ANDRE GARZIA

LIVECODE ADVANCED APPLICATION ARCHITECTURE

From HOBBYIST to PRO

LiveCode Advanced Application Architecture

From HOBBYIST to PRO

Andre Garzia

This book is for sale at http://leanpub.com/livecodeapparchitecture

This version was published on 2018-11-19

Tweet This Book!

Please help Andre Garzia by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I am learning how to organize my code and create easy to maintain apps with LiveCode

The suggested hashtag for this book is #livecode.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#livecode

Also By Andre Garzia

To Elisangela Mendonça de Andrade Garzia

Contents

Disclaimer

This is an early release. Send feedback to feedback@andregarzia.com[1].

The content of this book will be expanded in the next months, I see this book as a living thing, much like LiveCode IDE.

This is the "minimum amount of book" I consider useful enough to make a release. I hope you all enjoy and benefit from it.

Intended audience

LiveCode[2] developers who want to learn a more structured way of organizing and reusing their code. This book assumes beginner or intermediate knowledge of LiveCode. If you joined our beloved community recently there are other resources that will help you get started[3].

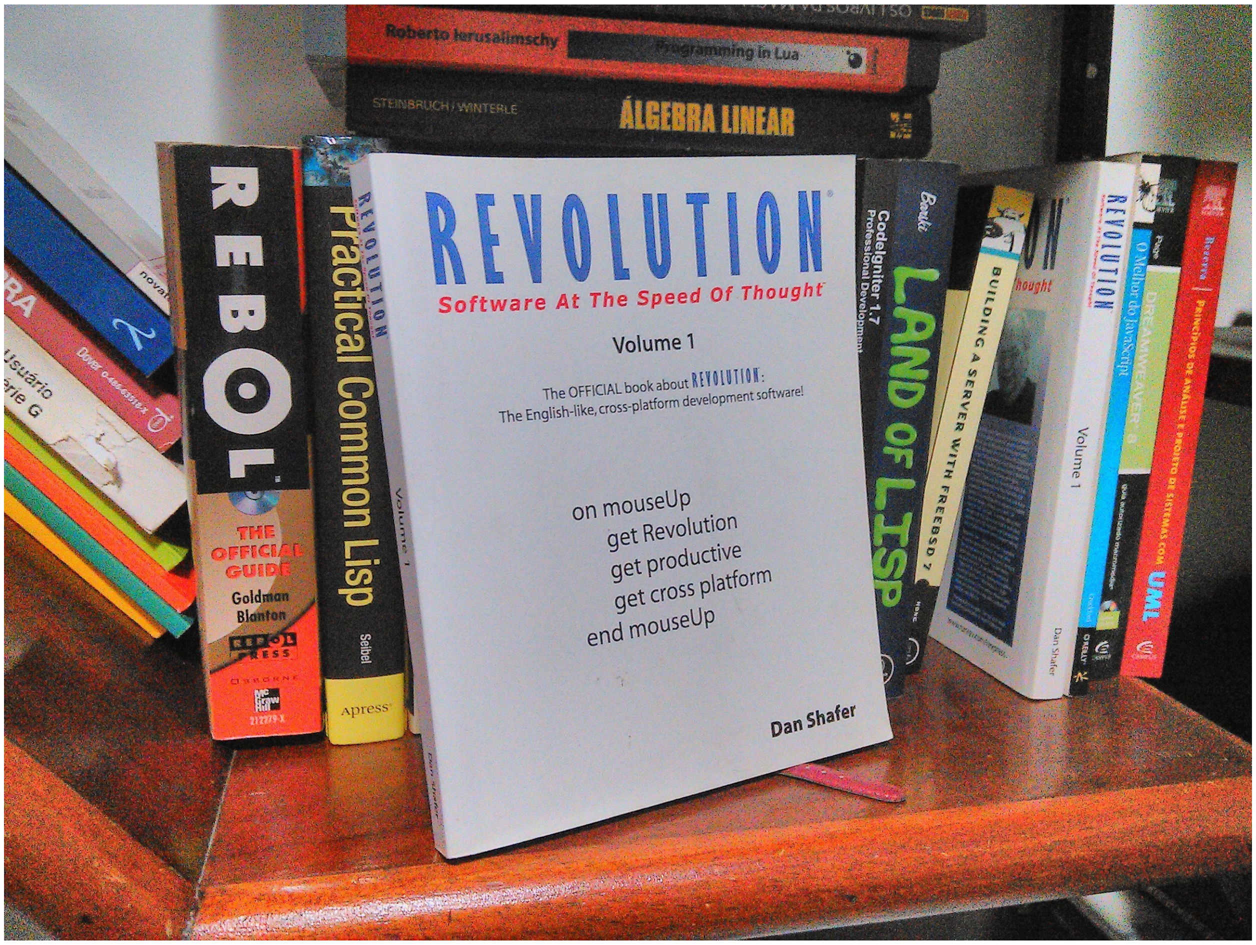
Thanks

Thanks to every single member of the LiveCode community. I remember being a newbie in 2004 and joining this awesome crew full of questions and doubts. You all received me and helped shape my career and life. If I were to thank people personally I would need Heather to give me a dump of all use-list subscribers from 2004 to 2018 (oh my, it has been more than ten years…).

You people are just too awesome!

Since this is a book, I want to specify a very special friend here in this section: Dan "the man" Shafer. From my early days till today, Dan was always someone I could look for inspiration. I remember meeting him in 2004 and thinking that one day I would like to write books as well. This is my third programming book, I might be getting somewhere…

[1]mailto:feedback@andregarzia.com
[2]http://livecode.com
[3]https://livecode.com/resources/guides/



Revolution at the speed of thought

I still consider myself an inventive user more than a developer anyway and the book above still on my shelf...

on openBook

I've been a speaker on LiveCode conferences multiple times speaking about many different topics such as web development, game design and application organization. The talk that received the best feedback was the one about application organization during RunRevLive 2013. I remember the smiles from the audience as I moved away from the speaker microphone. That mental image and the the kind words received as I left the stage kept rumbling on my mind for a while as I tried to understand why that session was so well received while other sessions were less impressing, then one day late at night, I grokked it!

Even though *Web development* and *Game design* are very interesting topics, they are not the most common activities in our community. When I gave sessions on web development, those doing that kind of work really enjoyed the session but those that don't do anything related to the Web were not all that impressed. The same thing happened with game design. Still, application organization is a topic that affects everyone. All LiveCoders, from desktop application developers, to WebMasters, to Game Designers, start with the same felling: *"LiveCode is so easy that I don't really need to organize my stuff!"* and then, like instantaneous karma, the regret from not having a defined methodology comes right back at us while we wonder: *"Where did I placed that handler?"*.

We enjoy throwing stacks around while we're learning the language, I have stacks so badly designed that they try to hurt you when you look at the code. Those stacks from *the early spaghetti days are a huge contrast to what I try to build now*. My objective with this book is to codify what I've learned as I figured out that my code needed to be better organized and easier to maintain if I were to win in the freelance market. Let us label this improvement of skills **going from hobbyist to pro**.

From hobbyist to pro

If you pick the average sample code contribution by new developers that are learning LiveCode as their first computer programming language and compare it with what the seasoned developers who have business built around the language and have been shipping applications for a number of years, you will notice a huge difference in both code clarity and organization.

Going from hobbyist to pro is not the act of building a business or shipping commercial software but the act of working using best practices. This book is about exactly this, picking knowledge from seasoned developers and moving so that we stop shipping spaghetti code.

I don't claim that the techniques presented here were invented by me and lots of seasoned developers will recognize their style in one part or another. This book has many techniques in it and while I try to present them as a solid and cohesive architecture, remember that you can tweak things to suit your application. You should tailor the methodology shown here to your own needs and not the other way around. With that in mind, lets review the books objective.

If I do my job right, I will

Present a solid architecture for building application using LiveCode using an MVC-like approach. By the end of the book (if you read everything) you should be able to:

- Name your variable and handlers with meaningful names.
- Correctly place code in stacks.
- Divide your application into logical units such as models, views, controllers and libraries.
- Have a clear picture where things should be at any time.
- Create code that is easy to change as your application scope evolves.
- Reuse more and more code as you build more applications.

This book roadmap

To achieve our objective we'll follow *a hero's journey* and begin with nothing and work towards **a simple address book application** by the end of the book. We'll begin talking about code conventions and then move to explain the programming pattern called MVC. Each chapter begins with some theory explaining the topic at hand and then moves to a practical section where we apply what we learned to the task of building the address book application.

Even though I am organizing this book as a little journey, you will still be able to cherry pick chapters and learn specific topics out of order since your needs or interests may not be in the same order as presented here. Just be aware that the book sample is built with the book order in mind and code from the middle of the book may depend on code from the early parts.

Code Conventions

The main objective of code conventions is to produce code that is identical no matter who the author is. Code that follow such conventions are immediately recognizable by developers and easier to maintain in the long run. We could say that code conventions are nothing but *codified common sense* which is true in hindsight but may not be clear when all you have is an empty stack and no plan. In this chapter we're going to learn the most common code conventions in our community, and we'll begin with how to name stuff.

How to name things

Names should always be descriptive and meaningful. LiveCode is a very verbose language and you should not be afraid of making your variables and handlers even more verbose if this saves your sanity six months from now. Other programming languages from old times had space and memory constraints that forced the developer to use the smallest names.

> I had a computer that used **BASIC** and each variable name could have only up to two letters, so you went from variable **a** to variable **zz** (and cried a lot while maintaining any code).

Many developers that learned to code with similar constraints still name things with cryptic names such as `curcont` claiming that saving some characters in the variable name is good because they type variable names a lot and typing less will lead them to work faster. Naming the same variable with a name like `tCurrentContact` makes your code obvious even to people who have never seen it before and is doesn't require much more typing while presenting many benefits in the long run. We should always name things clearly and avoid using contractions or abbreviations. Don't be afraid to type more characters, the more descriptive something is, the less code commenting you will need to do later. Lets agree on some guidelines for naming things:

- Use meaningful names.
- Use full words and separate them with a capital letter such as `currentContact`.
- Avoid generic names such as `count` and choose more specific ones such as `numberOfApplesIn-Basket`.

The rules above are great but lets dive deeper and focus on how to name variables.

Naming variables

In the LiveCode community it is very common to use Hungarian Notation[4] to name variables as explained in the Fourth World Scripting Style Guide[5]. This method of naming variables advocates for the usage of prefixes and suffixes in the variable name to make the variable type and scope clear.

In LiveCode we have global, script local and temporary variables. I almost forgot, we also have constants. Let us mark each scope with a prefix in the variable name:

- **g for global variable**: such as `gApplicationVersion`.
- **s for script local variable**: as in `sDatabaseConnectionID`.
- **t for temporary variable**: like: `tCurrentContact`.
- **p for parameters**: if the variable is passed as a parameter to a handler. Example: `pContactA`.
- **k for constant**: sometimes we also use ALL CAPS and underscores as separators for constants: `kUPDATE_URL`.

Another important caveat is that there is no way to tell if a variable is an array or not in LiveCode just by looking at the variable name, so **we usually use a capital `A` suffix for array variable names** as in this example: `gAllContactsA` which is a global array variable holding all contacts for a given application.

Tip: Its ok to name a variable **x** or **y** on loops such as `repeat with x = 1 to 10`.

Example:

```
global gAllContactsA

local sCurrentContactA

command sayHi pContactA
    local tGreeting
    put "Hello," && pContactA["name"] into tGreeting
    return tGreeting
end sayHi
```

[4]Check out http://en.wikipedia.org/wiki/Hungarian_notation.
[5]http://www.fourthworld.com/embassy/articles/scriptstyle.html

Naming handlers

Some people take a great care when naming their variables and then let all this care go down the drain when naming their handlers. If you think that most of your code ends up as being variable names and handler names, you notice that using some wisdom when naming them will pay well in the long run. Some guidelines for handler names:

- Always have a verb in the name as in `placeFruitInBasket` instead of `fruitBasket`.
- Be descriptive, use names as long as they need to be.

I have a handler called `checkApplicationVersionAndUpdateIfNeeded` and I think it is a great name because it instantly tells me what that handler does.

Example:

```
function getContactDataAsJSON
    // do something
end getContactDataAsJSON

command emailCurrentSelectedContact
    // do something
end emailCurrentSelectedContact
```

We should not underestimate how useful a good name is. Select your verb well and if possible use it as the first word of the handler. As a mental exercise, consider the code below:

```
put contacts() into tContactsA
```

From that code there is not much we can infer about what `contacts()` does. Does it accepts a parameter? Is it a function to set something or to retrieve something or both? Unless we have comments in the source code we'll need to read the implementation of that function to understand what it is actually doing. This could all be solved by better naming.

```
put getAllContacts() into tContactsA
```

Not only we know what it does since it uses the verb **get** but we know it retrieves **all** contacts and not just some selection. Another important naming convention for handlers is regarding their scope. In LiveCode we can have both public and private handlers. If you're building code that is going to be shared, such as a library, then you'd create a public API that is documented and available for others, and a private API that is used only by you. Handlers should not call private handlers of other stacks. And yes, now that we have `private` and `public` keywords there is no way to leak the implementation handlers anymore but still naming them differently helps with maintenance.

Lets use an underscore character before the name of any private handler. So if you have a public command called `saveContact` and this command makes use of the private command `writeContactToFile` then you should named the private handler `_writeContactToFile` like in:

```
on saveContact pContactA
    combine pContactA by cr and tab
    _writeContactToFile pContactA
end saveContact
```

So just by looking at that source code we know that `_writeContactToFile` is a private command. Sometimes, naming something wisely is not enough to convey why something is happening in the source code. For these cases we have source code documentation.

How to document your code

There should be a book dedicated only to teaching how to document code. I suspect developers hate to write documentation because when they think that when they are writing documentation they are not coding. There are some great tools to help you document your application such as screensteps[6] but this section is not about documenting your application for your end user (one of the tasks that screensteps excels on) but to create comments alongside the source code to better explain what is going on.

The rule of thumb for code comments is **not to explain what is happening but why it is happening**. The developer can understand what is going on from reading the source code but they may need some help understanding why it should work that way. For example, consider **the poor case of commenting** below:

```
function addTwo pNumber
    // below we add two to a number
    add 2 to pNumber
    return pNumber
end addTwo
```

That comment is **useless** because it doesn't help the developer at all. Now consider the following better example of commenting:

```
function movePlayerDown @pPlayerLongID
    // We add 48 pixels to the value of
    // top because our game tiles are 48 pixels
    // tall, so adding this value moves
    // the player one tile down.
    add 48 to the top of pPlayerLongID
end movePlayerDown
```

[6]http://www.bluemangolearning.com/screensteps/

If at anytime in your source code something is not clear why it should be that way, then it is a good time to write a comment explaining the reasons behind how that code needs to be the way it is. There is no such thing as excessive commenting if all your comments are good quality (mark this affirmation as my personal opinion as this is a controversial topic). The more code conventions and comments you use, the less you will need to remember when you need to fix something in your app in the future.

Tools that use comments for the greater good

There are some great tools out there that use source code comments to build amazing things. There is lcTaskList[7] that allows you to leave comments with markings such as TODO, FIXME and build reports on the notes you left for yourself. There is also NativeDoc[8] which is a documentation generator for LiveCode that picks comments structured in a specific way and builds HTML documentation from it.

I've also built RevDoc that is like NativeDoc a long time ago, you can check more in RevUp 64[9].

Summary

In this chapter we've learned:

- The importance of good names and conventions.
- That by using prefixes and suffixes in variable names we can convey scope and nature.
- That by being verbose and always using a verb in handlers names we can make our intention (and code) clearer.
- That source code commenting helps developers understand why something is happening and not just how.

The next chapter we'll talk about MVC, a mindset and strategy that is very successful in shipping maintainable code.

[7]https://livecode.com/extensions/lctasklist/1-3-0/
[8]http://www.nativesoft.net/products/nativedoc/
[9]http://newsletters.livecode.com/january/issue64/newsletter1.php

An introduction to MVC

The objective of the MVC paradigm[10] is to enforce loose coupling[11] between the representation of information and how the user interacts with it, so that you can more easily change one without breaking the other.

By *loose coupling* I mean that your application is divided into units (that in the MVC paradigm we call models, views and controllers) that are not dependent on each other in a hardcoded way. This means separating the way things are stored and retrieved (aka model) from your business logic (aka controller) from the way you display your application (aka view). If you follow this separation, you will be able to change each unit with minimal change to the surrounding units.

To be able to achieve this separation, we should follow the law of Demeter[12] which states that each of our application units should only have limited knowledge about the other units and restrict its interactions to its closest friends. For example in a complex application with an address book module and a project management module, each module should restrict its interactions to themselves. The project management module should not go changing the data from the address book module without talking to it. The law of Demeter helps us avoid the difficult debugging problem of figuring out where something was changed

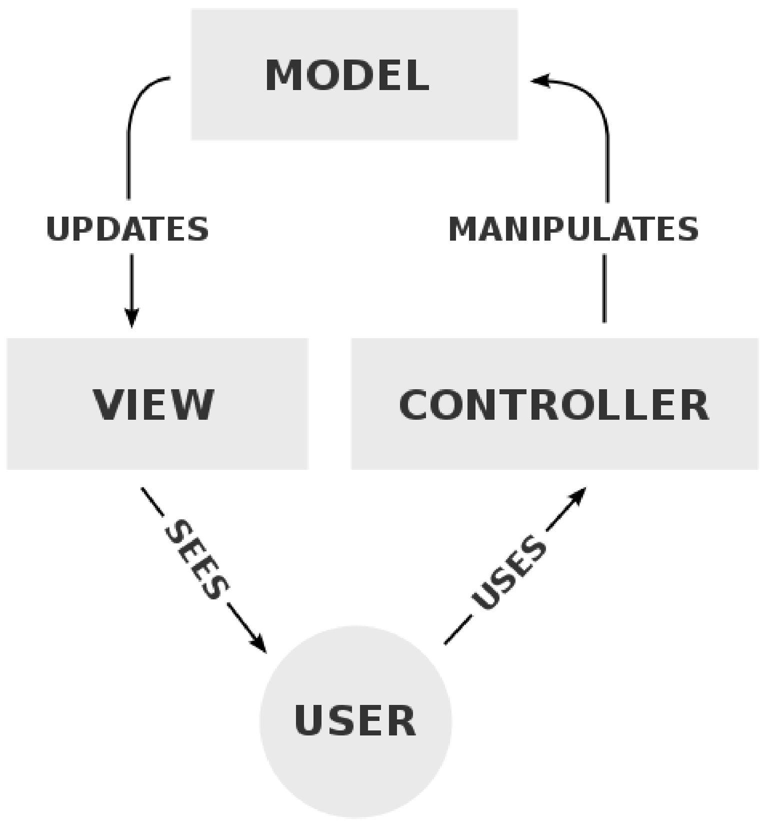
Ever caught yourself thinking: *"which part of the code changed that global variable?!"* thats a huge problem in LiveCode (and other languages). People didn't coin the term Globals are Evil[13] for no reason. Be aware of spooky actions at a distance, it scared Einstein and it should scare you too. Try replacing globals with variables with limited scope.

We can think of MVC as a way to untangle our stacks moving from the usual spaghetti-approaching state towards a neat and organized future. Since we're speaking about stacks, lets draw out some plans to explain how MVC applies to a project. As we progress in this book we'll build a simple address book example that will be divided into units called *model*, *view*, *controller* and *libraries*. Each of these units is a different stack and will deal with a different problem.

> **MVC & OOP**: The MVC paradigm was created alongside the Object-Oriented Programming[14] which is one of the main paradigms to approach programing. LiveCode is not an OOP language in the classical sense as understood by SmallTalk developers but that does not mean we can't leverage what we can from the MVC way of doing things when creating our own applications.

[10]http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
[11]http://en.wikipedia.org/wiki/Loose_coupling
[12]http://en.wikipedia.org/wiki/Law_Of_Demeter
[13]http://wiki.c2.com/?GlobalVariablesAreBad
[14]https://en.wikipedia.org/wiki/Object-oriented_programming

MVC in 20 seconds



The MVC process

The model

Our address book model will be responsible for loading, saving and manipulating address book contacts data. We're going to define an API that will be used by the other units to talk to it and how we implement this contact manipulation is of no concern to them. We could even have more than one model for the same data type but with different implementation, such as a model that saves to a database, a model that saves to text files, a model that saves to a remote server. As long as we implement the same public API, we can change the internals of the model without touching the other units. In the classic MVC implementation, the Model will notify interested parties if the data changes.

The controller

The controller is a stack that holds your business logic. While the model knows how to manipulate contact data, it doesn't know why or when it should manipulate it. The controller is responsible for talking to the model, like a transit cop directing traffic to and from the model to the other units.

In the classic MVC implementation, the controller is responsible for getting input from keyboard and mouse or whatever input gizmo you are using and decide what to do with it. It talks directly to the model telling it what it should do the data. It can also talk directly to the view since some of the input may require stuff in the view to change.

The view

The view is the presentation layer with your cards and controls. The user sees and interacts with the view and the view talks only to the controller which in turn talks to the model.

This way you can try different views and layouts without touching your business logic because that is located on the controller. When you're building cross-platform applications, you can build multiple views to create interfaces that are suitable for the various form factors you're deploying to, like desktop, tablets and smartphones, all from a single application source.

In traditional MVC implementations, the model use broadcasting techniques to inform the views when data changes. In this loose coupling, the model has no idea what the view needs to change, it is just broadcast to all interested parties that they should act upon data change.

Libraries

It is good practice to build code that is reusable beyond your current application development. Libraries due to not being tied to the business logic of your application, are the most reusable pieces of our softwares, the more generic looking code that you can build into libraries the easier your development will be in the future.

As an example, suppose you have a function that picks a name like *andre alves garzia* and converts it to TitleCase like *Andre Alves Garzia*. This function is useful for the address book application and many other applications, so it could be added to a library and reused in other projects.

Summary

This was just an at a glance introduction to MVC. Each chapter now will dive deeper into one of these concepts starting with models. By each chapters end, we'll have one unit ready and be closer to the finished address book application. This small chapter had a bunch of links in it, you really should check them all.

Now, lets see what models are made of.

on closeBook

This is the end of the book, I know I want more too. In the next months I plan to do monthly releases of this book, not only because I am sure I will need to issue erratas, but also because there is much more to cover. This was the minimal set of features and discussion I felt was needed to move a beginner LiveCoder into a more seasoned developer experience.

By reaching this point of the book and examining the bundled source code, you should be comfortable with the following things:

- **Naming anything** under the sun! You should be an expert namer now, and both your variables, stacks, and handlers should all use descriptive and meaningful names, all cohesive with the development experience you're building.
- **How to document your code** to save precious time in the future as you forget how stuff was made in the first place and, or, need to on-board more developers into your project. The reasons behind your coding choices should now be apparent not only in the code itself but in the accompanying code comments.
- **Splitting your code into the MVC pattern** making all the units loose coupled to each other by leveraging the message path and useful patterns such as the *publisher/subscriber* pattern.

This has been a quick journey but I hope it was worth it for you, I know I am sure it has been worth for me. Please, don't hesitate to reach out to me with comments, feedback and request for new topics or clarification. My email for this book is feedback@andregarzia.com[15].

Don't forget to check everything I am building for LiveCode, the entry point for all my creations is:

https://andregarzia.com/livecode[16].

See you all around! Andre

[15]mailto:feedback@andregarzia.com
[16]https://andregarzia.com/livecode