Architecting adaptive software

A. F. M. van der Sijpt,
luminis, December 2007

Abstract

All around us, devices are getting more aware of their environment. This opens new doors, but also poses new challenges. Handling this changing world requires *adaptivity*; this article details luminis' view on the notion of adaptivity, and shows how it can be brought about. Six concepts to be incorporated on an architectural level are introduced and discussed.

1 Why?

The world in which software systems live is getting more dynamic. Digital devices in our world are more fine grained than ever before: this started with moving processing power from mainframes to PCs, and now to notebooks, PDAs and mobile phones, as well as embedded devices. Combined with increased networking capabilities, both amongst each other and to the internet, this creates a 'living' network of many in(ter)dependent devices, using each other's services.

This dynamic environment provides few certainties: useful devices can show up and leave without notice, and any device could get swamped with requests from others. When services are available, we want to use them; when they are not, we should carry on to best of our capabilities. This calls for *adaptivity*.

2 What?

What exactly is adaptivity? *Adaptivity* comes from *adapt*, meaning 'to make fit'. So, defining adaptivity as a quality of a system, we propose the definition

> "Having the capacity or tendency to make fit."

This raises another question: *what* adapts *what* to *what*? This question can be answered in a number of ways, depending on the way we look at the system.

2.1 Measure of adaptivity

In stead of filling in concrete cases for the three *what*'s above, we will consider the system's boundaries, i.e., do the *what*'s represent the same, or different sides of the border. Using this notion, we can create a ranking of 'interestingness'.

Skipping the full categorization[1], we end up with two interesting categories of adaptive systems. Both adapt themselves (answering the first two *what*'s), with one adapting to itself (known as category 2), and the other adapting to the environment (known as category 3).

Both categories need information on the environment. Category 3 will capture this by looking around, while category 2 blends into the environment. When reconsidering a category 3 system, something peculiar happens: in order to react to influences, it measures or models these. So, in a way, the environment becomes part of the system, creating a category 2 system.

[1]See my thesis for the complete story, available at `http://alexandria.tue.nl/extra1/afstversl/wsk-i/sijpt2007.pdf`.

A preliminary measure of the success of a system could lie in the way it 'absorbs' it environment, or lets itself get absorbed. Many successful systems that exhibit adaptivity have themselves embedded in their environment. For example, the IP routing example below does not have a model of the topology of the network, it *is* the topology.

2.2 What does adaptivity look like?

Now we have identified the categories we want to work with, what does adaptivity actually look like? There is no objective measure for adaptivity, like there is no objective measure for intelligence. Both exist only in the eye of the beholder; humans perceive behavior as intelligent when it is similar to their own.

What is needed to perceive a system as being adaptive? A general consensus on this does not exist, but at least an adaptive system should exhibit behavior we (a) do not expect, and (b) makes sense. We want a system to react to relevant stimuli, and operate in a way that is similar to what we, as humans, would do.

Systems, perceived as being adaptive, do already exist. The classic example is internet packet routing, in which packets 'magically' find their way around a potentially changing network. For a novice, this is impressive, but the protocols governing this are rather simple, and have been around for a long time. An expert will not see this behavior as adaptive; he expects exactly what he sees. As soon as we know how it works, the 'magic' goes away, and it becomes 'just another system'. Or, to put this differently, "Any sufficiently advanced technology is indistinguishable from magic." –Arthur C. Clarke.

This leads me to believe that adaptivity does not actually *exist*; it is a property that we attribute to some behavior, but cannot be engineered directly. The same property applies to, for instance, neural networks. Even though their behavior is no longer directly predictable, it is still governed by rules, even though they may be too complex to describe or understand.

2.3 Now what?

Given the descriptions above, we end up with something we want, we do not understand and does not actually exist. Still, adaptivity is a desirable property, and we want to be able to create it in some way.

Since adaptivity is something we perceive, instead of something that is 'identifiable' in a system, we can conclude that it is a side-effect of other properties. The following section will describe six more-or-less engineerable properties of systems, which help in creating adaptive behavior.

3 How?

Being a side-effect of (a combination of) other system properties, adaptivity requires a focus on an architectural level.

We propose six *concepts* to be incorporated in a system, which support adaptivity. Like adaptivity, the concepts below are somewhat elusive too: none of these can be directly 'injected' into a system, but are the result of architecture-level decisions. They may be just as hard to introduce as adaptivity itself, but at least these can be more objectively measured.

3.1 Emergence

We stated earlier that "an adaptive system should exhibit behavior we (a) do not expect, and (b) makes sense"; (a) suggests the system's behavior will be more complex than is to be expected from its components.

Emergence may well be the most elusive concept in this list. The world around us is filled with examples of it; ranging from the spontaneous order in which cities and societies organize to the surprising complexity of cellular automata with sufficiently balanced rules[2]. There seem to be no basic rules for what is and what is not to be classified as emerging behavior, just as there is no real measure adaptivity.

The only binding property of all systems which display emerging properties can be described as 'sufficient complexity' [3]. Four ants do not make an ant hill; a few thousand might. It is not really possible to pinpoint the exact breakpoint, but it sure is there: above some threshold of complexity, the emergent properties of the system outweigh the individual influences.

In the end, there is not that much that can be directly engineered in the emergence department. Two elements can be held in mind, being (a) balance in rules, and (b) sufficient complexity.

3.2 Redundancy

'Adaptivity' comes with a ring of 'robustness' and 'resilience'. A system should be able to cope with its own malfunctions, like components breaking or behaving in a counter-productive way.

'Redundancy', on the other hand, has a more negative load, bringing images of superfluous and unnecessary components. However, the actual addition of those is not the interesting element. It is more useful to think about the way components can be *made* redundant, i.e., what provisions are necessary to make a component take over another's task.

Redundancy is one of the concepts which is observed most often in both real-world an technological systems, but at the same time, one of the hardest to articulate. Biology is riddled with redundancy: species consist of many animals, each of which as a unit is expendable, but together they can continue the species. In technology, a backup system requiring user intervention is a form of redundancy, just as using clustered hardware or redundant lines in the internet packet routing example above. Note that all of these measures does away with the single point-of-failure. Still, redundancy is something that goes against the deterministic nature of information systems as we know it; systems which have to rely on redundancy cannot be guaranteed to deliver a correct result, but should be characterised by the *probability* that a correct result shows up.

Note the link between this concept and the previous: systems with emergent properties do not rely on a single 'god'-component, but have many (more than strictly necessary) components, which are expendable. This brings resilience to 'breaking' components, but is also comes with a control issue: if not all components are controlled by a single entity, such as an organization or company, it can be very hard to influence the behavior of the system as a whole, let alone shut it down.

3.3 Decoupling

Just as redundant components increase the number of potential places where defects may occur, thus reducing the probability that any one flaw is a show-stopper, decoupled components allow the effects of a defect to be localized, and thus dampen the effect on the full system.

As an analogy of the effect of coupling, consider a guitar string. When it is wound to its required pitch on a guitar, exciting one part of the string will have an effect throughout the entire string, in a well-defined pattern. The disturbance is not localized at all, but distributed throughout the system. If we take the same guitar string, and lie it on a table, it can very well localize effects. Push a few bends in it, and the effects will be dampened, the further we get from the place where the original disturbance happens.

In both cases above, we see that 'something has to give', i.e., there *has* to be some reaction to the disturbance. In the wound case, the reaction is vibration of the entire string, whereas in the unwound state, the shape of some part of the string changes.

When considering the adaptive behavior of a system, the latter case is desirable: local disturbances should have a local effects. Traditional means of decoupling software systems rely on componentizing the system, and defining as-simple-as-possible interfaces for them to communicate. This approach tries to dampen the effect of a disturbance *in one component*; however, just

[2]For instance, see Conway's Game of Life, `http://en.wikipedia.org/wiki/Conway's_Game_of_Life`.

[3]Roger Lewin. *Complexity: Life at the Edge of Chaos*, 1992

as a car's suspension system does not take away all effects the road has, I don't think it is natural to dampen disturbances in a single component. Perhaps we should move to an understanding of decoupling where the amount of dampening is a variable, in stead of trying to fix it on 'localization to a single component'.

3.4 Service awareness

Up to now, we have discussed software as (potentially) consisting of components. However, these components are not interested in others of their own kind, but in what another can *do*, and components happen to be the logical unit of functionality. The identity of the provider is of no importance; hence, the component is only interested in some *service*.

When we move away from components, and view a system as a collection of interdependent services, we end up with a problem: the service-environment is not static. Services can come and go at undefined moments.

In preparing for an environment with an unpredictable set of services, we can choose to rely on the bare minimum of services we can get away with; this creates a service which can operate in most situations. However, when more services are available that we could use, this could increase the quality of our service.

This takes us back to the title of this concept, 'service awareness': components that want to use services should be aware that service may come and go, use them when they are available, and reduce quality when they are not. This reduction of on functionality or correctness in favor of operation is also a key issue for scalability, as discussed later in this document.

Examples of service awareness usually pertain to a service which does exactly the same in all situations, but will just have a lower quality when fewer inputs are available. Consider, for example, an internet search engine, which uses the results from a number of other search engines. These results can then, for instance, be combined into a more balanced result set. The quality of this service is roughly proportional to the number of inputs available.

In practice, services rarely provide the same functionality. That is not necessary: for instance, many systems contain some logging mechanism. For most applications, this mechanism is not an essential element, but if available, it can increase the quality of the service (by e.g. providing earlier warning that something is about to break).

Another example is the system of audio notifications as used by many OSes. The device's functionality does not rely on a sound system being available, but it may increase the quality of the service provided to the user.

3.5 Parallelizability $\implies$ distributability

Moderately-priced computer systems now ship with dual-core processors, embedded processors like the ARM11 contain up to four cores, and the increasing communication capabilities of embedded systems uncovers a processing potential which sits around unused for most of the time. To make use of these movements, the workload for a system should be shareable, i.e., divisible in smaller chunks, tasks or processes with a limited scope.

Sharing a workload is known as distribution. However, for a task to be distributable, it should be created in such a way that it *can* be parallelized. This means that separate chunks should share as little state as possible, and have as few as possible synchronization points with others. A combination of decoupling with highly specialized components could be a useful aid in this direction.

3.6 Scalability

Above, we have seen a concept that has to do with making systems 'bigger': redundancy, which adds components to increase reliability. However, we could also add resources to get higher quality of service, as shown by service awareness. Ideally, the characteristics of quality of service, resource consumption and performance are sides of the same triangle, and can be interchanged.

This ideal is known as scalability, or the ability to trade one characteristic for another. For example, investing in some extra hardware is usually hardly a problem, but making a system *use* this extra hardware to the best extent possible is an entirely different issue. In addition to manual intervention, a useful application of scalability is automatic negotiation. The primordial example in this case is a web server which, when confronted with more concurrent requests than it was designed for, switches to a text-only mode, thus serving more clients, but with a lower quality of service.

Scalability is the concept that has the most adaptive 'ring' to it, and it is one that combines a number of other concepts[4], most notably redundancy, decoupling and service awareness.

4 Concluding remarks

Before giving the impression that the six concepts above are sanctifying elements for creating an adaptive system, I would like to make a few notes.

First up, notice that there are rather intimate links between the concepts above. None of them can be engineered in isolation, or bolted on to an existing system. They should be introduced on an architectural level.

Second, the concepts above seem to glorify componentization. Coming from a background of formal systems design, the reductionist view is no stranger to me. Still, I would like to point out that componentization comes with its own perils. The main risk of 'chopping a problem up' is losing the inherent complexity of the problem. We may end up with chunks of solution to well-defined problems, but in the end, the chunks might not add up to solve the original problem. The problem has then been reduced to something that is *solvable*, but not necessarily a *solution*. ("Everything should be made as simple as possible, but not simpler." –Albert Einstein.)

In the end, the views above are just guidelines. They are no recipe for success, nor is it guaranteed that ignoring them results in a system with bad adaptive properties. I argue that taking some of these concepts into consideration when architecting a system is the first step into creating a system with adaptive properties.

[4]Although that could mean we can get scalability 'for free', it can also mean that scalability is harder to engineer than the other concepts.