Depending Less on Structure

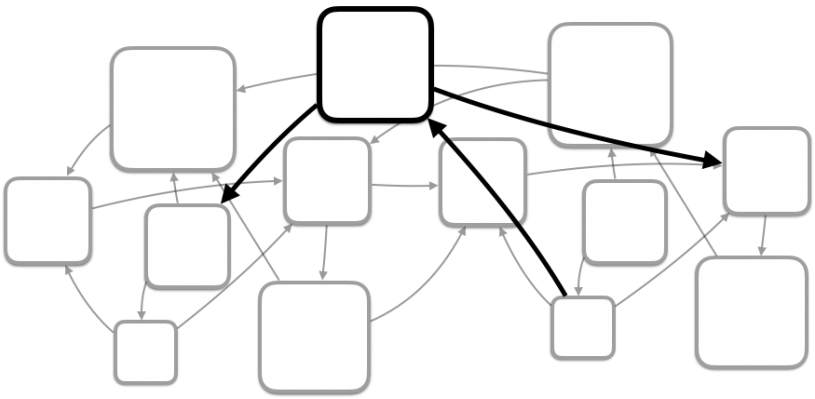
Luca Matteis, Dec 10 '19

Most of software development best-practices, architectures and patterns of the last few decades are essentially intricate ways of creating **structures** that allow humans to more easily understand and change what the software does.

Building software works similarly to constructing buildings: we carefully have to put pieces together in a sort of lego fashion.

The analogy of a lego building suits quite well since whenever one needs to modify the central parts of a lego structure they need to carefully take it apart and make sure the new pieces fit well with their surroundings.

The lego pieces are our modules, components, functions or objects that when plugged together in a specific way give rise to the actual intended behavior of our programs.



I'd argue that the issue of changing software stems from a deeper and more fundamental way of how programming is done: mainly that **software needs to adhere too rigorously to structure**.

Let's take a look at a simple program that takes as input a number x and decides whether it is a multiple of 3:

```
const x = readInput();
if (isMultipleOfThree(x)) {
  return true;
} else {
  return false;
}
```

Now let's imagine that we want to change this program to "also check whether it ends with the digit 5". To do this we can simply change our if statement to include

this check:

```
  const x = readInput();
- if (isMultipleOfThree(x)) {
+ if (isMultipleOfThree(x) && endsWithDigitFive(x)) {
    return true;
  } else {
    return false;
  }
```

I think the very fact that we had to make this modification to integrate this change is key to understanding why legacy code is so hard to change.

But this is crazy talk... how can we make changes to a program without doing what we just did? What kind of sorcery am I talking about?

Let's rewrite the program above using a sort of new "language" with different execution semantics. It looks like this:

```
const x = sync({ waitFor: 'input' })
if (isMultipleOfThree(x)) {
  sync({ request: 'good', waitFor: 'bad' })
} else {
  sync({ request: 'bad', block: 'good' })
}
```

Of particular importance are these `sync()` calls that allow a module to *peek* at other modules and control their execution.

When we run this program and we feed an event such as `input(6)` we get this output:

```
input(6)
good
```

and if we feed it a number that isn't multiple of 3 we get:

```
input(7)
bad
```

Nothing surprising. Let's try to implement the same change we did earlier to "also check whether it ends with the digit 5". Instead of changing the code we just wrote, **we'll write a new module** that looks like this:

```
const x = sync({ waitFor: 'input' })
if (endsWithDigitFive(x)) {
  sync({ request: 'good', waitFor: 'bad' })
} else {
  sync({ request: 'bad', block: 'good' })
}
```

This new module **will run in parallel** with the other one. Both modules run symmetrically. They both wait for input events. Whenever the `sync` function is called the two modules peek at each-others declarations.

For instance IF they both reach the second sync call:

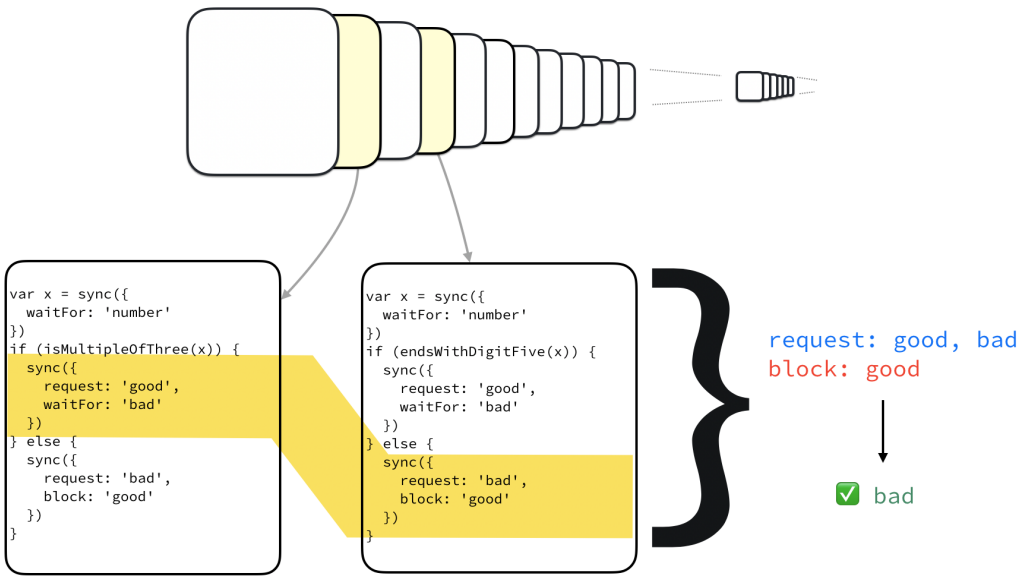
```
if (isMultipleOfThree(x)) {
  sync({ request: 'good', waitFor: 'bad' })
...
if (endsWithDigitFive(x)) {
  sync({ request: 'good', waitFor: 'bad' })
```

They are both requesting the `good` event hence that's what the program will output.

IF one of them is in another state such when the number ISN'T a multiple of 3 and it ends with 5, they'll find each-other at this sync point:

```
if (isMultipleOfThree(x)) {
  sync({ request: 'good', waitFor: 'bad' })
} else {
  sync({ request: 'bad', block: 'good' }) // <-- at this state
}
...
if (endsWithDigitFive(x)) {
  sync({ request: 'good', waitFor: 'bad' }) // <-- at this state
} else {
  sync({ request: 'bad', block: 'good' })
}
```

At this point the first module is requesting `bad` and the other is requesting `good`. Who will win? Because the first module is also blocking the `good` event, this makes the `bad` event win. Hence the program will output `bad`.



The semantics around how `request`, `waitFor` and `block` work are bit intricate and that's not the point of this article. For now we can think of them as a simple way to control whether other threads can continue their execution.

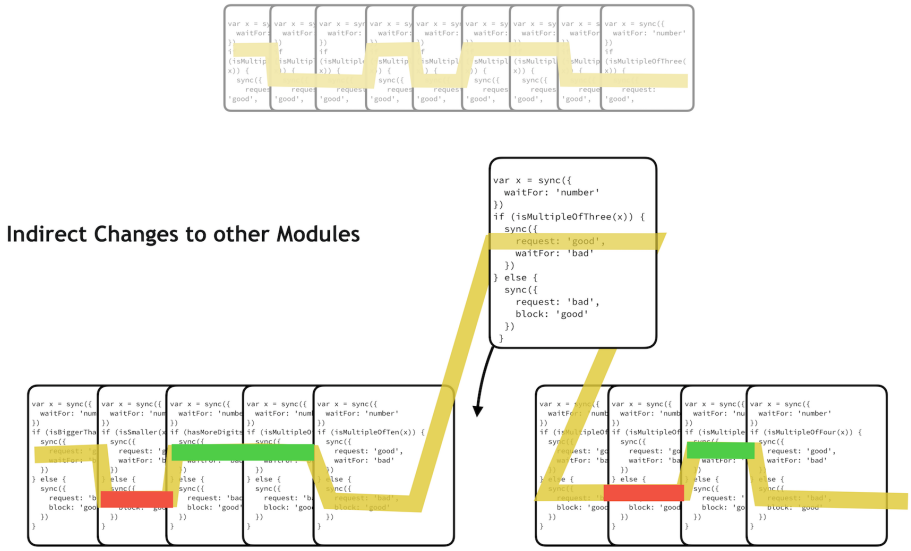
Having modules control the execution of other modules without direct communication is the key insight to this approach.

Integrating changes is where complexity lies

You might be asking: what's the point of programming this way using these sync calls, and these request/waitFor/block events?

Intuitively we just introduced a change to a program, albeit a simple one, without having to do any integration work.

Whereas before we had to write integration logic such as `&& endsWithDigitFive(x)` in order for our change to work, in this new system we simply had to create a new module that did exactly what we intended. Both modules could be swapped out without them knowing of each-other and without having to do any integration work.



This is a huge deal.

You might ask: but even with this new system we'll eventually have to modify and refactor existing modules based on the new change.

Indeed, but the change will be about enriching modules with semantics that allow them to collaborate better as a whole (such as waiting or blocking new events) rather than having to integrate or glue together parts of the modules to make them aware of how other modules work. Key difference is: **there is no direct communication between modules**. They are always oblivious about each-other.

But my pure functions are also oblivious of each-other

Pure functions are just input→output and in this context they are also written in a way that they are unaware of each-other.

For instance let's look at a simple data-transformation operation using `pipe`:

```
pipe(
  getName,
  uppercase,
  get6Characters,
  reverse
)({ name: 'Buckethead' })
// 'TEKCUB'
```

The problem is that these functions still have a point of communication: the point where they're used (aka the point of integration).

The difference is subtle but in my opinion crucial to understanding why the problem of integration will continue haunting developers for years to come.

Let's make this a little more concrete and discuss a change to the flow above regarding "reversing the name before it gets the first 6 chars". Obviously this is yet again a simply change. But what if we continue discussing the change where "the uppercase should only happen if the name is capitalized" and "reverse should only be done after successfully getting data from an API".

Things are getting a bit more hairy and complicated and yet only resemble a tiny and minimal version of the requirements that usually come up in real-world scenarios.

By not communicating directly these requirements seem less intimidating to implement: for instance a new module could be swapped-in to pause execution of the reverse operation once the API successfully responds without modifying existing code.

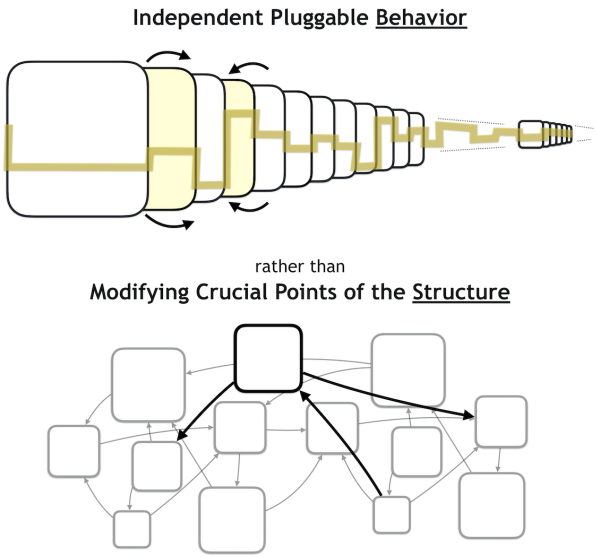
Behavioral Programming

This new method of executing programs is actually something that exists and is called Behavioral Programming.

We can enhance or refine a system by simply adding modules, similarly to how one can enhance a requirements document by adding clarifications, refinements and exceptions in the form of new sentences in the body of the document or as independent appendices and footnotes.

As goals are refined and requirements added to a program, or when bugs appear, rather than enhancing and often complicating existing modules, we strive to add new modules that precisely address the difference, or the gap, between the goals and the what the existing system accomplishes.

Modules can be thought of as books on a shelf that can be easily swapped out and back in, rather than lego pieces that might crumble or complicate existing structures.



Changing software you don't understand

But how does all this help with the infamous question of changing legacy code?

Intuitively a program written this way allows us to observe specific traces and swap-in and out modules to implement a change without having to deeply understand the structure of the program: because the changes don't depend on the structure but on the combined behavior of the modules.

For instance in a complex legacy program we might need to implement a requirement:

```
Given the user doesn't have a promo code
When the user adds an item to the shopping cart
  And is the first Monday of the month
Then they should not be able to add more than 3 items to the cart
```

In the common integration-style way we'd have to alter and somewhat complicate the modules that are responsible for these changes.

In this new Behavioral Programming style we can instead map these changes quite naturally to new modules that can be swapped into the program without touching or even seeing how the system works.

Which brings us to a new point: programming this way is more aligned with requirements:

```
promoCode = sync({ waitFor: 'promoCode' })
if (promoCode) return;
sync({ waitFor: 'itemAddedToCart' })
if (isFirstMondayOfMonth()) {
  sync({ waitFor: 'itemAddedToCart' })
  sync({ waitFor: 'itemAddedToCart' })
  // only 3 items max!
  sync({ block: 'itemsAddedToCart' })
}
```

Multi-Modality

In addition to the fact that these type of modules are loosely-coupled, as they depend less on the structure, they also allow us to describe the behavior of our software using three main modalities: things that **may** happen, things that **must** happen and more importantly things that **must not** happen. This is in contrast to most contemporary programming approaches, which are usually of a single modality (do this) often guarded by conditions, etc.

Specifying what may happen will provide the system with options and possibilities for things to execute (`request`), and specifying what must be done (`request` with high priority) and what may not be done (`block`) will constrain these options.

Imagine being able to specify what is not allowed to happen, using `block`, even before the logic that generates such behavior is written:

```
// Prevent manual updates
sync({ block: 'manualUpdate' })
```

In the code above we are preventing a user from manually doing updates. We can write this piece of code at any stage of the development process; even at the very beginning; even before the code that triggers `manualUpdate` is written.

Again this is in contrast with conventional programming that depends on structure and hence doesn't allow us to specify undesired behavior **before** the logic that leads to such behavior is written. I urge the reader to stop and think about this for a second as it's a quite crucial difference:

How would you write the logic for "preventing manual updates" using conventional programming? You'd have to find where in the *structure* the `manualUpdate` event is triggered and conditionally trigger that event. Instead what we are doing in the earlier snippet is quite different: we are blocking the event `manualUpdate` from happening **even before the logic that triggered it was written**.

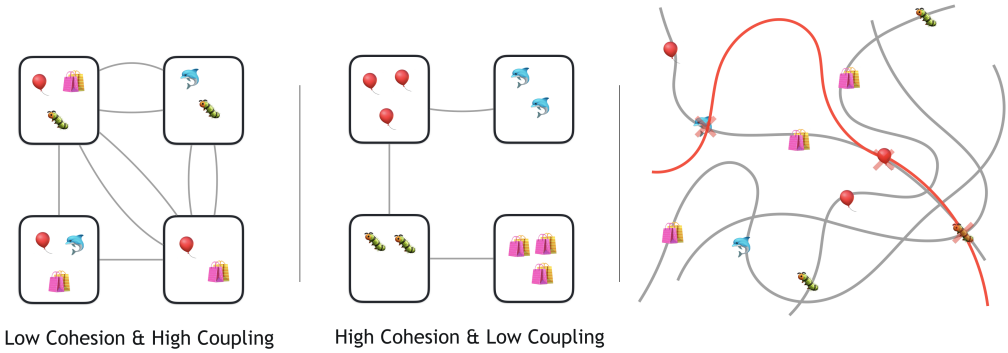
Imagine the learning potential and creative power of a human who is allowed to freely experiment with a variety of behaviors, except those that are forbidden (e.g., the illegal, expensive, or risky ones), figuring out if and when any of allowed actions produces valuable results

Conclusion

Obviously this does not mean that by programming this way we do not have to think about good software development practices. But I do believe it makes it easier and less daunting to make changes to complex systems: you can swap out and back in new modules based on the changes needed rather than having to modify crucial points of integration within the structure of the program.

However I think that this approach comes with some cost. For instance `if (isMultipleOfThree(x) && endsWithDigitFive(x))` is very easy to understand, but `isMultipleOfThree(x)` and `endsWithDigitFive(x)` in different modules is a lot harder. Personally I see this more as a trade-off: **trading control flow for extensibility**.

In this approach, modularity is not necessarily achieved by the structure, but can be done by behaviours. You don't have to think of your system's behaviour as being "chopped up" into objects or tasks or components; you can chop it up any way you want according to the way you like to think about the behaviour.



Low Cohesion & High Coupling

High Cohesion & Low Coupling

This quote from David Harel, one of the originator of this approach, really is a nice way of thinking about Behavioral Programming. I tried depicting the main differences using lines or "threads" of behavior that overlap, intersect, replace or extend other behaviors, rather than boxes with arrows that need to rigorously dictate the structure.

One amazing read that really made me change my entire attitude towards software development is The quest for runware: on compositional, executable and intuitive models. A whole lot of this article was inspired by this.

A final insight that I'd like to end with is the fact that no matter how intricate our solutions to software development are, we are still limited by the way our **brain** works, hence finding solutions that align to our way of thinking are probably going to be the most interesting approaches. From the runware article:

> *There is apparently no modification of existing initial memories -- no insertions, no cut-and-paste -- only more and more experiences. Images seen, sentences heard, pain felt, are all amassed as new memories and connected to existing memories in more ways than we can imagine today. Some of these, of course, explain, refine, correct, reorganize, or completely replace things that were previously experienced (or seen or heard or read) in how they affect future behavior.*

If you're interested in learning more on how to program this way using these modules (formally called b-threads) there is a lot of practical research on the subject, it's not just theoretical. Simply searching on google scholar for "harel behavioral programming" will point you in a good direction.