Lecture 21: Conceptual Models

The same object model notation that we've used to describe the structure of the heap in an executing program – what objects there are and how they are related by fields – can be used more abstractly, to describe the state space of a system or of the environment in which a system operates. I call these 'conceptual models'; in the course text, they're called 'data models'. You've already actually built some of these in Exercise 4, in the warmup examples, and when you used the object modelling notation to describe the structure of the Boston subway system.

The notation itself is very simple indeed, and models are easy to interpret once you loosen yourself from an implementation-oriented view, replacing Java objects by entities in the real world, fields by relations, and so on. After this lecture, you should have no trouble *reading* conceptual models.

Writing them, on the other hand, takes more practice. It involves making appropriate abstractions – just as you have to do when you design the interface of an abstract data type. Doing this well is hard, but the obstacle is nothing to do with object models in particular. It always difficult to get to the essence of a problem and articulate it succinctly.

Once you've overcome this obstacle, and constructed a conceptual model, you're half way to a solution of your problem. It's often been said that if you can say exactly what your problem is, then you've made progress towards solving it. In software development, you're more than half way there.

So don't expect to be able to build conceptual models without some practice. It's a lot of fun, though, and as you hone your modelling skills, you'll find that you become a better designer. As your conceptual structures gain clarity, the structures in your code will become simpler and cleaner too, and coding will be more productive.

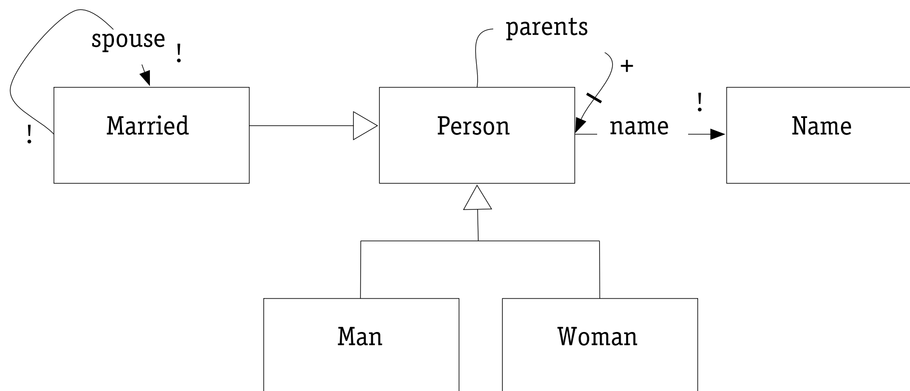
In the lecture itself, I'll try and give some sense of how models are constructed incrementally. In these notes, the models are shown in their final form.

21.1 Atoms, Sets and Relations

The structures of our models will be built from sets, relations and atoms. An atom is a primitive entity that is

- *indivisible*: it can't be broken down into smaller parts;
- *immutable*: its properties don't change over time; and
- *uninterpreted*: it doesn't have any built-in properties, the way numbers do, for example.

Elementary particles aside, very few things in the real world are atomic. But that won't stop us from modelling them as atomic. In fact, our modelling approach has no built-in



notion of composites at all. To model a part *x* that consists of parts *y* and *z*, we'll treat *x*, along with *y* and *z*, as atomic, and represent the containment by an explicit relation between them.

A set is just a collection of atoms, with no notion of repetition count or order. A relation is a structure that relates atoms. Mathematically, it's a set of pairs, each pair consisting of two atoms, in a specified order. You can think of a relation as a table with two columns, in which each entry is an atom. The order in which the columns appear is important, but the order of the rows is irrelevant. Each row must have an entry in every column.

21.2 Graphical Notation

There's no need to go through all the details of the graphical notation again; you've seen it before in the lecture on object models. All we need to do here is reinterpret the notation more abstractly, and add a few refinements we haven't needed before:

- the distinction between domains (top-level sets) and other sets;
- the idea of sets whose contents can change over time;
- marking subsets of a set as disjoint and exhaustive;
- ways to express ternary relations.

Take a look at the object model for the family tree. Each box denotes a set of atoms – not a set of objects in a Java program, or a class! The set `Married`, for example, represents the set of all persons who are married. (Of course, it's not clear what this means: the persons legally married? married now, or married at some time during their lifetime? As we'll say later, when you build a conceptual model, you need to define your terms carefully.)

Each (open headed) arrow denotes a relation from one set to another. It denotes an abstract association, not a field or an instance variable. So `parents` for example is a relation from a person to his or her parents. You can think of it as a predicate: `parents(p,q)` is true if `q` is a parent of `p`.

The direction of the arrow has semantic consequence of course: it makes a big difference whether `p` is the parent of `q` or vice versa. But for any relation, we could equal well use a different relation that is the transpose; `children` instead of `parents` for example. There's no notion of navigability, or a relation belonging to a set in the way an instance variable belongs to a class.

The fat, closed-headed arrow denotes subset. Two sets that share an arrow are disjoint. We can fill in the arrow head to say that the subsets are also exhaustive: that every member of the superset is a member of at least one of the subsets. In this example, we've said that every `Person` is a `Man` or a `Woman`, and every `Married` person is a `Person`.

The sets that have no supersets are called *domains*. They're assumed to be disjoint. No atom is both a person and a name, for example.

We won't review the multiplicity and mutability markings here; they're explained nicely in the course text.

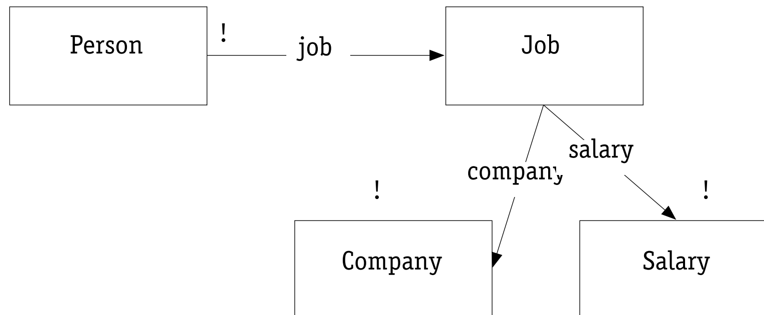
Our graphical notation has rather limited expressive power. You can record additional constraints textually. For example, we might want to say that a person and his or her spouse don't share the same parents. Sometimes you can express more in the diagram by introducing additional relations. For example, to say that a person has two parents, one a man and one a woman, you could add relations `mother` and `father`, marking them with the appropriate multiplicity.

What constraints actually hold is often a subtle and important problem. First there's the question of whether the constraint actually holds. Does a person have two parents? That depends on what exactly is meant by parent. If it's not the biological notion, things get very complicated. Second, there's the question of whether you're modelling something that's true in the real world, or instead are describing a representation of the real world to be constructed inside a computer, which may differ from the real world because of input errors, incremental construction, lack of information, and so on. For example, if you're building a genealogical database, it's probably not a good idea even to assume that every person has one biological mother. Otherwise, your data structures won't be able to accommodate persons of unknown parentage.

21.3 Ternary Relations

Sometimes we want to describe relationships that involve three, not two, sets. For example, we might want to record the fact that a person earns a salary working for a company. If persons can work for several companies, and earn a different salary at each, we can't just associate the salary with the person.

Often, the easiest way out of this is to create a new domain. Here, we could introduce *Job*, and draw an object model showing a relation *jobs* from *Person* to *Job*, a relation *salary* from *Job* to *Salary*, and a relation *company* from *Job* to *Company*.



This approach works well when the domain you introduce already corresponds to some natural set of atoms – it's a notion already understood in the problem domain.

Alternatively, you can introduce an *indexed relation*. If you mark the arrow from *A* to *B* with the label *r[Index]*, this means that for each atom *i* in the set *Index*, there is a relation *r[i]* from *A* to *B*. For example, to model naming in a file system, we might have an indexed relation *obj[Dir]* from *Name* to *FileSystemObject*, since there is conceptually a separate naming relation for each directory of the file system. Or, less naturally, in the case of our employment database, we might have a relation salary from Person to Salary indexed on Company.

Finally, you can draw the object model and say that it's a projection: that it shows the relationships for a particular atom in some domain. For example, in designing a word processor, there might be a ternary relationship *format* that associates a *Style* with a formatting *Rule* in a given *Stylesheet*. We may want to draw a model that considers only a single stylesheet, so that the relation becomes binary.

21.4 General Strategy

Here's a way to go about developing a conceptual model in easy steps:

- *Domains*. What is the problem about? Write a list of the domains with short but careful definitions.
- *Sets*. Think about how these domains are classified into sets. Write definitions for the sets, and construct the classification hierarchy, showing which sets are subsets of which other sets and domains.

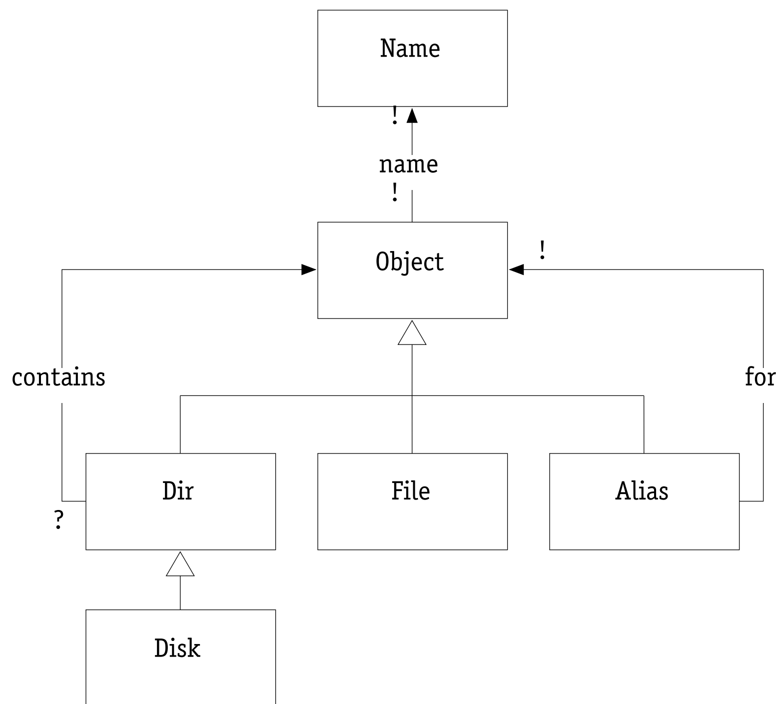
- *Relations*. Think about how domains can be related to each other. Write short and careful definitions of relations, and place them on the set hierarchy as low as possible (that is, using the set classification to constrain the domains and ranges of the relations. Then figure out the multiplicity and mutability properties of the relations.
- *Constraints*. Ask what constraints are not captured by the diagram, and record them textually. Think about relationships between relations: what happens if you follow one relation then another? For a relation that maps a set to itself, what happens if you follow it repeatedly? Is the set mapped by one relation determined by a property of another? Does a relation over a set map one of its subsets in a distinct way (to a subset of the range for example)? You may find it useful to introduce new sets or even new relations to make the structural properties of the model more evident in the diagram.

These are just rough guidelines. In practice, of course, you work iteratively: you build a small model, evaluate it, and then rework it, simplify it or expand it, and iterate again until you've landed on something you're happy with.

How do you know you're making progress? If you're going about conceptual modelling in the right way you'll find that you keep hitting conceptual stumbling blocks: things you thought were obvious turn out not to be after all. But by thinking through the questions that come up, and recording them in a succinct and precise model, you should feel that you're cutting your way through the undergrowth, slashing away the confusion. The questions that came up and how you answered them will only be partly evident in your final model, so you should keep a record of them, particularly of ideas that turn out *not* to be useful.

How do you know when you're done? When the main concepts that you're trying to elucidate are represented in a succint and clear way. You should *not* try to be exhaustive: modelling every even peripherally relevant concept you can think of is tiresome and not a good use of your time.

As you work on your final project, and in any subsequent developments you do, you should construct conceptual models as you need them. Don't feel a need to have a single, all inclusive model; sketch a variety of smaller ones, and then consider which of them may need to be integrated. Until you have some experience with conceptual modelling, you'll probably think some conceptual notion is obvious, and won't discover that it isn't until your deep into the code. So try and play around with more models that you think you need at first. And if you strike complexity while you're coding, back off, and sketch some models.



21.5 Three Examples

Let's look at three examples of conceptual models. They are all simple, but they are not trivial. They demonstrate, I hope, that constructing even very small models raises tricky questions, and is thus useful.

21.5.1 File System

Let's consider for our first example something we're very familiar with: filesystems. We'll model the most basic relationships between the elements of a file system.

For domains, we'll take:
- Object: the set of all objects that can reside ina file system
- Name: the set of all full names that objects can have.

The sets are:
- Dir, the set of directories, including the root of a file system
- File, set of all files
- Alias, the set of aliases, or shortcuts (in Windows), or soft links (in Unix)
- Disk, the set of root directories that represent different disks that can be mounted.

The relations are:
- contains, which maps a directory to the set of objects immediately contained in it
- for, which maps an alias to the object it represents.

The diagram below shows the object model. The multiplicity constraints are important: an alias must refer to exactly one object; every object is contained by at most one directory; names are unique. The mutability markings capture the facts that an object can be moved from one directory to another, but an alias is created pointing to a given object, and that relationship cannot be changed.

What textual constraints might we add? First consider the `contains` relation. We should say that the directory hierarchy under a disk forms a tree: disks aren't contained in directories, every non-disk object is contained by one directory, and no directory contains itself, directly or indirectly. Then consider the `for` relation. Can an alias be for itself? (Probably yes). Can an alias be for an object under a different disk? (Probably no).

File system structure is a tricky business. Try and build a model of the Macintosh trash (or Windows recycle bin) and you'll discover that you probably can't even predict how you're own file system behaves.

21.5.2 Style Sheet

Almost all text layout programs are based on the idea (developed at Xerox PARC) of the *stylesheet*, in which paragraphs are tagged with names of styles, and a separate style sheet assigns formatting rules to the styles. This follows the general principle of computer science that 'there is no problem that cannot be solved by introducing another level of indirection', in this case making a map from paragraphs to formats two maps, one to styles, then one from styles to formats. The advantage, of course, is that you can then make wholesale changes to a document by modifying the style sheet, and every paragraph of a given style will be reformatted when the format associated wit that style is changed.

For domains, we might have:

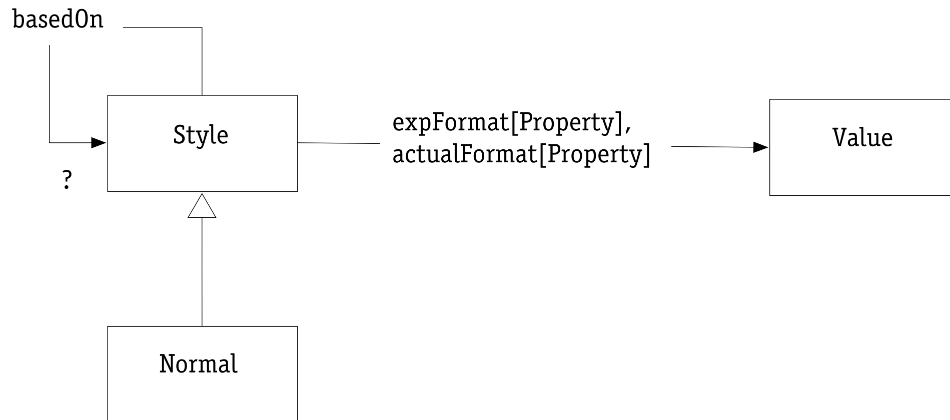
- `Style`: the set of all possible styles, viewed as just names distinct from their formatting rules
- `Property`: the set of formatting properties that can be given values in a stylesheet
- `Value`: the set of values a property can take on, such as font sizes, indentations, alignments, etc.

For a set, we might have:

- `Normal`: a built-in style assigned to paragraphs by default.

For relations, we might have:

- `basedOn`, which associates a style with a style from which it inherits formatting features (`noindent`, for example, may be based on `normal`, with one additional rule saying that the first line indent is zero)
- `expFormat`, which maps a style and a property to a value, indicating the value of the formatting property that is defined explicitly for a style by the user



- actualFormat, which maps a style and a property to a value, indicating the value of the formatting property that is actually associated with a style, due to a combination of inherited and explicit declarations.

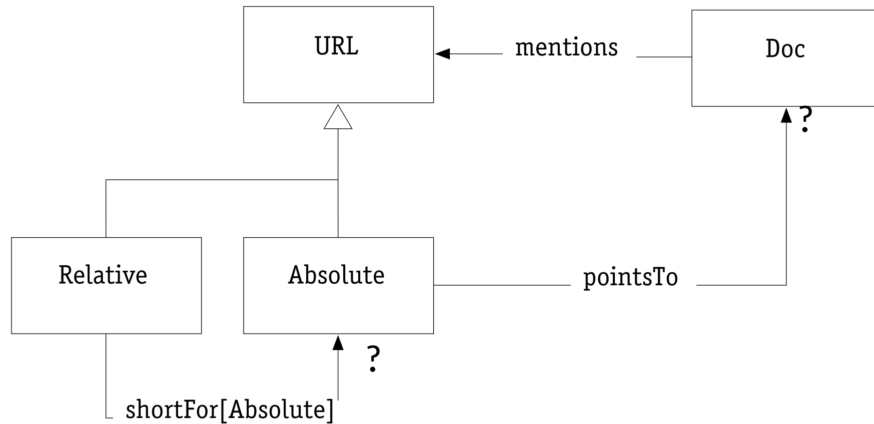
Constraints that aren't expressed in the diagram that we might want to record are:

- The graph of styles defined by the basedOn relation forms a tree rooted at Normal.
- For a given style and property, the actual format is the actual format of the style on which it is based, overridden by any format declared explicitly.
- The Normal style's explicit format gives a value to every property, and its actual format is the same as its explicit format. Cosnequently, every property has a value for every style.

Style sheets are actually rather complicated and interesting. Two features that this model doesn't address are: toggle properties (such as italic/roman), and the ability to maintain multiple style sheets for a single document. There are also some tricky questions about how formatting rules are entered. Suppose, for example, we declare noindent to be based on normal, and set its indentation to zero. If we then change the indentation of normal (increasing it, say), we'd expect the indentation of noindent to remain unchanged. But in many word processors, if normal started out with zero indentation, setting noindent to have no indentation would not in fact create an explicit formatting rule at all. In some applications (such as Framemaker), you can indicate whether a style redefines a property of its parent, even if the properties are the same.

21.5.3 WWW Structure

The next example illustrates how an incredibly small and focused model can raise tricky questions. Consider the basic structure of the web: that there are URL's, and documents. A URL pointsTo a document; a document itself contains URL's to other documents. Now let's consider relative links. We can model these too, by an abuse of terminology, as URL's, which we now classify as either Absolute or Relative. The interpretation of a



relative link can be represented by a ternary relation shortFor which takes a relative link and an absolute URL, and yields an absolute URL. The object model is shown below.

The interesting question now is how the absolute URL is obtained as the context for interpreting the relative URL's of a document. Note that the pointsTo relation may map several URL's to the same document: a document can have aliases because of multiple domain names mapping to the same machine, multiple paths mapping to the same file in a filesystem, or because of explicit rewrites performed by the web server. So we can't speak of *the* URL of a document. How then is this absolute URL obtained?

Usually, since a document's source is first interpreted by the browser before any links are followed, the browser itself uses the URL by which the document was obtained for the context. This does mean that a relative URL in a document may be interpreted differently, depending on how the document was obtained. This also explains, incidentally, a problem with many browsers: when a page is saved, not enough context is saved to resolve all the URL's in it.

21.6 Conclusion

The graphical notation is described in more detail in the course text by Liskov. You may also find helpful previous years' 6170 lecture notes, which have more examples of conceptual models.