Machinery Object Model – MOM

An Object Model for Interactive System Design

https://webist.eu
welkom@webist.eu
BTC Wallet: 1wEbisUZkybCV1yJrhUTsirU1hVVXcnyG

Definitions
Abstraction: the quality of dealing with ideas rather than events.
Machinery:

- machines collectively.
- The organization or structure of something or for doing something.
- The means or system by which something is kept in action or a desired result is obtained.
- The components of a machine.

Abstract
Abstractional development model is by default implemented in programming languages and it is accepted as one of the basics of software architecture.
With the expansion of code base the scalability and overhead issues emerge. Refactoring due to dependency management and quality assurance testing become important aspects of software development.
Machinery Object Model (MOM) is the ultimate software development solution as expand itself towards the right architecture from the beginning. MOM is a beforehand high discipline development, testings afterwards is generally limited to result set.

Abstractional Development Model - symptom fighting
Continuation within abstractional development model comes with the consequence of regular dependency, quality assurance and unit testing on frequent schedule.
SOLID principle is put forward as the best practice, however some examples of SOLID principle illustrates essentially the problematic.

- Continuously taking care of ‘high cohesion and low coupling’
  This is a symptom of the fact that objects are coupled via inheritance mechanism.
  In fact, coupling in a system should be there from the beginning to the end, as a whole and single process. High cohesion is a best practice for a small set organization wherein the dependencies should not have effect on the bigger whole.
- Liskov Substitution Principle “*functions that use pointers to base classes must be able to use objects of derived classes without knowing it*.”
  This, often problematic, parent-child relation is generally cover-fixed with polymorphic approach. Polymorphic on itself is a pseudo-science. Trying’s real world mapping via polymorphic is a never ending refactor-development.

SOLID principle is solid as long as the architectural environment consists of abstract structures.

Inheritance by extending do not scale well
Encapsulation introduces the need of a higher level object and that leads to dependency management. It becomes burden that need to be maintained by refactoring.

Inheritance in an abstractional structure is done by extending. Since an abstract object behave as origin (or root) it can not expand backwards. Connecting to a back reference in an abstract environment is not a regular expansion of code base but manual refactoring of existing code base first, then being able to expand it.

Abstraction is a fabrication
Abstraction is described as ‘the process of generalization’. It does not directly address the process, instead it is a structure where-on a process runs. It is ‘not for the active transition’ but ‘for the structural resource’s at the end points’.
Abstraction remains sufficient for the two and three-dimensional structures.
There are many abstract structures.
Math, geometry, algebra is abstract. Path, URL, Document structures are abstract. Database table structures are abstract. Origin-branch structures are abstract. 2D, 3D structures are abstract.
Abstract structures are regularly fabricated (or invented) for problem solving. Such as a metric system, a calculator machine.

Distinction by type or intent
Within the abstractional structures there is distinction by type, distinction by intent do not exist.
Distinction by intent is possible when an active object is able to ask the intent to the preceding object (c.q. back reference). This indicates that for distinction by intent the objects-structure must be able expand not only forwards but also backwards.

It is wise to know that in an abstract world ‘a car never hits the wall’.
Because there is an infinite dividable distance, think of metric values, to the wall.

Events – Recognizing the Interactivity
A user send post data with a certain intent and expecting a response. This is a typical example of an interactivity required event. It is time related, it has history and future expectations.
Software development for such an event should seek a solution in relational process instead of relational structure’s. In another words in bi-directions of active transition instead of in prepared generalization.

Science of Relational Processing
The main concern of a software programmer is writing computer language code. Soon the programmer faces the need of expansion, the code base must be well organized. Therefore Software code writers consider themselves more like software developer rather than software programmer.
This is another indication of a relational process.

Interactivity needs back reference software ~~programming~~ development solution. The living objects in the real world showing good example’s of interactivity.
Each living object hold a unique code, called DNA, and via this unique code the object is able to back reference all the way to the origin. The coupling is loose and traceable. Possible risk of stack overflow is minimum.
And yet it’s association maintained by weak relationship. Relation coupling seem like ‘being part of it’, but it is in reality a ‘has-a’ association and therefore a ‘make use of it’ coupling.
The DNA solution in the real world is equivalent to the blue-print solution in software development.
Pass by reference (holder-reference where value to be assigned), instead of pass by value (holder-variable where value is already assigned), in modern programming languages is an example of it.

As it can be concluded from the past developments and headed future developments, all the involved classes in a process must remain connected. In other words there must be no state loss before the IO (Input/Output) cycle of the system is completed.

Machinery Object Model – the solution
When there is interaction between objects, accompanied by the move (c.q. time) element, then the abstractions, such as metric values, should be considered as consumables.
The abstracted values are to be considered as wrapped values into a value request-able object to ask the outcome where an input or configuration value will be perceived as intent.
This is how the MOM (Machinery Object Model) implements the back reference for interactivity.

MOM respects I/O (Input/Output) standard and accept it in Consumable/InterActor context.

```
class Machinery < Consumable, InterActor > implements Machine
 {
   function use( Usable )
   {
     return [ ];
   }
 }
```

Illustration above is the vastly formulae of object construction in MOM.

Do’s MOM

- Consumables and InterActor construction injection is mandatory.
- Every object is a machinery or a supplement to another machinery.
- Interface implementation is recommended
  Two types of interfaces;
  - Specification interfaces contains constant values
  - Behavioral interfaces contains method declarations
- Initiation or final level objects are exemption to the machinery construction.
- An InterActor is an object type and sound like the gateway to the governance, handlers, OS environment and external resources.
- A consumable is mixed type and sound like input, request-ability recommended.
  Notice the distinction between configuration and specification values.
  Configuration values sound like input, while specification values belong to interface’s.
- Usable method injection instead of setters and getters. A Usable is;
  - A specific task handler, it does not have to be in ‘using’ association with the current machinery object.
  - A supportive object that can be ‘used’ for the result and do not change the internal state or functionality. A setter injection on the other hand sets a new value to a property that change the state.
  - Injection-only object that is not to be questioned by others. A method-injected object made available to others then the design mistakes should be questioned.
- InterActor is the first to think about, it will implement the business logic.
- An InterActor should have a direct access to a shared memory for reuse and reference things.
- When there is uncertainty about the Consumables, it should be an empty object or an empty array.

- No reference loss, full integrity.
  Every level machinery object remain connected to the initial access of the system.
  Other than classes, there is no need for global constants or variables, statics, singletons and such. System with MOM discipline is always ready to daemonize with any option parameters and there is no need to develop a separate ‘console’ system.
- Semantics are generally resolved by namespace.
  A correct object name usually covers multiple interpretations.
- Ask the intent to the consumable, for example configuration, get the answer from interActor and tell the story.
  Cost 100 satoshi? I check my wallet, I send 100 satoshi.
- A machinery object can be the consumable of the other, no problem. Call it micro-service, middleware, whatever you like it.
- Wrap a callable in an abstractional model allowed. Such as HTML document build for HTTP response.
  In that case the callable, which is access to M.O.M environment, is the result set delivery engine.

Do Not’s MOM

- Extending parent classes (inheritance by extension) is not M.O.M.
- Architecture before coding is not needed, highly disciplined M.O.M development with good semantics will grow itself to the right architecture
- No setter methods, but a single method injection
- Object-in-object wrapping can lead to memory leak
- Broken for test unit? Then it is not a M.O.M

Other considerations

- OO Design patterns, most of them, are fairy tale. Fairy tale is good for personal fantasy.
  The real world patterns are aggregations and composites, no more.
- Do not put business logic and handler into the same class.

Conclusion

Events and interactivity should not be overlooked. Software development relying on abstraction with inheritance mechanism is not the proper solution for interactivity. Machinery object development model is functional and appropriate solution for interactive relational process as well as for abstract relational structures.

References

[1] ‘a car never hits the wall’ Math teacher Yılmaz Üçgöz, 1988
[2] https://www.merriam-webster.com/dictionary/machinery