aspects of the domain object. For example, the customer form exposes `StringProperty` objects for the customer's first name, last name, street address, city, and zip code. It exposes a `DateProperty` for the customer's club membership expiration date.

Some domain objects would be awkward to expose this way. Connecting a slider that controls dilation of the image or embedded image in a design to the underlying geometry would have required more than half a dozen properties. Having the `Form` juggle this many properties just to drag a slider seemed like a pretty clear code smell. On the other hand, adding another type of property seemed like the path to wild type proliferation.

Instead, we compromised and introduced an object property to hold arbitrary Java objects. The animated discussion before that class appeared included the phrases "slippery slope" and "dumping ground." Fortunately, we kept that impulse in check—one of the perils of a type-checked language, I suppose.

We handled actions by creating a "command property," which encapsulates command objects but also indicates enablement. Therefore, we can bind command property objects to GUI buttons, using changes in the property's enablement to enable or disable the button.

The UI Model allowed us to keep Swing contained within the UI layer itself. It also provided huge benefits in unit testing. Our unit tests could drive the UI Model through its properties and make assertions about the property changes resulting from those actions.

So, forms are not visual themselves, but they expose named, strongly typed properties. Somewhere, those properties must get connected to visible controls. That's the job of the bindings layer.

Bindings

Whereas properties are specific to the types of their values, bindings are specific to individual Swing components. Screens create their own components, and then register bindings to connect those components to the properties of the underlying `Form` objects. An individual screen does not know the concrete type of form it works with, any more than a form knows the concrete type of the screen that attaches to it.

Most of our bindings would update their properties on every GUI change. Text fields would update on each keystroke, for instance. We used that for on-the-fly validation to provide constant, subtle feedback, rather than letting the user enter a bunch of bad data and then yelling at them with a dialog box.

Bindings also handle conversion from the property's object type to a sensible visual representation for their widgets. So, the text field binding knows how to convert integers, Booleans, and dates into text (and back again). Not every binding can handle every value type, though. There's no sensible conversion from an image property to a text field, for example. We made sure that any mismatch would be caught at application startup time.