

UML and specifications

N. Botta

1 UML basics

1.1 Introduction

The Unified Modeling Language (UML) is a graphical notation and a hierarchy of models. In this note I discuss the UML graphical notation from the perspective of program specification. For a more comprehensive introduction to UML, see http://en.wikipedia.org/wiki/Unified_Modeling_Language and [1].

1.2 The UML graphical notation

The UML graphical notation consists of 13 *diagrams* grouped into two categories: structure and behavior:

- Structure diagrams
 - Class (classes, features, relations)
 - Component (self contained software entities, often w.r.t. upgrade ability, purchasability)
 - Composite structure (class decompositions)
 - Deployment (maps software -> hardware)
 - Object (run-time snapshots of instances)
 - Package (compile-time software setup)
- Behavior diagrams
 - Use case (interactions between system and user -> functional requirements)
 - Activity (flow-charts, concurrent behaviors)
 - Sequence (objects lifelines and interactions)
 - Interaction overview (“activities” with behaviors blown-up as sequences)
 - Communication (interaction graphs with time tags on the edges)
 - Timing (interaction graphs with timing constraints)
 - State machine (state transitions)

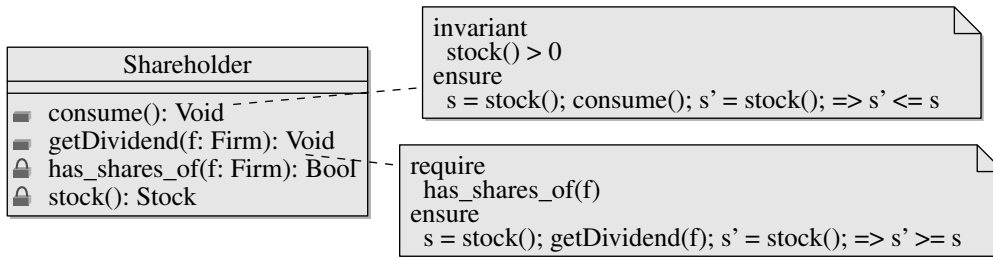


Figure 1: Shareholder class.

An example of a class diagram representing an idealized *shareholder* in the context of economic modeling is sketched in figure 1, see also section 2.

As the above list suggest, UML diagrams allow to describe a software from different viewpoints. Depending on the purposes of the description (see next section) different sets of diagrams might be appropriate. While some diagrams have been used since the first UML standard (UML 1, 1997), others have been introduced with UML 2.

1.3 What is UML used for ?

According to [1], most ways of using UML can be described as:

- Sketching
- Blueprinting
- Programming

Sketching is used both for software design and for model description. The difference between these two purposes is more a matter of intent and cannot be characterized by disjoint subsets of UML diagrams. However, sketches based on UML diagrams like *Package* or *Deployment* tend to be used more for software design or software documentation than for model description. The converse is true for *Use Cases*, particularly when expressed in narrative form.

UML blueprints are detailed software descriptions. They are developed by software designers and used by implementers. Being more precise than UML sketches and more abstract than implementations, they can be seen as *specifications*.

In programming, UML diagrams are used together with or within CASE (computer aided software engineering) tools, to generate executable programs, possibly via different programming languages and respective compilers. In this context, the term “executable UML” is often used. As in the case of blueprints, executable UML can be seen as a specification for the generated programs.

UML diagrams are at a higher level of abstraction than compilable or executable programs. From a software development perspective, they can be seen as *prescriptions* or specifications for these programs. This is what is usually referred to as the *forward* engineering usage of UML.

In this perspective, a time line goes from sketches to blueprints towards programs. However, UML diagrams are also used for program analysis, e.g. prior re-factoring. In this *backward* engineering perspective, the time line goes the other way round and UML diagrams play the role of *descriptions* of existing programs.

1.4 UML and software development

UML was developed after J. Rumbaugh and G. Booch from Rational (former Rational Environment) proposed a *Unified Method* to merge a number of graphical analysis and design methods developed to assist object-oriented programming in the eighties and early nineties.

These methods – mainly developed by practitioners – were similar in principle but different in notation and details. This fragmentation was considered at Rational as a major obstacle towards the commercial adoption of modeling tools.

The Unified Method was presented in 1995, the first UML draft was proposed by Rational (bought by IBM in 2002) in Jan. 1997, the first UML standard was released by OMG (Object Management Group, an open consortium of companies) in the same year.

While the Unified Method was a product of a single company, UML was the result of a negotiation process among the companies represented in OMG.

UML has been from the very beginning closely related with methods or styles of organizing software development: the way UML (and, in general, descriptions of software systems at a higher level of abstraction than programs) is used largely depends on the method adopted.

Methods of organizing software development are sometimes called development processes. Examples of development processes are “waterfall”, “iterative” (incremental, spiral, evolutionary), “agile”, “rational unified” and “extreme programming”.

While methods of organizing software development have been and are an issue in commercial software development, their role in the academic world has been less significant. This fact, together with the traditionally limited role played by *exploratory programming* in science, might explain the limited impact of UML on the academic world.

It is possible that, as the role of modeling and exploratory programming in science becomes more and more relevant, e.g., in climate research, socio-economic modeling, policy advice, UML or more formal model specification approaches will become more popular.

2 Example: a simple economic system

We consider a simple model of an economic system. The system is defined in terms of three entities or agents: a firm, a shareholder and a household. The agents repeatedly update their state according to agent-specific sequences of actions. We first use activity diagrams (the good old flow-charts !) to outline such sequences, see figures 2 to 4.

Activity diagrams provide useful hints for understanding the agents behavior from the functional viewpoint. However, they provide little information on interactions between agents.

It is therefore difficult to derive, on the sole basis of activity diagrams and in an object-oriented framework, a meaningfully class decomposition for the whole system.

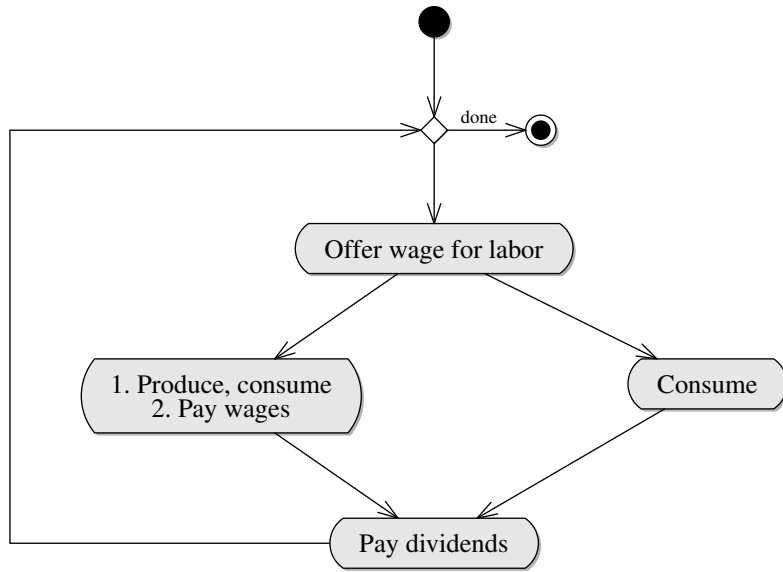


Figure 2: Firm activity diagram.

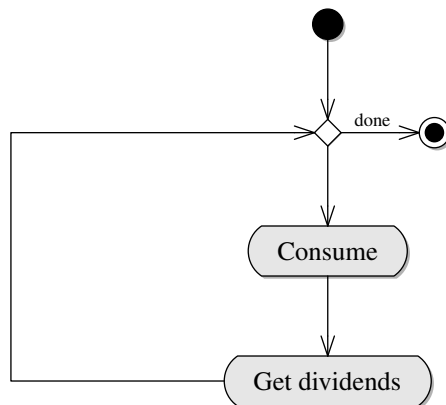


Figure 3: Shareholder activity diagram.

Some more information can be obtained by looking at agents interactions. These take place during well-defined actions and can be represented by use case diagrams. At the beginning of a time step, for instance, firms and households interact to trade labor for wages. In general, a household interacts with one or more firms, e.g., to determine the best wage it can get for its labor offer. Similarly, a firm might interact with a number of households. Thus, a necessary condition for a trade is that the system contains at least one firm and one household. This requirement can be expressed with in a use case diagram, with the strings 1..* on the left (for Household) and on the right (for Firm) of the link between Firm and Household, see figure 5. Similar diagrams can be used to describe the interaction between the firm and its shareholders.

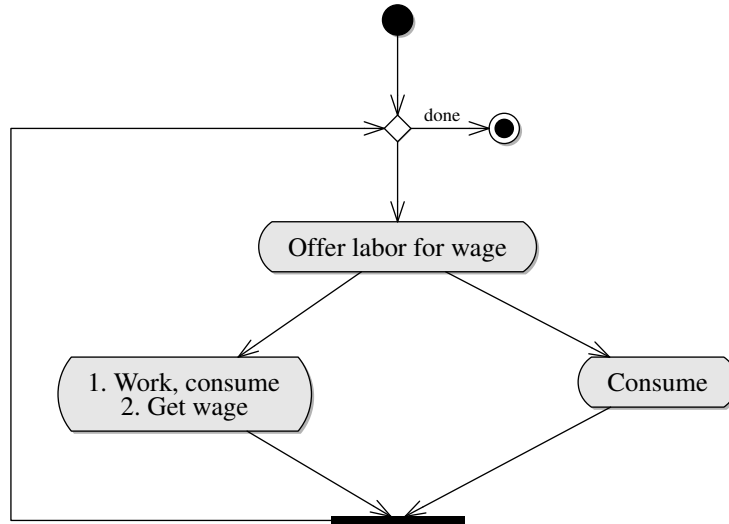


Figure 4: Household activity diagram.

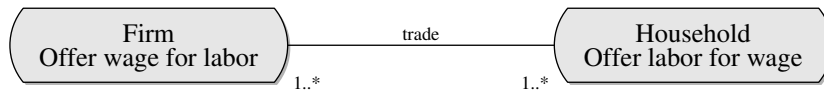


Figure 5: Use case diagram for the interaction between firms and households.

With some basic understanding about functionalities and interactions, one can start outlining class diagrams for the basic system components. A shareholder’s class diagram has been sketched in figure 1. Diagrams for firms and households are drawn in figure 6.

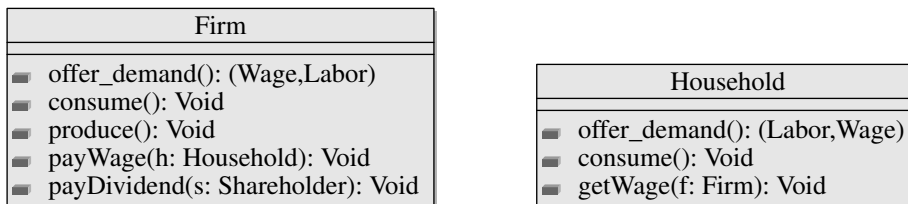


Figure 6: Class diagrams for firms (left) and for households (right).

The class diagrams of figure 6 are minimal descriptions and require further refinements. From the modeling perspective, two major steps are required:

- Specification of pre- and post-conditions.
- Specification of class relationships.

An example of pre- and post-condition specification for shareholders is given in 1: here the public interface of the Shareholder class (the set of functions and data with the “unlocked” symbol) has been annotated with *require*, *ensure* and *invariant* clauses. To express such constraints, the auxiliary functions `has_shares_of` and `stock` have been introduced. These functions are tagged with the “locked” symbol which means that they are not accessible from the outside.

The specification of class relationships can be deduced from relationships between notions in the application domain or induced by commonalities between classes. The function `consume`, for instance, appears with the same signature in Shareholder, Household and Firm. This suggests that these classes might have a common “Consumer” ancestor:

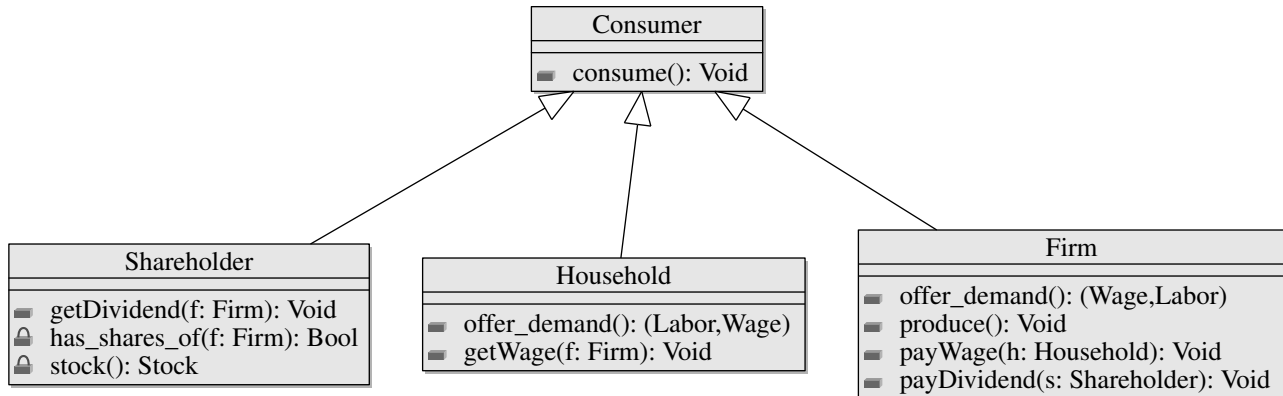


Figure 7: Shareholders, firms and households are all consumers!

To be prescriptive to the implementer, minimal class descriptions like those outlined in figure 6 have also to be refined from the implementational point of view. This means identifying data, often private data, and expressing complexity requirements.

We do not follow this thread further but it is clear that functions like `Shareholder.stock()` could easily be implemented if `Shareholder` would contain a (private) member of type `Stock` representing a shareholder’s stock.

By looking at the functionalities outlined in the activity diagrams through the use case lenses, it becomes clear that, for `Firm.offer_demand()` and `Household.offer_demand()`, `Firm.payWage(h: Household)` and `Household.getWage(f: Firm)` and `Firm.payDividend(s: Shareholder)` and `Shareholder.getDividend(f: Firm)` to represent interactions between agents, something more is needed.

What we need is to *model* such interactions. We can rely on some known communication and interaction patterns (synchronous, asynchronous) but it is obvious that understanding and specifying, e.g., how and under which conditions the offers and the demands of firms and households do match is a core modeling issue. UML supports the modeler with specific behavioral diagrams: state machine, timing, communication, interaction overview and sequence, see [1] for an introduction.

3 Remarks

Since it's first appearance in 1997, UML has been criticized from different viewpoints, see http://en.wikipedia.org/wiki/Unified_Modeling_Language. A frequent criticism has been that UML has played a much more significant role for consultants, research managers, decision makers and book writers than for software developers.

In the academic world this is certainly true not only for UML but also, possibly to a lesser extent, for object-oriented programming.

The relationships between UML and object-oriented programming have themselves been subject of technical debate. While UML has been proposed in the very beginning to support object-oriented software construction, its usefulness for this aim been authoritatively questioned, see <http://archive.eiffel.com/doc/manuals/technology/bmarticles/uml/page.html>.

After having read [1] and tried to apply UML to the economic system example discussed above, I can summarize my main criticism in two point:

- The aims of UML are vague.
- It is not clear how UML can contribute to improve program correctness. In particular:
 - UML does not provide expressive diagrams for describing programs which are conceived to solve problems where what is given and what is sought are explicitly stated.
 - UML does not provide expressive diagrams for specifying under which conditions programs are correct.

In spite of these criticisms, UML sketches like those presented in the example above can be very useful to convey an impression of what a software should do or, in a documentation or backward engineering perspective, of what a software does.

References

- [1] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, 3rd edition edition, 2003.