What is *Slamdunk*?

Slamdunk is the grand unified design pattern and coding template for user interfaces.

First, a little history...

Way back in 1968, one of the founders of Computer Science, Edsger Dijkstra, wrote an infamous technical paper known today as "Goto Statement Considered Harmful". This ignited a firestorm in the software world that lasted for about a decade. What this paper said, and what Dijkstra meant by it, is still debated in some circles. But it had an effect on how people write software that persists to this day.

In the early days of computing, CPU main memory was a scarce and valuable commodity. If you didn't live and work in those times, it may be hard for you to understand just how scarce and valuable it was. Of course, programs in those days had to work, just as they do now, and getting your software to work was and still is the primary goal. But an equally important skill for a software developer at that time was the ability to *make your program fit* inside the computer. This resulted in coding techniques that would be considered insane today. And the reason they would be considered insane today is because of Dijkstra's paper.

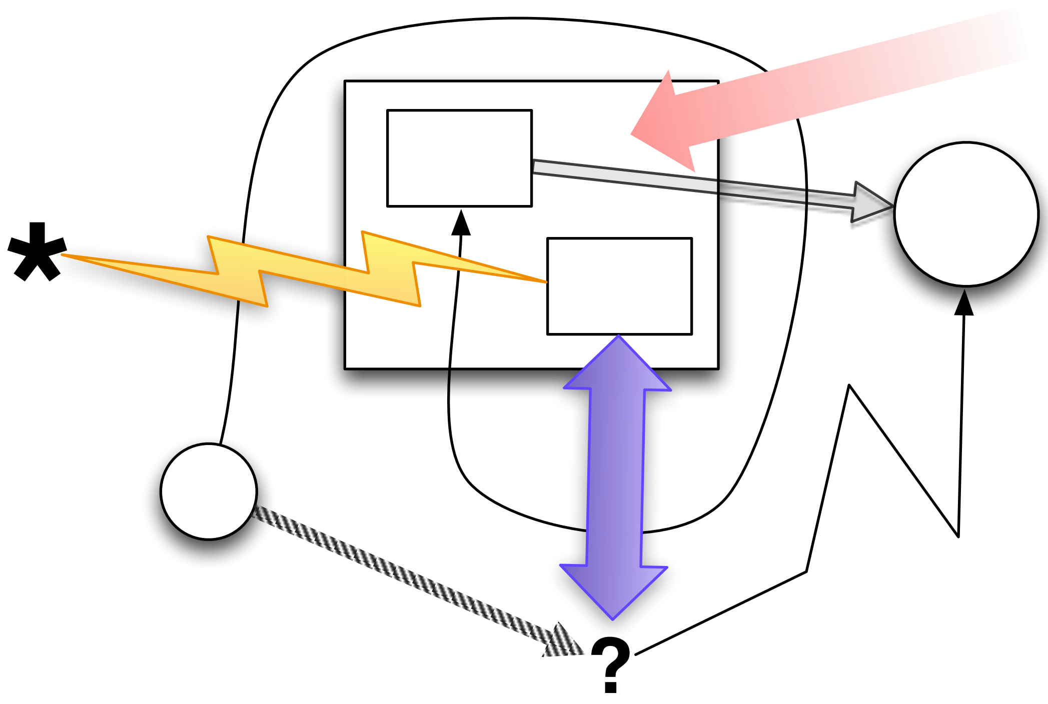
The practical consequence of "Goto Statement Considered Harmful" was an insistence that program flow be limited to: 1) function calls, 2) decision trees, and 3) purely nested loops. What you weren't allowed to do anymore was use `goto` just anywhere it might seem convenient, and better not to use it at all. ***This was a big deal.*** Many experienced and highly competent programmers revolted at this requirement, asserting that real software could not be written that way. Just a few years before, that was probably true. But things were changing. Memory was cheaper, a little more plentiful, and not quite so precious. Also, people started to notice this new *structured programming* didn't often cost all that much more memory than unstructured techniques, which were hard to write, understand, and debug. Straightforward simplicity had become more important than mere size, and functionality never suffered. This had increasingly significant economic effects as the relative costs of computers and programmers inverted over time. Eventually, pretty much everyone decided to get on board. These days, some programming languages don't even have a `goto` statement. In those that do, it is rarely used, and used only in very limited ways when it is used. Breaking this rule will get you in trouble in most development shops, and with good reason.

The Goto Wars were an important milestone in the history of software development, largely forgotten today. We have never seen anything like it in UI construction. But maybe we should.

Typical UI Code Considered Harmful

A common sales pitch for UI Component sets and development tools is "*You can do anything you want!*" How often have you heard this line yourself? It sounds appealing. What could go wrong? Demonstrations of UI tools make a point of connecting anything to anything and making that look useful, desirable, sexy. We might call this ***Bordello Morality***.

Some Typical UI Structure



Is this supposed to be a joke? Sadly, it isn't.

This is similar to the power offered by the `goto` statement: go anywhere, anytime, and do anything when you get there. But we learned from painful experience that this kind of thinking is not a good idea. It all too often results in wasted time, unexplainable expenses, and incurable bugs. One of life's hard lessons is that some restraint, maybe even a lot of it, results in more happiness overall.

Restraint is what we *don't* find in much UI code. We are talking about desktop UIs here. Web UIs have their own vast array of problems, and the subject of this article is relevant to some of them. The subject of this article is *central* to desktop UIs.

If we are to develop a helpful kind of restraint in UI construction, what form should it take? Here is the recipe:

What Your Professor Said Is Still True

The first order of business in UI construction is to separate the Model from the View. If you ever took a class with a title anything like "Software Engineering", you heard this from someone, somewhere, sometime. So everybody knows this. What I don't often see is developers following this advice. The usual approach is to create some subclass of `Panel` and dump everything having to do with the problem at hand into it. Then, when it turns out the problem is bigger than you thought it was, probably interacting with things outside the scope of your initial solution, or even if you just want to change the existing UI, you have to go back and re-engineer the whole thing. Either that, or your new code invades your old code, making connections to anything that might conveniently deal with your new concerns. Here, the similarity to the old `goto` issue becomes more clear. The result is code that only you can understand. And if you go away and come back to look at this code a year later, or maybe even a week later, you may discover that you don't understand it either. This is not acceptable.

So let's revisit that Model/View thing your professor told you about.

A Model does two things: it 1) holds Model state, and 2) implements Model behavior. In other words, it does everything your *application* is supposed to do (query a database, operate a process, etc.), but it is *not concerned with how it appears on the screen*. A good Model may make allowances for a good View, but this is incidental to Model design.

A View does two things: it 1) displays Model state, and 2) invokes Model behavior. And that's *all* it does.

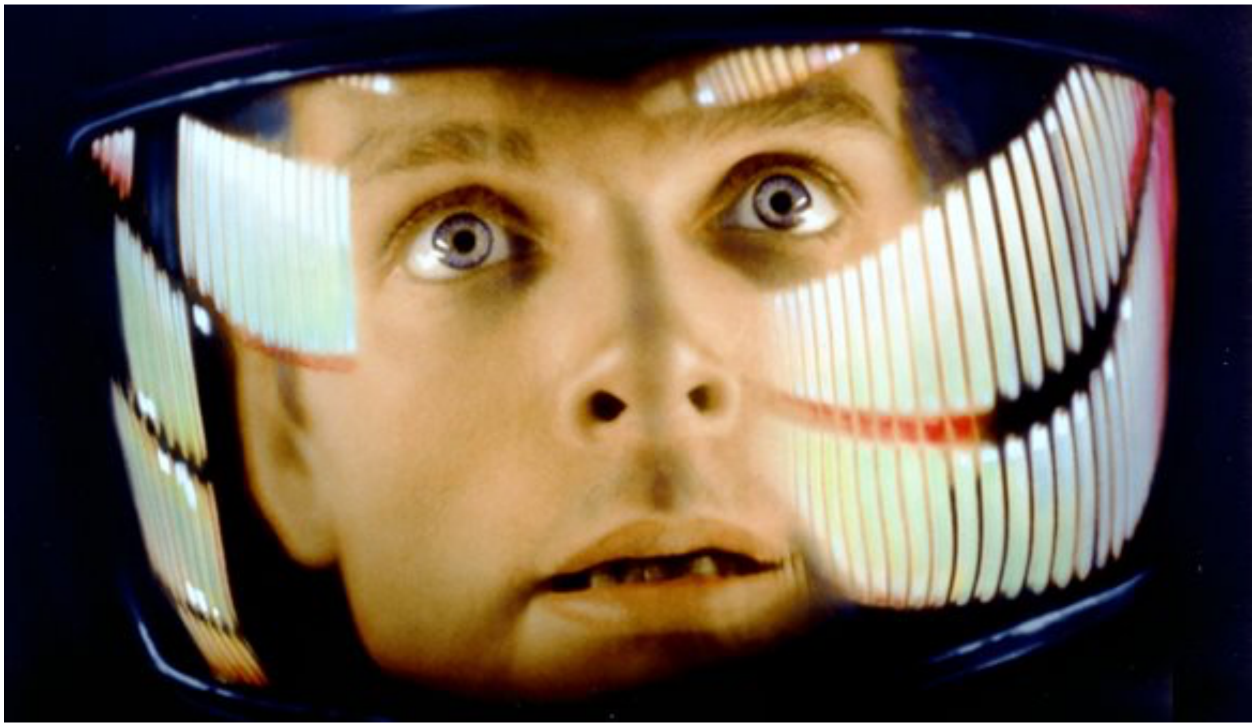
Once we understand the Model/View distinction and decide firmly to follow this good advice, we are ready to take the next step.

Now... *Slamdunk*

The essence of the Slamdunk UI Design Pattern is this:

For every View, there is one and only one Model.

It may not seem like much, but ***this is a big deal.*** Many experienced and highly competent programmers revolt at this requirement, much as they did when `goto` was banished from polite conversation. It seems like an unnecessary, maybe even crippling constraint. But you will change your mind. Slamdunk brings a tremendous amount of order to your UI code. It also enables powerful constructions you probably never envisioned because, without the Slamdunk constraint, there is not any effective way to envision them, nor is there any reason to try.


The look on a developer's face when he first really understands Slamdunk. No kidding.

Whenever I first describe Slamdunk to a new UI team, one or two members might understand its importance right away, but this is rare. Some go along with it simply because I am The Boss and I said so. Others are not so willing. One young fellow went over my head to insist to *my* manager that "this Slamdunk thing" was suicide. Whatever their initial reactions, *everyone* eventually accepts and embraces it. Even the kid who thought it was suicide did. That's because it works. Bordello Morality doesn't.

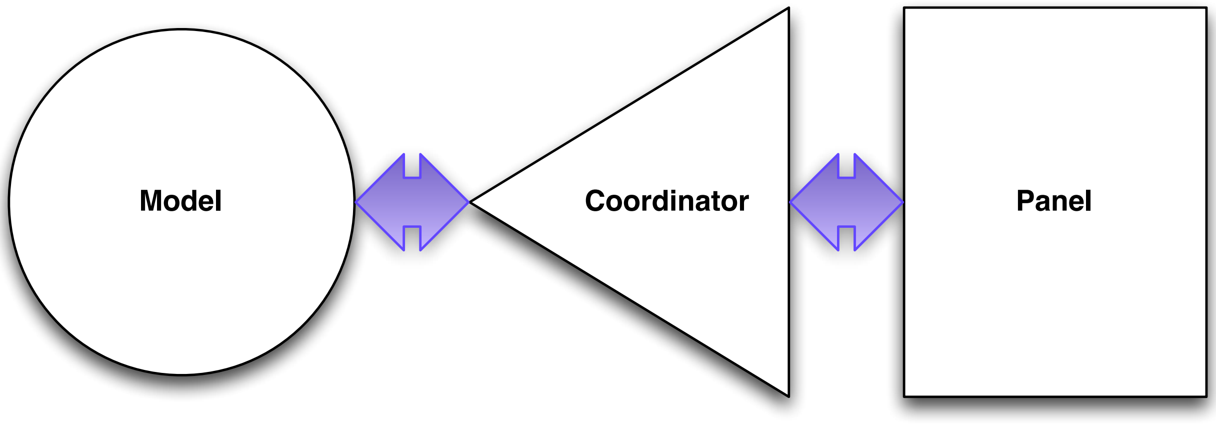
Someone always asks, "What if my UI really does display two or more different Models?" That's a fair question. If you encounter this (and you will), then create another Model class that encapsulates the submodels, and use this encapsulating Model as the single Model for your View. While this may seem like extra work now, it saves more work later on, and also makes it possible to do other good things you haven't seen yet.

There are two forms of Slamdunk: weak and strong. Don't infer too much from these labels. The weak form is not bad, just not as powerful as the strong form. Most people find the weak form easy enough to digest. Many people find the strong form frightening, maybe even terrifying... at first. You can implement the weak form today and reap substantial benefits in your current project. The strong form requires a supporting library that is easy to use but not easy to construct. We have such a library here at `brising.com`, but you might prefer to use the weak form for a while as you become accustomed to this kind of thinking.

Slamdunk: The Weak Form

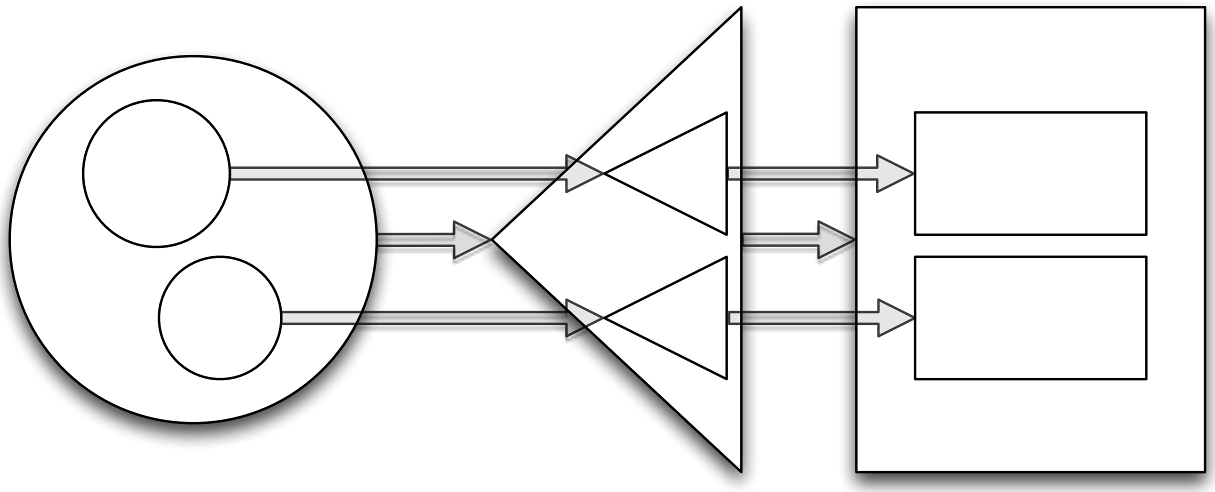
Before you put anything up on the screen, you must first have a Model. Perhaps you think you can put this off until later. You can't. Go design, build, and thoroughly debug your Model now. Then you can come back and build a View for it.

When you have your Model well in hand, sketch out how you want it to appear on the screen. Then 1) create, 2) configure, and 3) connect your UI Components. Where to do this? You might think *this* is the place to make a subclass of `Panel`. That isn't the worst choice you could make, but it isn't the best, either. Instead, create a new object called a Coordinator (No, don't call it a Controller). Its job is to create, configure, and connect your UI Components. In the weak form of Slamdunk, the UI Component is always a `Panel`, which is created in the Coordinator's constructor. Then create, configure, and connect whatever Components you need inside that `Panel`. Set up any Component event handlers now, where it is the Coordinator that handles the events.



The weak form of the Slamdunk UI Pattern

Now, create a method in your Coordinator that attaches the Model. The traditional name for this method is `setValue(Model)`. This tradition is worth following. Be sure that *every* Coordinator you ever build has this same method, because most of the payoff in using Slamdunk happens right here. Of course, the best way to ensure you have this method is to subclass an abstract Coordinator class. The structure of the `setValue(Model)` method is not trivial, but it is arranged so your implementation always contains trivial code. This is described in detail in a separate article.



Composition of Weak Slamdunk Assemblies

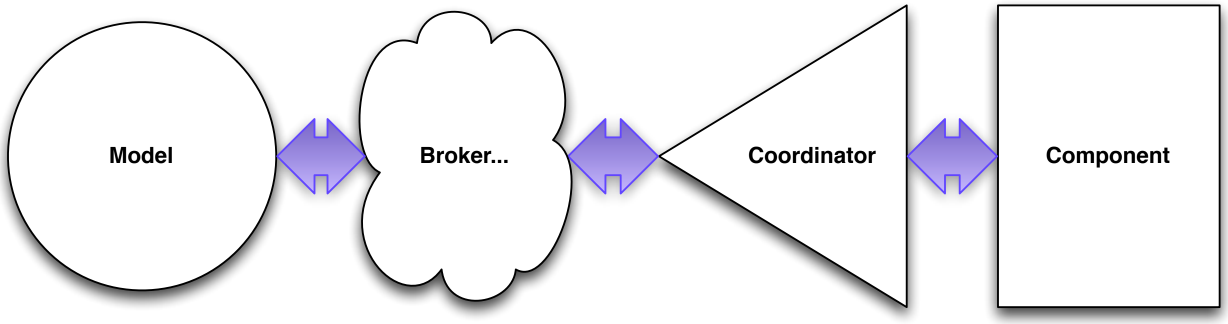
There is something important to notice about this picture: Slamdunk is a *fractal* pattern. In any realistic example, it probably exists at many different levels, always essentially the same. At any given level, you can apply the same kind of *thinking*, the same kind of *coding*, and the same kind of *tooling* as at any other level.

Slamdunk: The Strong Form

After you have built several weak-form Coordinators, you will become impatient. A lot of common code appears where you create, configure, and connect Components. You will want to write this code *one last time* and put it somewhere so you won't have to look at it again. This means you are ready for the strong form of Slamdunk.

Instead of thinking at the level of Panels, think about all of the Components in your Component set. Think about having a distinct Coordinator class for *every one of them*. When building screen layouts in this environment, you don't instantiate Components. You instantiate the Coordinators for those Components. When the Coordinator's constructor returns, its Component exists, fully configured, ready to work. All of its event handling code is already in place, hooked up, and ready to respond in ways that are well understood and fully debugged. Those annoying snippets of orphan code you have to write in most GUI builders? They go here, written in a way that works for all situations. *Can we do that?* Yes.

Another thing you will notice about weak-form Coordinators: their code differs in the presence of specific getters and setters *on the Model*. If we can extract these getters and setters and invoke them in some indirect way, more boilerplate code disappears. The object that does this is a Broker, which also has a `setValue(Model)` method. If you think you're getting a whiff of that Object-Oriented magic you hear about from time to time, yes, you're in the middle of it now. This is where Slamdunk starts to compound upon itself, resulting in a whole that is greater than the sum of its parts.



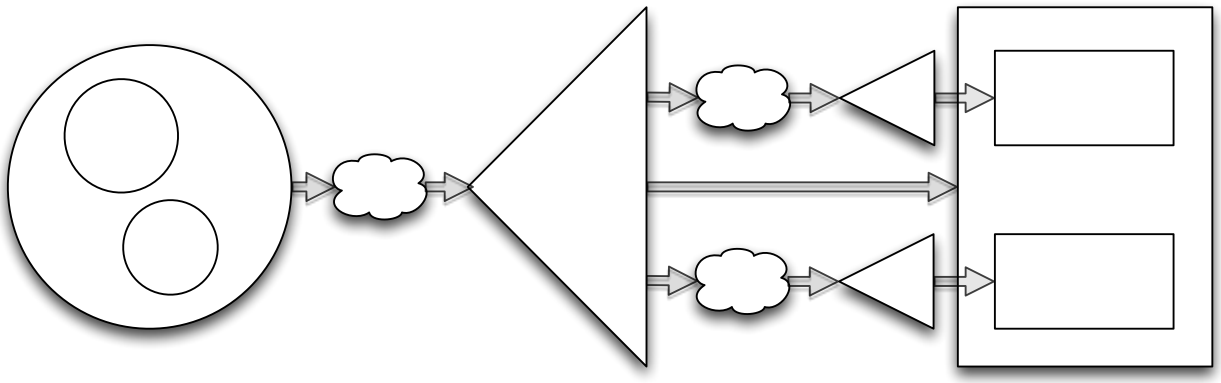
The strong form of the Slamdunk UI Pattern

There are about a half-dozen kinds of Brokers commonly used in practice. They are instantiated and configured, not subclassed. You can make your own unique Broker types if you want to, but you probably won't. Each kind of Broker *delivers* a Model in a different way. Note that there can be a *chain* of Brokers between the Coordinator and the Model.

So now we're saying this:

A Coordinator *creates, configures, and connects* one UI Component for one Model.
A Broker *delivers* one Model to a Coordinator.

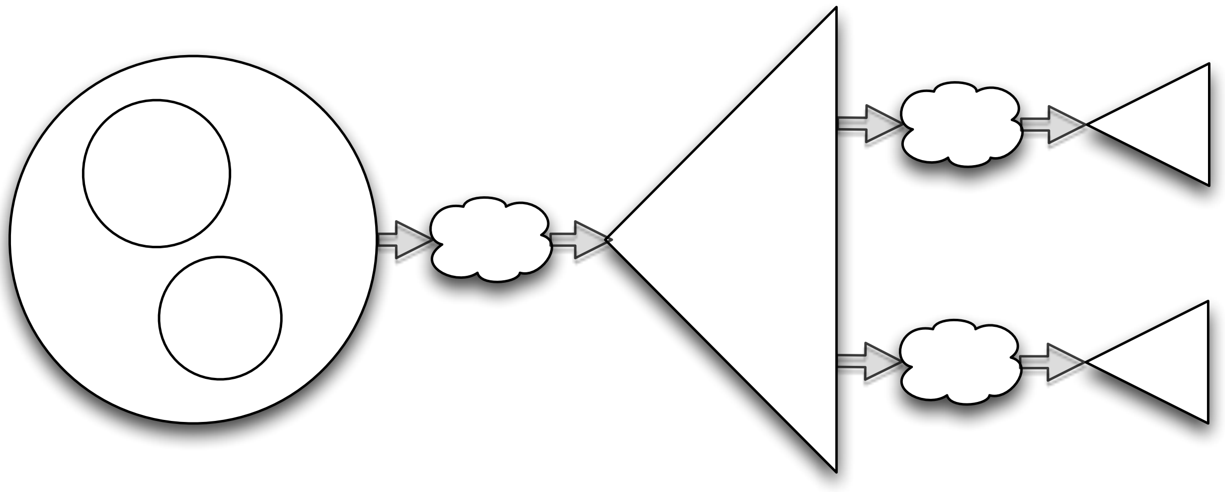
Consider the possibility that the abstract class Coordinator is a subclass of the abstract class Broker, which is itself a subclass of the abstract class Model. Now consider the possibility that a UI is *exclusively* a tree of Brokers and Coordinators.



Composition of Strong Slamdunk Assemblies

Wait a minute... *Does that really work?* Yes, it does. It works so well that building UIs in a programming language like Java becomes pretty much a waste of time. Strong Slamdunk UIs are described in a terse schematic language with high semantic density. Is it XML? It could be. But Lisp S-expressions are more useful, and easier to write, too. Ordinary mortal programmers can learn this stuff in short order. If the word *Lisp* strikes fear into their hearts, just don't tell them that's what they're doing. And if you have a deeper understanding and command of Lisp, you can easily do useful things far beyond the scope of this article. Does this mean you're now programming your UI in Lisp? Not really, or at least not much. In the course of normal Slamdunking, you won't write any CARs or CDRs. Don't think of it as programming at all. Instead, *describe* your UI in this convenient notation. If that notation also happens to be executable, is that a problem? Many runtime environments have Lisp add-ons these days, so you can Lisp when it's helpful, and speak your regular language for everything else.

Let's draw a slightly simpler form of the diagram shown above. This is how strong Slamdunk UIs are designed. This kind of schematic is called a *Windchime Diagram*. Imagine the wind is blowing strongly from left to right, and the name will make sense.



A Slamdunk *Windchime Diagram*

This diagram can then be translated into the following form:

```
(bc (A (B)(C))(cp (lg)(bi "B" (cw))(bi "C" (cw))))
```

If you have a Slamdunk library from `brising.com`, this little bit of text creates, configures, and connects a UI that is *visible* and *works*. With only a few minutes of education, this code will make infinite sense to you and your design collaborators. If you were thinking you were then supposed to translate this back into your favorite programming language, or that some tool generates such code for you, you have missed the point. *This is the code you deploy.*

How Do I Start This Thing?

A common stumbling block when learning Slamdunk is the question "Who invokes `setValue(Model)`?" This is usually obvious when you need to do it, but maybe not before then. Almost always, it happens inside another Coordinator's `setValue(Model)` method. Occasionally, it happens elsewhere. So how do you start an application? Here's how:

1) In your `main()` routine, instantiate your top-level Model. Don't have one? Shame on you! Go back and re-engineer your code until you have a clearly defined top-level Model. If it operates a process or talks to a client/server IO engine, start that now.
2) Instantiate the Coordinator that goes with your top-level Model. Don't have one? Shame on you! We are writing Slamdunk here, which is all about Models and Coordinators. Go back and make one.
3) Obtain your top-level Coordinator's Component, which is probably a Panel. Put that Component in a Window.
4) Invoke `oCoordinator.setValue(oModel)`.

Your application is now visible and running. All Slamdunk applications start this way.

Why Do This?

Here are several important consequences of building UIs with Slamdunk:

1) **There is no Bordello Morality here.** Every Slamdunk UI is a simple tree. There are no strange connections to decipher and understand. If you follow the coding template, most of your code-structuring decisions are already made for you, so you don't waste time inventing things that don't need inventing. An ever-increasing portion of your life is spent writing minor variations of the expression `oCoordinator.setValue(oModel)`. And when you go on vacation or get hit by a truck, other Slamdunk-literate developers will already know everything they need to know about how your UI is structured and where things belong.

There is what some might consider a downside to this. On one Slamdunk UI team I managed, a senior developer quit after a few weeks. At his exit interview, he was asked why he was leaving. His response: "Slamdunk took all the fun out of my job." That may be true. The other thing that was true: as soon as we completed our strong Slamdunk library, we took UI design and development away from R&D and put it into the hands of Field Engineers, who did it successfully while seated in front of impatient customers. Design and development times for *deployable* UIs went from months to minutes. What the smart developer will learn from this is that all the *fun* and *interesting* code found in UI projects is really *complicated* and *unnecessary*. Slamdunk is a simple recipe that never fails.

2) **You can test it.** Any Slamdunk Model-[Broker...]-Coordinator-Component assembly is a complete and closed system. For any Model in your application, you can instantiate a UI on it and test it *completely* in isolation. How? Reread *How Do I Start This Thing?* and recognize that the recipe works for *any* Model, not just the application top-level Model.

3) **You can build UI abstractions that actually work.** Remember the fundamental rule of Slamdunk? *For every View, there is one and only one Model.* Because of this, a Slamdunk UI has properties beyond just being easier to write. How to make use of these properties is beyond the scope of this article, but in summary: Slamdunk-based applications often display real working UIs that no human being designed, and they are *right*. It is hard to overstate the payoff.

Is There More to Know?

Of course there is. But each next step is simple in its own way, bringing a new level of power to your development efforts. We can show you how.

A Little More History

The Slamdunk UI Design Pattern originated in machine control applications written in the Smalltalk language. At first, it was a coding pattern that developers learned to hammer out by hand. It was in that body of programmers that Slamdunk acquired its name. In 1995, Steven T Abell, then at ParcPlace, learned this pattern and firmed it up into a bona fide framework that is still used in the active Smalltalk community. Later, he translated it into Java, in which the first implementation of strong Slamdunk was built for David Taylor's Enterprise Engines project.

© 2018 `brising.com`