

LBi ActionScript Project Structure

Michael Forrest

Friday, 2 November 2007

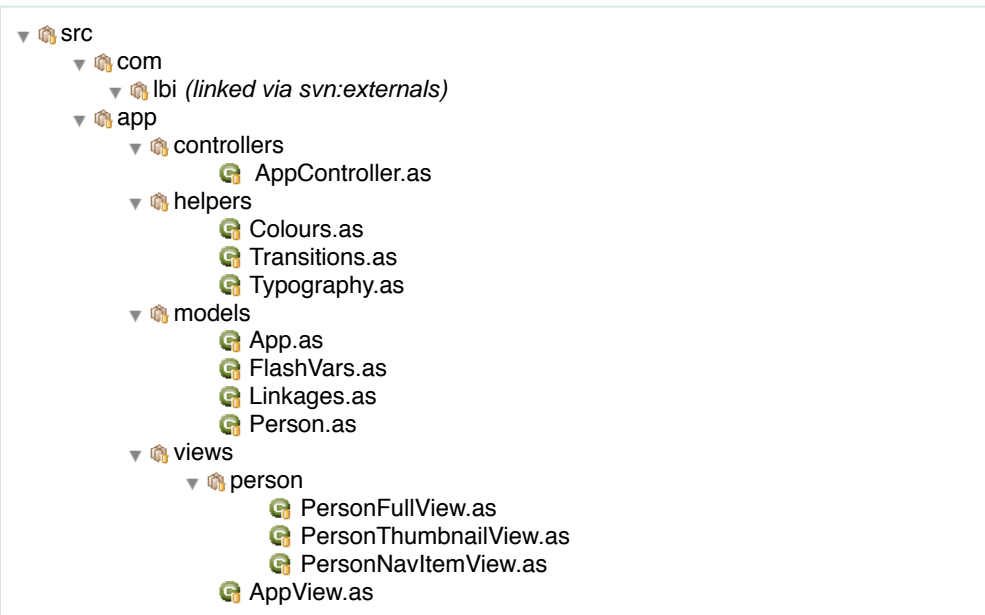
Contents

Introduction	3
Basic Project Structure	3
<i>A word of warning</i>	<i>3</i>
<i>Philosophy employed</i>	<i>3</i>
Standard approach to models and collections	5
ActiveRecord in Flash	5
Solution	5
Views	9
Basic view class	9
Adding Rollovers	10
Animating the rollovers	10
Animating custom properties	11
Modifying animator properties	11
Responding to state changes	12
Creating a view of a collection	16
Running the example	17
Standard models and helpers	18
Controllers	21
Some examples of where you need controllers	22
Unit tests	23
The Test-Driven Development Mantra	23
That's all for now	24

Introduction

This isn't so much a 'framework' as an entire approach encompassing the use of specific tools, libraries and conventions.

Basic Project Structure



(Look at all those resources all safely checked into Subversion ;o)

The four top-level folders will be familiar to Ruby On Rails users. However, Flash is a different sort of beast, requiring extensive support for asynchronous activity and a generally more events-driven structure. So all is perhaps not as it seems...

A word of warning

Some of the conventions outlined in this document may seem somewhat restrictive. This is deliberate. By encouraging and enforcing certain conventions we can drastically simplify many common issues with project maintenance, as well as making it far easier for different developers to work with each other's stuff. If you need to do something especially fruity, that is always perfectly acceptable but **ONLY** if it is the only way to do something! However, a lot of common problems have been solved in many different ways. The point here is to **CHOOSE ONE WAY**. The practices will mutate but this set of practices provides a solid foundation for most Flash development.

Philosophy employed

- Minimise repetition. This extends even to the level of file names, variable names, etc.. if something is repeated more than once, you need to encapsulate.

- Minimise event dispatching. It is very difficult for another developer to follow a process flow when lots of events and listeners are distributed throughout the application. Events are generally ONLY used to...
 - facilitate encapsulation (e.g. objects that dispatch “ready”)
 - allow multiple objects to respond to changes in one object
 - simplify animation (using the lbi.animation library)

Standard approach to models and collections

ActiveRecord in Flash

The LBi Useful library provides classes to mirror the simplicity and ease of use of Ruby On Rails' *ActiveRecord* class. We use similar assumptions to those in Rails, storing collections statically against model classes.

We assume that, as a general case, we are always working with a linear collection of model objects. We want these objects to be

- strictly typed
- stored in a collection held statically against the object class
- automatically retrieved and parsed with only code relating to the translation of an XML structure into ActionScript variables

Solution

Application level

Say we have an XML file with the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<people>
  <person name="Michael" image="img/michael.png" role_id="1"/>
  <person name="Christian" image="img/christian.png" role_id="2"/>
  <person name="Alex" image="img/alex.png" role_id="3"/>
  <person name="Justin" image="img/justin.png" role_id="2"/>
  <person name="Filipe" image="img/filipe.png" role_id="4"/>
</people>
```

To trigger the XML file to be loaded we do the following:

```
var events : EventDispatcher = Person.prepare();
events.addEventListener(Event.READY, Delegate.create(this,onPeopleReady));
```

An *EventDispatcher* is returned so we can listen to something and find out when it's done.

Class Level

The only thing we want in our *Person* class is code that directly relates to the *Person* domain.

This is all we need to parse the above XML:

```
class app.models.Person {
  private static var XML_URL : String = "xml/people.xml";
  private static var COLLECTION_PATH : String = "people.person";
  private static var COLLECTION_NAME : String = "people";

  // access to the collection, in this case
  // is through People.person
  public static var people : Collection;

  public static function prepare() : EventDispatcher{
    return XMLListLoader.initialize( Person,
                                     COLLECTION_NAME,
                                     XML_URL,
                                     COLLECTION_PATH
                                   );
  }

  // the public properties of a Person model
  public var image_url : String;
  public var name : String;
  public var role_id : Number;

  // constructor takes an XMLModel
  public function Person(model : XMLModel) {
    name = model.name;
    image_url = model.image;
    role_id = model.role_id;
  }
}
```

After the *EventDispatcher* returned from the *prepare()* method dispatches *Event.READY*, the typed collection can be accessed through *Person.people*. Note that we didn't even have to extend anything to make this class function this way - it's all wrapped in the *Person.prepare* call.

What if my XML is more complex than the previous example?

XMLModel is simple and powerful. Any list of elements with the same name can be found using Array access, and other properties can be accessed with normal 'dot' syntax. There is also a very recursive findByValue() method that can be used to find items with an ID or class name. Of course, because XML is a text format, XMLModel always returns String objects, and we always type these explicitly.

Examples are always best... Here is the XML for an *Office* model with its address marked up using the [hcard microformat](#).

```
<office name="London">
  <div class="hcard">
    <div class="adr">
      <div class="street-address">1 Naoroji Street</div>
      <div class="locality">Islington</div>
      <div class="region">London</div>
      <div class="postal-code">WC1X 0JD</div>
      <div class="country-name">United Kingdom</div>
    </div>
    <span class="tel">
      <span class="type">work</span>:
      <span class="value">+44 (0)20 7446 7500</span>
    </span>
    <a href="mailto:michael.forrest@lbi.com" class="email">email mike</a>
  </div>
  <div class="links">
    <a href="http://useful.lbi.co.uk">useful.lbi.co.uk</a>
    <a href="http://code.google.com/p/projectsprouts">Sprouts</a>
  </div>
</office>
```

Here's are the bits of the class that would parse the above XML.

```
//constructor receives the <office> node
function Office(model:XMLModel) {
  name = model.name;
  setLocation(model);
  setEmail(model);
  setAddress(model);
  phone = model.findByValue("class", "tel")
              .findByValue("class", "value").toString();
  setLinks(model);
}
private function setLocation(model : XMLModel) : Void {
  // Convert the "x,y" coordinate into a Point object
  var loc : String = model.loc;
  var pair:Array= loc.split(",");
  position = new Point(Number(pair[0]),Number(pair[1]));
}
private function setEmail(model : XMLModel) : Void {
```

```

// create a HyperLink object from the content of the email link
// found by looking for an element with class='email'
var a : XMLModel = model.findByValue("class","email");
email = makeLink(a); // see function below
}
private function setAddress(model : XMLModel) : Void {
// turn the array of divs inside <div class='adr'>
// into an array of String objects
var adr : XMLModel = model.findByValue("class","adr");
address = new Array();
for (var i : Number = 0; i < adr.div.length; i++) {
    var line_model:XMLModel = adr.div[i];
    if(line_model!=null){
        address.push(line_model.toString());
    }
}
}
private function addLinks(model : XMLModel) : Void {
// looks for class='links' and turns into an array of HyperLink objects
links = new Array();
var raw_list : XMLModel = model.findByValue("class","links");
// note that we're looking in raw_list.a - where 'a' is a link
for (var i : Number = 0; i < raw_list.a.length; i++) {
    var raw:XMLModel = raw_list.a[i];
    var link : Hyperlink = makeLink(raw);
    links.push(link);
}
}
private function makeLink(a:XMLModel):Hyperlink{
return new Hyperlink({text:a.href,target:a.target});
}
}

```

Nothing too spectacular really - but also nothing where it isn't obvious what's going on.

What if I have more than one set of data in one XML file?

Chances are that at the top level of your file you will have a list of some kind or other. If so, I recommend simply reloading the same file in different Models and relying on the browser's cache. This is the best way to retain the elegance of the solution

What if I have different data sets that need to be parsed in similar ways and stored in different collections?

Make a superclass with instance methods that do the parsing, and then create a subclass for each collection, and hold each collection on the appropriate subclass.

I only have one item - I don't have a collection

You can still use this - just copy the value from (in this case) `people.first()` into your class.

Views

A view is a view of an model instance or a Collection of model instances. Always.

We take the view that you are likely to need multiple views to reference the same models, and so at the core of our approach we decide that *views always update their state in response to events dispatched from their models*.

Basic view class

Here's a basic view class.

```
class example.app.views.people.PersonThumbnailView extends ViewBase {
  // this is a view of a person, so...
  private var person : Person;

  // view elements
  private var photo : Loader;
  private var label : TextField;

  // the static create method is created on any view with an FDT template.
  // we keep the dynamically passed arguments next to the constructor
  public static function create (    $person : Person,
                                     $parent : MovieClip,
                                     $depth : Number ):PersonThumbnailView
  {
    return Attach.MovieClipClass($parent,
                                  PersonThumbnailView,
                                  $depth,
                                  [$person]);
  }
  function PersonThumbnailView($person : Person) {
    person = $person;
    addPhoto();
    addLabel();
  }
  private function addPhoto() : Void {
    photo = Loader.create(this,getNextHighestDepth());
    photo.load(person.image_url);
  }
  private function addLabel() : Void {
    var style : Typography = new Typography("Name");
    label = Attach.TextField(this,style,getNextHighestDepth());
    label.text = person.name;
  }
}
```

This class can be instantiated after the Person class has loaded by calling, for example:

```
var person:Person = Person.people[0];
PersonThumbnailView.create(person, this, getNextHighestDepth());
```

This gives us a labelled view of the first person in Person.people. Labelled with their name.

We're using Loader to load in an image (you can add a "ready" event listener if you need to know when it's loaded the image).

Note the use of the Typography class and Attach.TextField methods. Here, we are using the "Name" style.

Adding Rollovers

Okay- that's great, but this is Flash. It should be flashy. It should have rollovers. Here's a rough and ready `_alpha`-based rollover:

```
private function doRollOver() : Void {
    _alpha = 100;
    label._visible = true;
}
private function doRollOut() : Void {
    _alpha = 50;
    label._visible = false;
}
```

We'll wire up those rollovers with:

```
setRollovers(doRollOver,doRollOut);
```

Why not just override `onRollOver` and `onRollOut`? Because we need to do `onReleaseOutside` too. `setRollovers` just wraps this.

Animating the rollovers

Now here's a clever bit. Say we want the rollout to animate smoothly back to `_alpha = 50` instead of going straight there. Easy. Just change

```
_alpha = 50;
```

to

```
animator._alpha = 50;
```

The `animator` member is automatically created on any `ViewBase` class. This encapsulates a whole lot of functionality that you'd usually end up doing yourself through Tween or Animation objects. There's nothing stopping you making these animations yourself, but this approach covers about 95% of the cases you're likely to need. And you can experiment freely without the pain.

Oh - you needed to register that `_alpha` animation. I should have mentioned.

In your constructor you need to put:

```
animator.registerTransition("_alpha");
```

If you have lots of properties to animate you can use `registerTransitions`:

```
animator.registerTransitions(["_alpha", "_x", "_yscale",]);
```

You probably don't want to do an animation on those properties together because it will look mental.

Animating custom properties

If you create an accessor on your view (use the *FDT template accessor*) then you can just as easily register that on your animator.

For example, say we want to do something fancy where we blur out the image and make the text build in on each rollover / rollout. Let's define a "highlight" property and register it.

```
private var __highlight:Number = 1;
public function get highlight():Number{
    return __highlight;
}
public function set highlight(v:Number):Void{
    __highlight = v;
    var blur : Number = (1-v) * 10;
    filters = [new BlurFilter(blur,blur,3)];
    label.text = person.name.substr(0,person.name.length*v);
    _y = blur;
}
```

So now we can assign highlight=any value (ideally between 0 and 1 if we don't want insanity) and we'll get a nice blurry movey text-buildy animation.

So we'll register it and change our rollover/rollout handlers.

Constructor:

```
highlight = 0; // let's set the default while we're here
animator.registerTransition("highlight");
```

Handlers:

```
private function doRollOver() : Void {
    animator.highlight = 1;
}
private function doRollOut() : Void {
    animator.highlight = 0;
}
```

And now we get a nice animation.

Note that we've still only written code that directly relates to how we want our transition to work! We're not doing monkey work like creating and destroying tweens or anything like that - we're just assigning a property. If you want the highlight to be instant, you can just remove the animator.bit and set it instantaneously on rollover.

Modifying animator properties

You will probably want to override the default animation settings each time you use *animator*. Do this like this:

```
private function doRollOver() : Void {
    animator.change("highlight").setFrames(3).setEasing(Easing.easeOutCirc);
```

```
        animator.highlight = 1;
    }
    private function doRollOut() : Void {
        animator.change("highlight").setFrames(10);
        animator.highlight = 0;
    }
}
```

Responding to state changes

This is all very well, but what if we have more than one view of a person? (There's always more than one view of each model. Don't forget that.)

We need to make sure that if the model gets changed in one place then it gets simultaneously updated everywhere it is represented.

Let us, for now, directly modify the properties of our *Person* model from the *PersonThumbnailView*. As we get more advanced we'll pipe this through a Controller in case there are complex dependencies between different model types, but for now, it's safe to say we can change person directly.

Note that our view class's *person* field is a reference, not a copy, so each person is only represented once throughout the entire application.

Deciding on a behaviour

Let's make it so that only one person can be 'selected' at a time. This is easily achieved through the *Collection* class. Let's add an *onPress* override (because *onPress* is much easier to work with than *onRelease*!)

```
private function onPress(): Void{
    Person.people.select(person);
}
```

We've gone to the static *Person.people* collection and told it to select our thumbnail's *person* model.

A note on the internals of *Collection*: When you call *select*, the item you want is exclusively selected. If you have made your model class *Selectable*, then this can be easily mapped through to the model's state and event dispatchment.

We need to make a *Person* instance extend *Selectable* to make this work.

Because we need our *Person* instances to dispatch events, we need to employ the services of *EventDispatcher*. Actually, we're going to use *EventManager*, so that we can write less code when wiring up events. Just add the *IEventManager* interface and call *EventManager.initialize(this)*.

Note that this could have happened the other way round - we could have extended *EventManager* and used *ISelectable* and *Selectable.initialize(this)*.

The *Person* class is declared as follows:

Except there's currently a bug in *Selectable.initialize()*;

```
class example.app.models.Person extends Selectable implements IMapper
```

You will get compile errors if you do not extend (at the very least) `EventDispatcher`. That's because you need to be able to dispatch events for `Selectable` to do its work.

Person constructor contains:

```
EventMapper.initialize(this);
```

Mapping events

We could map events using the normal `EventDispatcher.addListener` method, like so:

```
person.addListener(Selectable.SELECTED,
                  Delegate.create(this, respondToSelected));
person.addListener(Selectable.DESELECTED,
                  Delegate.create(this, respondToDeselected));
```

But that's a bit repetitive, especially if we have a lot of events to respond to. Too much copy-and-paste.

`EventMapper` allows us to say:

```
person.registerEvents(this, [Selectable.SELECTED,
                             Selectable.DESELECTED])
```

The advantage of this method is that we enforce the convention of naming event handlers to corresponding `respondToEventName` events. Making our code easier for other developers to understand, as well as meaning we can write less code to achieve the same results. In this system, an event called `"selected"` gets mapped to a handler called `respondToSelected`. We map underscores to pascalCasing, thus:

```
"changed" => respondToChanged
```

So here, we say

```
Selectable.CHANGED => "changed" => respondToChanged
```

Static event names should always map directly to string versions of the same name. We use the static access to make it easier to remember what events classes have available.

If you don't have the correct dispatcher methods on your model class, or the correct `respondTo` events in your view, you will see errors in your error log telling you what's missing.

`PersonThumbnailView` constructor is now :

```
person.registerEvents(this, [Selectable.SELECTED,
                             Selectable.DESELECTED],
                    "PersonThumbnailView"); // There is a template for this
// putting the calling class name makes the application easier to debug
```

This requires the methods `respondToSelected` and `respondToDeselected`. We'll also add a line to `doRollOut`. Here's the view class now:

```

class example.app.views.people.PersonThumbnailView extends ViewBase {
    // this is a view of a person, so...
    private var person : Person;

    // view elements
    private var photo : Loader;
    private var label : TextField;

    private var __highlight:Number;
    public function get highlight():Number{
        return __highlight;
    }
    public function set highlight(v:Number):Void{
        __highlight = v;
        var blur : Number = (1-v) * 10;
        filters = [new BlurFilter(blur,blur,3)];
        label.text = person.name.substr(0,person.name.length*v);
        _y = blur;
    }

    public static function create($person : Person,
                                $parent : MovieClip,
                                $depth : Number ) : PersonThumbnailView
    {
        return Attach.MovieClipClass($parent,PersonThumbnailView,
                                     $depth, [$person]);
    }
    function PersonThumbnailView($person : Person) {
        super();
        person = $person;
        addPhoto();
        addLabel();
        highlight = 0;
        animator.registerTransition("highlight");
        setRollOvers(doRollOver,doRollOut);
        person.registerEvents(this, [Selectable.SELECTED,
                                     Selectable.DESELECTED],
                              "PersonThumbnailView");
    }
    private function addPhoto() : Void {
        photo = Loader.create(this,getNextHighestDepth());
        photo.load(person.image_url);
    }
    private function addLabel() : Void {
        var style : Typography = new Typography("ToolTip");
        label = Attach.TextField(this,style,getNextHighestDepth());
        label.text = person.name;
        TypographyHelper.fitHorizontally(label);
    }

    private function doRollOver() : Void {

```

```

        animator.change("highlight").setFrames(3);
        animator.highlight = 1;
    }
    private function doRollOut() : Void {
        // don't want a rollout if the object is selected
        if(person.isSelected()) return;
        animator.change("highlight").setFrames(10);
        animator.highlight = 0;
    }
    private function onPress(): Void{
        Person.people.select(person);
    }
    private function respondToSelected() : Void {
        doRollOver();
    }
    private function respondToDeselected () : Void {
        doRollOut();
    }
}

```

Still less than 60 lines of code.

We've changed our Person class too - let's see where that is now:

```

class example.app.models.Person extends Selectable implements IEventMapper{
    private static var XML_URL : String = "xml/people.xml";
    private static var COLLECTION_PATH : String = "people.person";
    private static var COLLECTION_NAME : String = "people";

    public static var people : Collection;

    public static function prepare() : EventDispatcher{
        return XMLListLoader.initialize(
            Person,
            COLLECTION_NAME,
            XML_URL,
            COLLECTION_PATH
        );
    }
    // the public properties of a Person model
    public var image_url : String;
    public var name : String;
    public var role_id : Number;

    // constructor takes an xmlmodel
    public function Person(model : XMLModel) {
        EventMapper.initialize(this);
        name = model.name;
        image_url = model.image;
        role_id = model.role_id;
        deselect();
    }
}

```

```

// These methods are autogenerated by FDT and left empty.
// Our call to EventMapper.initialize(this) fills these methods in.
public function registerEvents(view : ViewBase, events : Array,
                               class_name_for_debugging : String) : Void {
}
public function getPossibleEvents() : Array {
    return null;
}
public function eventIsPossible(event_name : String) : Boolean {
    return null;
}
}

```

Creating a view of a collection

Let's make a little thumbnail menu.

```

// Note the name of the class. Singular vs plural is an important
// distinction when naming things and can be an easy way
// to enhance the readability of your code
class example.app.views.people.PeopleMenuView extends ViewBase {
    // model
    private var people : Collection;

    // layout constants
    private var THUMBNAIL_SPACING : Number = 50;

    // Again - create() was created with the FDT template
    public static function create ( $parent : MovieClip,
                                    $depth : Number ) : PeopleMenuView
    {
        return Attach.MovieClipClass($parent, PeopleMenuView, $depth);
    }
    function PeopleMenuView() {
        super();
        people = Person.people;
        for (var i : Number = 0; i < people.length; i++) {
            var person:Person = people[i];
            addThumbnailForPerson(person, i);
        }
    }
    private function addThumbnailForPerson(person : Person,
                                            index : Number) : Void {
        var thumb : PersonThumbnailView =
            PersonThumbnailView.create(person, this,
                                       getNextHighestDepth());
        thumb._x = index * THUMBNAIL_SPACING;
    }
}

```

You know what? Why don't you just check out the project from Subversion and run it.

Running the example

1. Skip this step if you've run a Sprouts AS2 project before

First make sure you have Ruby installed - install this from a one-click installer from <http://www.ruby-lang.org>

Then... (you will need *sudo* for the *gem* bit if you're not *root*)

```
$ gem install Sprout
[installs the Sprout gem]
$ sprout -s as2 DummyProject
[downloads the flash debug player, mtasc, swfmill, etc...]
$ rm -fr DummyProject
[removes the temporary DummyProject]
```

Then...

2. Check the project out of Subversion and build with rake

```
$ svn co http://lbi-useful-actionscript-2.googlecode.com/svn/trunk/LBiUseful
[...]
$ cd LBiUseful/project
$ rake example
```

And you should be able to see the cheesy example in action.

Standard models and helpers

app.models.FlashVars

All application parameters should be statically accessed via this class - a typical FlashVars class looks something like the following:

```
class app.models.FlashVars {
  public static function getXMLUrl() : String {
    // “||” usually works
    return _root["xml_url"] || "../xml/case_studies.xml";
  }
  public static function getImageRootPath() : String {
    return _root["image_root_url"] || "../assets/menu_images";
  }
  public static function getSWFRootPath() : String {
    return _root["swf_root_url"] || "../assets/swf";
  }
}
```

app.models.Linkages

The Linkages class is where you keep track of any assets that are embedded in your swf. It's the library. You shouldn't put references to embedded assets in application classes, but always statically access them via this class. This becomes a place you can cross-references with your SWFMill file.

In future this file may be autogenerated by the rakefile to automatically match and provide a reference to the contents of the template file SWFMillTemplate.erb

Here's an example of what you'd expect to see:

```
class app.models.Linkages {
  public static var MINUS_UP : String = "minus_up";
  public static var PLUS_UP : String = "plus_up";
  public static var MINUS_OVER : String = "minus_over";
  public static var PLUS_OVER : String = "plus_over";
  public static var LINK_BULLET : String = "link_bullet";
  public static var ARIAL : String = "arial";
  public static var ARIAL_BOLD : String = "arial_bold";
}
```

This provides code completion and creates a clean separation of the library from the application.

app.helpers.Typeography

All typography in the application should be defined in this class. Naming conventions have been twisted a bit to make this look more like a CSS stylesheet.

For example:

```
class app.helpers.Typography extends TypographyBase {
  // All styles call this method first:
  private function Defaults() : Void {
    font_size = 10;
    font = Linkages.ARIAL; // access fonts via Linkages class
    word_wrap = false;
    multiline = false;
  }
  // First set of 'selectors' resemble those used in HTML
  private function H1() : Void {
    font_size = 24;
    auto_size = null;
  }
  private function H2() : Void {
    font_size = 16;
    font = Linkages.ARIAL_BOLD;
    // keep colours somewhere where other
    // parts of the application can use them:
    colour = Colours.ORANGE;
  }
  private function A() : Void {
    colour = Colours.GREEN;
    font_size = 11;
  }
  private function CloseButton() : Void {
    A(); // this is how you do inheritance
    font_size = 12;
  }
  function OfficeName(): Void {
    font_size = 16;
  }
}
```

All TextField objects are instantiated using code in the following format:

```
label = Attach.TextField(this, new Typography("H2"), getNextHighestDepth());
```

The string "H2" above is mapped onto the H2 method in the Typography class. In a complex application it is worth assigning these names as constants on the Typography class, e.g.

In Typography:

```
public static var OFFICE_NAME : String = "OfficeName";
```

In the View class:

```
label = Attach.TextField(this,
  new Typography(Typography.OFFICE_NAME),
  getNextHighestDepth());
```

app.helpers.Transitions

A good way to simplify the animation process in your application is to keep all transition information in one place that can be referenced by your different views. This way you have less need to rely on a complex event management systems within nested views in order to make sure all your timings work out properly. A word of caution though: this approach falls over if you depend on asynchronous network operations for dependencies within transitions. In this case, you need to use the Stack class to make sure a sequence is complete before moving to the next animation group.

Example of a Transitions class.

```
class app.helpers.Transitions {
    public static var MAP_PAN_FRAMES : Number = 25;
    public static var DETAIL_FADE_FRAMES : Number = 10;
    public static var DETAIL_REVEAL_FRAMES : Number = 15;
    public static var ONE_SECOND : Number = 25;
}
```

Not much to it really, but it can come in handy. We have some namespacing of variables here (the MAP_ and DETAIL_ prefixes) which is not ideal but you can see everything relating to timing in one place.

Controllers

Sometimes you have more than one view changing your models. It is a good idea to make all the views go to the same place before they start doing things. Most of the time you can get away with a central controller for your application called *AppController*. I'm just going to put an example here- it's not to do with *People*, but I want to get this document out ;)

```
class app.controllers.MapController{
  private static var instance : MapController;
  private var map : Map;
  private var offices : Collection;

  public function MapController() {
    map = Map.getInstance();
    Key.addListener(this);
    Mouse.addListener(this);
  }
  public static function getInstance(): MapController {
    if(!instance) instance = new MapController();
    return instance;
  }
  public function changeTimeZone(increment : Number) : Void {
    map.changeTimeZone(increment);
  }
  public function setZone(zone : Zone) : Void {
    map.setZone(zone);
  }
  public function showOffice(office : Office) : Void {
    map.selectOffice(office);
  }
  public function closeDetail() : Void {
    map.deselectOffice();
  }
  private function onKeyDown() : Void {
    switch(Key.getCode()){
      case Key.LEFT : map.changeTimeZone(-1); break;
      case Key.RIGHT : map.changeTimeZone(1); break;
      case Key.ESCAPE : map.deselectOffice(); break;
    }
  }
  private function onMouseWheel(delta:Number) :Void{
    map.changeTimeZone(delta>0?-1:1);
  }
}
```

A controller class really is a judgement call. You need to use your OOP / design patterns / code smells knowledge to keep it small, and to break it up into multiple controllers where necessary. The complexity of your application dictates the number of controllers you need.

Some examples of where you need controllers

- If you are switching to different contexts / 'sections' within your application
- If you have views that might interrupt other views and you want to encapsulate this all in one place

Unit tests

Should come first. Everybody always says this. The key place to put your unit tests is against your model classes. Check that data is converted correctly. Write all code in response to tests. Yes, this is a big subject.

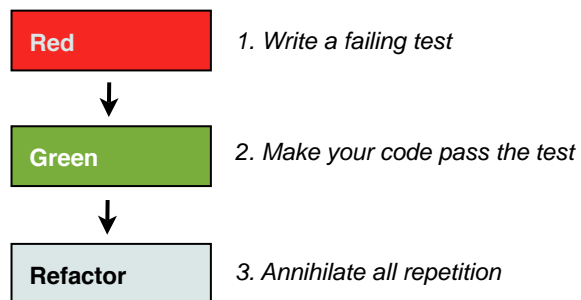
Look at the tests in the LBiUseful project/test folder. You can run these tests with:

```
rake test
```

If they don't all pass, there is a problem ;)

These are framework-related tests, but have a look through and you'll see why they're important.

The Test-Driven Development Mantra



Put everything you can under test. It is more work up front, but

- it will clarify your thinking around the application domain
- put you in a *usage-orientated* mindset (as opposed to *implementation-oriented* state)
- make future changes immeasurably safer and more pleasurable to make

That's all for now

Yes yes yes, it all needs to be done for AS3.

I haven't said anything about com.lbi.animation really. You can have a look at some examples on <http://useful.lbi.co.uk>

If something doesn't make sense, or you disagree, talk to me.