Changing Requirements and How to Minimize Their Impact

Stephen J. Mellor

In which we describe techniques for organizing the analysis and design so that it is resistant to requirements change. The fundamental technique is to find, and model, the invariant in the problem. The invariant of the problem is then expressed in logic, while the variants are expressed in data.

These techniques reduce the size of the analysis and design models, as well as the code. They also provide a basis for understanding the size and true impact of scope changes ,and maximizing the value of existing analysis and design work.

1. THE PROBLEM

Changing requirements are a fact of life on any project, and it can be extremely expensive in time and effort to manage requirements changes. Several techniques have been tried with varying degrees of success:

- Freezing requirements: This approach declares the requirements "frozen" at some point in the project, and design and code then begin. This may occur for the whole system (largely discredited now), or on the basis of groups of use cases.
- Cost plus: If any change is allowed after the freezing date, any changes are charged back to the customer at cost, plus a (sometimes large) premium.
- The most common technique, however, is to whine: "The client has changed the requirements again. Why do they keep doing this to? Life would be so much simpler if they just went away!"

Of course, none of this helps. And even if it did, eventually we'll deploy the system and then we'll still have to deal with change. Maintenance and modification are, after all, only requirements changes after deployment.

The good news is that the impact of a requirements change can vary significantly depending on how you organize the analysis and design of the system.

2. THE SOLUTION

The solution to the problem is Gordian in its simplicity: Find the part that doesn't change, and can't ever change, and base the logic of your requirements on that. The part that does change must then be expressed in data.

This approach is called *finding the invariant*. Rather than wrestle with each requirement strand in the problem knot, avoid the problem by raising the level of abstraction. Alexander the Great didn't bother with trying to untie the Gordian knot. He just cut the thing, and went ahead with the business of conquering Asia.

There are two problems with this. First, the advice so far doesn't seem too helpful, and second, it seems altogether too obvious. What we need are some concrete examples.

3. AN EXAMPLE: PLANT CONFIGURATION

The job here is to replace an aging chemical manufacturing plant. A team of mechanical, chemical, electrical and software engineers assembled to produce the specifications for a replacement. Figure 1 shows a simplified snapshot of the plant.

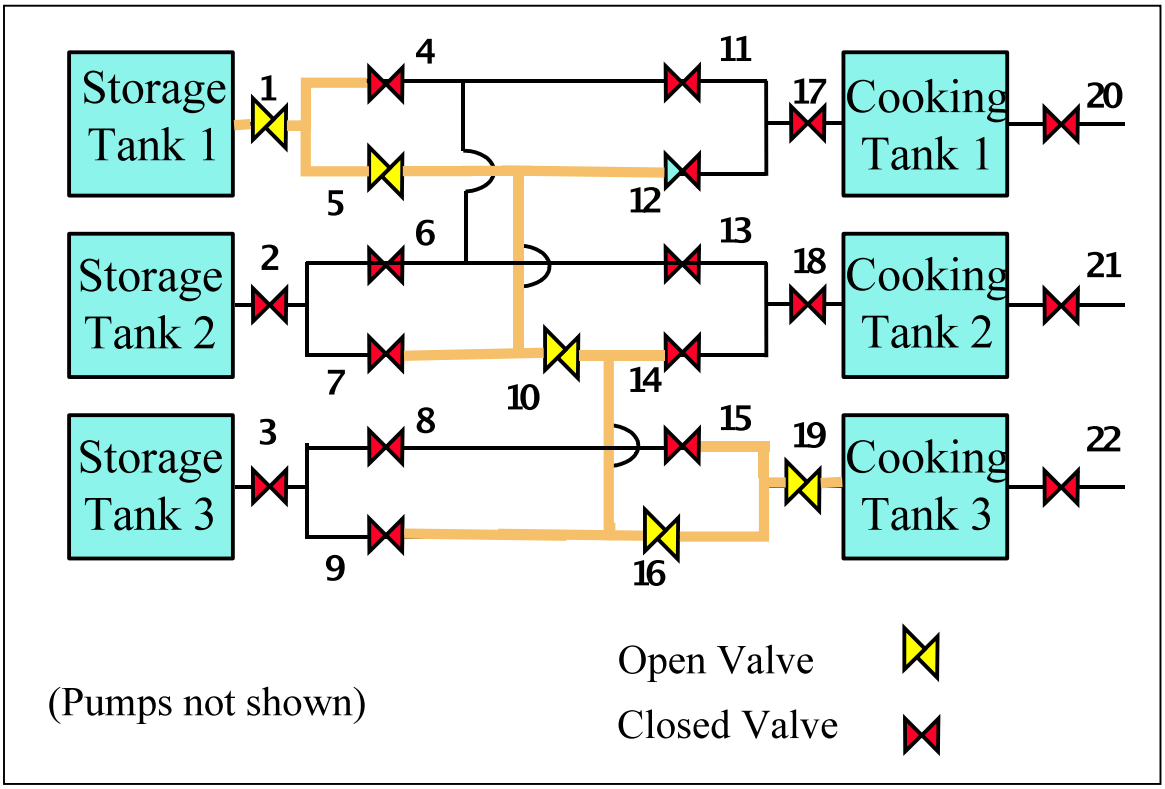

Figure 1: The plant

One unfortunate software engineer had the task of specifying the movement of the chemicals through the plant from the storage tanks to their destination, the cooking tanks, as shown by the shaded path on the figure. After eighteen months of work to produce detailed requirements, the software engineer was frustrated. Extremely frustrated. The requirements had changed yet again.

After the initial requirements gathering phase, the requirements were sent out for review, and multiple changes had come back—many of them contradictory. After resolving most of these to produce a correct model based on the information so far, the procedures were changed; the plant configuration changed; a new supervising engineer had a "better" idea; and, as a consequence, the review cycle lengthened.

Worse, the review cycle had not finished before the whole process started again: the procedures were changed; the plant configuration changed; someone found the flaw in the "better" idea; and the review cycle itself became unstable. The whole process was simply not converging.

Our frustrated software engineer was no fool. Naturally, the procedures were parameterized to refer to Inlet Valves and Outlet Valves, Storage Tanks and Cooking Tanks, but there were many special cases and exceptions. Figure 2 shows an example of the type of logic that results from this level of abstraction. I'm sure you can find many things wrong with it. But before continuing to read, think about how you would improve (well, reconstruct) the necessary logic..

```
Function OpenReservedPath(
    StorageTank, StWhich
    CookingTank, CkWhich );
OpenValve( StorageTank.Outlet);
OpenValve( StorageTank.StWhich);
If ( StorageTank = 1 and
     CookingTank = 3 ) then
     OpenValve( Middle );  //Valve 10
OpenValve( StorageTank.Outlet);
OpenValve( CookingTank.CkWhich);
OpenValve( CookingTank.Inlet);
EndFunction;

    OpenReservedPath( Storage1, Cooking3 );
```

Figure 2: Sample logic for requirement, expressed as pseudocode

The way to improve this is to raise the level of abstraction until you find the invariant, until you find out what won't and can't change. For example:

- A closed pipe will contain the same fluid.
- A pump can move fluid from one pipe to another. If a valve is open between two pipes, they behave like a single pipe forming a path, a "pipe-in-path."
- Each component has to be connected to other components for it to be in the same plant (closure).

The requirements must reflect these invariants. To find them, we must turn our attention away from the things we can control, like valves, to the things that connect them, like pipes. The list of invariants above suggests following abstractions:

- A closed pipe. Consider a piece of pipe with closed exterior valves, and no interior valves. The pipe may be very simple, with only two ends, each with a valve, or it could be a T with three closed valves, or an X with four. Indeed, it can be any shape terminated by closed valves. We shall call this a Pipe, and it is defined to have no interior valves.
- A connection: "If a valve is open between two pipes…" A connection, then, is the fact that two pipes are connected. The valve that connects them is modeled as a PipeValve.
- When the valve between two pipes is opened, a new abstraction exists: a Pipe-in-Path. When we attempt to construct a path from a storage tank to a cooking tank, it comprises a set of pipes-in-paths.
- All pipe-in-paths must be connected—if we're in the same plant. This raises a whole set of new possibilities: we may check to see that no pipe is disconnected, that each pipe-in-path connects to another, with no "dead-ends" and so on.

Figure 3 models these abstractions as a set of classes using UML notation.

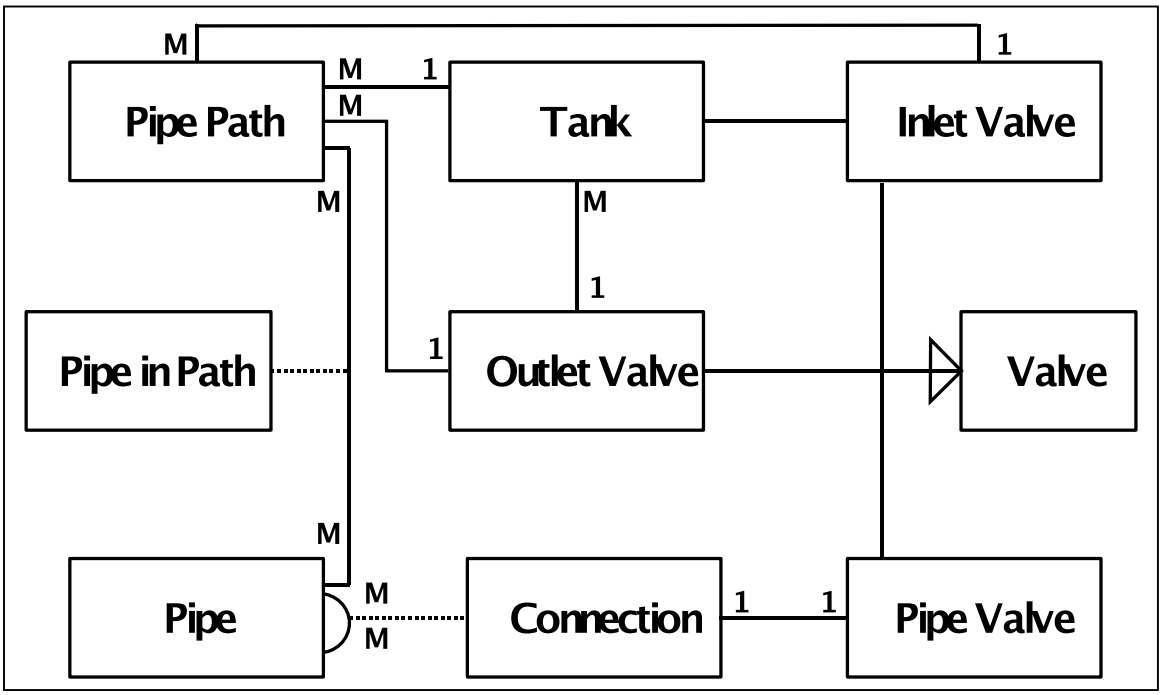

Figure 3: UML model of required abstractions

To understand how this set of abstractions works, consider the shaded path from Storage Tank 1 to Cooking Tank 3. First, we need to add names for the various pipes that comprise the path, which we'll boringly name Path 1. Figure 4 shows these names.

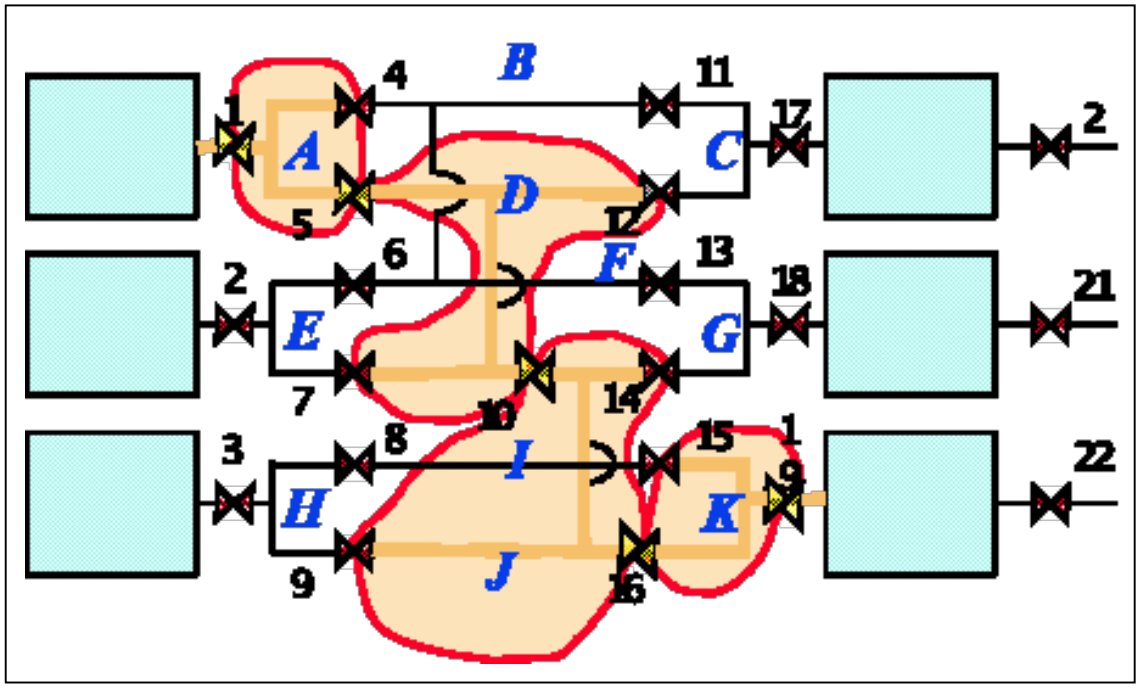

Figure 4: Marked up plant schematic to show Path1

The tables below show the object instances of Connection and Pipe In Path.

Pipe-in-Path		
Pipe 1 ID	***Pipe 2 ID***	***Pipe Valve ID***
A	D	5
D	J	10
J	K	15

Connection	
Path ID	***Pipe ID***
Path1	A
Path1	D
Path1	J
Path1	K

With this data, we can formulate the required behavior. Path1 is the path that connects Storage Tank 1 with Cooking Tank 3. Examining the instances, we can see that Path1 comprises pipes A, D, J, and K. To open Path1, then, we need to find all the valves that connect the pipe paths; namely, 5, 10 and 15. We open these valves, and that's it.

This formulation is resistant to changes in the configuration of the plant. Consider now placing an intervening valve somewhere along Pipe D, thus dividing Pipe D into two pipes D1 and D2. The modified entries are in italics. The object instances now look like:

Pipe-in-Path		
Pipe 1 ID	***Pipe 2 ID***	***Pipe Valve ID***
A	*D1*	5
D1	*D2*	*23*
D2	J	10
J	K	15

Connection	
Path ID	***Pipe ID***
Path1	A
Path1	*D1*
Path1	*D2*
Path1	J
Path1	K

The logic remains exactly the same. Consequently, the requirements can be written in an abstract way that relies on data describing the plant. Once the plant has been constructed, we (or better yet, someone else) may create the object instances and we're done.

Once you see this solution—or if you've seen it before—it seems obvious, in the same manner that Alexander's solution today seems obvious. Of course, the trick is to understand the principles involved so that you can apply them on cases you haven't seen.

A potential objection to this formulation is that the resulting code can be inefficient if the number of pipes is large. And so it is. However, the description of the problem remains valid. To make an efficient implementation, we need some translation of the same behavior that avoids the run-time search for instances.

Many such translations exist. Which one you choose will depend *not* on the semantics of the application, but on the manner in which it is used. Hence, a large number of pipes with relatively few operations would animate one design, while frequent operations on a small number of instances would cause to choose quite another.

One such translation simply moves the search off line. You write a program that converts the Pipe-in-Path table into a set of calls to OpenValve for each Path. This yields one program per path; we activate the appropriate one on execution of the associated operation.

4. EXAMPLE: MANUFACTURING OPERATIONS

For another example, consider the same plant from an operational point of view. We may define a set of use cases for manufacturing chemicals that involve mixing, heating and cooking chemicals in a tank, while others involve moving chemicals around in the plant. For example:

Mix Chemicals
Precondition: (none)
Postcondition: Requested quantities moved between tanks

Heat Chemicals
Precondition: Cooking tank contains mixture

Postcondition: Mixture at requested temperature at one degree per minute

Each use case comprises a sequence of steps, such as moving one chemical to a tank, then some amount of another, then mixing until turned off, and so on.

These use cases will certainly change. While the system will always Mix Chemicals, the details of how the chemicals shall be mixed is quite changeable. Perhaps one has to begin with one chemical in the tank and add another slowly to prevent an explosion. Perhaps this fact is considered only late in the project after the code has been written, or even after an explosion. The scope for change is enormous.

5. USING LEVELS OF ABSTRACTION

What, then, is the invariant in this problem, and how does one find it? One technique is to raise the level of abstraction. As a different example, consider producing a profit-and-loss statement for a service company that also requires detailed time sheets from each service provider that feed into the profit-and-loss statement. Instead of casting this as a profit-and-loss/timesheets problem, the next level of abstraction up is the notion of adding and subtracting columns of numbers: a cellular automaton, more commonly called a spreadsheet.

The process here is to look for common elements that can apply uniformly without worrying about what—exactly—is to be done. In the spreadsheet, this is "adding numbers"—we don't care what the meaning of those numbers is. In the plant, we have:

- activation of some operation (whatever it is)
- an operation ends, with some return code (whatever the meaning of that code is)
- sequencing operations (regardless of the meaning of that sequencing)
- naming and reusing sequences so they may re-appear in multiple use cases
- naming and reusing named sequences in a nested manner

The core of this problem is sequences of operations, just as the core of the spreadsheet is adding numbers. Now comes the question of determining what additional features are useful in the context of this specific problem. For example:

- Should the return codes be true/false, or more general?
- Is sequence adequate, or should concurrency be allowed?
- Concurrent at the level of two independent sequences that share equipment?
- Or concurrency in the sense of forks and joins?
- Should operations always be primitive, or may we name groups?
- Should named blocks be independent, or should they be context dependent?
- Should nesting be allowed, or not?

Seemingly innocuous features may add a significant amount of work. The simple idea of allowing two sequences of operations to execute concurrently requires a larger change than just adding what looks like a single new requirement. Adding concurrency would require us to prevent multiple operations from using the same pipes, valves, pumps and

tanks at the same time. There is also the matter of whether an operation may begin when one, some number, or all preceding operations complete. Or even some named subset. Then we'd have to provide mechanisms for the logical equivalents of forking and joining.

Happily, we may choose to require only a subset of all these potential features based on the underlying needs of the application. The degree to which this model is resistant to change is dependent how general the model of execution is.

6. EXAMPLE: PROFILES

Consider the manufacturing plant with multiple different kinds of pumps. Each pump type has a kind of "gearing factor" that must be supplied depending on the final desired speed. The following table provides examples.

Type	Speed	Value
Acme 101	0-10 10-30 25-50	speed * 1.0 speed * 1.75 speed * 2
NewPumpCo	No gearing	
Pump 'em	0-20 20-up	speed * 1 speed * 3

Potential changes include:

- different numbers of tiers
- the start and stop value for each tier
- the variety and number of devices
- further idiosyncrasies of the various devices (possibly other than pumps)

The currently popular approach to solving this problem is to build an inheritance hierarchy for the pumps. Each class will have an operation TurnOn that is implemented differently for each type of pump. Hence TurnOn for objects belonging to the Acme 101 Pump class will implement one algorithm, while TurnOn for New Pump Co pumps will use a different one. Let's evaluate this for adaptability.

This approach abstracts away the differences in pumps, localizing the individual behavior of each pump type in its own class. The abstract Pump has operations that turn each pump on and off, and allow for other "pump-ish" operations. This may generalize to allow for other kinds of devices by using the same technique. Note then that the addition of a new pump type requires writing a new class that encapsulates its behavior. This class must conform to the interfaces defined by the more general classes in the hierarchy.

Now let's consider another approach. Let us abstract the tiers that determine the gearing factor. We represent this as tables below.

Structure	Start	Stop	Factor
Acme101	0	10	1
Acme101	10	27	1.75
Acme101	27	50	2
Acme101	50	None	0
NewPumpCo	0	None	1
Pump'em	0	etc	

The logic required here is extraordinarily simple:

- Find the tiers matching the named device
- Find the tier matching the desired speed
- Compute the desired speed * gearing factor
- Send it to the device

This approach works for any device, not just pumps, that require control of this nature. No new code is required; only the data tables that describe the tier structures and tiers.

8. THE VARIANTS

Because we have placed the variants in data, the data can address other problems than the one we actually solved. For example, the "plant configuration" abstraction applies not just plant configuration, but in problems like telephone networks, automated ground vehicles in factories, or ventilation systems in automobiles. In fact, the "plant configuration" abstraction is actually a certain kind of network/connectivity abstraction. From the base abstraction, we may modify it to handle least cost algorithms for a network, what happens if paths cross, and what should happen if the a connection fails to open or close. This particular abstraction was first formulated to turn off power to a third rail in an urban mass transit system. In addition to simply opening breakers, the system had to deal with breakers that failed to open.

The second abstraction, operation sequencing, could be used in robot control, computational networks, programmable push buttons, and decision tree of various kinds. This solution relies both on expressing the ordering of the operations in data, like the network, but also on allowing the "operation" to be specified separately. At implementation time, this often entails invoking disembodies functions to open a valve, heat the mixture, urn on pumps etc. For ease of testing, these operations should be functions in the mathematical sense of the word: they have no state and always operate in exactly the same way. If this is not practical, you will need some way of understanding what stored data is changed as a result of an operation.

The third abstraction, tiers, applies to profile control in devices such as copiers, robots and the like, and to differential billing (i.e., charging a different amount at different times or for different amounts, such as "3 cents per minute for the first 10 minutes between

6:00pm and 8:00am, them 2 cents per minute thereafter." Each different amount becomes a tier, and each different circumstance a new tier structure.

9. CONCLUSIONS

The analyst has to take responsibility for changing requirements. Whining simply doesn't cut it. This axiom implies that the analyst must also take responsibility for the manner in which the requirements are understood and expressed. In turn, this fundamentally affects the design and the code.

The underlying technique is deceptively simple to express, but surprisingly difficult to apply: model the invariant.

This paper has provided a number of example ways to find the invariant and model it. The variants are then expressed in data, which has the happy effect of shifting responsibility for application decisions where they belong—with the client and customer.

The approach described herein also provides a basis for understanding the size and true impact of scope changes, and maximizing the value of existing analysis and design work through reuse and variation the underlying abstractions.