Framework Design

Oliver Szymanski, David Tanzer

Schlüsselworte
Framework, Library, Readability, Usability, Abstraction, Simplification, Extensibility, Succinctness

Introduction

Designing a framework or a library is hard, just like good software design in general. In this paper we explain five principles that are important for good framework design: Readability, Usability, Abstraction and Simplification, Extensibility and Succinctness. Based on these principles you can create your own frameworks or maybe even more important check if a framework is worth using it.

Readability

A framework or library should make it really easy to write readable code. Often we have to read more code than we write, so reading code can be more important than writing code. That makes writing readable code mandatory.
Reading code means trying to figure out what it does. This means processing a lot of information in our mind - within our "working memory" (C.f. “Visual Language for Designers” by Connie Malamed, ISBN-13: 987-1-59253-515-1).

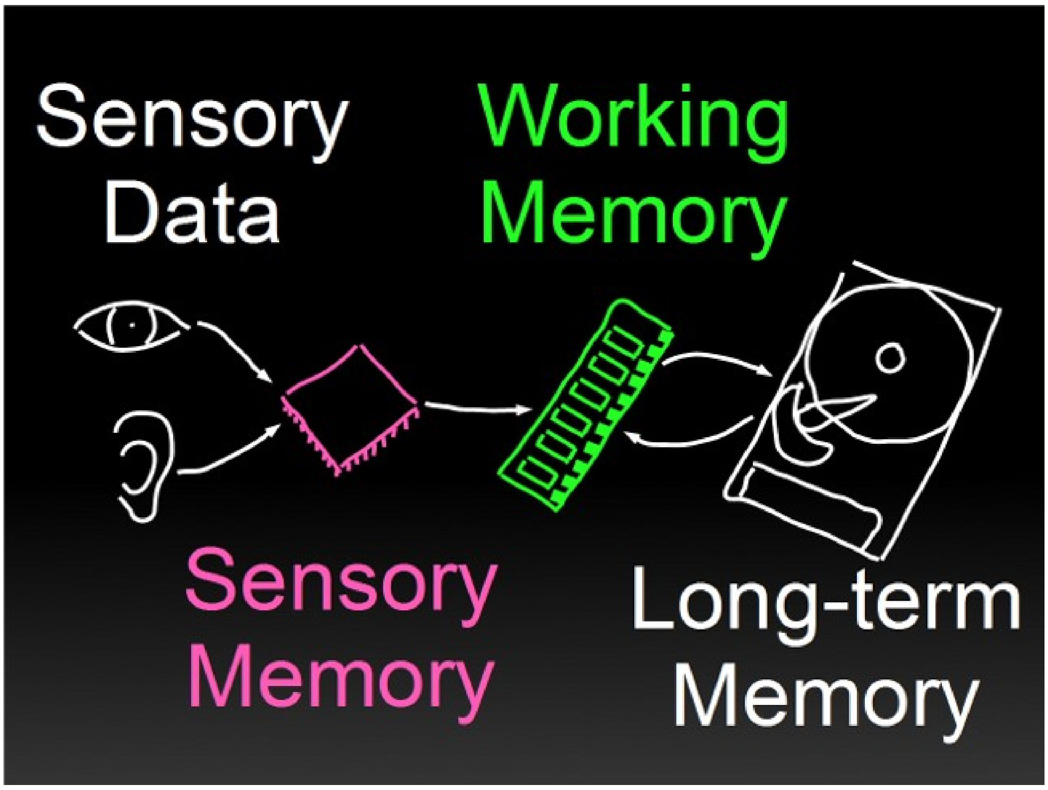


Abb. 1: How our brain processes information

Unfortunately our working memory is limited to about 3-5 items. If we have to process more items they have to be constantly "loaded" from long term memory or our sensory organs. The good news is: The size of these items is almost unlimited. By building junks of information we can keep more in

working memory. When we think of the term "context free grammar" we can process it as a single concept - that is, one item, not as a sequence of 20 letters.
But: What does this have to do with readability? The idea is pretty simple: Help the reader to build junks of information and to minimize "cache misses" - that is, loading from long term memory. To achieve this, the immediate surroundings of some piece of code should give the user hints about how it works and how to modify it.
Fluent interfaces can help here. Let's start with a negative example. Some frameworks force their users to write code like this:

```
new User(
    „David‟, „Tanzer‟,
    „foo@bar.com‟,
    „secret‟,
    Roles.USER);
```

While this code is quite easy to write, it can be really hard to read. There is no way to find out what the parameters mean by looking at this code. You either have to read the API documention or open the User-class. You might say that this is not a lot of work, but still, it is annoying. Nobody reads the documentation, don't make me do it.
There's a nice solution to this problem called "fluent interfaces" or "method chaining": Instead of one method/constructor with lots of parameters define lots of methods with a single parameter. This example is from Apache Wicket:

```
add(new Label("id")
    .setMarkupId("mid")
    .setVisible(true)
    .setEnabled(false)
);
```

It is a little bit harder to write (but still no problem when you have auto complete in your IDE) and much easier to read. A lot of the things you need to know are right there, before our eyes.
There are two minor problems with this approach: First, you cannot use it for mandatory parameters (because you cannot guarantee that all the methods are called). Second, in a statically typed language like Java, once you call a method from the base class, the compiler does not know the child class methods anymore. The compiler could fix that, but I would not wait for it. You could use generics, but that can become ugly.
Named Parameters are another technique to improve readability. Smalltalk, for example, has named parameters:

```
collection copyFrom: 1 to: 10
```

This code sends the message "copyFrom:to:" to the object "collection". The parameters are called "copyFrom" and "to", their values are "1" and "10". Unfortunately, in Java we don't have a feature like this. We could simulate named parameters when initializing objects using Double Brace Initialization:

```
persistUser(
    new User(){{
        userName="david";
        password="secret";
}});
```

But this also has some disadvantages. First, it cannot be used for mandatory parameters (same reason as above). Second, this creates an anonymous class that is derived from "User". So, if the class "User" uses a naive implementaion of "equals" like the following (very common in java) it does not work.

```
public boolean equals(Object other) {
    //...check for null...
    if(getClass().equals(other.getClass) {
        //...test if the objects are equal...
    }
    return false;
}
```

And third: A lot of Java programmers don't know this. But I guess that does not count as an argument because every programmer should be able to learn it.
There are other techniques we can use to allow users of our framework or library to write readable code. The Single responsibility principle (SRP) and Single Level of Abstraction (SLA) come to mid. Ralf Westphal has a good step by step explanation of SLA in his blog:
http://ralfw.blogspot.com/2009/07/partial-classes-helfen-dem-single-level.html.
When writing code that others have to use, we should think more often: "Does this allow the users of my code to write easier readable code?"

Usability
A framework or library should make it really easy to write code. It should be obvious how the library can be used and how the functionality can be extended. Also, we should help our users save keystrokes by supporting the auto complete features of our IDEs. And we should make it easy to write testable code and to find errors in the resulting code.
Don't repeat yourself (“DRY”). And by "yourself" I don't mean "yourself" (the framework developer), I mean the framework users. That is: Allow your users to follow this principle - do not force them to repeat themselves. For example, Ruby on Rails has several features that enable writing code without repeating stuff.
With other frameworks, DRY is very hard or even impossible. Take android for example. This is some content of a layout file for an activity:

```
<Button android:id="@+id/submitButton"/>
```

But wait! The id has to be unique across all activities, so better prefix it:

```
<Button android:id="@+id/myactivity_submitButton"/>
```

Now we repeated some information: That the button "submitButton" is part of the activity "myActivity". When you write the Java class for this activity it gets worse. You have to tell the system that yes, the layout "myactivity.xml" is the layout for "MyActivity" and that the "Button" object with the name "submitButton" corresponds to the button with the id "submitButton" from the xml file:

```
setContentView(R.layout.myactivity);
Button submitButton
    =(Button)findViewById(R.id.myactivity_submitButton);
```

In a similar way, Apache Wicket forces you to repeat some information from the HTML files in the corresponding Java classes: You have to create the same components and the same structure in the Java class. This is one of the reasons why Oliver Szymanski and I started the JSXP project.

Another important usability issue is auto complete. Never underestimate how much help the auto complete feature of your IDE can be for your users - and how annoying it can be if your framework does not support it well!
Just try this in a Java IDE:

```
JButton button = new JButton();
button.set[Ctrl]+[Space]
```

(Or whatever your shortcut for auto complete is) You'll get an endless list of methods without any way to sort or filter them. For example, from this list you cannot filter the methods that only effect the view.
Help your users to find or - even better - avoid errors. Let me start with a negative example again: In Apache Wicket, you have to match literal strings from the HTML file in your Java code: The "wicket:id"s from the HTML and the ids in Java must be the same. And the components must be nested in the same way.
And you can only find errors by starting the application - or unit tests with WicketTester. You should absolutely use WicketTester. So, yes, you can find these kinds of errors automatically. But the situation is still pretty bad:

- It is really easy to produce these kinds of errors
- You find out pretty late - only when running the program
- There are a lot of "false positives": Changes that should be compatible but still cause this error, because the structure has changed a little bit.

The thing is: Many of us use a statically typed language. And this has some advantages, at least it should have. One of the advantages is that we know very much about a program during compile time. We should use this knowlege to detect errors early - And most of the time, this is not done, so I see a lot of wasted potential here.
Actually, we don't know anything about the program. The compiler does. So, if we can somehow phrase our expectations about the program in a way the compiler understands the compiler could make sure that our expectations are met. This is easier than you might think: The Java compiler understands Java, so we have to write about our expectations in Java. And to make sure the error is not caused by our expectations, we should generate this code.
I guess you think right now "an example would be really handy at this point". Well, ok ;) The second reason Oliver and I wrote JSXP was to eliminate certain kinds of errors.
When you defined an element in the view (HTML file) you should be able to make sure that it is available in the controller (Java file). And when you change these files in an incompatible way, you get a compiler error. A code generator is responsible for bridging the gap between HTML and Java: From the HTML file we generate code that lets you access the elements in a checked, type safe way, by calling a method. When you change the HTML or the Java code in an incompatible way, the method call cannot succeed, so the compiler shows you an error.

Make writing software based on your library or framework easy for your users! Make sure they don't have to repeat certain kinds of information. Think about auto complete when designing public interfaces. Make testing easy and make sure certain errors cannot be made or detect them at compile time.

Abstraction and Simplification
When writing a framework or library, we have to simplify things and create abstractions. But: With every new concept or abstraction, our users have to learn more. They cannot simply forget about the underlying stuff. And they cannot simply ignore other concepts. This does not have to be a problem,

we just have to think about it.
Abstractions make working on large code bases possible: We don't want to deal with low level stuff. In fact, we cannot deal with it - A large project would not be feasible if we had to write it in assembler. Or if we had to take care about disk I/O or scheduling. So, we need libraries that create new abstractions and/or simplify things for us.
But, as Joel Spolsky has said: “All non-trivial abstractions, to some degree, are leaky”. This is called the “Law of leaky abstractions”.
This means that the abstraction, in some cases, does not work anymore. The user has to "dig deeper". The underlying concepts "leak" through the abstraction. For example, Apache Wicket tries to hide the fact that we are dealing with HTTP, a stateless protocol. Writing a web application with Wicket feels a little bit like writing a desktop application: There is a flow of events, components pass data to callback methods, etc. The communication over HTTP is not an issue, ever.
Except when it is. Every wicket component has a strange method:

```
setOutputMarkupPlaceholderTag(boolean)
```

To really understand what this method does and why it is needed, one has to understand how AJAX works and that we deal with a stateless protocol here. This method is only needed when a hidden component should be manipulated with an AJAX call. Hidden components are normally not written to the output (HTML) at all. The AJAX request is a new, stateless HTTP request, and the only thing at the client side that can be manipulated is the HTML from the last (stateless) HTTP request. Because of this, when an AJAX result should manipulate the hidden element, there has to be a HTML element which can be replaced with the AJAX result - the "Markup Placeholder Tag".
Another example is Moq, a neat mocking framework for .NET. Moq is great: It uses lambda expressions to configure mocks - They are checked by the compiler at compile time and refactoring safe. Configuring the return value of a method looks like this:

```
mock.Setup(
    foo =>foo.DoSomething(It.IsAny())
    ).Returns(true);
```

This means: If a method called DoSomething is called on a given mock object ("foo") with a string parameter (It.IsAny() - the value of the string doesn't matter), then return true. There is some magic going on behind the scenes to make this work, but normally you don't have to care about that. Except when the magic stops working: Like, when you want to mock a protected method. Then, suddenly, you have to write different code:

```
using Moq.Protected()
...
mock.Protected()
    .Setup("DoSomething", ItExpr.IsAny<string>())
    .Returns(true);
```

If you want to truly understand why this difference exists you'd have to understand how the magic works in the first place and why this is not possible with protected methods. The easier way is: Just use the second code snippet for protected methods. This probably works for most of the users of this framework, but when there are unexpected problems, they still have to dig deeper.
So, all abstractions we create will be leaky. But we have to create them, otherwise developing large system might not be possible anymore some time in the future. For every new concept our users potentially have to deal with 2 concepts: The new one, and the old one. But inventing and creating new concepts is part of the essence of our job as programmers.

Are these "leaks" a bad thing? Not necessarily, I think. We just have to be aware of them when designing a framework. And we have to make sure that they happen at defined places, and in defined situations. We have to know the limits of our abstractions and concepts. And we have to look at our framework or library from the perspective of our potential users. Both are a good idea anyway.
Anyway, we should be really careful with creating new abstractions in the first place. We are always at risk of causing more trouble for our users than what they get as benefit: They have to remember more concepts and their programs become potentially harder to read, write and debug.

Extensibility
How can we make sure that our code is extensible? How can we enable or even encourage the users of our library to use the library in new, interesting and unexpected ways? First of all, Simplicity is really important here. Also, we should support inversion of control (IOC) - and if possible we should not constrain our users to a specific IOC container. You can deliberately create "leaks" in your abstractions. And, last but not least, lambda expressions and closures can be an awesome tool for providing extensibility.
Only simple code is extensible. Useless abstractions in the code, trying to anticipate future requirements and even adding design patterns where they are not needed can make the code unnecessary complicated. Just as I have mentioned in my blog post "Simplicity", the benefits of these things are local and grow linear, while the drawbacks have global effects and grow exponentially.
To ensure extensibility it is vital to keep things simple. There are some principles we can keep in mind that are based on the idea of simplicity. The Single Responsibility Principle, the Single Level of Abstraction Principle and the Direct Call Pattern are only a few of them. See also SOLID object oriented design.
Inversion of control - and especially dependency injection - can make extending existing code easier (even if it violates the direct call pattern - at least somewhat ;)). This does not mean that your framework should be an IOC container. It also does not mean that you should force the users of your framework to use a specific IOC container.
You should just make sure that the framework or library you created can be used with an IOC container. Wicket, for example, lets you specify an "Injector" which is called every time a component is created to inject dependencies into this component. There are several Injector-implementations that support popular IOC containers, like Spring or Google Guice.
Your library and framework should create new Abstractions and/or Simplifications. These abstractions will be (to some degree) leaky. But is this a bad thing?
Not necessarily. You can even create such "leaks" deliberately if it makes extending the framework easier! For example, Apache Wicket abstracts the fact that you are dealing with a web container. But sometimes, when writing a web application, direct access of the container would be handy! And it does not really make sense for the wicket developers to wrap all of the functionality of the container. So, when you really need access to the container, you can get it:

```
getRequest().getContainerRequest()
```

You can do this too: Create your abstractions for the common case - for the 90% of the use cases where creating abstractions is easy. But, when these abstractions are not enough, allow your users to access the underlying technology in a controlled and defined way.
Another way to create extensible code is with lambda expressions and closures. Paul Graham posted the following challenge in his essay Revenge of the Nerds:

„We want to write a function that generates accumulators-- a function that takes a number n, and returns a function that takes another number i and returns n incremented by i.
(That's incremented by, not plus. An accumulator has to accumulate.)“

In Javascript the solution looks like this:

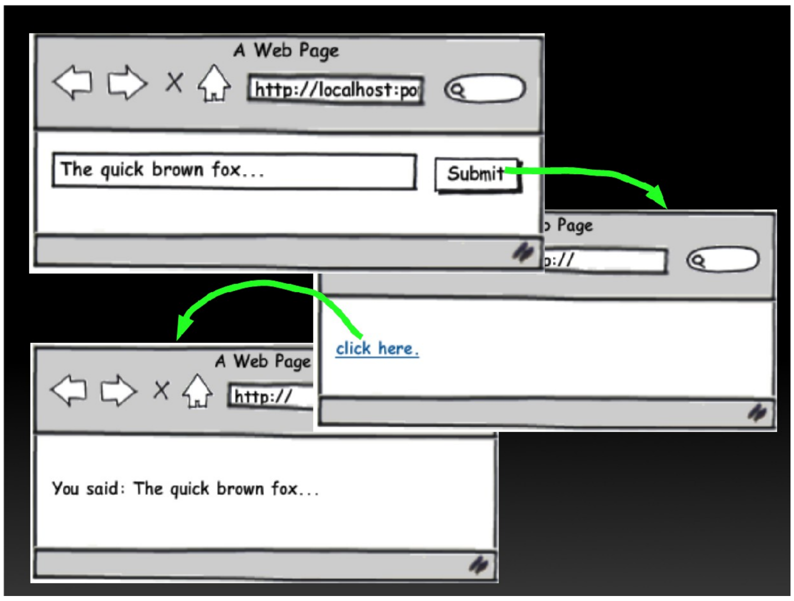
```
function accgen(n) {
    return function(i) {
        return n += i;
    }
}
```

A function that creates another, anonymous function. How would the solution look in Java? Well, you cannot solve this problem in Java - That is, you cannot write a (reasonably short) solution that fulfills all the requirements. You might think "Well, nice, but what does that have to do with extensibility?". With anonymous functions and closures it is very easy to write reusable algorithms, like map reduce. Of course you can simulate a lot of this behaviour in Java too, but there is a lot more boiler plate code you have to write. So, if your programming language supports closures and anonymous functions, use them to make your libraries and frameworks extensible.

Succinctness

Fewer lines of code are better. Given that it still solves the problem. Less code means less debugging, less testing and fewer bugs. A framework has to make sure that the programmers using it can express concepts succinctly and still clearly. Also, the framework or library should not encourage the developer to hide essential parts of the code just for the sake of succinctness.
I guess an example would be handy right now. This is from Paul Graham's article Take the Arc Challenge:
Write a program that causes the url said (e.g. http://localhost:port/said) to produce a page with an input field and a submit button. When the submit button is pressed, that should produce a second page with a single link saying "click here." When that is clicked it should lead to a third page that says "you said: ..." where ... is whatever the user typed in the original input field. The third page must only show what the user actually typed. I.e. the value entered in the input field must not be passed in the url, or it would be possible to change the behavior of the final page by editing the url.

Imagine solving this challenge using your favourite web framework... In Apache Wicket, for example, one would have to write 2-3 Java Classes, 3 XHTML-Files and an XML file to solve the challenge. In JSXP it's just about the same effort (Without the XML file and most of the Java code will be generated).



This is the solution written in ARC, the Lisp-dialect created by Paul Graham:

```
(defop said req
  (aform [w/link (pr "you said: " (arg _ "foo"))
           (pr "click here")]
    (input "foo")
    (submit)))
```

The beautiful thing about this solution is that it is really short, but still does not hide anything essential from the reader: Everything from the original requirements can be found directly in the code.
There are several reasons why less code is better. An important one is Working Memory Capacity (which we have explained in the section about readability). Programmers have to remember large parts of the code correctly so they can modify it. They have to "Load" it into working memory - This is what gets programmers into "flow". Succinct code means that the programmer can "load" more code more quickly.
There is also some evidence that the number of lines of code a programmer writes per hour is independent of the programming language - also the number of bugs per line of code seems to be pretty independent of the technology used (see for example this paper by Ericsson: http://www.erlang.se/publications/Ulf_Wiger.pdf). So, less code means less bugs and more productivity.
Finding errors is easier too: There is less code where the bug can be hidden, and there are probably less tests affected.
Can a piece of code be too succinct? It can, if it hides things that are essential to understand what the code does. Consider this piece of Ruby code:

```
while file.gets
    print if ~/third/ .. ~/fifth/
end
```

This code reads all lines of a file, and as soon as one line matches the string "third" it starts printing all the following lines - until a line matches the string "fifth" (this is what the ".." operator does - it acts like a toggle switch).
But how does it do that? There are no variables involved whatsoever!

The code works because several ruby functions use global variables by default if no other parameters are given: "file.gets" saves its result in the global variable $_, the "~" operator can match against $_, and if "print" is called with no argument it prints $_.
Something very essential is hidden here: How the data flows through the program. A better - and still very succinct - solution would be:

```
while line=file.gets
            puts line if line=~/third/ .. line=~/fifth/
end
```

Less code is better - unless it hides something essential about the solution. Make sure that programmers who use your framework or library can create succinct solutions. Don't force them to write boilerplate code or configuration files where defaults would do.
Think about one of your favourite frameworks or libraries. Does it force you to create boilerplate code or does it enable you to create succinct, elegant solutions?

Kontaktadresse:
Oliver Szymanski
Lindenstr. 31
D-49751 Sögel

E-Mail	oliver.szymanski@source-knights.com
Internet:	http://source-knights.com

David Tanzer
Strachgasse 12
A-4020 Linz

Telefon:	+43 (0) 676-53 47 723
E-Mail	david@davidtanzer.net
Internet:	http://davidtanzer.net