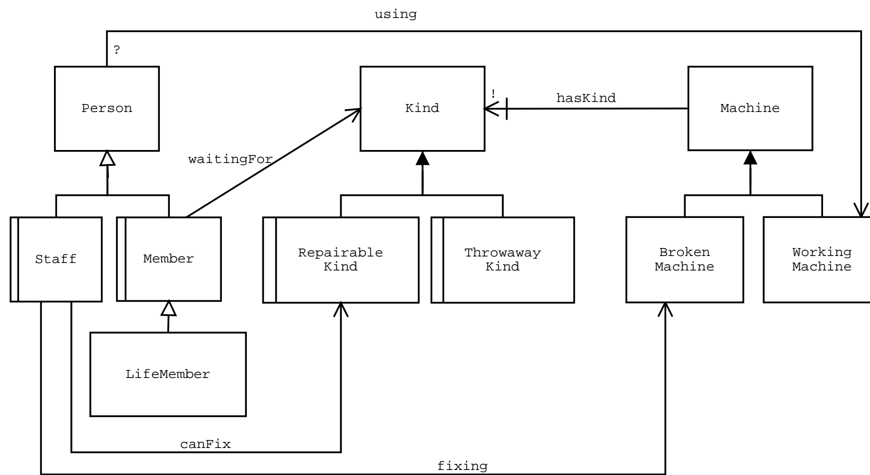
Top Down Development of Code Object Models

October 15, 2000

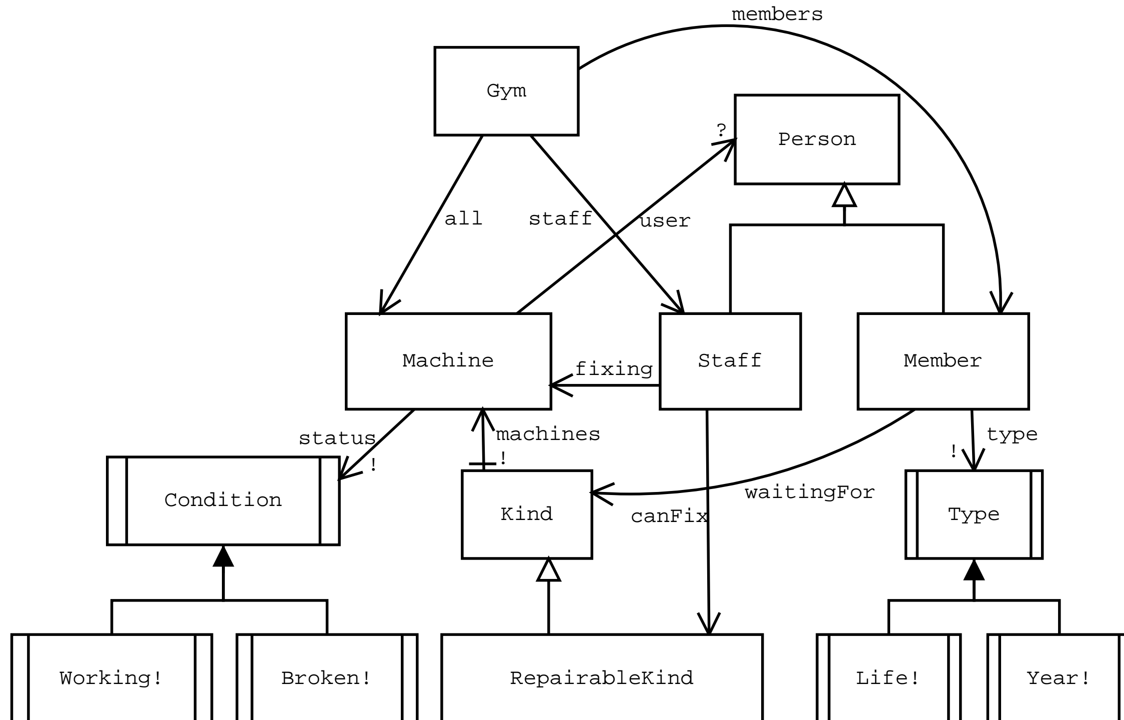


Problem Object Model

Introduction The topic of Recitation 5 was medium scale design. Specifically, the goal was to practice taking the results of a high-level problem analysis and mapping them to the actual data structures that would be used in an implementation. Above is the Problem Object Model (also called a Conceptual Object Model or Analysis Object Model and many variations thereof) that some third party developed for the system we explored in Recitation 4. Now the question is how to begin designing the modules that will represent the data that we need to perform the operations in our equipment tracking system.

One important point of this exercise is that even though we are talking about the code now (and not just the abstract entities that occur naturally in the problem itself, as above), we still do not want to get our hands dirty in the specifying the implementations that our objects will have. It is better to delay that as long as possible, since it will give us maximal flexibility and freedom as we develop (and redevelop) our design. Thus, we will talk about designing classes and giving them fields (since we do want to know which objects will store information about their relationships to other objects in the system), but the fields we are discussing are not the concrete fields defined in the implementation, but rather fields in the specification. We have been touching on this issue throughout the course, when we have been stressing that fields given in the specification (referred to as abstract-fields or spec-fields) do not have to (and in some cases cannot)

map directly to fields in the implementation of the class itself. I will point out specific cases of this in the proposed Code Object Models as I explain them below.



Code Object Model Attempt 1 Above is our first attempt to develop a code object model for our system. Notice that we have **not** extended the `Machine` class with `WorkingMachine` and `BrokenMachine`. This would be a terrible mistake. The whole point of partitioning the *Machine* domain into the *Working* and *Broken* partitions was to reflect the fact that a machine may break over its lifetime, but an object cannot change the Class it belongs to over the course of its lifetime. When an object is constructed, it is part of a Class, and it stays in that Class forever. Therefore, to properly represent dynamic properties like Working/Broken, we should not use subclassing, but instead track that information in some other way.

Notice also that this code object model introduces a `Gym` class; we did not have a *Gym* domain in the Problem Object Model, because there was not any interesting relationships between a Gym and the other parts of the system given above. However, a `Gym` class **is** a useful notion here because it is useful to store certain information in one place (such as the staff members who are currently on-duty) so that we can retrieve it easily when needed.

So already we have identified two large differences between the Problem Object Model and the Code Object Model. The Problem Object Model gave us a lot of freedom (in terms of partitioning domains into sets without concerning ourselves with issues of subclassing/subtyping, etc) which is very useful when trying to analyze, understand, and document a problem. But the Code Object Model is concerned with how the modules in the implementation are going to represent state, which means that we should keep in mind what ways our programming language will limit us when we do implement the classes.

Notice also that we have created the Domains `Condition` (and `Type`, to which this discussion applies, indirectly). The `status` field could be reflected in the specification of our Machine class as follows:

```
/** A Machine represents a mutable piece of equipment in the gym.
    @specfield status : Condition // Working or Broken
    @endspec
*/
```

However, just because the code object model has created this abstract domain and placed it in the specification does not mean that we actually **need** a `Condition` class in our system. Notice that `Condition` in the code object model does not have any interesting state; it just defines a static set of values, all of which (`Working`, `Broken`) are enumerated in the model. Therefore, in the implementation of the `Machine` class we could represent the state as just a boolean field, and the abstraction function would relate the value of that field to the actual condition of the machine.[1] In other words, the code for `Machine` might appear as follows:

```
public class Machine {
    // AF: status = if this.broken then Broken else Working
    private boolean broken;
    ...
}
```

Likewise, some of the relationships will need a little more fleshing out in the implementation. Every relation that does not have a `!` or a `?` on its target end represents the possibility of mapping to more than one object; however Java fields can reference at most one object at a time. So, fields without `!` or `?` need to have some intermediate structure that can hold multiple references at once. I will use the example of the `staff` relation from `Gym` to illustrate this.

```
/** A Gym is a central repository of information about equipment usage.
    @specfield staff : Collection of Staff
    ...
    @endspec
*/
public class Gym {
    // AbsFun: staff = this.staffLst
    //      ...
    // RepInv: staffLst != null && forall s in staffLst.elts | s is-a Staff &&
    //      ...
    private List staffLst;
    ...
}
```

So, now that we have a rough idea of how one might derive a specification from a code object model (and thus get a rough idea of how state will be divided up in the system), lets try to judge how good an implementation based on this code object model would be.

There are a number of interesting aspects to this model. For example, a Kind has a direct link to the machines that it is a kind for. This seems very useful at first: after all, when someone requests to use a machine of a certain kind `k`, you just iterate over `k.machines` and find a machine whose `user` is not set.

[1] Obviously if there were more than two values enumerated in the `Condition` domain then we could not use a single boolean to represent the information, but instead could use an `int` or some other way of enumerating the possibilities. The `Type` domain is actually a place where this could be applicable, if the gym decides to add other membership types in the future.

Sounds like a great design, right? Looks fast, easy to understand, etc. Unfortunately, such a naive approach to organizing your system state is only going to get you into trouble.

Problem 1 with `Kind`. Storing the information relating *Machine* and *Kind* in the `Kind` class is a very error-prone approach. Notice that the `machines` relation has a `!` and a static-marking on the source end. These are both strong constraints to put on the source-end of any relation in the code object model. The `!` says that you can never have two `Kinds` have their `machines` point to the same `Machine` (a constraint which is difficult to enforce in the code). It also means that every `Machine` has a `Kind`; if you construct one and forget to add it to the `machines` field for some `Kind`, you will be violating the constraints given here. The static-marking on the source-end means that every instance of a `Machine` can only be pointed to by the same `Kind` throughout its lifetime.

In practice, you can write code that will update the data-structures appropriately in the `Machine` constructor; it could look like this:

```
public Machine(String id, Kind myKind) {
    this.id = id;
    myKind.addMachine(this);
}
```

This will indeed set up a mapping from `myKind` to the constructed machine, presuming that `Kind.addMachine` is defined appropriately. If this is the only externally visible constructor, the design of `Machine` will force client code to pass some sort of `Kind` in at construction time, which allows us to preserve the fact that every `Machine` will have a `Kind` at the start of its lifetime. The problem here, however, is that you have forced the `Kind` class to provide some method of adding machines (or expose its representation to allow the `Machine` to update the mapping itself). No code except the `Machine` constructor should **ever** call this method, and yet it is exposed to classes other than `Machine` and `Kind`, polluting the `Kind` interface and opening the chance for other code to add arbitrary `Machines` to the `Kind.machines` mapping. Not only that, but since the information about the `Kind` a machine has is held outside of the machine object, it is impossible to reason about the code of `Machine` and `Kind` being able to preserve the constraints on the source of the `machine` relation. The system as a whole **may** preserve the constraints, but convincing ourselves (and anybody else) of this fact will require reasoning about the code as a whole. This is an invitation for disaster.[2]

Problem 2 with `Kind`. While it seems very convenient to be able to just take a `Kind` object and immediately have access to all `Machine` objects that have that `Kind`, there is a subtle issue here: where are you going to get that `Kind` object? When a user comes up and requests to use a Stair-Master, you cannot say

```
Kind k = new Kind(``Stair-Master'');
```

[2]This is not a statement that you should never put static-marks or `!` on the source ends of your relations. There are situations where you can reason about a single class preserving such an invariant, or where you may want to document that such an invariant holds even if you cannot prove that you preserve it.

For example, `Vectors` maintain an internal array of objects. Since that array is part of the `Vector`'s internal representation, it should never be pointed to by any other object in the system (otherwise the `Vector`'s rep would be exposed). This is a case where the static-mark on the source end is **very** informative.

and get an instance of a `Kind` that will have a correct `machines` field.[3] Instead, since `Kind` is a *mutable* type in this design, we need to get at the exact instance of a `Kind` that is **`the`** stair-master-kind, and then ask that particular instance what `Machines` it knows about.

This is a classic problem with mutable types; since they are storing important dynamic information, you need to be able to find the particular instance of Kind to resolve queries involving the `machines` relation. To get at the needed `Kind`, your code is going to have some way of retrieving `Kinds`. Perhaps it will take the form of a mapping from `String` to `Kind` in the `Gym`, or perhaps it will just be a `List` of `Kind` in the `Gym`. Whatever form it takes, it is a necessary piece of state in the system that this model has forgotten to include, because it forgot about the need to resolve mutable types to particular instance values.

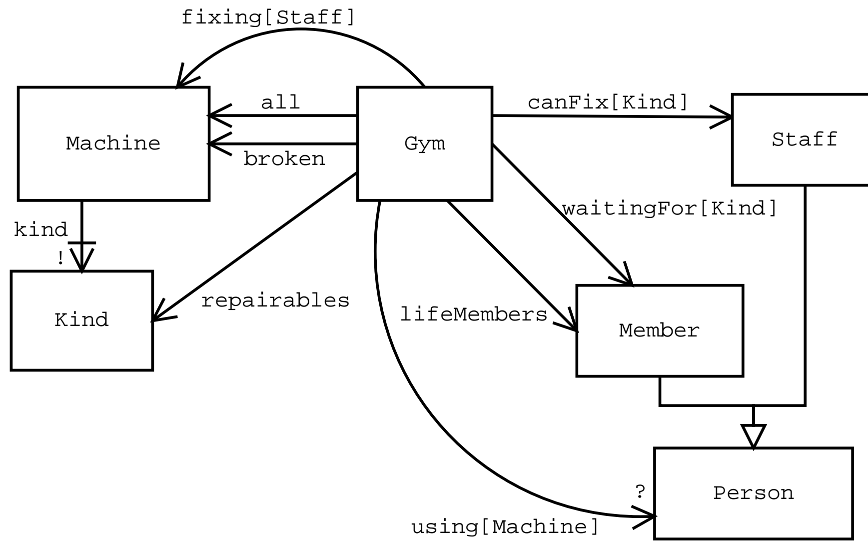
In general, we advise you to use *immutable* types where possible. They are easier to reason about, and since they do not track dynamic state, your code should be able to reconstruct them as necessary, rather than having to lookup the old ones to find needed information.

`Kind` is not the only class with problems in this version of the system. For example, `Member` stores the kinds that it is waiting for, but when a message that a machine of some kind has become available for someone else to use, there is no direct way to map that information back to a collection of `Members` who are waiting for that `Kind`. Instead, you need to iterate through all m in `gym.members` and ask each m, does m.`waitingFor` contain `kind`? This scales terribly as the `gym.members` grows, and the code looks ugly too.

Plus, as a side note, this object model looks terribly complex. Just the fact that there are two crossed lines[4] is a good hint that this object model might suffer from needless complexity. Excessive complexity is bad for this form of documentation: the whole idea of object models is to succinctly describe properties of the system.

[3] At least, not without putting hidden class-wide (called static in Java, but I'm attempting to avoid that definition of the word to avoid confusion) information in the `Kind` class and sharing the data in subtle ways between the various `Kind` instances. There are techniques that some will use to make the above code work, but since it tends to be bad style, I will not elaborate further on how to do it.

[4] And it is not trivial to uncross them, and even if you do, you just end up with many edges traversing the perimeter of the picture, which can be even harder to interpret than crossed lines.



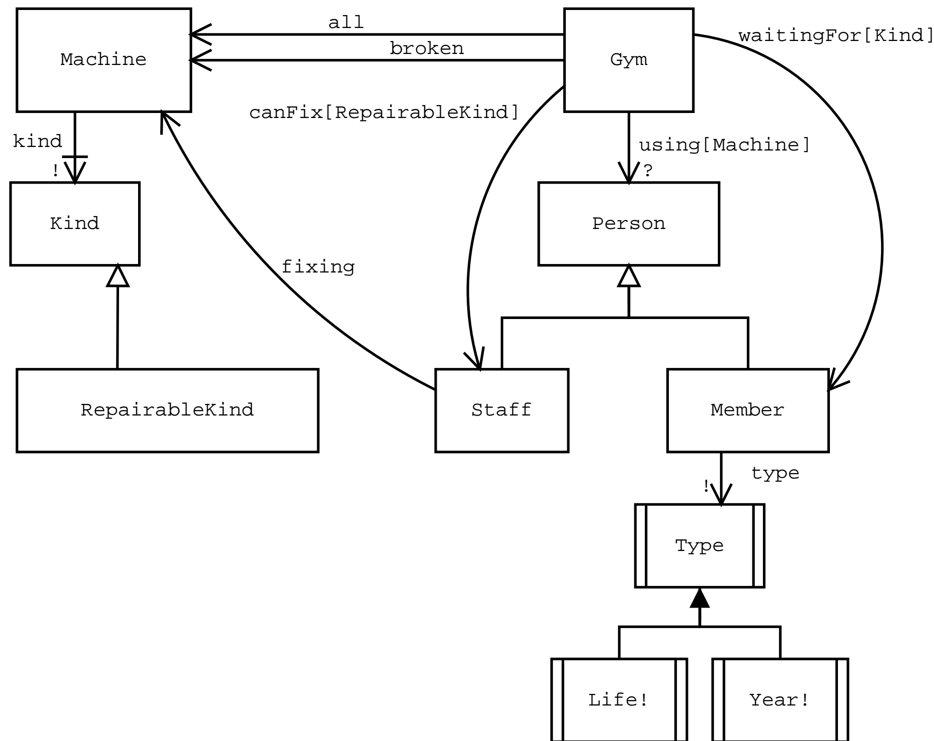
Code Object Model Attempt 2: The object model above attempts to correct the problems in our first attempt. In this scenario, `Gym` maintains a number of relations (some indexed), so that it can easily do what was problematic before. For example, when a `Machine` m becomes available for use, we can find who to notify by looking at the set: `gym.waitingFor[`m`.kind]`, eliminating the need for the `members` relationship from before.

Indexed relations can be implemented in a variety of ways. In this case, we would probably keep a `HashMap` in the `Gym` class mapping a `Kind` k to the `Members` who are waiting for k. Thus, assuming a well-written `hashCode` method in the `Kind` class, queries will resolve much more quickly than our prior approach.

Likewise, we've removed the need for most of our objects to be mutable in this design. For example, `Machines` no longer track within themselves whether they are broken or not; instead the `Gym` class maintains `broken`, which is the set of broken machines. `Machine` still has some state (since it still has a `kind` field) but that is a property which is static on the target end, and so `Machine` is a perfect candidate for immutability.

However, this design has one fairly serious flaw; nearly **all** of the interesting information is being maintained in a single place: the `Gym` class. The `Gym` class is likely to be maintaining several different data structures, which means that it is going to be a large, complex class. Having a single class maintain multiple related data structures can be useful. The `Gym` class probably has a fairly clean manner of reacting to a breakdown, first checking if the machine's kind is in `repairables`, dispatching `Staff` who can fix it, and then adding that particular `Staff` to its `fixing` relation. The main objection is that this is a case of over-zealous combination; if you put **all** of the data-structures that interoperate in one place, then you are guaranteed to have all the ones that interoperate in one place, *along with the ones that don't interoperate.*

Also, centralizing all of the functionality in one class limits reusability of the pieces of the system. If some of the relationships in the problem change, we're going to have to sort through all of the code in `Gym`; if that functionality were distributed across the classes in the system, we might be able to reuse the pre-existing implementations more easily.



Code Object Model Attempt 3: Above is a final attempt at a code object model which takes the good features of the other attempts while eliminating the bad. Some objects are candidates for immutability, others are not. Sometimes relationships are expressed directly between two objects, sometimes it is maintained in a seperate class. This is probably **not** the best possible object model, but many times in software design (and in other places[5]), the "best possible answer" simply costs too much to produce. One of the hardest lessons to learn in engineering is how to know when to stop looking for a better answer and to just accept the best one so far as "good enough." The hope here is that this object model modularizes the code effectively, provides a nice setup for good class design and specification, and allows for acceptable performance in the implementation itself.

However, you should review it with the same scrutiny that we had when reviewing the models above. For example, encoding the information about which `Kinds` are repairable with a subclassing relationship to `RepairableKind` has number of potential problems. I encourage you to

1. Identify what they are
2. Ask what representing partitionings this way lets us do that other approaches would not allow.
3. Propose a different handling of the *Throwaway/Repairable* paritioning that has not appeared in any of these three Object Models.

[5]other places far far away from M.I.T.