Out of the Tar Pit

MARCH 30, 2006

Out of the Tar Pit • Moseley & Marks 2006.

This is the final Desert Island Paper choice from Jonas Bonér, and a great way to round out the week. Out of the Tar Pit' was the 10th paper that I covered in the #themorningpaper series, but at that time I was only giving highlights on twitter, not via the blog format. So I'm grateful for a chance to revisit it. It's a long read at 96 pages, but written in a very easy to digest essay style. The crux of the argument is that complexity lies at the root of our problems with software systems. By separating the essential (to the problem domain) complexity from any accidental complexity – and avoiding the latter altogether wherever possible – the authors argue for a better way of building software.

Complexity and its causes

> Complexity is the root cause of the vast majority of problems with software today... The primary status of complexity as the major cause comes simply from the fact that being able to understand a system is a prerequisite for avoiding all of them, and of course it is this which complexity destroys.

How do we try to understand systems? We can try to understand a system from the outside, by testing it and seeing what it does, but this approach has limits. Far more important is the process of informal reasoning. This is greatly hampered by state.

> The mental processes which are used to do this informal reasoning often revolve around a case-by-case mental simulation of behaviour... As the number of cases to enhance the number of possible states, the mental leaps become correspondingly larger, the effectiveness of this mental approach buckles almost as quickly as testing. For every single bit of state that we add, we double the total number of possible states.

State also contaminates stateless procedures if they make use of – even indirectly – any other procedure that is stateful.

> ...it is our belief that the single biggest remaining cause of complexity in most contemporary large systems is state, and the more we can do to limit and manage state, the better.

After state, control logic is the biggest source of complexity: "When a programmer is bored (through use of a language with implicit control flow) to specify the control, he or she is being forced to specify an aspect of how the system should work rather than simply what is desired."

> Like basic control, such as branching, but as opposed to sequencing, concurrency is normally specified explicitly in most languages. The most common model is "shared state concurrency" in which specification for explicit synchronization is required. The impacts that this has for informal reasoning are well known, and the difficulty comes from adding further to the number of scenarios that must mentally be considered as the program is read.

In the same vein, we've seen again and again that distribution brings with it the same problems.

Once complexity is let into a system, things tend to go downhill fairly quickly: "Complexity breeds complexity."

> Duplication is a prime example of this – if due to state, control, or code reuse it is not clear that the functionality already exists, then it is very simple to duplicate functionality between more or less arbitrary components. There will be a strong tendency to duplicate. This is particularly true in the presence of time pressures.

Powerful languages can also be a source of complexity when trying to understand a system.

> The bottom line is that the more powerful a language (i.e. the more that is possible within the language), the harder it is to understand systems constructed in it.

Moseley and Marks then turn their attention to programming language paradigms to see how they stack up. Object orientation (from traditional passive objects to the actor/actor styles!) can make it hard to enforce constraints across multiple objects, and introduces complexity of its own with concept of object identity.

> The bottom line is that all forms of OOP rely on state (contained in objects) to model the external all behaviour is affected by state. As a result of this, OOP suffers directly from the problems associated with state, and as such we believe that it does not provide an adequate foundation for avoiding complexity.

In contrast, the primary strength of (pure) functional programming is that by avoiding state and side-effects the entire system gains the property of referential transparency. The problems of state moving from encapsulation within an object to a growing list of parameters that are passed around from function to function is discussed. "This does not detract from the general power of the functional approach."

> More generally, we would argue that whatever the language being used there are large benefits to be had from avoiding hidden, implicit, mutable state.

Writing in 2006, the authors lament that such arguments have been insufficient to result in widespread adoption of functional programming (Adoption and awareness has certainly increased since then, but not to the point that Moseley and Marks would desire).

> We must therefore conclude that the main weakness of functional programming is the flip side of its main strength – namely that problems arise when (as is often the case) the system to be built must maintain some kind of state. ... One potential approach is the elegant systems of monads used by Haskell. Again, despite their huge strengths, monads have an air of the academic about them, give rise to the widespread adoption of functional techniques.

Logic programming "offers the tantalising promise of the ability to escape from the complexity problems caused by control."

Essential and Accidental Complexity

Essential Complexity is inherent in, and the essence of, the problem (as seen by the users).

Accidental Complexity is all the rest – complexity with which the (development team would not have to deal in the ideal world (e.g. complexity arising from performance issues and from suboptimal language and infrastructure)

> Note that the definition of essential is deliberately more strict than common usage. Specifically when we use the term essential we will mean strictly essential to the users' problem (as opposed to – perhaps – essential to some specific implementation of a system, or even – essential to software in general).

Essential complexity is that which even in the ideal world the team will have to be concerned with. However,

> given that in the real world not all possible ways are practical, the essential complexity will often be compounded by that contained with some accidental complexity. The definition does not seek to deny this – merely to identify its secondary nature.

State will be considered accidental state if it can be omitted in the ideal world, and the same applies to control. In the ideal world we would produce formal requirements ensuring that there is no *relevant* ambiguity in them, and then be able to simply execute those formal requirements.

> This state of affairs is absolute simplicity – it does not seem conceivable that we can do any better than this even in the ideal world. It is interesting to note that what we have just described is in fact the very essence of declarative programming – i.e. that you need only specify what you require, not how it must be achieved.

All control logic is therefore considered to be accidental complexity.

Data may be provided directly to the system as input, or derived. All derived data is accidental state as it can always be re-derived. (See the Factom paper for an example of this concept in the large)

> It is our belief that the vast majority of state (as encountered in typical contemporary systems) simply isn't needed (in the ideal world). Because of this, and the huge complexity which state can cause, the ideal world removes all non-essential state.

How close is it possible to get to the ideal world in the real one?

The Way Forward

There are two possible reasons why in practice – even with optimal language and infrastructure, we may require complexity which strictly is accidental:

- Performance – when accidental state and control is required for efficiency
- Ease of expression – accidental state can be the most natural way to express logic in some cases

> We believe that – despite the existence of required accidental complexity – it is possible to retain most of the simplicity of the ideal world in the real one. We now look at how this might be achievable. Our recommendations for dealing with complexity (as exemplified by both state and control) can be summed up as avoid, and separate.

Avoid accidental state and complexity wherever you can, and if you can't, then separate it from the rest of the system. "There is nothing particularly profound in these recommendations, but they are worth stating because they are emphatically not the way most software is developed today." The functional separation is a logic/state split in which all complexity of any kind is separated out from the pure logic of the system.

> One implication of this overall structure is that the system (essential + accidental but useful) should still function completely correctly if the 'accidental but useful' bits are removed – albeit possibly unacceptably slowly.

If we've separated out the parts in this way they will each be of a very different nature.

> ...as a result, it may be ideal to use different languages for each. These languages would each be oriented (i.e restricted) to their specific goal – there is no sense in having control specification primitives in a language for specifying state.

The recommended architecture is shown below:



Essential state is the foundation of the system. It makes no reference to any of the other parts. Essential logic is the *beta's* of the system (sometimes termed the 'business logic'). It expresses what must be true but does not say anything about how, when, or why the state might change dynamically. (We could also describe this as the system *invariants* to be true is back to the concepts we were looking at yesterday). The logic-specification references only the essential state. Accidental state and control is the third component, and changes in it can never affect the other two.

> The key difference between what we are advocating and existing approaches (as embodied by the various styles of programming language) is a high level separation into three components – each specified in a different language. It is this separation which is the key to the power of the approach. Each individual component, and it is this use of restricted languages which is vital in making the overall system easier to comprehend. Doing this separation when building a system may not be easy, but we believe that for any large system it will be significantly less difficult than dealing with the complexity that arises otherwise.

For specifying state the authors recommend the relational model, and the relational algebra for specifying manipulations. "Integrity in the relational model is maintained simply by specifying – in a purely declarative way – a set of constraints which must hold at all times.

This is augmented by functional programming, to give "Functional Relational Programming (FRP)."

> FRP is currently a purely hypothetical approach to system architecture that has not in any way been proved in practice. It is a combination of the ideas of Codd's relational model, functional and logic programming which have been widely proven. Its FRP-ness comes from the form of relations, and the essential logic is expressed using relational algebra extended with pure user-defined functions

Essential state is a relational definition of the stateful components of the system. Essential logic contains derived relation definitions, integrity constraints, and pure functions. The essential state and control components contains a declarative specification of a set of performance optimizations for the system.

The system components convert inputs into relational assignments – i.e. cause changes to the essential state. Observer components generate output in response to changes which they observe.

> At a minimum, observers will only need to specify the name of the relation which they wish to observe. The infrastructure which runs the system will ensure that the observer is invoked (with the new relation value) whenever it changes.

The concluding part of the paper is a worked example of a system to support a real estate business

Dedalus, Bloom, and Edelweiss

It's interesting to compare the vision set out in "Out of the Tar Pit" with some of the research undertaken by Joe Hellerstein and his team at Berkeley

- In The Declarative Imperative we see the imperative to adopt a declarative approach grounded in relational logic, and the creation of a datalog derivative called Dedalus for this purpose
- In Consistency analysis in Bloom: A CALM and collected approach we see how Bloom, based on Dedalus, can be used as a separated embedded language to specify essential state and logic
- And in Edelweiss we see the pursuit of a clean separation between essential and accidental complexity that can significantly simplify Bloom programs leaving the programmer to specify the desired outcomes, and Edelweiss to worry about making achieving them efficient.

Share this:

Twitter LinkedIn Email Print

Loading...

Related

On understanding software agility – a social complexity point of view
November 7, 2013
In "Software Engineering"

Distant Island Papers: Jonas Bonér
March 29, 2006
In "Desert Island Papers"

The Engineer's Toolchest
September 7, 2006
In "Software Engineering"

PROGRAMMING LANGUAGES SOFTWARE ENGINEERING

PREVIOUS
Distributed databases in Stephen Hawking's universe

NEXT
Finding out the hard way

4 thoughts on "Out of the Tar Pit"

Pat Deduck
MARCH 30, 2006 AT 9:56 AM

This was very interesting. I personally like the (beginning of) an approach as to why functional programming has not been widely adopted:

I think there are a few key issues, perhaps I will follow up with an article myself on it.

1. State reflects the real world. I think a major reason OOP and imperative in general, are so popular is because they model the real world very well. Real world objects have state. Like a vending, etc. cash have state – dosing, packed. But not here, etc., diffuse have state. It is far easier to conceptualize implementing real world processes in software when they simply reflect our understanding of the real world. Very little or no physical world is "functional" with inputs and outputs and no state.

2. "Recursion is bad". OK, it isn't necessarily but we are taught it is, in university courses and books and online. Heck, most courses and books teach you how to do recursion, immediately because, as they say, using the stack is an terribly expensive and inefficient compared to modifying mutable state in a memory with iteration.

3. Recursion is hard. Yes, once you get it right, functional programming can be easier to understand and dramatically easier to test, especially for complex systems. Knowing that a func(a) will reflect nothing beyond its scope and return only a result, which is immutable (and therefore can be cached, etc.) is very powerful. However, assessing how to write a func(a(a)), and then successively is rarely found in most projects.

All of the above means non-functional languages have the more people who can do them. (simple analogy: lots of imperative exist, and even translation between libraries – it is easier to learn JS from Java or Ruby from C++ than Haskell from any of them). And therefore for more "soft-ware" – libraries, frameworks, app servers, monitoring services, IDEs, you name it – which makes it lower cost to use them, at least in the short term.

Jonathan Wiemers
AUGUST 31, 2010 AT 2:41 PM

Very old comment, but I just wanted to add my own comment "the real world has state". I'd argue that it depends on what granularity we look at things. If we retain someone in the bar we see state. But if we stand above someone at OS feet, and watch their day – we see the processes they engage in. Flows even, and functional programming represents these flows much easier.

That's really all I had to say, and I think when talking about that level of abstraction and flows, functional programming helps us represent them better (more explicitly and cleanly) than (our current version of) OOP.

Angry Old and Expendable Coder
APRIL 29, 2020 AT 9:56 AM

We hear

> 1). "Recursion is bad". OK, it isn't necessarily, but we are taught it is, in university courses and books and online. Heck, most courses and books teach you how to do recursion, immediately, because, as they say, using the stack is an terribly expensive and inefficient compared to modifying mutable state in a memory with iteration."

Immediately below the university hears the books and the children outside that peddle this crap. You cannot get rid of recursion, however you try. At best you end up with a pseudo-stack that looks like a basket but that you then have to manage yourself. I remember an episode from the 90s where folks were implementing recursion in COBOL, maintaining stack inside DB2 because that was all the excessively recursive data structure to manage and the language – wouldn't allow it.

Also, we have had compilers that can do tail call optimization to transform apparently recursive calls into iteration since Chomo, Schwartz ? Compiler?. which was 1960? Yeah, Java doesn't have it. Life is great.

Pingback: Software Engineering and Human Nature | Atomic Energy
Pingback: Shared Mutable State – SimaDream
Pingback: Stuff The Internet Says On Scalability For March 16th, 2020 – Amazon Web Services Feeds

Comments are closed.

BLOG AT WORDPRESS.COM