LayerCake

Software Made Better

Jeff Vroom
jeff@jvroom.com
www.layercake.lc

I'd like to share with you some ideas I have for making software better and introduce the project I'm developing called LayerCake.

This Talk

- Declarative languages
- State of the art of good software
- Inefficiencies with the state of the art
- LayerCake
- Demo

In this talk

I'll introduce what I mean by "declarative languages"

show how they are important to state-of-the-art software

discuss areas of inefficiency that exist even in today's best software patterns.

introduce LayerCake and show how it improves the state of the art

followed by a demo

Declarative Languages

- Declarative: tree of things with properties, expressions (e.g. XML, HTML, CSS, Excel)

- No statements, loops, program counter

The most significant advances in software development efficiency in my career have come from doing more in declarative languages and less in code.

By declarative language, I mean a format which supports a hierarchical collection of things with properties, sometimes expressions like XML, HTML, CSS, and Excel.

There are no statements executed, loops etc. so these formats are much easier to use.

State of the Art Software Design

Declarative Languages	**Java/.NET**	**PHP/Scripting**
HTML UI	*SP/*MVC + CSS	*PHP Template
Components	Inversion of control: Spring/Unity	None or various
Persistence	ORM: JPA/nHibernate	SQL/ActiveRecord
Rules + workflow	Various: XML	None
Services + messaging	Various: Spring/WCF	JSON
Content Management	Day/Sitecore	Wordpress, Drupal

This chart shows my take on the best patterns used for each of the main sub-systems of an enterprise application in today's state of the art. What jumps out to me is that each sub-system uses some type of declarative language. You implement user interfaces via declarative template pages that are mostly HTML. Configure components, persistence, services and messaging via XML or code annotations. Use high level tools and formats to let business users manage rules, workflow, and content.

More Good Patterns

- Keep domain model separate, contained
- Inherit and override for customization - not copy
- Rich business user tools: editing, staging, review, publishing: “empower the organization, not IT”

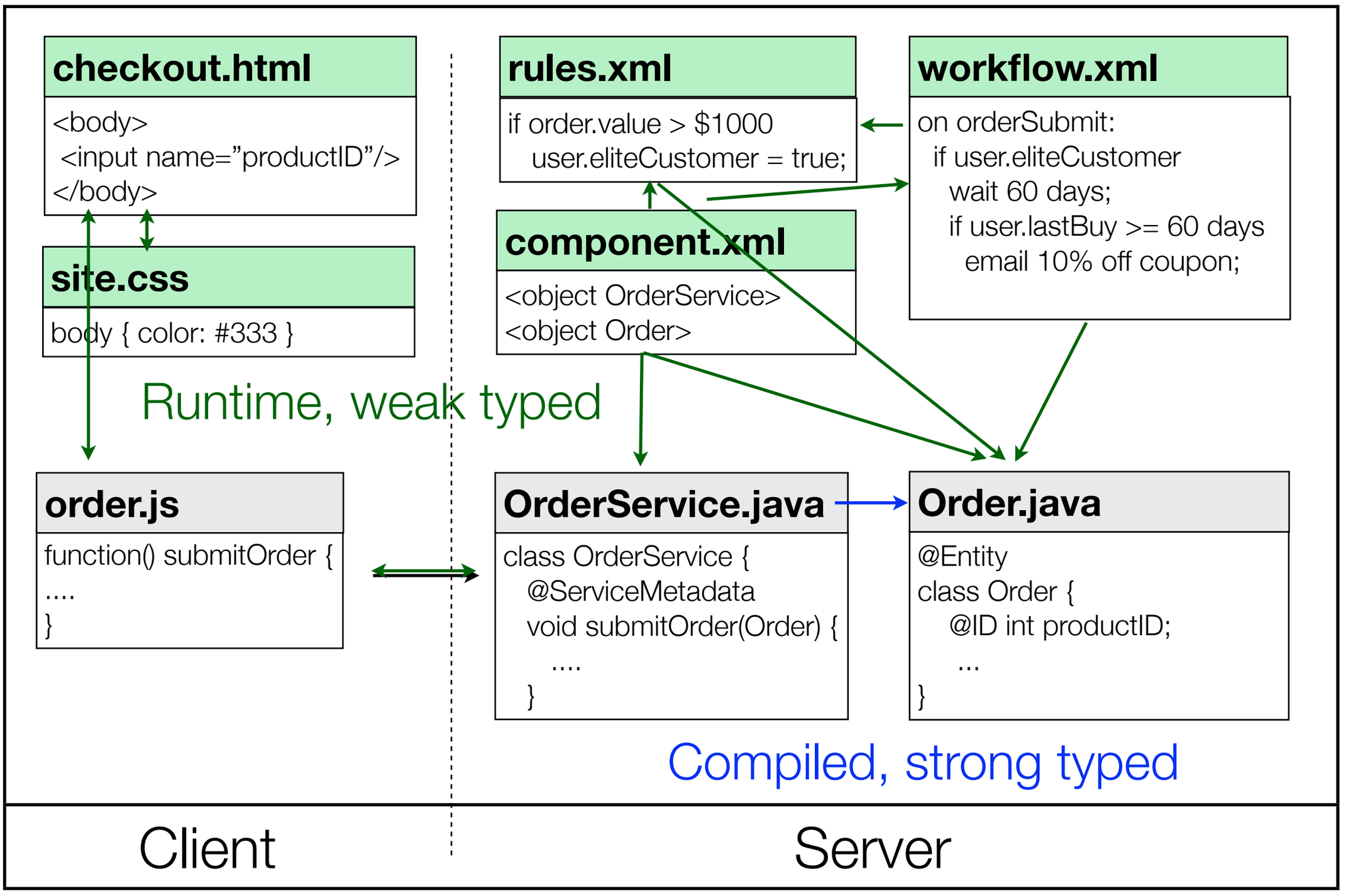
Let’s look at some more good patterns.

The best systems keep the domain model separate and do not replicate that logic in multiple places.

They use inheritance and override APIs for customization and do not rely on copies.

They provide rich business user tools which empower the organization, not IT. These tools should support source control and collaboration features like diffs, sync, rollback, merge.

State of the Art Mockup - The Good and Bad



Here's a mockup of state of the art software.

It's good that key files are declarative, exposing UI, rules, components, and workflow to declarative programmers and tools.

And good that annotations allow the persistence and service layers to avoid creating copies of the domain model for each of those layers. But bad that the domain model is somewhat split and replicated between the client, the component framework, the service layer, and the core model itself and tied explicitly to the persistence implementation.

And bad that most files use runtime, weakly typed connections. These connections lack integrity checks and tooling for managing references such as name-completion, edit-time errors, navigation and refactoring. Instead you find errors the first time you test a page or worse no error at all for a bad reference - just the wrong result. At runtime, each lookup takes hundreds of machine instructions if not thousands. <pause> In contrast, compiled, strongly typed connections fix these integrity problems and compile down to a single load-from-offset instruction. But in the state of the art, only the Java code uses them.

<next>

More Problems With Declarative Languages

- Many incomplete programming languages
- Code paths are not traceable
- Customization needs inheritance, variables, expressions
- Complex and often weak tools

Let’s look at some more problems with declarative languages.

They are also frequently missing the required expressiveness to do what you need. Then you are forced to generate the declarative file from code losing the benefits. Logic is spread out across file types making code harder to read. And there are so many file types it may be that no one on the team is an expert in them all.

Code paths are not traceable via debuggers.

And they are missing important language features for customization, particularly inheritance. And when you don’t have inheritance, the only way to customize is to copy files, and merge later changes made to the originals by hand.

The tools tend to be complex, incomplete, and lack necessary hooks into your development environment for versioning, staging, deployment, and A/B testing.`

Coding Inefficiencies

- Code copied across javascript, server, mobile, etc.
- Long restarts
- Predict customization needs
- Code customization hooks explicitly
- Design for modularity
- Compatibility between releases: APIs and declarative formats

With today's state of the art, programmers also face challenges with their code which cause engineering inefficiencies.

The platform matrix appears to be expanding, not shrinking requiring copies of code to be made across client, server, and mobile.

Frameworks frequently parse lots of declarative files at startup resulting in long restarts, particularly as projects get very large slowing development.

Programmers must predict how their software will need to be customized and

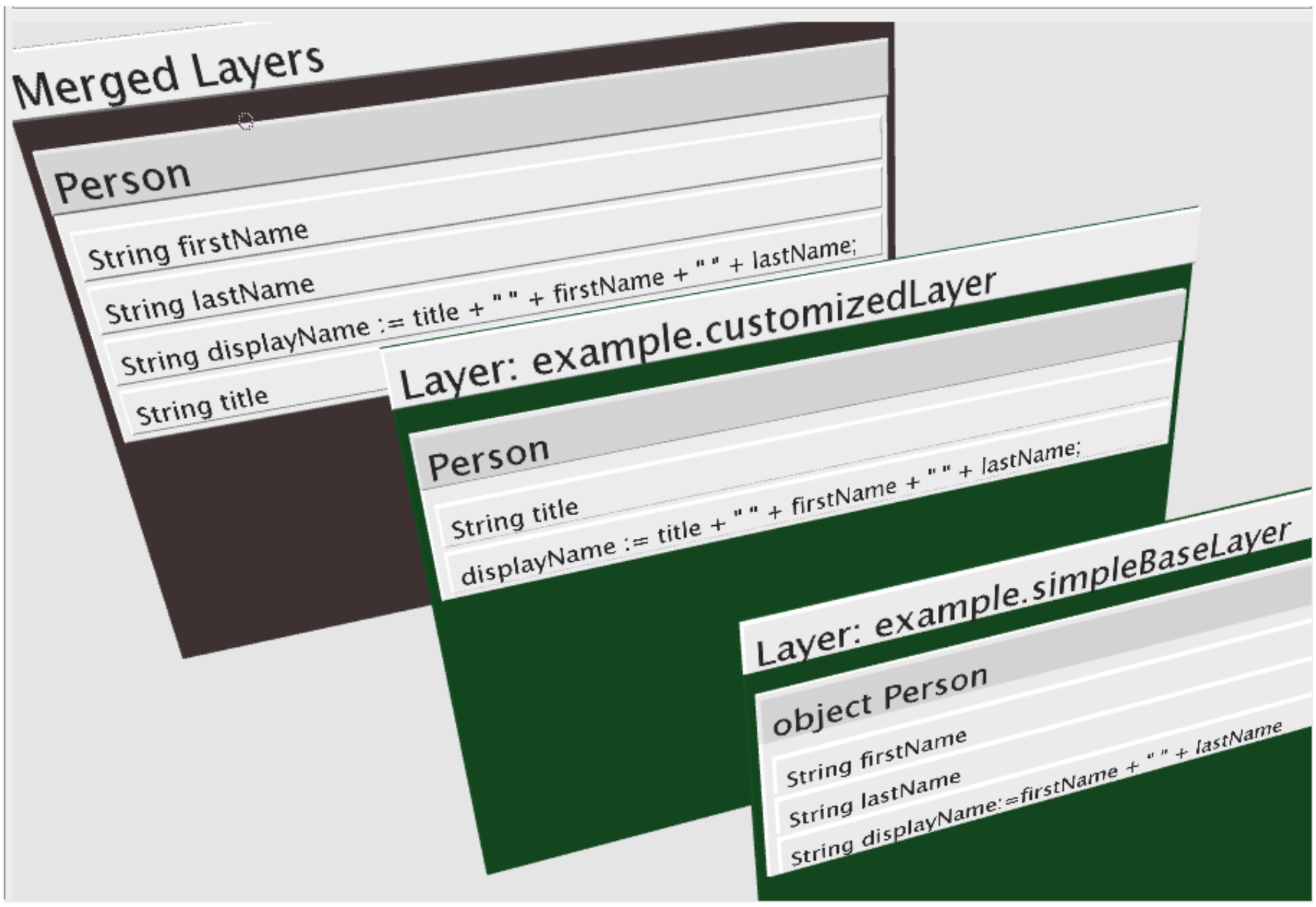
they must write explicit code for customization hooks.

At the same time, they must design their code to be modular and

ensure compatibility between releases, not only to the code out there but for all of the declarative formats which use their code.

LayerCake

- Layers: like photoshop image layers for code



- Cake: make coding more declarative, faster and easier

I believe LayerCake improves the state of the art.

It generalizes the concept of object oriented inheritance with a new construct called layers.

If you've used photoshop, you are probably familiar with how you edit your image in layers. Photoshop automatically merges them to build the final image but manages each layer independently. LayerCake uses a similar concept to let you organize your code in layers. It automatically merges them to build the version which is executed. Instead of image location, it uses a tree of names and files to do the merging, similar to Java's classpath but where files may also be merged.

In addition to layers, LayerCake adds a few features to make coding more declarative, faster, and easy as, well, cake.

Layers: A Missing Core Construct

- Customization built in
- Better dependency management
- Partition code by function (e.g. domain model, UI, app, style, rules) or project (e.g. next release)
- Slice up monolithic types: mix properties, rules, code
- Replace name schemes: UIUser, DBUser, UserService
- **Traceable code paths**

All words in both human and computer languages can be split into two types – the control words which form the structure of the language and the set of words we extend to describe things. I think that the construct of layers, which occurs commonly in nature and programming, is a control word we missed in computer languages.

With layered code, you can customize any aspect of any part of the system with no hooks required by the programmer. You can keep those changes separate and later merge them automatically.

I'll show how layers let you manage dependencies more naturally in your code with less up front design.

Layers let you partition code by it's function in the application or by it's role in the project.

Today some designs end up with massive classes which become hard to manage. If you refactor into smaller less natural types you create name space complexity and difficulty finding things. Layers let you find natural ways to organize code in complex types.

Today programmers use naming conventions like UIUser, DBUser, UserService to partition aspects of code but unlike layers, this does not let you manage the overlap which occurs between these different aspects.

Above all, layers add a tremendous amount of power without sacrificing code integrity. Unlike aspect oriented programming, you can trace all code paths and benefit from the same level of tooling as Java.

The Cake: Java, Faster, Easier, More Powerful

- More declarative: components, properties, data binding
- Mix dynamic and compiled code
- Framework for frameworks: build/run, mini-IDE, REPL
- Keep app code separate from framework
- Multiple inheritance
- Inherited imports, package, public

But LayerCake is way more than just Java with layers.

LayerCake contains just the right set of extensions which I've found are necessary for a complete declarative framework. It includes support for definition of object graphs with forward references and cycles, multi-stage initialization, get/set conversion and an intuitive and flexible data binding feature.

Dynamic layers provide excel-like interactivity for changes, even with large code bases. And compiled layers provide Java compatibility and performance.

It's a framework for frameworks doing all the heavy lifting with an easy-to-use, highly customizable UI for building programs.

Application code is kept separate from framework code so it is readable and reusable.

LayerCake adds multiple inheritance to Java and

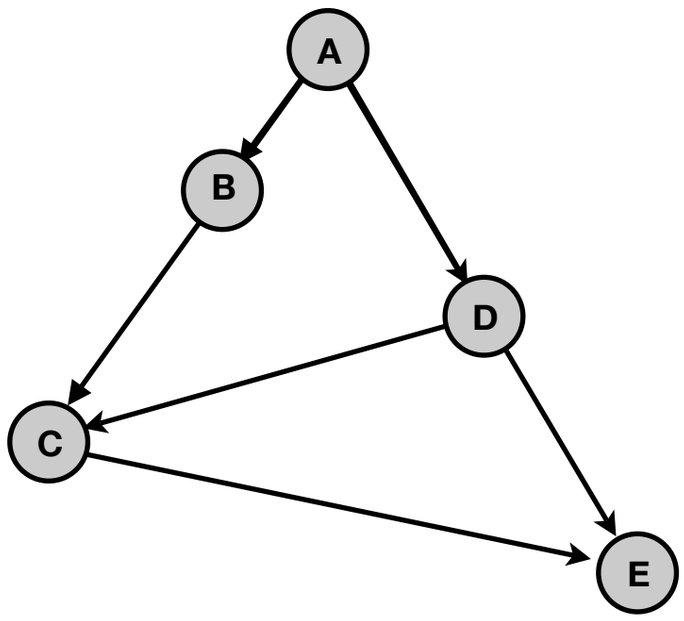
removes some of the verbosity and complexity of imports and protection access by specifying them per-layer.

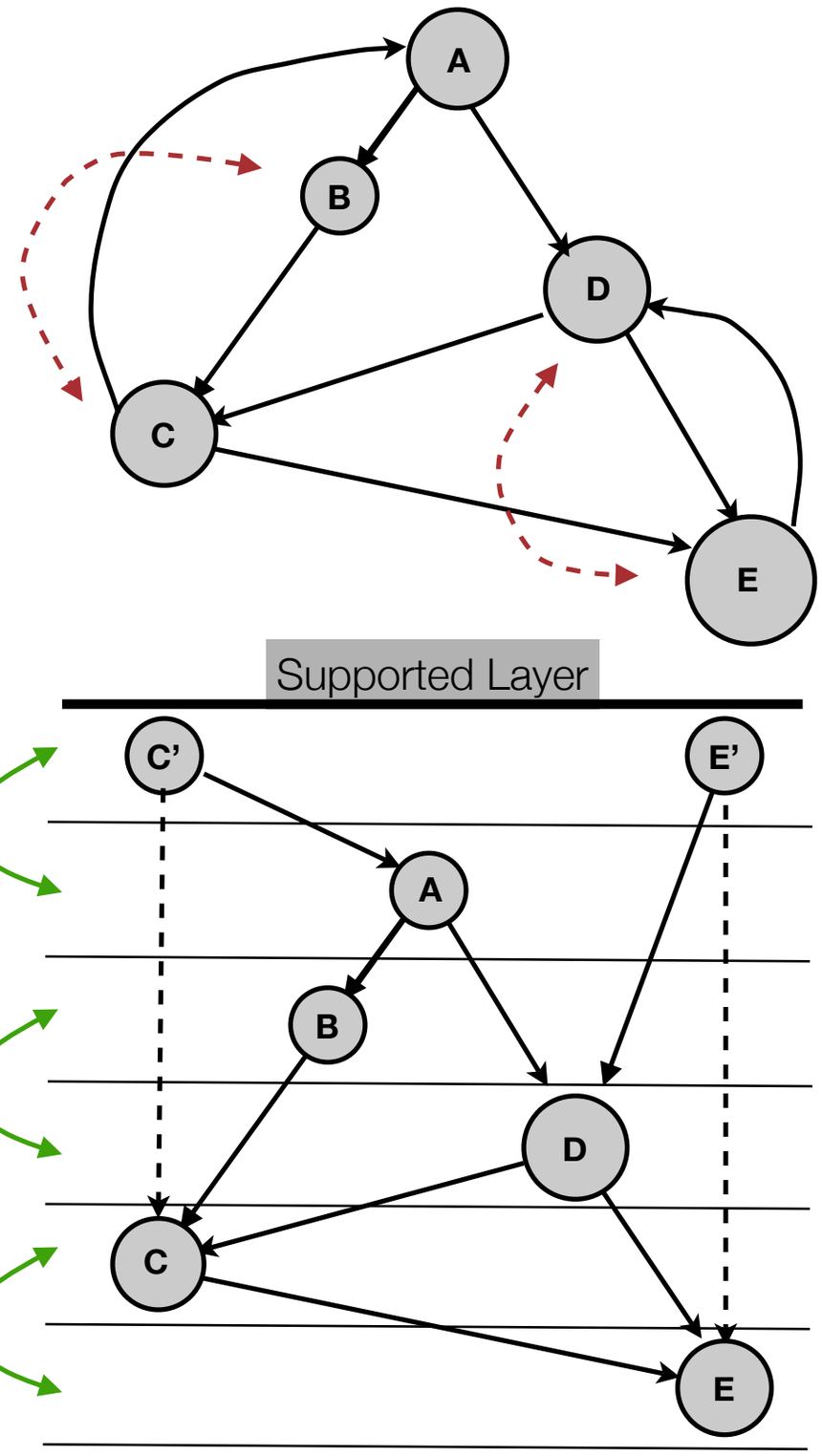
Dependencies: Modules vs Layers

Version 1 | Version 2

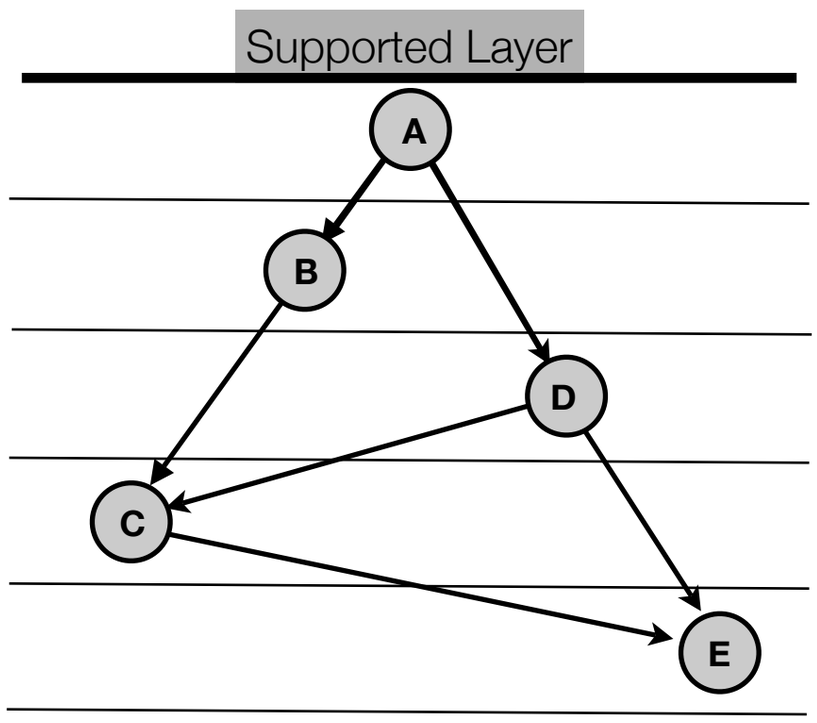
Modules evolve into hairballs* as systems grow

*inseparable: only testable/ deployable as one unit





Layered code separates features and naming from packaging to improve code independence



Let's look at how layers compare with modules in managing dependencies in your code.

If you are careful, you can maximize the amount of independent code as we see in Version 1. Independent code is tested in isolation and deployable as a separable unit, thus easier to reuse. But when you publish APIs to your code, you've tied modules to names.

In Version 2 those modules grow in complexity and minor features in one module start depending on minor features in some up-stream module. Yes, you could probably move code around but doing so breaks compatibility and is a lot of work changing names, files, and build scripts. Too often you accept the new dependencies as in Version 2 creating one much larger unit of code which must be tested and deployed in tandem.

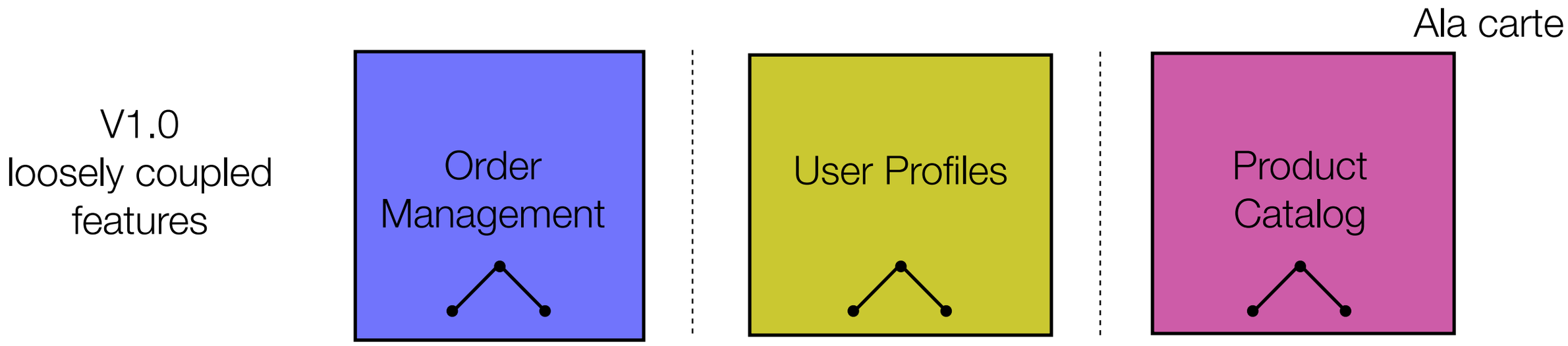
With layers the first version can be the same - use a layer for each module,

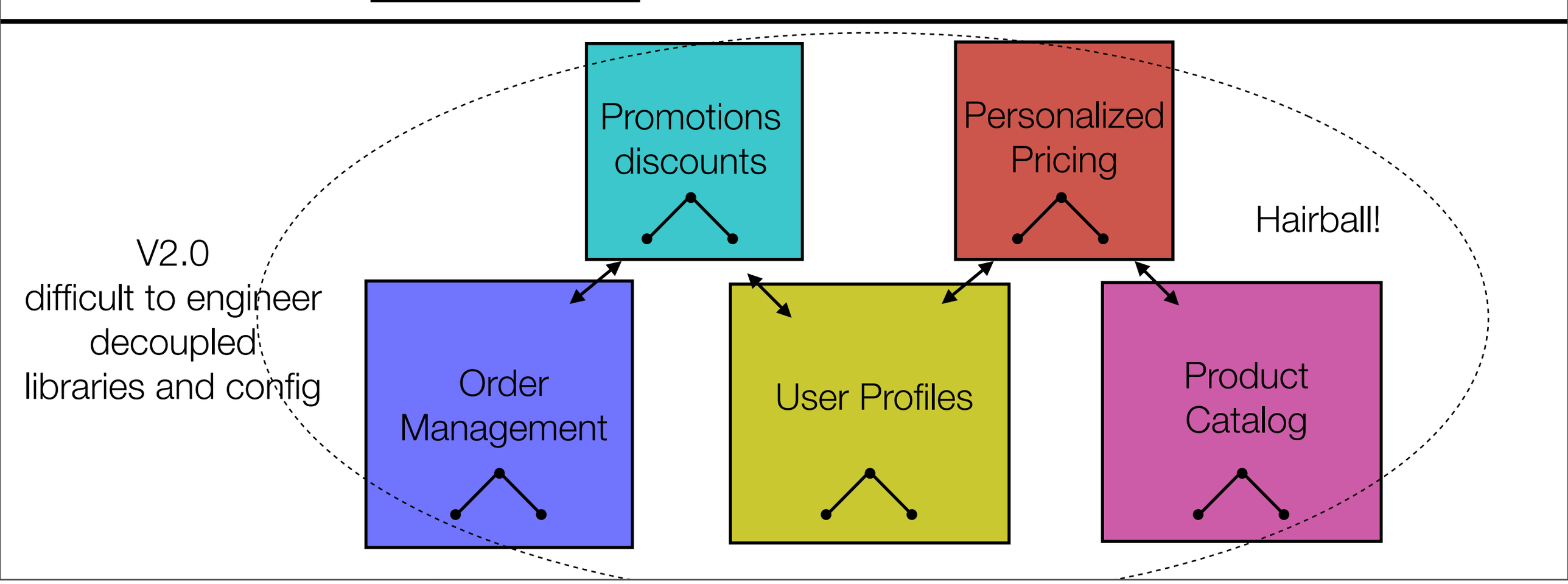
But in version 2, instead of introducing the cyclic dependencies, use the same names in a new layer to introduce the features which add those dependencies.

Give customers an empty supported layer which you can point at the new layer to give them these new features.

Unlike modules you are free to move code back and forth between layers and create new intermediate layers as long as the published APIs of the supported layers do not change. This reshuffling lets you reorganize the code based on dependency independent of how you originally packaged the code without breaking compatibility. And you still have the benefits of modularity for development, packaging, and testing the bulk of your code.

Real World Scenario - ECommerce Framework



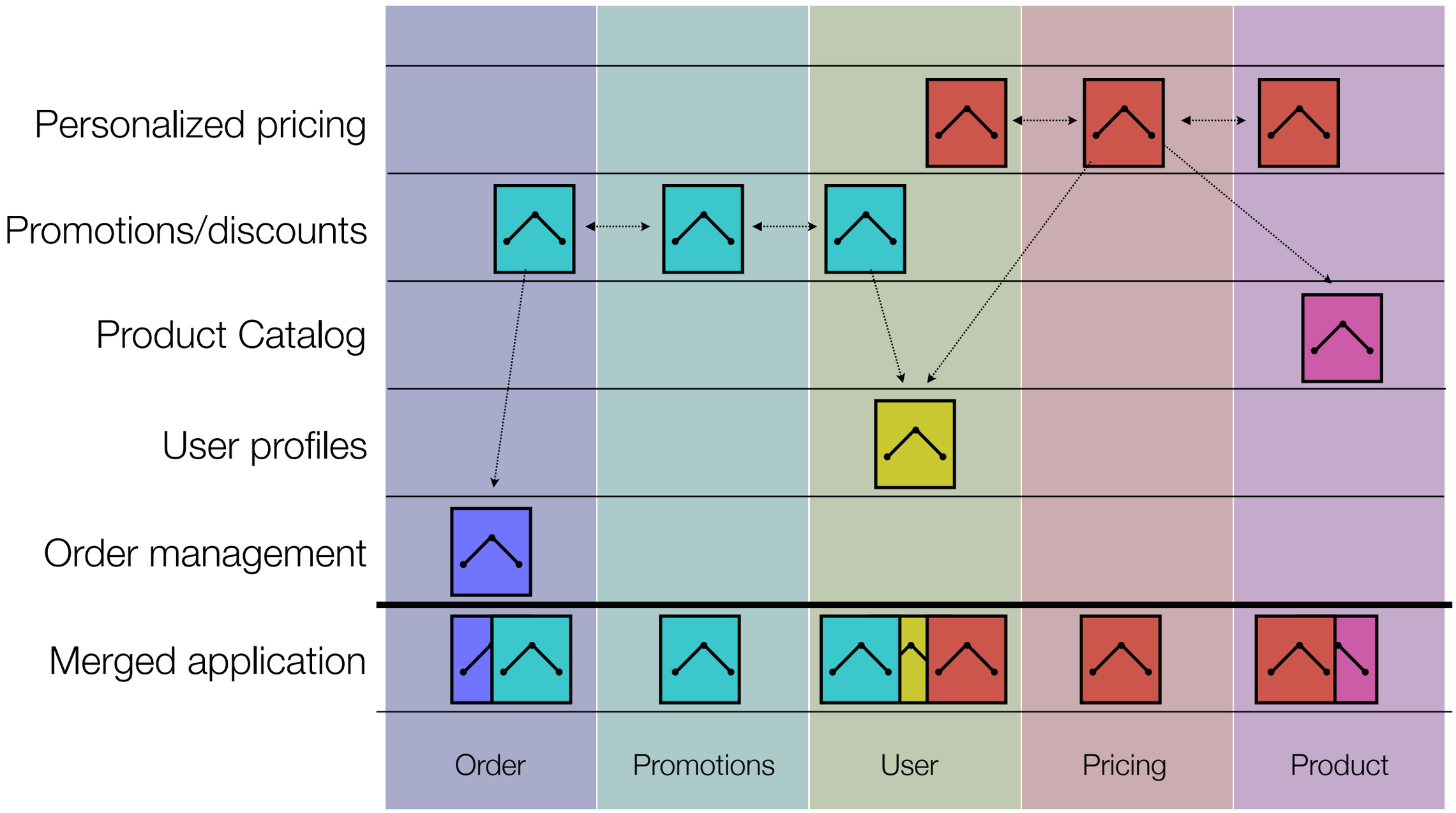


Here is a more concrete example.

Your first eCommerce framework shipped as three separate modules: order management, user profiles, and the product catalog. Sales wants them separate as people may only need one or more.

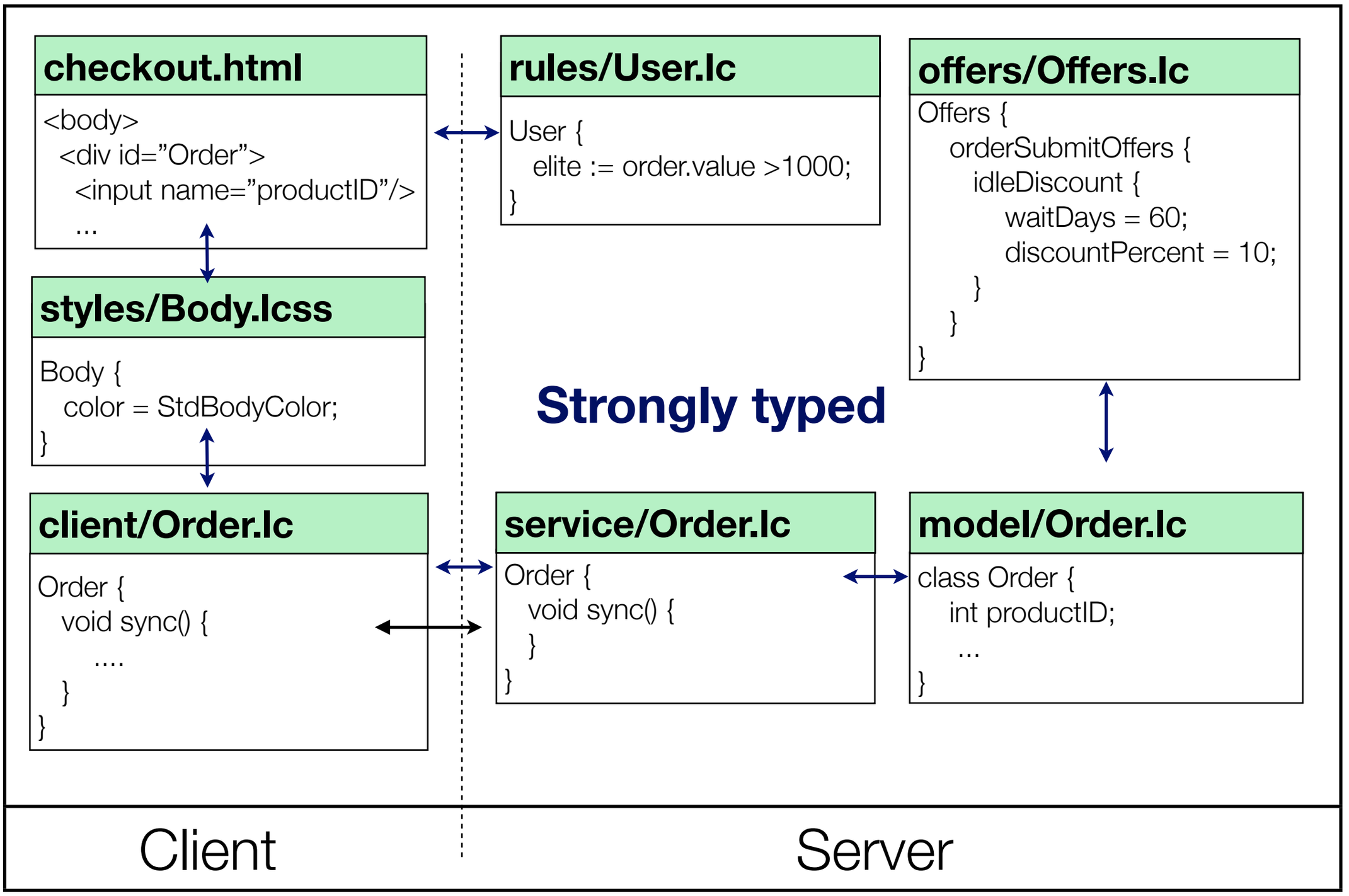
In 2.0 new features have been added: promotions and discounts requires adding information to the user profile and using that in business rules as part of order management. Similarly we need to customize product display based on who you are and so also need to tie the user profile to the product catalog. Engineering must work hard to keep the three independent code bases separate. Subclasses don't work.

ECommerce Framework With Layers



- With layers, you retain the independent versions of the order, user and product modules even in version 2.
- The promotions/discounts feature is added in a new layer which enhances the order and user objects with new information.
- Similarly the personalized pricing feature may add its own set of features to the user module as well as enhancing the product. Engineering benefits from more independent code and sales benefits from more packaging options.
- Note also that there are four different versions of the user object - clearly something you cannot do with simple subclasses.

LayerCake Mockup



Now let's look at the mockup from before in LayerCake.

Just as before UI, rules, and workflow are declarative. The component configuration is not necessary because layers let you customize any property after the fact and supports the other features component frameworks provide.

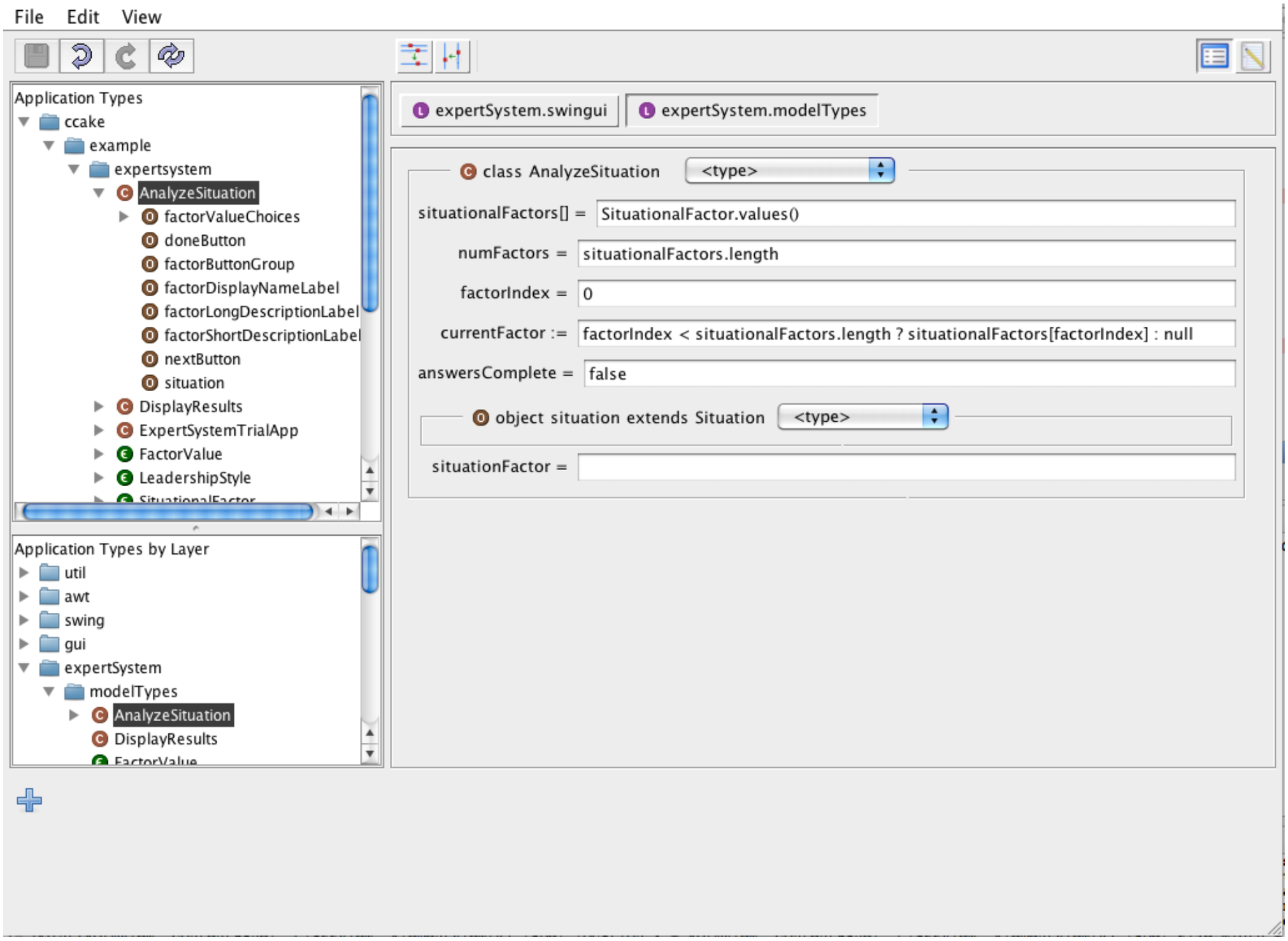
Now all are dynamic in development.

And all of the server files are compiled in production.

Even the client code benefits from strong typing and unified tooling of references, refactoring, etc.

The code can share the model layers between client and server and synchronize the overlap. The persistence layer (not shown) augments the model as needed.

LayerCake: Demo



Learn More

- Visit www.layercake.lc
- Get involved: partners, pilot projects, beta testers
- Contact me: jeff@jvroom.com