Why OO Sucks

When I was first introduced to the idea of OOP I was skeptical but didn't know why - it just felt "wrong". After its introduction OOP became very popular (I will explain why later) and criticising OOP was rather like "swearing in church". OOness became something tha every respectable language just had to have.

As Erlang became popular we were often asked "Is Erlang OO" - well of course the true answer was "No of course not" - but we didn't to say this out loud - so we invented a serious of ingenious ways of answering the question that were designed to give the impression that Erlang was (sort of) OO (If you waved your hands a lot) but not really (If you listened to what we actually said, and read the small print carefully).

At this point I am reminded of the keynote speech of the then boss of IBM in France who addressed the audience at the 7th IEEE Logic programming conference in Paris. IBM prolog had added a lot of OO extensions, when asked why he replied: Our customers wanted OO prolog so we made OO prolog

I remember thinking "how simple, no qualms of conscience, no soul-searching, no asking "Is this the right thing to do" ...

Why OO sucks

My principle objection to OOP goes back to the basic ideas involved, will outline some of these ideas and my objections to them.

Objection 1 - Data structure and functions should not be boun together

Objects bind functions and data structures together in indivisible units. I think this is a fundamental error since functions and data structures belong in totally different worlds. Why is this?

- Functions do things. They have inputs and outputs. The inputs and outputs are data structures, which get changed by the functions. In most languages functions are built from sequences of imperatives: "Do this and then that ..." to understand function you have to understand the order in which things get done (In lazy FPLs and logical languages this restriction is relaxed).
- Data structures just are. They don't do anything. They are intrinsically declarative. "Understanding" a data structure is a lo easier than "understanding" a function.

Functions are understood as black boxes that transform inputs to outputs. If I understand the input and the output then I have understood the function. This does not mean to say that I could have written the function.

Functions are usually "understood" by observing that they are the things in a computational system whose job is to transfer data structures of type T1 into data structure of type T2.

Since functions and data structures are completely different types of animal it is fundamentally incorrect to lock them up i the same cage.

Objection 2 - Everything has to be an object.

Consider "time". In an OO language "time" has to be an object. But ir a non OO language a "time" is a instance of a data type. For example in Erlang there are lots of different varieties of time, these can be clearly and unambiguously specified using type declarations, as follows:

```
-deftype day() = 1..31.
-deftype month() = 1..12.
-deftype year() = int().
-deftype hour() = 1..24.
-deftype minute() = 1..60.
-deftype second() = 1..60.
-deftype abstime() = {abstime, year(), month(), day(), hour(), min(), sec()}
-deftype hms() = {hms, hour(), min(), sec()}.
...
```

Note that these definitions do not belong to any particular object. they are ubiquitous and data structures representing times can be manipulated by any function in the system.

There are no associated methods.

Objection 3 - In an OOPL data type definitions are spread out all over the place.

In an OOPL data type definitions belong to objects. So I can't find all the data type definition in one place. In Erlang or C I can define all my data types in a single include file or data dictionary. In an OOPL I can't - the data type definitions are spread out all over the place.

Let me give an example of this. Suppose I want to define a ubiquitous data structure. ubiquitous data type is a data type that occurs "all over the place" in a system.

As lisp programmers have know for a long time it is better to have a smallish number of ubiquitous data types and a large number of sma functions that work on them, than to have a large number of data types and a small number of functions that work on them.

A ubiquitous data structure is something like a linked list, or an array or a hash table or a more advanced object like a time or date or filename.

In an OOPL I have to choose some base object in which I will define the ubiquitous data structure, all other objects that want to use this data structure must inherit this object. Suppose now I want to create some "time" object, where does this belong and in which object...

Objection 4 - Objects have private state.

State is the root of all evil. In particular functions with side effects should be avoided.

While state in programming languages is undesirable, in the real world state abounds. I am highly interested in the state of my bank account, and when I deposit or withdraw money from my bank I expect the state of my bank account to be correctly updated.

Given that state exists in the real world what facilities should programming language provide for dealing with state?

- OOPLs say "hide the state from the programmer". The states is hidden and visible only through access functions.
- Conventional programming languages (C, Pascal) say that the visibility of state variables is controlled by the scope rules of the language.
- Pure declarative languages say that there is no state. The global state of the system is carried into all functions and comes out from all functions. Mechanisms like monads (for FPLs) and DCGs (logic languages) are used to hide state from the programmer so they can program "as if state didn't matter" but have full access to the state of the system should this be necessary.

The "hide the state from the programmer" option chosen by OOPLs is the worse possible choice. Instead of revealing the state and trying to find ways to minimise the nuisance of state, they hide it away.

Why OO was popular?

- Reason 1 - It was thought to be easy to learn.
- Reason 2 - It was thought to make code reuse easier.
- Reason 3 - It was hyped.
- Reason 4 - It created a new software industry.

I see no evidence of 1 and 2. Reasons seem to be the driving force behind the technology. If a language technology is so bad that it creates a new industry to solve problems of its own making then it must be a good idea for the guys who want to make money.

This is is the real driving force behind OOPs.