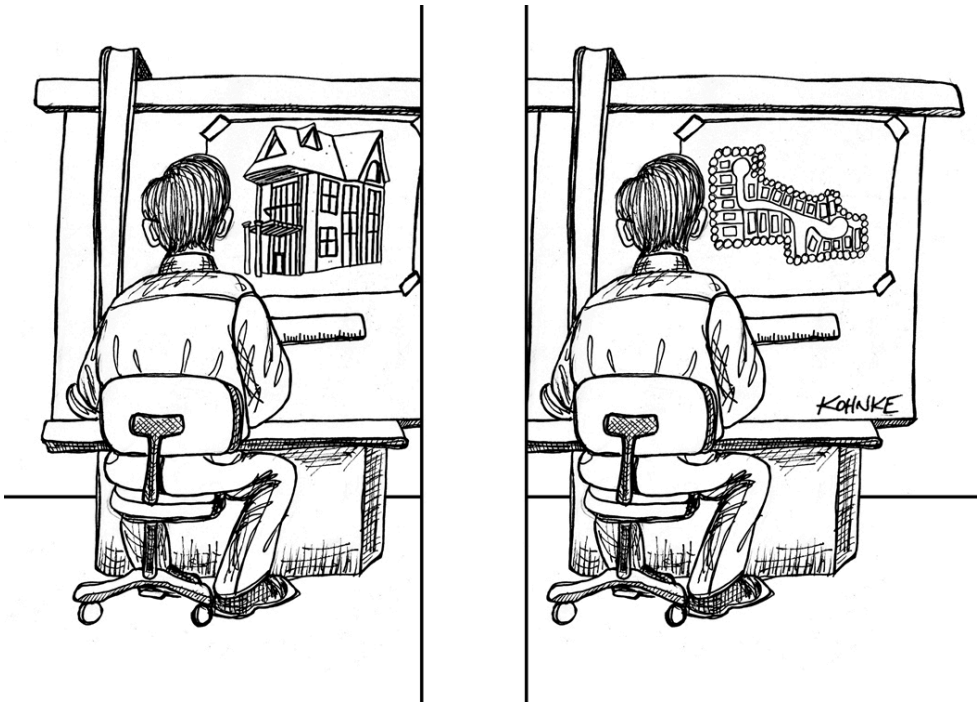
25

Architectural Boundaries



Software architecture is the art of drawing lines that I call *boundaries*. Those boundaries separate software elements from one another, and restrict those on one side from knowing about those on the other. Some of those lines are drawn very early in a project's life—even before any code is written. Others are drawn much later. Those that are drawn early are drawn for the purpose of deferring decisions for as long as possible, and of keeping those decisions from polluting the core business logic.

Remember that the goal of an architecture is to minimize the manpower required to build and maintain the required system. What is it that saps manpower? *Coupling*; and especially, coupling to premature decisions.

What kinds of decisions are premature? Decisions that have nothing to do with the business requirements—the use cases—of the system. These include decisions about frameworks, databases, Web servers, utility libraries, dependency injection, and the like. A good system architecture is

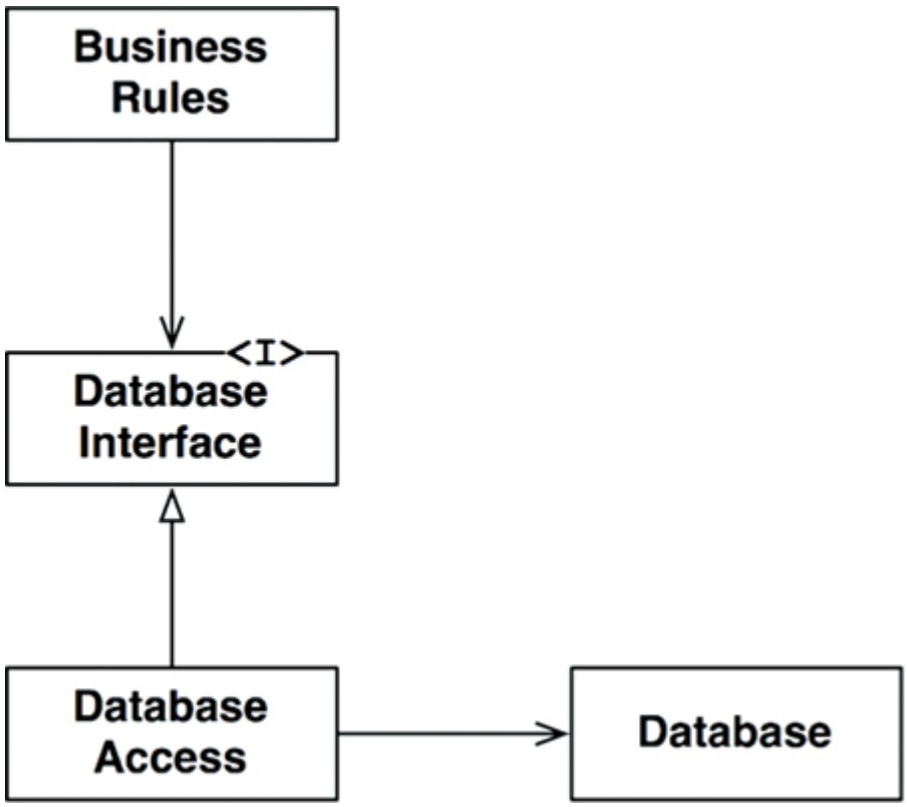
one in which decisions like these are rendered ancillary and deferrable. A good system architecture does not depend on those decisions. A good system architecture allows those decisions to be made at the latest possible moment, without significant impact.

What Lines Do You Draw, and When?

You draw lines to separate things that matter from things that don't. The GUI doesn't matter to the business rules, so there should be a line between them. The database doesn't matter to the GUI, so there should be a line between them. The database doesn't matter to the business rules, so there should be a line between them.

Some of you may have rejected one or more of those statements; especially the part about the business rules not caring about the database. Many of us have been taught to believe that the database is inextricably connected to the business rules. Some of us have even been convinced that the database is the embodiment of the business rules.

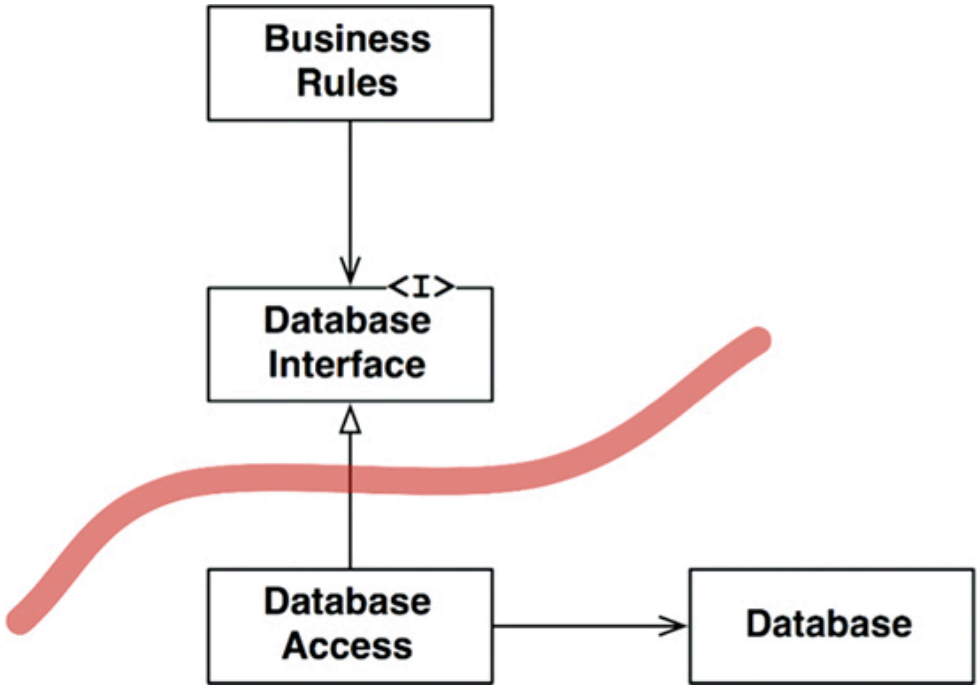
But this idea is misguided. The database is a tool that the business rules can use indirectly. The business rules don't need to know about the schema, or the query language, or any of the other details about the database. All the business rules need to know is that there is a set of functions that can be used to fetch or save data. This allows us to put the database behind an interface.



You can see this clearly in the diagram above. The `BusinessRules` use the `DatabaseInterface` to load and save data. The `DatabaseAccess` implements the interface and directs operation of the actual `Database`.

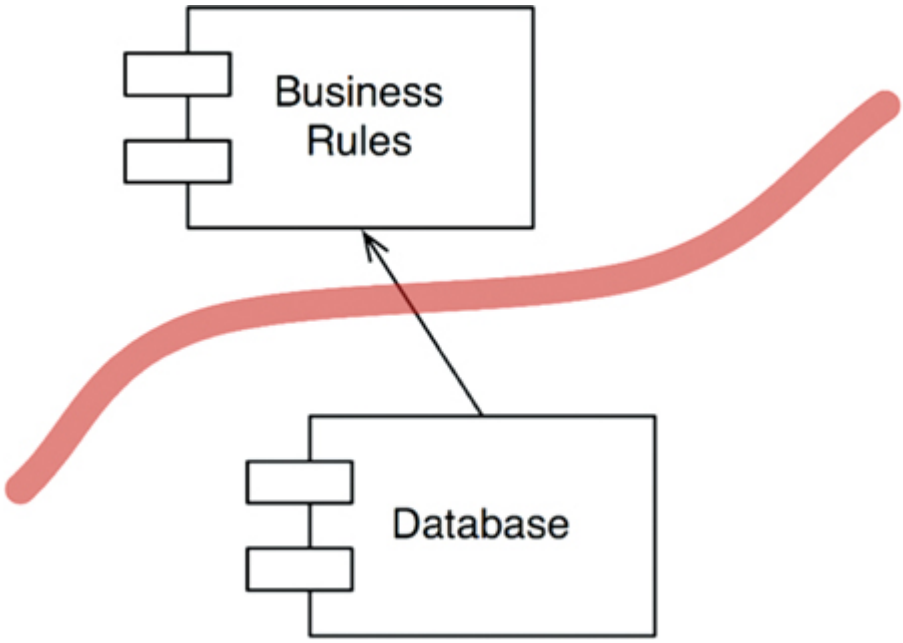
The classes and interfaces in this diagram are symbolic. In a real application, there would be many business rule classes, many database interface classes, and many database access implementations. But they would all follow roughly the same pattern.

Where is the boundary line? The boundary is drawn across the inheritance/implements relationship, just below the `DatabaseInterface`, as shown in the following diagram.



Note the two arrows leaving the `DatabaseAccess` class. Those two arrows point away from the `DatabaseAccess` class. That means that none of these classes knows that the `DatabaseAccess` class exists.

Now let's pull back a bit. We'll look at the component that contains many business rules, and the component that contains the database and all its access classes.



Note the direction of the arrow. The `Database` knows about the `BusinessRules`. The `BusinessRules` do not know about the `Database`. This implies that the `DatabaseInterface` classes live in the `BusinessRules` component, and the `DatabaseAccess` classes live in the `Database` component.

The direction of this arrow is important. It shows that the `Database` does not matter to the `BusinessRules` . But the `Database` cannot exist without the `BusinessRules` .

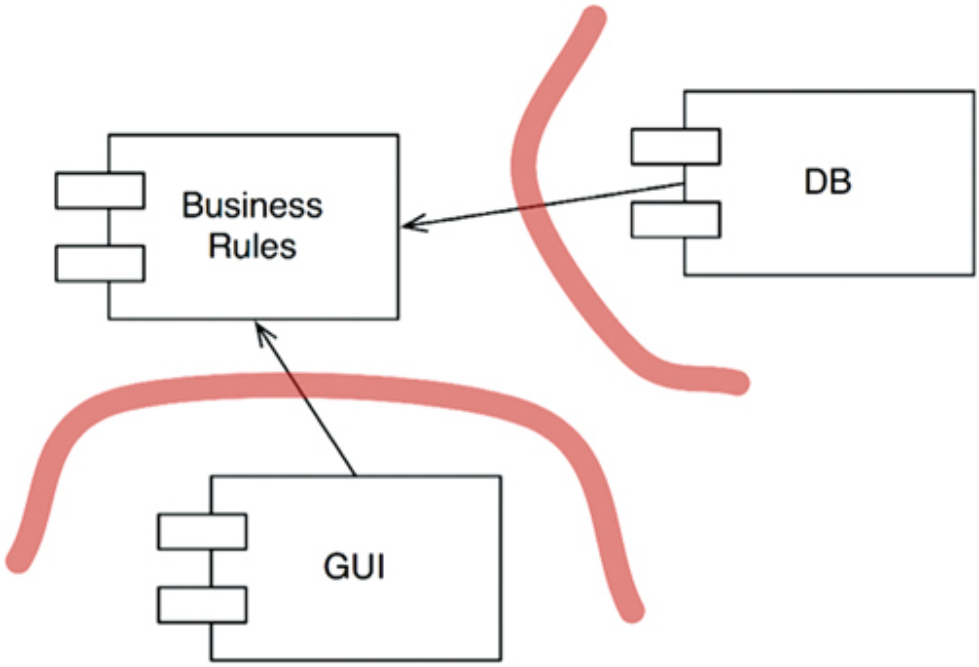
If that seems strange to you, just remember: That `Database` component contains the code that translates the calls made by the `BusinessRules` into the query language of the database. It is that translation code that knows about the `BusinessRules` .

Having drawn this boundary line between the two components, and having set the direction of the arrow toward the `BusinessRules` , we can now see that the `BusinessRules` could use *any* kind of database at all. The `Database` component could be replaced with many different implementations. The `BusinessRules` don't care.

The database could be implemented with Oracle, or MySQL, or Couch, or Datomic, or even flat files. The business rules don't care at all. And that means that the database decision can be deferred and you can focus on getting the business rules written and tested before you have to make the database decision.

Plug-in Architecture

Look at the two boundaries in the diagram below. They separate the `Database` and the `GUI` from the `BusinessRules` . The arrows point from the lower-level components to the high-level `BusinessRules` component.



The `GUI` and `DB` are plug-ins to the BusinessRules. This is a pattern that can be repeated for many different kinds of components. The core business rules can be kept separate from, and independent of, any components that either are optional or can be implemented in many different forms.

So, because the user interface in this design is considered to be a plug-in, we have made it possible to plug in many different kinds of user interfaces. They could be Web based, client/server based, SOA based, console based, or based on any other kind of user interface technology.

The same is true of the database. Since we have chosen to treat it as a plug-in, we can replace it with any of the various SQL databases, or a NoSQL database, or a file system–based database, or any other kind of database technology we might deem necessary in the future.

These replacements might not be trivial. If the initial deployment of our system was Web based, then writing the plug-in for a client/server UI could be challenging. It is likely that some of the communications between the `BusinessRules` and the new UI would have to be reworked. But by starting with the presumption of a plug-in structure, we have, at the very least, made such a change practical.

Case Study: FitNesse

My son, Micah, and I started work on FitNesse in 2001. The idea was to create a simple wiki that wrapped Ward Cunningham's `FIT` tool for

writing acceptance tests. We wrote FitNesse in Java.

I was adamant that anything we produced should not require people to download more than one `jar` file. I called this rule *Download and Go*. This rule drove many of our decisions.

One of the first was to write our own Web server, specific to the needs of FitNesse. This might sound absurd. Even in 2001 there were plenty of open source Web servers that we could have used. Yet writing our own turned out to be a really good decision because a bare-bones Web server is a very simple piece of software to write, and it allowed us to postpone any Web framework decision until much later.[1]

---

1. Many years later we were able to slip the Velocity framework into FitNesse.

Another early decision was to avoid thinking about a database. We had MySQL in the back of our minds, but we purposely delayed that decision by employing a design that made the decision irrelevant. That design was simply to put an interface between all data accesses and the data repository itself.

We put the data access methods into an interface named `WikiPage`. Those methods provided all the functionality we needed to find, fetch, and save pages. Of course, we didn't implement those methods at first; we simply stubbed them out while we worked on features that didn't involve fetching and saving the data.

Indeed, for three months we simply worked on translating wiki text into HTML. This didn't require any kind of data storage, so we created a class named `MockWikiPage` that simply left the data access methods stubbed.

Eventually, of course, those stubs were insufficient for the features we wanted to write. We needed real data access, not stubs. So we created a new derivative of `WikiPage`, named `InMemoryPage`. This derivative implemented the data access methods to manage a hash table of wiki pages, which we kept in RAM.

This allowed us to write feature after feature for a full year. In fact, we got the whole first version of the FitNesse program working this way. We could create pages, link to other pages, do all the fancy wiki formatting, and even run tests with `FIT`. What we couldn't do was save any of our work.

So, when it came time to implement persistence, we thought again about MySQL, but we decided that wasn't necessary in the short term because it was really easy to write the hash tables out to flat files. So we implemented `FileSystemWikiPage`, which just moved the functionality out to flat files, and then we continued developing more features.

Three months later we reached the conclusion that the flat-file solution was good enough, and we decided to abandon the idea of MySQL altogether. *We deferred that decision into nonexistence, and never looked back.*

That would be the end of the story if it weren't for a customer of ours who decided that he needed to put the wiki into MySQL for his own purposes. So we showed him the architecture of `WikiPages` that had allowed us to defer the decision. He came back a day later with the whole system working in MySQL. He simply wrote a `MySqlWikiPage` derivative and got it working.

We used to bundle that option with FitNesse, but nobody else ever used it, so eventually we dropped it. Even he eventually dropped it.

Early in the development of FitNesse, we drew a boundary line between business rules and databases. That line prevented the business rules from knowing anything at all about the database, other than the simple data access methods. That decision allowed us to defer the choice and implementation of the database for well over a year. It allowed us to try the file system option. And it allowed us to change direction when we saw a better solution. And yet it did not prevent, nor even impede, the original direction, MySQL, when someone wanted it.

The fact that we did not have a database running for 18 months of development meant that, for 18 months, we did not have schema issues, query issues, database server issues, password issues, connection time issues, or all the other nasty issues you get when you fire up a database. It

also meant that all our tests ran fast, because there was no database to slow them down.

In short, drawing the boundary lines that helped us delay and defer decisions saved us an enormous amount of manpower and headaches. And that's what a good architecture should do.

Conclusion

This is how you draw boundary lines in a software architecture. You partition the system into components. Some of those components are core business rules. Some are plug-ins that contain necessary functions that are not directly related to the core business. Then, you arrange the code in those components such that the arrows between them point in one direction —toward the core business.

You should recognize this as an application of the Dependency Inversion Principle (DIP) and the Stable Abstractions Principle (SAP). Dependency arrows are arranged to point from lower-level details to higher-level abstractions.

And that is the Dependency Rule of Architecture.

Dependencies that cross architectural boundaries must always point toward the higher-level side.