ECE155: Engineering Design with Embedded Systems Winter 2013

Lecture 33 — April 4, 2013

Patrick Lam *version 1*

Unified Modelling Language

The *Unified Modelling Language* (UML) is a language for specifying and documenting the architecture of large object-oriented software systems, using diagrams. UML version 2.2 includes 13 different diagrams, which can summarize a system's intended structure (e.g. classes), behaviour, and interactions of components.

Historical Note. UML combines earlier modelling languages, and was initially developed by Grady Booch, James Rumbaugh, and Ivar Jacobson (The Three Amigos) in the 1990s. Version 1.4.2 of UML is an ISO standard, and future versions of UML are controlled by the Object Management Group consortium.

Resources. There are a lot of webpages (some inaccurate) with information about UML. Here are some resources:

- The official UML page is at `http://www.uml.org`; it contains links to the official specification, tutorials, and links to tool support.
- One useful tutorial on practical UML is at `http://edn.embarcadero.com/article/31863`. It contains sample diagrams, and also broken links.
- The definitive UML book is:

  Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide,* 2nd Edition, Addison-Wesley, 2005.
- Another book is

  Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell.* O'Reilly Media, 2005.
- Official specification: `http://www.omg.org/technology/documents/formal/uml.htm`.

Benefits of UML. We are teaching UML because it is a well-known language for talking about software designs. In particular:

- UML is a visual language, and UML models are diagrams, so they're easy to glance at.
- UML models are largely self-documenting.
- UML is agnostic in terms of software processes or development lifecycles.
- UML is an open standard (not owned by any one company).
- UML is good for object-oriented languages, which are quite common in industry.

Criticisms of UML. Here are a few complaints one might have:

- UML is all syntax, no meaning.
- UML contains redundant and infrequently-used constructs.
- UML is complex and difficult to learn.
- UML only works for object-oriented languages.
- UML tools don't play nicely together.

UML Diagrams. Let's look at the different categories of UML diagrams in UML 2.2:

- *Structure Diagrams* represent the static application structure.

  Examples: class diagrams, object diagrams, component diagrams, composite structure diagrams, package diagrams, deployment diagrams.
- *Behaviour Diagrams* encode the usage of components, the activity of components, and finite-state machines summarizing components' states.

  Examples: use case diagrams, activity diagrams, state machine diagrams.
- *Interaction Diagrams* describe how components interact; they may communicate or synchronize with each other, potentially respecting timing constraints.

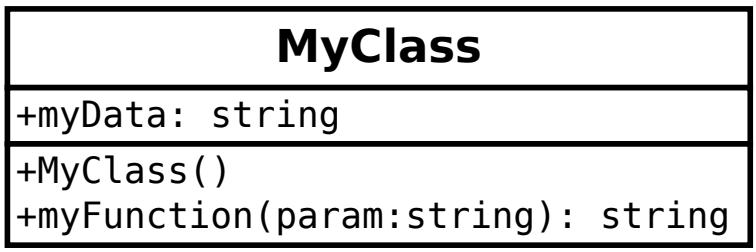
  Examples: sequence diagrams, communication diagrams, timing diagrams, and interaction overview diagrams.

Tool Support. A number of software tools can create UML diagrams, including Microsoft Visio and `dia`. Some tools can generate source code from UML diagrams and UML diagrams from source code; if a tool can do both, we say that it *round-trips*. Tools can also, to some extent, automatically generate test cases and test suites from UML diagrams.

UML Class Diagrams

The basic UML diagram is the class diagram, which describes the members of a class. You saw this in ECE150, although maybe it wasn't called a class diagram. Members include attributes/fields and operations (constructors, destructors, methods, indexers, and properties).

Here's a simple example of a UML diagram.



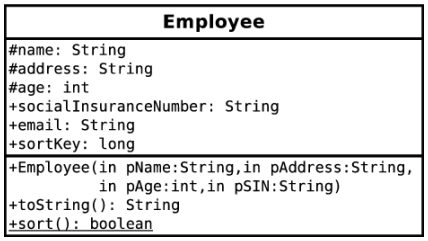
UML Symbols. Note that the above diagram includes a + for public visibility. Here are the possible visibility symbols:

- A + (plus sign) indicates public visibility.
- A - (minus sign) indicates private visibility.
- A # (hash sign) indicates protected visibility.

UML also uses the following symbols:

- A ~ (tilde) indicates a destructor (in front of an operation) or package (as a visibility symbol).
- An .. (ellipsis) indicates a range of values.
- A : (colon) separates a name from a type.
- A , (comma) separates items in a set.

More Complicated Example. Here are some more features of UML:




```
class Employee {
  protected String name;
  protected String address;
  protected int age;
  protected String socialInsuranceNumber;
  public String email;
  private long sortKey;

  public Employee(String pName, String pAddress, int pAge, String pSIN)
    { /* ... */ }
  @Override public String toString() { /* ... */ }
  public static boolean sort() { /* ... */ }
}
```

Class Hierarchies in UML

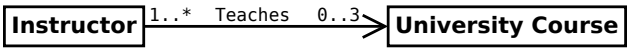
UML includes a number of notations for representing relationships between classes and instances.

The following relationships are between instances of classes.

- An *association* represents a relationship between instances of two classes.
- An *aggregation* is a type of association, typically between a collection or container instance and its contents.
- A *composition* is another type of association, typically used between a container and its contents. For a composition, the contents don't make any sense if the container isn't around.

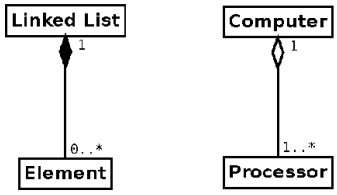
On the other hand, a *generalization* relates base classes (not instances) and their derived classes.

Associations. There is an association between two classes if there is a relationship between instances of the classes. This is one of the "has-a" links between classes. Typically one of the classes can call methods on the other.



Instructors and courses are associated, but there is no subtype relationship, and the types are not collection types. Instructors teach between 0 and 3 courses per term. Each course has at least 1 instructor, but possibly an unlimited number. A method that the instructor might call on the course might be `giveLecture()`.

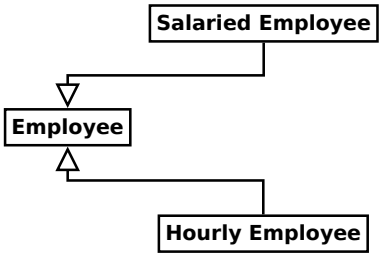
Aggregations and Compositions. These specialized types of associations also represent "has-a" relationships, but more "part-whole" relationships.



A Linked List contains 0 or more Elements, but Elements belong to only one Linked List; however, the composition means that we are saying that Elements may not exist independently of a Linked List.

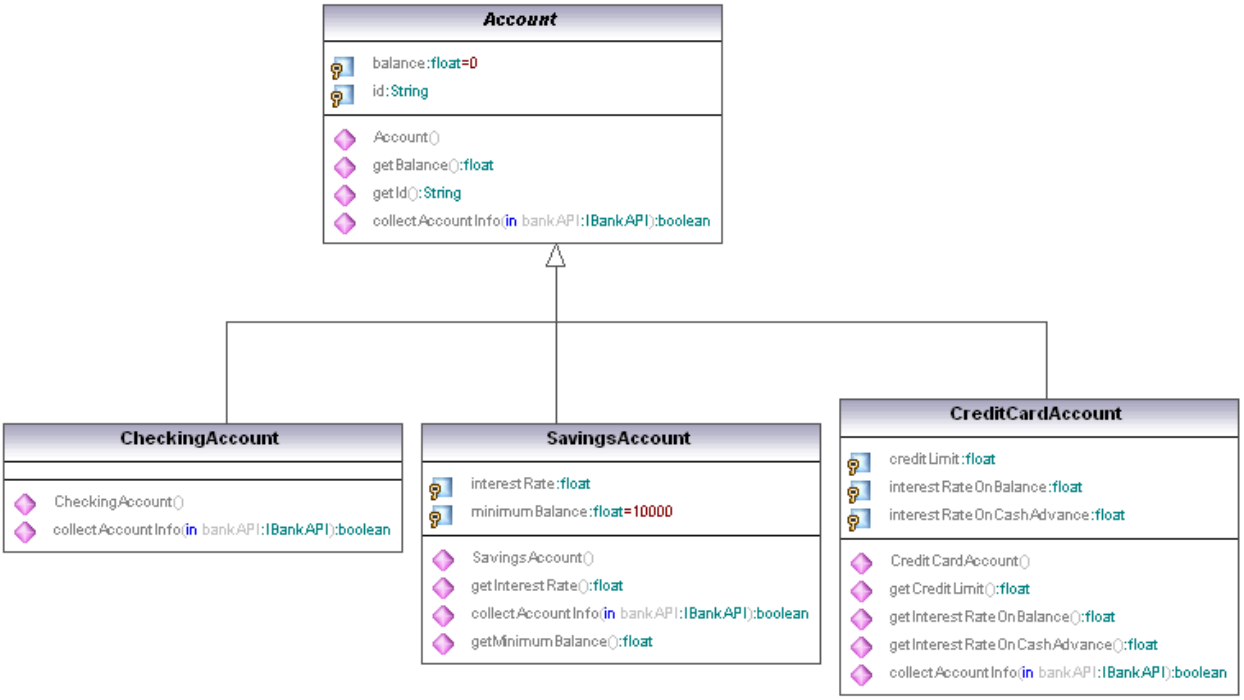
Computers contain 1 or more Processors, but Processors only belong to 1 Computer. In a composition, we are stating that Processors can exist independently of a Computer.

Generalizations. These relationships are between classes.



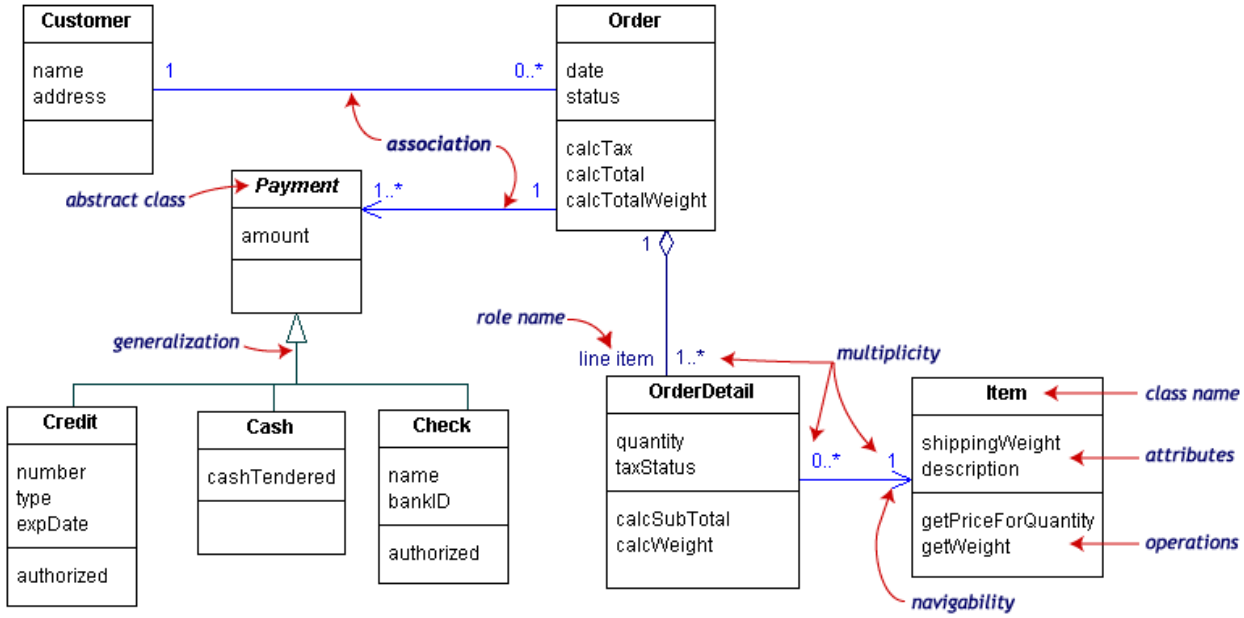
Every Salaried Employee and Hourly Employee is an Employee, so Employees are generalizations of the Salaried and Hourly Employee classes. In the code, you'd see that Salaried Employee and Hourly Employee are subclasses of ("extend") Employee.

Fancier UML Class Diagrams. Here is a complex UML class diagram from `http://www.altova.com/umodel/class-diagrams.html` (accessed March 10, 2011).



Note the member field initializations and the generalization link.

Here is another class diagram, from `http://edn.embarcadero.com/article/31863` (accessed March 10, 2011).



UML State Diagrams

Goal. You should be able to read a description of a finite state machine (either in code or in text) and produce a syntactically correct UML state diagram summarizing that design.

(The UML State Diagram examples are from *UML Distilled, Second Edition*, by Martin Fowler with Kendall Scott.)

We've talked about finite-state machines a few times in ECE155 and your other courses. It is possible to document these machines in UML using state diagrams. Here's an example.

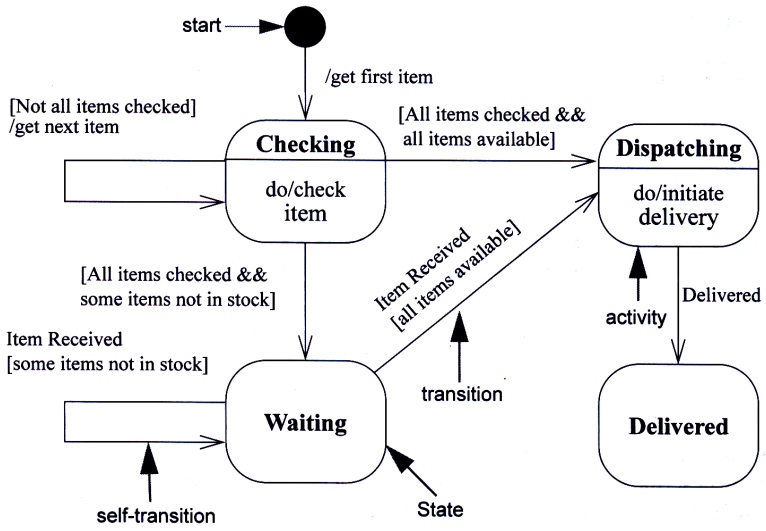


Figure 8-1: *State Diagram*

States. This state diagram illustrates the states of an order in an order processing system. The order object has four states, represented by boxes, with the name of the state in the upper half of the box: "checking", "dispatching", "waiting", and "delivered". The lower half explains what the activity in that state is; for instance, the "dispatching" state initiates the delivery of the order. In ECE155, you can always put "do" before the slash and the activity after the slash.

In general, a state has a name and may have an activity. Objects carry out activities, and these activities take some amount of time.

Transitions. The edges in the graph represent transitions between states. Transitions may optionally be labelled. There are three parts to a transition label, as follows: *Event [Guard] / Action.* For instance, the self-transition from the "checking" state contains the guard "[not all items checked]" and performs the action "get next item".

An event is something that happens to an object, like "Item Received". A guard is a logical condition that states the conditions under which the event may occur (e.g. "[some items not in stock]"). Objects carry out actions, e.g. "get next item"; actions must occur quickly and be uninterruptible (contrast this with an activity, which takes longer and gets interrupted by events).

An object takes a transition if the event occurs and the guard is true. Before it completes the transition and enters the destination state, it carries out the action.

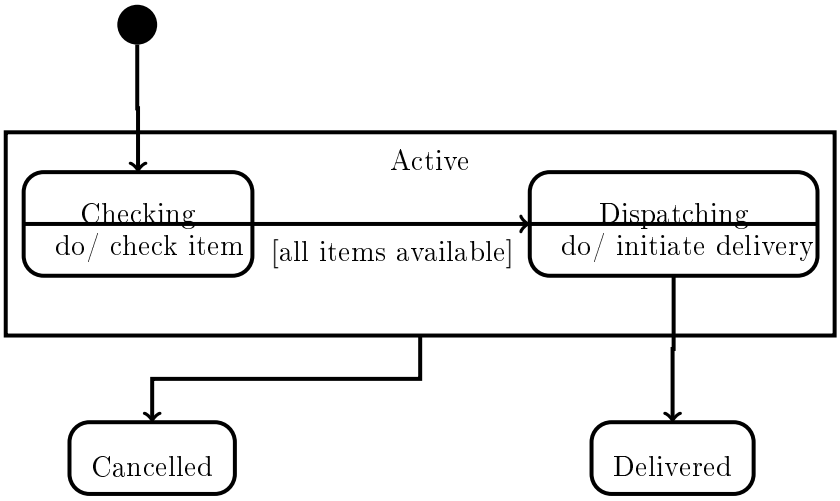
UML state machines should be designed to be deterministic: at most one transition should be activated at once.

Superstates. State diagrams may get too complicated to easily understand. UML state diagrams include the notion of hierarchical nesting to help you write clearer diagrams; such diagrams allow you to see the higher-level structure of the system.

In particular, UML state diagrams include the notion of a *superstate*. A superstate may contain

nested states and transitions, which make up a state machine for some subset of the entire machine's behaviour. The following example includes an example of a superstate, "active", which includes the "checking" and "dispatching" states inside it.

Note also that there are transitions from individual states inside the superstate, e.g. from "dispatching" to "delivered", as well as a transition from the superstate to the "cancelled" state. This transition means that either of the states in the superstate can get to "cancelled". It is equivalent to drawing transitions from both "dispatching" and "checking" to "cancelled".



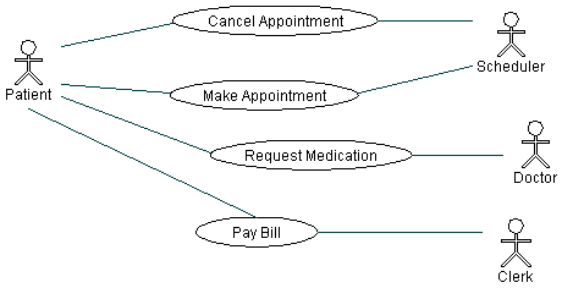
(From: `http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/state.htm`. Accessed July 3, 2011.)

Timing. Timing is important for embedded systems. Your documentation should explain what "quickly" (as in an action) means; UML doesn't say, and it depends on the system. You can also write an event that happens after some time, e.g. "after (20 minutes)".

UML Use Case Diagrams

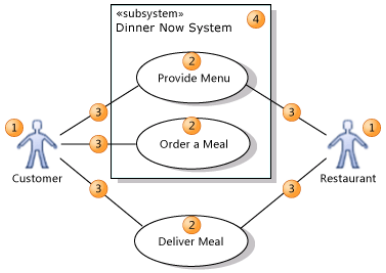
The next kind of UML diagram we'll talk about is the use case diagram. We'll talk about use cases shortly; a use case describes a user's interaction with the system. In these diagrams, users (or other systems) are known as *actors*, who play roles in the behaviour of the system. The following figure from the `embarcadero` site explains:

Here is a case with multiple actors:



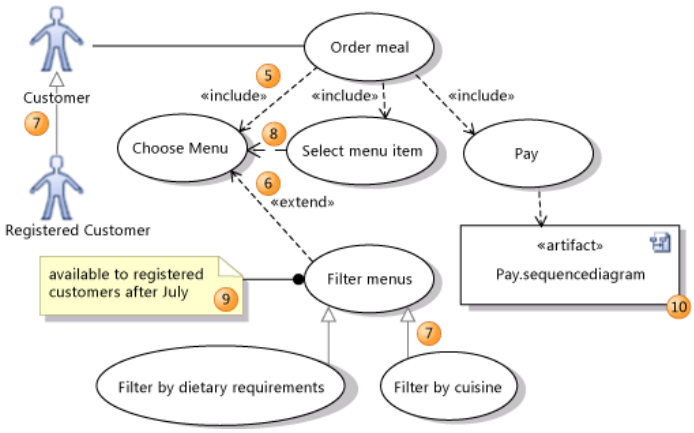
It's just a coincidence that each use case has two actors.

Finally, here's another use case diagram, from `http://msdn.microsoft.com/en-us/library/dd409427%28VS.100%29.aspx` (accessed March 10, 2011).



Again, we have 1) actors, 2) use cases, 3) communication, and, new to this figure, 4) subsystems or components.

The following diagram indicates relationships between use cases and actors.



Here we have 5) inclusion stereotypes on dependencies; 6) extension stereotypes on dependencies; 7) inheritance relationships between use cases; 8) plain dependencies; 9) comments; and 10) references to other artifacts.

- A *communication* (solid line) represents a relationship between an actor and a use case.
- An *inclusion* (dashed line, «include »stereotype) represents an invocation relationship between use cases; the first use case must call the second one.
- An *extension* (dashed line, «extend »stereotype) represents an optional invocation of a use case.
- A *generalization* or *specialization* represents a case where an actor or use case inherits from another actor or use case.

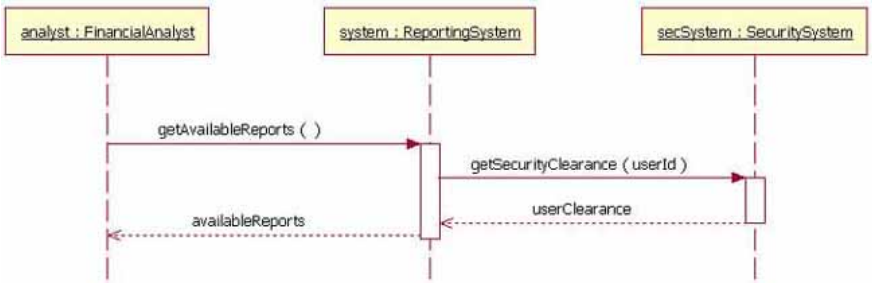
Why use UML Use Case Diagrams? These diagrams can be useful for 1) documenting essential features (requirements) of a software system; 2) communicating system behaviour to clients; 3) generating appropriate test cases for scenarios.

UML Sequence Diagrams

The last kind of UML diagram we'll talk about is the sequence diagram. UML sequence diagrams express system behaviour as a sequence of events and activities. (Message sequence charts do the same thing.) In a sequence diagram, you'll find objects, lifelines, messages, action boxes, and gates.

- *Instances* of objects are shown as boxes at the top of the diagram.
- *Lifelines* are shown as vertical dashed lines, extending downwards from instances of the objects, to denote the passing of time.
- *Messages*, drawn as horizontal lines, denote the communication of events between objects.
- *Action boxes*, drawn as boxes on top of lifelines, denote the occurrence of activities.
- *Gates*, shown as filled circles on the boundary of the diagram, denote the occurrence of an external event.

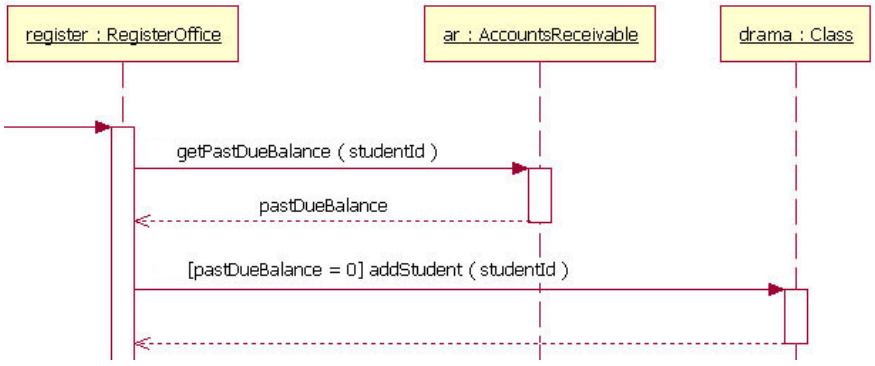
Example 1. Consider the following sequence diagram from a financial reporting system. (IBM, *UML basics: The sequence diagram*, `http://www.ibm.com/developerworks/rational/library/3101.html`, accessed March 10, 2011).



Solid lines are initiating messages, while dashed lines are responses. We can also see synchronous messages (solid arrowhead—all initiating messages in the above example) and asynchronous messages (stick arrowhead—responses).

There are more elements that you can put in UML sequence diagrams, including actors (indicating interactions between the system and its users); destroy elements (indicating the end of an object instance); scenario elements (denoting a set of alternative actions); and timer elements (for discussing timed events). You can find more information about those elements at `http://www.sequencediagrameditor.com/uml/sequence-diagram.htm`.

Example 2. Here is another sequence diagram, again from the *UML Basics* site.



We can note guards on the messages, which need to be satisfied before the message gets sent.

When to use Sequence Diagrams. UML sequence diagrams help document messages going between different instances of objects, including the ordering in which these events occur, if applicable (for synchronous events).

You need to exercise judgment to include the right level of detail; as in any model, too much detail makes the model difficult to understand, and too little detail doesn't document anything. You can use more than one UML sequence diagram to document a large system.