Module 1

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 1 Study Guide and Deliverables

Module Themes:	• Introduction and Design Principles • Unified Modeling Language (UML) • Class Diagram, Sequence, Use Case, State
Readings:	• Module 1 online content - Appendices section is optional

Module 1 Part 1: Introduction and Design Principles

Principles of Software Design: Overview

Every software application must have requirements (i.e., a set of required outcomes) and an implementation (i.e., code). Software design fits in between. It is, in effect, a means of communicating to humans how the implementation fulfills the requirements. Real applications consist of thousand or millions of lines. Unless they are very well organized (i.e., based on a good design) they can be difficult—sometimes impossible—and very expensive (mostly in person-hours) to fix or perfect.

In this module, we will be concerned with the desired qualities of software designs.

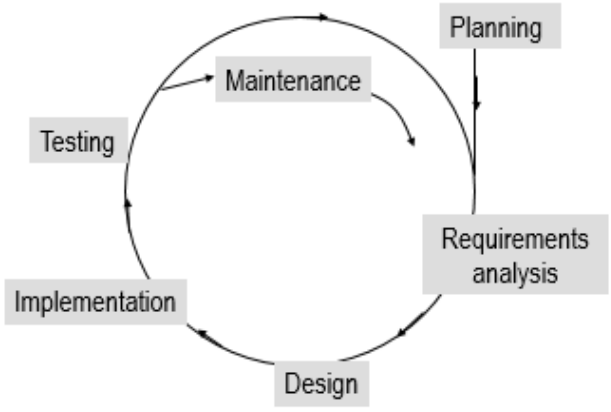
Learning Objectives

After successfully completing this part of the module, students will be able to do the following:

1. Retain the goals of software design.
2. Use various design models for a single application and explain the reasons.
3. Compare use case models, class models, data flow models, and state models.
4. Use frameworks in design.

Recall that software development consists of various phases, shown in the figure, and that we typically transition among them. Most commonly, the agile methodology transitions through requirements analysis → design → implementation → testing of additional functionality every two weeks.

The Software Development Lifecycle



All figures in Module 1 are adapted from Software Design: From Programming to Architecture by Eric J. Braude (Wiley 2003), with permission.

Goals of Software Design

Let's discuss what we want from software designs.

- **Sufficiency**: handles the requirements.

  First and foremost, they have to do the job of illustrating how requirements are going to be met. This is the "sufficiency" quality. How detailed a design has to be depends on many factors, and will be discussed throughout this course.

- **Understandability**: can be understood by intended audience.

  Since software designs are a means of human communication, they have to be *understandable*.

- **Modularity**: divided into well-defined parts.

  A closely related quality is *modularity*: how effectively the design is decomposed into parts. Decomposition is a key factor in making something understandable.

- **Cohesion**: organized so like-minded elements are grouped together.

  Parts that are *not* decomposed should belong together. This is the property of *cohesion*.

  > Example
  >
  > For example, if we are designing an animal shelter, the decomposition into dog area, cat area, and horse area results in cohesive groupings.
  >
  > On the other hand, a decomposition into 0-10 inches tall, 10-20 inched tall, and 20-30 inches would not result in cohesive groupings (thus low cohesion).

- **Coupling**: organized to minimize dependence between elements.

  We should also look for *coupling*: the width of the connection between groups. We want coupling to be low.

  > Example – continued
  >
  > The dog area / cat area / horse area decomposition has low coupling because of their separation. The 0-10 inches tall / 10-20 inched tall / 20-30 inches decomposition; on the other hand, has very high coupling.
  >
  > For example, cat food attendants would need to move among several of these, and so would dog food attendants.

- **Robustness**: can deal with wide variety of input.

  We want programs to be able to handle imperfect conditions. This "robustness" quality is largely implemented at the code level but we also try to design for it.

- **Flexibility**: can be readily modified to handle changes in requirements.

  Requirements change for various reasons. For example, the business in which the application operates may change; or the customer may decide that the evolving application has undesirable features. The code must change accordingly. The question of how the code has been organized—i.e., the application's design—now becomes critical. Anticipating this, designs should allow for change: they must be *flexible*.

- **Reusability**: can use parts of the design and implementation in other applications.

  Creating software is expensive, and we try to reuse parts whenever we can. For this reason, we want our designs to promote *reusability*.

- **Information hiding**: module internals hidden from others.

  When you operate a car, you use the accelerator, brake, and steering wheel. You should not be contemplating at the same time how each of these achieves its functionality. In other words, information is deliberately hidden from the users of accelerators, brakes, and steering wheels. Such *information hiding* is effective for all design. It helps us to concentrate on one thing at a time..

- **Efficiency**: executes within acceptable time and space.

- **Security**: promotes defense against deliberate harm.

  Efficiency is an obvious quality, and anything a design can do to promote it is beneficial. Efficiency of algorithms is, by itself, an old subject of continuing importance. It assisted at the code level as well. The same can be said for *security*.

- **Reliability**: executes with acceptable failure rate.

  Reliability refers to the average time that an application operates without evidence of a defect. Anything that a design can do to promote reliability is beneficial, and it, too, is assisted at the code level.

Test Yourself as You Learn.

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer first, and then click "Show Answer" to compare yours to the suggested answer.

Test Yourself 1.1

Without checking the list, recall four goals for software design.

Suggested answer: any four goals in the list below:

Sufficiency, Understandability, Modularity, Cohesion, Coupling, Robustness

Flexibility, Reusability, Information hiding, Efficiency, Security, Reliability

Test Yourself 1.2

For an application that controls a nuclear weapon, which design criterion would be most important?

Suggested answer: Sufficiency

Test Yourself 1.3

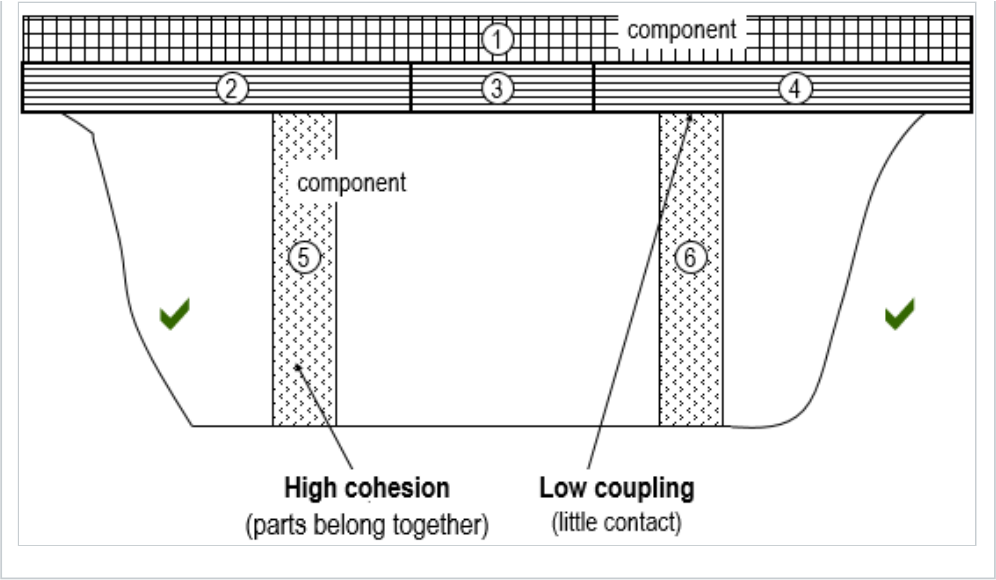
For an application that displays the latest models in you automobile showroom, which would be the least important?

Suggested answer: Robustness

Selected Goals in Detail

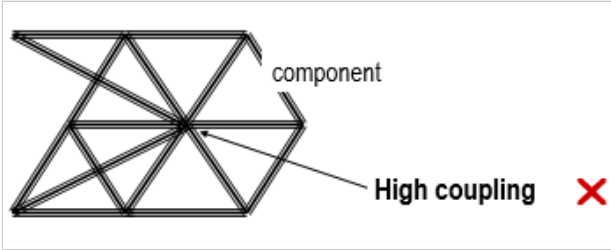
In the example of bridge construction in Figure "Cohesion and Coupling: Bridge Example", the components have high cohesion. For example, the two supporting pillars are presumably composed of a single, highly integrated material such as reinforced concrete. On the other hand the six components contact each other in an economical way rather that all depending on each other. For example, component 3 depends on 2 and 4, 4 depends only on 6 and 4, and 5 depends on none. The highest dependency ("coupling") is a coupling of 2. If they were all stacked in a teepee shape, for example, every component might be dependent on a number of others.

Cohesion and Coupling: Bridge Example



The figure "Questionable Architecture: High Coupling in Truss" illustrates how the high coupling (at the intersection of 7 components) is a bottleneck in that its failure has significant consequences. The software analogy is having seven components, each of which depends on the other six.

Questionable Architecture: High Coupling in Truss



Aspects of Flexibility

Consider design flexibility. Keep in mind that this does not concern the behavior of the application—only the ease or difficulty of altering or adding to it.

There are three ways in which functionality can be altered, as shown in the bullets. So you consider the application in the advent of either.

1. **Obtaining more or less of what's already present**

   Example: handle more kinds of accounts without needing to change the existing design or code

2. **Adding new kinds of functionality**

   Example: add *withdraw* to existing *deposit* function

3. **Changing functionality**

   Example: allow withdrawals to create an overdraft

Types of Reuse

Let's consider the quality of reuse in a little more depth. Here we summarize major ways in which software and its design can be reused.

The wider the community in which re-use is contemplated, the more components are needed for a unit of reuse. For example, the Python community makes extensive use of libraries. Each library typically consists of a substantial number of classes. At the other extreme, when an individual programmer is considering reusing a module in the same program, that module can be as simple as a single class or function. The bullets in the figure support this, and show examples of various kinds of collections.

We can reuse …

- **Object code (or equivalent)**
  Example: sharing dll's between word processor and spreadsheet
- **Classes — in source code form**
  Example: Customer class used by several applications. Thus, we write generic code whenever possible
- **Assemblies of Related Classes**
  Example: the java.awt package
- **Patterns of Class Assemblies**

Software Design Example

In this section, we'll take a very simple example and use it to discuss some of the design qualities.

Requirements for Command Line Calculator Example

The list below shows some requirements for the sample application. There are many ways in which this application can be designed.

Requirements for Command Line Calculator Example

1. CommandLineCalculator begins by asking the user how many accounts they want to open. It then establishes the desired number, each with zero balance.
2. CommandLineCalculator asks the user which of these accounts they want to deal with.
3. When the user has selected an account, CommandLineCalculator allows the user to add a whole numbers of dollars to, or subtract them from the account for as long as he requires.
4. When the user is done with an account, they are permitted to quit, or to pick another account to process.

Example One of Command Line Calculator Application

The figure "Typical I/O For Command Line Calculator" shows the operation (or "execution") of our application. Note that, even if this were a GUI, it would not be the "design" of the application in the sense that we use that word in this course. (In the context of "UI Design" it would be considered "design." But ours is not a course about UI or UX design.)

Let's consider the qualities required of a design for this application.

Typical I/O For Command Line Calculator

```
================== How many accounts do you want to open: ==================
11
The application will deal with 11 accounts.


---------- Please enter account number between 1 and 11 or type 'Quit application': ----------
2
    The balance in this account is 0

    Please enter the amount you want added, or type 'stop' to stop adding for now:
33
    Added 33, getting total of 33
    Please enter the amount you want added, or type 'stop' to stop adding for now:
44
    Added 44, getting total of 77
    Please enter the amount you want added, or type 'stop' to stop adding for now:
stop
    Addition ends.

---------- Please enter account number between 1 and 11 or type 'Quit application': ----------
5
    The balance in this account is 0

    Please enter the amount you want added, or type 'stop' to stop adding for now:
66
    Added 66, getting total of 66
    Please enter the amount you want added, or type 'stop' to stop adding for now:
stop
    Addition ends.

---------- Please enter account number between 1 and 11 or type 'Quit application': ----------
2
    The balance in this account is 77

    Please enter the amount you want added, or type 'stop' to stop adding for now:
88
    Added 88, getting total of 165
    Please enter the amount you want added, or type 'stop' to stop adding for now:
stop
    Addition ends.

---------- Please enter account number between 1 and 11 or type 'Quit application': ----------
Quit application

        ================ Application ends. ================
```

Problems With CommandLineCalculator Implementation (See appendix to this chapter)

The classes and their relationship should be selected so that the interaction shown can be readily implemented. *Robustness* in this case is manifested in the points shown in the list below—where the user does not enter expected values. The list also includes a reminder that the application is useless (unless it is never changed!) without appropriate method documentation.

Continuing with the desired qualities of the (anticipated) design, consider *reusability*. Most development shops are engaged in a sequence of application that have some commonality. For example, this application would probably be part of a suite of financial programs, and so reuse of various classes may be practical. Account and Customer come to mind, especially if they accumulate significant functionality.

Problems With CommandLineCalculator Implementation

- How do we know that all required functionality has been handled? (**Sufficiency**)
- If the user makes a mistake the system crashes or performs unpredictably (**robustness**)
  The following cause crashes:
  - Invalid number of accounts
  - Invalid account

- Invalid amount to add (not an integer)
- Invalid string (not "stop" or "Quit application")

- Clear what the method are intended to do? (**documentation**)
- Easy to modify, add or remove parts? (**flexibility**)
- Executes fast enough? (**speed efficiency**)
- Satisfies memory requirements? (**space efficiency**)
- Class usable for other applications? (**reusability**)

Key Concept: → Ensure Correctness ←

We are primarily responsible for ensuring that our code does what it is intended to.

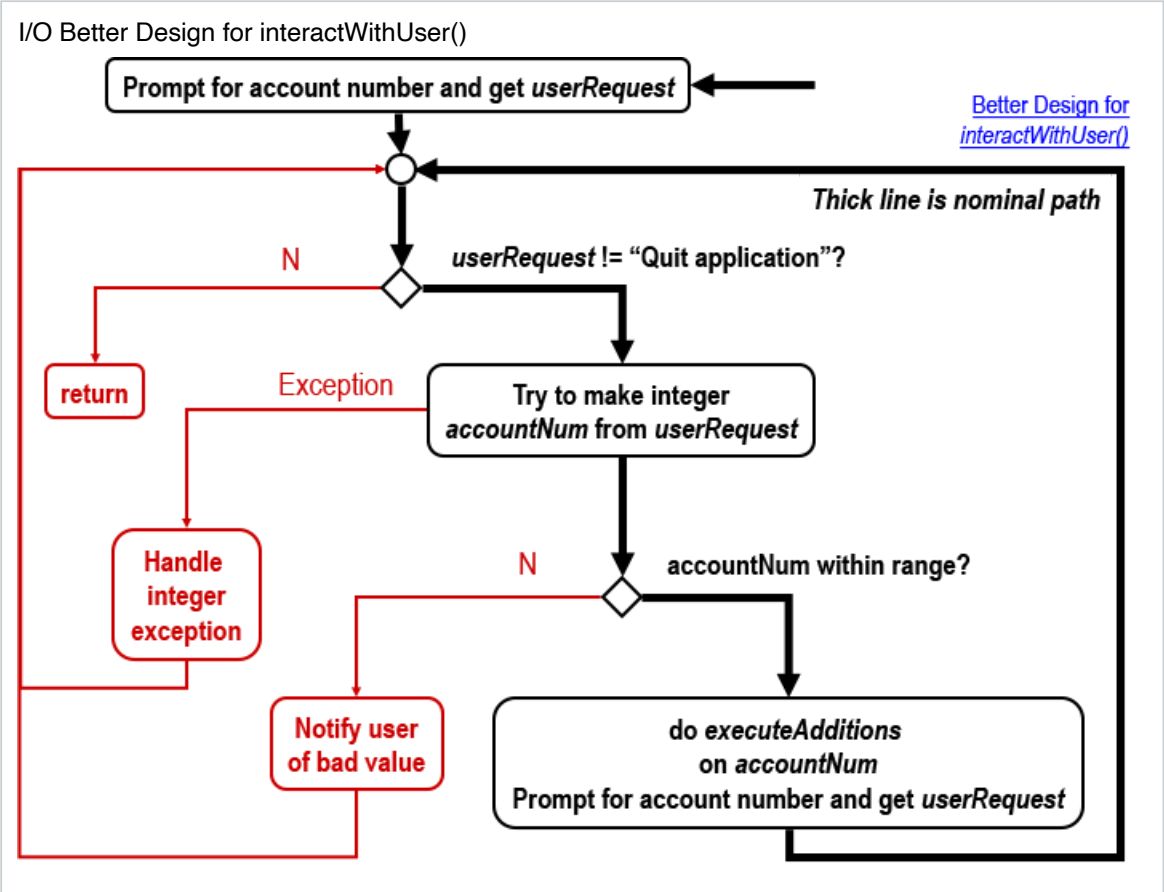
Example Two of Command Line Calculator Application

The figure below is a Command Line Calculator I/O. Let's view this as an activity diagram (UML flowchart), as in the figure that follows, where we have made the design more robust.

I/O For Robust Command Line Calculator

```
================== How many accounts do you want to open: ==================
xyz
Sorry, not an integer: Try again.

================== How many accounts do you want to open: ==================
11
The application will deal with 11 accounts.


---------- Please enter account number between 1 and 11 or type 'Quit application': ----------
22
Please enter a legal account number.

---------- Please enter account number between 1 and 11 or type 'Quit application': ----------
3
    The balance in this account is 0

    Please enter the amount you want added, or type 'stop' to stop adding for now:
abc
    Sorry -- incorrect entry: Try again.
    Please enter the amount you want added, or type 'stop' to stop adding for now:
44
    Added 44, getting total of 44
    Please enter the amount you want added, or type 'stop' to stop adding for now:
55
    Added 55, getting total of 99
    Please enter the amount you want added, or type 'stop' to stop adding for now:
stop
    Addition ends.

---------- Please enter account number between 1 and 11 or type 'Quit application': ----------
6
    The balance in this account is 0

    Please enter the amount you want added, or type 'stop' to stop adding for now:
77
    Added 77, getting total of 77
    Please enter the amount you want added, or type 'stop' to stop adding for now:
stop
    Addition ends.

---------- Please enter account number between 1 and 11 or type 'Quit application': ----------
Quit application

        ================ Application ends. ================
```

In the figure "Better Design for interactWithUser()", the black arrows show the "normal" (usually called the nominal) path. The red parts are included for robustness, and are usually tied to decision points and to operations that mayß cause exceptions.



Key Concept: → Good Code May Not Be Good Design ←

The code implied by the activity diagram is more robust, but does it exploit object-orientation and exhibit a clear design?

If it's not easy to verify, it would be inflexible and unlikely to be reused.

Recall the robustness property. An example of a non-software artifact that's not robust is a parking garage entry device that jams when a user slides their card into the wrong slot, and the boom stays stuck in down position while cars back up.

Key Concept: → Write Robust Code ←

Good designs withstand anomalous treatment.

Remaining Problems With CommandLineCalculator

The bullets in the list below are explicit about the limitations of the presumed design on the Calculator application. For example, including multiplication can easily be done but the question is whether the current design allows for this in a natural manner.

It is realistic to view an application as being part of a larger future application. This influences one's design thinking. So the naturalness of where multiplication goes becomes key. Think of it like a book in a library: if properly shelved, there is no problem but when not properly shelved, the book can be effectively lost and unused. While development environment tools mitigate this, they are not substitutes; and poor designs generally depreciate faster in value.

Remaining Problems With CommandLineCalculator

- Insufficient *flexibility*
  - To add subtraction, multiplication etc.
  - To change the nature of the project
- Speed *efficiency* not explored
- Space efficiency not explored
  - Limit to number of accounts?
- *Reusability* doubtful
  - OO not leveraged
- No visualization of design provided

Test Yourself as You Learn.

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer first, and then click "Show Answer" to compare yours to the suggested answer.

Test Yourself 1.4

Would making the Calculator design object-oriented probably make it more flexible?

Suggested answer - Yes.

Test Yourself 1.5

Is an activity diagram a good way to assess efficiency?

Suggested answer - Yes.

Test Yourself 1.6

Is an activity diagram a good and sufficient way to ensure reusability?

Suggested answer - No.

Evaluating Designs

When you want to modify your house to accommodate objectives, you naturally compare various alternative designs. The same should apply to software designs. In this section, we'll discuss how such comparisons can be performed.

Comparisons require criteria, and we'll use the ones listed in the notes—as shown on the left. The columns are hypothetical alternative designs (for now, it does not matter to us exactly what they are). A reasonable approach is to evaluate each design against each criterion—using a 4-point scale (0=worst, 2=best), for example.

Design Comparison Example

Quality Criteria	Candidates			
	Al Pruitt's	Parallel Communicating Processes	Event Systems	Layered
Sufficiency: handles the requirements	1	1	2	2
Understandability: can be understood by intended audience	0	2	1	2
Modularity: divided into well-defined parts	0	0	1	2
Cohesion: organized so like-minded elements are grouped together	1	0	2	2
Coupling: organized to minimize dependence between elements	0	1	0	1
Robustness: can deal with wide variety of input	1	0	2	1
Flexibility: can be readily modified to handle changes in requirements	1	0	0	0
Reusability: can use parts of the design and implementation in other applications	0	0	1	2
Information hiding: module internals are hidden from others	1	1	2	2
Efficiency: executes within acceptable time and space limits	1	2	0	1
Reliability:	0	1	1	2

TOTALS	**6**	**8**	**13**	**18**

Module 1 Part 2: Unified Modeling Language (UML)

Learning Objectives

In this section we review the Unified Modeling Language (UML).

After successfully completing this part of the module, students will be able to do the following:

1. Deal with collections of classes.
2. Relate classes.
3. Design runtime behavior of methods and objects.

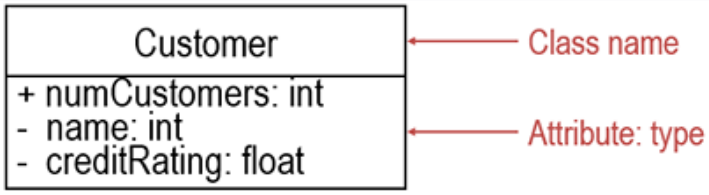
Classes

The "Classes at Detailed Design" figures illustrate all parts of a class representation. The most widely used UML feature is a rectangle —to represent a class.

Classes at Detailed Design

Class name: The most important part is the top inner rectangle, which holds the class name. We'll review the other parts of a class representation but it is only sometimes worthwhile to show them all. One usually wants to show key parts. As we will see, what usually counts the most are the relationships among the classes. If you add the details, it becomes possible to show only a few classes on a page, which is usually too restrictive.

Attributes: The second box lists the attributes, their type, and whether they are accessible only from within instances ("+") or not.



1 2

Click the above numbers 1 and 2 to browse different parts of a class representation.

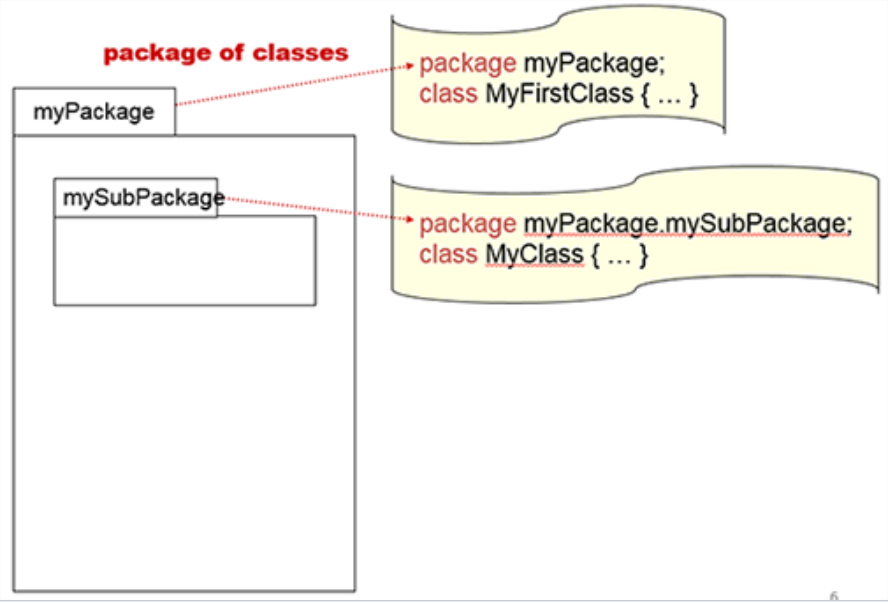
Collections of Classes

Much of the benefit of classes derives from their mutual relationships.

Look at the figures "UML Notation … and … Typical Implementation":

UML Notation … and … Typical Implementation

Packages (and their sub-packages) are collections of classes. For Java and Python, these correspond to the same term. Packages help us to organize designs. For example, 8 packages, each consisting of 10 classes, is much easier to think about than 80 classes.



1 2

Click the above numbers 1 and 2 to see collections of classes example.

Test Yourself as You Learn.

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer first, and then click "Show Answer" to compare yours to the suggested answer.

Test Yourself 1.7

What are the three major parts of a class in UML?

Suggested answer - name, attributes, and methods.

Test Yourself 1.8

Which of the three major parts of a class in UML is most important?

Suggested answer -name.

Test Yourself 1.9

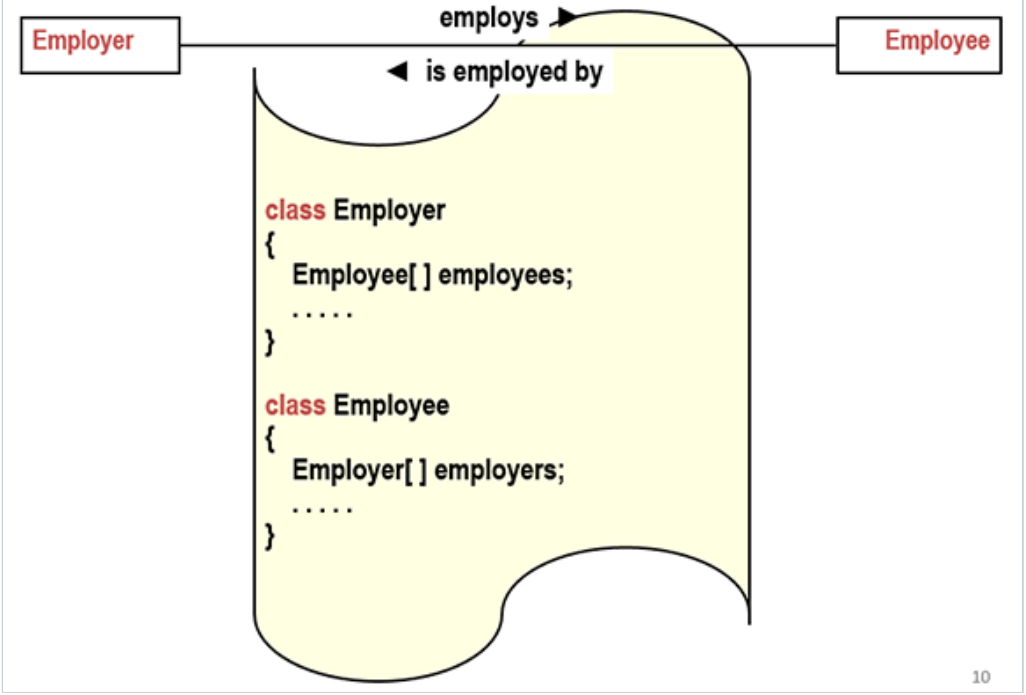
Is more information in a busy design diagram usually better than less?

Suggested answer - No.

Relationships between Classes

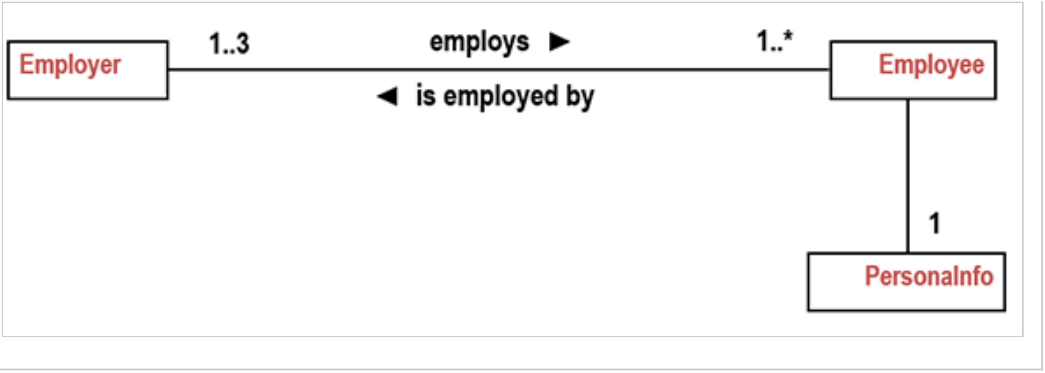
Association. Association is a popular kind of class relation, and can be labeled in each direction with an arrow head, as shown. In Eric Braude's opinion, it is better to use aggregation, which focuses on each direction separately, and which is discussed below.

Association: UML Notation



Multiplicity. It is usually useful to note multiplicity—the range showing how many objects on one class relate to another. For example, in the figure, an *Employer* instance employs one or more *Employee* objects. The word "employs" is for the reader's convenience, and may not occur in the code. The model also specifies that an employee can have up to three employers.

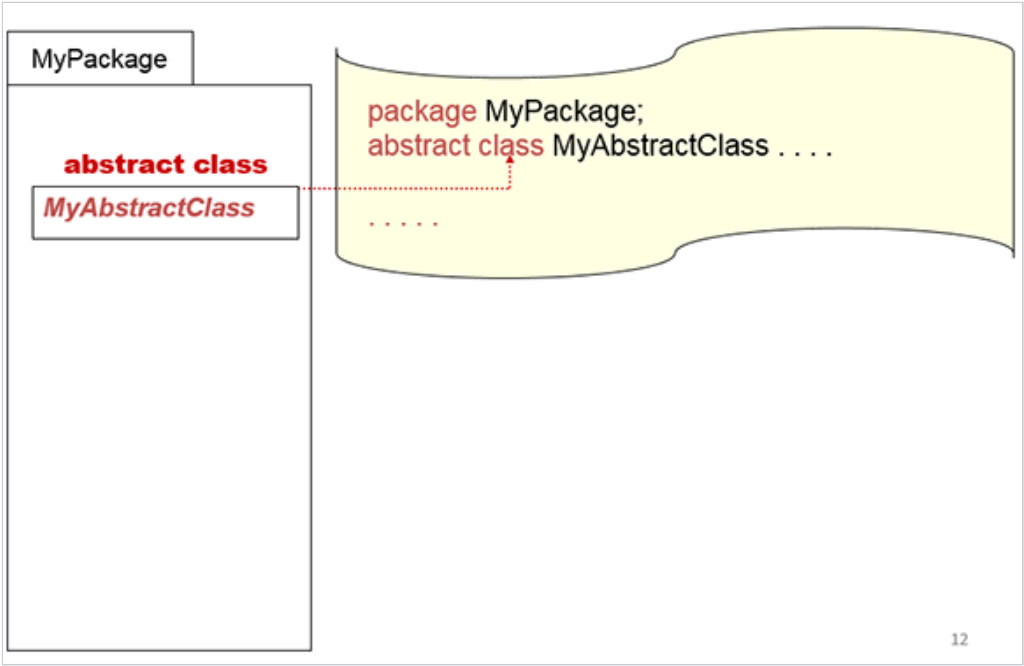
Multiplicity: UML Notation



As we will elaborate on throughout the course, one should design against interfaces rather than implementations. This is an old principle, really. As an example, imagine designing a tow truck. You don't design this against Chevy Malibu's alone (an implementation of *Car*), but against cars in general. The latter is the interface of cars for this situation, which consists of four wheels and an available neural gear. This is an abstract concept. Hence the use of abstract classes or Java Interfaces, which UML shows via italics. Interfaces provide the form of methods (name, argument types, and return types).

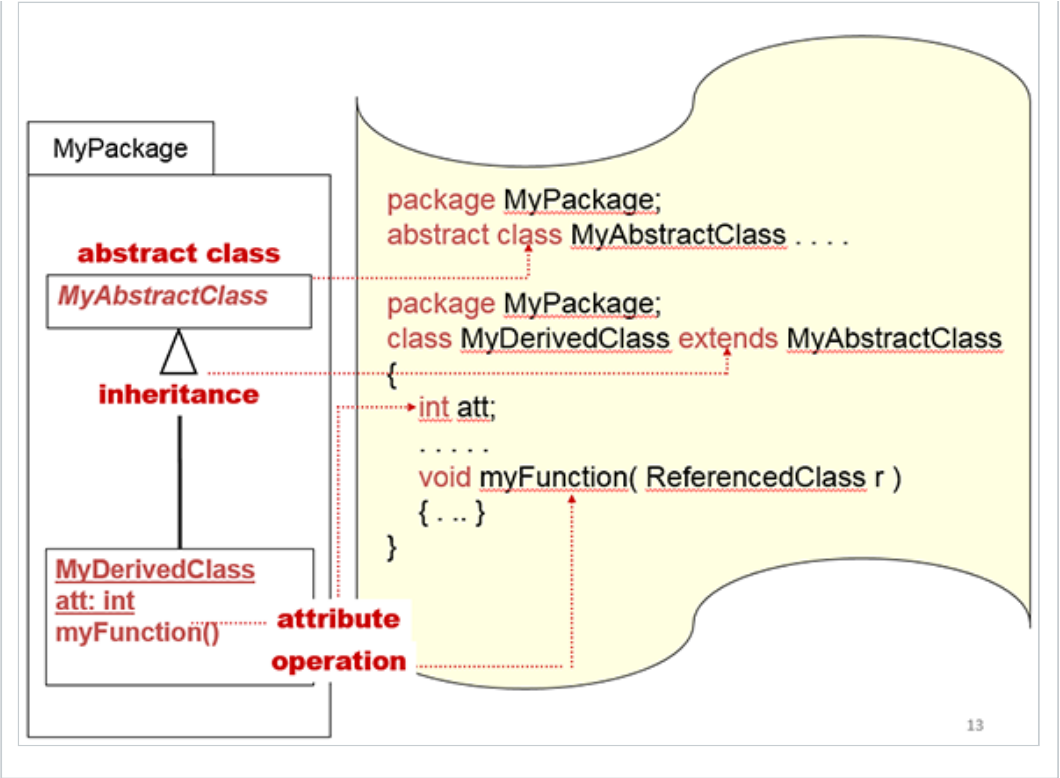
This idea is somewhat language-inspired. For example, Python does not do things this way (read Python Glossary duck-typing).

UML Notation … and … Typical Implementation

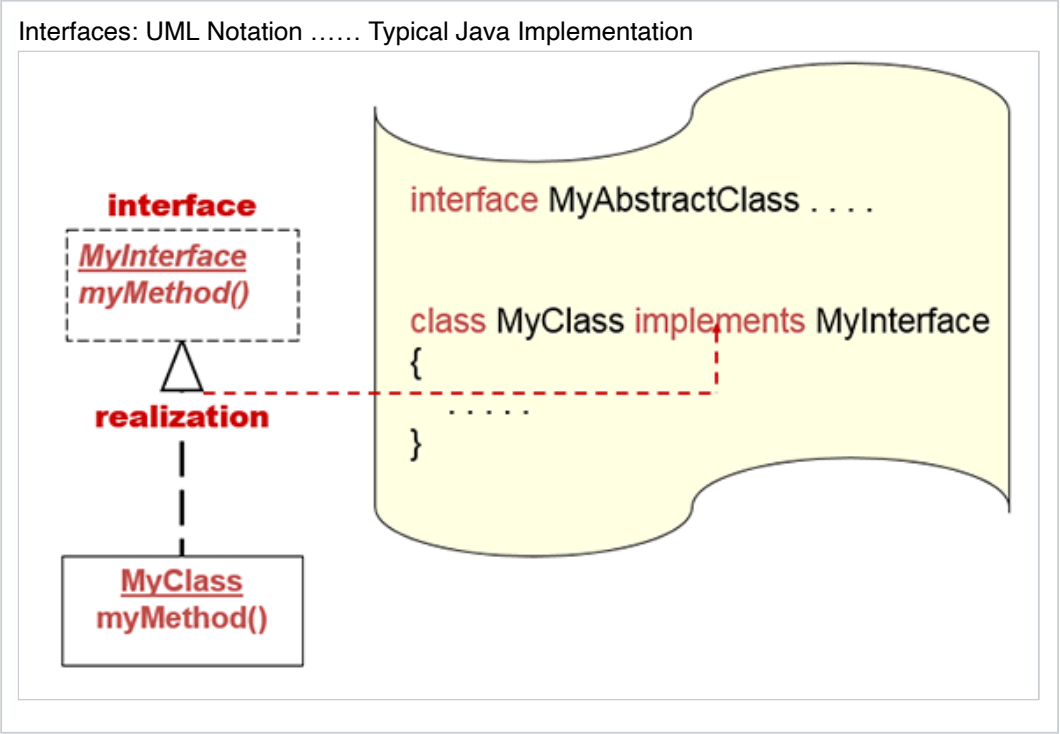


Inheritance. The figure "UML Inheritance … and … Typical Implementation" shows the example of code expanded with inheritance. (We assume that the reader already understands inheritance.)

UML Inheritance … and … Typical Implementation



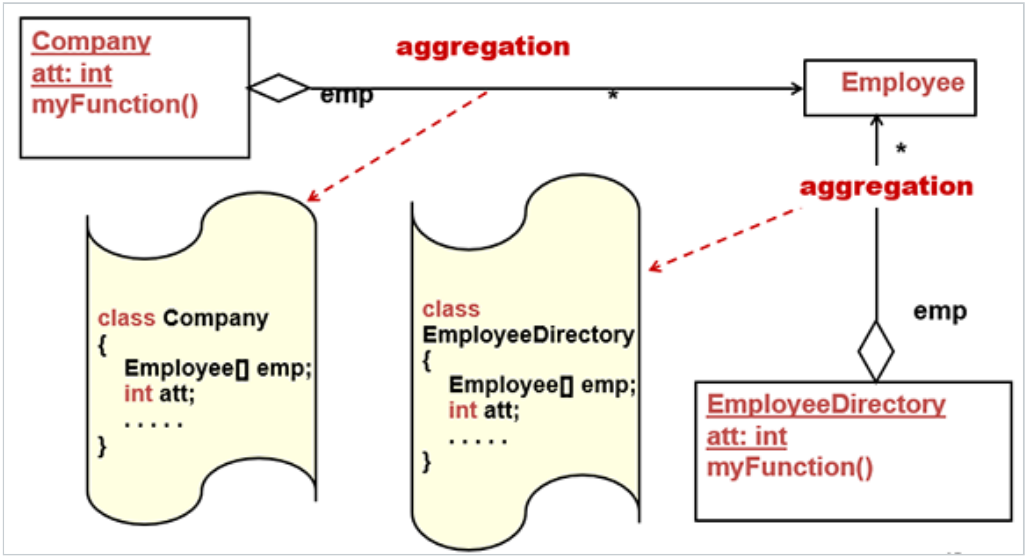
Realization. *Realization* is where a class satisfies an interface, and is shown with a dotted line in UML. In other words, it possesses the methods promised by the interface.



Aggregation. It is common for objects of one class to contain or be associated—one way—with objects of another. Aircraft objects "contain" (have) wings, for example. This is called aggregation, and is shown in UML with a diamond and arrow, as in the figure. The figure also shows how this is almost always implemented as a variable or attribute of a class.

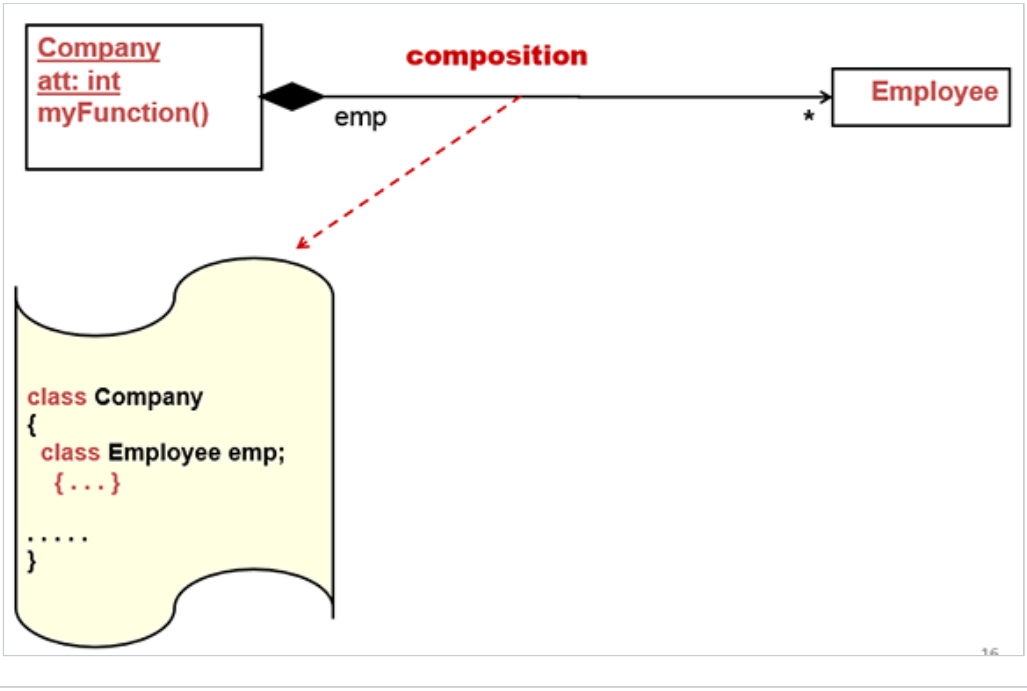
Just as wings can exist independent of airplanes (e.g., during manufacturing), aggregated objects can exist without the objects that aggregate them. Several classes can aggregate the same class.

Aggregation: UML Notation and Typical Implementation



Composition. Sometimes objects are not meant to exists without objects that aggregate them. This may be an intention of the designer, so the notion depends on the context. For example, each wing in an application may need always to be considered part of a particular plane. In that case, we apply the design principle to *enforce what's intended*. UML shows this strong kind of aggregation as composition, using a solid diamond shape, as in the figure. It indicates that in the context of the application, Employees exist only in the context of a company.

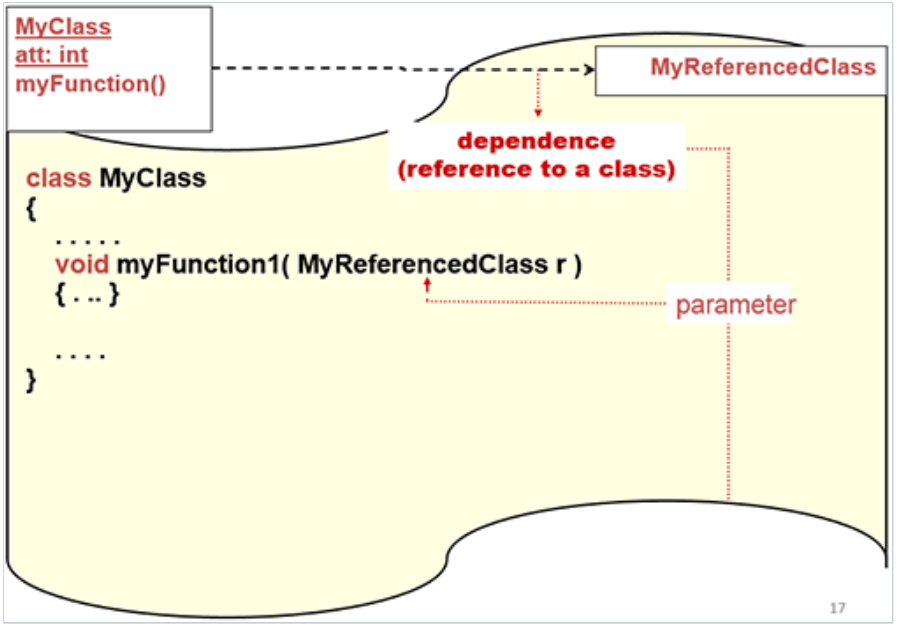
Composition: UML Notation and Typical Implementation



Dependence.

Dependence: UML Notation … and … Typical Implementation

Inheritance and aggregation are *dependencies* of one class on another. But they are not the only ones. For example, a method parameter may be of a class type, which is, in effect, a dependence of the current class on another class.



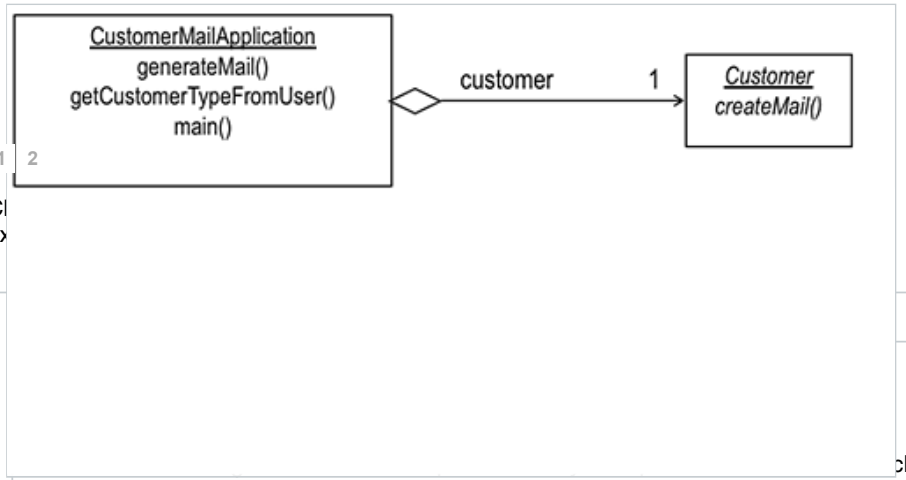
1 2

Click the above numbers 1 and 2 to see how different types of dependence occur.

Let's look at an example that combines aggregation with inheritance and dependence.

Customer Mail Application

Suppose that we are building a mail application pertaining to customers. This might require the aggregation shown in the figure.



ch question, think carefully, figure out your own answer first, and then click "Show Answer" to compare yours to the suggested answer.

Test Yourself 1.10

What are the two major kinds of class dependencies?

Suggested answer - aggregation and inheritance.

Test Yourself 1.11

Name two other kinds of class dependencies.

Suggested answer - where the dependent class is mentioned as the type of a method parameter or return.

Test Yourself 1.12

What can we conclude about instances when a class X has concrete subclasses A and B?

Suggested answer - instances of X are instances of A or B.

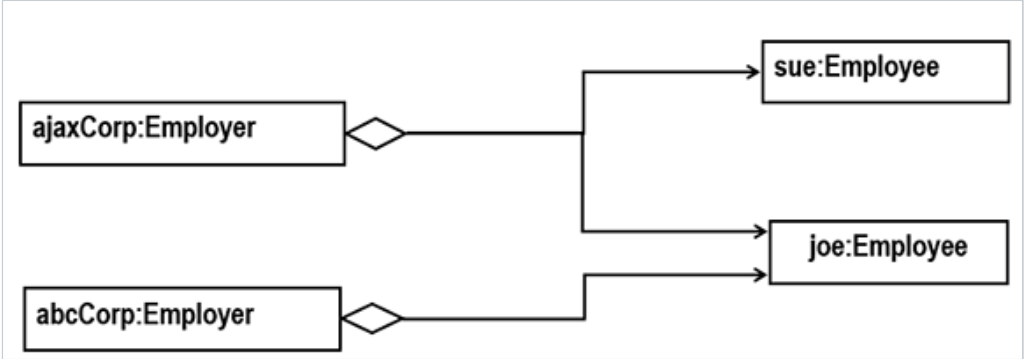
Runtime Executions

Class models show the building blocks of a design and their interrelationships. They say nothing much about instances of the classes or about the order in which methods are invoked.

Class models are powerful. Each class model can typically give rise to many object models (relationships among objects). The figure shows an example of one of the many object models that could realize the simple class model shown.

Example of an Object Model - Class Model

Example of an Object Model - One Particular Derived Object Model



Sequence diagrams in the figure "Sequence Diagram for Check Out Use Case" show the order in which a use case (or scenario or user story) is executed.

Use Case

1. User swipes bar code
2. Application prompts for rental duration
3. User enters duration
4. Application stores record
5. Application prints customer's account status

They concern methods associates with objects, and are read top-down, and in the direction of the arrows. We denote objects with <class name>:<object name> such as Person:maryJones and Person:johnSmith. When there is no need to distinguish among instances of a class Person, or when there is only one, we use the notation :Person.

Every function invocation involves the initiator (an object of a class or else a user) and the object that hosts the performing method.

Building a Sequence Diagram

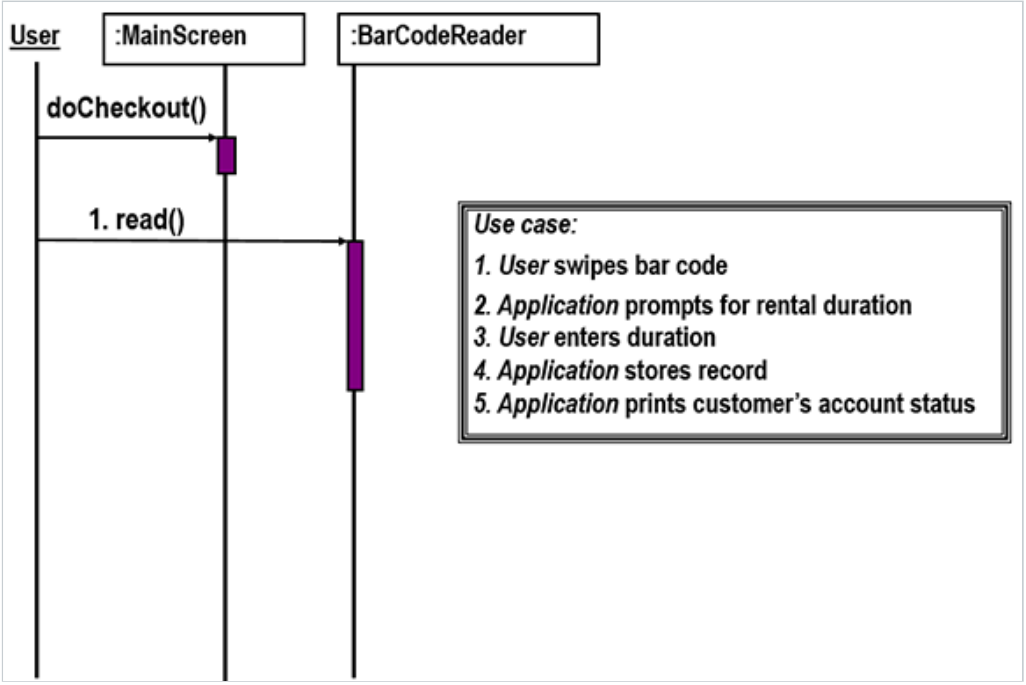
The numbered items explain the order in which one develops a sequence diagram.

1. Identify the use case whose sequence diagram you will build (if applicable).
2. Identify which entity initiates the use case
    - the user, or
    - an object of a class
        - name the class
        - name the object if possible
3. If not the user, draw a rectangle to represent this initiating object at left top
    - use UML *object:Class* notation

4. Draw an elongated rectangle beneath this to represent the execution of the operation initiating the process
5. Draw an arrow pointing right from it
6. Identify which entity handles the operation initiated
    - an object of a class
        - name the class
        - name the object
7. Label the arrow with the name of the operation
8. Show a process beginning, using an elongated rectangle
9. …… Continue with each new statement of the use case.

The figure "Sequence Diagram for Check Out Use Case" shows shorthand for the user physically touching something (typically a button) on the main screen, which invokes doCheckout() (the user can't actually invoke a method directly). The numbering keys the invocation to the corresponding step in the use case.

Sequence Diagram for Check Out Use Case

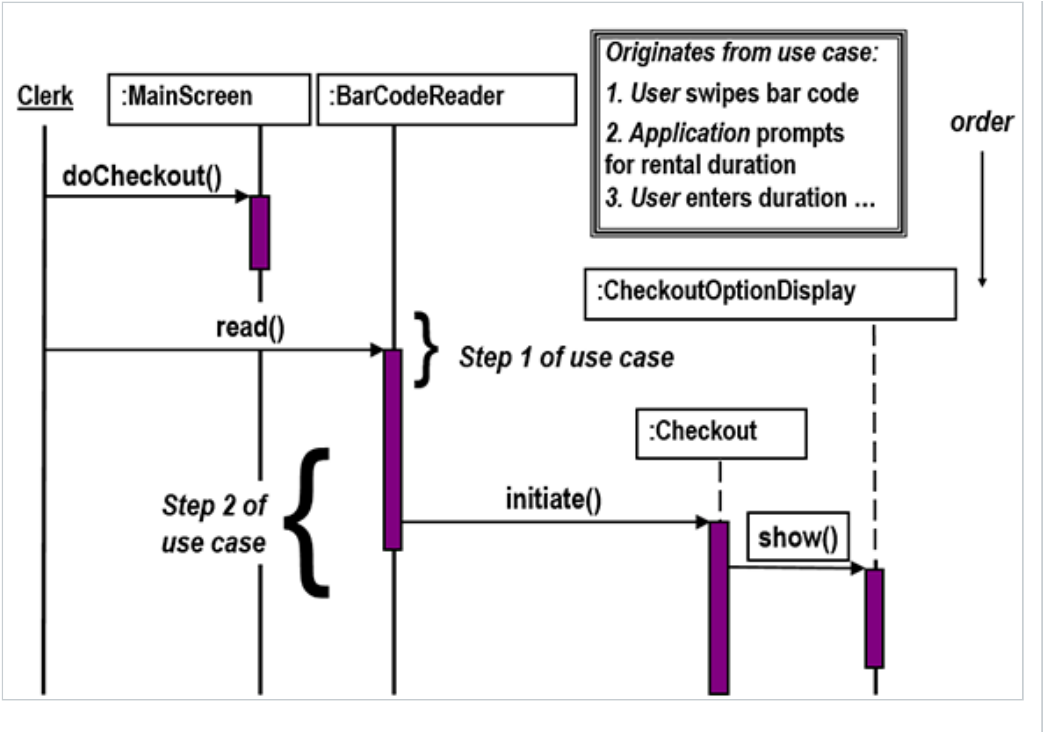


The rest of this sequence diagram shows how the designer intends to implement the use case.

Originates from use case:

1. User swipes bar code
2. Application prompts for rental duration
3. User enters duration …

Beginning of Sequence Diagram for Check Out Use Case

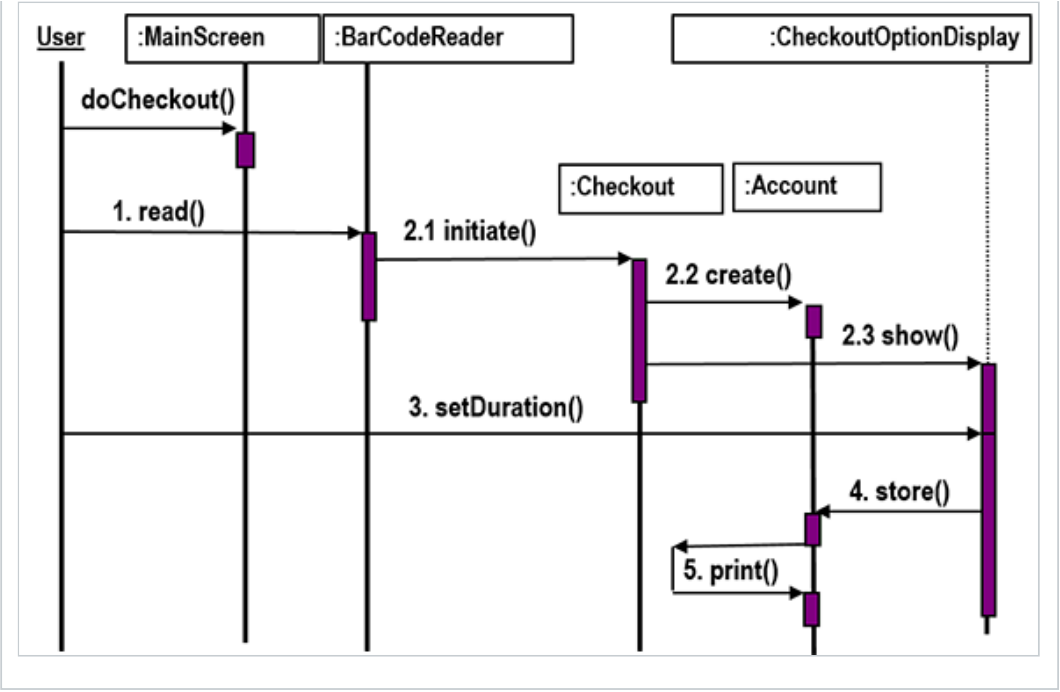


Sequence diagrams below can show one method of an object invoking another, as with print(). The realization also decomposes the steps for a single use case step.

This sequence diagram originates from use case:

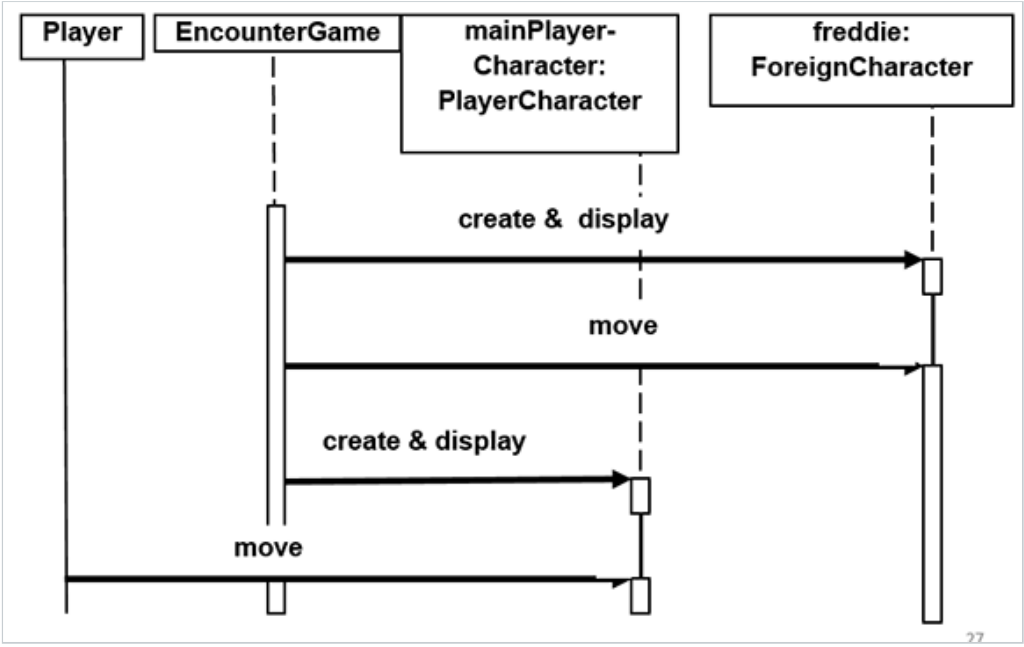
1. User swipes bar code
2. Application prompts for rental duration
3. User enters duration
4. Application stores record
5. Application prints customer's account status

Sequence Diagram for Check Out Use Case



UML allows us to show methods that execute in parallel, using half-arrows, as in the figure "Sequence Diagram Showing Concurrency". This example is a role-playing video game in which the player and Freddie the foreign character more in parallel.

Sequence Diagram Showing Concurrency



State and State Transitions

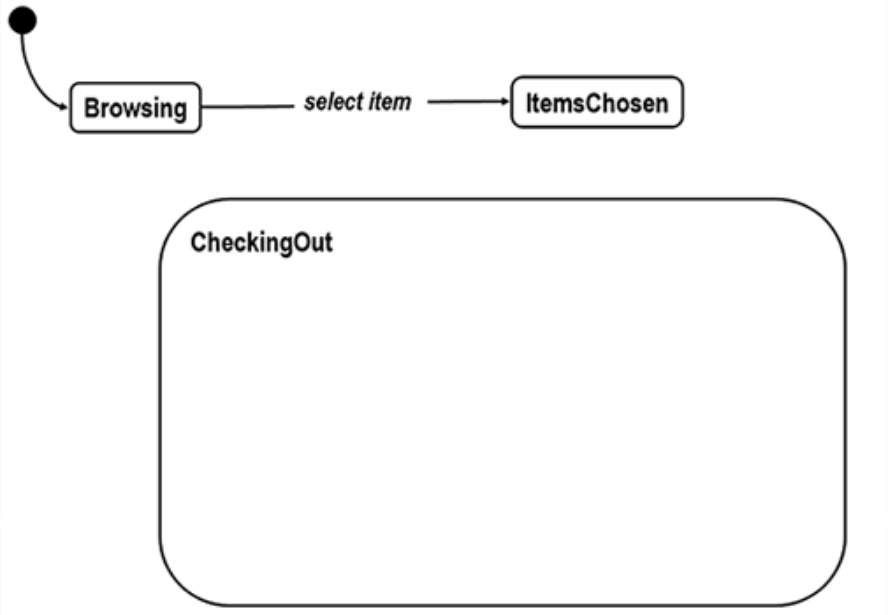
Recall that software designs are described from various perspectives that, taken together, provide a whole design. Class models are one perspective. We discuss state models next.

States apply to objects. A state diagram usually applies to the objects of a single class. For example, if a person (an object of Person) goes for a run, she may be in *warm-up* state, *running* state, or *cool-down* state. Notice that it's not the class (in this case *Runner*) that is in one state or another. Different *Runner* objects can be in different states at the same time.

Intuitively, a state is a status or mode. But these are merely synonyms: the real definition of state has to do with the value of variables of the instance. For example, assume that a *Runner* object has two variables: *breathing_rate* and *speed*. You can recognize that runner jane is in warm-up state from the combination of values *breathing_rate*< 18 and *speed* < 5, in *running* state for *speed* > 5, or *cool-down* state for *breathing_rate*>= 18 and *speed* < 5.

State/Transition Diagram for OnlineShopper Class

The figure number 1 shows the beginnings of a state model. The black dot denotes the automatic starting state of every instance. The instance transitions to a new state only when the action indicated takes place. Actions are events external to the application, such as a user pressing a button.

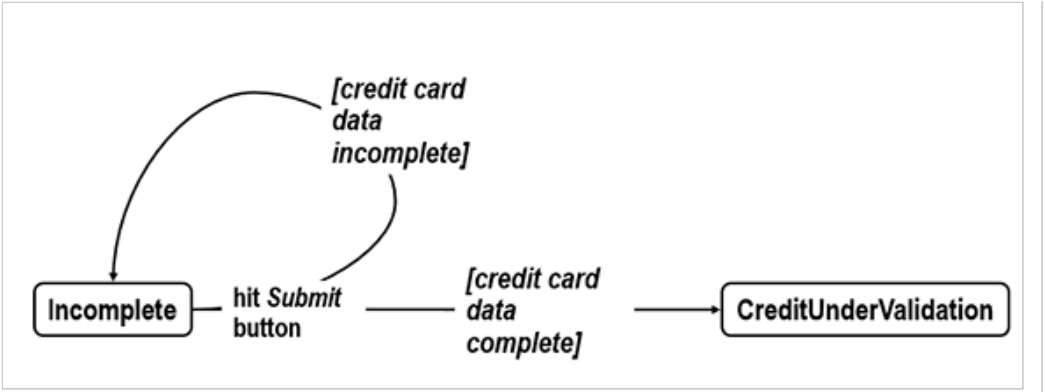


1 2

Click the above number 1 to see the beginnings of a state model; click the number 2 to see the details of each state.

Once can place conditions on actions so that their transitions are affected. Conditions are represented in square brackets, as showing in the figure "Conditions on Events".

Conditions on Events



Activity Diagrams ("Flowcharts")

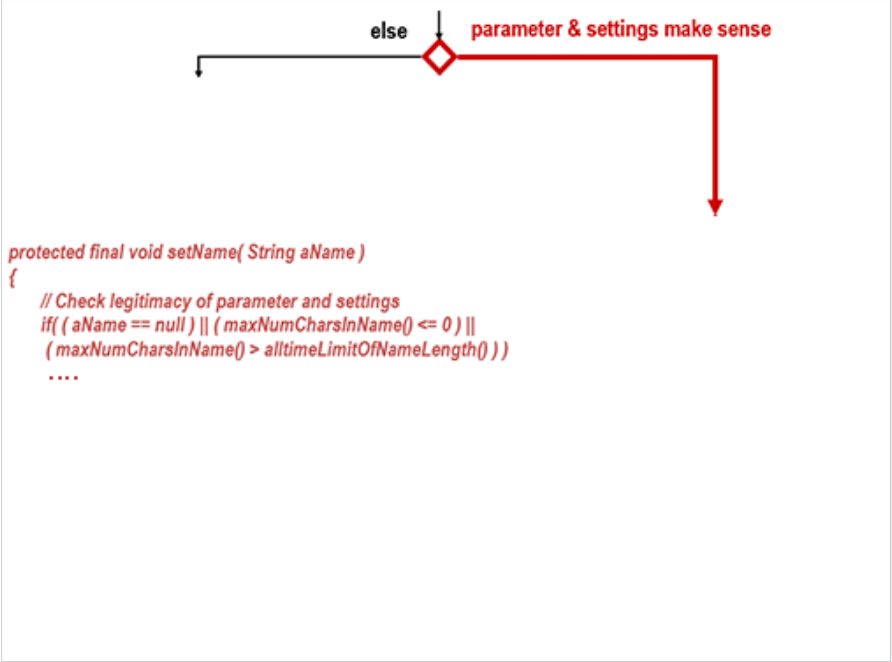
Classes give the building blocks of a design, sequence diagrams show the order of method executions, and state-transition models specify the reactions of instances to events. Individual methods can be written directly in code but flowcharts are sometimes used first for key functions because they are graphical. In UML, activity diagrams fulfill the function of flowcharts and more. They include the possibility for parallelism.

Activity Diagram Example

Let's look at one activity diagram example, as show in the set of figures "Activity Diagram Example".

Activity Diagram Example

Branch points are shown with a standard diamond, the branch conditions written on the side, one of them "else." Corresponding code is shown in the figure number 1.



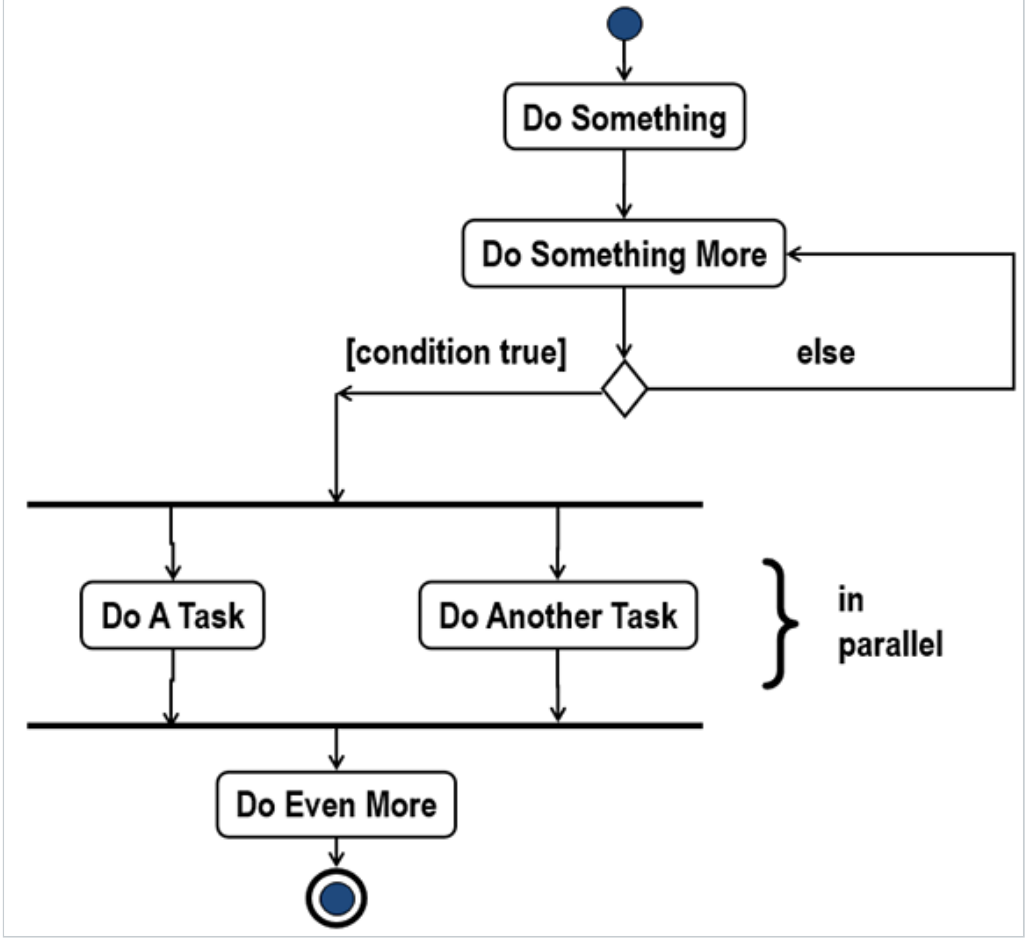
1 2

Click the above number 1 and number 2 to see the details of activity diagram example.

Activity Chart Notation

Parallelism is shown using a pair of horizontal lines. In the example (as shown in the figure "Activity Chart Notation", the *Do Even More* functions is begun only when both of *Do a Task* and *Do Another Task* have completed.

Activity Chart Notation



Test Yourself as You Learn.

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer first, and then click "Show Answer" to compare yours to the suggested answer.

Test Yourself 1.13

Consider a mobile app, AutoFind that takes as input some text, such as "Home soon." Its output is a prioritized list of apps on the phone that probably deal best with the input text such as (1) SMS (2) E-mail (3) MS Word. Rearrange the following into the most meaningful paragraph (repeats allowed).

and
a good package organization for AutoFind

active.
applications and categorization
could benefit from an activity diagram.
inactive
the class App
the class Category
the method execute() of App
the states for objects of Ap
would
would belong in applications
would belong in categorization.
would probably include

Suggested answer:

A good package organization for AutoFind would include applications and categorization.
The class App would belong in applications and the class Categorywould belong incategorization.
The states for objects of App would probably include inactive and active.
The method execute() of Appcould benefit from an activity diagram.

Test Yourself 1.14

The mobile app takes as input some text, such as "Home soon." Its output is a prioritized list of apps that probably deal best with the input text such as (1) SMS (2) E-mail (3) MS Word. (Click each question to reveal its suggested solution.)

▶ **Give at least two different kinds of classes for an app with the requirements above *.**

App, Category

*There is no single right answer to this question but you can compare your answer with the sample we'll show you.

▶ **What 2 packages should these be in?**

applications, categorization

▶ **What 2 states would be involved for one of the classes?**

for App: Inactive, active

▶ **What method of either class could benefit from an activity diagram?**

execute() for App

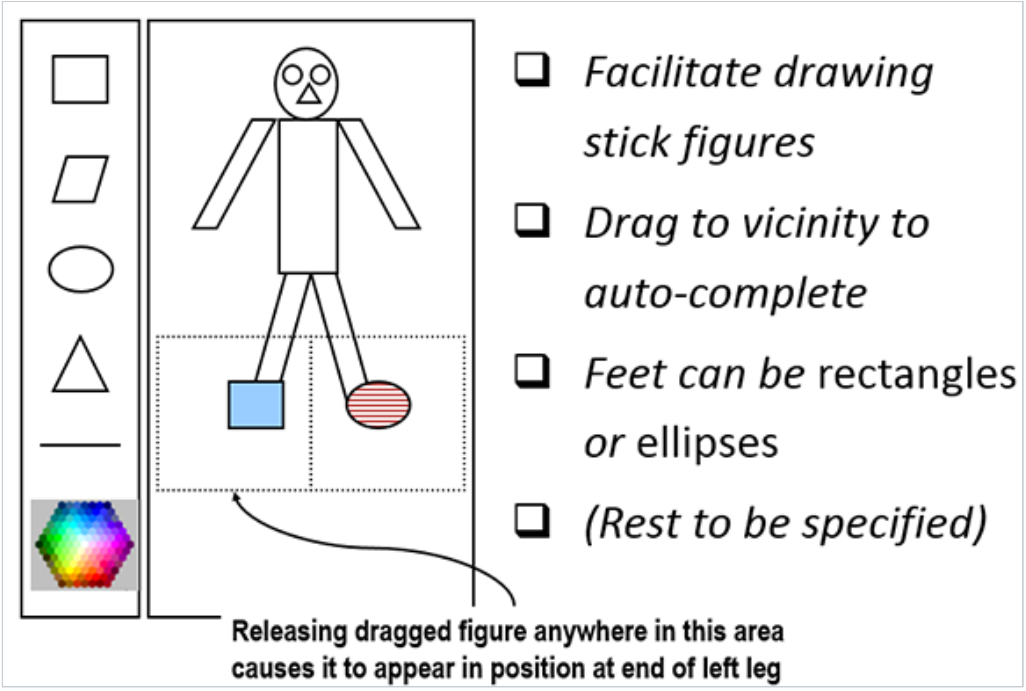
Example: Facility for Drawing Figures

For this example, we'll consider the design of a drawing application with an interface like that shown. The application is specialized to drawing stick figures. It understands something about the user's mission—e.g., the "arm-hood" (or "arm-ness") of a parallelogram, for

example. We'll focus on its ability to understand that a rectangle or ellipse dragged to the region near the end of its leg is automatically positioned appropriately as a foot when the user releases the cursor.

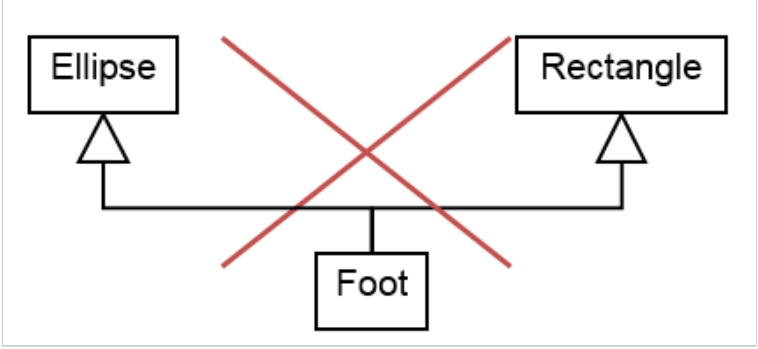
We'll consider how to design for this—the fact that a foot can only be an ellipse or rectangle.

Figure: Specifications for Figure Drawing



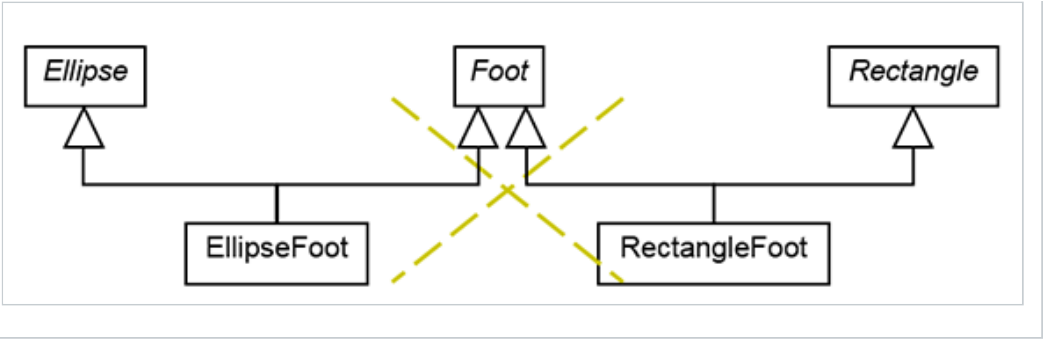
In the attempt as shown in the figure "Bad Attempt: 'A Foot is Either an Ellipse or a Rectangle'", we inherit the class Foot from both Ellipse and Rectangle. This is plain wrong because a foot is not both of these at the same time.

Figure: Bad Attempt: "A Foot is Either an Ellipse or a Rectangle"



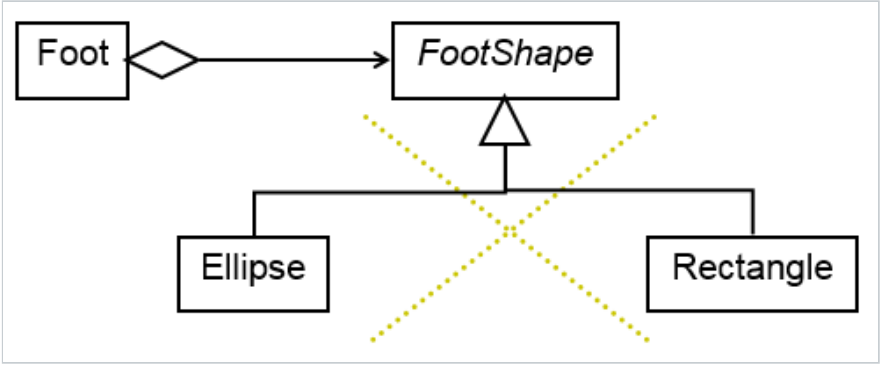
In the attempt as shown in the figure "Better Attempt: 'A Foot is Either an Ellipse or a Rectangl'", we create classes EllipseFoot and RectangleFoot to express the two options. This is not a bad option, although it's not obvious how to interface with it, i.e., how to use it.

Figure: Better Attempt: "A Foot is Either an Ellipse or a Rectangle"



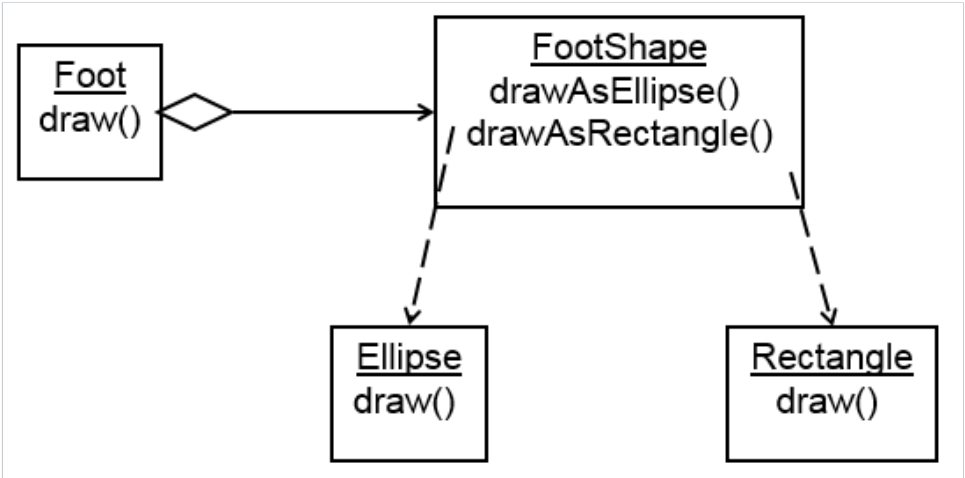
In the attempt as shown in the figure "Another Attempt: 'A Foot is Either an Ellipse or a Rectangl'", we extract the shape of a foot as a separate class. The technique of extraction like this is often a good idea. Footshape is abstract and so such objects can only be an ellipse or a rectangle. That part makes reasonable sense. The problem here is that it makes a Rectangle inherit from FootShape. It's a little strange to have "every ellipse is a foot shape." Also, this idea is not reusable.

Figure: Another Attempt: "A Foot is Either an Ellipse or a Rectangle"



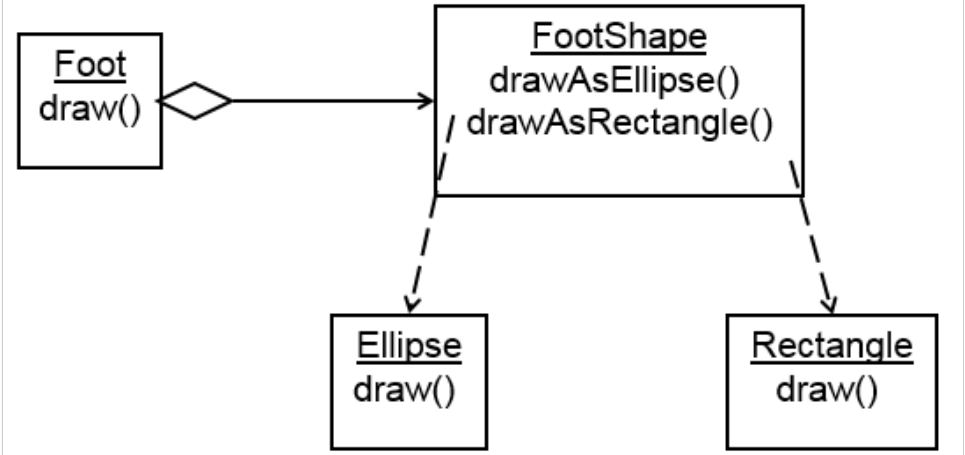
Let's look at the attempt as shown in the figure "Best Attempt So Far: 'A Foot is Either an Ellipse or a Rectangl'". This is a good solution. It retains the idea if extracting *FootShape* but it anticipates that to *draw()* a foot, the decision is made to invoke *draw()* of *Ellipse* or of *Rectangle*.

Figure: Best Attempt So Far: "A Foot is Either an Ellipse or a Rectangle"



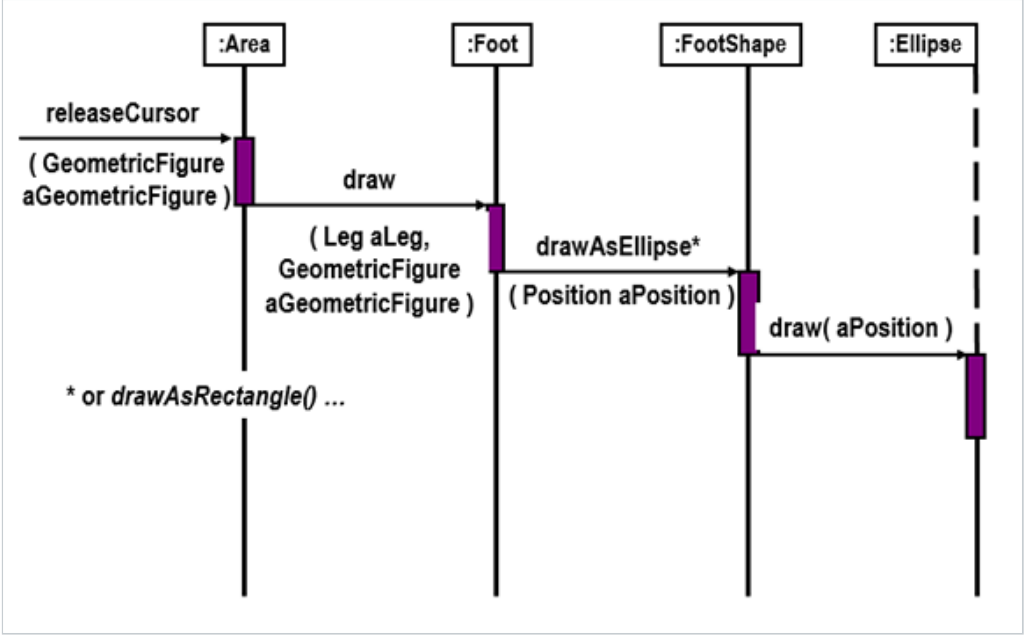
The figure "Sequence Diagram for Figure Drawing Application" shows how common use case gets executed.

Figure: Sequence Diagram for Figure Drawing Application



When you show all dependencies, the class model is quite busy. One should try to reduce them, so further improvement of the model may be possible. However, the dependencies are dispersed, and that is positive.

Figure: Class Model Showing All Dependencies



Summary

Relationships Between Classes

This session discusses association, aggregation (one-way association) and inheritance.

Association denotes a structural relationship between two classes. Because the meaning of association is vague, it tends to cause confusion. I recommend using it only as a temporary expedient if you need to postpone figuring out exactly how the classes relate (i.e. as an object reference or an aggregation).

- **Package**
  - group of related classes
- **Inheritance**
  - take on attributes and operations from other classes
  - "is-a" relationship
  - e.g. an Employee "is-a" Person
- **Association**
  - structural relationship between classes
- **Aggregation**
  - structural inclusion of object on one class by another
    - "whole-part" relationship
- **Composition**
  - Stronger form of aggregation
  - Included object is created and destroyed when including object is created and destroyed
- **Dependency**
  - one class depends on another
  - changes to depended on class affects dependent class

Appendices

Optional Appendices

The content in the appendices is optional - enjoy further exploration if you have time.

UML Example: chaining

We'll discuss an example from A.I. It involves rules, each composed of facts, taking the form:

> IF <fact 1> AND <fact 2> AND … AND <fact n> THEN <conclusion fact>. <fact 1>, <fact 2>, …, and fact n are called antecedants and <conclusion fact> the consequent.

An example is:

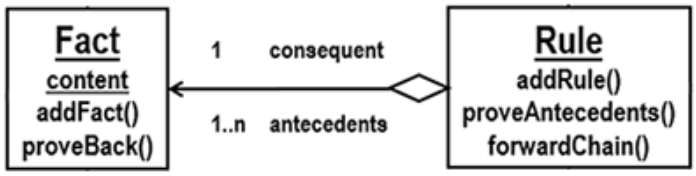
> IF animal has stripes AND animal roams in large African herds THEN animal is a zebra
>
> In forward chaining, the algorithm repeatedly applies all of the rules to the known facts (factList), and adds to them all derived facts.
>
> In backward chaining, one is given a fact that one wants to establish and proceeds recursively to derive the facts that imply it.

A Class Model for Chaining

Let's take a look at the set of figures "A Class Model for Chaining".

A Class Model for Chaining

The figure number 1 shows beginning of the simple class model for what is a non-trivial application. The labels on the aggregation refer to the names of the variables that implement the multiplicity shown. We know that rules consist of facts and that facts should be able to exist independently of rules. An example of a rule is IF business is slow (a fact) THEN a sale is advisable (a fact). The figure shows only the most apparent aggregation.

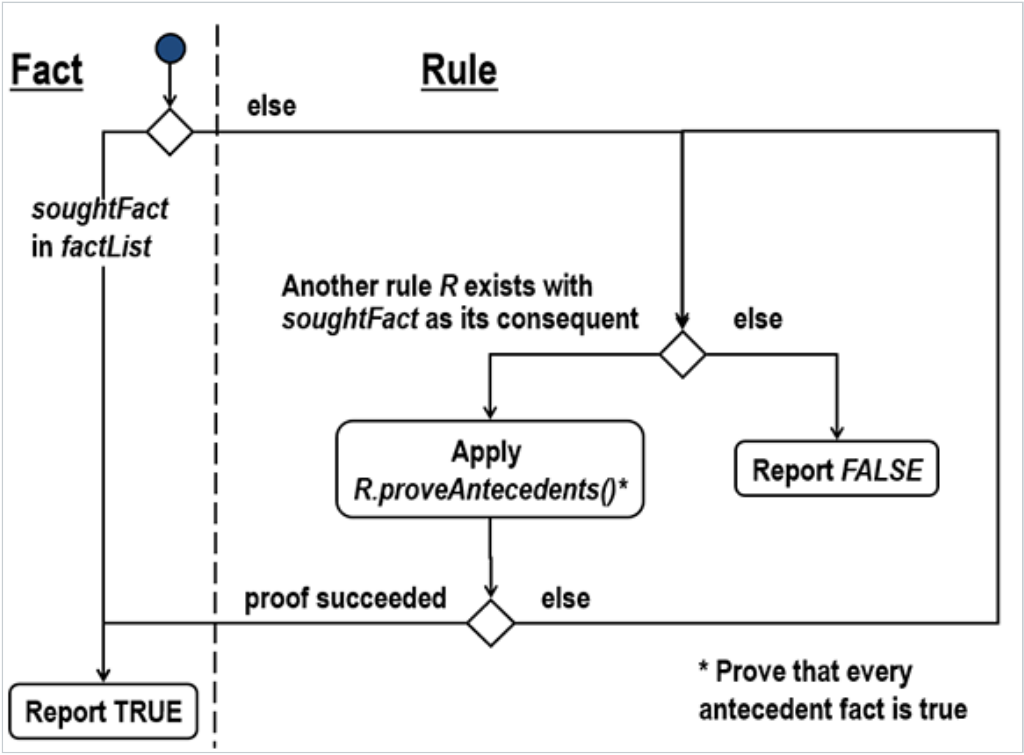


1 2 3

Click the above number 1 to number 3 to see the example: A Class Model for Chaining.

The activity diagram shows backchaining on a fact, which we'll name soughtFact. It incolves the Fact class as well as the Rule class.

Activity Diagram for *soughtFact.proveBack()*

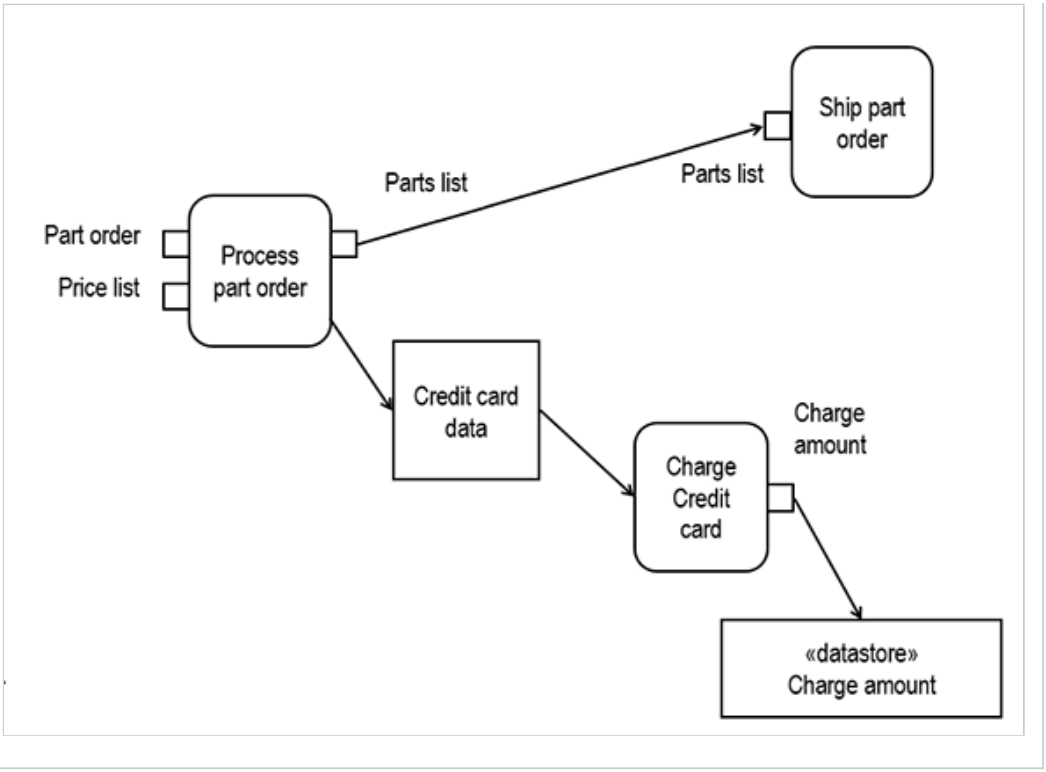


Data Flow in UML

A class model specifies a design's building blocks and the state model specifies how it reacts to external events. But neither addresses the simple question of where the data flows with the application. In fact, this is the oldest design models, far predating OO.

Data flow models show the functional processes within an application, as well as what type of data flows among them. In the example, the type of the data flowing from the process part order to the ship part order functional process is Parts list. The functions are in rounded boxes. Data stores—typically data bases—are marked as such.

Figure: Data Flow Models in UML


Ship part order
Parts list
Parts list
Part order
Price list
Process part order
Credit card data
Charge amount
Charge Credit card
«datastore»
Charge amount