8. Intro to Object-Oriented Classes

Now that you have a basic understanding of standard types, variables, and functions, and how to organize a multi-module program, we will venture forth into advanced types. Advanced types are programmer defined, and usually aggregate (sometimes called composite) types. By this I mean that YOU will be able to specify the details of a type, each instance of which is possibly made up of several standard types. For instance, a string of characters, or a university's student record (e.g. name, address, phone number). Note that we will also investigate aggregate types which contain aggregate types (e.g. a record that contains name strings that contain characters).

There are 4 statements used to define new types in C++: structure, class, typedef, and enumeration statements. Also, there is the aggregation feature called arrays. In this section we will look only at the first two.

Readings: Chapters 6 and 8 of [Savitch99]

8.1 Structures

Structures are new types that are aggregates of the pre-defined types already built into C++. They are exactly similar to Pascal RECORDs.

The nice thing about structures is that:

- they can hold several disparate types of values in one composite container (i.e. variable).
- they can be assigned from one to another as a whole.

Though we have not studied 'arrays' yet, in C and C++ (and most other languages) arrays have neither of these features. This is what makes structures, and their more interesting brother, C++ 'classes' more interesting and more easy to use.

Structures are the first step on the road of encapsulation and abstraction: putting things together that belong together, and be able to treat the composite as a whole. Often a client programmer can treat structures as a whole without needing or wanting to know their internal details. Client programmers are those who use composites types defined and programmed by other programmers. (Do you really need to know how istream cin works as long as it does its job?)

8.1.1 Example of Struct

Here is how you declare a new, *programmer-defined* structure type called AirPosition. You use the keyword **struct**, create your own name for the type, and in braces {} specify the list of elements you want in your new type.

```
struct AirPosition{
  float latitude;
  float longitude;
  long altitude;
};                  <--need semicolon!
```

Note that we did not use the Pascal keyword TYPE or the C/C++ keyword 'typedef' to define this new type. The 'struct', 'class', and 'enum' keywords are adequate to define new types of their respective natures, without needing the C++ keyword 'typedef'.

It is convention to name user defined types to begin with an upper case letter. Some companies encourage programmers to begin the names of all programmer defined types with an upper case "T". Either is fine in Cmpt 101, but you will loose marks on assignments if you mix the two styles. Adopt one convention.

Note why data attributes have their own names. If a structure contains two data attributes of the same type, we have to have a name for each so that we can later specify which one we may want to assign to or from.

In Pascal these would be called (record) 'field names'. In C++ they are called structure or class 'data members' or 'member attributes'.

For those unfamiliar with geographic position terms, latitude is a measure of the angle north or south of the equator. Let us assume the south pole has a latitude of -90 degrees. Longitude is a measure of East-West position relative to Greenwich, England. Though generally places like Vancouver are considered to have a longitude of 123 degrees West, let us assume this is -123 degrees.

Here is how you can use a structure in C++.

```
AirPosition pos1, pos2;

pos1.latitude = -123.;
pos1.longitude = 49.5;
pos1.altitude = 35000;

pos2 = pos1;

cout  << pos2.latitute << endl
      << pos2.longitude << endl
      << pos2.altitude << endl;
```

That is all there is to it. You define as many instances of AirPosition as you need. You use the so called '**dot operator'** to 'reach' into a structure to set or access the individual record fields. Assignment of a whole structure instance is possible because the three fields are stored adjacent to one and other in RAM, and the compiler just arranges to have the whole section of RAM occupied by the source record instance copied to pos2's location.

One thing that doesn't work is outputting a structure instance as a whole:

```
cout << pos2;    // <--does not work!
```

This is because the compiler doesn't know which of the many possible ways you might like the output presented:

- -123   49.5   35000
- -123, 49.5, 35000
- (-123, 49.5, 35000)
- -123 deg. longitude, 49.50000 deg. latitude, 35000 feet

There are 3 ways to handle this:

1) As shown above, specify the attributes separately in a compound output statement.
2) Write a function like print(ostream, AirPosition) which prints the second parameter to the stream specified by the first.
3) Overload operator<<() to handle AirPositions. This will be covered later in the course if time permits.

8.1.2 Composition Hierarchy

In an air traffic control system, it is often important to keep a record of a particular aircraft's path by indicating the start and ending position. So we could define a new type called:

```
struct Path{
  AirPosition startPosition;
  AirPosition endPosition;
};

Path pathVariable;

pathVariable.startPosition.altitude
                                   = 2000;
```

This illustrates that **structures can be composed of *<u>any</u>* data attribute types**, even including those of other programmer defined types! The dot operator is just used several times. Note that the dot operator works from left to right, so the last statement above can be equivalently written:

```
(pathVariable.startPosition).altitude
```

Notice that the above example has introduced a **composition hierarchy**. This is not to be confused with what we will later call a class or inheritance hierarchy. There are two kinds of hierarchy in object-oriented programming. Composition hierarchy is an equally important hierarchy that has been around for a long time, even before object-orient programming. It allows us to think of a car as body, wheels, and motor, and not be bothered with the internal details/ components of a motor. They are abstracted and may not be of interest. You can treat the major components as whole. e.g.

```
AirPosition pos3;

pos3 = pathVariable.startPosition;
```

Notice that if I was not interested in the internal details of variable startPosition (that possibly my teammate had authored), I can still grab one (as an encapsulated whole) and move it around in my job of dealing with whole aircraft paths!

8.2 Intro to Object Orientation

Generally, most computers are doing nothing most of the time. In fact, most programs are doing nothing most of the time. Generally, they are waiting for input from a person, disk, or network. When input comes, a program makes sure that the user sees the appropriate response. It does this generally by having the input event detection part of the program make function calls to functions in modules which will cause the user to see the appropriate reaction. These functions may in fact call other functions, etc.

This then is the way that you should think of designing a program. *Each external request made of a program (from menu, or network, etc.) is implemented by a sequence of function calls among a set of reactive software components*. The importance and utility of this vision of a system's architecture is under-emphasized by most authors, but is key to understanding object orientation.

The software components that export the functions that give the appropriate reaction might be:

- simple modules like a trigonometric library with sin() and cos() functions.
- Microsoft ActiveX COM components or other dynamic link library-like things (e.g. modules in a different address space to which one can make remote procedure call).

- or object instances.

The trigonometric library reacts in exactly the same way every time you call sin(PI). But more interesting and complex components may be embodied by their authors to react differently on weekends, or when called for the ninth time since program start-up. So we may regard these components as having:

- intelligence because they can decide to react differently.
- memory because they can remember things like how many times previously they have been called, or the data attributes of a structure they contain. (And can use that data to help them with a decision).

In the C++ we have studied so far, we have seen how to create types, variables, functions, and modules. What most systems are made up of is nice cohesive reactive modules. But when we have a module, we have only one. This is unlike types which from which we can create many variables. Wouldn't it be nice to be able to create multiple variables of **a type that 'has some reactive ability and intelligence'**? This would allow us to create a program for say airplanes in an airline, which operated by modeling the multiple instances of airplane 'objects' in the real world. If your airline had 50 airliners, you could define 50 airliner objects in a computer program designed to help manage the physical airliners. And because they are

reactive, you could ask (i.e. call) each one for its position, its flight hours since last oil change, or to tell it that it now has long range fuel tanks installed. Your airplane management system is then designed around these. Each command the management system receives from its user is programmed to set off the correct sequence of calls to the various modules and object instances in the program.

What's more, is that these objects model in a limited way the real airliners. Real airliners have:

- an ID,
- other stored data, and
- reactive abilities.

Humans naturally think of most things in the world as objects of one sort (i.e. class) or another. And it helps us to design computer programs, and to make less mistakes doing so, if our design technique and programming language supports this same paradigm.

In addition, it has been found that permitting encapsulation of data and reactivity in an abstract way allows us to build less brittle designs that are more easily maintained. This is because objects tend to be fundamental to a design. During future enhancement of an airline application, you might need more than 50 airliners, or airliners with more attributes or more reactive functions, but you will always need airliners components. **Objects provide a sturdy foundation around which to build your software design!** During future maintenance, you will unlikely have to tear your airliner types apart, or merge two different types of objects.

8.3 Class == Type

When we say the variable's 'type', we are referring to the type classification that the variable falls within. Can it be classed as an short int, a float, or an AirPosition. Please don't get confused by the term 'class' in object-oriented programming. It is just a programmer defined type, like a struct, but which additionally contains reactive functions.

Because it is a type, we can create many variables of that type. These are called instances of the type. So:

- object class == type
- object instance == variable

Unfortunately, many programmers use the word 'object' without clarifying whether they mean object class or object type. Usually they mean object instance, but not always. I will always try to be specific, and if I slip, please stop me and ask me to clarify which of these two case I mean.

We will now start looking at simple C++ class types. But I have put in the appendix some extra, preliminary reading on object-orientated analysis and a cool object oriented design technique. It is important to read as background, especially since I will likely require you to draw a so called Object Communications Diagram (OCD) for your next assignment.

8.4 Basic C++ Classes

A class in C++ is the same as a C struct, except that:

- in addition to member attributes, it can have member functions.
- classes allow you to reduce the scope of the members with particular keywords. In fact, the default scope is that nothing is usable outside the class. This is a safe default, but you usually have to add some public parts for applications programmers.
- actually, there are a lot of other interesting things, but those are the essential features you need to know for the moment.

Here is the AirPosition record turned into a C++ class:

```
class AirPosition{
public:
  float latitude;
  float longitude;
  long altitude;
  void drop(int dropHeight){
    altitude = altitude - dropHeight;
  }
};                  <--need semicolon!
```

Notice that this class has, in addition to member attributes, also a member function! You use a class very similarly to a C struct.


```
AirPosition pos1, pos2;

pos1.latitude = -123.;
pos1.longitude = 49.5;
pos1.altitude = 35000;

pos2 = pos1;

cout  << pos2.latitute << endl
      << pos2.longitude << endl
      << pos2.altitude << endl;

pos2.drop(1000);

cout  << pos2.latitute << endl
      << pos2.longitude << endl
      << pos2.altitude << endl;

pos1.drop(2000);
```

Note that to specify which AirPosition variable (i.e. object instance) that is to be dropped in each call, you didn’t need to send in the variable (i.e. object instance) as a parameter. Instead, you regard each variable as having its own copy of the member function, and each instance’s member function knows to drop the altitude of its own data attribute.

In actual fact, there is not multiple copies of the function as that would be terribly wasteful. Instead the C++ compiler secretly passes the object instance by reference into the function as a hidden zeroth parameter (i.e. before the first parameters). On rare occasions you may need to refer to the passed in instance, so it is give the formal parameter name of ‘this’. As a result, I could have equivalently written the body of the drop function as:

```
void drop(int dropHeight){
    this.altitude =
          this.altitude - dropHeight;
}
```

Read this on your own: Sometimes you will find your own member functions hard to read (!) because they have so many names in them, some of which are from a long formal parameter list, and some of which are member attributes but unfortunately have names similar to the formal parameter names. Thus while reading a particularly long complicated member function you will get confused. If you add the keyword ‘this’ followed by a dot to all the member attributes in a member function, it will become much clearer which names are member attributes and which are formal parameters. This is rarely done as it clutters up the code, but is a worthwhile thing to do if you are a novice or are writing particularly complicated member functions. If you have done this, don’t bother removing them later as they will likely be helpful to all readers of your code.

8.5 Member Scope

First, generally a class creates a new name scope. The attribute ‘altitude’ will not conflict with a similarly named module variable which is defined in a module which uses that class. This is because the module programmer will always have to ‘qualify’ a reference to the altitude member attribute with the instance name. e.g.

```
pos1.altitude = 35000;
```

This is true for functions too.

Note that a class author needn’t bother with this qualification, as he or she is programming within the scope {} of the class.

Second, recall that you will have to have a split personality in this course. As a class author, you may not want other programmers who will use your class definition to mess with certain internal details of your class, lest you later need to change them. Then you would have to find all the programmers who had used your class in their program, and make them find and change their code to comply with your changes.

To restrict them from messing with your code you can declare certain members off limits. e.g.

```
class AirPosition{
private:
  float latitude;
  float longitude;
  long altitude;
public:
  void drop(int dropHeight){
    altitude = altitude - dropHeight;
  }
};
```

Now application programmers cannot do the following

```
AirPosition pos1, pos2;

pos1.latitude = -123.;
pos1.longitude = 49.5;
pos1.altitude = 35000;

cout  << pos2.latitute << endl
```

They cannot set private member attributes, access private member attributes (which is what the cout statement needs to do). In addition, if there are any member functions that are declared in the private section, they too are unusable to an application programmer too. They might be there as local helper function for the class author. Note these private members are visible (unfortunately) in the class declaration, but they are *not usable* outside the class (not even in the same module as the class declaration!).

Note that the following are still usable by application programmers:

```
pos2 = pos1;
pos2.drop(1000);
```

The first is usable because of encapsulation; you can manipulate the class instances as a whole as long as you don’t directly set/get/call their private members. The second is usable because you can of course still use public members (be they functions or attributes)!

8.6 Initializing Member Attributes

Since the private members are not accessible, you might ask how they get values.

If they were public, you could as shown in your text do:

```
AirPosition pos5 = {-123.,49.,28000};
```

But this doesn't work if the attributes are private.
You could write and export a public member function like:

```
class AirPosition{
private:
  float latitude;
  float longitude;
  long altitude;
public:
  void initialize( float latParam,
                   float longParam,
                   long altParam){
    latitude = latParam;
    longitude = longParam;
    altitude = altParam;
  }

  void drop(int dropHeight){
    altitude = altitude - dropHeight;
  }
};
```

A client programmer could then use:

```
Airposition pos6;
pos6.initialize(123.,49.,28000);
```


She can do this because initialize is a public member function, and even public member functions have access to the class's own private variables!

Note that an initialization function, or a constructor (as discussed next) can be used to initialize non-private member functions. Such functions can also do useful work other than just setting variables.

8.6.1 Intro To Constructors

Another way to initialize member attributes is to use a ‘constructor’. There are many different kinds of constructors and we will look at them later. But let’s look at a simple one.

A constructor is kind of like a function but you can’t call it. It gets triggered by C++ when you create a new variable of the object type. Two examples of when this happens is when you use statements like:

```
AirPosition pos8(123., 49., 28000);
```

or

```
AirPosition pos8 = AirPosition(123.,
                               49., 28000);
```

The latter constructs an anonymously named instance with the appropriate attribute values, assigns it to the existing instance pos3, and then the anonymous one is destroyed. This is wasteful so the former is preferred. You might want to use the latter only when assigning to a existing instance as in:

```
pos3 = AirPosition(123., 49., 28000);
```


The interesting things about adding a constructor to a class are that:

- their function name is the same as the class name,
- there can be several all with the same name (as long as they have different lengths or types in their parameter lists),
- they must be declared to return nothing, not even ‘void’!

See the constructor I have added in the class below:

```
class AirPosition{
private:
  float latitude;
  float longitude;
  long altitude;
public:
  AirPosition(float latParam,
              float longParam,
              long altParam){
    latitude = latParam;
    longitude = longParam;
    altitude = altParam;
  }

  void initialize( float latParam,
                   float longParam,
                   long altParam){
    latitude = latParam;
    longitude = longParam;
    altitude = altParam;
  }

  void drop(int dropHeight){
    altitude = altitude - dropHeight;
  }
};
```

You can see that this class definition is getting very long. I will show you in future how to create a declaration that just contains member function prototypes rather than the whole function body's definitions. The definitions are put outside the declaration's braces {}, possibly even in a different file.

8.7 Appendix - Object Orientation

This rest of this set of lectures is an appendix containing multiple subsections of an introduction to object orientation, to object-oriented analysis (OOA), and one very nice technique for object-oriented design (OOD).

You can read this on your own.

8.7.1 Software Engineering Phases

Most projects have several phases. Software projects normally have:

- An analysis phase to gather and record the requirements,
- A design phase to plan the architecture and implementation strategies to be used, and
- An implementation phase where code is written.
- A quality assurance aspect. Final quality of the product is assured by actions taken throughout the project. e.g.
  - requirements, design, and code reviews,
  - unit and system testing, and
  - appropriate configuration management.

Approximately 15% of projects fail or are cancelled, usually because of failure to do some these aspects of the project properly.

8.7.2 What Is Object-Orientation?

Often there are specialists who work on each aspect of a large project. Object orientation means something different to each of them:

- To business system analysts it means determining and focusing on the business entities (e.g. sales item, customer, invoice, etc.) about which information must be processed or recorded. This pre-dates object-oriented languages.
- To a software designer, it is the architectural view that a system satisfies each external command by the set of actions resulting from the trace of calls or messages sent among various reactive software objects to implement that request.
- To a programmer, it usually means programming language syntax that allows the programmer to easily:
  - view data as having reactive abilities, and
  - re-use code via inheritance hierarchies, and
  - have both type flexibility and ease of maintenance via polymorphism.

8.8 Object Modeling

8.8.1 Introduction to Modeling in General

A model is a representation of a actual thing. To a child, a model is something created which is a 'smaller' but adequate likeness of the real thing. To a car dealer, a model is a bunch of cars which are near identical (cf. object 'class'). In systems analysis, a model captures the essential nature of something by indicating the essential details that need to be stored about things of that 'class', or by illustrating the flow of stuff required through a system, or by specifying the sequential ordering (e.g. making paper in a pulp mill, getting a university degree) within a process, etc.

Definition: A model is an alternate representation with an 'adequate likeness' of the real thing.

Some of the alternate representations we in systems design may use for the actual things are:

- a diagram or picture
- a form or computer record
- a process description, data flow diagram, or finite state machine

The purpose of creating a model is to represent <u>only the essential characteristics</u> of the thing so that:

- we may understand and clearly document the nature of the thing,
- we may store the essence of the thing for later retrieval,
- we may communicate the nature of the thing to someone else,
- they can think and/or reason about the correctness of the model without:
    - being distracted by the complexities of the complete real thing (i.e. abstraction)
    - having to travel to where the real thing is located
    - having to see the function of a real thing while it is operating very fast

- we needn't waste space storing useless information about the thing,
- we may write a program to implement a system which allows humans to better administrate the processes in which the thing participates.

8.8.2 Entities vs. Objects

The data that a system needs to store is mainly computer records of the instances of various record types in the application domain (e.g. orders, customers). Traditionally in information systems analysis, these things were called <u>**entities**</u>. Each entity class has a record/structure type with a different layout of attribute fields. Order instances have order ID number, part ID designator, and quantity of order fields. Customer records have name, address, and phone number record fields.

More recently, is has instead become popular to call domain entities <u>**objects**</u>. The term 'objects' has an <u>additional implied meaning</u> that the model of the object we are documenting contains data <u>**plus reactive abilities**</u> (i.e. plus 'functions', or 'operations', 'behavior', 'ability to control things', 'intelligence', 'liveliness'(e.g. can be sent messages or 'activated')).

In fact, this idea is carried even further by OO languages. **Rather than procedures having data parameters, instead object data is regarded as having operations/procedures that can be triggered by a message.** In fact, individual ***instance*** records (not just ADT modules) are regarded as having procedures.

e.g. Instead of (in C):

```
struct CustomerType custRecord;
printRec(custRecord, theFastPrinter);
```

You do this (in C++):

```
CustomerType custInstance;
custInstance.print(theFastPrinter);
```

Notice this is not like C, nor like Modula-2 where you would have done ModuleName.print(). The symbolic name to the left of the dot is a **variable name** (i.e. instance), *not* a module or class/type name. The procedure now appears to be a field of the instance, as if the instance ‘has/owns’ its procedures!

8.8.3 Object Data Analysis

In object data analysis, we try to determine an organized way of diagramming and storing information about the various *relevant* object types involved in the application domain. To a new analyst, sometimes it is not immediately apparent what kinds of data might need to be modeled. Examples of the object classes needed to be modeled within an application might be:

- a physical object (e.g. person, aircraft, robot, printer).
- an incident or transaction that needs to be recorded either for immediate use, for transmission to someone else, or for a historical log (e.g. order, purchase, sale, boarding an airplane, graduation, marriage, phone call). Note a purchase is from the purchaser’s application’s point of view, while a sale is from the seller’s. Interactions between two other objects sometimes fall in this category.
- a role (e.g. student, client, customer, manager, spouse).
- an intangible concept (e.g. bank account, time delay, date, sound recording),
- a place (e.g. parking space, warehouse #3, the 13th floor heat control),
- a relationship (e.g. customer’s sales representative, a flight’s captain),
- a structure - e.g. the list of an airplane’s component part numbers (body, wings, engines, tail), possibly even a hierarchy.
- an organization or organizational unit (e.g. university, department, corporation, submarine crew, sports team).
- a displayable field (e.g. string, icon, image) or printed report, or an I/O signal

- Specifications or procedures- e.g. organic compound or recipe.

8.8.4 Object Attributes and Attribute Values

We use the terms ‘object class’ to mean group of instances of things which have the same set of attribute names (e.g. car’s each have a licence number, color, and weight), but which have different values for each of those characteristics (this is what makes the instances of the same class different from each other).

It is common for a class of entity instances to be modeled as a table of fixed length records:

STUDENT TABLE

student-id	student-name	student-address	student-phone	high-school
93010-1234	Smith, Bill	123 Second St.	420-1234	Mt. Douglas
92010-4321	Jones, Jane	234 Third St.	123-4567	Burnaby
91111-1056	Able, Jim	345 Fourth Rd.	822-9876	John Oliver

This concept is in keeping with the view that a student file is a list of fixed length records.

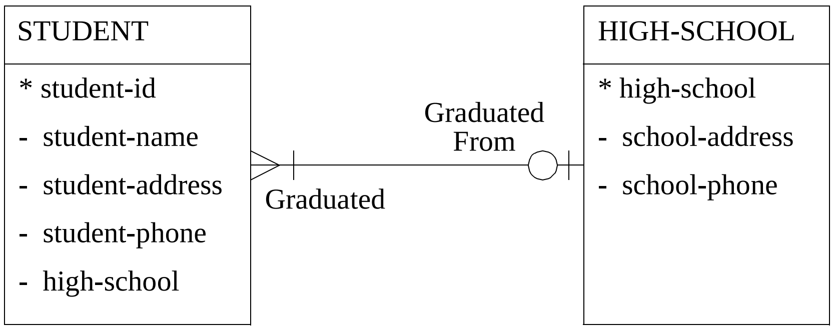
Each column represents an **attribute** of the type ‘student’ (i.e. a field of a student record). The legal set of values that an attribute may take on is called the **domain** of the attribute. Examples are date = (1..31), and day= (Sunday..Saturday).

Each row represents a particular **instance** of a student. Often the rows are sorted in order by a particular column or columns. That column(s) is called the **primary key**.

8.9 Object Relationship Diagrams

8.9.1 Object Icons

Let's examine an example of an Object Relationship Diagram (ORD) carefully. The one below shows two objects.

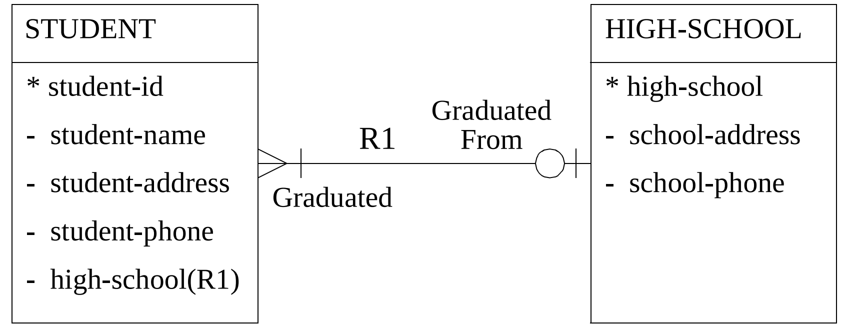


In is not clear whether they are object instances (since there titles are singular) or entity classes (since only their attribute names and not attribute values are shown). Normally in ORDs it is not really important that you differentiate between whether the boxes are classes or instances. You will probably find it ***best to think of them as generic instances*** (not having had attribute values assigned yet). i.e. they are an object storage/record layout plan.

Note that instead of having the attributes listed horizontally, as in the column titles of a table, we have the attributes listed vertically. This is widely done, though there is no reason for this except it makes the entity icons have a smaller maximum dimension. Also, note that the attribute(s) on which the records are sorted are called the **primary key** of the entity, and are labelled with a '*'.

8.9.2 Relationships

Object-Relationship Diagrams (ORDs) contain both entity classes and the ***relationships*** between them. An example of a relationship is that between a student and a high school.



Fundamentally, ***relationships are illustrations of links between entities***. These links are simply (but importantly) the referential routes that could be traversed by the application code to find other related data. Note that the high school attribute in the student class is a **foreign key** which provides the information needed to traverse R1. A foreign key is a value- or pointer-based reference to particular related instance (e.g. particular high school). Value-based foreign keys refer to the primary key of the other related (i.e. foreign) object.

ORDs provide a **<u>map</u>** showing **<u>all possible</u>** 'routes' over which the application can **navigate around the data**. For instance, given a student object, how does the application code find out the phone number of the high school she went to? Answer: Look in the High School attribute of that student to find out which high school, then find that high school record in the high school database, then look at the school-phone attribute in that record.

8.10 System Behavior

Recent methodologies suggests that you start analysis by determining an application's data model first. Even for non-database projects, this identifies early the application domain objects which will most likely **form core software elements of the eventual implementation**. In particular, the names of the important objects, their attributes, and their relationships are researched. Once this is done, we are in a better position to plan the implementation of the *behavior* of the system.

Previously, programs were regarded as a main module and subprograms which implementing an application's functionality. The newer, more object-oriented view is that a system's behavior is simply made up of the ***sum of the behaviors of the object classes and instances in the system***. The objects collaborate together during execution to get each user command done.

You can see why we had to identify the core object classes first, **as it is they what we now propose to embody with a behavioral nature**. But before we start writing code for the system's objects, we have to decide what behavior each will contribute to the whole. The next question then, is what behavior does each object class and instance need to export to the system, in order to that it satisfy it's behavioral responsibilities to the application? In the next few sub-sections of the lectures, I plan to introduce a very beautiful mechanism to synthesize the required behavior for each object class and instance.

8.10.1 Event-based Partitioning

Modern applications are event-driven in nature. Think of your personal computer; it idles for billions of instructions waiting for an event like a mouse click or a clock tick.

With this view, we will design the system by looking at how each external command or scenario-starting event is handled by the system. By looking at each external command/event one at a

time, we can reduce the scope of what we have to think about at any point in the design process to handleable proportions. When writing a requirements specification for a system, it is not uncommon to first list or diagram all the sources of external commands/events that the application must interact with (e.g. keyboard, mouse, clock, network, printer, etc.). Then in more detail, you should name/list each kind of event/command that the application program is to handle from each source.

8.10.2 External Design (User Manual)

Before beginning architectural design, it is not uncommon to write a draft user manual to firm up the behavior expected of the system for each user command. This sounds weird to some people who feel the manual is written after the coding is done. But those who finish Cmpt 275 realize that:

- you can't write the code until everyone on the team knows what the program is supposed to 'look like and behave like'!
- Often this look and behavior must be approved by someone else, so rather than spending months first writing a program that is not what the customer wants, you instead spend a week writing a draft version of the user manual for customer pre-approval.

8.10.3 Use Case Scenarios

An individual command may have several steps that should be documented in the draft manual. An example sequence might be clicking a menu command, entering several pieces of data in a dialog box, then clicking OK, the application checking and saving the entered data (often different pieces in different objects), then finally telling the user that the command is done and waiting for the user to click OK again. This is called a **use case scenario**.

Later during architectural design, we must plan what part of each step of a use case scenario will be handled by which different object.

We could thus define:

- ‘scenario appearance design’ to be deciding how a use case would appear to a user (i.e. write the user manual), and
- ‘scenario call trace design’ (or ‘scenario implementation design’) to be deciding the internal software architecture for a use case.

8.11 Object-Oriented Architectural Design

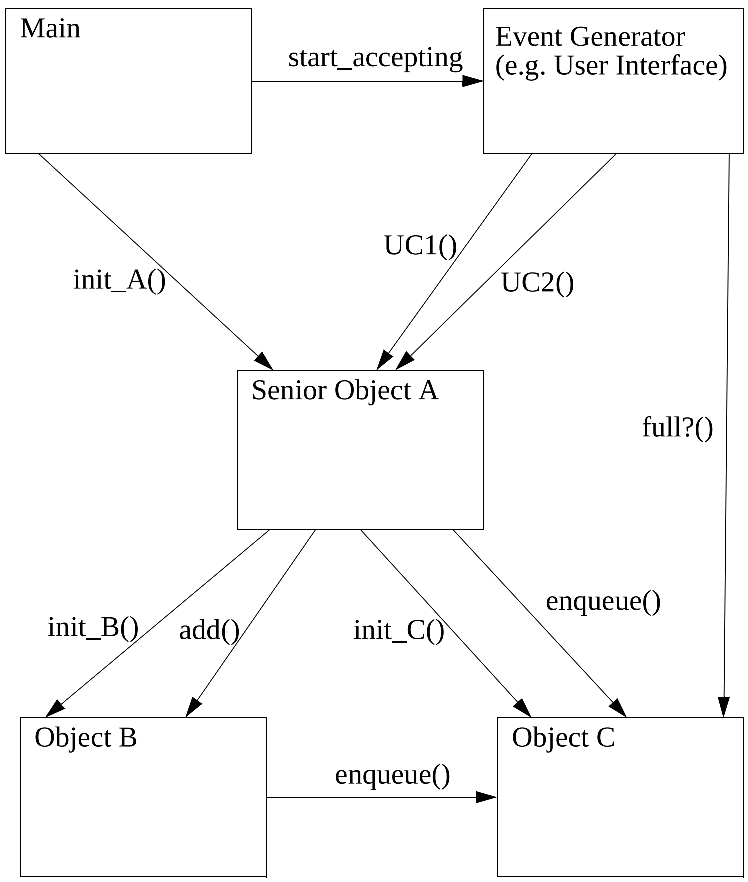
Though there are many aspects to architectural design, we will concentrate here on the design of internal call traces for the scenarios. [Rumbaugh96] states “designing the message flows is the main activity of the design phase of development”.

8.11.1 Object Communication Diagrams (OCD)

It has been common for many years to sketch a diagram indicating which procedures, or more recently which modules, *call/communicate*/*interact* with which others. This provides an interaction context which provides further understanding and documentation of the purpose, responsibilities, and dependencies of a module (often one module depends on services provided by another via exported procedures).

Very recently, we have started to diagram *object* (rather than module) interactions, and thus named such diagrams Object Communication Diagrams (OCDs) or Object Interaction Diagrams. Typically, each object class in your ORD which is reactive should be put in your OCD (note: some structures which are simply data records are not reactive and needn’t show in the OCD). Also, you may consider modules which are not objects (e.g. the main program or other utility modules) to be reactive objects. *The primary consideration here is that we identify* ***islands*** *of reactive ability/behavior/intelligence/data/control*. These islands (i.e. components), working together, implement the behavior of system.

Note that such a diagram is not to show ‘relationships’, but instead *interactions.* Two objects which have no data relationship could potentially send messages (i.e. call) each other. So an OCD is a somewhat orthogonal view of the objects in a system, and provides a kind of 2nd dimension to their definition.



Though this looks like a call structure chart, in fact the rectangles are to be regarded as components (i.e. modules or object classes) which export several differently named functions!

The main concept here is to regard and diagram the system as a collection of interacting **reactive objects**. The arrows show messages (e.g. procedure calls) from one object to another. Receiving objects should be programmed to react appropriately to each message which they receive.

8.11.2 Scenario Call Trace Design

In order to determine each reactive component's responsibilities and the operations it must export, we will examine how *each* module participates in *each* use case scenario. In order to reduce the complexity of this design step, we do this **one scenario at a time**.

In the movie industry, planning for a film segment to be shot is often done on a 'story board'. The sketches on this board provide anticipated camera shots (angles, scenery, costumes) at various moments through the progression of the scene. In essence, the user manual provides sketches of what the application will look like and do, at various points through each scenario. It is a story board. Scenario call trace design will also be done using a kind of story board. A visual plan and textural explanation of which procedure calls will be made (and why) between which objects at each point during the execution of the scenario.

Note: We could also call this scenario message trace design, because in the Smalltalk OO language, function calls are termed 'sending a message' to another object. Yet other names could be scenario implementation design, scenario event trace design, or scenario internal interaction design.

External events will be the primary driver in our design process. More specifically, a ***scenario-starting*** *external event* is a special kind of external event which initiates a *sequence* of interactions between the user and the application which carries out a use case scenario as described by the use manual. In menu-driven applications, menu selection events start most use case scenarios. The activation of a menu command results in the application receiving a message from MS-Windows. The user interface component of the application which handles these messages subsequently makes procedure calls to other application objects appropriate for the command, and these objects may in turn call other objects or modules.

If the menu command starts a long dialog with the user to enter a number of pieces of data (e.g. customer name, address, phone number) one after the other, the calls may solicit *other external* events associated with that scenario. These latter events are termed 'solicited' as the application subsequently solicits *specific* further input from the user as is needed to complete the command. The application responds to each solicited event in the appropriate way for that step of the scenario (e.g. read the data, do something with it, prompt for the next entry).

8.12 Synthesizing Object Requirements

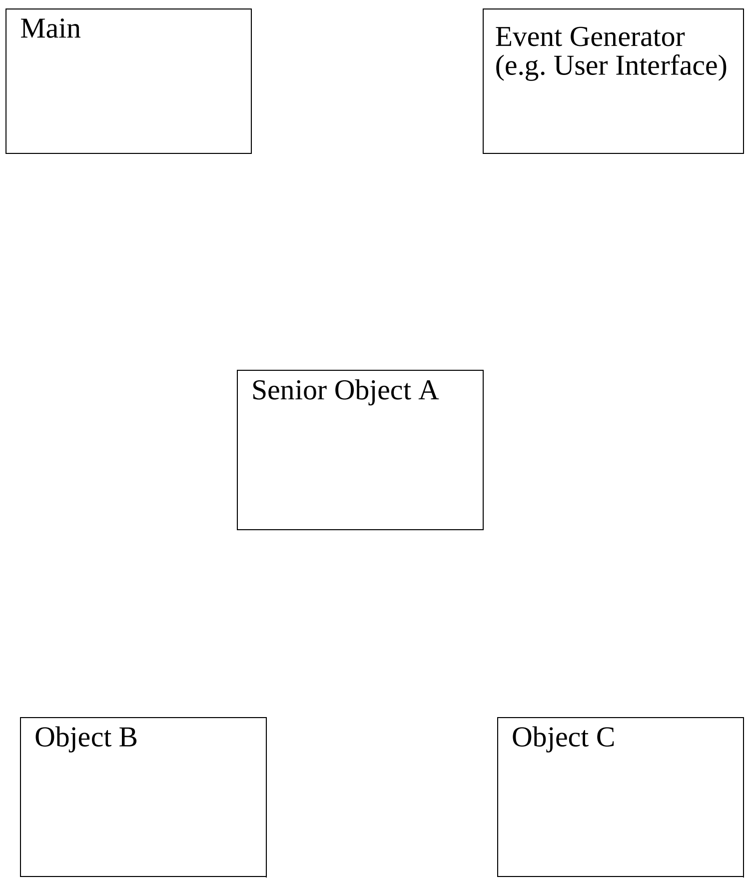
This subsection looks at a beautiful, step-by-step process by which the requirements for individual reactive components can be obtained from the overall system requirements (as embodied in the use cases).

8.12.1 Step 1 - Generate As Scenario-Starting Event List

From the user manual, generate a list of all scenario-starting external events that are required to be handled by the application. There could be dozens or hundreds in a big system.

8.12.2 Step 2 - Blank Master OCD

An Object Communication Diagram is a diagram which shows the objects from the ORD in a diagram without the relationships, and shows additional reactive components such as main, UI, and control modules. Generally, the objects are not placed in the same position on the diagram page as they were in the ORD (where they were arranged to make the relationships most tidy). Instead, place the objects in a hierarchical manner radiating away from the principle external event source (typically the user interface).

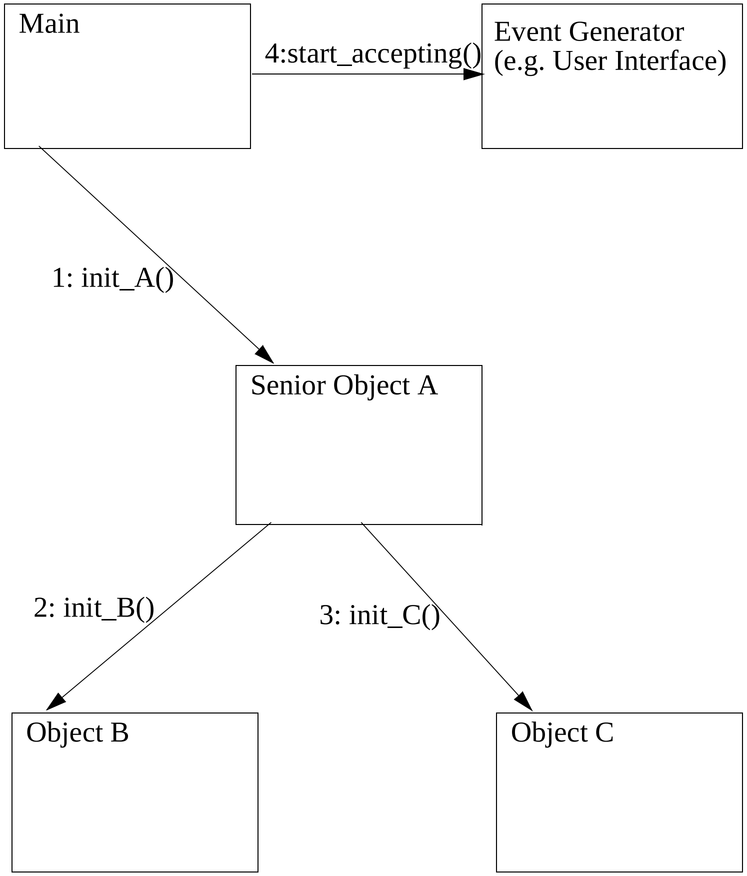


8.12.3 Step 3 - Make an Internal Call Trace for Each Scenario

Make many copies of the blank OCD diagram, one for each scenario-starting external event. For each scenario-starting event, design a trace for the anticipated calls needed to implement the proper response to that external event. (Some of the design issues which impact the choice between different trace options are discussed later). Document the trace on a single, blank OCD page. (By confining ourselves to designing one scenario's implementation at a time, we need not be distracted by arrows involved in other scenarios).

- The first scenario you should consider is the 'program start' event. This scenario should be designed to have the main module send a tree of internal initialization events (i.e. calls) to the key objects telling them to initialize (open their files, set stack to empty, etc.). The principle of low coupling dictates that the main module should not know the name of all the objects/ modules in the system, but only those directly below it. Those mid-level objects in turn send initialization messages to their subordinate objects. Any of these calls might create a number of default RAM objects as necessary for the initial functioning of the program. Once the system is initialized, the main tells the external event source components (e.g. the user interface) that they can start accepting external user events.

Start-up Implementation Call Trace



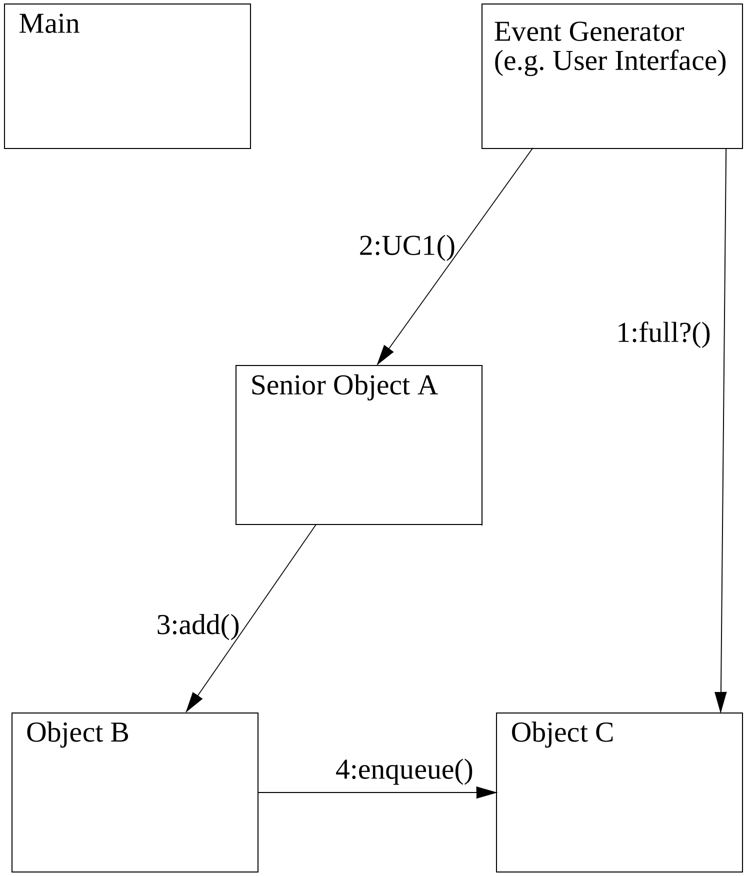
Label each message/call with a number indicating it's sequence in the execution of that scenario, and with the name of the procedure being called.

- On another diagram, for the first **external** scenario-starting event on your list, draw the trace of calls/messages that will be sent from the external interface object receiving the starting event to the principle reactive objects required to implement the response to that event. This will, in turn, sometimes cause an intermediate control/handler object to send one or more internal messages on to one or more other objects. Give each internal message a sequence number and a name which indicates what procedure is being called (or what the purpose of the message is).

  Each time you do this, you must think of **all** the internal object interactions that could take place in handling a particular external event. For instance, to register a student in a course offering, you must first check whether the course offering exists before adding a record to the association object called student-registration.

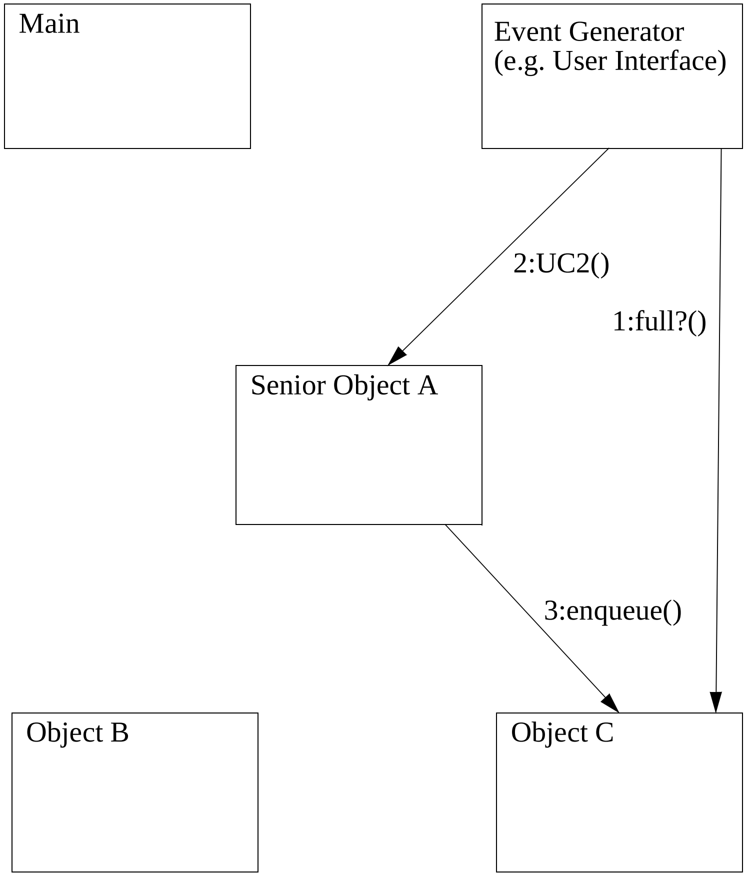
  For each diagram, it is usually necessary to document in either a paragraph, list of steps, or pseudo-code, a textural description of how the scenario is planned to be implemented. e.g. "check course exists and has space, then add student to course offering, and update available remaining course space". This provides reviewers and subsequent implementation programmers with a more understandable idea of how the scenario is to unfold.

User Command #1 Implementation Call Trace



- On a yet another diagram (see next page), do the same for the second user scenario-starting event on your list.

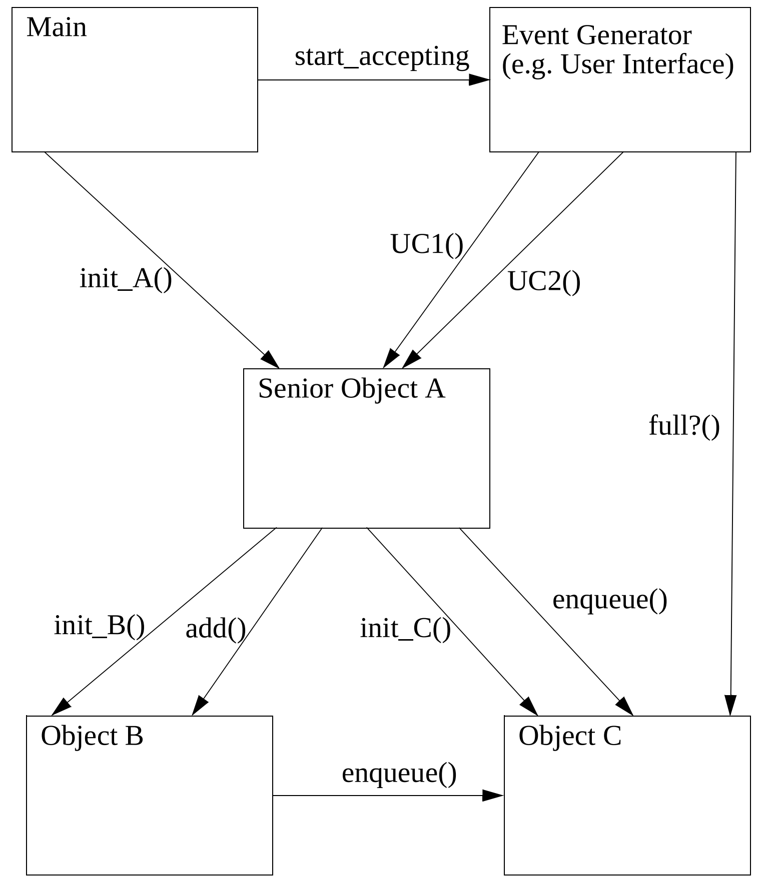
User Command #2 Implementation Call Trace



- On a last diagram, show which module(s) can initiate program shutdown, and the trace/tree of calls to the reactive components which need to be informed of the upcoming shutdown. Such components, upon being notified, shut files, flush buffers, reset the video display mode (e.g. from MS-Windows graphic mode back to DOS text mode, etc.), and delete themselves as appropriate, before the main program ends. (I have not drawn this trace to keep the resulting OCM simple).

8.12.4 Step 4 - Take the Union of All Traces

The result is the complete Object Communication Diagram:



Notice in particular how two different scenarios both had calls to the full() procedure of class Object_C. The **(first) union** operation has merged these two into one arrow in the overall OCD. All sequence numbers should be removed from the labelled arrows since with so many different scenarios shown, they no longer make sense.

The result is a fantastic diagram!

- The (first) union synthesizes an OCD from which the requirements spec for an object class can be determined. Obviously, the class must export a function for each different type of arrow entering it. e.g.
  - The UI must export start_accepting().
  - Object A must export init(), UC1(), and UC2().
  - Object B must export init_B() and add().
  - Object C must provide/export empty(), enqueue(), init_C(), and enqueue().
- Notice that the above list seems to imply Object_C should export enqueue() twice. By taking a **second union**, you can merge the two different enqueue() calls to Object_C (which are not merged by the first union because they are from different callers), into one item in the list of procedures that Object_C must export. Basically you must regard the list of exported procedures as a true 'set' where duplicates are not allowed.
- In addition, you get a requirements spec for each object's **responsibilities** to call/notify other modules/objects. An object will do some internal processing when called, and then likely some interaction with other objects. *The diagram shows all the other objects that a particular object is planned to get info or processing from, or must notify in order to fulfill its responsibilities. e.g. Senior Object_A has the responsibility to notify those below it that they should initialize themselves.*

8.13 Alternative Control Architectures

The above strategy is very powerful as it constructively **synthesizes** the requirements for individual modules and object classes from an application's external requirements. This makes it an extremely appropriate technique to bridge the so called 'design gap' that exists between the end of analysis and the beginning of writing code for individual modules.

Please note that there are many alternatives in constructing the trace of a scenario. This is where the real design decisions are made. (The diagramming with a CASE tool and the double union are basically just documenting the design decisions and constructively gathering object specifications from the traces). Trace alternatives will be discussed in the next section of the course.

As with all design, there are usually several ***alternate*** ways to design a sequence of internal call events that will carry out a particular scenario. For example, when the UI receives an 'exit program' command from the user, should it send messages to all the objects telling them to shut down? Or should it call a procedure in the main module which should then tell the objects to shut down? **'Design' is choosing between workable implementation alternatives to pick the one that is most elegant, most easy to maintain, uses the least memory, and/or is best performing**.

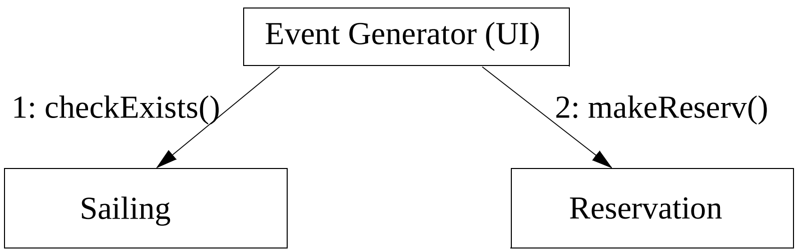
Let us consider a simple reservation system. Generally a reservation instance is for a particular flight, sailing, or video rental instance, etc. A reservation typically is related to a particular, say, sailing via a foreign key. When dealing with user-entered data, we must use every effort to maintain referential integrity of the database. Thus before creating a reservation instance for a person on a sailing, we must check that that particular sailing actually exists. This scenario implementation can be designed in one of three alternative ways. These three ways will be shown in the next 3 sub-sub-sections.

8.13.1 Centralized Scenario Design

In this design, a particular reactive component which both is informed when the scenario is to be initiated, and which understands the scenario to be carried out, orchestrates the execution of the scenario.

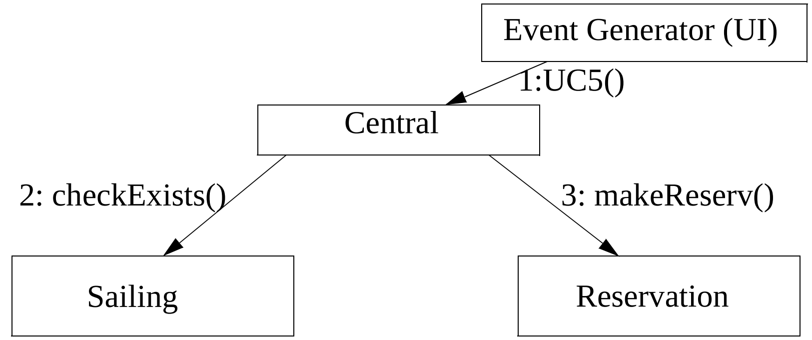
Although often not the ideal design, this component may the event generator itself (e.g. user or network interface module), in which case application scenario code (possibly unfortunately) gets added to the event generator module.

.



Alternatively, as shown below, an extra control module or object can instead be added whose only job is to orchestrate scenarios. It is not unusual for this module to export more than one function, one in fact for each scenario to be orchestrated in an application (or for a particular subset of scenarios in the application). The

event generator is programmed to simply call the correct scenario orchestration function given the event that has just happened.



Scenario Description:

```
1) Prompt user for all info;
2) If Sailing exists
3) THEN make reservation
4) ELSE re-prompt user.
```

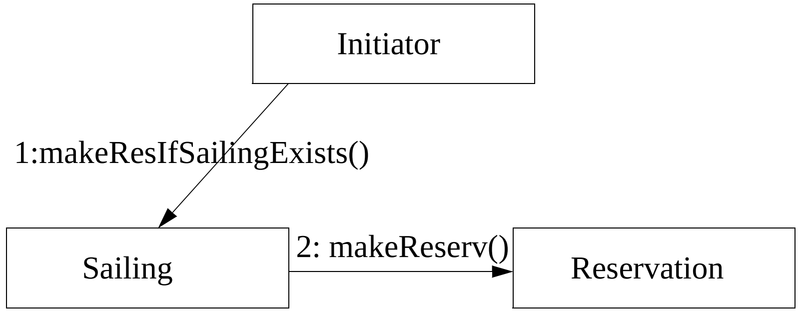
In both the above centralized schemes, the controller sends a message first to the sailing object to check that the sailing exists, *then waits for the return from that call*, then makes a call to the reservation object (supervisor/shepherd) to actually create the new reservation, the waits for that call to return. The centralized control scheme has the advantage of cohesively encapsulating in one function of one module (be it the Event Generator or a special component) the control and sequencing of internal calls needed to carry out the processing needed in the scenario. Its advantage is that if the control or sequencing of the scenario might later during maintenance need change, only one function in one module needs to be updated. Also notice that the sailing and reservation objects do not communicate with each other, and thus don't have to know about each other (this is occasionally a good design feature). On the other hand, the central object unfortunately gets coupled to all the parameter types of all the lower calls.

Notice the explanatory text or pseudo-code that can be included under a scenario trace diagram to more fully document the logic of the scenario. This pseudo-code might, for instance, indicate whether the sailing information needed from the user is read by the sailing module or by the central control module.

This pseudo-code may or may not eventually be put into any particular module. It may end up in the central module, or alternatively be spread out over several modules if either of the following designs is adopted. It is therefore not to be thought of as programming, but instead as documentation of the scenario logic from an architect's point of view, so that programmers could later implement the design properly as per the architect's specifications.

<u>8.13.2 Roundabout Route Scenario Design</u>

The name of this section is a Tront'ism and is not widely used terminology. The idea is that control is passed from the initiator (i.e. event generator) to the first module which must supply preliminary checking or data, and then that module forwards the request to the final object. The control thus travels a rather roundabout path to the terminal object. When the makeReserv() procedure is done, it returns control to the Sailing, which in turn returns from the makeResIfSailingExits() to the initiator.



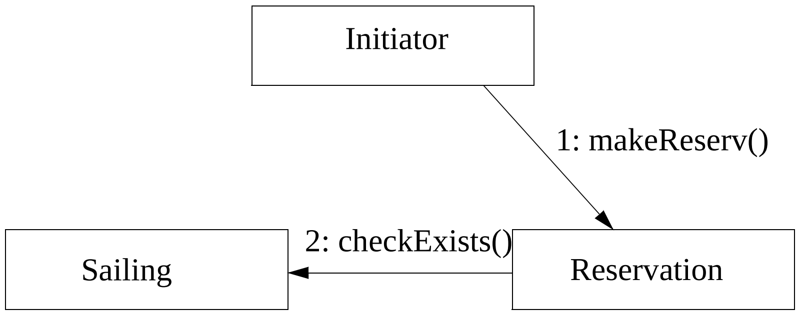
<u>Scenario Description</u>:

```
1) Ask Sailing if it exists, and if so
2) THEN have it make reservation
3) ELSE have it return an exception to
    the initiator which will then
    re-prompt the user.
```

This design strategy is particularly good if using asynchronous one-way messages, rather than procedures calls, as it requires no data to be returned to callers.

8.13.3 Principle Object-based Scenario Design

This design alternative has the initiator first informing the principle application object involved, in our case the reservation object. After that, the principle object (which may understand its creation needs best) does whatever is necessary to accomplish the request. In the example below, the reservation checks the sailing exists, waits for the reply, then if ok makes a new instance of its type, and then finally returns control to the initiator object.



Scenario Description:

```
1) Ask reservation to make an instance
2) It checks if Sailing exist.
    If so reservation makes an instance,
3) ELSE return exception to initiator.
```

Note that these diagrams do not show the procedure returns, but this design requires an OK to be returned to the reservation via a parameter/return value. Either that, or if using one way messages, a return message would have to be added to the trace.

