8. INTERACTION+INTERFACE DESIGN

Last Mod: 2024-06-11

Must have a directory where the information can be backed up.
Russell: You can say that you presume the database will back up the information
Look up into backing up data using MySQL

This section of the course will look at:

- Planning the **interaction** of the software modules and objects in the system.
  - I.e. Which object/module calls which other, when, and in what order.
- Actual function prototype parameter and return type details and strategies.

In the issues, you must add a date of creation field

Table of Contents

8.1 Introduction to Interaction

Generally, 3 ‘aspects’ of an object-oriented system need to be specified with “models”. The data flow diagram (DFD) aspect previously mentioned with a more object-oriented model using UML Object Communication Diagrams.

So, more modernly, 3 aspects are:

1. The data retained by the system, for use in constructing later outputs.
   - This data and its relationships documented in Object Relationship Diagram (ORD).
2. The sequence of calls/messages that propagate through a software system for each command or use case.
   - Documented using one of two forms of Object Communication Diagram (OCD).
3. The behavior of each generic object instance, and each class's non-instance aspects, can, if appropriate, be documented using two Finite State Machines (FSM).

These 3 models are essential, in the same way that an architect must specify, via a 3-view drawing, for the construction of an unusually-shaped building.

- But unlike 3 architectural views, where all 3 views are the same kind of thing (2D graphics), the 3 OO models are each different in nature.

8.2 Modules and Class Static Members

This instructor often used the term "module" to describe a .cpp (or .java) file generally.

.cpp files may contain no classes (like C language). But even no-class .cpp modules can have global and private data, and global and private functions.

- Just like in C language before C++.
- In C++ we use keyword **`static`** rather than **`private`** to limit a non-class function or variable to a single "compilation unit" (i.e. .cpp file).

E.g. stack.cpp written in C manner.

```
int pop();                      //global func
void push(int i);               //global func.

static int stack[100];    //module scope var
static int helperFunc(); //module scope

//Other legal examples:
static int modVariable;   //module scope var
int globalVarable;              //global var.
```

Module scope variables and function can only be access from inside the .cpp file.

- Their names are not even put in the compiled .obj file consumed by the linker.

If writing in C++, and you don't need multiple instances of your entity (e.g. stack), do NOT put the functionality in a class and create one instance of the class.

- An instance is simply not needed.

E.g. Java math class:

- You do NOT need to instantiate an object of type **`Math`** to use **`sin()`** and **`cos()`** functions.

In fact, your **`main()`** function in Java is just a static member function of your application class. It exist whether you program has instantiated any elements of your application class or not.

8.2.1 Static Class Members

In both C++ and Java, you can have member functions and data members that are labelled **`static`**.

This means they are present and usable whether there are any instances created yet, or not.

The **`static`** keyword has 5 meanings in C++:

- Make local variable keep its value rather than disappear when goes out of scope. It's still there when you come back into that scope. (original meaning)
- Make a variable outside of all classes only .cpp-module-wide in scope.
  - Prevents polluting the global namespace with to many variable names that might conflict with other programmers global variable names.
- Make a function outside of all classes only .cpp-module-wide in in scope.
  - Prevents polluting the global namespace with too many functions with same name.

- Make a class data member NOT an instance member.
- Make a class member function NOT an instance function.

E.g. Java Math class:

```
class Math{
   static double sin(double angle);
   static double cos(double angle);
}
```

Use in Java:

```
x = Math.sin(myAngle);
```

Note **Math** in above statement is *not* an instance but a class name.

If C++ had a class called Math, I would call it this way:

```
X = Math::sin(myAngle);
```

using the “scope resolution” operator.

I call these ‘class supervisor’ methods and variables, though that’s non-standard terminology.

Advantages:

- They exist and are usable whether any instances created yet or not.
- Useful when don’t need instances, and want to use more selective private/protected/public classifications than using **`static`**.
- Only way to get global-like functions and variables in Java.
  - Java **`main()`** must be static.

In fact, they can be used to create and manage a flock of instances.

In analogy to flock of sheep, I also refer to them as “shepherd” members of the class.

- The shepherd might know where the sheep are.
- The shepherd might keep a count of sheep instances.
  - This is not something that belongs in a sheep’s brain.
- The shepherd may (or may not) be the only one allowed to operate on the sheep instances.
  - Create, import, sheer, sell, destruct.

The purpose of teaching you the above is to show you are 3 parts of a .cpp file that can be shown receiving a function call in an Object Communication Diagram (OCD):

- Non-class functions.
- Shepherd functions.
- Instance functions.

You may, or may not, want to draw these as separate targets for call arrows.

8.3 System Behavior

Recent methodologies suggest starting analysis by determining an application's data model first.

- Even for non-database projects, identifies early the application domain entities/objects which will form the core software elements (i.e. reactive components).

In particular:

- names of the important objects,
- their attributes, and
- their relationships.

We're then 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 implemented an application's functionality.

- More object-oriented view is a system's behavior is 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, or handle each significant external event (e.g. network packet arrival).

You see why we identified core object classes first.

- They are what we now propose to embody with a behavioral nature.

Before we start coding, must decide what behavior each will contribute to the whole.

- What behavior does each module/class/ instances export to the system, in order that it satisfy its behavioral responsibilities to the application?

In the next sub-sections, I'll introduce a beautiful mechanism to synthesize the required behavior for each object class and instance, from the required behavior of the system.

8.3.1 Event-based Partitioning

Most modern applications are event-driven in nature.

- Consider a personal computer: It idles for billions of instructions waiting for a mouse click, a clock tick, or network packet.

So we'll design a system by looking at how each external command or scenario-starting event is handled by the system.

We'll look at each external command/event one at a time.

- This reduces the scope of what we have to think about at any point.

When writing a requirements specification for a system, it's common to 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 name/list each kind of event/command that the application program is to handle from each source.

- "Use cases"

8.3.2 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,
- clicking OK,
- application checking and saving the entered data (often different pieces in different objects),
- finally telling the user that the command is done. This is called a *use case scenario*.

We must plan what part of each step of a use case scenario will be handled by each different module/object.

We could thus define:

Must write exhaustive user manual

- 'scenario appearance design' to be deciding how the progress of 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 interactions needed to implement a use case.

[Rumbaugh96] states “designing the message flows is the main activity of the design phase of development”.

8.3.3 Object Communication Diagrams (OCD)

It's been common to sketch indicating which modules/objects, *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 from the other.

More recently, we diagram *object* (rather than module) interactions, and thus have named such diagrams Object Communication Diagrams (OCDs) or Object Interaction Diagrams.

Typically, each object class in your ORD *that is reactive* should be put in your OCD.

- Some classes that are simply dumb data records are not reactive, and needn't show in the OCD).
- You may consider modules which are not C++ objects (e.g. the main program or other utility modules) to also be key reactive components if they export procedures in header .h files.

Review this

- *The primary consideration here is that we identify* **`islands`** *of reactive ability/behavior/intelligence/data/control.*

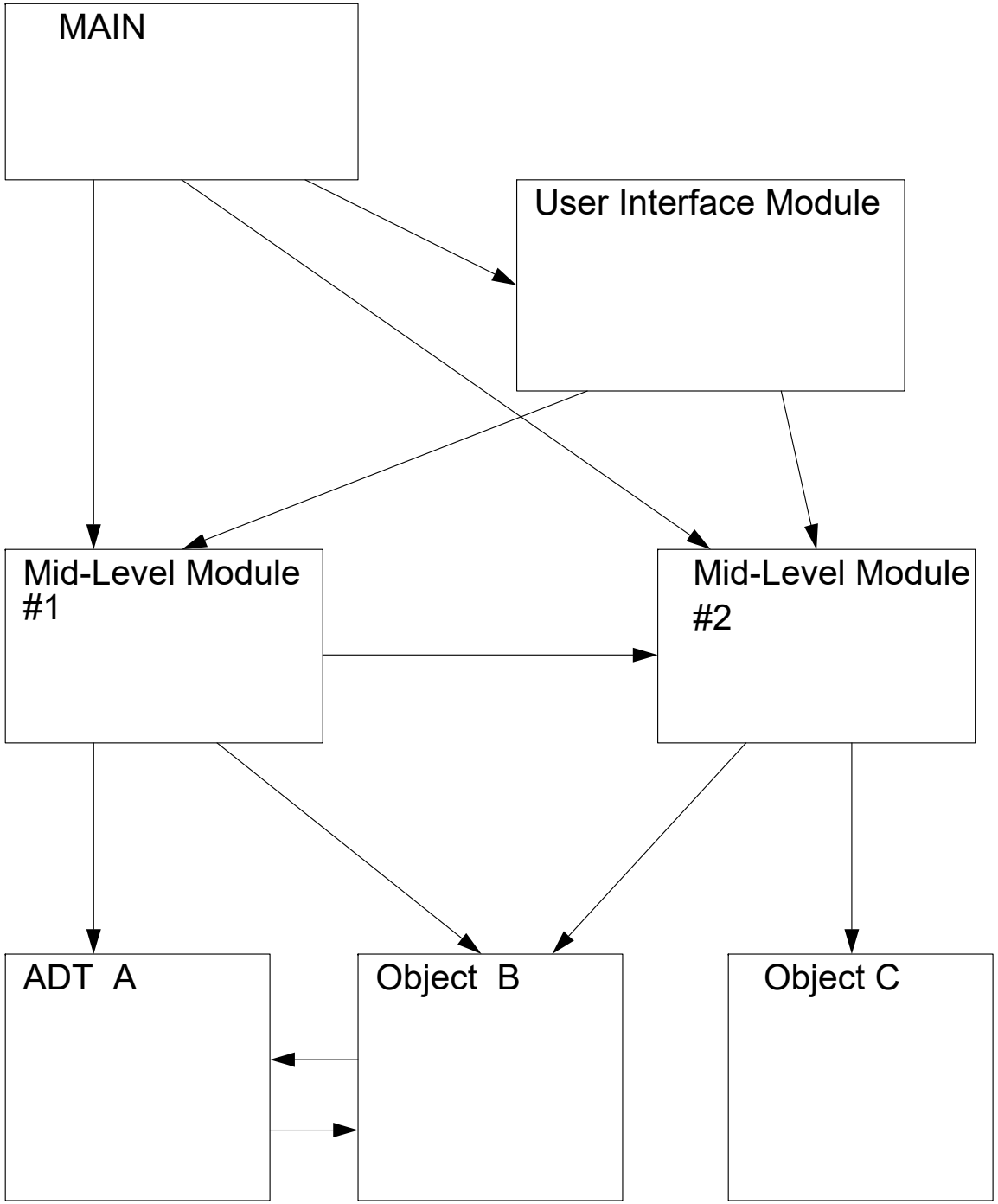
These islands, working together, implement the behavior of system.

1NF, 2NF, or many-to-many

Such a diagram is <u>not</u> to show 'relationships', but instead *interactions*.

- Two objects which have no data relationship (no foreign keys) could potentially send messages (i.e. call) each other.

An OCD is a somewhat orthogonal view of the objects in a system, and provides a 2nd dimension to their definition.


MAIN
User Interface Module
Mid-Level Module #1
Mid-Level Module #2
ADT A
Object B
Object C

The main concept here is to regard and diagram the system as a collection of interacting **reactive modules/objects**.

The arrows show messages/function calls moving from one object to another.

- Arrows can be labelled.
- Receiving objects must handle type of message/call they receive.

8.3.4 The Reactive Software Components

5 kinds of reactive software components:

- Function modules, like the main, and user interface (UI) library modules. They may have some data too.
- Application domain object instances, like particular customers and invoices.
- Application domain object class supervisors (i.e. shepherds).
- Implementation domain object instances, like queue and timer instances
- Implementation domain object class supervisors.

Implementation domain:
things that are inherent in the OS, IDE, or language

Notice first step in identifying reactive components in OO analysis and design is object data analysis.

For a small application, if add a main module and a user interface module to the application domain objects, we have an initial set of components that could make up the program!

At program start,

- the main module sends start-up messages (i.e. function calls) to the important modules telling them to initialize themselves and their subordinates.
- The main then creates any necessary transient objects.
- Finally, the main sends a kick-start message to the UI module indicating that it is now OK to start accepting user commands.

So, to begin an Object Communication Diagram (OCD):

- First, put a module on the diagram to represent the main program module.
- Then add an icon for each of the objects from your ORD.
  - Do **not** draw any of the relationship lines.

- Next, add a module for each major external interface the system will have.
  - External interfaces are sources of events that drive the system.
  - And are exits for output data and control signals.
  - Most programs need User Interface (UI) module.
  - Some include a Network module.
  - For systems which control external mechanisms (like robots in a manufacturing process, or a printer), a Process Control module is necessary.

- Finally, you may need to add a module/class for orchestrating all or the intricate use case scenarios.

Orchestration is encapsulation of the code into a descriptive class/function

  - Perhaps they export one function for each use case.

In the non-OO past, it's been suggested for small applications that the:

- main go at the top of the diagram,
- key application abstractions representing objects from your Object Relationship Diagram (ORD) go at the bottom, and
- some mid-level control modules go in the middle.

Notes:

- The main module may only initialize things, and shutting them down. (Or in a small program may orchestrate all the scenarios.)
- Low level objects/modules may just provide storage services for different data types.

- The mid-level modules are often control/orchestration modules:
  - they control and sequence operations such as getting data one record at a time from a storage object to print a report (storage objects shouldn't print!).

Ask Russell about this.
Since it is just a record, it should not know how to do anything? Should it only store the information?

  - The mid-level modules know which storage object functions are to be called in which order (possibly in a loop) to accomplish each particular user command (i.e. each use case scenario).
  - The mid-level modules also handle exceptions, such as a storage object rejecting an attempt to read a non-existent record or running out of space to write into.

If each call arrow in the diagram can be labelled with the message/function name it represents, results in **a diagram that shows every function name that every module/object has to handle**!

Therefore, a complete and properly-labelled OCD has the information on it to determine every function name that needs to be exported by, and then coded in, every module/object.

8.3.5 Scenario Call Trace Design

To determine each reactive component's responsibilities, and the operations it must export, we'll examine how

- *each* module participates in
- *each* use case scenario.

In order to reduce the complexity of this design step, we do it **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 are like a comic book.
- 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.

Internal scenario call trace design can also be done using a kind of story board.

It's a visual plan and textural explanation of which function 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.

Other names could be

- ‘scenario implementation design’,
- ‘scenario event trace design’, or
- ‘scenario internal interaction design’.

External events are the primary driver in our design process.

More specifically, a *scenario-starting external event* is a special kind of external event that 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 GUI 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 that handles these messages subsequently makes 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), 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 that command.
- The application responds to each solicited event response in the appropriate way for that step of the scenario (e.g. read the data, do something with it, prompt for the next entry).

Note: Some methodologies [Shlaer92] consider the calls from one object/module to another to be 'internal' events. Each object is then regarded as a finite state machine reacting appropriately to internal events which hit it.

8.4 Synthesizing Object Requirements

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

8.4.1 Step 1 - Generate A 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.

8.4.2 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, network interface, 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).

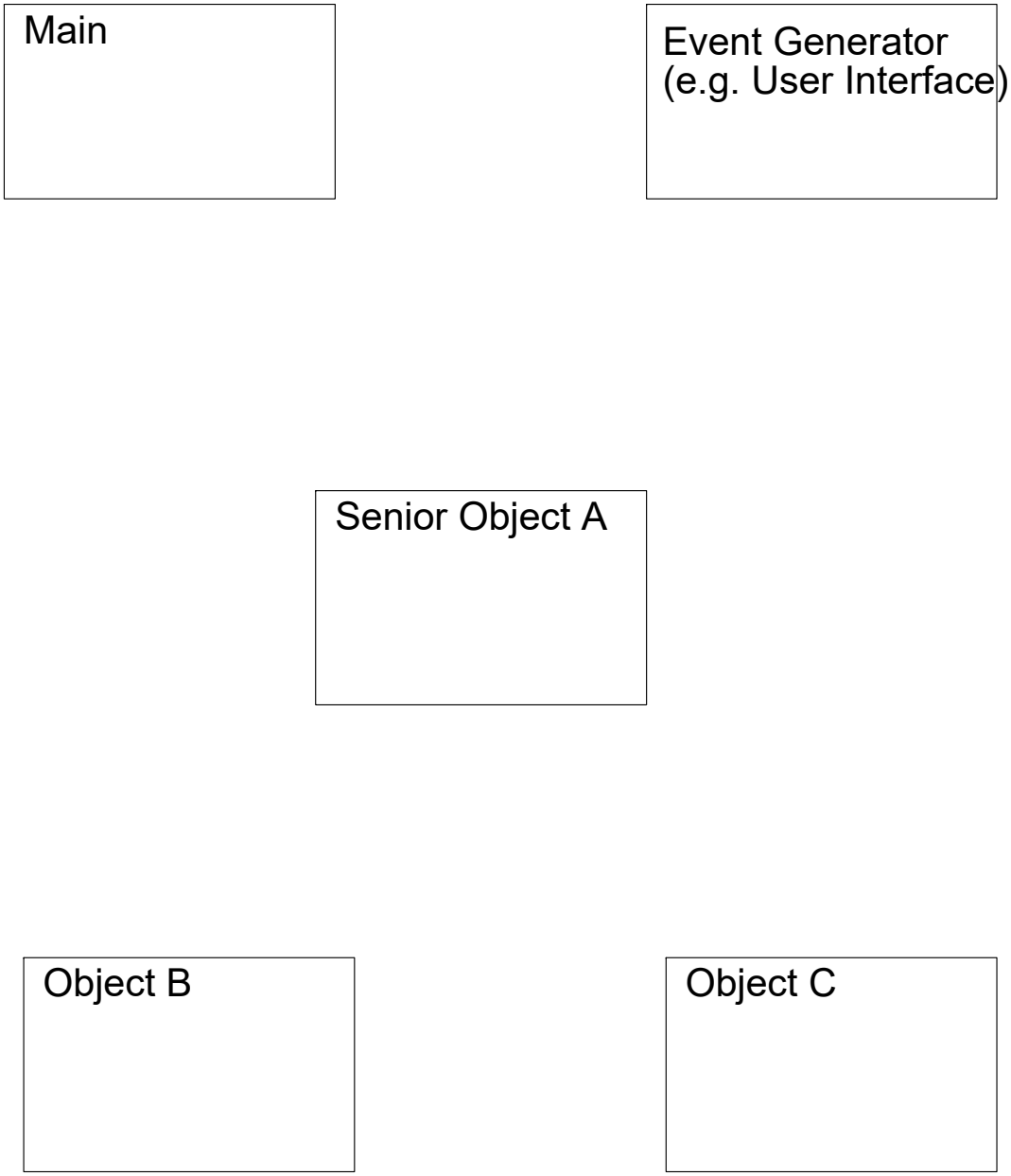
Instead, place the objects in a hierarchical manner radiating away from the principle external event sources (typically the user interface).

(Note: **UML** notation does not have a “master” OCD that we are seeking, which shows all calls from all use case scenarios.

Nonetheless, for an individual scenario, UML does have a ‘Collaboration Diagram’ that is perfect for one scenario.

Collaboration diagrams are just one of two different types of '**Interaction Diagrams**' offered by UML.

- Both types are equivalent.
- But so-called 'Sequence Diagrams' are portrayed differently.
- See UML Quick Ref Charts.pdf, in the course references.


Main
Event Generator
(e.g. User Interface)
Senior Object A
Object B
Object C

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

If using a pencil and paper, *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 as a collaboration diagram on a single, blank OCD page.

- By confining ourselves to one scenario's implementation at a time, we're not distracted by arrows involved in other scenarios.

The first scenario to consider is the ‘program start’ event.

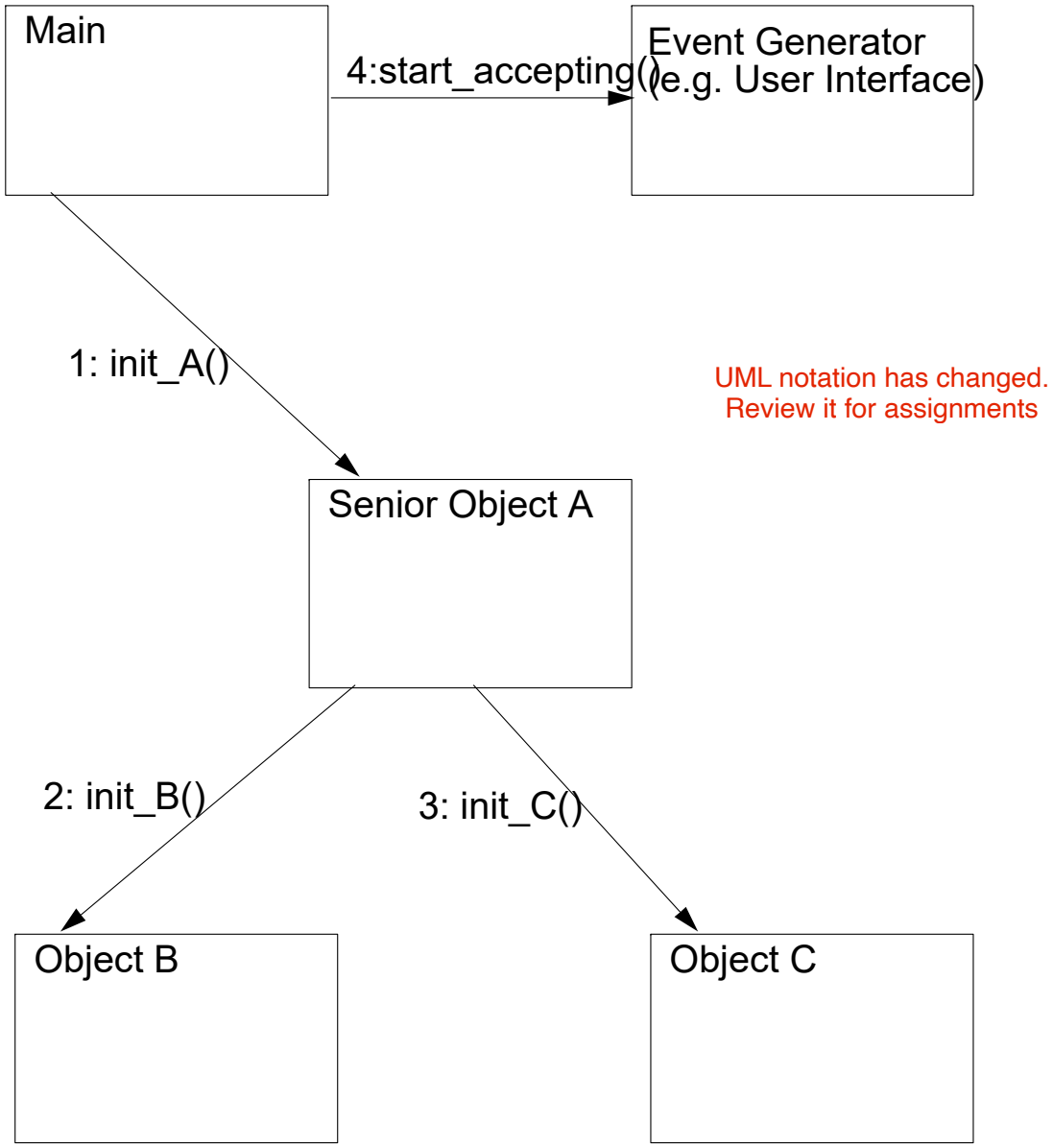
- This scenario is designed to have the main module send a tree of internal initialization calls to the key objects telling them to initialize (open their files, set stack to empty, etc.).
- The principle of low coupling dictates the main module not know the name of all the objects/modules in the system, but only those directly below it.
  - The mid-level objects in turn send initialization messages to their subordinate objects.

- Any of these calls might also create some default RAM objects necessary for the initial functioning of the program.

The objects receiving your messages/function calls might not have had their functions defined yet.

- Now is the time to give each message/function a name.

Start-up Implementation Call Trace



Once the system is initialized, the main tells the external event source components (e.g. the user and/or network interfaces) that they can start accepting external events.

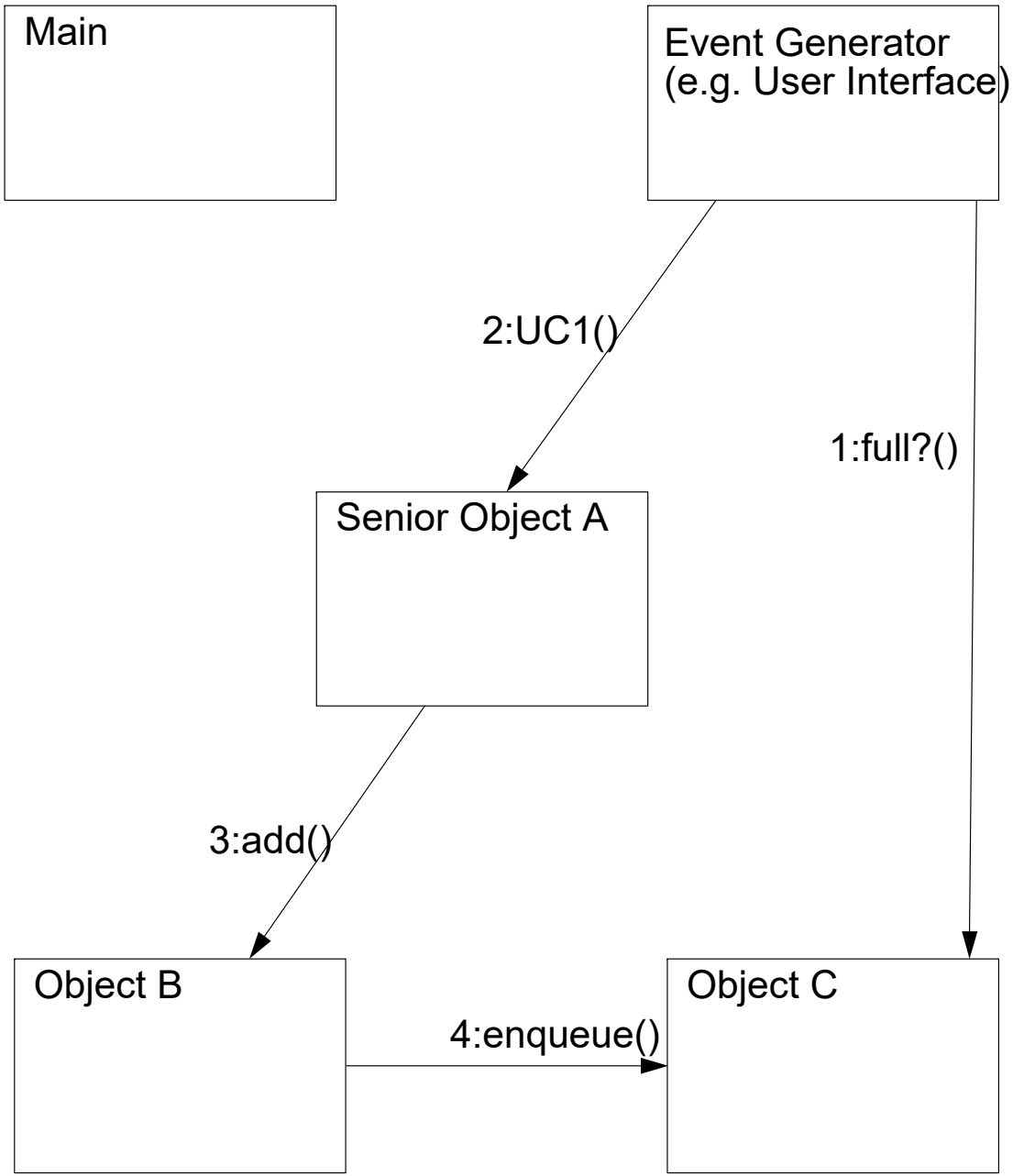
Label each message/call with a number indicating its sequence in the execution of that scenario, and with the name of the function being called.

- Need numbering in case two calls come out of one module: Which comes first?

On another diagram, for the first external scenario-starting event on your list, draw the trace of calls/messages sent from the external interface object receiving the starting event to the principle reactive objects required to implement the response to that event.

- This, in turn, sometimes causes an intermediate control/orchestrator object to send one or more internal messages on to one or more other objects.

User Command #1 Implementation Call Trace



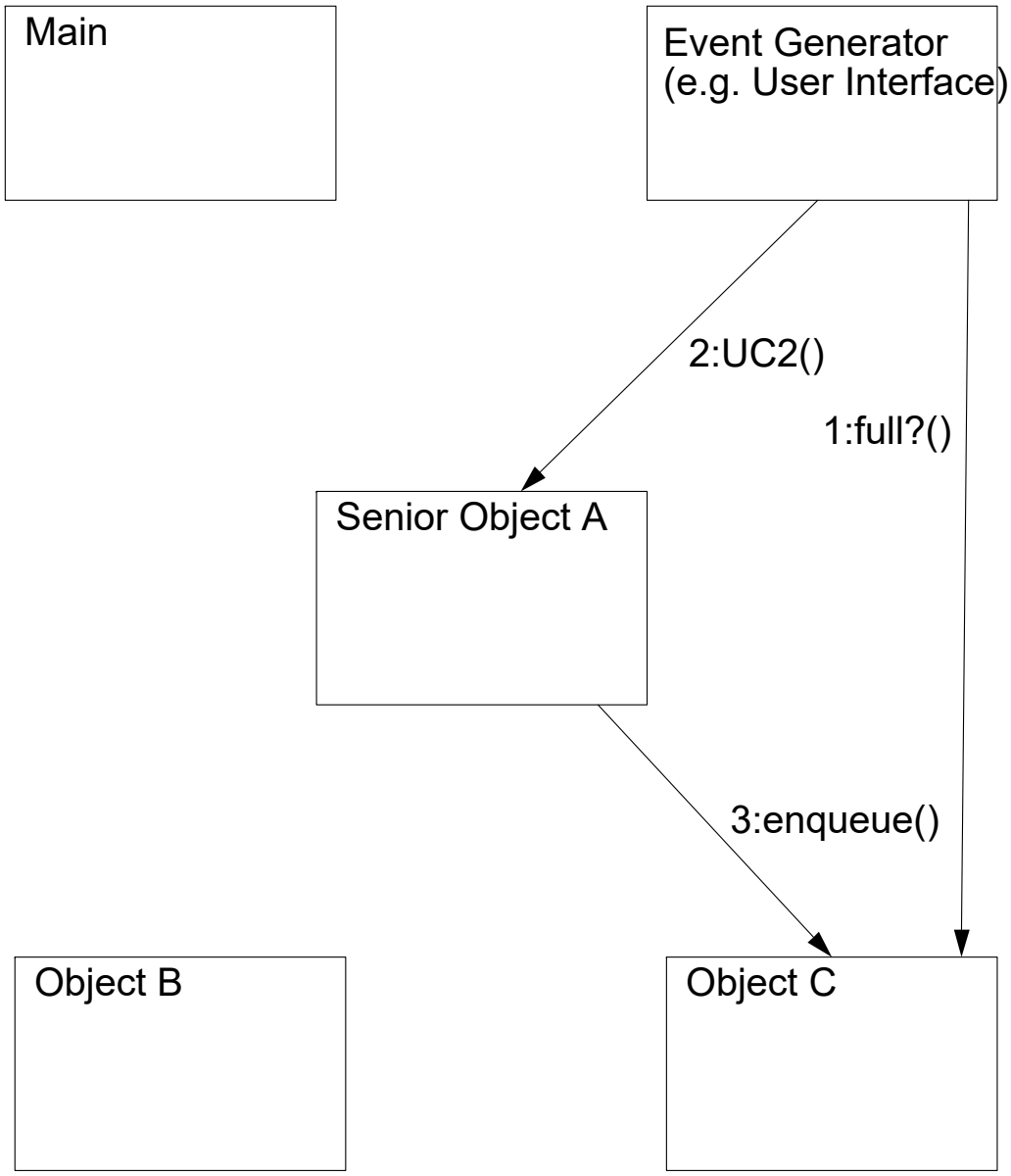
Each time, think of all the internal object interactions that *need take place* in handling a particular external event.

- E.g. To register a student in a course offering, first check whether the course offering exists, then add a record to the association (XRef) object called student-registration.

For each diagram, 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 how the scenario is to unfold.
- On 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 modules/objects 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,

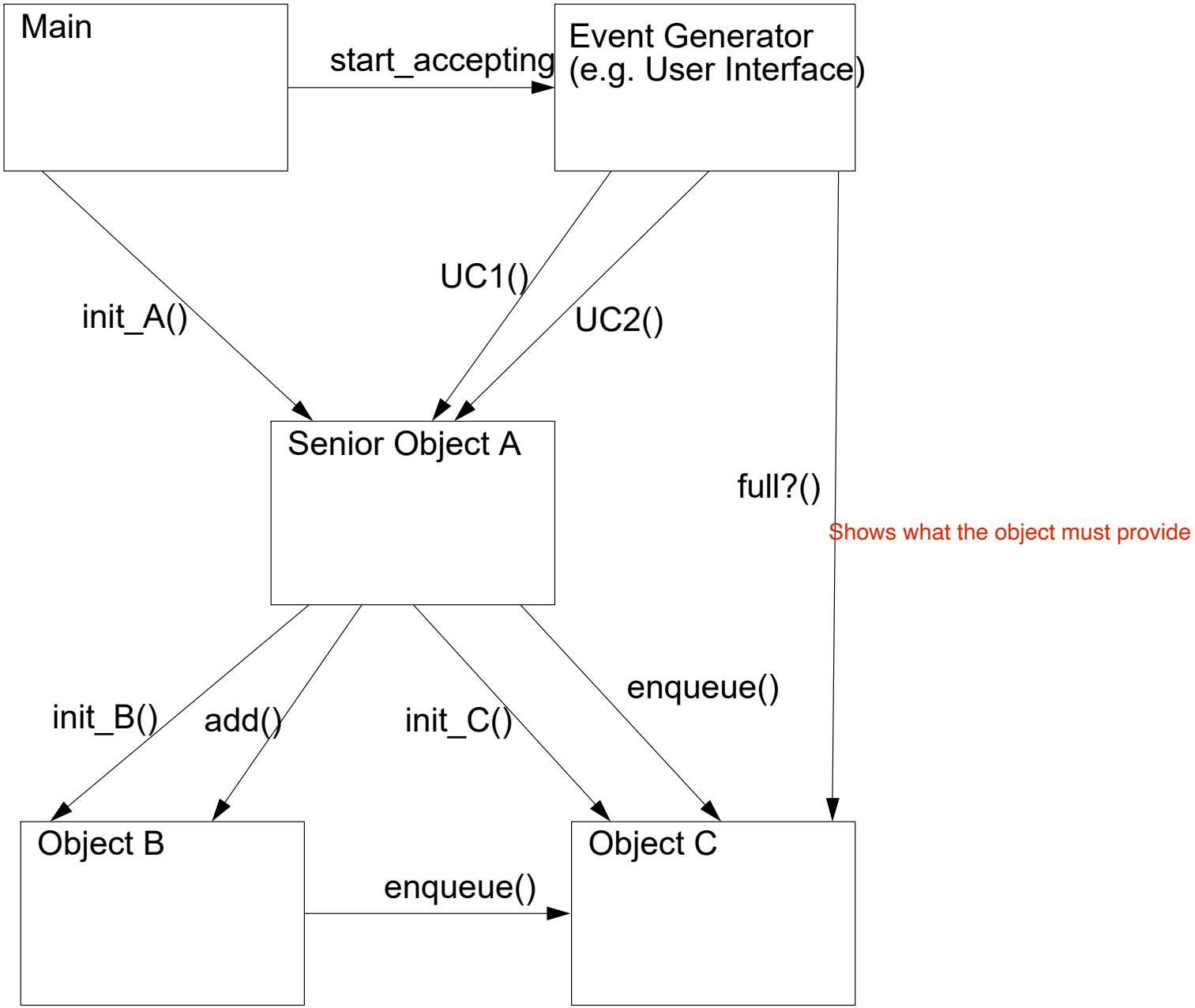
- shut files,
- flush buffers,
- empty tanks,
- reset the video display mode (e.g. from MS-Windows graphic mode back to DOS text mode, etc.), and
- delete themselves as appropriate,

before the main program ends.

(I have not drawn this trace to keep the resulting OCM simple).

8.4.4 Step 4 - Take the Union of All Traces

Take the **union** of all collaboration diagrams to get a complete Object Communication Diagram:


Main
start_accepting
Event Generator
(e.g. User Interface)
init_A()
UC1()
UC2()
Senior Object A
full?()
Shows what the object must provide
init_B()
add()
init_C()
enqueue()
Object B
enqueue()
Object C

Notice 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 of a full OCD, since with so many different scenarios shown, they no longer have relative meaning.

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_A( ), UC1( ), and UC2( ).
  - Object B must export init_B( ) and add( ).
  - Object C must provide/export empty( ), enqueue( ), init_C( ), and enqueue( ).

- 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 to fulfill its responsibilities. E.g. Senior Object_A has the responsibility to notify those below it that they should initialize themselves.

To make the double union more clear:

- The first union constructs the full OCD overlaying all the individual scenario message traces together.
  - Since it's common the same object could call a certain function in another object in two different scenarios, this first union removes this duplication.
  - This is also why the trace arrow sequence numbers must be removed, since a single arrow in an overall OCD can correspond to two or more sequence numbers from different scenarios.

We need an OCD for each menu item
We need an OCD for the startup and end sequence

- The second union is the union of the sets of calls from *each* other reactive component to a particular target object (E.g. Object C).
  - If two different other components each call the same function in object C, the second union removes the duplications *from the list of functions object C must support*. e.g.
  - Set of calls from UI to object C = {**`full()`**}
  - Set of calls from A to C = {**`init_C()`**, **`enqueue()`**}
  - Set of calls from B to C = {**`enqueue()`**}

  The second set union is:

{**`full()`**} + {**`init_C(),enqueue()`**} + {**`enqueue()`**}

= {**`full(), init_C(), enqueue()`**}

This resulting list has *fantastically* formed the set of functions that must be:

- advertized in **`object_c.h`** and
- implemented in **`object_c.cpp`**!

8.4.5 Miscellaneous Comments

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.

Several methodologies suggest designing critical scenario architectures individually, using some kind of object 'interaction' diagram (and some accompanying descriptive text).

But none that I know of suggest a significant benefit gained by graphically designing them all, then using the double union to synthesize the requirements for every object.

- I hope you appreciate the beauty of the technique.

This technique could be easily automated.

- A Computer Aided Software Engineering (CASE) tool could be written.
- It allows you to graphically enter a blank OCD (possibly using objects from your ORD).
- You then construct scenario traces into collaboration diagrams involving those objects.
- CASE tool perform the unions.
- And generates function prototypes for each object class!

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 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.

Finally, the arrows you drew represented messages or function calls.

- Data can be passed back to the caller at the end of the call.
- But not all systems support direct function calls.
  - The interaction between different applications, or between different parts of a distributed application, may only allow *one-way* messages.
  - For these systems, return data must be sent (rather than passed) back by an additional return arrow.
    - This is why I have been hesitant, or vague, about calling the arrows functions.

- In some systems they might not be functions, but either one-way operating system messages or network packets!

8.5 Scenario Trace Design

Every scenario should be designed with care before you can truly know the system architecture. We'll look at three issues to be careful about.

8.5.1 Adding and Designing Non-User Scenarios

First, realize that certain *previously un-thought of scenarios* may be needed.

Does the user not trigger the startup scenario?
Am I assuming this?

- These scenarios may not even be part of the user scenarios.
- Good examples are the start-up and shut down scenarios. There will be others.
- Those of you familiar with modern languages are aware they often provide each module or class with an constructor that's automatically executed at start up.
  - You might not think, then, that a start up scenario is necessary. You think each object or module can be written to automatically initialize itself.

- Systems can be constructed to this way.
- But what if a system, while running, needs to do a ‘warm reset’.
  - E.g. A user is tired of the situation he has got himself in, and wants to reset everything to its starting condition.
  - Such a system needs a ‘re-start’ scenario.

Also, shut down may be necessary for reasons other than just closing files.

- E.g. In a milk-processing plant, you may want to shut down the system.
  - This requires telling all tank objects/modules to drain themselves.
  - Even if the main module or user interface detects that the user wants to do a shut down, for reasons of abstraction and design information hiding, the main or UI might not even know that a tank object exists.
  - The shutdown scenario designer is aware of all the objects, and constructs a trace which will get the necessary shutdown control signal down to the tank via a trace of calls.

– This shutdown trace is constructed to weave down through the abstractions one layer at a time, since usually each object or module knows only about those immediately below it.

8.5.2 *Labelling Semantic Order*

In addition to documenting which objects interact with which other objects in a scenario, *the numbers on a call event trace document* **semantic ordering**.

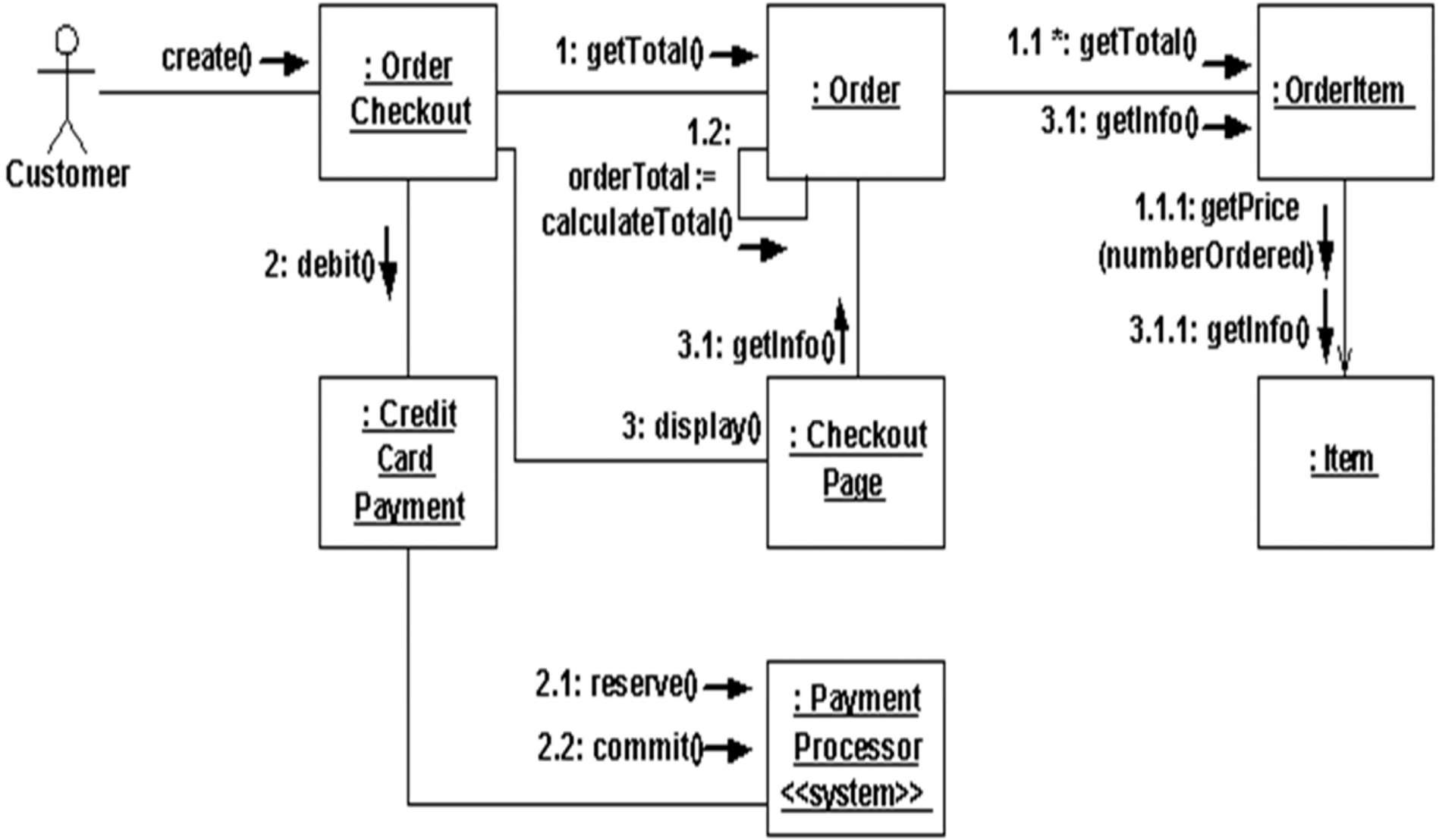
- Sometimes the tree of traces could have several alternative orderings, only one or two of which result in the correct processing of the scenario.

- If the ordering is not made clear, an implementation programmer who didn't understand the overall system needs, might write the wrong code.
  - E.g. Given an object which can be both read and written, one scenario might require that the data be written (i.e. initialized) before anything can be read from it.
  - Another scenario might need to read it first (in order to extract and preserve the data) before over-writing it.

  Numbering the individual calls in a trace documents the ordering you have decided is correct.

Arrow numbering:

- Two calls with the same precedence number indicate that they can be made in either order.
- Using numbers like 2 and 2’ mean that either one, or the other message is sent, but not both.
- Some propose using 2* to indicate repetitive calls.
- Most show meaning to decimal numbers (e.g. 2.1, 2.1.3, etc.).



Reference: agilemodeling.com/style/collaborationdiagram.htm

Each call depth increases number of decimals.

8.5.3 Alternative Control Architectures

As with all design, there's 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 one that’s:

- **most elegant,**
- **most easy to maintain,**
- **uses the least memory, and/or**
- **is best performing**.

Consider a simple reservation system.

- A reservation instance is for a particular flight, ferry sailing, or video rental instance, etc.
- A reservation 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, must check 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.

8.5.4 Centralized Scenario Design

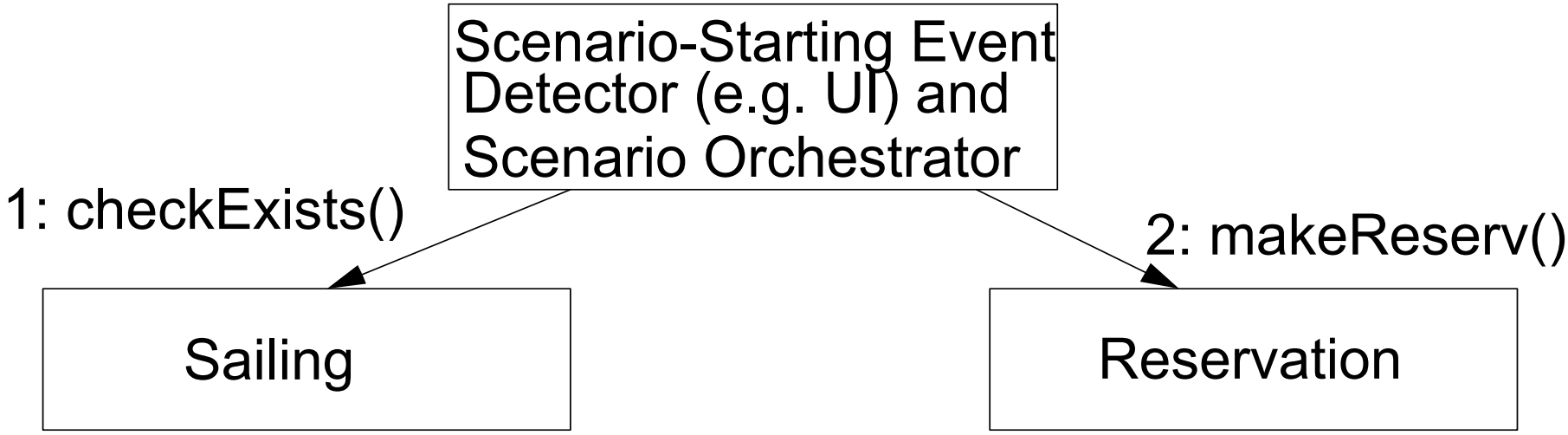
In centralized scenario design, a particular reactive component that's both:

- informed when the scenario is to be initiated, and
- understands the scenario to be carried out,

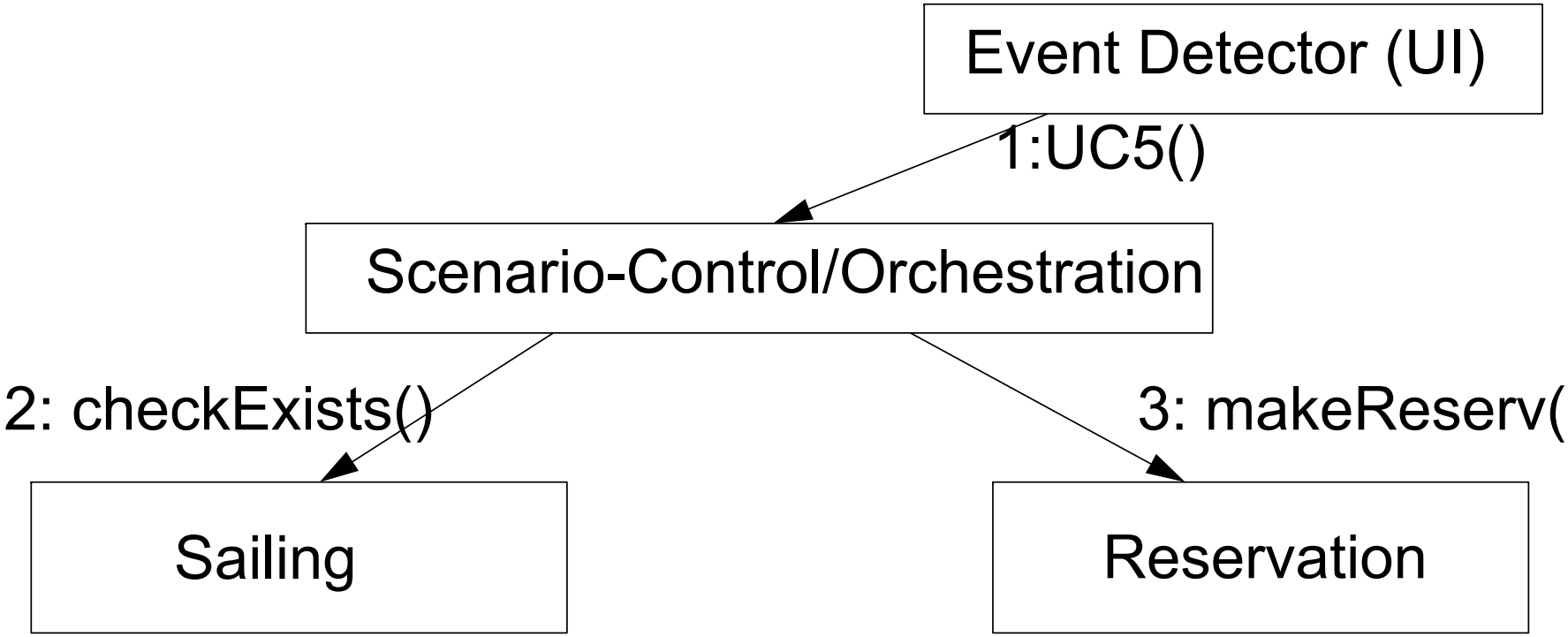
orchestrates the execution of the scenario.

Although often not the ideal design, this central component may be the event detector itself (e.g. user or network interface module).

- Though the scenario control code (possibly unfortunately) gets added to the UI or network interface module.



Alternatively, as shown below, an extra control module or object can instead be added to house a function that orchestrates a particular scenario.



Scenario Description

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

- It's not unusual for a control module to export more than one function.
  - Perhaps one for each scenario to be orchestrated in an application.
  - Or for a particular subset of scenarios in the application.
- The external event detector is programmed to simply call the correct scenario orchestration function, given the particular scenario-starting event that it just detected.

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,
- then 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 detector or a special orchestration component)

the control and sequencing of internal calls needed.

One advantage of centralization is control or sequencing might later during maintenance need change, only one function in one module needs to be updated.

Notice sailing and reservation objects don't 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 was included under a scenario trace diagram to more fully document the logic of the scenario.

- This pseudo-code might, say, indicate whether the sailing information needed from the user is read by the sailing module or by the central control module.

The pseudo-code may or may not eventually be put into any particular module.

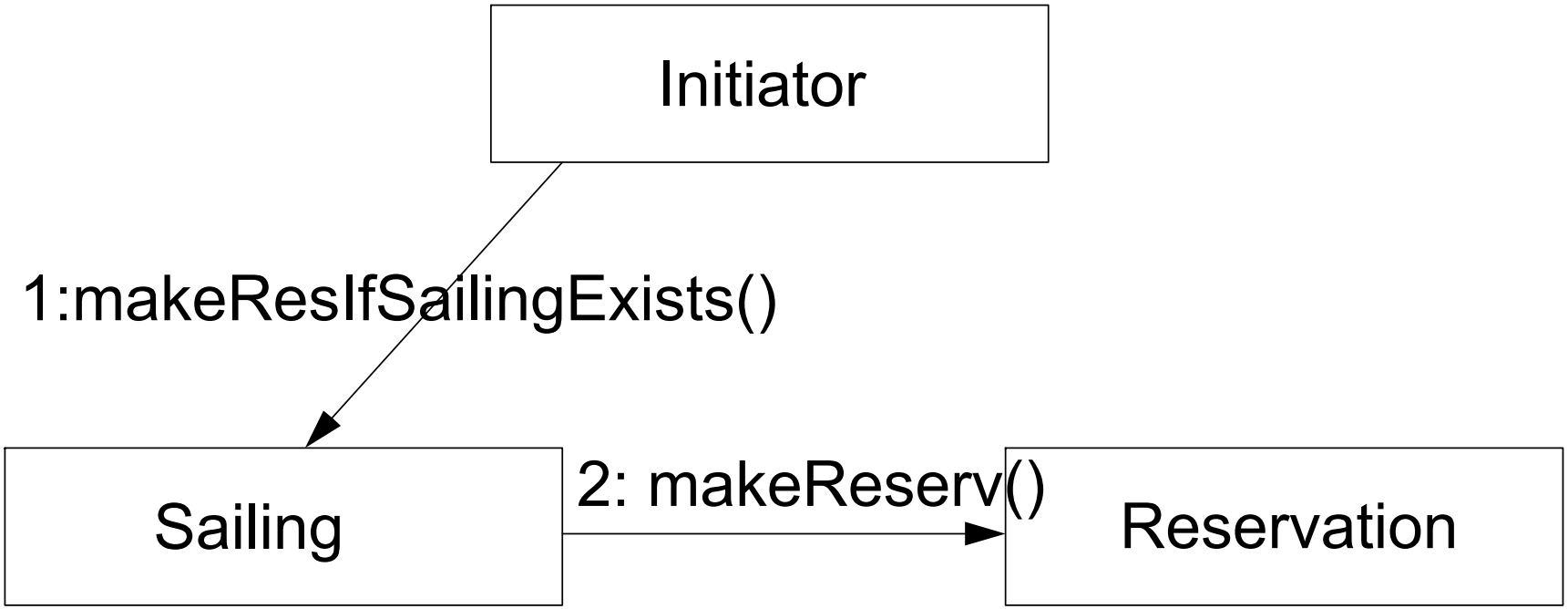
  - It may end up in the central module.
  - Or alternatively spread out if either of the following designs is adopted. It's not to be thought of as programming, but instead as documentation of the scenario logic from an architect's point of view.

8.5.5 ***Roundabout Route Scenario Design***

The name of this section is a Tront'ism and is *not* widely-used terminology.

The idea is that orchestration control is not centralized in one function, but is distributed.

- Control is passed from the scenario initiator (i.e. event detector) to the first module which must supply preliminary checking or data,
- then that module forwards the request either directly or indirectly to the final object.
- The control thus travels a rather "roundabout" path to the usually rather important terminal object.



Scenario Description

```
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.
```

When **`makeReserv()`** function is done, it:

- returns control to the Sailing,
- which in turn returns from the makeResIfSailingExits() to the initiator.

This design strategy is good if using

- asynchronous one-way messages,
- rather than function calls,

as it requires no data to be returned to callers.

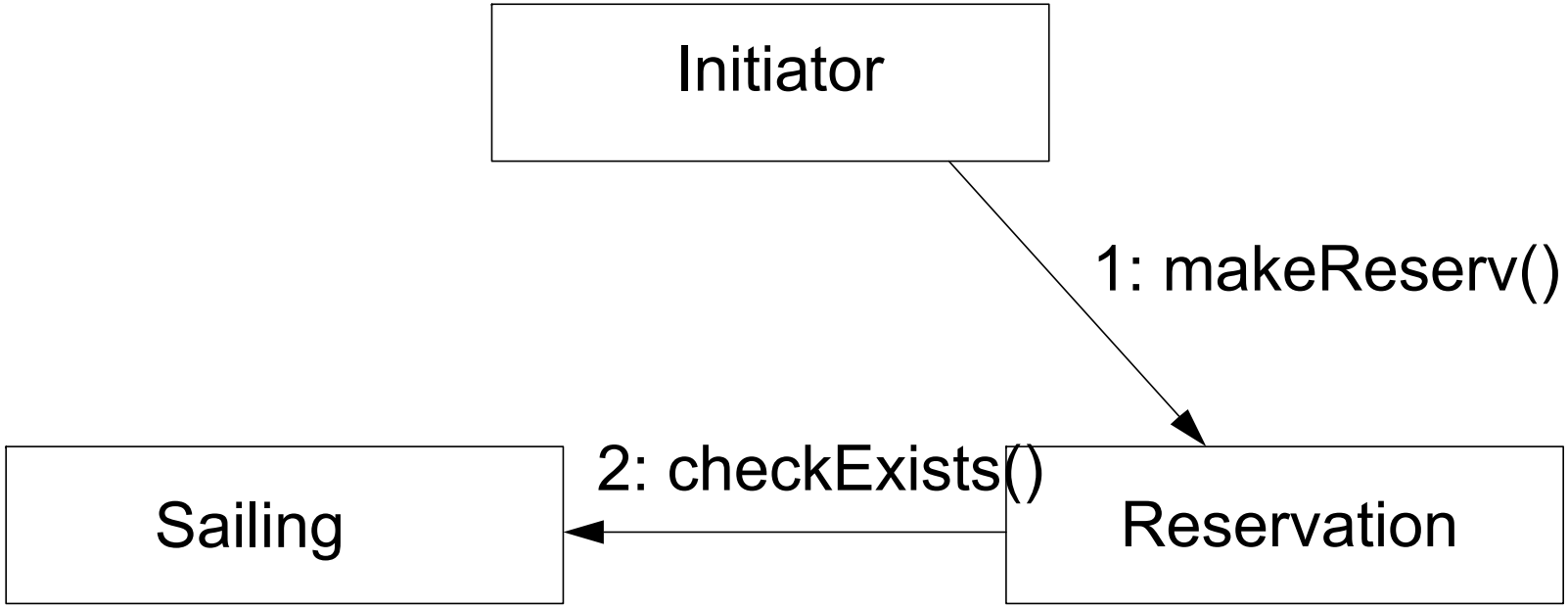
8.5.6 *Principle Object-based Scenario Design*

Another decentralized design alternative has the initiator first informing the principal application object involved (or class supervisor), in our case the reservation class.

- After that, the principal object (which may understand its creation needs best) does whatever necessary to accomplish the request.

In the example below, 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.



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 these diagrams don't show the function returns.

- But the above design requires an OK be returned to the reservation via a parameter/return value.
- Or if using one way messaging, a return message would have to be added.

8.5.7 I/O Library Call Placement

A troubling design question relates to whether all modules/objects should be allowed to call the I/O library.

- Or whether this privilege should be restricted.
- This is of concern, as one major maintenance headache is the possible future porting of the application to a different operating system or hardware platform (e.g. from MS-Windows to Mac).

- If likely, keep I/O calls confined to be:
  - from within a few modules (e.g. somewhat restricting you to centralized control),
  - or within only one module (each object sends its I/O requests to the UI module which is the only one, by architectural policy, allowed to call the OS I/O library).

This would reduces porting effort, as all I/O calls needing changing would be localized to a small number of source modules.

On the other hand, this distributes information about the data types of the attributes of *every* object needing I/O to the modules allowed to do I/O.

- It might be better if a student object (which defines the type and length of stud-name, address, phone) do it's own I/O.

- That way if:
  - a new attribute must be added,
  - or the length of the address field lengthen,
  - or type of phone number ever need changing,

  the changes would be restricted to this one object class's code.

There is no best answer.

- To port a distributed control application, you could always implement a translation module.
- Or perhaps C++ provides a good compromise.
  - Define the student phone number type in the student object, overload the output operator for this type, and then let cout<< and cin>> work as they see fit on that type.
  - Unfortunately, now that most UIs are GUI based, you must instead provide 'convert to ASCII' member functions for each attribute, and then the phone number (previously an integer) can be displayed via the GUI API.