

7. Design Methodology

In this section of the course, we will look at what a design methodology is. Major methodology families, namely Structured Analysis and Design (SAAD), Information Engineering (IE), and OOA/OOD, will be briefly compared. Then we will look in depth at one lovely method to synthesize object requirements from system requirements, thus learning a way to span the so called 'design gap'.

Readings:

- Chapter 6 of [Maciaszek2001]. Note that one of the things that Maciaszek discusses in this chapter is collaborations. A collaboration is just a society of classes or use cases that work together. These are not the same as collaboration diagrams, which are a kind of interaction diagram. Interaction diagrams in general show sequencing somehow. A collaboration diagram shows sequencing by numbered function call arrows, where the number indicates the sequence. Unfortunately, Maciaszek used the term 'collaboration diagram' to mean a diagram of a collaboration rather than a specified sequence of collaborating. To see that Maciaszek does actually know about proper, numbered UML collaboration diagrams, see his Figure 2.2 on page 32. In general, I will talk about collaboration diagrams in this chapter. However, one might say that my overall Object Communication Diagram (OCM) tries to illustrate the same thing as a general collaboration: the union of which class call which functions in which other classes.

Section Table of Contents

7. Design Methodology.....	7-1
7.1 Methodologies.....	7-3
7.1.1 Structured Analysis And Design	7-4
7.1.2 Information Engineering.....	7-5
7.1.3 Object-Oriented Analysis and Design	7-6
7.2 Key Design Strategy.....	7-8
7.2.1 Modelling Objects and Their Relationships First.....	7-9
7.2.2 Identification of External Events and Event Sources	7-10
7.2.3 Use Case Scenarios.....	7-12
7.2.4 Inter-Object Communication	7-13
7.2.5 Event Partitioning and Message Traces.....	7-15
7.3 Interaction Diagrams.....	7-18
7.3.1 Collaboration Diagrams.....	7-19
7.3.2 Sequence Diagrams	7-21
7.3.3 Fork vs. Staircase Interaction Diagrams.....	7-24
7.4 Insert Cmp212 Lectures 5.5-5.10 Here	7-29
7.5 Other Comments.....	7-30
7.5.1 Object State Models.....	7-32
7.5.2 Summary.....	7-33
7.6 References.....	7-34

7.1 Methodologies

A methodology is a way of doing things. According to Webster's dictionary, a methodology is:

- a particular set of procedures.
- the principles or procedures of inquiry in a particular field.
- a body of methods, rules, and postulates employed by a discipline.

As you know, there are often several ways to accomplish something. In fact, there are often several major classifications of ways of doing something. For instance, you can travel to Calgary by automobile, train, or airplane. Within each classification of methodologies, there are specific methodologies. e.g. travel by Volkswagen or by Cadillac, etc.

In the analysis world there are three major classes of design methodologies:

- 1) Structured Analysis and Design (SAAD) - which start with DFDs
- 2) Information Engineering (IE) - which start with ERDs
- 3) Object Oriented Analysis and Design (OOA) - which start with ORDs.

These major classes differ mainly in the importance and order that they develop the three parts (data, control, and process) of any model of the system. Within each class, there are a number of specific methodologies that have been defined and documented by book authors (e.g. [Shlaer 92] [Coad 91]). The specific methodology is often named after the author(s). Generally each methodology, *even within a class*, has an (annoyingly different) diagramming convention.

Also, each particular methodology captures and documents well slightly different advanced elements of the system compared to another, and/or adopts slightly different rules about the way the model is to be constructed (e.g. multiple inheritance is allowed, or not).

7.1.1 Structured Analysis And Design

This methodology [Demarco 79] was described briefly in CMPT 275. It revolved around constructing data flow diagrams (DFDs) that **show the union of all flows** needed to obtain all the various kinds of outputs that come out of an application at any time. These could be decomposed hierarchically. When suitably refined, a technique called either Transform Analysis or Transaction Analysis was performed. This resulted in a potential procedure call structure chart for the application, which could subsequently be implemented.

This technique works well for batch applications, and was ideal for magnetic tape processing. It is falling out of favor now that:

- interactive applications have become important
- object rather than procedure-oriented design is used
- and that the weaknesses of taking the union of flows has been realized.

Regarding the latter, it is important to capture the ordering that processing fragments are executed in response to a particular external event. ***Modelling the system with a diagram that takes the union of all the flows resulting from all the possible external user requests, hopelessly blurs the ordering researched so carefully during analysis!***

Later in the course we will examine data flow diagrams. They will be discussed in regards to SAAD enhancements (how they can be augmented with finite state machines to document ordering), and how in their simplest form, they can document process ordering (and ordering options) in response to a single external event.

7.1.2 Information Engineering

Information Engineering [Martin 89] is a methodology that has two aspects:

- a) It emphasizes the importance of enterprise-wide data (i.e. long term persistence and departmental sharing). It also emphasizes an early attention to data modelling (e.g. Entity-Relationship Diagrams).
- b) It denotes a design process where:
 - the departments that use the data are first identified,
 - then the applications that each department needs to do their job are enumerated,
 - then the top level menu items that each application needs are listed,
 - then the subsidiary menu items that each main menu item has on it are identified, and
 - finally the steps needed (possibly interacting with the user for data) to execute the subsidiary item “use case scenario” are specified and programmed.

IE has good applicability to menu-driven information systems. It is not so applicable to systems that have more than one source of external events (e.g. keyboard, mouse, network, clock, and process control I/O) that arrive in possibly unpredictable order. Nonetheless, we will look at some of the techniques of IE (data modelling via ERDs) now and others later, as IE has contributed several features to the art of design.

7.1.3 Object-Oriented Analysis and Design

Has the advantages of:

- applicability to interactive systems
- enhanced encapsulation of behavior with the data
- modularity, and design information hiding
- more suitable for applications with multiple, unpredictably-ordered inputs sources (e.g. which window will he click next, vs. the old prompt/wait cycle envisioned by IE).
- more suitable for distributed processing.
- code re-use by subclasses, and reduced object file size.
- more intuitive for humans
- being a very solid foundation on which to build your application’s design. They tend to be stable over time. You may have to add an attribute or an object to a system occasionally, but rarely during maintenance do you have to split or merge classes because each tends to be so fundamental on it’s own.
- Objects form excellent boundaries to use as the foundation of the decomposition of a complex system.

One very interesting design strategy that can be taken during OO design (not all OO methodologies are this good) is:

- 1) identification of objects and relationships first,
- 2) identification of external events and event sources,
- 3) plan the message-passing trace (i.e. sequence) required between objects for each use case scenario-starting event,
- 4) specify the object state lifecycles necessary for each object to handle each particular type of message appropriately (appropriate to the history leading it to its present state).
- 5) illustrate each state change’s necessary processing steps, and allowable process step ordering using an activity or data flow diagram,

- 6) finally, program the processing fragments necessary for each state change.

Not all OO design strategies use this pattern, but most use many of these steps. One of the ones that is most interesting, [Shlaer92], uses most of these steps. One of its advantages is that it decomposes the application in a lovely manner and deep enough that only very small/short code fragments need be written by the programmer. Small fragments are easy and less error prone to program for humans.

Note that this instructor will add to Step 3 significant material which is not in [Shlaer 92].

7.2 Key Design Strategy

The Shlaer Mellor methodology is an analysis technique that **decomposes the CONTROL PLAN up into four VERY SENSIBLE parts**, each of which represents (different) abstractions, and each of which is a reasonably graspable size. Remember, small size is necessary for humans to comprehend, specify, and review for correctness, *without making mistakes or oversights*. The 4 parts are:

- 1) Divide the system up first into (packages/sub-systems, and thence into) object classes (i.e. decompose by the hunks of data that must be retained, even if there not any persistence objects.). Determine the relationships between them. An entity or object relationship diagram is an appropriate output workproduct.
- 2) Reduce the emphasis on a main program and control modules. Distribute the control decisions to objects by assuming they are intelligent, capable entities aware of their responsibility to respond appropriately (i.e. to 'service') messages sent to them. The map of which objects send which messages to which other objects is called the object communication model (OCM diagram). The entities/nodes in an OCM can be thought of as 'islands' of intelligence, perhaps even state machines.
- 3) Decompose the intelligence within objects by regarding them as state machines, able to take different actions depending on which messages arrives while in which state. e.g. IF in_state_ready AND message_A_arrives THEN ELSEIF message_B_arrives THEN The decomposition involves identifying states and mapping which message-triggered transitions can occur.
- 4) When a state transitions occur, a certain amount of processing is usually necessary. An UML activity diagram or a SAAD data flow diagram can be used to document this. And since this flow is a result of a single event, a DFD can show **ordering/precedence** (i.e. control) information. (Note: This would not be possible if a DFD were documenting the union of flows from several kinds of events that could hit the object).

7.2.1 Modelling Objects and Their Relationships First

[Shlaer92] is fairly interesting in that its roots (authors) come from the real-time and process control application area.

Often such applications have complex timing and control aspects, and modelling the data to be retained (in an IE ERD way) would not, at first, seem to be a priority. This is especially true since such applications may not typically have large information storage requirements, or any disk requirements!

The fact that a methodology exists that has come from such a background, and yet concentrates on retained data, is further evidence to bolster the database-related IE methodology's correctness in emphasizing data first. Since the modern OO class of methodologies is supposed to be applicable to most, if not all types of applications, it is not surprising to see this trend even coming out of the real-time and process control area.

In the previous two major sections of the course, we have discussed data modelling, and I think no further discussion is needed here. The point here is that it is these entities that we plan to embody with member functions that get the processing done. Even in real-time embedded software, if we don't find the classes first, where would we considering putting the functionality?

7.2.2 Identification of External Events and Event Sources

If you look at almost any computer, it spends more than half its time (and often the majority of its time) *doing nothing*! Essentially, it has been programmed to either:

- loop endlessly either waiting or polling for some external change to its inputs, or waiting to be interrupted from its “busy waiting” by an external signal, upon which it will handle this ‘external event’.
- Or it has executed a machine language HALT instruction, and is dormant pending an interrupt (this saves power on a battery-powered laptop).

External events can take the form of keyboard entries, mouse clicks, incoming network packets, clock ticks, and inputs from hardware interfaces to the real world).

Any methodology that hopes to document what a system must do, must address the question “do what, in response to what?” In essence, an computer is an event-response machine: when it detects an external event, it should be designed and programmed to respond ‘appropriately’.

The importance of external event identification was first introduced in “Essential Systems Analysis” [McManamin84]. Though McMenamin's methodology was basically SAAD, he makes a particular point of examining the external events that a system handles. This is because he and his co-author were trying to extend the DFD-based SAAD with state machines to specify when (i.e. in which response to which external events) and in which order the processes in the DFDs should be activated.

First, event sources must be identified. Events can be internal like keyboard, mouse, or clock tick. Others can be external like networks or sensors.

Then, **a list of each different external event** from EACH source of EACH kind must be made. Generally, a big system handles so many

events, that there has to be some way to partition them into manageably small and analyzable groups.

My primary suggestions for abstracting/partitioning possibly hundreds of events into groups (subclasses?!) are:

- You can group events that affect the same data. A good example can be seen in some of the projects I have given in 275. Consider an automobile ferry reservation system. The core operation is to take reservations, delete reservations, and prevent overloading. But the administrative operations triggered by user requests (i.e. events) are things like add/modify/delete ferry vessel, add/modify/delete sailing, etc. Notice how these latter two group together operations on the same data!
- You can group events by the IE method of decomposing by department, departmental application, main menu and then sub-menu operations. The event is the user selecting a particular leaf menu command. Another way to think of this is to group by operations that ‘belong’ together in a sub-menu.

IE and menu-driven systems were barely started in 1984, but [McMenamin84] made these event grouping suggestions:

- events that are related temporally (e.g. student enrolls, then student registers in courses, then student graduates) are candidates for grouping. i.e. events that push a lifecycle of a particular object along (not that [McMenamin84] discussed objects or lifecycles).
- an external event which affects the way another type of future event is handled is a way of grouping events. These two events are obviously related somehow.

7.2.3 Use Case Scenarios

A use case scenario is *a description of a particular sequence of interactions a user has with a system to perform a particular user operation.* e.g. Make ferry reservation: 1) select operation, 2) read prompt for sailing and reply, 3) read prompt for vehicle information and reply, and 4) note reservation made or no space message appears.

Note that a Use Case describes a type of interaction a user has with a system. There may be several alternate scenarios for a use case, depending on whether the user enters bad data or not, etc. Thus a scenario is a particular variant of a particular use case.

Generally a use case scenario is started by an **external event**, and progresses through a number of steps, before ending leaving the computer idle or ready to start a new scenario

Scenarios infect the development process. They are:

- identified and documented in the requirements analysis phase.
- detailed from user’s point of view in the external design subphase (e.g. draft user manual) so architectural behavior can be done next.
- architecturally planned using message traces on a single scenario object communication diagrams during architectural design.
- tested during the system test phase.

You can see that scenarios are a foundational part of software development and will have key prominence throughout the software lifecycle.

7.2.4 Inter-Object Communication

We need a new kind of diagram to express the ‘architectural’ response to a particular external event. These are called Interaction Diagrams in UML. There are two variants of interaction diagrams: collaboration diagrams where sequencing is numbered, and sequence diagrams where time is basically down the page. We will see these shortly.

[Shlaer92] also has a communications diagram called an object communication model (OCM) diagrams, though this instructor (R. Tront) tends to call them Object Communication Diagrams (OCDs). They basically show all the interactions between classes, rather than just the interaction during a single use case scenario.

The high level control paradigm used in most OO systems is that objects communicate using messaging. In other words, each object can send **messages** to inform other objects of particular **internal or external event occurrences** and any data associated with that occurrence. In many implementations, these would just be function/method calls. However, there are other kinds of messaging, like network messages from one program to another, and which might not have a ‘return’ (c.f. regular function calls).

The general concept of constructing a system via communicating objects allows the receiving object to do what it thinks is best with this event occurrence. In this way, objects are simply servers which only have to know what event message types they have to service, and *not* about who might send them. Their job is abstracted to simply responding appropriately to messages destined for themselves which announce event occurrences and parameter data of interest.

The objects in an collaboration diagram or an OCD are not necessarily orientated relative to each other as on the ERD. An object that was on the left of the ERD may be put on the top of the OCD. It is better to arrange the objects on the OCD in a layered manner. Generally, the more application-aware, user/external interface classes are put near the top. Controller/coordinator objects are in a middle layer. The lesser

intelligent and more utility/service/storage objects (possibly not even having state machines) are put near the bottom.

You should regard the objects in an interaction diagram or OCD not so much objects as islands of intelligence that can receive messages and take appropriate action. Note that an object with both instance and supervisor/shepherd aspects could be considered two islands in one class!

External events that trigger scenarios can be categorized as either:

- a) **solicited**, if the external user (e.g. external actor) was prompted to act on the system (e.g. to enter some data), or
- b) **unsolicited**, if the external event is not part of a message thread initiated by a previous scenario.

A **message trace** is the *sequence* of messages/calls that occur in response to a **particular** unsolicited external event. i.e. that occur as a particular use case scenario is being executed. A message thread trace can be either:

- **top driven** - where the external event that started the trace was sent to one of the top objects in the OCM. Or,
- **bottom driven** - where the external event that started the trace was initially received by one of the bottom objects in the OCM.

A trace can take a temporary visit to the outside to solicit information from the user or some external source (e.g. network), before terminating to leave the computer idling.

The **OCD is the union of all message interactions** that take place during the response to **all** external events.

7.2.5 Event Partitioning and Message Traces

Since humans frequently make mistakes when dealing with too many considerations at once, decomposition of the problem of characterizing the desired behavior of the system is a desirable goal. The concept of **event partitioning** was introduced in [McMenamin84]. The idea was to partition/**focus** the designer's attention at documenting the appropriate behavior of the system in response to **one single scenario-starting event at a time**. Though this sounds like an obvious concept, it was not widely recognized at the time, and is still not too widely practiced. Did you do it in 275? Did you write the draft user manual that dealt with one user operation scenario at a time? Before architectural design? Unfortunately, the user manual does a good job of defining each user scenario but not of determining the architecture of the system, and the behavioral requirements for each object classes' nature.

I suggest that the best way to design a system is to hypothesize/plan/design each scenario's implementation message trace on a bare (initially no events shown) OCD. I call this a '**single scenario OCM**'. Do this starting on a bare OCM for each user scenario started by a particular unsolicited external event. For example, user selects "Make Reservation" from Customer sub-menu of a ferry reservation system. Identify the sequence thread of procedure calls, starting from the User Interface (UI) object and proceeding from object to object, possibly including a detour back to the UI for solicited input, until the entire operation is complete.

For instance, if adding a new customer reservation on a ferry sailing:

- Does the UI object first send a message (i.e. call) the sailing instance to see if the sailing exists? If all is OK, then the sailing instance should reserves space and send a message to the customer reservation associative class to create an instance of a customer reservation.

- Or does the UI first send a message to the customer reservation object, which then sends a message to the sailing object to check for and reserve space? The reservation waits for a return or reply before creating an instance of a customer reservation.
- Or does the UI first send a message to the sailing object, wait for a reply, and then send a message to the customer reservation object?
- Also, does the UI solicit all the data initially, or does it ask for the sailing first, check it exists and if not ask the user to correct their typing, and only then ask for the customer name and address.

This is exactly what is meant by *architecture*.

Here's another example of a scenario, from a different application, that would have an interesting message trace. Consider a message-passing police department:

- 1) A crime report object instance is created by the 'create crime report' user command. The crime report instance sends a message to the police car shepherd object, which assigns a car from the limited number available.
- 2) Police patrol car object instances send requests for work to the car shepherd, and get given a reference to a crime report object. The patrol car accesses the particular crime report and goes to the scene of the crime. The officer notes down the particulars of the case, and updates the crime report. The patrol car officer enters a judgement/decision that the crime needs further investigating, and sends a message to the detective shepherd object.
- 3) The detective shepherd assigns a detective car (from the limited supply of detectives) to one of the many crime report objects needing investigating, based on severity of the crime and other info.
- 4) The assigned detective investigates the crime further and determines a suspect. The detective sends a message to the patrol car shepherd object to arrest the suspect at a particular address.

- 5) The patrol car shepherd assigns a car to the arrest request, the suspect is picked up, and that crime report object archived and deleted.

I hope this helps you get the idea of a message passing architecture using control encapsulated in objects, but distributed/shared among a number of classes. An algorithm to get a use case scenario done does not necessarily have to be all in one class.

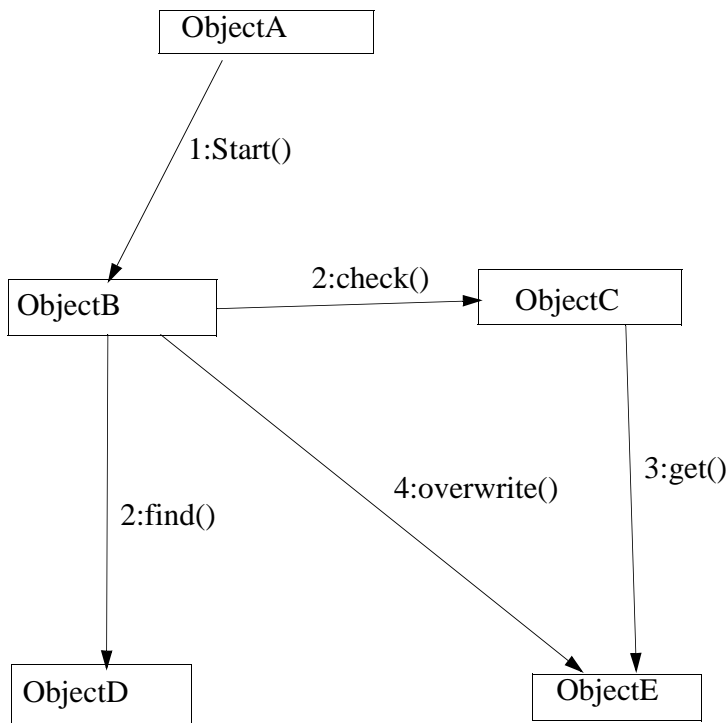
7.3 Interaction Diagrams

In earlier sub-sections of this methodology material, we introduced the concept of a scenario message trace. This was the sequence of interactions between objects, expressed at the architectural level, required to implement the response to a particular external event.

UML provides two different but almost equivalent diagrams for drawing object interaction: collaboration diagrams and sequence diagrams.

7.3.1 Collaboration Diagrams

A collaboration diagram shows numbered arrows labelled with function or message names to indicate the time ordered progression of a message trace. An example is shown below.



The numbers before the colons indicate the required sequencing of the message sends necessary to effectuate the operations required to respond correctly to the externally-started scenario.

Note that ObjectE is specified in the illustrated scenario as being ‘gotten’ before it is ‘overwritten’, otherwise data necessary to the

correct functioning of the application will be destroyed by the overwrite. Thus get() is prefixed with a 3:, while overwrite must come later as indicated by the prefix 4:. This is the **prime purpose of scenario modelling**: to reason about, and to plan/document the correct ordering.

Also note in the above diagram that there are two messages that are labelled with the number “2”. This indicates that they may take place in either order, or even in parallel, *without affecting the correctness of the response* to the external event.

Another methodology called Fusion described in [Coleman94] suggests an even more elaborate prefix adornment to the messages shown in a collaboration diagram. They suggest:

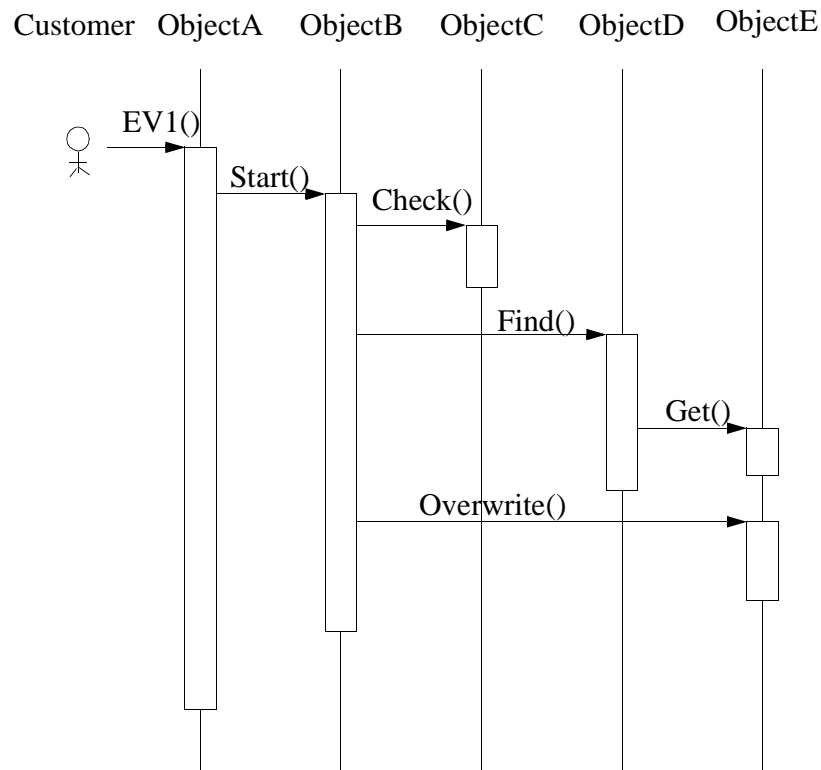
- That the presence of both a 2’ and a 2: on different messages in the trace indicates either one or the other message is sent, but not both (the apostrophe indicates boolean NOT).
- 2* means that the message is sent several times in a row, possibly from within a loop, before message 3 is sent.
- 2.1 and 2.1.1 have additional meanings.

Unfortunately, traces for a scenario are not very convenient for documenting the trace with a narrative justifying the particular design choice of message ordering and scenario handling.

It is very common to supply a narrative to indicate what each step is doing. The narrative for a use case discusses only externally visible behaviour, and discusses it in customer terms. In contrast, the narrative for an interaction diagram describes internal program functioning, and describes in terms that are perhaps very technical or implementation name specific (e.g. Java Swing GUI, Oracle Database connector, etc.).

7.3.2 Sequence Diagrams

An alternate interaction diagram format is the so-called UML ‘sequence diagram’. Like collaboration diagrams, they illustrate arcs between nodes in a directed graph. But they have the advantage of offering a linear axis to work with, in addition to just documenting which nodes are connected to which other nodes. Though the linear axis is not shown, everyone who works with sequence diagrams knows that **time goes down the page**.

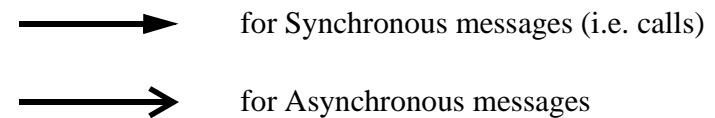


This is a great diagram. It shows the same information as a directed graph like a collaboration diagram, yet has a linear time axis. It shows which object is to initiate what actions in which others in which order.

One drawback is that it forces a time line, and thus can't show two messages that could be sent in either order, or in parallel.

The tall skinny rectangles represent the time the thread of execution control exists in/through the object. Remember that the control thread is in an object, even if that module has synchronously called another. It has temporarily passed control to another, but it still retains control when it returns. So the height of the rectangle suggests, for synchronous calls, the duration between its reception of a call, and its return to the caller. Note that it is not always necessary to use the tall skinny rectangles to indicate the duration the control thread is in a particular object; sometimes we just use a simple vertical line.

Also note that the diagrams can look a little different if asynchronous messages are being illustrated. [Jacobson92] suggests using:



Of course splitting threads of control are possible with asynchronous messages.

Additionally, information is normally written (in the space on the left if there is room, else below) describing what each step in the scenario is to do, and under what conditions. The narrative can include IF conditions regarding whether a message will be sent, and loops indicating that messages can be sent repeatedly. Thus sometimes the narrative looks a lot like pseudo-code, though this is not because this code will ever become source code. The source code that gets executed for a scenario is spread out in a distributed manner over many objects that participate in the scenario; this pseudo-code just documents the *logic* which, when

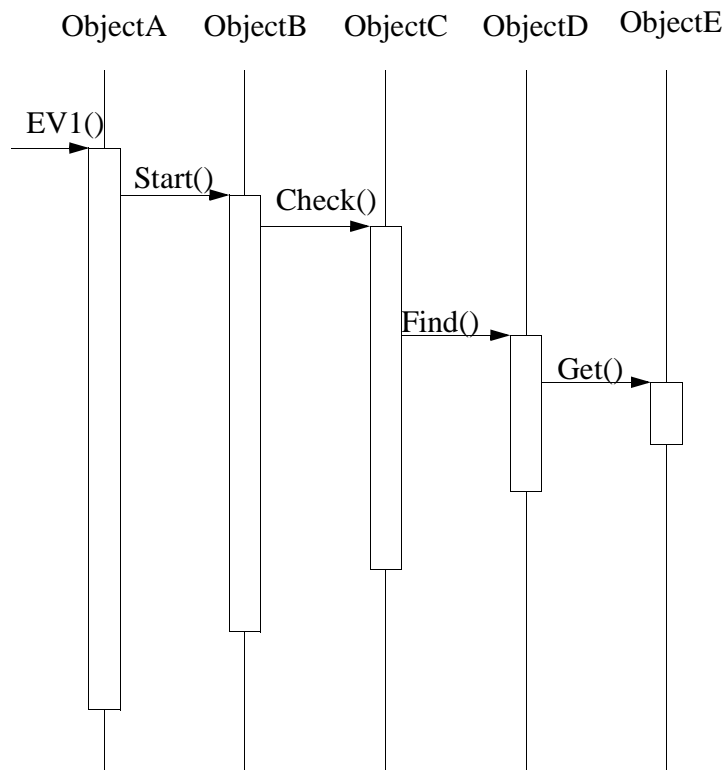
spread among the objects, will control the required response to the scenario.

At the bottom of the page, you can document why you, as the scenario response architecture designer, decided to do it a particular way. Remember, often there are several options as to how you might have architected a response. Designing how the trace flows is A MAJOR ASPECT OF DESIGN.

7.3.3 Fork vs. Staircase Interaction Diagrams

[Jacobson92] provides a great discussion of when one architecture might be better than another.

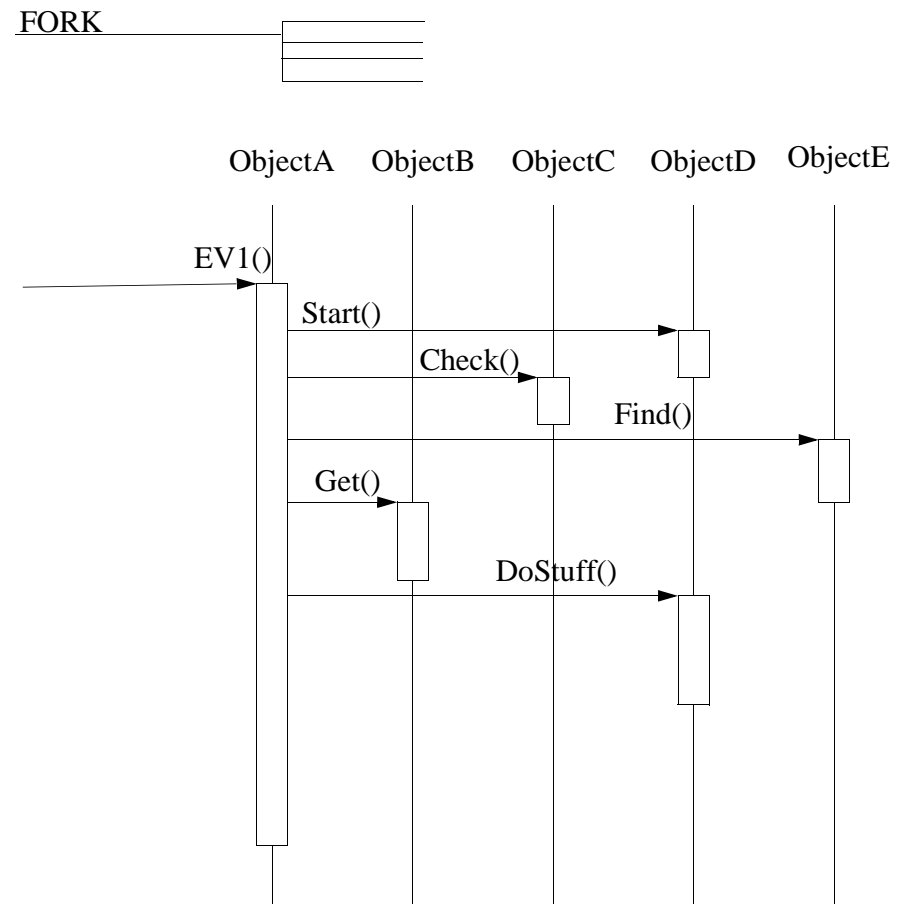
If using a decentralized control architecture for your scenario, decisions and actions regarding the rest of a scenario are delegated to lower and lower level objects (i.e. abstractions) in your OCD layers. An interaction diagram for such an architecture takes on a staircase-like nature with both a down to the right, followed by a down to the left nature. This is particularly noticeable if you put the objects left-to-right across the top of your interaction diagram in the order of nearest the external event to furthest from the external event (though this somewhat assumes a top-driven thread). The result is as shown below.



Note that if using synchronous messages, there would be left-going arrows additionally on the bottom staircase.

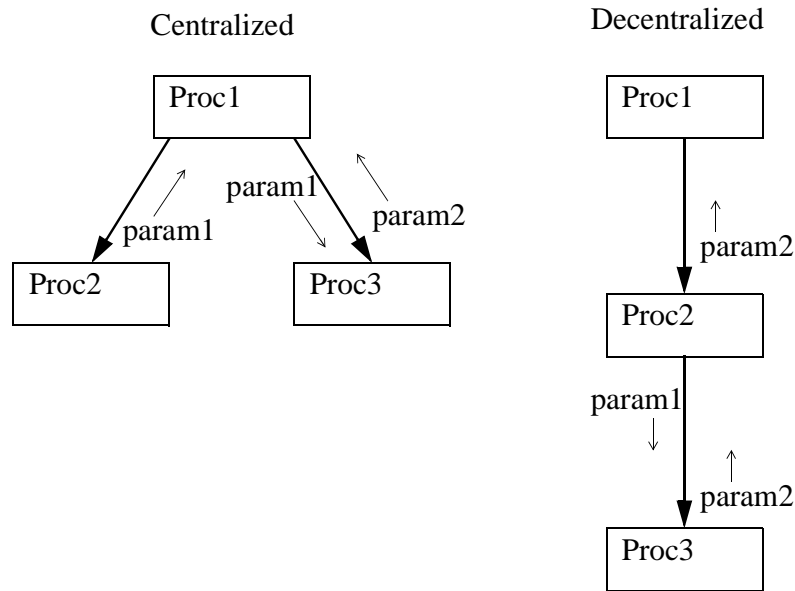
Optionally, it is interesting to examine Figure 5.4.2 of [Shlaer92] for their so-called ‘thread of control’ chart.

On the other hand, if using a centralized scenario control architecture, there is usually a primary object which manages that control. The primary object is usually drawn as the leftmost one in the diagram. It calls each of the objects required to handle the external event, in the proper order, directly. As a result, the interaction diagram looks somewhat like a fork with it’s tines to the right.



Though not show, there is no reason why an intermediate object could not call one further to the left, and still be mostly fork-like.

Jacobson suggests that the decentralized architecture is more object-oriented and generally more cohesive. Let me see if I can explain this. The following procedure call structure chart examples contrast the two ways to look at the centralized vs. decentralized question.



Notice that in the decentralized case, the top level procedure need not deal with parameter “param1”. In the centralized structure, the top level procedure must know about and deal with both “param1” and its type! Of course procedure calls are just synchronous messages.

Advantage of Decentralized/Staircase Scenario Architecture:

- If a maintenance change must be made to the way, say, Object D handles a message, it likely will only require a change to Object D.
- There also tends to be less parameter passing and better parameter type cohesion.
- Is good if in future only need to change the *nature* (not order) of the response by various objects.

- Good if the objects are strongly related by aggregation, by an aggregation hierarchy (e.g. in a wordprocessor: chapter, subsection, paragraph, word), by an inheritance hierarchy, or by a immutable temporal order (e.g. order, invoice, delivery, payment)

Disadvantage of Staircase Scenario Architecture:

- If a maintenance change is needed in the *order* in which objects must be notified and must react to the external event, then the possibility exists that many objects must be modified during such a change.

Advantage of Centralized/Fork Scenario Architecture:

- If a maintenance change might in future be needed to the *order* in which objects are notified and react to the external event, then only the primary object must be modified. The ordering is nicely encapsulated there.
- If *during* a program’s *actual execution*, the sequence order must be changed on the fly, then this control is better encapsulated in one central place.
- If during maintenance, it is likely new elements to a sequence of operations must be added or deleted, then centralized control is best.

Disadvantage of Fork Scenario Architecture:

- Does not encapsulate/abstract behavior (other than ordering) as well as the centralized/staircase architecture.

You can’t win, but you can do some scenarios fork and others staircase.

7.4 Insert Cmpt212 Lectures 5.5-5.10 Here

This section is like a UML use case <<include>> dependency. Please download and read these specified sub-sections of R. Tront's Cmpt-212 lecture notes. You should be able to find a copy on the Cmpt 370 web site under the name 212s05.OOAD.pdf

7.5 Other Comments

When using Structured Analysis and Design (ASSD) we are advised to draw data flow diagrams that are the union of all flows for all use cases. This union does not accomplish much (except perhaps to illustrate all processes that access a particular data file).

In contrast, what is most interesting about the synthesis union is that a **message trace union actually accomplishes something much more significant**. It accomplishes a very useful design task. **It builds the requirements spec for each class's exported functions/methods.** All you have to do is look at all the different types of internal message arrows terminating in a particular object, and you realize that that object must export one function or each kind of call! If a stack object has Push, Pop, and Full messages going into it, you know it needs to export those functions (and also know you do not need Top() and Empty() function)! I think this is a stunning feature of the trace union technique.

You might also ask what all the internal message arrows coming out of an object tell us. Well, they indicate one of three things. Either:

- 1) An outward arrow could be a **responsibility** to notify another object of some event, or
- 2) An outward arrow could be a **responsibility** to check with another object before doing something, or
- 3) An outward arrow could be a **delegation** of all or some part of a responsibility to another object. A good example might be delegation of the calculation of ferry sailing duration to the ferry object (which knows it's cruise speed). The sailing instance sends the ferry instance the route distance, and knowing it's cruise speed, the ferry instance calculates and returns the expected duration enroute.

In closing this section I would like to make the following points:

- But I don't know of a single methodology or CASE tool which officially has the union as part of its important nature. It is possible to kluge Rational Rose into doing this though, and instructions will be given later.
- This is a very recent technique that I think some of the major OO methodologists may have a blurry idea of, but don't realize its significance in bridging the so-called design gap. Some indicate that you should draw interaction diagrams for the important use cases/scenarios. But they don't seem to appreciate the ability to completely synthesize each class's member functions if ALL the use case scenarios are drawn!

7.5.1 Object State Models

In the second part of the control plan, each object is given a state machine which:

- gives it processing intelligence, and
- breaks the processing intelligence into smaller, more understandable pieces (pieces for each individual state change or transition).

An object's state machine is a control plan for the response to events external to the particular object (not necessarily external to the system). In particular, it provides a specification of what processing should take place given the mode the object has progressed into over time (i.e. the **historical context** which affects the way it should respond to events).

Often an object should respond differently to the same event type, depending on which state it is currently in. This is a very common kind of intelligence needed at this level of decomposition. It is also basically just a different embodiment of IF THEN ELSE logic.

You should note that OO methodologists [Booch94] [Rumbaugh91] suggest using state models only for objects that have particularly complex mode or state dependent behavior. [Shlaer92], in contrast, uses it for almost all processing. This instructor (R. Tront) thinks it is important for every class instance or static (shepherd) aspect that will need a reaction that is dependent on historic context.

7.5.2 Summary

When you survey the references in this course's lecture notes, you are seeing the birth and adolescent development of analysis and specification techniques for computer software! There will be more developments yet in the coming years, before we get near being able to specify software easily like we can specify buildings (e.g. with all their structure, wiring, plumbing, and heating).

7.6 References

[Booch 94] "Object-Oriented Analysis and Design" 2nd ed., by Grady Booch, Benjamin-Cummings Publishing, 1994. (Note: the first edition is also good in that it has case study chapters in a variety of OO programming languages.)

[Coad91] "Object-Oriented Analysis" by Peter Coad and Ed Yourdon, Prentice-Hall, 1991.

[Coleman94] "Object-Oriented Development - The Fusion Method" by Derek Coleman et al, Prentice Hall, 1994.

[Demarco 79] "Structured Analysis and System Specification" by Tom Demarco, Prentice-Hall, 1979.

[Martin 89] "Information Engineering" Vol. 1,2, and 3, by James Martin, Prentice-Hall, 1998.

[McMenamin 84] "Essential Systems Analysis" by Stephen McMenamin and John Palmer, Yourdon Press, 1984.

[Rumbaugh 91] "Object-Oriented Modelling and Design" by James Rumbaugh et al, Prentice Hall, 1991.

[Shlaer 92] "Object Lifecycles: Modelling the World in States" by Sally Shlaer and Stephen Mellor, Prentice-Hall, 1992.

[Ward 85] "Structured Development for Real-Time Systems" Vol. 1,2, and 3, by Paul Ward and Stephen Mellor, Yourdon Press, 1985.