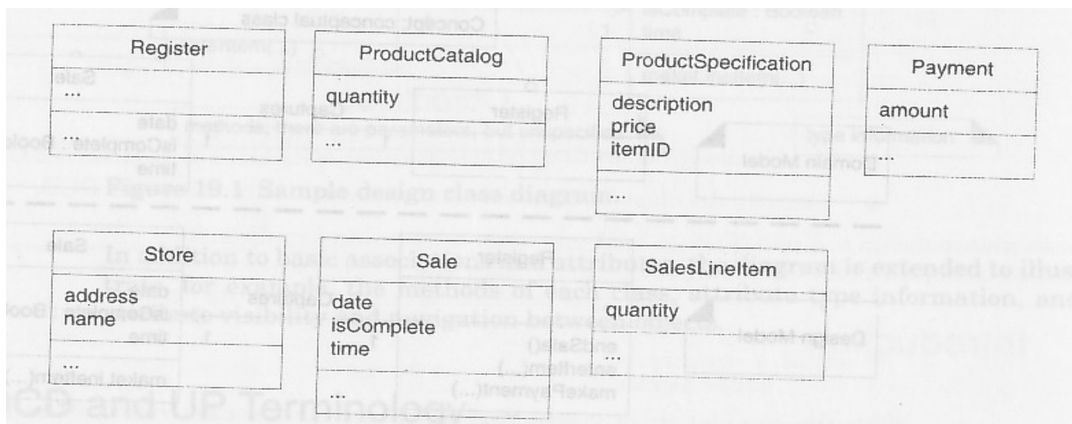
? Add Association and Navigability

? Add Dependency relationship

2.1. Identify Software Classes and Illustrate Them by Class Diagrams

The first step in the creation of DCDs as part of the solution model is to identify those classes that participate in the software solution. These can be found by scanning all the interaction diagrams and listing the classes mentioned.

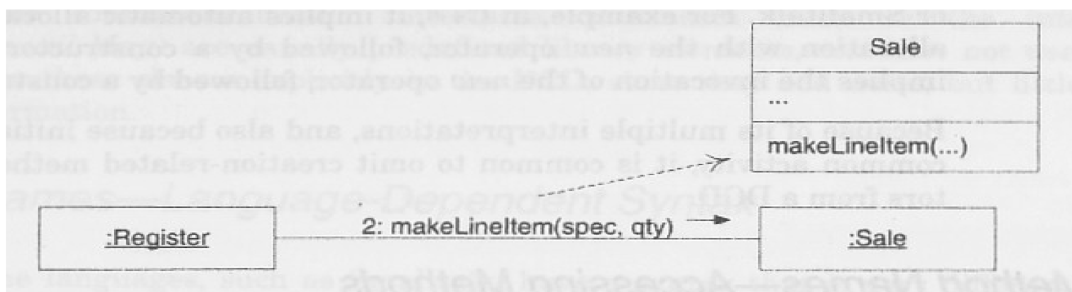
The next step is to draw a class diagram for these classes and include the attributes previously identified in the Domain Model that are also used in the design.



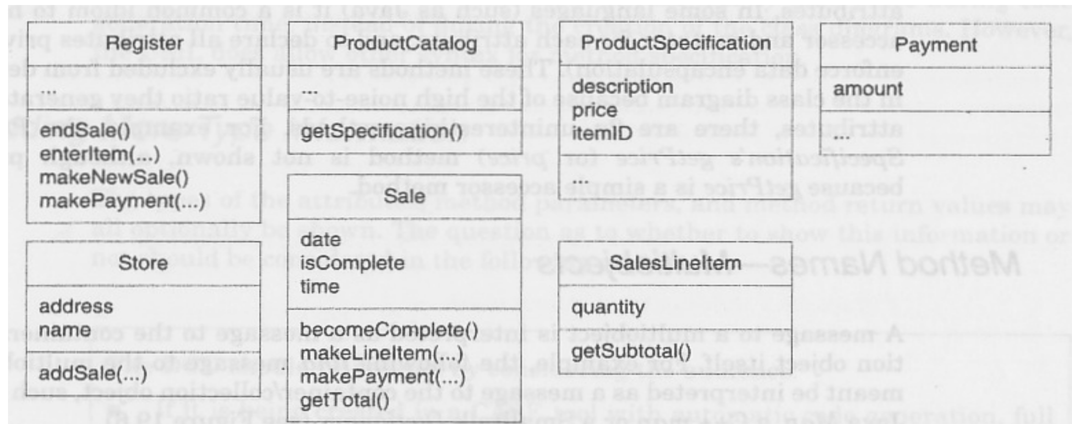
Note that some of the concepts in the Domain Model, such as Cashier, are not present in the design. There is no need—for the current iteration—to represent them in software. However, in later iterations, as new requirements and use cases are tackled, they may enter into the design. For example, when security and log-in requirements are implemented, it is likely that a software class named Cashier will be releva nt.

2.2. Add Method Names

The methods of each class can be identified by analyzing the interaction diagrams, For example, if the message *makeLineltem* is sent to an instance of class Sale, then class Sale must define a *makeLineItem* method.



In general, the set of all messages sent to a class X across all interaction diagrams indicates the majority of methods that class X must define.



2.2.1. Method Name Issues

The following special issues must be considered with respect to method names:

? interpretation of the create message

? depiction of accessing methods

? interpretation of messages to multiobjects

? language-dependent syntax

2.2.1.1.Method Names—create

The create message is a possible UML language independent form to indicate instantiation and initialization. When translating the design to an object oriented programming language, it must be expressed in terms of its idioms for instantiation and initialization. There is no actual create method in C++, Java, or Smalltalk. For example, in C++, it implies automatic allocation, or free store allocation with the new operator, followed by a constructor call. In Java, it implies the invocation of the new operator, followed by a constructor call.

Because of its multiple interpretations, and also because initialization is a very common activity, it is common to omit creation-related methods and constructors from a DCD.

2.2.1.2.Method Names—Accessing Methods

Accessing methods retrieve (accessor method) or set (mutator method) attributes. In some languages (such as Java) it is a common idiom to have an accessor and mutator for each attribute, and to declare all attributes private (to enforce data encapsulation). These methods are usually excluded from depiction in the class diagram because of the high noise-to-value ratio they generate; for n attributes, there are 2n uninteresting methods. -

2.2.1.3.Method Names—Multiobjects

A message to a multiobject is interpreted as a message to the container/collection object itself. These container/collection interfaces or classes (such as the interface java.util.Map) are usually predefined library elements, and it is not useful to show these classes explicitly in the DUD, since they add noise, but little new information.

2.2.1.4.Method Names—Language-Dependent Syntax

Some languages, such as Smalltalk, have a syntax that is very different from the basic UML format of methodName{pararncterList}. It is recommended that the basic UML format be used, even if the planned implementation language uses a different syntax. The translation should ideally take place

during code generation time, instead of during the creation of the class diagrams. However, the UML does allow other syntax for method specification.

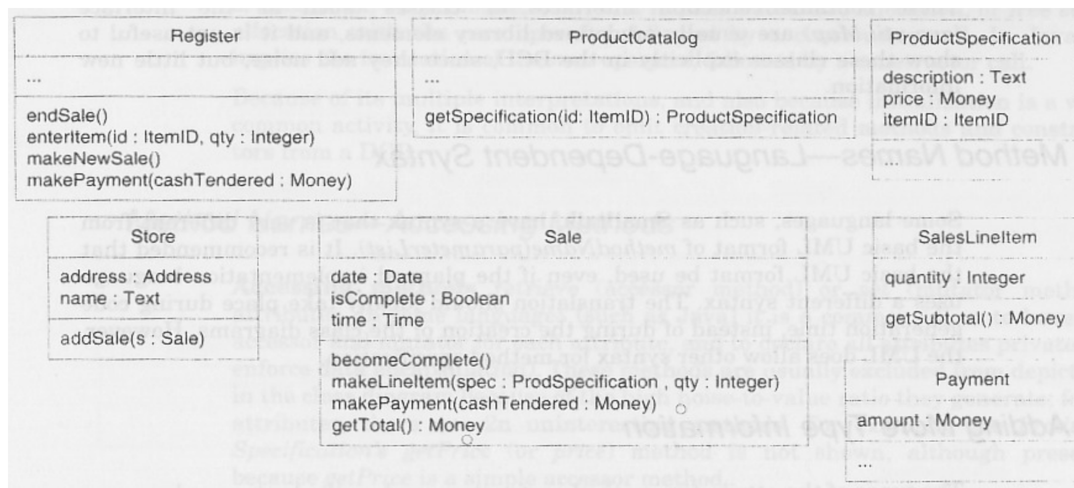
2.2.2. Adding More Type Information

The types of the attributes, method parameters, and method return values may all optionally be shown. The question as to whether to show this information or not, should be considered in the following context:

The DCD should be created by considering the audience.

- If it is being created in a CASE tool with automatic code generation, full and exhaustive details are necessary.
- If it is being created for software developers to read, exhaustive low-level detail may adversely affect the noise ratio.

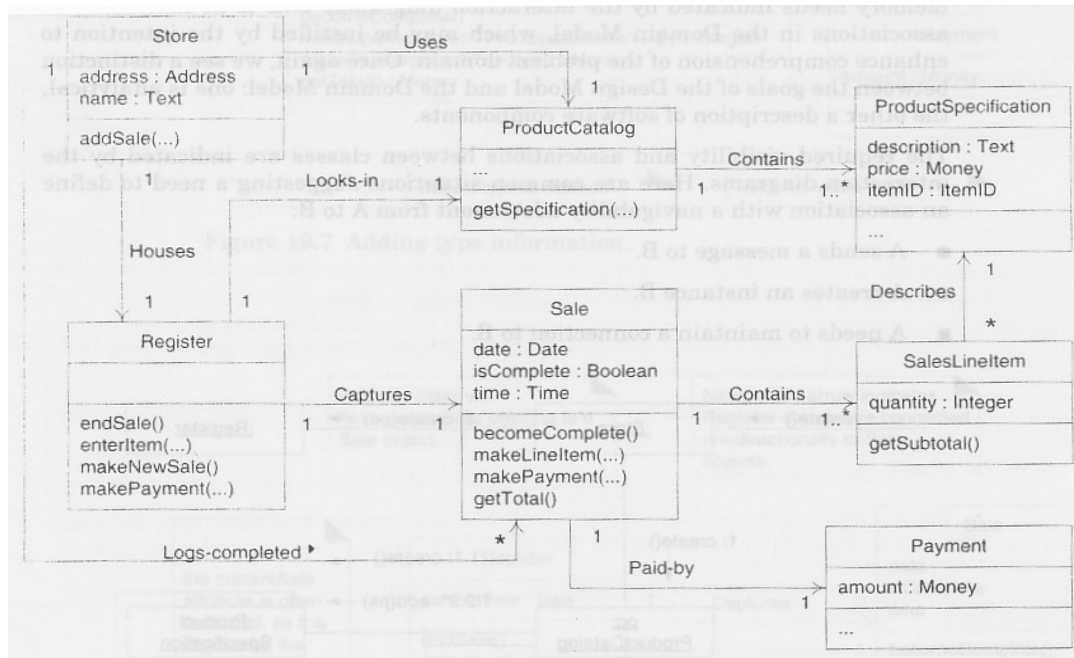
For example, is it necessary to show all the parameters and their type information? It depends on how obvious the information is to the intended audience.



2.2.3. Adding Associations and Navigability

Each end of an association is called a role, and in the DCDs, the role may be decorated with a navigability arrow. Navigability is a property of the role that indicates that it is possible to navigate uni-directionally across the association from objects of the source to target class. Navigability implies visibility. The usual interpretation of an association with a navigability arrow is attribute visibility from the source to target class.

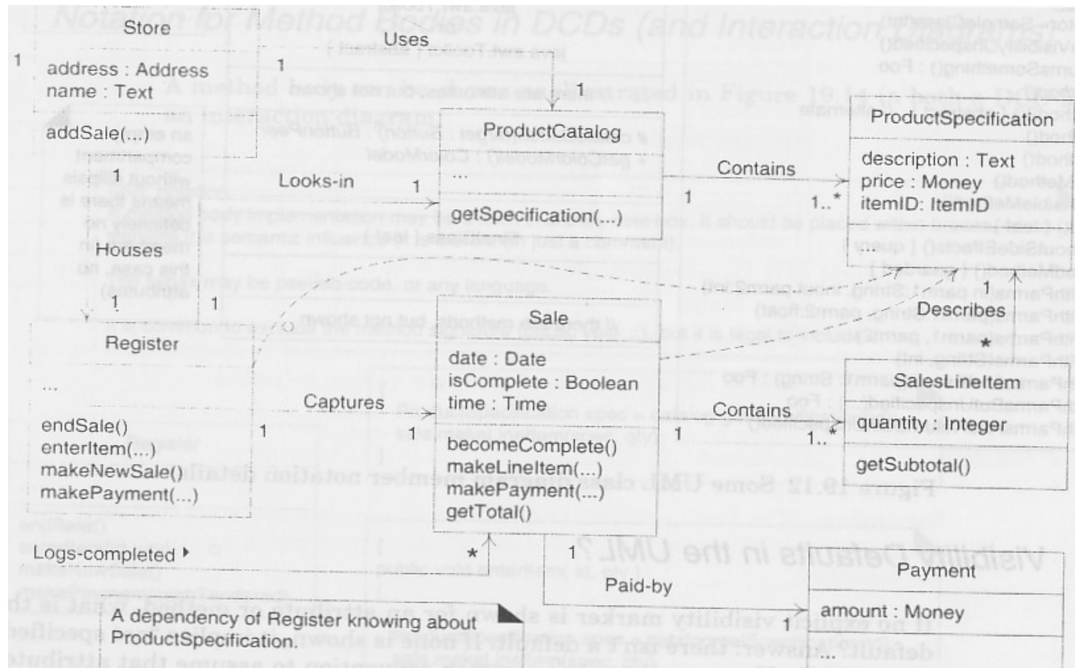
During implementation in an object-oriented programming language it is usually translated as the source class having an attribute that refers to an instance of the target class. For instance, the Register class will define an attribute that references a Sale instance. Most, if not all, associations in DCDs should be adorned with the necessary navigability arrows. The required visibility and associations between classes are indicated by the interaction diagrams.



2.2.4. Adding Dependency Relationships

The UML includes a general dependency relationship, which indicates that one element (of any kind, including classes, use cases, and so on) has knowledge of another element. It is illustrated with a dashed arrow line. In class diagrams the dependency relationship is useful to depict non-attribute visibility between classes; in other words, parameter, global, or locally declared visibility. By contrast, plain attribute visibility is shown with a regular association line and a navigability arrow. For example, the Register software object receives a return object of type *ProductSpecification* from the specification message it sent to a *ProductCatalog*. Thus Register has a short-term locally declared visibility to *ProductSpecifications*. And Sale receives a *ProductSpecification* as a parameter in the *makeLineItem* message; it has parameter visibility to one.

These non-attribute visibilities may be illustrated with the dashed arrow line indicating a dependency relationship. There is no significance in the curving of the dependency lines; it is graphically convenient.



2.3. DCDs Within the UP

Inception—The Design Model a DCDs will not usually be started until elaboration because it involves detailed design decisions, which are premature during inception

Elaboration—During this phase DCDs will accompany the use-case realization interaction diagrams; they maybe created for the most architecturally significant classes of the design.

Note that CASE tools can reverse-engineer (generate) DCDS from source code. It is recommended to generate DCDs regularly from the source code, to visualize the static structure of the system.

Construction—DCDs will continue to be generated from the source code as an aid in visualizing the static structure of the system.