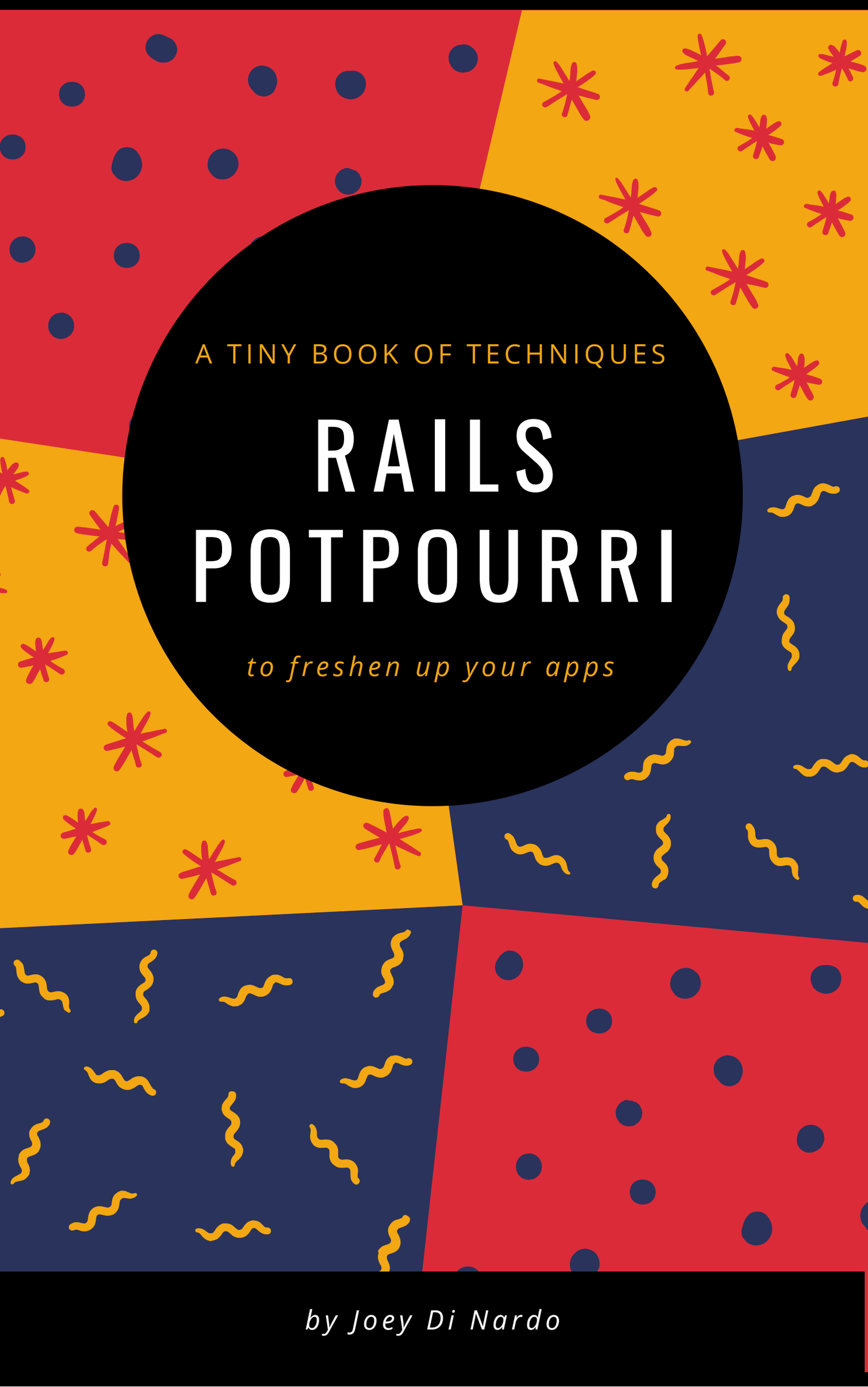

A TINY BOOK OF TECHNIQUES
RAILS POTPOURRI
to freshen up your apps
by Joey Di Nardo

Thanks for taking the time to read this little book. I know it's free, but I put a lot of time into it. The fact that you're willing to put the time into reading it means a lot to me.

Please feel free to share the book under the CC BY-NC-ND 4.0 license. If you found this book has provided you value please consider donating to the monks at one of the following Benedictine Monasteries:

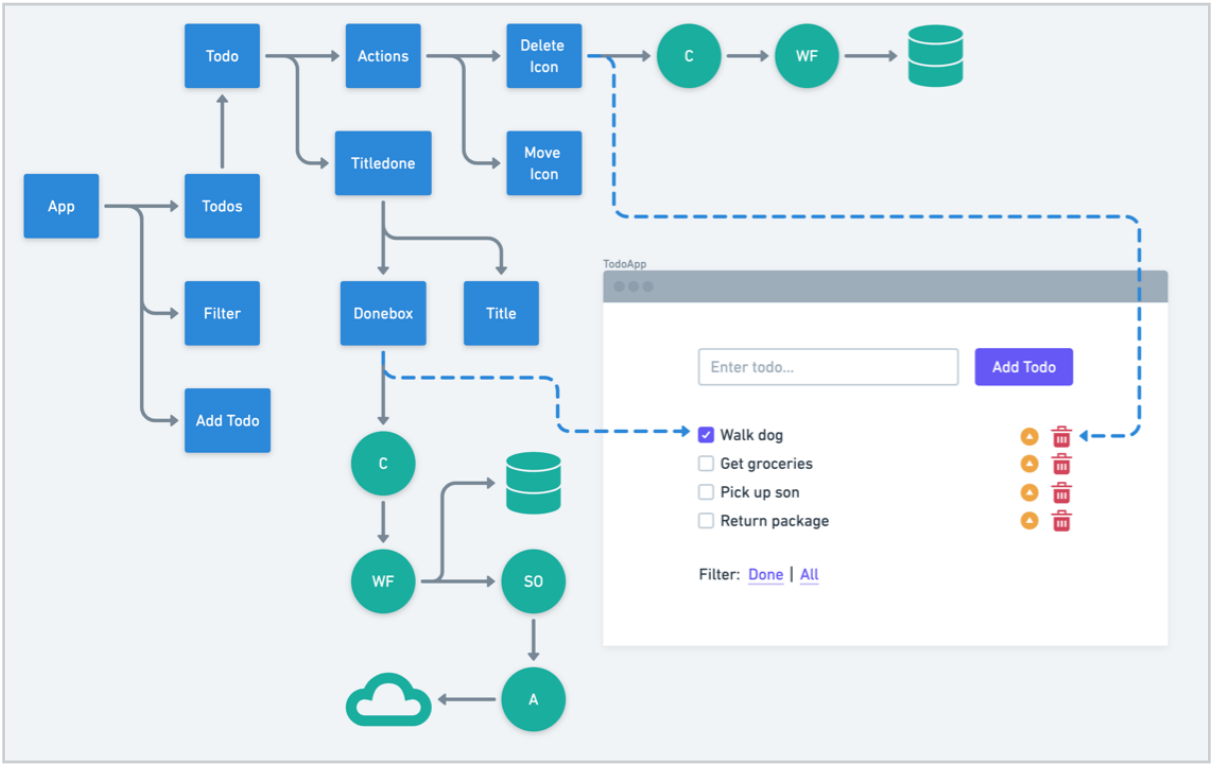
St. Vincent Archabbey in Latrobe, PA
St. Meinrad Archabbey in St. Meinrad, IN
Mepkin Abbey in Moncks Corner, SC

Sketching features

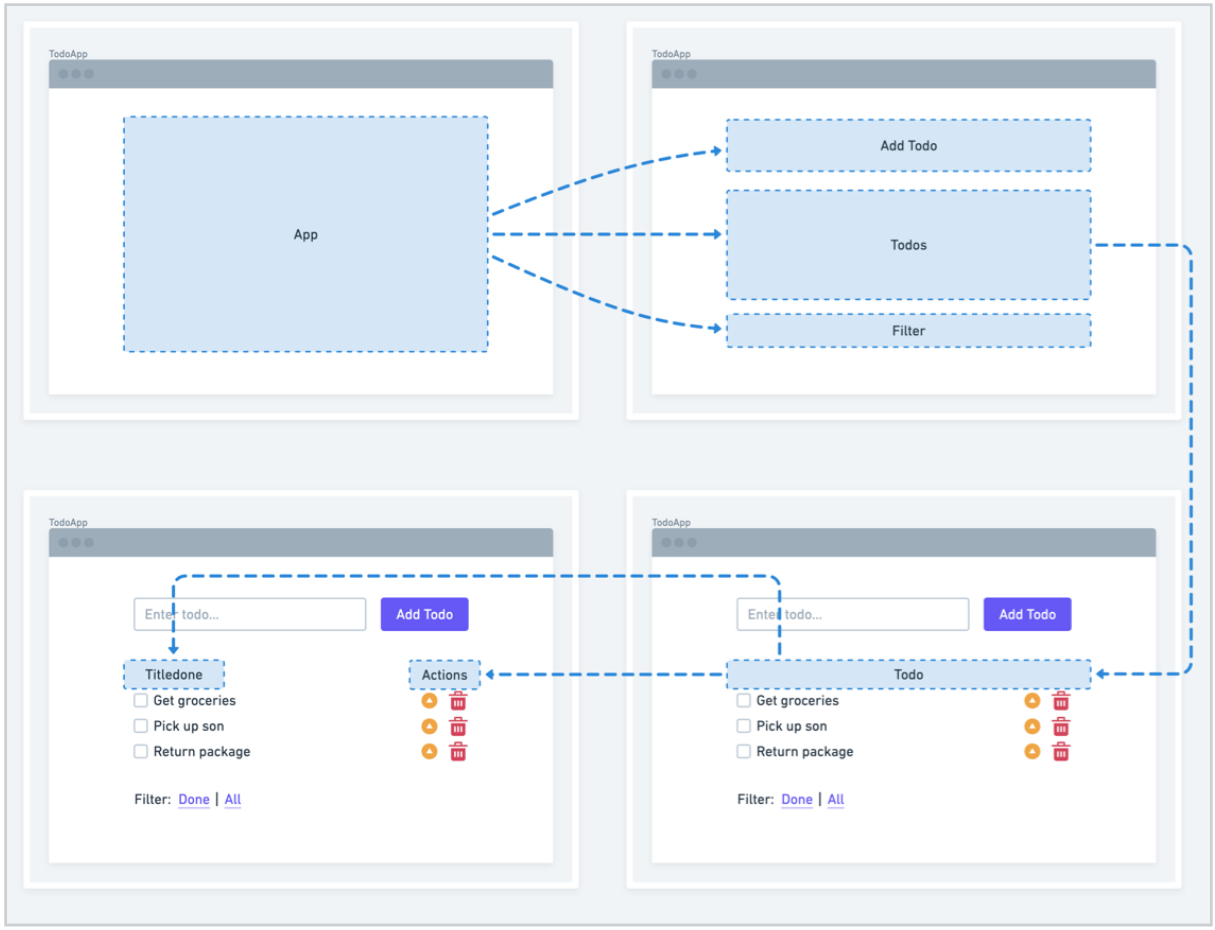
Shorthands are powerful. Whether it's Don Norman's wireflows or Ryan Singer's breadboard technique, shorthands quickly eliminate uncertainty in the problem domain and lead to new avenues of discovery. They also give the organization a shared language to describe different problems.

Most shorthands you'll encounter in the wild will fall into one of two categories: *structural* or *behavioral*, with the latter being far more popular. Wireflows and breadboards are behavioral. They "transition" from one node to another given some sort of impetus. For wireflows and breadboards, the impetus is user interaction. Structural diagrams show intrinsic relationships between parts of the system. They are time-independent.

I have two personal shorthands I like to use when sketching features, one structural and one behavioral. They plug in nicely to one another and help me create a solid mental model of the overall system.



The structural shorthand - indicated by the blue rectangles - is a shorthand for UIs. I find the components by looking for visual groups of elements. People order their environments (including UIs) according to the principles of Prägnanz, which makes agreeing on group boundaries a rather simple task, though there is room for disagreement about the granularity.



As I'm decomposing the wireframe into groups I give each group a name. Then I find the next group inside that group and give it a name. So on and so forth until I reach a level that I'm happy with. At this point every leaf in the tree has a name I use to communicate about that part of the UI with my design team or external stakeholders. When I begin to implement this feature I use these names usually as my CSS attributes, ID variables, or more recently as my ActionView::Components or Stimulus Controllers.

I then take each leaf and connect it to my behavioral shorthand - represented with the green circles. Each letter in the circle represents an acronym, so for the above flow:

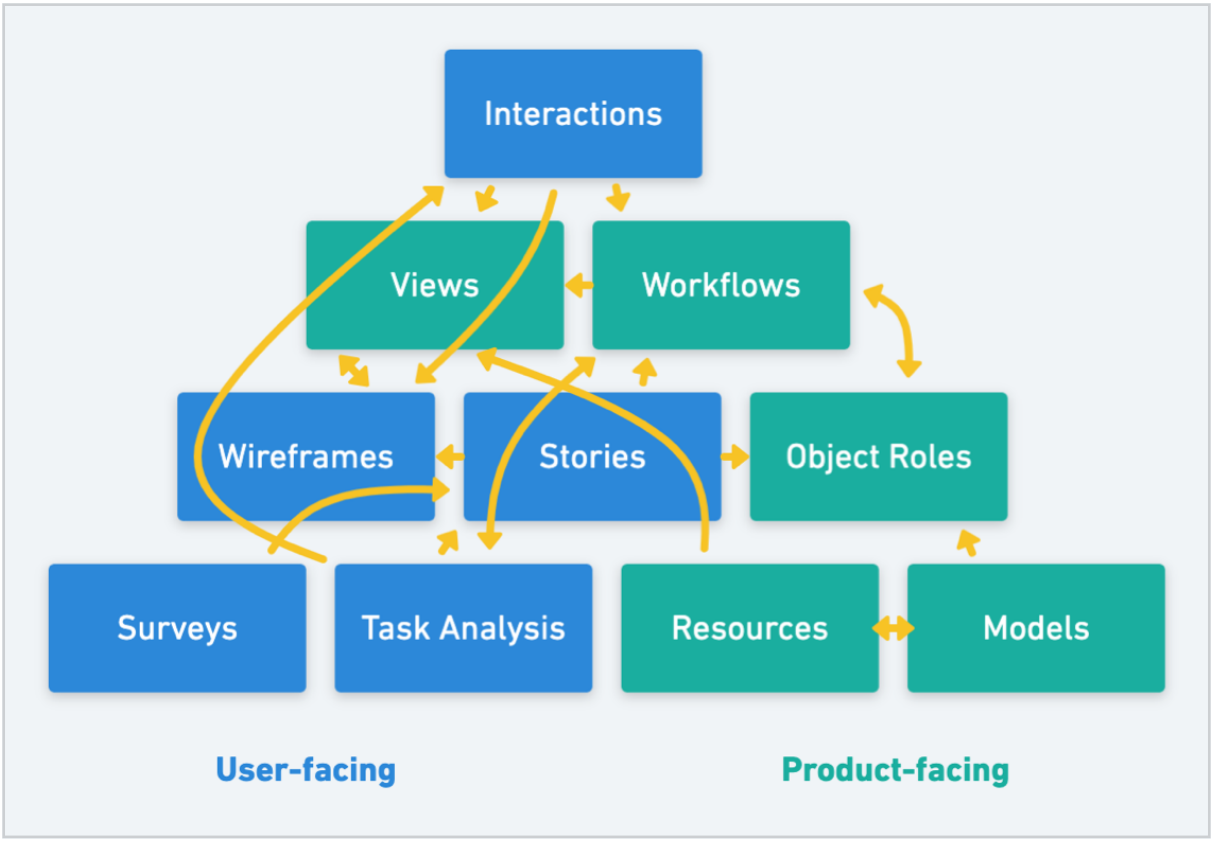
- C: Controller
- A: Adapter
- WF: Workflow
- SO: Service Object

I use these patterns to get a feel for the interactive flow of the system. I also like to keep them abstract so that you can reason about the shape of the feature without getting bogged down in the details. One can easily see that an interaction with the Delete Icon will interact with a kind of `DeleteIcon` workflow object that hits the database. Likewise, an

interaction with the Donebox will send a message to the controller, which sends a message to a kind of `MarkDone` workflow object, which hits the database and sends a message to a service object, which sends a message to an adapter, which sends a message to the cloud. We can assume that the service object is interacting with some sort of 3rd party logging service - hence the adapter and cloud - and logging metadata about the completed todo. If we need to be more explicit, we certainly can.

Shorthands are easily changeable and easily disposable. The amount of value they provide for the amount of risk they generate is outstanding. Once we have a shared mental model and agreed upon approach, we can reify our patterns into a runnable system.

When I'm sketching features I jump back and forth between product-facing elements and user-facing elements. Building a feature is like build a pyramid, if you focus too heavily on one side of the equation it'll topple over. Do a little domain discovery, a little bit of research. Stay flexible and bounce back and forth. Too often we consider ourselves a "DevOps" person, or a "UX" person, or a "front-end" person, but if we focus too heavily on a specialized part of the pyramid we lose sight of how interconnected our piece is to the whole.



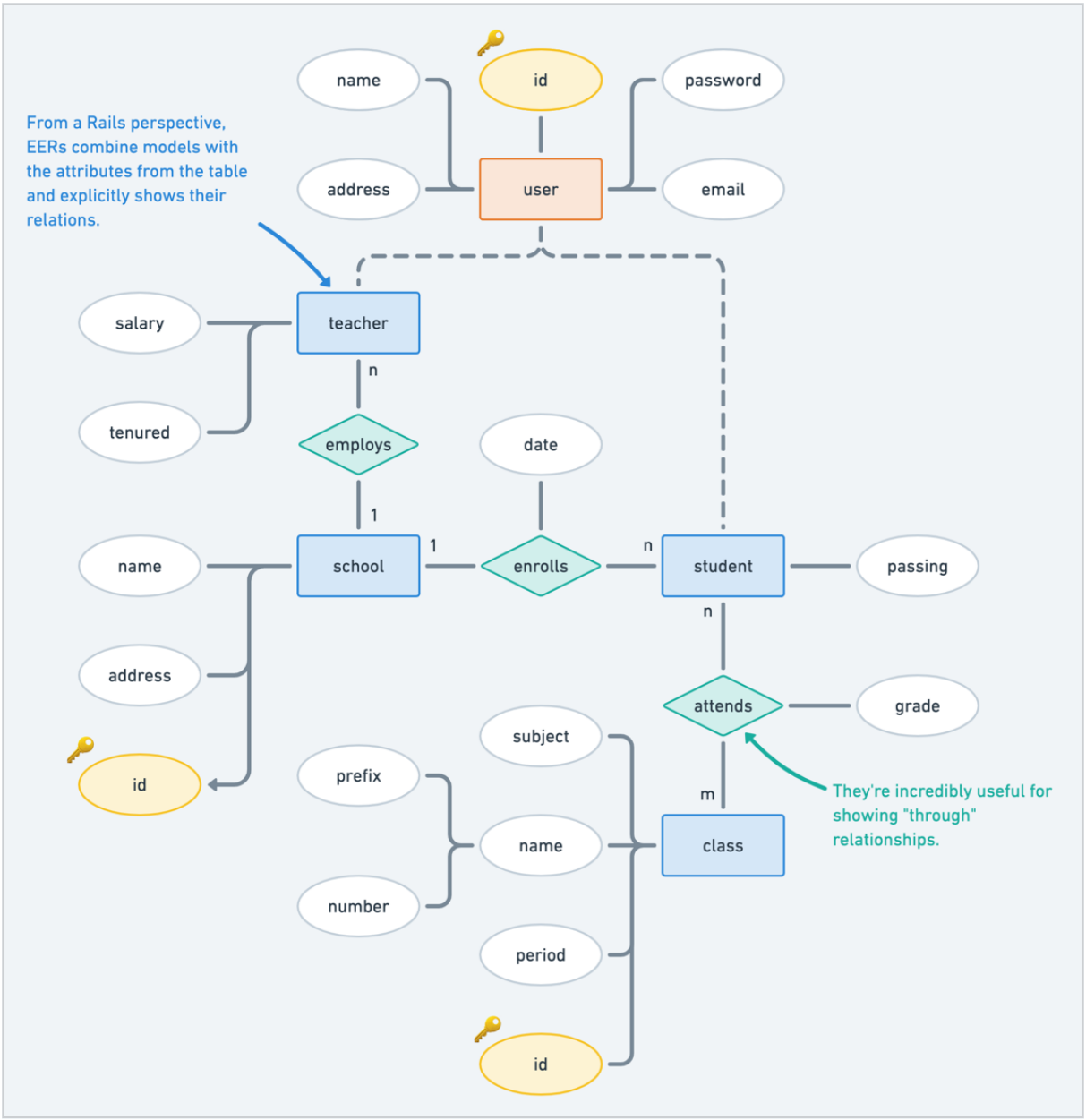
The pyramid is not designed to imply that we start at the bottom and build up. Instead all the forces that make our product cohesive are interconnected and they strengthen one

another and we shouldn't put undue emphasis on one over the other. By better understanding our user we better understand our domain, and by better understanding our domain we better understand our workflows. And by better understanding our workflows we better understand our user. Transitive properties abound.

So much of the domain is tied up in a holistic understanding of the user. If you have a design team, learn the lingo. Understand fundamental forms of user research like task analysis, surveys, and mockups (P.S. I'm working on another small book addressing these sorts of user-facing techniques for Rails developers to be released later). Not only will you gain a better understanding of the domain by doing so, but you'll also make the organization stronger by reducing handoff friction and making the team more cohesive. Handoffs generate silos. Reduce them at all costs.

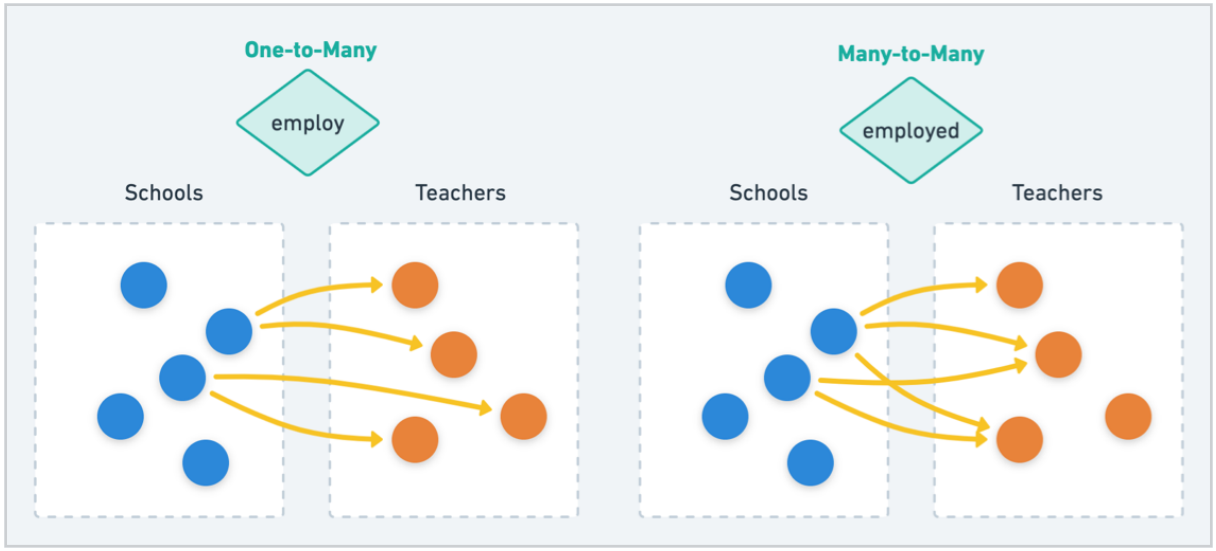
Drawing your data

I like to use Entity-Relationship models to get a "big-picture" understanding of how my models depend on one another. The blue rectangles represent my models that inherit from `ActiveRecord`, the green diamonds describe the relationship, and the orange squares are superclasses. The ovals are attributes and the yellow oval is the primary key.

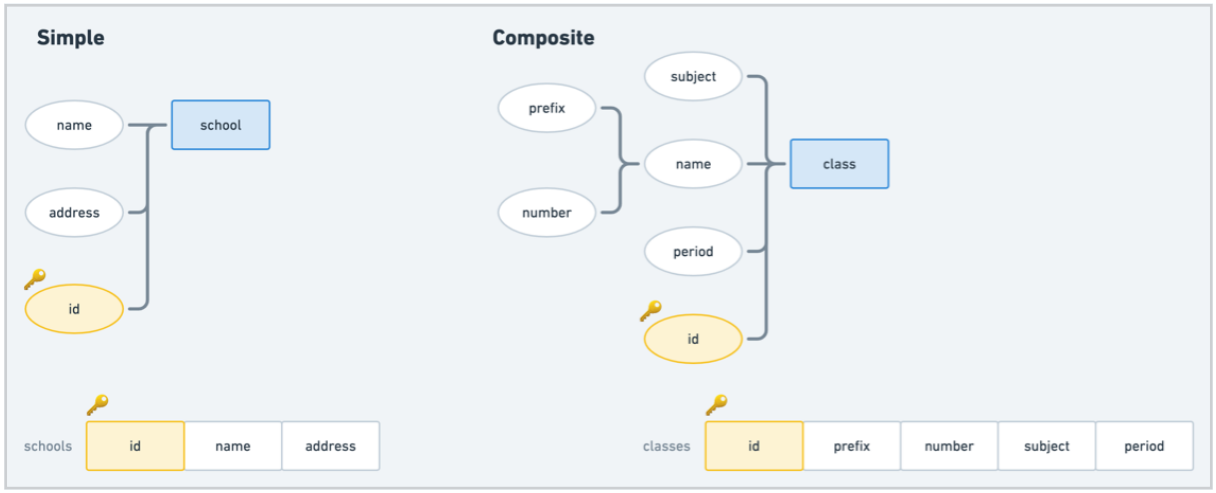


What I really love about the ER models is how the relationships between models is explicitly given a name. For many-to-many relationships this makes creating “through” tables a snap. I also find ER diagrams much easier to reason about than looking at snapshots of the database, migrations, or even reading model code since model code often mixes in constraint logic.

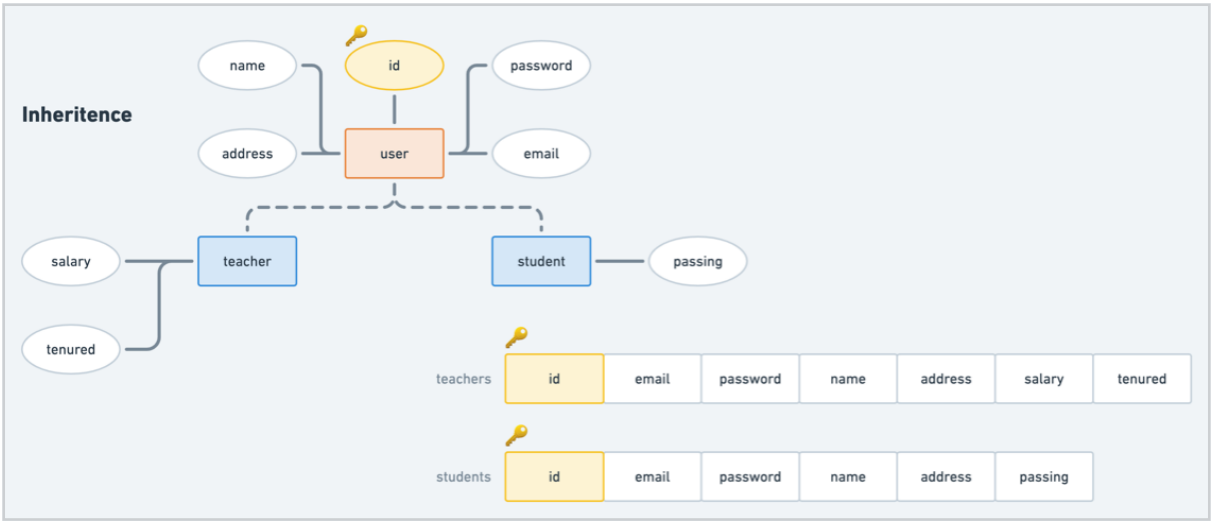
ER models are also excellent for quickly reasoning about the nature of relationships. Below we see how by simply changing the tense of the relationship the nature of the relationship completely changes. ER models give us the flexibility to think about these things with little risk. No migrations have been run and no model files are checked into version control.



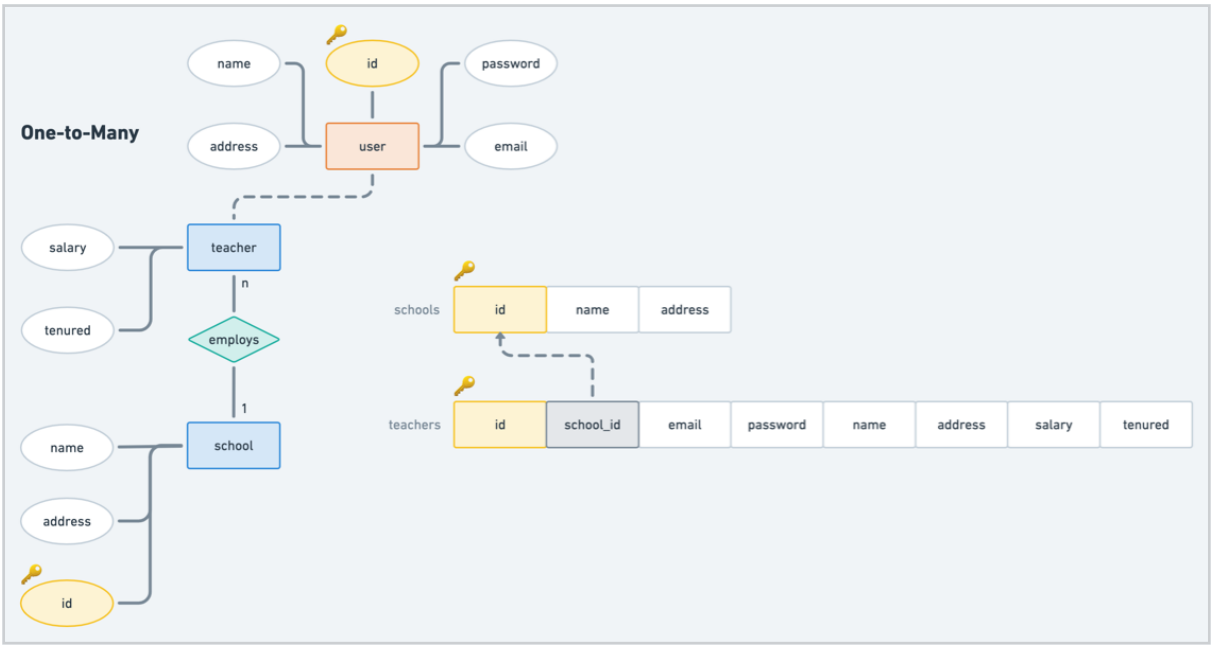
The best part of ER diagrams is that they map directly our tables allowing us to easily infer the migrations necessary. Below are examples of how simple and composite models with attributes become tables. The properties become attributes, and for composite nodes the leaf-most properties become attributes.



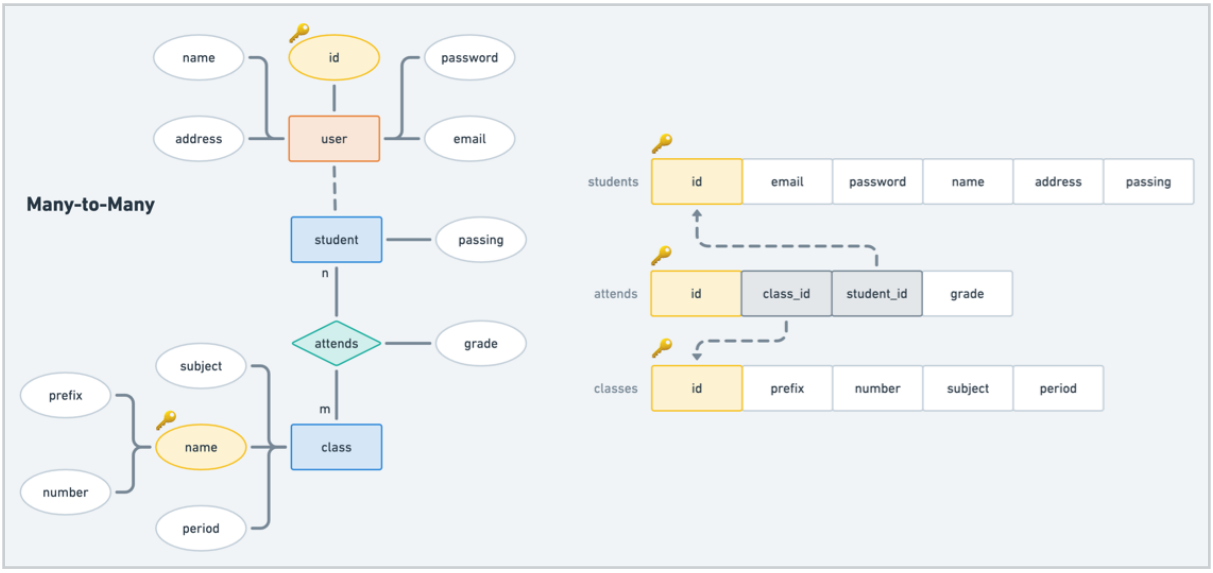
For inheritance we're using Multi-Table Inheritance. I find using Multi-Table Inheritance to be more consistent than Single-Table Inheritance since we can avoid null attributes and are not reliant on strings to reference the type of our children. The properties of the superclass get propagated downwards into the attributes of the children.



Here also are examples of relations that have foreign key dependencies. Note that in one-to-many relations the foreign key goes on the *many* part. Although not shown, in one-to-one relationships you can put the foreign key on either table.



For many-to-many relations the ER diagram allows us to simply reify the verb describing the nature of the relationship into the name of the table we will be using for our `through` tables.

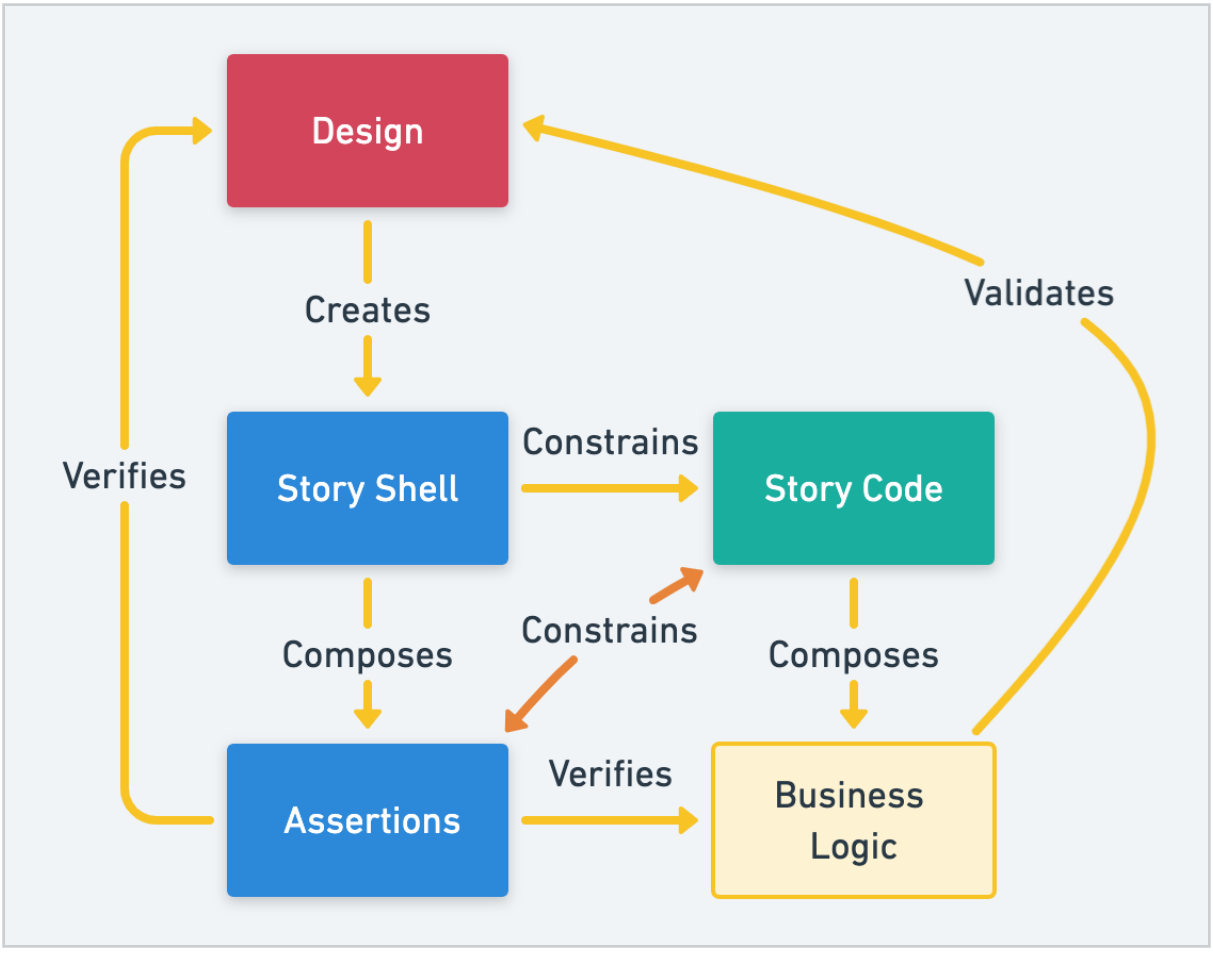


Because ER declarative and decoupled from a running system, they're an excellent opportunity to dive deeper into SQL and how it maps to ActiveRecord. Draw your data model before you write your migrations. Put it all out there on the whiteboard. Think about the shape of your model as groups of sets, intersections, and relations. Keep it lightweight. Expose your assumptions about the shape of your data to others, the wisdom of the crowds is an incredibly effective method of normalization for declarative data modeling.

TDD-ish

If you're writing Rails apps, you're writing tests. Few people discount the value of automated testing, but many are critical of the logistics of writing tests. Do we follow strict Red-Green-Refactor and write isolated units with strict boundaries? Or do we take a more relaxed approach and push our units to the edges of the system? Or do we do both? Neither?

Below is a diagram of how I approach TDD. The red box is design, green is code, blue are tests, and the yellow is rules. I start with a mockup and general design shorthands and diagram. These are tangible deliverables that the business unit has signed off on and guide me in ensuring the code I'm about to write will match the designer's intent.



The story shell is just the shell of a feature spec written in either RSpec or MiniTest and is named after the feature being designed. It has a name like **add todo** or **login** and, unsurprisingly, matches the name of your feature or story.

```
# story shell
feature "add todo", type: :system do
  before {}
  specify {}
end
```

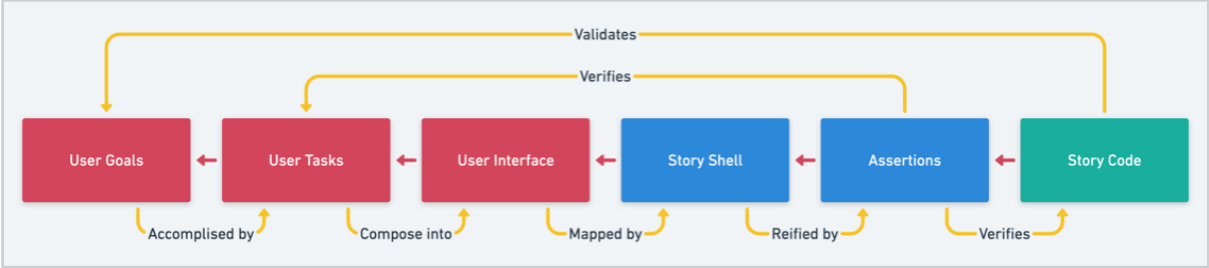
Story code is the code you write to validate your design. Story code is **models**, **views**, **controllers**, **partials**, **helpers**, **service objects**, **form objects**, and the like. Story code composes the business logic necessary to validate your designs. Story code often has an explicit API that our assertions must leverage in our Capybara.

As business logic grows, unit assertions verify integrity and coherence. As story code

makes the feature *real*, system assertions verify correctness.

Story code is also why doing strict TDD means you must be fluent in parts of our framework API before it's written. When we're writing POROs it's easy to write the name/shell of a method we then implement, like `Todo.is_current?` and then write the object, the method, or both. But often story code depends on a predetermined shape in our code, like with Capybara assertions, and requires foreknowledge of these shapes if we desire to use true TDD. When I was learning Capybara I started with the story code and then learned the Capybara API by bouncing against it until my assertions passed. A TDD-ish approach, but it helped me learn Capybara faster than searching through the right query method or assertion against which I could then write my story code. Hence the constraint arrows between assertions and story code points both ways.

Here's another view of the flow showing the dependencies of each artifact on the other from right to left, and emergence from left to right. User tasks emerge from user goals. User Interfaces emerge from composites of user tasks. Story shells emerge from UIs. Assertions emerge from the semantics of the story shell. And story and domain code emerges from the constraints imposed by the assertion. If you take away only one thing from this section, it should be that without a solid understanding of the user, you cannot write functional code. These are not runtime dependencies, they are knowledge dependencies, and they are more real and more important to the business domain than runtime dependencies.



Story shells are generated by the requirements of the feature. A solid understanding of the feature implies a solid understanding of the domain. The feature is only a projection of the domain. We should not seek to represent the entire domain. Our job is to represent with story code the smallest part of the domain necessary to fully satisfy the feature. By composing bits and pieces of the domain to satisfy the feature we keep responsibilities only to the necessary.

It's important to note that this does not preclude us from having to think about our system at a higher level. We do not write our models to the use case. That is what Controllers are

for. We construct our model attributes to satisfy the domain. Controllers extract only the necessary from multiple entities. Entities should be resilient across multiple features, and our story code should be written only against the feature we're working on.

If stories are composed of entities and behavior, and controllers compose behavior from multiple entities, how similar are stories and controllers? Very. But because Controllers have many technical responsibilities beyond the story - composing partials/components, rendering views, etc. - I use an abstraction called a Workflow object to handle the feature-specific object. Workflow objects are named after the feature, use case, or story they represent and handle the logic necessary to satisfy the feature. Ivar Jacobsen calls them Controllers, and Uncle Bob calls them Interactors. RESTful design too has an abstraction called a Controller Resource that's largely the same idea. We'll see how to represent these story-like Controller Resources in Rails a little later on.

I write my story shell, give it a name, and then I write some story code. Then I assert against my story code using system tests, and it verifies my designs. If, to satisfy my story, my story code needs refactoring into business logic, I do that during the writing of the story code. I stay focused, and I do not anticipate my business logic needing to satisfy other story code. If, when writing another feature, my business logic does need to satisfy multiple features to keep my code DRY then I consider an abstraction.

Going wide and staying flexible

We're familiar with Model-View-Controller, but consider other patterns like Boundary-Controller-Entity, Data-Context-Interaction, or Domain-Driven-Design. Don't set out to immediately implement them. Set out to study them. Find their edges and overlaps. You'll soon find many similar patterns have different names. The key is to notice the emerge of similar patterns across different techniques. But MVC is king. It's a fine, fine pattern for the web and interactive systems with hard boundaries. Start there. Once you get comfortable enough with MVC you have a vantage point from which you can look upon other patterns and view their suitability accordingly.

For example, with Boundary-Controller-Entity the controller object is incredibly useful as a workflow/use case surrogate object. The boundary objects are fantastic substitutes for view-models or even web components. There are excellent examples in Ivar Jacobsen's book Object-Oriented Software Engineering where he decomposes various UIs into their boundary and controller objects. But without a fundamental understanding of the system boundaries present with a pattern like MVC we don't have a model against which to find the similarities and difference.

Data-Context-Interaction is an incredibly powerful way to think about sets of objects interacting. It places emphasis on the mental models of the user and the behavior of the system over the state of the runtime. It focuses on the use cases and sets object boundaries around their contexts and the roles they play in interaction. Similarly, Domain-Driven-Design is a methodology for thinking about objects in the context of their business domain. There are many patterns, and it's very nuanced, so really only start with understanding services, entities, and value objects. I've found these three concepts alone are worth exploring even if you don't touch the rest.

One of the best ways to learn is to learn from fine examples. It can be rather difficult to know what's good and what's bad if you haven't spent a lot of time with various codebases. Here are two fine example of Rails apps that are a bit unconventional in 2019, but provide an incredible opportunity to learn.

I mentioned Jim Coplien and Trygve Reenskaug's DCI earlier, and this repo is an example of the DCI infrastructure implemented in a Rails app. DCI maps very well onto the MVC paradigm because the MVC paradigm matches the client-server paradigm so well. DCI brings focus to the user's mental model and the problem domain and does not concern itself with system boundaries. The repo is a fine example of a well-tested Rails application representing a simple problem domain and provides a rich teaching opportunity.

Another example of a teaching repo is the Whitehall repository provided by GOV.UK. What makes this repo a fine example is the amount of effort they've put into maintaining their feature specs. They use gherkin and cucumber for their acceptance testing, and they are an example of what a product can look like when you have good stakeholder communication and a shared vision.

One more example of a fine repo is for the website Lobste.rs. What's great about this repo is how normal it is. Controllers have too many actions, tests hit the database, and coverage isn't perfect, but that's what adds to its charm. It's just a normal everyday Rails app, the kind you'll find in the wild. Everything is well named, the models are explicit, and the unit tests are sound. It's also a great site with a great community.

Dive in, go broad, get a feel for the edges by ingesting a wide variety of sources. Much of the best material was published in the 90s and can be found by third-parties for $5-10. Start a library. One of the beautiful things about Rails is that 15 year old material is still relevant and applicable today. There's a lot of fantastic material out there, and curating it is a bit of an art.

But don't go too deep at first. If you're like me, it's easy to get sucked into these things. When I get sucked in, I lose perspective and start to see everything in terms of the pattern. This is polluting and serves no one. It takes self-control and temperance to pull back and see the broader context. Get excited about these ideas - they're salient and enriching - but don't let them dominate your thought.

Web fundamentals

There are a lot developers who are proficient at writing code for web frameworks but know very little about the fundamentals of TCP/IP, congestion control, HTTP, latency, and the rendering lifecycle. Basic knowledge of network and browser performance is imperative, and there's no better source than Ilya Grigorik's High Performance Browser Networking. Make sure you understand the difference between long-lived and short-lived flows, how these can impact battery life for mobile devices, how TCP slow-start can cause your app to load slower with smaller payloads, and how to order your assets to let the browser do its job performantly.

The TCP discussion brings up a very interesting consideration about the use of Phoenix Channels and LiveView, real-time communication channels for managing general data-transfer and front-end state respectively. Channels and LiveView use a websocket connection to managing the flow of data, and a websocket is a long-lived TCP flow. If we assume that the requests are multiplexed on a single connection and that the flow of data is limited only by the distance to the server and its current receive window, then we could bypass the limitations mentioned in the previous section. With HTTP/2, this multiplexing happens by default and competing TCP flow congestion becomes a non-issue. One must still consider slow-start restart where idle connections will reduce their receive window in response to periods of inactivity, but this can easily be disabled on the server.

One nice benefit of maintaining an open TCP connection is that mobile devices are smart enough to not engage the antenna for the entirety of the connection, though you do want to be cautious. Any request needs to use the antenna (obviously), but transitioning the antenna from an on-state to an off-state and vice versa will make your battery take a significant hit. If possible send most of your data earlier and as fast possible, avoid sending smaller bursts periodically. Polling - yes, even with websockets - on mobile networks is an anti-pattern.

Bundle size too is a primary concern for web developers. *Smaller is better* is the convention. But is this true? Performance is a function of bandwidth and latency, but size is

a function of bandwidth. If I have 50Mbps down, and I reduce my payload from 800KB to 300KB, is this significant?

The answer is more nuanced than you might think. When you send a bundle over TCP it enters a slow-start phase. This phase saturates your connections bandwidth slowly so that it plays nicely with other TCP flows. This happens for every TCP connection. If you break up a bundle into a bunch of tiny little bundles, you'll have a bunch of tiny little slow connections that terminate before they are can fully ramp up. Prefer one smaller request to one larger request. Prefer one larger request to many smaller requests.

Minimum Viable Refactoring

Whether you're testing or coding, you'll be refactoring. Be it breaking up features into objects or methods into smaller methods, refactoring is a major part of a developer's life.

When we hear the term *refactoring* it's natural to think of the techniques in Martin Fowler's classic book Refactoring. I find Fowler-style refactoring to be too difficult for beginner and even intermediate developers. There are simply too many patterns. The book contains around 25 code smell patterns with over 65 refactoring techniques. Instead of memorizing the patterns and trying to spot the patterns in your codebase, I recommend thinking of refactoring like philosophy or a good argument. Philosophy is the "art of making distinctions", and developers love their distinctions. We spend hours bikeshedding and yak shaving in futile attempts to uncover trace nuances of our systems in the name of robustness.

Refactoring is the art of making new categories and giving them names. Once we have a name, it becomes easier to find a place. Refactoring is putting things with names into other things with names and giving them a reason. These reasons are often defined by their relationships to other reasons. Developers use the "blueprint" metaphor to describe classes because that's what we're taught in school. But "class" is short for "classification", as in "the art of categorization", a domain of philosophy. Classes are not templates for our objects, they are abstract categories that just happen to *reified* by a runtime.

Take polymorphism for instance. You can either have all the verbs change and the nouns stay the same…

```
# not polymorphic
House.turn_on_light
```

```
House.turn_on_stove
House.turn_on_car
```

…or have the nouns change and the verbs stay the same.

```
# polymorphic
Light.turn_on
Stove.turn_on
Car.turn_on
```

Why is this better? Try to explain why it is without using lingo like "Because it obeys the Single Responsibility Principle." Come up with an explanation in your own terms. To really understand something you should be able to have a conversation at length about it, provide arguments for it, and counter-arguments against it. If I have a `Daemon` that needs to process a request, do I use `Daemon.process_request` or `Daemon.process HTTPRequest.new`? Or do I put it on a `Request.process` object and generalize the `Daemon` with `Daemon.exec Request.process`? Why or why not? It's a matter of categories and it's highly philosophical. They each have their pros and cons, can cause alterations of mental models given the rigidity or flexibility of their semantics, and they need to be deeply discussed, logged, considered, and brainstormed. There's a 2300 year old history of western philosophy that can help you think about these things. Plato, Aristotle, and Aquinas were just programmers without the runtime.

Refactoring is the art of describing a mess. It gets confused with cleanliness, and cleanliness gets confused with being sterile. But messes have meaning. Messy desks are snapshots of problems being solved. Einstein was notorious for his messy desk, and he came up with the cleanest equation in scientific history. Steve Jobs too had an incredibly messy desk, and from that messy desk emerged the iPod, the iMac, and the very notion of "clean" for consumer electronics.

Refactoring is the art of making a mess and then picking up after it. Talk about your messes. Think aloud about them. Be slow about them. Change your mind about them. Don't just throw your mess into a pattern because of the shape of its syntax. Extract classes, extract methods. Keep them small. Make them legible and memorable. We have giant displays - make your names long and well-described, but not too long. Read Bob Martin's Clean Code.

We embed our thoughts in objects using a technique called distributed cognition, enabling us to think in complex ways. People are constantly going bigger to go smaller. Their mental models are "laying the chips out on the table". This is a necessary part of problem solving. But messes are terrible things for others because others don't have our mental models. When a mental model in progress gets checked into master it looks sloppy. When code comments or `puts` statements get checked into master, it's a code smell. It's because someone put their mental model onto our desk. And our desk is designed for our own mental models. Check iPods and E=MC^2 into master, not mental models.

Noun-Oriented Programming

Many web developers are under the impression that if a resource is a verb/action then it's not RESTful. There are excellent talks, tutorials, podcasts, and blog posts encouraging the substitution of verb/action resources for noun resources that map logically to the primary HTTP verbs.

Instead of downloading a video by calling `/videos/:id/download` (`VideosController#download`), one would create a `videos/:id/download` resource (`Videos::DownloadsController#create`) returning a video representation. This gives us the benefit of isolation and encapsulation of domain logic as we use a separate DownloadsController to handle the request/response cycle. If there's any sort of external download logic that needs to be performed to satisfy the request, it makes more sense that a DownloadController would handle it over a more general VideosController. There are other benefits to thinking this way, as expressed in the links above.

But this is hardly a one-size fits all solution. There are incredibly simple use cases that can be incredibly difficult to model with Noun-Oriented Programming. One example is an ordered todo list. If you want to move a todo up one place in line, there's no direct path short of having the client manipulate the representation client side and then `PATCH` or `PUT` the entire modified representation back to original `/todos` resource. Letting the client dictate the shape of the representation is not a good idea.

Another solution would be to make a `/todos/:id/moves?direction=up` resource that anticipates the direction query parameter and manipulates the state of the /todos resources, and sends a redirect to the /todos resource. We've managed to adhere to the "noun-ish" form, but imagine what this does to the addressability principles of web resources. Would you ever consider bookmarking or sharing `/todos/:id/moves?`

`direction=up`? Not quite.

Surprisingly, REST provides a solution that fits our mental model. They're classically called Controller resources, and they're recommended by respected members of the REST API community endorsed by O'Reilly and Morgan and Claypool. These resources are named with verb/actions, return only a See Other redirect in the location header, and are still perfectly RESTful. Best of all, in Rails, they still give us the benefit of isolation and encapsulation.

So let's apply the Controller resource to our two example problems: The "download video" use case and the "move todo up" use case. In order to download a video using our Controller resource, we use the same URI scheme as the other two proposals: `/videos/:id/download`. This is a bad example because the word "download" is a gerund (both noun and verb), but in this case we intend for it to be a verb. We create the resource with a `POST` which creates the resource within the Rails controller `Videos::DownloadVideoController#create`. Note that we name the controller after the use case and namespace it as a subresource.

Once we finish our use case we redirect to a specified resource and send a 303 See Other with the desired resource URI in our location header. If we don't want to redirect, just send a 202 OK response with at least some sort of canonical response or link to navigate to so the client isn't left hanging.

For the "move todo up" use case, we'd use a URI like `/todos/:id/up` which maps to `Todos::MoveTodoUpController#create`. The controller would use the route params to manipulate the data, persist the changes, and redirect to the `/todos` resource. Feel free to do this in an async request and re-render. Use your best judgement.

The more we look into the more nuanced sides of REST and best practices the more surprising things we discover. For instance, we often hear that GraphQL is an alternative to REST. Would you be surprised to learn that it's actually a RESTful pattern known as Command Query? Always remember to do due diligence before signing off on an approach or technique, you could end up saving yourself a lot of mental energy.

The Problem with Pickles

As we're integrating a feature into a product, there are hidden forces that will inform the eventual shape of our solution. It's important to mold the shape of your solution to fit the shape of your problem. Too often we assume the shape of the problem by starting with a

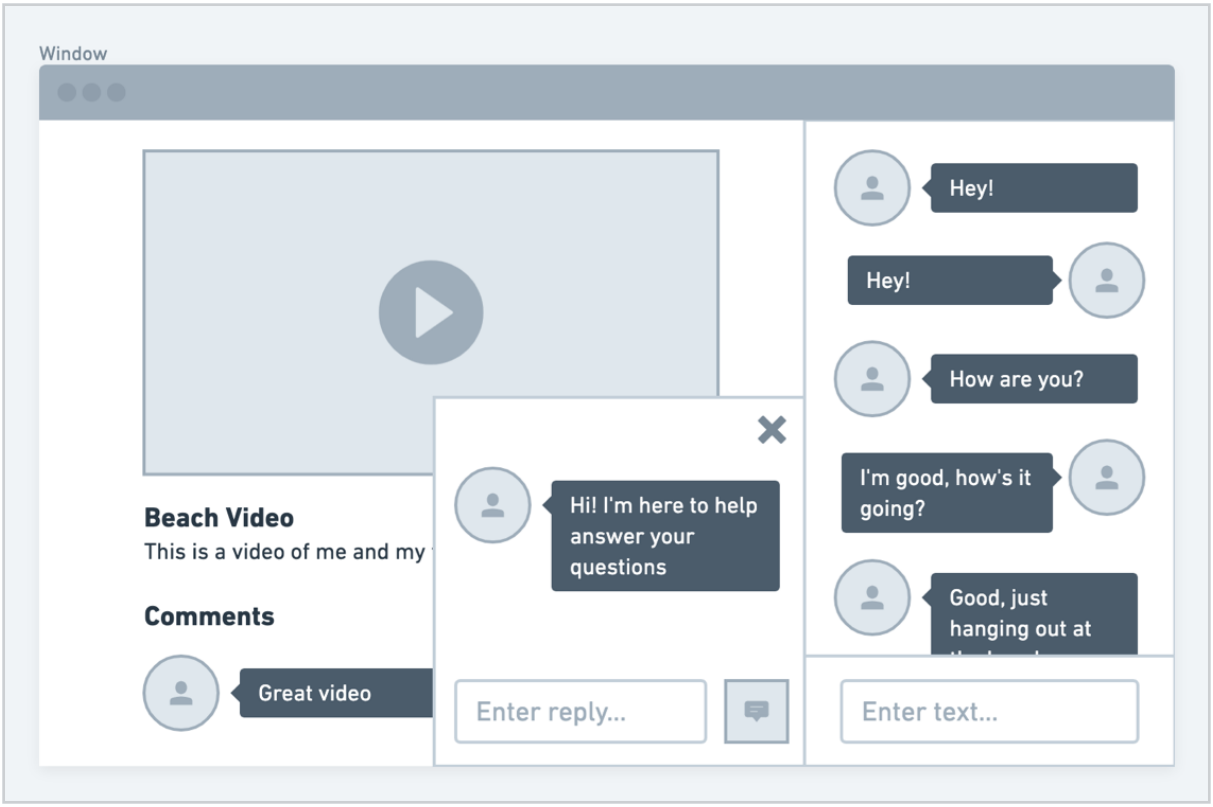
solution. Facebook-shaped problems generated React. Netflix-shaped problems generated microservices. Basecamp-shaped problems generated Rails. If we start with the intent of using something like React or Angular, we're assuming the shape of out problem is the shape of Facebook of Google. I find it safer to start by assuming the shape of the problem will resemble conventional web sites or community patterns. The large majority of problems are Web 1.0 or 2.0 problems, and we've had the technologies to solve these problems for a long, long time.

Pickles are difficult situations that require a technical decision. A pickle tells me something very important about the problem shape and is rarely predictable. I like to start off with traditional approaches to web development, and when I run into a pickle only then do I think about decorating my current solution shape with something more befitting my understanding of the problem. It's very tempting to construct an entire solution out of a new technology because we can see a potential solution through the feature set, but to best serve our clients we must choose the prudent over the attractive. Leverage the power of your browser and the limitations of your network protocols to craft an appropriate solution. Use the platform, don't try to make square pegs fit round holes.

The problem with pickles is that they are often a Pandora's box. They're opportunities to inject undue complexity into the system. Solving pickles can lead to over-architected solutions. This leads to entropy in code, greater cognitive load, and team silo-ing. Greater effort is spent on communicating requirements and balls get dropped between handoffs. Pickles can also lead to cheap and lazy solutions through shopping and patching. An overabundance of external dependencies induces an undue maintenance burden managing packages that change according to forces outside the organization's control.

Sometimes pickles can be uncovered before we get stuck if we do a cognitive walkthrough. Take the example below. We have a standard UI with 3 to 4 major use cases.

1. I'm talking to some friends
2. I'm chatting with support
3. I'm watching a video
4. I'm reading some comments



Seems straightforward enough. We'll have a `Chat` model, a `Video` model, a `Help` model that perhaps inherits from `Chat`, and some `Comment`s. We'll also need to set up some resources so that we can see and interact with these models. This is where we encounter our first pickle. With exception to the comments inside the video, all these use cases seem pretty independent to me. There's nothing really about watching a video that would preclude me from getting some support or chatting with my friends.

But the way the web works is that what I'm looking at is basically a big table represented as a tree, and the way a tree works is that one of the nodes in the tree has to govern all the children below it. So I need to pick one of these things to be the primary resource that's going to coordinate everything we're looking at. Which do we pick? What does your gut say? Mine says "Start with the video." I don't know why it says that, perhaps because it's biggest, or that the other representations seem to be either "inside" it - as with comments - or "on top of it" - as with the chat. But either way, I suspect we'd all make sure that these resources are coordinated with a `VideoController`.

So we code everything with our `VideoController` and delegate the responsibilities of the chat functionality and comments to presentation helpers and partials. But then we need our help chat to stick around when we go to `/home` or anywhere else. Well since it's just another partial we can decorate it into all of our other parent resources or abstract it and the parent resources into layouts and put them as siblings and adjust the styles to match. But then with Turbolinks it'll blow away the scroll state when I traverse the parent resource

and preserving the state of the conversation is really important. Uh oh, another pickle.

Prefer a conservative solution that is well documented at a high level, preferably with an accompanying shorthand and process diagram, and retain a history of conversations about the pros and cons of potential solutions. Do not expect the changes necessary to solve one pickle to help in solving another pickle. We don't yet have the proper language to describe the ontology of problem-types we encounter. There are currently no categories, and do not use your gut or adductive reasoning to presume you will know. Even if you do have a good gut and it's likely right, there has to be solid evidence of disputation for the organization to learn.