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 2020-08-10

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

Guia Rápido de Desenvolvimento para Firefox OS

Quick Guide For Firefox OS App Development

Building Games for Firefox OS

Quick Guide For Firefox OS App Development

To Elisangela Mendonça de Andrade Garzia

Contents

Disclaimer

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

I will expand the content of this book 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.

Version

The version you're reading is the: `2020.08.1`

I'm using CalVer scheme to create version numbers for this book. They are in the format of `<year>.<month>.<release>` so the first release in November 2019 is `2019.11.1`, if I end up doing a second release in the same month, it will be `2019.11.2`.

Intended audience

[LiveCode](http://livecode.com)[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](https://livecode.com/resources/guides/)[3].

Thanks

Thanks to every single member of the LiveCode community. I remember being a newbie in 2004 and joining this outstanding 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 over 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. I have since written many programming books, but I always go back to him for inspiration.

---

[1]mailto:feedback@andregarzia.com
[2]http://livecode.com
[3]https://livecode.com/resources/guides/
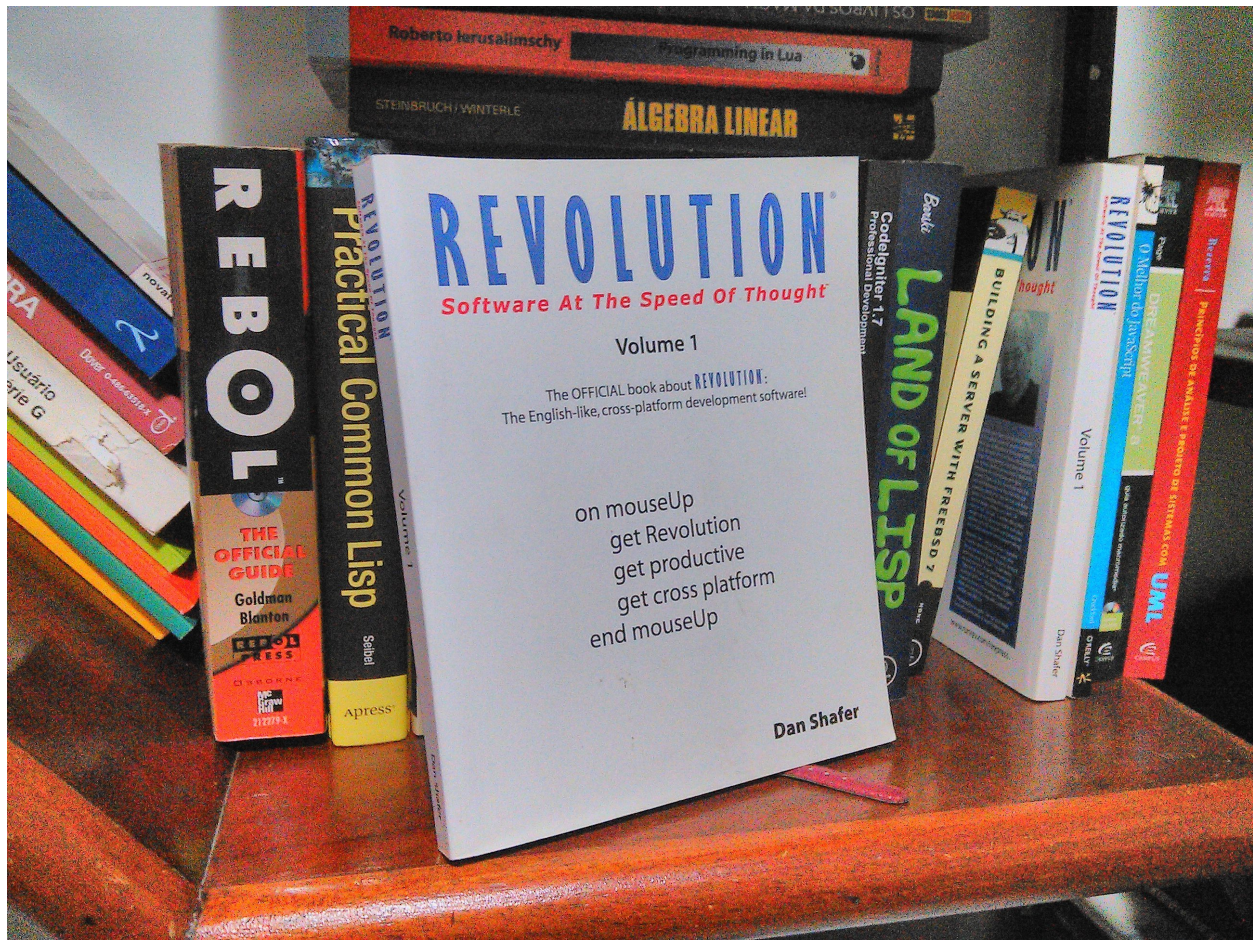
Revolution at the speed of thought

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 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 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 memorable, 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 whose work is not 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 for not having a defined organisational method 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 aim with this book is to codify what I've learned as I figured out that my code needed to better organization. I needed it to become easier to maintain, a crucial skill 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 several 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 we stop shipping spaghetti code.

I don't claim that I invented the techniques presented here, and lots of seasoned developers will recognize their style throughout the chapters. 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 methods shown here to your own needs and not the other way around. With that in mind, lets review the book's objectives.

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.
- 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 aim, 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 building the address book application.

Even though I am organizing this book as a brief 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 I build the sample code with the book order in mind and code from the middle of the book may depend on code from earlier chapters.

Code Conventions

The primary 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 will 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 `numberOfApplesInBasket`. 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. Constants are like variables, but they can't be changed once assigned. 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 an application.

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

Example:

```
1   global gAllContactsA
2   local sCurrentContactA
3
4   command sayHi pContactA
5         local tGreeting
6         put "Hello," && pContactA["name"] into tGreeting
7         return tGreeting
8   end sayHi
```

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 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:

---

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

- Always place a verb in the handler's 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:

```
1   function getContactDataAsJSON
2           // do something
3   end getContactDataAsJSON
4
5   command emailCurrentSelectedContact
6           // do something
7   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:

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

From that code there is not much we can infer about what `contacts()` does. Does it accept a parameter? Is it a function to set something or to retrieve something or both? Unless we have comments in the source code, we must read the implementation of that function to understand what it is doing. Better naming could solve this.

```
1   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 will 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 name the private handler `_writeContactToFile` like in:

```
1  on saveContact pContactA
2          combine pContactA by cr and tab
3          _writeContactToFile pContactA
4  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 it causes them to think about the same problem twice, once when coding and again when documenting. 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 task 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 to understand why it should work that way. For example, consider **the poor case of commenting** below:

```
1  function addTwo pNumber
2          // below we add two to a number
3          add 2 to pNumber
4          return pNumber
5  end addTwo
```

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

```
1  function movePlayerDown @pPlayerLongID
2          // We add 48 pixels to the value of
3          // top because our game tiles are 48 pixels
4          // tall, so adding this value moves
5          // the player one tile down.
6          add 48 to the top of pPlayerLongID
7  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 excellent quality (mark this affirmation as my opinion as this is a controversial topic). The more code conventions and comments you use, the less you will need to remember when it is time for you to fix something in your app.

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 a little plugin similar to NativeDoc. 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 the developer 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 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?!"* that's a tremendous problem in LiveCode (and other languages). People didn't coin the motto 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 distinct 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.

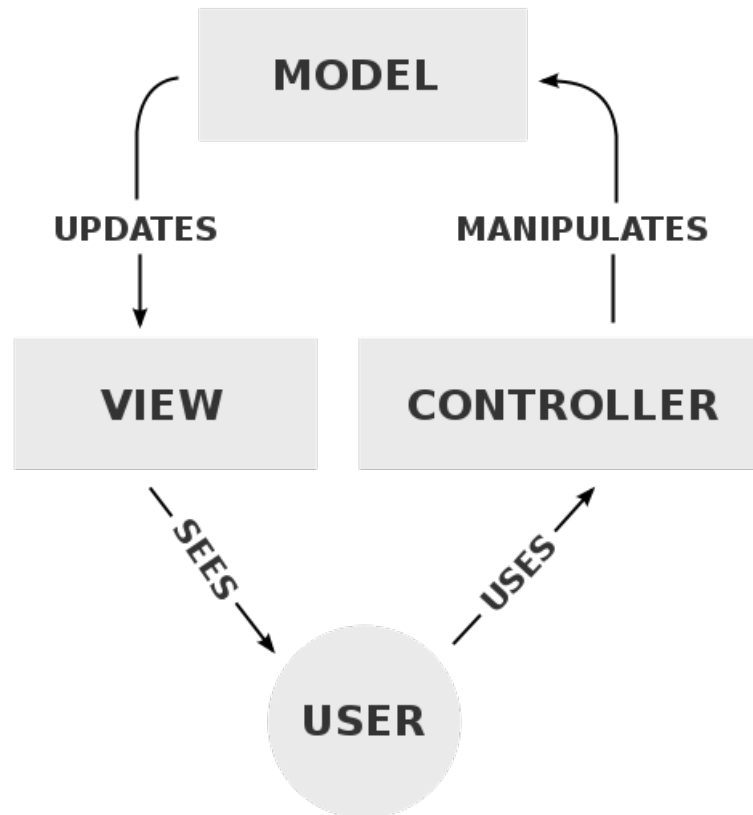[10]http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
[11]http://en.wikipedia.org/wiki/Loose_coupling
[12]http://en.wikipedia.org/wiki/Law_Of_Demeter
[13]http://wiki.c2.com/?GlobalVariablesAreBad
[14]https://en.wikipedia.org/wiki/Object-oriented_programming

MVC in 20 seconds

The MVC process

The model

Our address book model will load, save and manipulate address book contacts data. We will define an API that the other units will use. 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, and 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 talks 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 gets 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 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 talks to the model.

This way you can try different views and layouts without touching your business logic because that code is on the controller. When you're building cross-platform applications, you can build multiple views to create interfaces 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 uses broadcasting techniques to inform the views when data changes. In this loose coupling, the model has no idea what inside the view needs changing; 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 are not tied to the business logic of your application, making them the most reusable pieces of our softwares. The more generic looking code that you can build into libraries, the easier your development will be.

As an example, suppose you have a function that picks a name like *andré alves garzia* and converts it to TitleCase like *André Alves Garzia*. This function is useful for the address book application and many other applications, so it could be part of 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 chapter's 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 should check them all. Now, lets see what how we make models.

Models

Models are the collection of routines that deal with the manipulation of some domain specific data used in your application. Not all applications need models, you can build a calculator without adding any model. Models are most useful when you're dealing with complex data or when you need to keep track of stuff between sessions of your applications. If you need to save and restore data in your application, then you will enjoy using models.

What goes into the model and what doesn't

Models should contain all code needed to manipulate some domain specific data used by your business (I am repeating myself intentionally, to fix this message). With an address book application, our model will handle contact information data and have all the routines needed to create, delete, edit, update and search for contacts. Our model will deal with all that, but it won't know how to display a contact on the screen or what combination of mouse and keyboard is used to edit them. Models are accessed by their public API, which is the collection of handlers that a controller uses to manipulate them.

Building a sane API

It is easy to conclude that **models are as good as their API**. You can have the most elegant file representation for an address book contact, but if the routines that the controller will use are poorly designed, then you'll end up adding more friction to your code maintenance chores.

A **good API design does not let implementation details leak**. For example, suppose we want to save a contact in the address book, the controller could use code similar to:

```
1   on mouseUp
2       put getContactFromCardControls() into tContactA
3       openAddressBookContactFile
4       addContact tContactA
5       saveAddressBookContactFile
6   end mouseUp
```

Even though we can understand what is going on, **this is not a good design because it leaks all the implementation details**. The controller need not know about files, nor does it need to remember to open and save them. This is a responsibility of the model. A better API would be:

```
1   on mouseUp
2       put getContactFromCardControls() into tContactA
3       saveContact tContactA
4       if the result is not empty then
5           // handle error
6       end if
7   end mouseUp
```

We leave the entire business with files to the model implementation. The controller should not care if the contact is saved to disk, to a database, to a remote server, to a shared drive or printed. All it needs to do is request a contact to be saved and check the result of such a call.

With that in mind, it is time we get practical and build our model.

Practical: Building an address book model

This book is called *LiveCode Advanced Application Architecture* and not *The Art and Science of Contact Management* so our model will be naive. I am more interested in showing you a workflow and strategy for LiveCode software development than creating a bullet proof address book implementation. We could turn this book into a 900+ pages behemoth if we started going down that rabbit hole, so we'll keep things simple.

Many developers will think about models by thinking on how data will be stored and then creating the API to match that train of thought. I don't like that approach. I prefer creating the API the controller will call first and then ponder about how to implement it. This way we can have an elegant API first and change implementation as needed.

The public API

First let's figure out what our needs are. We need to manipulate contacts and keep track of them. So our model needs to:

- Create a new contact.
- Return all contacts.
- Return a specific contact.
- Search for a contact.
- Save a contact.
- Delete a contact.
- Update a contact field.

Lets look at our public API:

Type	Name	Params	Purpose
function	createNewContact	pFieldsA	Creates a new contact based on the fields passed
function	getAllContacts		Returns an array of all available contacts
function	getContactByID	pID	Return a specific contact
function	filterContactsByField	pField, pValue	Return an array of contacts by searching a field
command	saveContact	pID,pContactA	Saves a contact
command	updateContactField	pID, pField, pValue	Updates a field in a contact
command	deleteContact	pID	Deletes a contact

This public API is all the controller will use to implement our Address Book application. The inner working of the model will be of no concern to both the controller or the view. Since our model will not depend on anything besides itself, we will reuse it in other future projects. The collection of handlers above is the only public API for our model, all the other handlers are private. Lets explain the rationale behind each handler before we move onto the implementation.

Contact Model: Creating the stack

Create a new script-only mainstack in LiveCode and call it `model.contact` with its filename called `model.contact.livecodescript`. This is not a convention followed by others as far as I know. In this book we'll use the following files:

Role	Stack name	File name
Model	model.contact	model.contact.livecodescript
Controller	controller.addressbook	controller.addressbook.livecode
View	view.addressbook	view.addressbook.livecode

All our implementation will go into *the stack script*. Our address book application will use the model as a Library Stack[15] so that its public API is available on the message path. By keeping the model on its own file, we'll be able to reuse it easily.

Contact Model: the data storage

To keep the implementation simple, our model will work with files. We'll use a single file called **mycontacts.data** to hold our data. We could make a more robust address book by using a SQL database or even an online storage, but we will keep things simple. The implementation we'll be as simple as possible.

This `mycontacts.data` file will just be the dump of an array into the disk. We'll write and read from it as needed.

---

[15]http://livecode.com/developers/api/6.0.2/command/start%20using/

> **Warning**: If you plan to ship software using the iTunes App Store or the App Store for macOS then you need to know that Apple is very strict on what you can write to disk and where. Personally, I think the App Store requirements are hurting developers, but that is a different matter altogether. Just remember to learn the requirements before trying to get your software approved by Apple.

Contact Model: the public API

We'll implement the public API as if the private handlers we need existed already then we'll implement them afterwards. I call this **the naïve dreamer approach** where you dream you have the handlers you need and then later you're forced back into reality and need to implement them all yourself (also known as: the principle of eventual implementation).

Creating a new contact

This handler is a function because it needs to return a value which is the ID of the created contact or false. We could use the `result` for this, but it makes it harder to compose with other statements such as `if` clauses. This function will receive an array where each key and value pair is a field for the contact entry. There will be no checking against a schema or field validation, whatever goes inside the array will become the user record.

This function will be called by the controller when it wants to add a new contact. There are potential sources of difficulties on this function, more specifically, it needs to make sure that the ID generated for a user will not collide with any other pre-existing user. This is easy in a single-threaded application such as a normal LiveCode based application, but if we think in terms of networked computers accessing the same contacts database, then new safeguards must be put into place.

To ensure the atomicity of the `createNewContact` operation, the function will make sure that it saves the data before it returns control to the calling handler.

```
1  function createNewContact pFieldsA
2     put _getAllContactsFromFile() into tAllContactsA
3     // below is a poor man unique id to be used as primary
4     // key for the new contact. This is not the correct way
5     // of doing this, but it is good enough for a book sample.
6
7     put the millisecs & random(999) into tUniqueID
8
9     // the code below checks to see if the key generated above
10    // is already in use and will keep generating keys until it
11    // generates something that is not currently in use.
```

```
12
13    repeat while the keys of tAllContactsA[tUniqueID] is not empty
14       put the millisecs & random(999) into tUniqueID
15    end repeat
16
17    put pFieldsA into tAllContactsA[tUniqueID]
18    _writeContactsToFile tAllContactsA
19    return tUniqueID
20 end createNewContact
```

This code loads the data from the file, generates a unique id, adds the new contact to the array of contacts and finally saves the file. Your handlers should always be as short as possible. Rule of thumb is that if your handler is taller than your screen, then you're doing something wrong and should refactor into multiple entries.

> **Warning**: That unique id on that code is all but unique. Thats not the correct way of generating ids. Its a naive implementation just to get us going. The wonderful Mark Smith made a Unique ID library available on his site*[a]*, if you need UUIDs, you need this library.
>
> ---
> [a]http://marksmith.on-rev.com/revstuff/

Our function returns the unique id for that contact so we can refer to it in other handlers. This implementation leaves two private handlers for us to implement later: _getAllContactsFromFile() and _writeContactsToFile.

Getting contacts

Lets implement both getAllContacts() and getContactByID because their code is very similar and straightforward.

```
1 function getAllContacts
2    put _getAllContactsFromFile() into tAllContactsA
3    return tAllContactsA
4 end getAllContacts
5
6 function getContactByID pID
7    put _getAllContactsFromFile() into tAllContactsA
8    return tAllContactsA[pID]
9 end getContactByID
```

Be aware that our routines are as atomic as possible. They tend not to rely on stuff such as files being open or a variable in memory having all the contacts. Every time we need to check the contacts,

we're loading them again from disk. This way we guarantee that the information on the file is always what we're dealing with. This lower our chances of having a state where the data in memory is out of sync with the data on the disk, but that causes more file activity and latency than needed. If we were dealing with thousands of records, we would need to implement things differently.

There are situations where you don't want to hit the disk that often. You need to know which solution is better for your case. There is no new private handler here, so lets move on.

Filtering contacts by field

Filtering and searching are among the family of tasks that are usually better handled by a SQL-based database. Our address book will be simple and will allow just filtering by a single field at once.

There are entire books, master degrees, PhD thesis, about filtering and searching. If you have complex searching needs, I suggest you learn more about patterns such as Map and Reduce[16].

```
1   function filterContactsByField pContactListA, pField, pValue
2       put _getAllContactsFromFile() into tAllContactsA
3
4       repeat for each key k in tAllContactsA
5           if pValue is in tAllContactsA[k][pField] then
6               put tAllContactsA[k] into tContactMatchesA[k]
7           end if
8       end repeat
9
10      return tContactMatchesA
11  end filterContactsByField
```

Again, another easy to follow implementation. We load all contacts, loop them checking if pValue is inside the field and return the matches.

Saving a contact

```
1   command saveContact pID, pContactA
2       put _getAllContactsFromFile() into tAllContactsA
3       put pContactA into tAllContactsA[pID]
4       _writeContactsToFile tAllContactsA
5   end saveContact
```

It's very similar to our createNewContact() function, but since we already know the unique id for that contact, we don't need to compute it.

All functions that return a collection of contacts have the unique IDs as the first level of the returned array. So you will only call the saveContact command if you already know which contact you want to save (by knowing its unique id).

---

[16]http://en.wikipedia.org/wiki/MapReduce

Updating a contact field

This is just a convenience handler. The `saveContact` command could be used for all update needs. I am doing this implementation just to show that your public API handlers can build upon other handlers from the public API itself.

```
1  command updateContactField pID, pField, pValue
2     put getContactByID(pID) into tContactA
3     put pValue into tContactA[pField]
4     saveContact pID, tContactA
5  end updateContactField
```

Contact Model: the private API

To finish our model, we need to implement two private calls: `_getAllContactsFromFile()` and `_writeContactsToFile`. These handlers are not supposed to be called by any routine outside the model stack. We could replace the whole implementation of that stack with a SQL-based solution, maintaining the same public API calls, and the controller and view wouldn't notice.

Loading contacts from file

```
1   private function _getAllContactsFromFile
2      // We need to preserve the current default folder
3      // The rest of the app may depend on it and we don't want
4      // side effects bugs plaging us.
5
6      put the defaultfolder into tCurrentDefaultFolder
7      set the itemdel to "/"
8      set the defaultfolder to item 1 to -2 of the effective filename of this stack
9
10     if there is a file "mycontacts.data" then
11        put url "binfile:mycontacts.data" into tFileContents
12        put arraydecode(tFileContents) into tAllContentsA
13     else
14        put empty into tAllContentsA
15     end if
16
17     set the defaultfolder to tCurrentDefaultFolder
18     return tAllContentsA
19  end _getAllContactsFromFile
```

There are two possibilities for our routine. Either there is a file holding the contacts in which case we need to load and return them or there isn't a file with contents and we can just return empty.

One important thing in this implementation is our attention to avoid side effects[17]. Side effects happen when a handler does something that is outside its domain specific needs.

For example, suppose you are building an application that uses our contact model and is reading and writing to the Desktop folder by setting it as the default folder. If our _getAllContactsFromFile() doesn't change the default folder back to what it was before it was called then when that function returns, the default folder would have changed to the same location as the model stack. A developer doesn't expect a function called _getAllContactsFromFile() to have the *unintentional and undocumented* effect of changing the default folder. That's why we keep track of what the default folder value and remember to set it back before returning. If we keep our handlers nice and tidy, we minimize the risk of unintentional side effects.

Writing contacts to file

```
1   private command _writeContactsToFile pAllContactsA
2      // We need to preserve the current default folder
3      // The rest of the app may depend on it and we don't want
4      // side effects bugs plaging us.
5      put the defaultfolder into tCurrentDefaultFolder
6      set the itemdel to "/"
7      set the defaultfolder to item 1 to -2 of the effective filename of this stack
8      put arrayencode(pAllContactsA) into url "binfile:mycontacts.data"
9      set the defaultfolder to tCurrentDefaultFolder
10  end _writeContactsToFile
```

The code to prevent side effects is longer than the actual code to write the contacts to a file. With that final command, we end our model implementation. That's all we need for a basic address book app. We may refine the code as new needs arise along the book.

Summary

In this chapter we had a lot of work. We learned about models and what we use them for. We also implemented a simple contact model to be used by our address book sample stack.

In the next chapter, we'll build our controller.

---

[17]http://en.wikipedia.org/wiki/Side_effect_%28computer_science%29

Controllers

Controllers are like transit cops

The primary responsibility of the controller is to organize the business logic and the flow of our software. Like traffic cops, the controller will stay in the very middle of your app, orchestrating the flow of data and events between the different units. This is the unit where you should pour most intelligence and attention because a good controller design will help experiment with variant models and views with ease. Just like traffic cops have their own attributions and you don't expect them to act like detectives or the swat, controllers have their own tight scope. It is time for us to learn what controllers are made of.

What goes into the controller and what doesn't

The controller is where you solve your problem. Each software is trying to solve a problem or accomplish something. In this task, it uses different models, networked resources and user interfaces, many of which are also used by other softwares in the same problem space. The *unique selling point* of your software, that spark that makes your solution special, that goes into the controller.

> **Disclaimer**: Yes, you can have software which the unique selling point is its fancy alien UI or software that has some clever and elegant file structures which make it more useful that the alternatives. This is all well and if you come to realize that your special thing is not how you solve a problem but what lies elsewhere in your structure, then so be it. Still, most software will have its special thing in *how it solves a problem*.

To use a culinary analogy, lets think about pizza. Most pizzas use the same ingredients, which are analogous to our models. Pizza ingredients don't know they belong into the pizza, we could use them for other recipes. Some pizzas are beautifully arranged and looking at them is almost as good as eating them. Those pizzas are like software with beautiful UIs. Still, the pizzas you remember the most and come back to, are the ones that are prepared in a special way. This preparation step, where a master pizzaiolo takes the common ingredients and makes them into something special, that is what a good controller is. Everyone loves problems that are solved in an elegant and tasty way. Specially if your problem is craving for a pizza as 1h27 AM like I am right now.

A good controller deals with some domain specific problem[18] you're trying to solve. If your app deals with multiple domain specific problem such as an app that deals not only with contacts but

---

[18]https://en.wikipedia.org/wiki/Problem_domain

also with project management, then you will end up with multiple controllers (and also more than one model).

Controllers and views are the least reusable part on an app because they are the business logic and the specific UI to apply that logic. Models and libraries are the most flexible parts, which are used over and over. If you end up reusing not only them but also controllers and views in a different application, then you're embedding one app into another. That is not a bad thing, but it's not the reusability that we're talking about here.

What shouldn't be on a controller

Controllers will exchange data with the views and the models by calling their public APIs, because of this they should not make any assumption about anything in your app besides such APIs. They can't reference controls or files or anything that should belong into one of the other units.

If you're wondering if some code belongs in the controller or model, then ask yourself these questions:

- Is this code something that I might use in another application?
- Is this code related to my business logic, or it is just some model manipulation?

If the answer to any of these questions is *YES* then you should probably implement that in the model layer. Similarly, if any of the code you're writing on the controller (or the model, or the view) is so generic that you could reuse it elsewhere, then place it in a library (more about that later in this book).

A controller assumes nothing about the views, it just calls their public implementation. The views themselves also call the public methods from the controller, the less coupling or dependency we have on hard-coded stuff the better. Let's get practical!

Practical: Building an address book controller

In the MVC chapter, we've talked about *loose coupling*, if you haven't read that or don't remember it, please go back to that chapter and check it out because that is crucial for understanding how and why we will implement our controller in a certain manner.

There are many ways to achieve *loose coupling* in LiveCode, basically you can over-engineer or under engineer as much as you want and still end up with something usable. In this section, I will introduce a common OOP pattern called publish and subscribe pattern[19], this will be used by both the controller and the view and it is a very elegant way to achieve independence between our units.

---

[19]https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

The publish and subscribe pattern

This is a messaging pattern in which the primary aim is to exchange data between *publishers* and *subscribers*. The beauty of it is that the *publishers* don't know who the *subscribers* are, they just broadcast messages, and those are picked automatically by the subscribers.

> [Wikipedia page for the publish/subscribe pattern](20): "In software architecture, publish–subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are."

By using this pattern our controller will at the same time subscribe to some messages and broadcast others, our view will do the same. They will not be aware of each other, but messages will flow from one to the other.

Using the publish/subscribe pattern in LiveCode

There are multiple implementations of this pattern in LiveCode. I am bundling a simple one I made alongside the source code of this book. We will use this stack as a library stack. It is called `aagPubSubLib.livecode`. I will not explain how we implement the pattern itself here, you can check the source code if you are curious about it. I'll explain how to use it though.

That library public API has four handlers:

subscribe command

This command is used to register your interest in a specific message. It accepts four parameters: the message, the callback for when it is received, the target for the callback and an option flag.

```
1  on mouseup
2      subscribe "search contacts", "_searchContacts"
3  end mouseup
```

In the example above, we're registering interest in the message *"search contacts"* and telling that when such message is broadcast, that the handler called *"_searchContacts"* should be called. By default, if the third parameter is empty, then the caller of `subscribe` is marked as the target for the callback. As for the first parameter, we can name it whatever we want as long as both the publisher and subscriber use the same name (or they won't find each other when the broadcast happen).

There is an analogous `unsubscribe` call that removes the callback.

---

[20]https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern

broadcast command

Use that to broadcast some message. It will run all scripts that registered interest in that message.

```
1   on mouseup
2      broadcast "search contacts", "andre garzia"
3   end mouseup
```

The example above would trigger the `_searchContacts` call since it subscribed to the *"search contacts"* message.

Implementing our controller

That is all we need to begin working on the controller, we'll define some messages that the controller broadcasts and some messages it is interested in (the view will broadcast them). We'll also add some housekeeping routines to start the view and make our life easier. In this book example, we'll have only one controller and if we were building a standalone, that would be our mainstack, so some initialization routines that would usually go into that stack are being placed into the controller. Since our controller is our main stack, we will make it a normal LiveCode stack or it won't be able to hold *standalone settings data* from the IDE. You can't build standalones from script-only stacks.

The controller is called `controller.addressbook.livecode`.

Messages it is interested in

Again, lets pretend that the messages exist and devise what we want our view to send us.

Message	Callback
search contacts	_searchContacts
update contact	_updateContact
delete contact	_deleteContact
save new contact	_saveNewContact
list contacts	_listContacts
get contact	_displayContact

On the left column, we have the messages we will subscribe. Those are the events that the *view* will send in the *controller* direction.

Messages we will broadcast

Message	Params
display contacts	tAllContactsA
display contact	tContactA

These are the two messages that our controller broadcasts. Orders to display the list of contacts or a specific one.

Intermission

Now, lets do a health check and check if you, dear reader, are still with me. Publisher and Subscriber patterns are not something we usually see in most stacks in the LiveCode community and the primary reason I added *advanced* to the title of this book.

If you're having trouble wrapping your head around it. Don't worry, those things are complex. Checking the source code for both the *controller* and the *view* will help a lot. To summarize using other words: publishers are broadcasting events and subscribers are listening to them. When an event happens (aka is broadcast) the callbacks in the listeners will trigger. The advantage over the normal LC message path is that a single event might trigger multiple callbacks across different stacks. So different parts of your application might listen for the *"list contacts"* message. If that message is broadcasted, they would all receive the event.

For our simple demo stack, this might sound overkill, but once you think in those terms, it becomes a very useful pattern that leads to code that is easier to maintain.

The controller script

Our controller has no UI even though it is a normal LiveCode stack. In the version bundled with this book, I've added label fields and a description so it is easier to spot that window on the IDE. All the code for the controller is in the *stack script* and that's what we will discuss here.

Once the *controller stack* opens, it needs to set up the *subscriptions* for the messages it wants and get out of the way by showing the view. So our preOpenStack becomes:

```
1   on preOpenStack
2       _initializeController
3   end preOpenStack
4
5   private command _initializeController
6       start using stack "aagPubSubLib"
7       start using stack "model.contacts"
8       _setupEventSubscriptions
9       _launchView
10      if the environment is not "development" then
11         hide this stack
```

```
12        end if
13    end _initializeController
```

I've decoupled the `preOpenStack` from the *controller initialization* to make our life easier in case we decide that the *controller* needs to be initialized using some alternative method. The first lines of the code just put our model and publisher/subscriber pattern library into use, after that it does exactly what we mentioned above, it subscribes to the messages it is interested in, display the view and then get out of the way. If you run this code it will fail because we don't yet have the two private commands used, lets build them.

```
1    private command _setupEventSubscriptions
2        subscribe "search contacts", "_searchContacts"
3        subscribe "update contact", "_updateContact"
4        subscribe "delete contact", "_deleteContact"
5        subscribe "save new contact", "_saveNewContact"
6        subscribe "list contacts", "_listContacts"
7        subscribe "get contact", "_displayContact"
8    end _setupEventSubscriptions
9
10   private command _launchView
11       go stack "view.addressbook"
12   end _launchView
```

You might notice a tight coupling there. Our *controller* is assuming the name of our *view stack*. We could solve that with more messages (like having a *show view* message) but to be honest, some coupling is OK. The controller is already putting that stack into use anyway in the `_initializeController` command.

Now, all that is left is implementing the callbacks for all those subscribe calls. As you can see, I've created the messages mimicking the model API (mostly because I am lazy), so each callback will be straightforward. Our commands will communicate back to the view using `broadcast` calls.

```
1    command _listContacts
2        put getAllContacts() into tAllContactsA
3        broadcast "display contacts", tAllContactsA
4    end _listContacts
```

The `_listContacts` command shows a pattern we will reuse in the other handlers. We call some routine from the *model*, wrap the results into a variable and *broadcast* it.

```
1  command _displayContact pContactID
2     put getContactByID(pContactID) into tContactA
3     broadcast "display contact", tContactA
4  end _displayContact
```

In the commands below, we're showing composition by making them rely on the command above.

```
1  command _saveNewContact tContactA
2     put createNewContact(tContactA) into tContactID
3     _displayContact tContactID
4  end _saveNewContact
```

```
1  command _updateContact tContactA
2     saveContact tContactA
3     _displayContact tContactA["id"]
4  end _updateContact
```

There are some extra handlers there to deal with contact deletion, but it is similar to what we already have here. Check out the *stack script* of the bundled source for the complete script.

Summary

In this chapter we dived deep into *controllers*, arguably the most important part of an application, the one thing tieing it all together. We also learned about a very useful pattern called *publisher/subscriber pattern* and how to leverage it from LiveCode. We're now ready to build our *view*.

Views

I am tempted to say that *"views are the most visible part of your application"* but I guess that would be too cheesy. Yet, it is true. Views are what the end users will intact with and what they will grow to assume is the whole of your application. Software is like an iceberg. Your models, libraries, controllers, all belong to the large piece of ice under the sea while your view is the small piece visible above. Users see that piece above the sea and assume that's the whole thing, giving no notice to the vast volume below.

Crafting good user experiences is not only a genuine science, but it is also an art. It is very hard to convey how to do it in a small chapter like this, but I will give you some advice on common tips and tricks. Still, if you want to dive deeper into usability and user interfaces, there is a ton of information and courses online.

Practical tips for interface design

- Don't make your software a scavenger hunt. If the options available for your user are hard to find or require some complex interaction for them to work, you're adding friction to the usage of your software which will lead to more support requests and potentially more bugs as the software is used in the wrong way.
- Group features related to the same domain together. If you have a modal dialog with a button to accept some change or to cancel them, those buttons should be near one another so that when the user focuses their attention to that area, they perceive all options available to them.
- Reduce the amount of clicks needed to navigate inside your software. Try to achieve the "under 3 clicks" dream which makes the user go from any place in your application to any other place in it under three clicks. It is an arbitrary number but if it takes your user 10 clicks to navigate from one part of the application to another they will never go there.

What goes into the view and what doesn't

In some languages, *views* don't have any code in them, they are this *dead thing* and the controller does everything. I think this is a waste of potential and I think *views* should contain the code needed for them to execute their work of being the user interface to your application. For example, consider a financial application that is listing deposits and withdraws from a bank account. Suppose you want to color withdraws in red and deposits in blue. The code to do that can be in the view as it doesn't affect the controller or the model, it is just cosmetics added to provide a better user experience. It is not useless but doesn't affect or rely on stuff from the other units, it is OK to add that code to the *view*.

In LiveCode it is very tempting to add code to views. How many times we have built a whole feature inside a `mouseUp` handler? it is so easy! Later, when it is time to build a different view, or trigger that feature from some other control, we end up with calls like

```
1  send "mouseUp" to button "myButton" of card "that distant card" of stack "some compl\
2  etely different location"
```

If you're building a feature, place it on a model, a library, or a controller depending on the nature of it unless it is something that relates only to the view.

Multiple views are the key to cross-platform applications
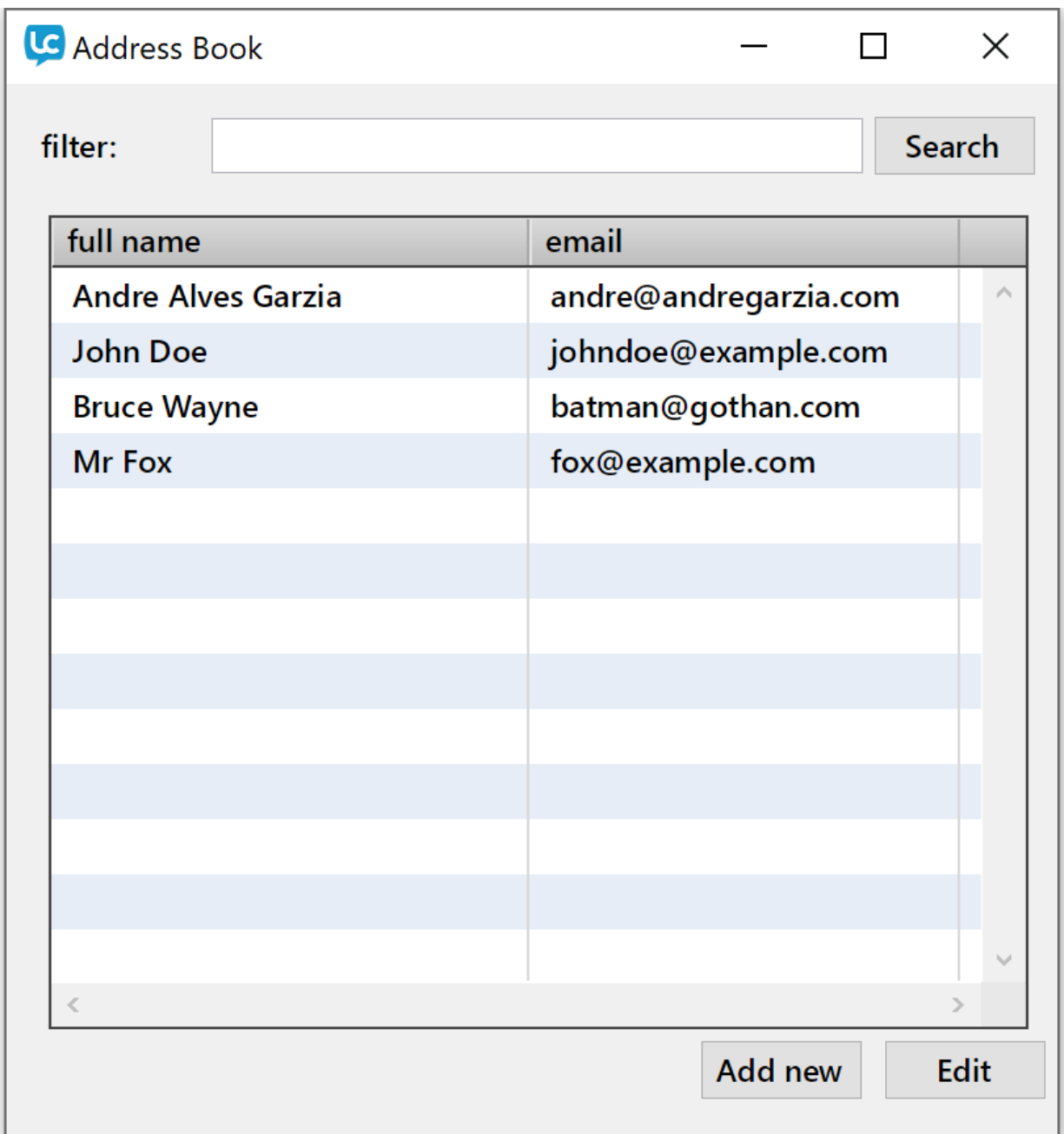
The major benefit of our *loose coupling* and the"*publisher/subscriber pattern* we used in the previous chapter is that there is nothing preventing us from having multiple views. We could have `view.addressbook.mobile` and `view.addressbook.desktop` with the *controller* deciding which one to launch depending on the device being used and everything would still work. You could even launch both of them at the same time and they'd work in tandem because they would both subscribe and react to the same messages.

Building multiple views makes it a lot easier to ship cross-platform code from a single codebase and is much easier than using a single view and `resizeStack` handlers to fix everything at runtime.

We will build a single *view* considering only Desktop usage but there is nothing preventing you from building a second one for mobile. It is an outstanding exercise actually, and you should totally do it.

Practical: Building views for the address book application

For the *view* we'll take a fresh approach. If I guide you into building the entire view, we'll waste a ton of time with silly instructions such as *"place this button at that location with this name"*. Instead, lets just open the bundled stack and I will describe how it works. The *view* file name is `view.addressbook.livecode` and it looks like:

The contacts list view

The best way to follow this chapter is to open the bundled controller called `controller.addressbook.livecode`. By opening it, you'll end up with the library, model, controller, and view in memory.
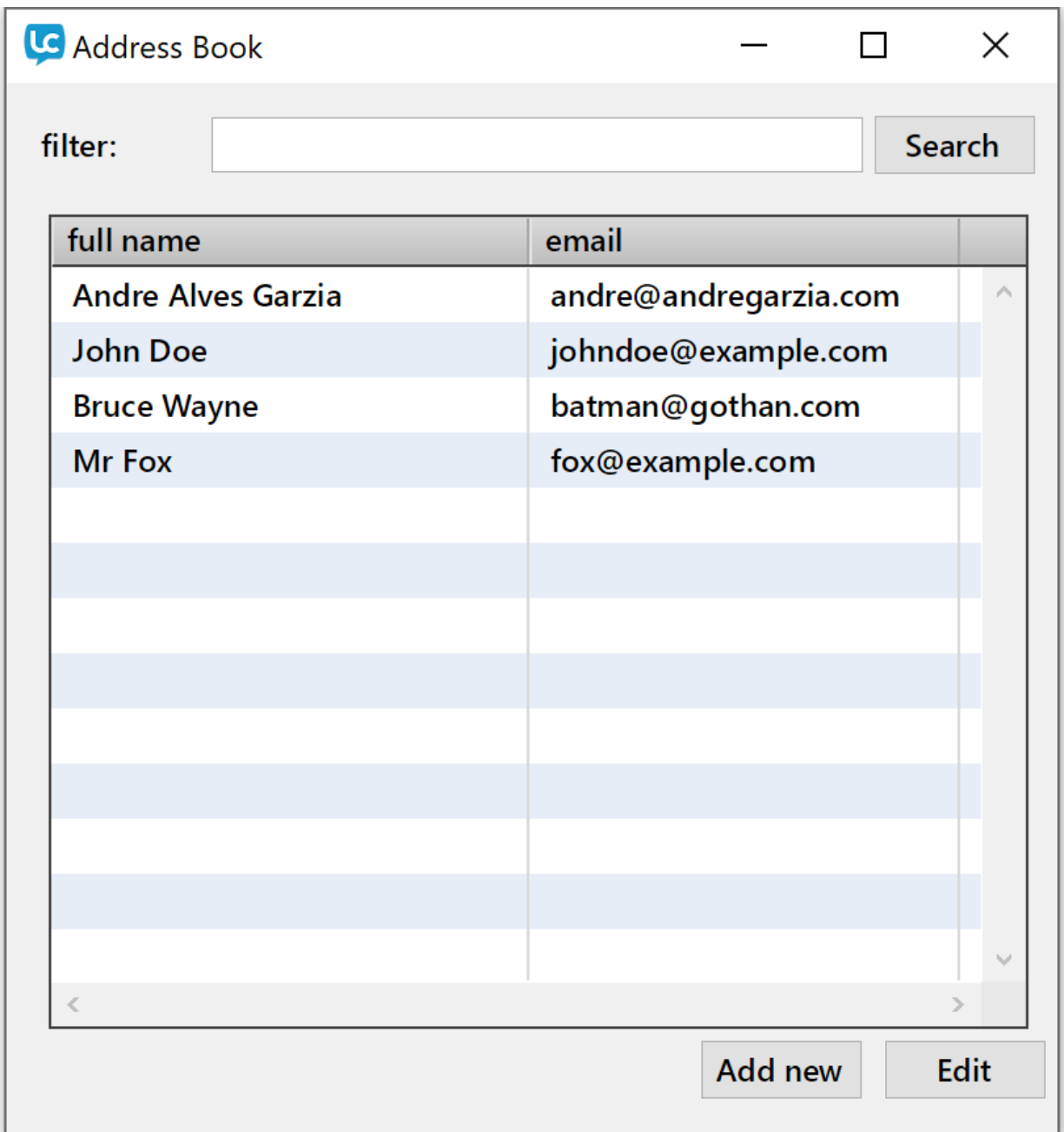
How our view work?

Our *view* is a stack with two cards. We will use one card for listing contacts, this card name is *list view* and another card called *details view* that is used for viewing and editing a single contact detail. Unlike our controller, in which we placed all the logic into the *stack script*, for the *view* we will make use of both *card scripts*.

When a card opens, it will *subscribe* to the messages it needs and when it closes, it will unsubscribe to those messages thus keeping our subscription lists nice and tidy. Each card will only know about the task it needs to do. Each control in the cards has just the minimal amount of code to call a handler on the *card script* to do whatever needs to be done.

There is no business logic in the view, just calls from the controls into the *card scripts*, which will do the message exchange with the *controller* using the *publisher/subscriber* routines.

The contacts list

The contacts list view

This card has the following controls:

- a *label field* for the search feature.
- an *input field* for holding the search terms.

- a *search button* for triggering the search feature.
- a *datagrid* for listing contacts.
- an *"add new contact" button* for adding new contacts.
- an *"edit contact" button* for editing the selected contact in the DataGrid.

If you check the scripts in any button or field, you will find single line scripts calling into the *card script*, which is the interesting part. Lets go step by step into it.

```
1   on preOpenCard
2       _intializeView
3       broadcast "list contacts"
4   end preOpenCard
```

On `preOpenCard` we `_initializeView` and broadcast a message asking for the contact list (which is handled by the *stack script* of the *controller* under the `_listContacts` handler).

```
1   command _intializeView
2       subscribe "display contacts", "_displayContacts"
3       dispatch "resetData" to group "contact list"
4   end _intializeView
```

The `_initializeView` is straightforward, it subscribes for the only message that this card is interested in, which is *list contacts* and it also clears the DataGrid so it doesn't contain leftover data from some previous interaction.

```
1   on closeCard
2       unsubscribe "display contacts", "_displayContacts"
3   end closeCard
```

`closeCard` is all about tidiness, it unsubscribes to that message.

```
1   on searchContacts
2       put field "search terms" into tSearchTerms
3
4       if tSearchTerms is not empty then
5           broadcast "search contacts", tSearchTerms
6       else
7           broadcast "list contacts"
8       end if
9   end searchContacts
```
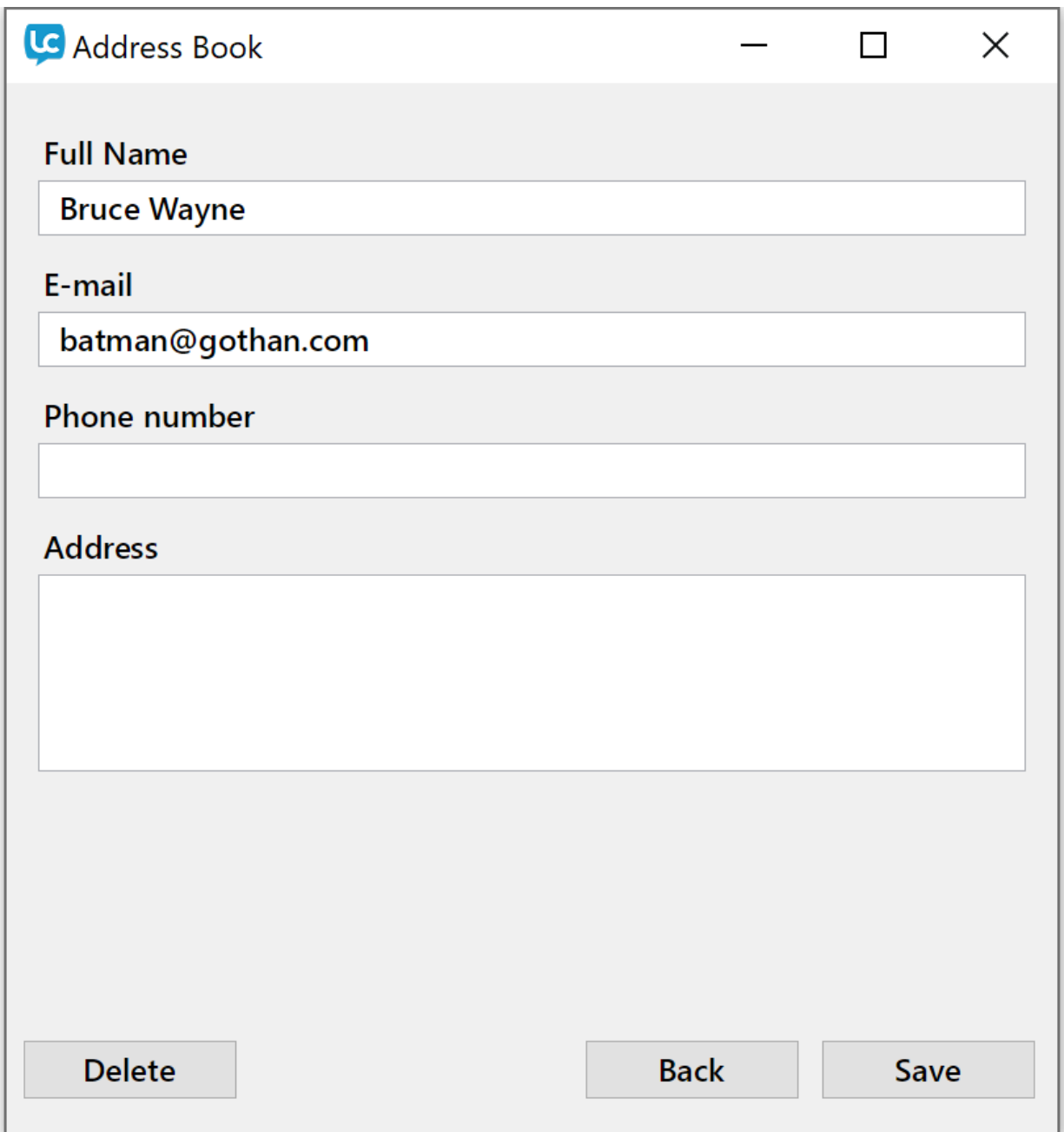
Searching is more complex because it needs to take into account if the user cleared the search field, thus causing all the contacts to be displayed. Searching and listing contacts are two different messages. When they are broadcast, they are picked up by the appropriate handler in the *controller stack script* — _listContacts or _searchContacts — which broadcast the *display contacts* message passing all contacts or the result from searching. So from the point of view of the DataGrid, both cases are nothing but *display contacts*, it is not aware of the search feature; it is completely decoupled from it.

```
1  on _displayContacts pContactsA
2     go card "list view"
3     set the dgData of group "contact list" to pContactsA
4  end _displayContacts
```

For the final handler, which is the one that displays the list of contacts, we first make sure we're in the right card and then set the data onto the DataGrid.

You might wonder where is the code for the *Add New* and *Edit* buttons, but those buttons send events to the *details view* card, their logic is not handled by the *list view* card. This is to enforce separation of concerns. Only the *details view* card know about how to add or edit contacts, all the *list view* knows is how to list contacts.

The details view

The details view

This card presents a collection of fields holding the contact information. There is no logic in the input or label fields. The buttons *Delete*, *Back*, *Save* have the minimal necessary code to call into the *details view card script* which is the script that holds all the logic. Lets go over it.

```
1   local sContactID
2
3   on preOpenCard
4       _clearUI
5       _setupEventSubscriptions
6   end preOpenCard
7
8   command _clearUI
9       repeat for each item tField in "full name,phone,email,address"
10          put empty into field tField
11      end repeat
12  end _clearUI
13
14  on closeCard
15      _removeSubscriptions
16  end closeCard
17
18  private command _setupEventSubscriptions
19      subscribe "deleted contact", "_deletedContact"
20      subscribe "display contact", "_displayContactDetails"
21  end _setupEventSubscriptions
22
23  private command _removeSubscriptions
24      unsubscribe "deleted contact", "_deletedContact"
25      unsubscribe "display contact", "_displayContactDetails"
26  end _removeSubscriptions
```

Let's analyse a longer chunk of the script this time. On preOpenCard we clear the user interface to remove any leftover data from previous interaction and also subscribe for the messages we want while still being careful to unsubscribe from them if we move to another card.

The most interesting part is the use of a *script local variable* to hold the ID of the contact we're editing. There were other ways of solving this but this was quick and won't cause too much trouble as it can only be changed by the *card script*. This variable is also used to figure out if we're adding a record or editing an existing one. If sContactID is empty then, it is a new record.

```
 1  command addNewContact
 2      go card "details view"
 3      put empty into sContactID
 4  end addNewContact
 5
 6  command editContact pID
 7      go card "details view"
 8      put pID into sContactID
 9      broadcast "get contact", sContactID
10  end editContact
```

As explained above, editing or creating a record is a matter of holding the ID of an existing record in sContactID or leaving it empty. The commands above are the ones called by the buttons on the list view card. In the case of editContact we also need to broadcast a message requesting the contact details. We could get those details from the DataGrid itself but to ask for them again is good practice as we could have built this in a way that the DataGrid only receive a subset of each contact data.

```
 1  command _displayContactDetails pContactA
 2      go card "details view"
 3
 4      repeat for each item tField in "full name,phone,email,address"
 5          put pContactA[tField] into field tField
 6      end repeat
 7
 8      put pContactA["id"] into sContactID
 9  end _displayContactDetails
```

The command above is the callback triggered from a broadcast from the *controller stack*. When our *view* asks for a contact information using the *get contact* message, the *controller* responds by broadcasting a *display contact* message which ends up triggering this command.

```
 1  command saveCurrentContact
 2      repeat for each item tField in "full name,phone,email,address"
 3          put field tField into tContactA[tField]
 4      end repeat
 5
 6      if sContactID is a number then
 7          put sContactID into tContactA["id"]
 8          broadcast "update contact", tContactA
 9      else
10          broadcast "save new contact", tContactA
11      end if
12  end saveCurrentContact
```

Saving a contact is just a matter of assembling an array with the data and broadcasting the appropriate message. There is a bit of code there to double check if we're editing an existing record or adding a new one. All that is left is deleting records. The code below takes care of it. There is a broadcast to delete a contact and a callback so we know it worked.

```
1   command deleteCurrentContact
2      if sContactID is not a number then
3         go card "list view"
4         exit to top
5      end if
6
7      broadcast "delete contact", sContactID
8      go card "list view"
9   end deleteCurrentContact
10
11  command _deletedContact
12     answer info "Contact Deleted!"
13     put empty into sContactID
14     go card "list view"
15  end _deletedContact
```

There you have it! Congratulations! You now have a complete MVC-based LiveCode application.

Going Forward

Now it is time to talk about extra topics which don't exactly fit into our MVC model but are crucial to achieving a healthy development workflow. For the rest of the book we'll approach these different topics one by one without each chapter building upon the previous chapter. Cherry pick which of the techniques you want to use from now on but keep in mind that *Libraries* which is the topic of the next chapter is actually very important and will probably be used by all of your medium and large projects.

Extra: Using Libraries

There are parts of the code that don't fit the MVC model. For example, code to trim extra spaces in strings is useful, but it doesn't really matches our current paradigm of models, views, and controllers. These reusable pieces of code, which might be shared among different projects, should be made into *libraries.*

> The value of a library lies in the reuse potential of the behavior. When a program invokes a library, it gains the behavior implemented inside that library without having to implement that behavior[^b1] itself. Libraries encourage the sharing of code in a modular fashion, and ease the distribution of the code. — (from the Wikipedia definition of Libraries[21]).

Most intermediate developers using LiveCode are already familiar with libraries. They are the most common way of sharing code in our community.

What goes into a library and what doesn't

Libraries are self-contained collections of routines different applications can make use of. You won't place code related to your business logic in a library. If your application is solving project management in some novel way, that novel way is not what goes into a library (it probably belongs in a controller). If the same application has auxiliary routines to read files from OmniProject and MS Project, and that sounds like something that might be useful beyond your current application, something that could be shared between multiple products. Then those auxiliary routines make an excellent opportunity to create a *library for project file handling*. Did you notice that it sounds suspiciously like a model? A library can be a reusable model. Do not worry too much about it. In my personal opinion the most important part of MVC is not MVC at all but the practice of decoupling parts of the project and organizing them in domain specific collections.

Most of the libraries are composed of **generic code, stuff that is tedious, repetitive, and useful**. An example of it is my own DB Lib[22] which is an ORM[23]-like library for LiveCode to make it easier to work with databases. This library mimics the Active Record pattern[24] to help work with RDBMS[25] without writing SQL. It is useful code, but it is very boring to write for each application (and also a waste of time), so I made it reusable.

---

[21]https://en.wikipedia.org/wiki/Library_(computing)
[22]https://soapdog.github.io/livecode-dblib/#/
[23]https://en.wikipedia.org/wiki/Object-relational_mapping
[24]https://en.wikipedia.org/wiki/Active_record_pattern
[25]https://en.wikipedia.org/wiki/Active_record_pattern

Libraries make a community more powerful

It is important to know that you'll not only be writing your own libraries. You will probably end up using third-party libraries made by other community members. The more libraries we make and share (even for a profit), the more powerful our community gets. Leveraging other people's expertise is a good way to add features to your application. A good example is when you leverage third-party libraries for social network integration. You may not feel comfortable programming all that *webby network code* but you know you can use it once it is available to you through an easy-to-use library.

There is no successful programming language with an ecosystem devoid of good libraries, after all there is only so much you can bundle with your language. So when you're building your own applications, consider if there is code there that can be made into a library. If there is, then do it and make it available to the rest of us. It doesn't really matter if it is a paid or free library; it is more important that you make it available.

This is the first time in this book we're exploring the availability of third-party code and how easy it is to use a third-party library.

Practical: Adding support for vCard in the address book application

In this practical section, we will use an ancient library I made for handling the vCard[26] file format. This will allow our demo application to import and export address book entries. I made that library in 2005 and it still works fine with LC even after more than a decade. It is the only vCard library in our community if I remember correctly. For this book, I've updated the library and made it into a plain-text stack which you can fetch from its GitHub repository[27].

> vCard, also known as VCF (Virtual Contact File), is a file format standard for electronic business cards. vCards are often attached to e-mail messages, but can be exchanged in other ways, such as Multimedia Messaging Service (MMS), on the World Wide Web, instant messaging or through QR code. They can contain name and address information, telephone numbers, e-mail addresses, URLs, logos, photographs, and audio clips. — vCards on Wikipedia[28]

vCards are plain-text files using a structured text format called *vObject*. If we open a vCard file using a text editor, we'll see something similar to this:

---

[26]https://en.wikipedia.org/wiki/VCard
[27]https://github.com/soapdog/livecode-vobjectlib
[28]https://en.wikipedia.org/wiki/VCard

```
 1   BEGIN:VCARD
 2   VERSION:2.1
 3   N:Gump;Forrest;;Mr.
 4   FN:Forrest Gump
 5   ORG:Bubba Gump Shrimp Co.
 6   TITLE:Shrimp Man
 7   PHOTO;GIF:http://www.example.com/dir_photos/my_photo.gif
 8   TEL;WORK;VOICE:(111) 555-1212
 9   EMAIL:forrestgump@example.com
10   END:VCARD
```

And your computer or phone might render this like:

**Windows People application showing the same vCard as above.**

The complete files for this practical section are included in a folder called *vCard Support* which is included in the source code you received for this book.

Using a third-party library

Inside the folder *extra chapters/libraries* you'll find a file named `vObjectLib.livecodescript`. Copy this file to the same folder where your project is. It should be placed just next to your controller file.

> The *extra chapters/libraries* folder contains a full copy of the final project for this section.

The first step is altering our controller so it loads the `vObjectLib.livecodescript` file when it launches. The way LiveCode works is that once you tell it to load a stack by passing it a stack name, it looks into the `the stackfiles` property of the current stack that is trying to load the other stack. This is a property that matches stack names to stack files which enables the developer to refer to stacks by name even if they are not opened. We need to add `vObjectLib.livecodescript` there. To do it, open the *stack inspector* for the *controller stack* and browse to the second tab, the one with the icon that looks like two computer files overlapping. Then use the little open file button in the top right corner to navigate to `vObjectLib.livecodescript`. After you select and open it, the properties should look like:
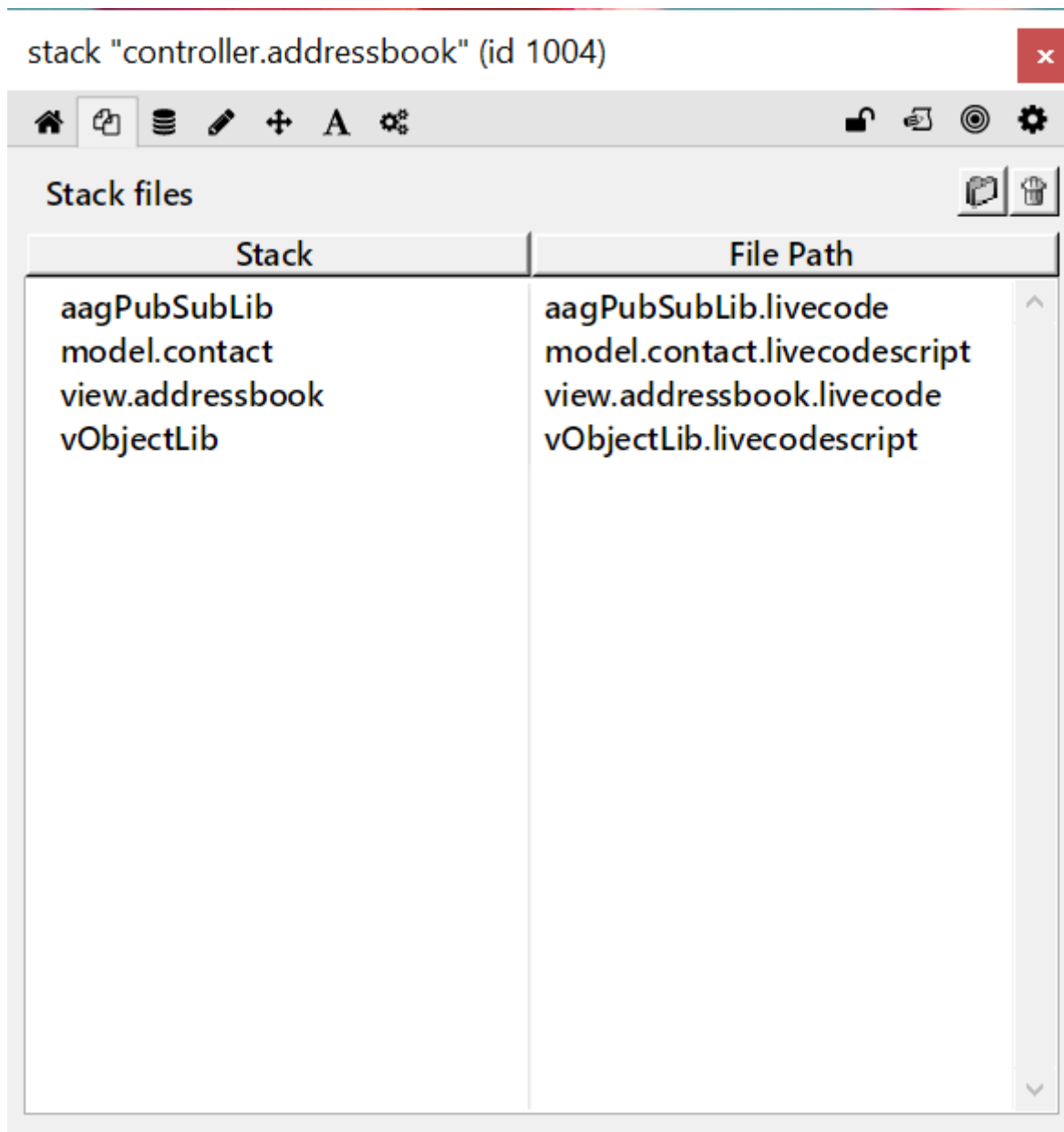
Stack inspector for the controller stack showing the stackfiles property.

IMPORTANT: Double check that the right-side column list the file names without their folders in it. If you copied the file to the same folder as the controller you're using, adding it to the stack files should not contain a folder.

Now that LiveCode knows where to find that file, lets change the `_initializeController` handler in the *controller stack script* to:

```
 1  private command _initializeController
 2     start using stack "aagPubSubLib"
 3     start using stack "model.contact"
 4     start using stack "vObjectLib"
 5     _setupEventSubscriptions
 6     _launchView
 7
 8     if the environment is not "development" then
 9        hide this stack
10     end if
11  end _initializeController
```
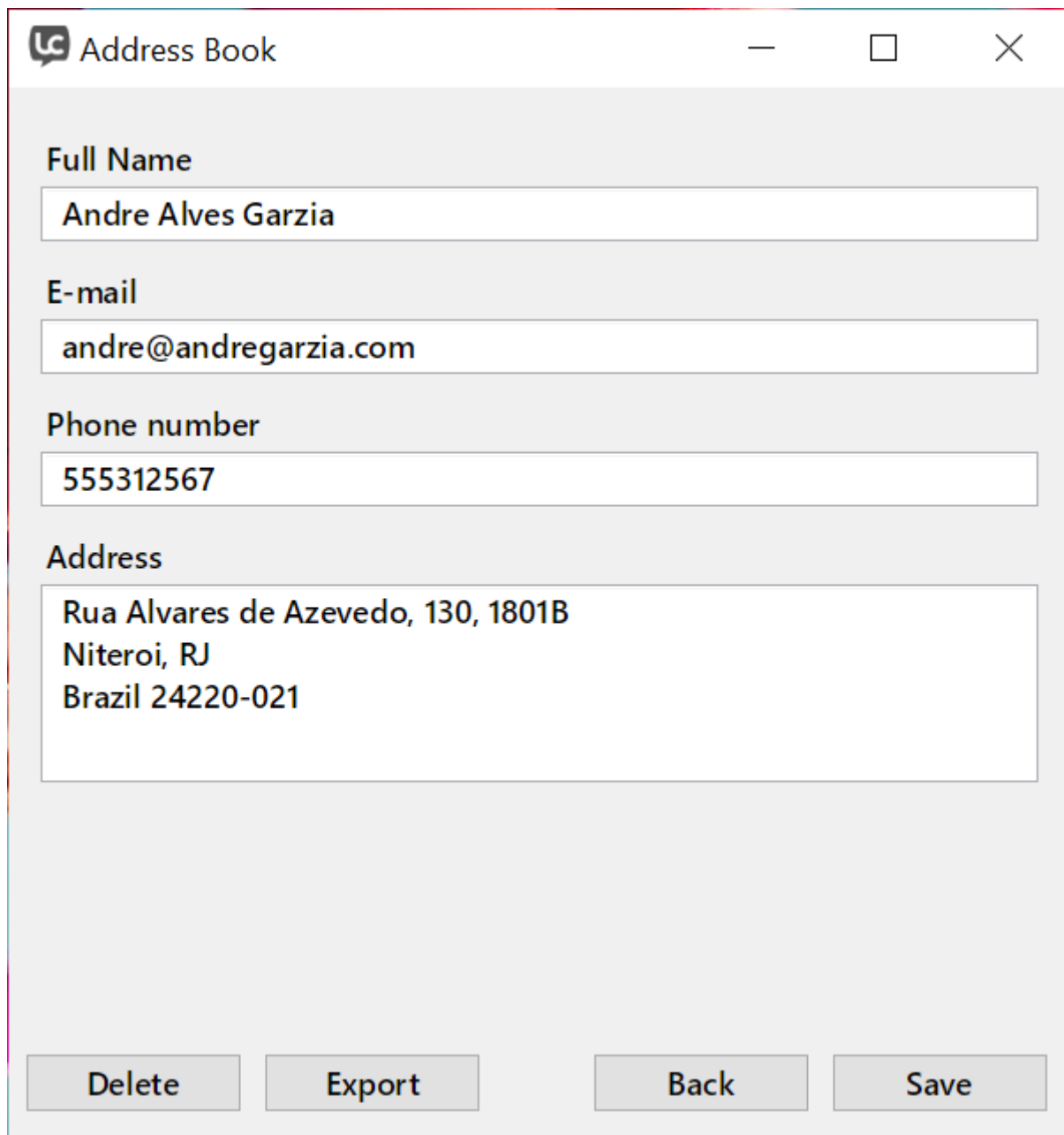
The addition we made is just loading `vObjectLib` on the top part of the `_initializeController` handler. Now, our contacts app can benefit from the features offered by this library.

Exporting a contact

Our first change is to add a feature to enable the user to export a selected contact as a *vCard* that can be read by other apps. Let me summarize the changes we will make and then we'll go into more details. To add this feature we will add a new button to the *details view* card of our view. This button will call a new handler in the card script, which will broadcast a message to the controller. The controller will have the code to export the *vCard*.

Adding an `Export` button to the *details view* card

In our view stack, lets change the *details view* card and add an export button to it.

Our *details view* with an export button

The script for that button is simple, it just calls a handler we will implement on the *card script.*

```
1  on mouseUp
2      exportCurrentContact
3  end mouseUp
```

Adding `exportCurrentContact` the *details view* card script

In the card script for the *details view* card, add the following handler:

```
1   command exportCurrentContact
2       repeat for each item tField in "full name,phone,email,address"
3           put field tField into tContactA[tField]
4       end repeat
5
6       broadcast "export contact", tContactA
7   end exportCurrentContact
```

The script is similar to the `saveCurrentContact` script but we need not disambiguate between new or existing contact for exporting.

Listening for the `export contact` event

In our controller stack script, lets add the new event inside the `__setupEventSubscriptions` handler.

```
1   private command _setupEventSubscriptions
2       subscribe "search contacts", "_searchContacts"
3       subscribe "update contact", "_updateContact"
4       subscribe "delete contact", "_deleteContact"
5       subscribe "save new contact", "_saveNewContact"
6       subscribe "list contacts", "_listContacts"
7       subscribe "get contact", "_displayContact"
8       subscribe "export contact", "_exportContact"
9   end _setupEventSubscriptions
```

All we did so far is register a new event at the end of the script matching the `export contact` to a handler called `_exportContact`. This handler is the one that will call the routines from the `vObjectLib` library.

> We just changed a private command in the stack script, which means that we can't call it simply by right-clicking the stack and selecting it from the popup contextual menu. This means that even though you added the correct code for the stack to start listening to the new event, it didn't actually run and the stack is not listening to that event at all. The easiest way to fix this is to quit LC and open it again since that code runs when the stack opens.
>
> In a real life project, we'd have tools in place to make sure we can add and remove events without the need to quit LC.

It is easier if I show the script for _exportContact and then explain how it works. I will explain the calls from vObjectLib only briefly, they are not the subject for this section. We can fetch a copy of the library from this repository on github[29].

```
1   command _exportContact tContactA
2       put empty into vCard
3       vCardBegin vCard
4       vCardAddFullName vCard, tContactA["full name"]
5       vCardAddEmail vCard, tContactA["email"]
6       vCardAddPhone vCard, tContactA["phone"]
7       vCardEnd vCard
8       vCardSerialize vCard
9       ask file "File to save" with (tContactA["full name"] & ".vcf")
10
11      if it is not empty then
12          put vCard into URL ("file:" & it)
13      end if
14  end _exportContact
```

How vObjectLib works is by populating the content of a variable over multiple calls, so in the code above, that variable vCard is being passed by reference to all the handlers exported from vObjectLib. The vCardBegin command starts a *vCard* and the vCardEnd closes it. Between those two commands you can call commands to add details to that *vCard*. In this example we're adding the name, email and phone number of the contact we want to save.

The *vCard* variable structure is not the same structure that a *vCard file* would have on disk, so to save it we need to serialize that variable into the correct format which is done with the vCardSerialize command. After that, the content of the vCard variable matches what needs to be saved on disk, and we can proceed by using normal LiveCode commands to save the content of a variable to a file.

Below are two screenshots: one from the *details view* card showing a specific user's detail, the other from the Windows 10 built-in People program showing the imported *vCard file.*
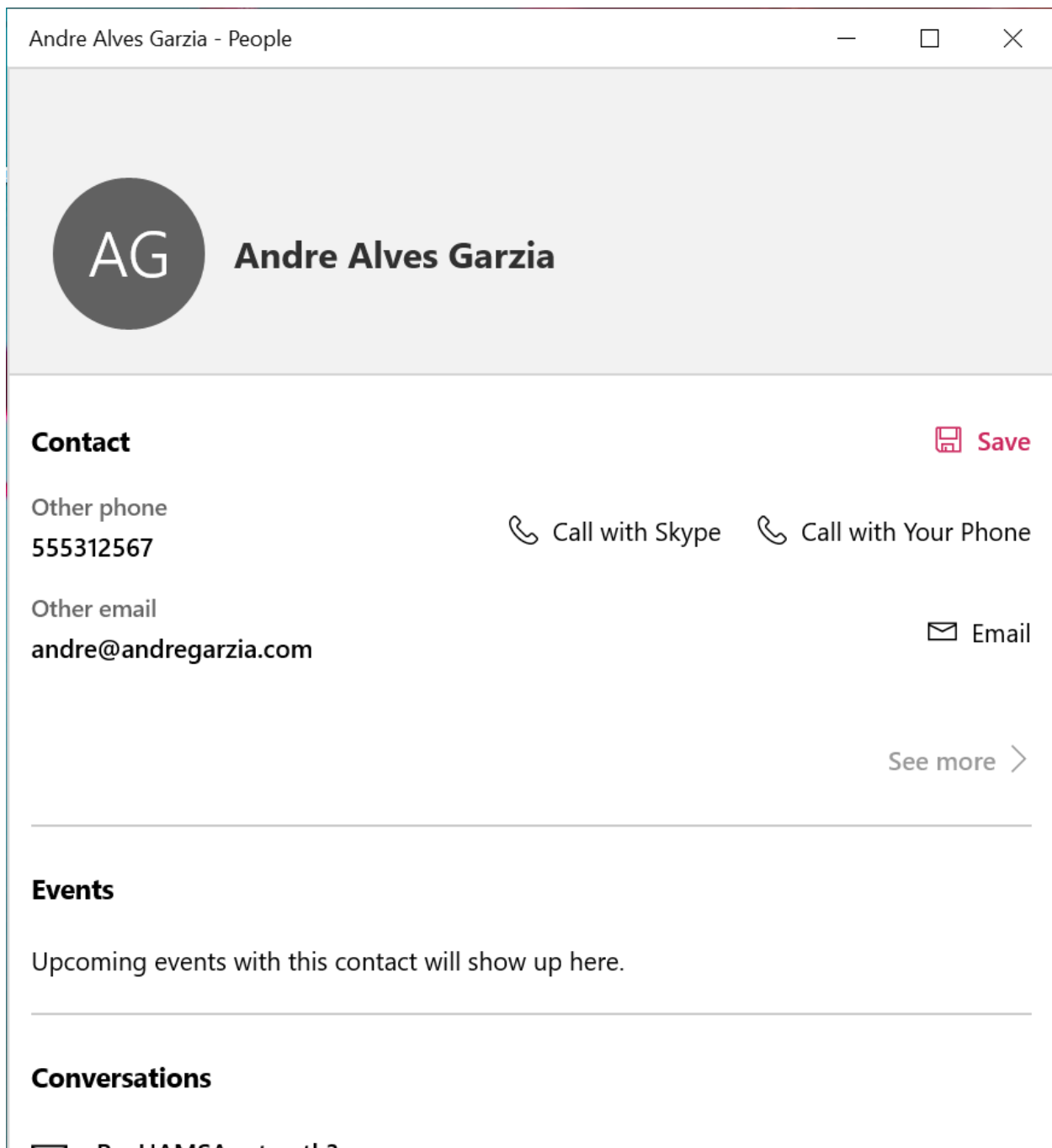
---

[29]https://github.com/soapdog/livecode-vobjectlib

Details for a given user

Same user as seen by Windows 10 People app

How to create libraries

These days most libraries are *plain-text stacks* because that format is easier to use with *version control system* like git. This stacks do not contain cards and controls, they are just a small nimble

container for a stack script. You can use binary stacks in case your library needs complex controls, they are just harder to use with *git* and similar tools.

All the handlers you add to a stack script are available to any stack that `starts using` your stack unless you declare that specific handler *private*. Thats basically it. It is interesting to check the `vObjectLib.livecodescript` and the `aagPubSubLib.livecode` to see how they are built. Both have the features they export in their stack script but one has been refactored as a plain-text file and the other still a binary stack (because I've built it eons ago).
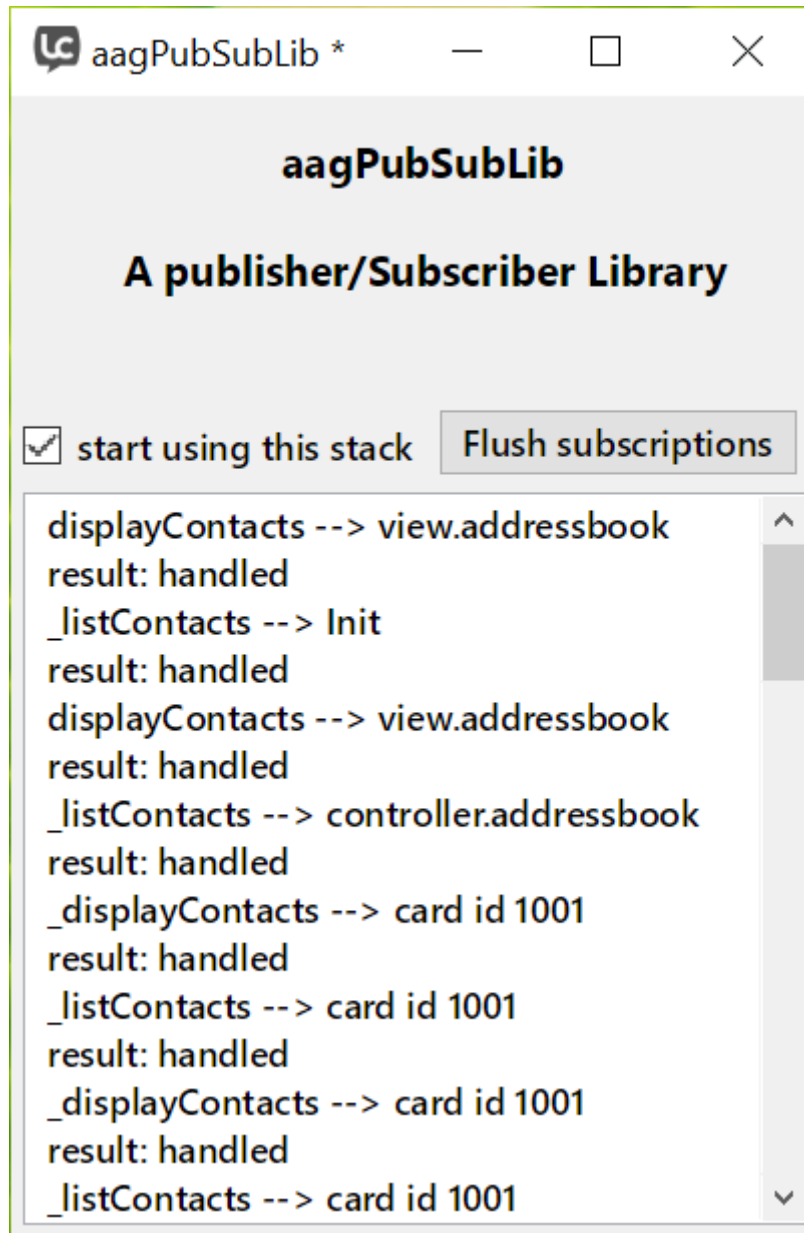
aagPubSubLib has extra features since it is a binary stack

As you can see in the shot above, the `aagPubSubLib` has some extra features since it is a binary

stack and can benefit from having cards and controls in it. It has a button to clear all the event subscriptions and a field that logs all the events dispatched from it. These features are only useful for debugging when you're developing your app, but they are very convenient and couldn't be easily replicated with a single plain-text stack.

Whats the difference between libraries and the MVC stuff again?

Let's revisit some content discussed at the beginning of the chapter. The mechanisms by which libraries work, is the same mechanism that enables us to use our different views, models and controllers in our projects. We've been using a library all along, which is the *pubsub library* and the distinctions between when to use one or another might still be a bit murky.

There are some key aspects of libraries which are:

- They are meant to be reused. They don't contain your business logic. They are agnostic of the kind of app you're doing and deal only with self-contained pure features.
- They are *tightly coupled* unless the author went an extra mile to support some pubsub mechanism (most don't). So when you *start using* a library, you put the handlers from that library into the message path and then call them directly from your code. This means that if you switch libraries, you need to hunt down all the times you called that previous library directly and refactor that code.

Libraries are not tied to the MVC paradigm and can be used by any kind of application. So if you build something useful that you want to share, you shouldn't assume the user is using an MVC paradigm like the one described in this book.

Conclusion

We've saved a lot of work by using reusable code that was shared by another developer (in this case by yours truly). There are a lot of libraries in the LiveCode community, some are open source software and free, others are paid products made by independent software vendors. I encourage you to check if complex parts of your source code can be made into libraries.

Still, you need to know that some of these libraries are shipped in *password protected stacks* so you can't see their source code and in fact can't see the internal working of the library at all. I will not delve into the reasoning behind this practice because I have very strong opinions about this. Let it be known that I always ship my own libraries with their source code visible, and I think this is better.

Summary

In this chapter you've learned how to harness the power of reusable libraries. You've used a third-party library which enables you to add features to your app without the need to write them.

on closeBook

This is the end of the book, I know I want more too. In the next months I plan to do more 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 youâ€™re building.
- **How to document your code** to save precious time 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 reach out to me with comments, feedback and request for new topics or clarification. My email for this book is feedback@andregarzia.com[30].

Writing this book has been a great experience for me and Iâ€™d really appreciate more feedback about it.

I'd love to write more books. I see myself more as a writer than a developer these days. If you like my work and would like like to support me buy me a coffee at ko-fi[31].

See you all around!

Andre

---

[30] mailto:feedback@andregarzia.com
[31] https://ko-fi.com/andregarzia