UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

IOS APP GENERATOR

Montini Nicola - 621802

Academic Year 2011/2012

To my family,
who have supported me over the years.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 THESIS' PURPOSE

The thesis's purpose is the construction of an editor of interactive books for kids. The idea of constructing interactive books started from developers of the Altera Company. About a year ago they got the request to translate the contents of a children's book into a digital format readable by a tablet pc like an iPad. As soon as they started they immediately saw the full potential of this kind of product: the possibility to make the book interactive was very clear. They decided with Ware's Me (the customer company) to insert touchable elements that react to some actions done with fingers and with voice. Clearly, for a kid starting to explore the world around himself, the possibility to read and watch a book with animated components, manageable objects, with a real speaking voice and stuff like that is definitely more interesting than a simple printed book. We have to say that this idea is not new: there are many interactive books already available on the App Store but in Italy there is no example of this kind of product.

The developers of Altera started working on this project with great power and they managed to build two interactive books: the first one, "Billo e il Filo", is a transposition of the homonymous tale written by Silvia So and illustrated by Nicola De Bello; the second one, "Chissà" is the translation of the tale written by Marinella Barigazzi and illustrated by Ursula Bucher. After several weeks on this last project the team reached a suitable version for publishing on the App Store and they decided to do it. Nowadays you can find the application there and download it. As a result of creating such a kind of App a local newspaper, interested in this new program, really decided to write an article about Chissà. Satisfied with the success of the application, Altera decided to use more power and find a way to build interactive books in a faster way. For this purpose they had the idea to create AppKid, a framework aimed to build this kind of applications without writing code, but rather editing a file (more specifically an XML[1] file), which depicts the features of what the user should see on the screen and what the user should tap on. They did not stop here and they wanted to construct an editor that would give every kind of people willing to create an interactive book the opportunity to do this. This was the situation when I started my work, which consisted in developing the framework and creating the editor.

1.2 THE ALTERA COMPANY

Altera deals with design, development and integration of web applications; its own skills go from Java programming to information security and cryptography.

---

[1] XML is a markup language used to represent data.

[2] For more details about Cocoa, see paragraph 3.3

[3] To get an object to do something, the programmer sends it a message telling it to apply a

Altera was born in 1999 with the aim of realizing customized web and e-commerce applications for the specific requirements of the customer. Over the years Altera developed various web projects and they extended their supply to Desktop (with Mac client applications) and to Mobile environment (iPhone and iPad applications). In 2011, Altera started a partnership with Ware's Me, a society dealing with digital publishing of interactive books designed for kids.
Altera is managed by Matteo Centro (founding member), who coordinates business and technical aspects, with particular attention architecture and software development activity. He coordinates a solid network of internal and external collaborators in the software development area, in the graphic area, in web marketing and in search engine optimization.

TECHNOLOGIES

The preferred development platform of Altera is WebObjects, which is based on the most sophisticated and mature database abstraction framework in the marketplace: the NeXT's Enterprise Objects Framework; it has been one of the first multi-tier web application servers and allows to release application on any kind of J2EE application server. WebObjects is distributed by Apple, free of charge. As a quality assurance, among the most important references for WebObjects there is Apple itself, because Apple used it to build, for example, iTunes store.
Inheriting experience from WebObjects, Altera started developing applications on Mac OS and iOS platform, using Cocoa and Cocoa Touch technologies provided, once again, by Apple.

1.3 THESIS' STRUCTURE

This thesis is organized in eight chapters, and this introduction is the first one. The second chapter deals with a brief history of interactive books, explaining the first approach used to build an interactive book from a printed one. It also explains the problems of constructing an App for an interactive book and what can be done to automate and speed-up this process. The third chapter is a digression and a deepening of the powerful technologies provided by Apple and used to create the framework and the editor. It starts with introducing the Objective-C programming language and then it goes on by explaining the Model-View-Controller design pattern, which is used by Apple developers to construct their applications and which the whole Cocoa framework is based on. The chapter ends with a description of the frameworks provided by Apple to build applications on Mac OS X and iOS. The last paragraph describes Sparrow, an open-source framework used here to make the management of animations and graphics easier. The fourth chapter is a description of the analysis phase, where we decided the goals of the project, after examining the architecture and implementation of the first generation of interactive books. The fifth chapter describes the design phase of AppKid Framework, after the listing of the motivations and the benefits that using a software framework can give in order to construct interactive book applications. The sixth chapter talks about AppKid Editor, and it starts describing, once again, the advantages of

using an editor, and the problems that it can solve for Altera and for anyone that wants to build an interactive book. Then a description of the architecture of the editor follows, explained in terms of the Model-View-Controller design pattern. In the seventh and last chapter we can find conclusions, a summary of what has been done so far, and a plan of what has to be done in the near and far future. At the end there is an appendix, with a detailed explanation of the particular technologies used during the development.

2 BIRTH OF INTERACTIVE BOOKS

In this chapter we introduce the concept of interactive book, explaining why they were born.

2.1 DEFINITION

Interactive children's books are a subset of children's books, which require participation and interaction by the reader. Participation can range from books with texture to those with special devices used to help teach children certain tools. The first interactive books known by the audience are movable books, defined as "covering pop-ups, transformations, tunnel books, volvelles, flaps, pull-tabs, pop-outs, pull-downs, and more, each of which performs in a different manner." (Interactive Children's Books). Thus, interactive books started being sold as printed books, with components that the reader (usually a kid) can manipulate to improve entertainment and participation. Of course a child can not read, but he can be fascinated by images, colours, shapes and sounds; these kind of objects have universal meaning and this is why also a very young kid can understand and get involved in "reading" books containing such kind of objects. Unfortunately we know that printed books can not contain something different than static images or words.

2.2 TRANSLATING A PRINTED BOOK INTO AN INTERACTIVE BOOK

The main limit of a printed book is the fact that it is made of paper, so it obviously cannot contain sounds, dynamic images or moving objects. The possibility to insert covering pop-ups, flaps, pull-tabs and stuff like that in a printed book, can improve the reader experience, but we are pretty sure that translating a printed book into a digital format opens up a very large uncharted area: first of all adding sounds makes the reader paying more attention than it would do by simply looking at images. Pictures can be combined with sounds in many different ways and this can lead to original and very diversified pages. This result is unthinkable in a printed book and it is not the most amazing; animations are even more interesting: we can in fact decide to animate an object in a page and make it move following a predefined path, or making it moving randomly on the screen. Imagine a tale that talks about a fish swimming in the sea: there is a big difference between looking at the static picture of the fish and looking at the fish that moves on the screen with the underwater view of the sea in the background. Although this can be a small difference for an adult reader, for a kid it can be magic and he will be really entertained by this. This is only a little example, because in a digital world almost everything on the screen can be animated; for example, if the author wants to focus the attention on a particular object he can make it appear after

other ones with a fade in effect, like a scene that fades in after another one in a movie. These effects are really nice to see and help the author to create a more interesting scene to watch.

Summarizing, the most important innovation is the transformation of the "reading" action from passive to active: the kid can actually *change* the picture on the screen, moving the objects that are provided to be interactive with his fingers. This is the concept of interactivity and with this great novelty we think that the kid could have a new, entertaining experience. Nevertheless, in printed interactive books there are interactive parts too, but they are not as interesting as the digital ones: the former ones let the child move some parts of the page, usually with very small consequences, when the latter ones can lead to very complicated behaviour. Imagine for example a story that talks about a shining star in the sky: the author of the book can insert the picture of a star on a page of the digital book and make it shine like it would do in the real world when the user touches it (this is not possible in a printed interactive book). Coming back to the fish example, we said that it would be nice to see the fish moving underwater, eventually seeing its fin animated, but the chance to acquire control of the fish and drag it on the sea background would be really amazing for a little child that is starting to explore the world around himself.

FIGURE 2.1 - AN IPAD

These are only some ideas, but the combinations are very large: as said before, almost everything on the screen can be animated, and everything that can be animated can be made touchable or draggable. These two important aspects, animation and interactivity, are the ones on which the development of the framework and of the editor focused on primarily.

Another important aspect was the choice of the device to reproduce the digital book on. It could have been a computer, but, a kid can not interact with it easily; furthermore, the need of using a touch screen makes the computer unusable. We could think of using the mouse as the interaction device, but this would lead to a lack of the user's participation degree. For maintaining this degree high, a touch device would have been much better and for this reason the choice was a tablet: it has a touch screen, it's compact and it has enough hardware resources to reproduce an interactive book. Altera also thought about using a simple e-reader and the most successful products in this area are the Kindle (produced by Amazon®) and the Nook (produced by Barnes & Nobles®). Unfortunately, they do not provide interactivity. Developers of Altera, in conjunction with Ware's Me, conscious of all of these arguments, decided to use all their experience, gained over the years, on iPad

programming and they chose to invest on interactive books applications using this device.

After this discussion we can conclude that the result of the translation of a printed book into an interactive one leads to something that is not a simple text, but something in the middle of a reading and a game. We know that kids like to play a lot and are fascinated by things they have never seen before. With this new kind of book we give them both a game and software able to reproduce books that can be deeply different one from another, not only in the contents, but also in the way they interact with them. Moreover, there is a learning aspect too: with this new kind of experience parents can teach their sons aspects of the world around us in a more realistic way. The innovations do not stop here, in fact Altera put in the interactive book other sections than the reading one: they inserted little games that kids can play with, such as puzzles and pictures that children can fill with their fingers after choosing the right colour. Another interesting tool is the possibility to record the voice of a speaker and reproduce it instead of the default voice. In this way parents can record their voice and let the baby listen to the story told by their parents. By the way, my work focused only on the reading part, but also these important features could be added to the framework in the future.

3 INTRODUCTION TO APPLE INC. TECHNOLOGIES

In this chapter we give a brief description of the technologies used to build the applications (framework and editor) and used by anyone who wants to develop an iOS or Mac OS X application. iOS (formerly iPhone OS) is Apple Inc.'s mobile operating system. Originally developed for the iPhone, it has been extended to support other Apple devices such as the iPod Touch, iPad, and Apple TV (iOS). Mac OS X is a series of Unix-based operating systems and graphical user interfaces developed, marketed, and sold by Apple Inc. Since 2002, Mac OS X has been included with all new Macintosh computer systems. It is the successor to Mac OS 9, released in 1999, the final release of the "classic" Mac OS, which had been Apple's primary operating system since 1984.

We start with a definition of the Objective-C programming language, then we go on explaining the Model-View-Controller design pattern, an important software architecture used in software engineering and in particular in the Cocoa framework. In the third paragraph the Cocoa framework is presented, followed by Cocoa Touch in the fourth one. In the last paragraph we talk about Sparrow, an open source framework built to make the development of multimedia applications easier on iOS platform.

3.1 THE OBJECTIVE-C PROGRAMMING LANGUAGE

The Objective-C language is a simple computer language designed to enable sophisticated object-oriented programming. Objective-C is defined as a small but powerful set of extensions to the standard ANSI C language. Its additions to C are mostly based on Smalltalk, one of the first object-oriented programming languages. Objective-C is designed to give C full object-oriented programming capabilities, and to do so in a simple and straightforward way (Apple). Primarily Brad Cox and Tom Love created it in the early 1980s at their company Stepstone (Objective-C).

Cocoa [2] is pervasively object-oriented, from its paradigms and mechanisms to its event-driven architecture. Objective-C, the development language for Cocoa, is thoroughly object-oriented too, despite its grounding in ANSI C. It provides runtime support for message dispatch and specifies syntactical conventions for defining new classes. Objective-C supports most of the abstractions and mechanisms found in other object-oriented languages such as C++ and Java. These include inheritance, encapsulation, reusability, and polymorphism. Nevertheless, Objective-C is different from these other object-oriented languages, often in important ways. For example, Objective-C, unlike C++, doesn't allow operator overloading, templates, or multiple inheritance.

3.1.1 THE OBJECTIVE-C ADVANTAGE

[2] For more details about Cocoa, see paragraph 3.3

Every Objective-C object hides a data structure whose first member—or instance variable—is the isa pointer. (Most remaining members are defined by the object's class and superclasses.) The isa pointer, as the name suggests, points to the object's class, which is an object in its own right (see Figure 2-1) and is compiled from the class definition. The class object maintains a dispatch table consisting essentially of pointers to the methods it implements; it also holds a pointer to its superclass, which has its own dispatch table and superclass pointer. Through this chain of references, an object has access to the method implementations of its class and all its superclasses (as well as all inherited public and protected instance variables). The isa pointer is critical to the message-dispatch mechanism and to the dynamism of Cocoa objects.

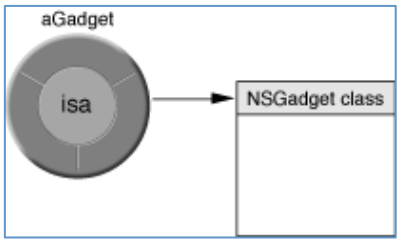

FIGURE 3.1 - AN OBJECT'S ISA POINTER

This peek behind the object facade gives a highly simplified view of what happens in the Objective-C runtime to enable message-dispatch, inheritance, and other facets of general object behavior. But this information is essential to understanding the major strength of Objective-C, its dynamism.

3.1.2 THE DYNAMISM OF OBJECTIVE-C

Objective-C is a very dynamic language. Its dynamism frees a program from compile-time and link-time constraints and shifts much of the responsibility for symbol resolution to runtime, when the user is in control. Objective-C is more dynamic than other programming languages because its dynamism springs from three sources:

- **Dynamic typing**—determining the class of an object at runtime
- **Dynamic binding**—determining the method to invoke at runtime
- **Dynamic loading**—adding new modules to a program at runtime

For dynamic typing, Objective-C introduces the `id` data type, which can represent any Cocoa object. A typical use of this generic object type is shown in the following code fragment:

```
id word;

while (word = [enm nextObject]) {
// do something with 'word' variable...
```

```
}
```

LISTING 3.1 - EXAMPLE OF USING ID DATA TYPE

The `id` data type makes it possible to substitute any type of object at runtime. The programmer can thereby let runtime factors dictate what kind of object has to be used in code. Dynamic typing permits associations between objects to be determined at runtime rather than forcing them to be encoded in a static design. Static type checking at compile time may ensure stricter data integrity, but in exchange for that stricter integrity, dynamic typing gives our program much greater flexibility. And through object introspection (for example, asking a dynamically typed, anonymous object what its class is) we can still verify the type of an object at runtime and thus validate its suitability for a particular operation. (Of course, we can always statically check the types of objects when we need to.)

Dynamic typing gives substance to dynamic binding, the second kind of dynamism in Objective-C. Just as dynamic typing defers the resolution of an object's class membership until runtime, dynamic binding defers the decision of which method to invoke until runtime. Method invocations are not bound to code during compilation; they are bound only when a message is actually delivered. With both dynamic typing and dynamic binding, we can obtain different results in our code each time we execute it. Runtime factors determine which receiver is chosen and which method is invoked.

The runtime's message-dispatch machinery enables dynamic binding. When the programmer sends a message[3] to a dynamically typed object, the runtime system uses the receiver's isa pointer to locate the object's class, and from there the method implementation to invoke. The method is dynamically bound to the message. And the programmer does not have to do anything special in Objective-C code to reap the benefits of dynamic binding. It happens routinely and transparently every time he sends a message, especially one to a dynamically typed object.

Dynamic loading, the final type of dynamism, is a feature of Cocoa that depends on Objective-C for runtime support. With dynamic loading, a Cocoa program can load executable code and resources as they're needed instead of having to load all program components at launch time. The executable code (which is linked prior to loading) often contains new classes that become integrated into the runtime image of the program. Both code and localized resources (including nib files) are packaged in bundles and are explicitly loaded with methods defined in Foundation's NSBundle class.

This "lazy-loading" of program code and resources improves overall performance by placing lower memory demands on the system. Even more importantly, dynamic loading makes applications extensible. We can devise a plug-in architecture for our application that allows us and other developers to customize it with additional modules that the application can dynamically load months or even years after the application is released. If the design is right, the classes in these modules will not clash with the classes already in place

[3] To get an object to do something, the programmer sends it a message telling it to apply a method

because each class encapsulates its implementation and has its own namespace.

3.2 THE MODEL-VIEW-CONTROLLER DESIGN PATTERN

Model–view–controller (MVC) is a software architecture, currently considered an architectural pattern used in software engineering. The pattern isolates "domain logic" (the application logic for the user) from the user interface (input and presentation), permitting independent development, testing and maintenance of each. Model View Controller (MVC) pattern creates applications that separate the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements (Model-View-Controller). It is not the only design pattern exploited in the development of the project, but it is for sure the most important one: first of all because it gives a guide in building an application from scratch; the other patterns used here and by Apple developers, such as Key Value Coding, Key Value Observing, delegation and so on, are not as essential as this one in developing an application that must be compliant to the Apple standards. We remark that using MVC is not mandatory in the construction of an application, but it strongly recommended for the reasons seen above and for the arguments that we are going to see in the next paragraphs.

3.2.1 OVERVIEW

The Model-View-Controller design pattern is simply stated (Bucanek, 2009):

- Data model objects encapsulate information
- View objects display information to the user.
- Controller objects implement actions.
- View objects observe data model objects and update their display whenever it changes.
- View objects gather user input and pass it to a controller object that performs the action.

The key to successfully implementing an MVC design is to pay close attention to the role of objects and the communications between those objects. The first three rules define the role of our objects.

- Data model objects store, encapsulate, and abstract data. They should not contain methods or logic specific to making the application function. For example, a data model class should implement a method that serializes its data, but it should not implement the “Save As...” command. Even if the two functions are nearly identical, the code that deals with the abstract data transformation should be implemented in the data model object and the code that implements the user command should be implemented in the controller object.

- View objects display the information in the data model to the user. View class design runs from the extremely generic to the very specific; generic view classes are provided by the framework to display almost any kind of string, number, or image, while it is more likely to implement very specific view objects designed for the application. View objects also interact with the user and initiate actions by interpreting user-initiated events, such as mouse movement and keystrokes. It converts those events into actions that are passed to the controller object for execution. The event and resulting action are often very simple; clicking the mouse over a button object will send an action message[4] to a controller. Complex gestures, like drag-and- drop, are more involved.
- Controller objects implement application's actions. Actions are usually initiated by view objects in response to user events.

The other aspect of the Model-View-Controller design is the communication between the data model, controller, and view objects. The fundamentals communication paths are illustrated in Figure 2.

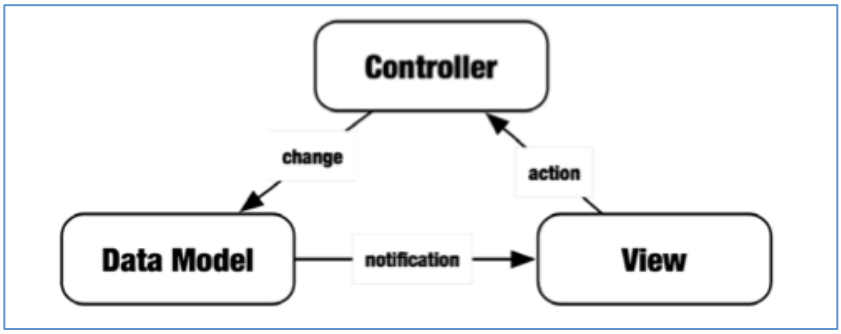

FIGURE 3.2 - FUNDAMENTAL MVC COMMUNICATIONS

When the user does something, like typing a keyboard shortcut, a view object interprets it and sends an action message to the controller object. The controller object performs the action, which often involves sending messages to the data model object. When the data model changes, it notifies its observers by sending messages to the view object, which updates the display.

3.2.2 MVC VARIATIONS

There's a lot of latitude to the basic MVC model. Roles can blend or use alternate communication paths. The next few sections describe the significant variants.

COMBINED CONTROLLER AND DATA MODEL

[4] For more details about sending messages, see note 13

The controller and data model may be the same object, as shown in Figure 3. This is often the case when the data model is trivial. Technically, a single integer is a data model, but it would be a waste of our programming talent to create a data model class just to encapsulate one number. Instead, the value is stored in the controller and attached to a view object for display.

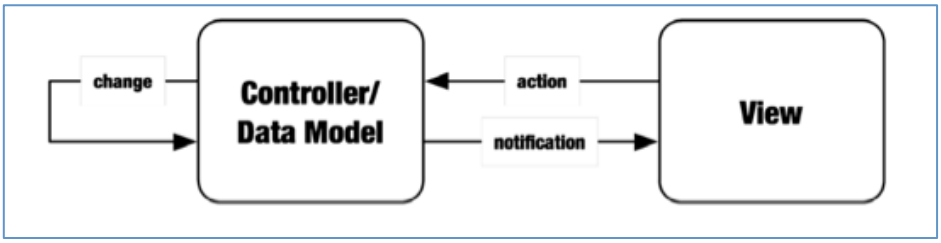

FIGURE 3.3 - COMBINED CONTROLLER AND DATA MODEL

This limits the modularity of the design. However, unless the data model and controller are inappropriately entangled, it should be easy to refactor the application into separate controller and data model classes in the future, without disrupting our design. The key is to keep the *concept* of the data model independent of the controller, even while they occupy the same object.

MEDIATING CONTROLLER

A popular variation of the Model-View-Controller design pattern is the mediated MVC pattern, as shown in Figure 4. This pattern appears repeatedly in the Cocoa framework[5].

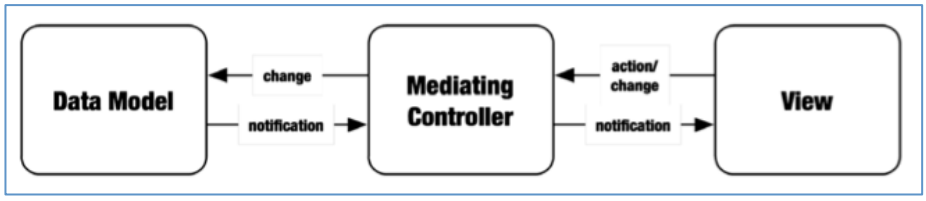

FIGURE 3.4 - MEDIATED MODEL-VIEW-CONTROLLER DESIGN PATTERN

In a mediated MVC design, the controller also acts as the data model for the view. It is a data model proxy. The mediating controller passes data model messages to the actual data model, and forwards any change notifications from the data model on to its observers.

This design is particularly well suited to database applications. The data model object encapsulates the raw data. It takes responsibility for fetching the data from a persistent store, caching it, and performing any alterations to it. The controller object performs it normal role, but also presents the data, as it will appear in the application, to the view objects. This might involve filtering, sorting, and transforming the data. How the data is

[5] for more details about Cocoa see paragraph 3.3

sorted, the user's current selection in the table, and so on, are specific to the application, not the data. From the view object's perspective, the controller is the data model. The view displays the sorted, filtered, and transformed data presented to it by the controller, and has no direct knowledge of the underlying data model.

DIRECT VIEW AND DATA MODEL BINDING

Often, there's no "action" per se associated with a change to a displayed value. If the view object is displaying a simple value (a number, string, or Boolean), it can communicate changes directly to the data model object, as shown in Figure 5.

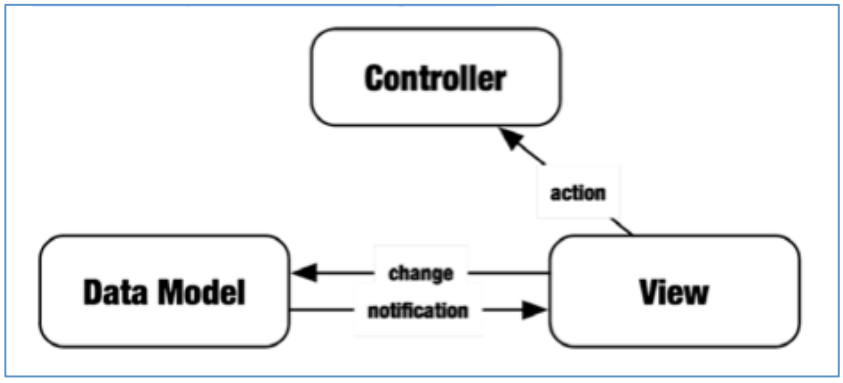

FIGURE 3.5 - DIRECT DATA MODEL AND VIEW BINDING

The relationship between the data model property and the view is described as a binding. The view object displays the value and updates its display whenever the value changes (via observing). If the user edits the displayed value, the view object communicates that to the data model by setting the new value directly. There are some other common variations to the basic MVC design pattern but these where the most important ones.

3.2.3 ADVANTAGES OF MVC

At first blush, it seems like MVC creates a lot of additional work for the programmer, taking what could be a single object and breaking it up into multiple objects with complex communications. In a very small set of circumstances, this would be right, but most applications are not simple, or do not remain simple. MVC is, in many respects, like object-oriented programming itself. It's a discipline that occasionally creates more work, but more often allows for the design of large, elegant, flexible, and sophisticated applications that are both comprehensible and well behaved.
The following sections highlight some of the significant advantages of employing the Model- View-Controller design pattern.

MODULARITY

The Model-View-Controller design pattern is an extension of the computer science principles of *separation of concern* and *encapsulation*. It encourages the programmer to identify the roles that the application plays and compartmentalize those roles into distinct objects.
These principles improve the MVC design for all the same reasons they improve object- oriented programming in general. They localize interrelated functionality into containers with identifiable boundaries. Changes to a class tend to be localized. When the change affects other classes, it's easier to analyze how they will be affected since the interface to the changed class is well defined. They can later be subclassed, replaced, or reused with little or no impact on the rest of the design.

FLEXIBILITY

One of the greatest strengths of the Model-View-Controller design pattern is its flexibility. By separating the data model and controller concerns from the display view, the two can be mixed and matched at will. MVC's most powerful feature is probably the ability to effortlessly replace view objects, or use multiple view objects, without changing the data model or controller.

REUSE

Reuse is closely related to flexibility. By abstracting the functionality of the classes, the programmer can reuse objects in other applications or for other purposes. The most commonly reused objects are data model and view objects. For example the core Cocoa view objects (buttons, text fields, image viewers) are reused daily for thousands of different purposes, just by connecting them to different data model and controller objects.

3.3 COCOA ENVIRONMENT

Cocoa is an application environment for both the Mac OS X operating system and iOS, the operating system used on Multi-Touch devices such as iPhone, iPad, and iPod touch. It consists of a suite of object-oriented software libraries, a runtime system, and an integrated development environment (Cocoa Fundamentals Guide). More specifically it is a set of object-oriented frameworks that provides a runtime environment for applications running in Mac OS X and iOS.
In this paragraph and in the following one we will discuss about Cocoa and Cocoa Touch, the two layers (seen as sets of frameworks) used for developing, respectively, AppKid and the app generator. In addition we will explain how they fit in the software architectures of Mac OS X and iOS.

3.3.1 MAC OS X AT A GLANCE

Mac OS X is organized as a set of layers. At the lower layers of the system are the fundamental services on which all software relies. Subsequent layers contain more sophisticated services and technologies that build on (or

complement) the layers below. Figure 6 provides a graphical view of this layered approach (Mac OS X Technology Overview).

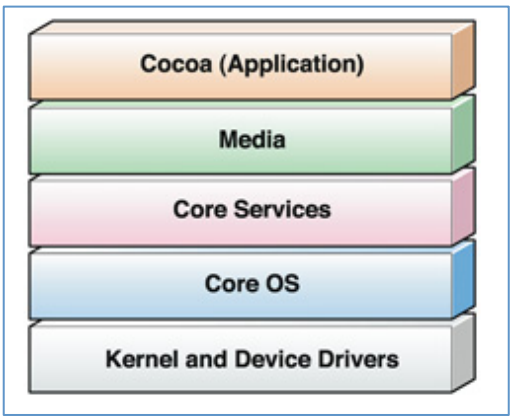

FIGURE 3.6 - THE LAYERS OF MAC OS X

The lower the layer a technology is in, the more specialized are the services it provides. Generally, technologies in higher layers incorporate lower-level technologies to provide common app behaviors. When developing a Cocoa app, the developer should always use the highest-level programming interface that meets the goals of the app. The layers (starting from the top) are:

- The Cocoa application layer incorporates technologies for building an app's user interface, for responding to user events, and for managing app behavior.
- The Media layer implements specialized technologies for playing, recording, and editing audiovisual media and for rendering and animating 2D and 3D graphics.
- The Core Services layer contains several core services and technologies, including collection management, data formatting, memory management, string manipulation, process management, data-model management, XML parsing, stream-based I/O, low-level network communication, and many others.
- The Core OS layer defines programming interfaces that are related to hardware and networking. It provides services for accessing information about network users and resources, for determining the reachability of remote hosts, and for enabling access to devices and services on a network. It also publishes interfaces for running high-performance computation tasks on a computer's CPU and GPU.

- The Kernel and Device Drivers layer, also known as Darwin[6], consists of the Mach kernel environment, device drivers, BSD library functions, and other low-level components. The layer includes support for file

[6] The kernel, along with other core parts of Mac OS X are collectively referred to as Darwin. Darwin is a complete operating system based on many of the same technologies that underlie Mac OS X. However, Darwin does not include Apple's proprietary graphics or applications layers, such as Quartz, QuickTime, Cocoa, Carbon, or OpenGL (Kernel Programming Guide)

systems, networking, security, interprocess communication, programming languages, device drivers, and extensions to the kernel.

3.3.2 HOW COCOA FITS INTO MAC OS X

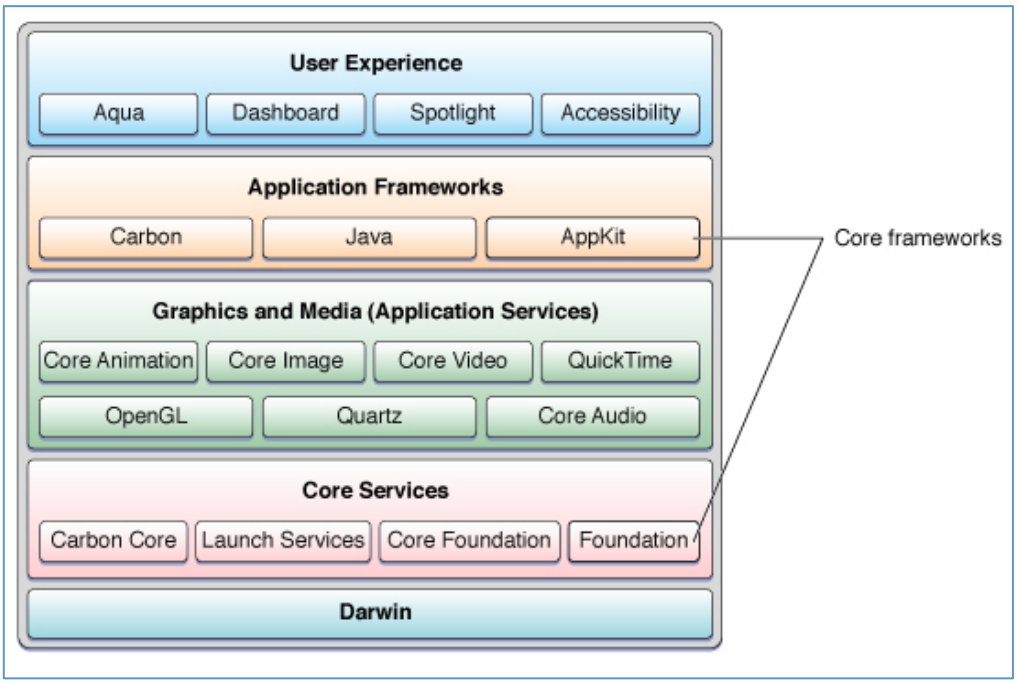

FIGURE 3.7 - COCOA IN THE ARCHITECTURE OF MAC OS X

In Mac OS X, Cocoa has two core Objective-C frameworks that are essential to application development for Mac OS X:

- **AppKit:** AppKit, one of the application frameworks, provides the objects an application displays in its user interface and defines the structure for application behavior, including event handling and drawing.
- **Foundation:** these framework, in the Core Services layer, defines the basic behavior of objects, establishes mechanisms for their management, and provides objects for primitive data types, collections, and operating-system services. Foundation is essentially an object-oriented version of the Core Foundation framework.

AppKit has close, direct dependences on Foundation, which functionally is in the Core Services layer. If we look closer, at individual, or groups, of Cocoa classes and at particular frameworks, we begin to see where Cocoa either has specific dependencies on other parts of Mac OS X or where it exposes underlying technology with its interfaces. Some major underlying frameworks on which Cocoa depends or which it exposes through its classes and methods

are Core Foundation, Carbon Core, Core Graphics (Quartz[7]), and Launch Services:

- Core Foundation: many classes of the Foundation framework are based on equivalent Core Foundation opaque types. This close relationship is what makes "toll-free bridging"—cast-conversion between compatible Core Foundation and Foundation types—possible. Some of the implementation of Core Foundation, in turn, is based on the BSD part of the Darwin layer.
- Carbon Core: AppKit and Foundation tap into the Carbon Core framework for some of the system services it provides. For example, Carbon Core has the File Manager, which Cocoa uses for conversions between various file-system representations.
- Core Graphics: the Cocoa drawing and imaging classes are (quite naturally) closely based on the Core Graphics framework, which implements Quartz and the window server.
- Launch Services: the NSWorkspace class exposes the underlying capabilities of Launch Services. Cocoa also uses the application-registration feature of Launch Services to get the icons associated with applications and documents.

Apple has carefully designed Cocoa so that some of its programmatic interfaces give access to the capabilities of underlying technologies that applications typically need. But if the programmer requires some capability that is not exposed through the programmatic interfaces of Cocoa, or if he needs some finer control of what happens in the application, he may be able to use an underlying framework directly. (A prime example is Core Graphics; by calling its functions or those of OpenGL, the code can draw more complex and nuanced images than is possible with the Cocoa drawing methods.) Fortunately, using these lower-level frameworks is not a problem because the programmatic interfaces of most dependent frameworks are written in standard ANSI C, of which Objective-C language is a superset.

3.4 COCOA TOUCH

Description of iOS deserves a separate discussion. In this paragraph we will see the architecture of iOS and the most important features of every layer. Cocoa Touch is the iOS version of Cocoa environment.

3.4.1 IOS AT A GLANCE

The iOS architecture is similar to the basic architecture found in Mac OS X. At the highest level, iOS acts as an intermediary between the underlying

[7] Quartz specifically refers to a pair of Mac OS X technologies, each part of the Core Graphics framework: Quartz 2D and Quartz Compositor. It includes both a 2D renderer in Core Graphics and the composition engine that sends instructions to the graphics card. Because of this vertical nature, *Quartz* is often interchanged synonymously with *Core Graphics*.

hardware and the applications that appear on the screen (iOS Technology Overview). The implementation of iOS technologies can be viewed as a set of layers. At the lower layers of the system are the fundamental services and technologies on which all applications rely; higher-level layers contain more sophisticated services and technologies.

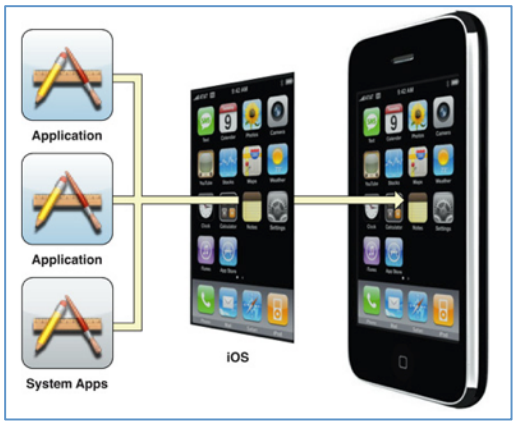

FIGURE 3.8 - APPLICATIONS LAYERED ON TOP OF IOS

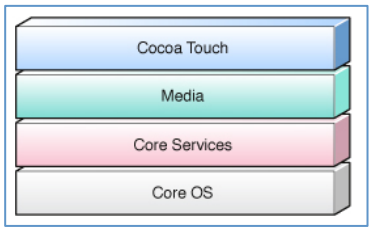

FIGURE 3.9 - LAYERS OF IOS

3.4.2 HOW COCOA TOUCH FITS INTO IOS

The application-framework layer of iOS is called Cocoa Touch. Although the iOS infrastructure on which Cocoa Touch depends is similar to that for Cocoa in Mac OS X, there are some significant differences. We can see that the iOS diagram (figure 10) also shows the software supporting its platform as a series of layers going from a Core OS foundation to a set of application frameworks, the most critical (for applications) being the UIKit framework. As in the Mac OS X diagram (figure 7), the iOS diagram has middle layers consisting of core-services frameworks and graphics and media frameworks and libraries. Here also, a component at one layer often has dependencies on the layer beneath it.

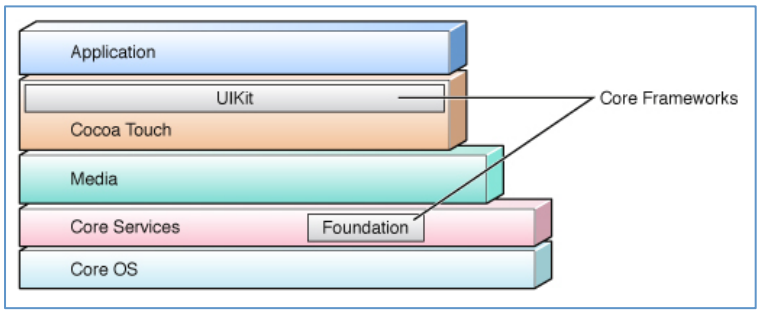

FIGURE 3.10 - COCOA IN THE ARCHITECTURE OF IOS

Generally, the system libraries and frameworks of iOS that ultimately support UIKit are a subset of the libraries and frameworks in Mac OS X. For example, there is no Carbon application environment in iOS, there is no command-line access (the BSD environment in Darwin), and QuickTime is absent from the platform. However, because of the nature of the devices supported by iOS, there are some frameworks, both public and private, that are specific to iOS.
The following summarizes some of the frameworks found at each layer of the iOS stack, starting from the foundation layer.

- **Core OS:** This level contains the kernel, the file system, networking infrastructure, security, power management, and a number of device drivers. It also has the libSystem library, which supports the POSIX/BSD 4.4/C99 API specifications and includes system-level APIs for many services.
- **Core Services:** The frameworks in this layer provide core services, such as string manipulation, collection management, networking, URL utilities, contact management, and preferences. They also provide services based on hardware features of a device, such as the GPS, compass, accelerometer, and gyroscope. Examples of frameworks in this layer are Core Location, Core Motion, and System Configuration. This layer includes both Foundation and Core Foundation, frameworks that provide abstractions for common data types such as strings and collections. The Core Frameworks layer also contains Core Data, a framework for object graph management and object persistence.
- **Media:** The frameworks and services in this layer depend on the Core Services layer and provide graphical and multimedia services to the Cocoa Touch layer. They include Core Graphics, Core Text, OpenGL ES, Core Animation, AVFoundation, Core Audio, and video playback.
- **Cocoa Touch:** The frameworks in this layer directly support applications based in iOS. They include frameworks such as Game Kit, Map Kit, and iAd.

The Cocoa Touch layer and the Core Services layer each has an Objective-C framework that is especially important for developing applications for iOS. These are the core Cocoa frameworks in iOS:

- **UIKit:** This framework provides the objects an application displays in its user interface and defines the structure for application behaviour, including event handling and drawing. For a description of UIKit, see "UIKit (iOS)."
- **Foundation:** This framework defines the basic behaviour of objects, establishes mechanisms for their management, and provides objects for primitive data types, collections, and operating-system services. Foundation is essentially an object-oriented version of the Core Foundation framework.

As with Cocoa in Mac OS X[8], the programmatic interfaces of Cocoa in iOS give our applications access to the capabilities of underlying technologies. Usually there is a Foundation or UIKit method or function that can tap into a lower-level framework to do what we want. But, as with Cocoa in Mac OS X, if the programmer requires some capability that is not exposed through a Cocoa API, or if he needs some finer control of what happens in the application, he may choose to use an underlying framework directly. For example, UIKit uses the WebKit to draw text and publishes some methods for drawing text; however, we may decide to use Core Text to draw text because that gives us the control we need for text layout and font management. Again, using these lower-level frameworks is not a problem because the programmatic interfaces of most dependent frameworks are written in standard ANSI C, of which the Objective-C language is a superset (Cocoa Fundamentals Guide).

3.5 SPARROW FRAMEWORK

Sparrow is a pure Objective C library targeted on making game development as easy and hassle-free as possible. Sparrow makes it possible to write fast OpenGL applications without having to touch OpenGL or ppure C (but easily allowing to do so, for those who wish). It uses a tried and tested API that is easy to use and hard to misuse (Sperl). It has been designed to be easy to learn for Java, ActionScript or .Net developers. It has been developed in parallel with two real life iPhone games, offering the possibility to optimize it depending on the game development needs. The result is a high performance framework hiding inside it many of the programming troubles given by a direct access to the UIKit and Foundation frameworks.

FIGURE 3.11 - THE SPARROW LOGO

[8] We use "Cocoa" generically when referring to things that are common between the platforms. When it is necessary to say something specific about Cocoa on a given platform, we use a phrase such as "Cocoa in Mac OS X."

4 ANALYSIS PHASE: PROJECT GOALS

In this chapter we start describing the first book developed by the company and published on the App Store. After that we analyse the problems of the architecture of such an application, explaining why we needed to use a framework instead of building a different complete applications for every printed book.

4.1 CHISSÀ: THE FIRST INTERACTIVE BOOK

In this paragraph we describe the design and implementation of the first generation of interactive books, taking as example "Chissà", one of the first Italian interactive books for sale on the App Store.

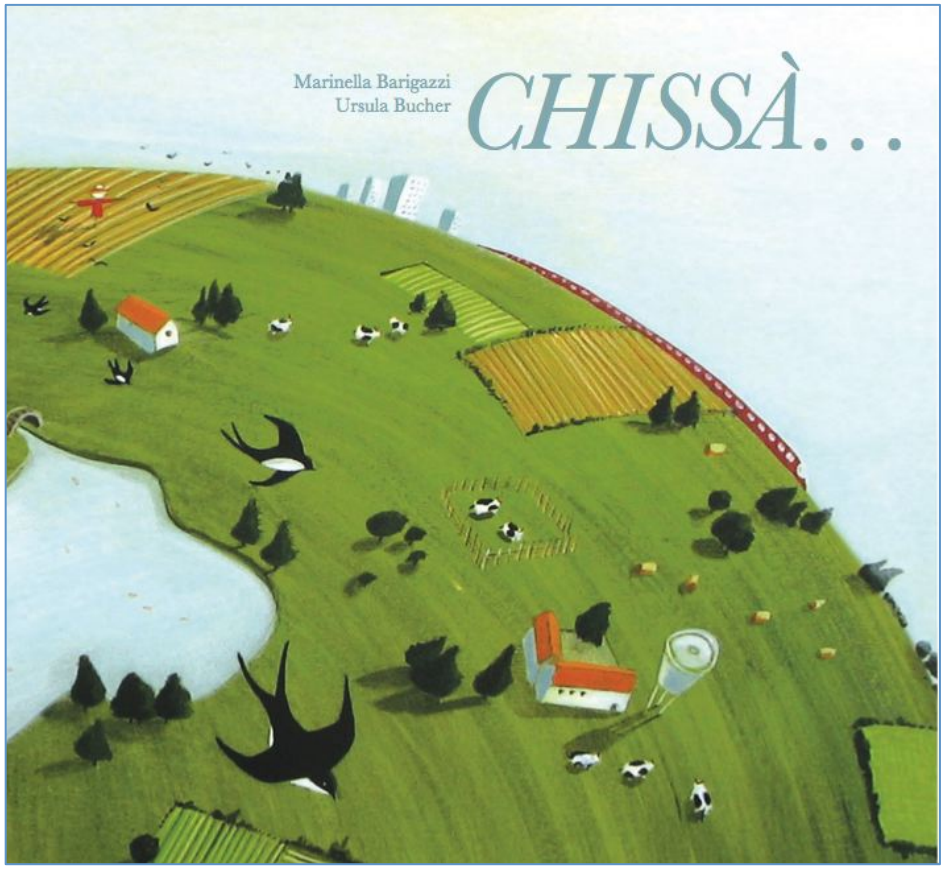

FIGURE 4.1 - THE ORIGINAL PRINTED BOOK COVER OF CHISSÀ

The original "Chissà" is a book of about ten pages, illustrated with hand made pictures and with some text for every page.

4.2 STRUCTURE OF THE APPLICATION

The application mixes the usage of UIKit and Foundation frameworks with Sparrow. This has been necessary because the application makes available some parts that would have been hard to develop entirely with Sparrow, such as the “Games” section and the “Record” section. By the way, in the next paragraphs we will explain the behaviour of the application and its internal structure.

4.2.1 BEHAVIOUR OF THE APPLICATION

After tapping on its icon on the iPad, the application starts with a short animation that presents the Ware’s Me and Altera logo and after that the main menu starts. There are five buttons, and each one permits to access to a different section:

FIGURE 4.2 - INITIAL MENU

1. **READ:** tapping on it starts the interactive book and brings us to the title page; a “next” and a “previous” button lets the user go to the next or previous page. There are also buttons to return to the main menu and to the index page;

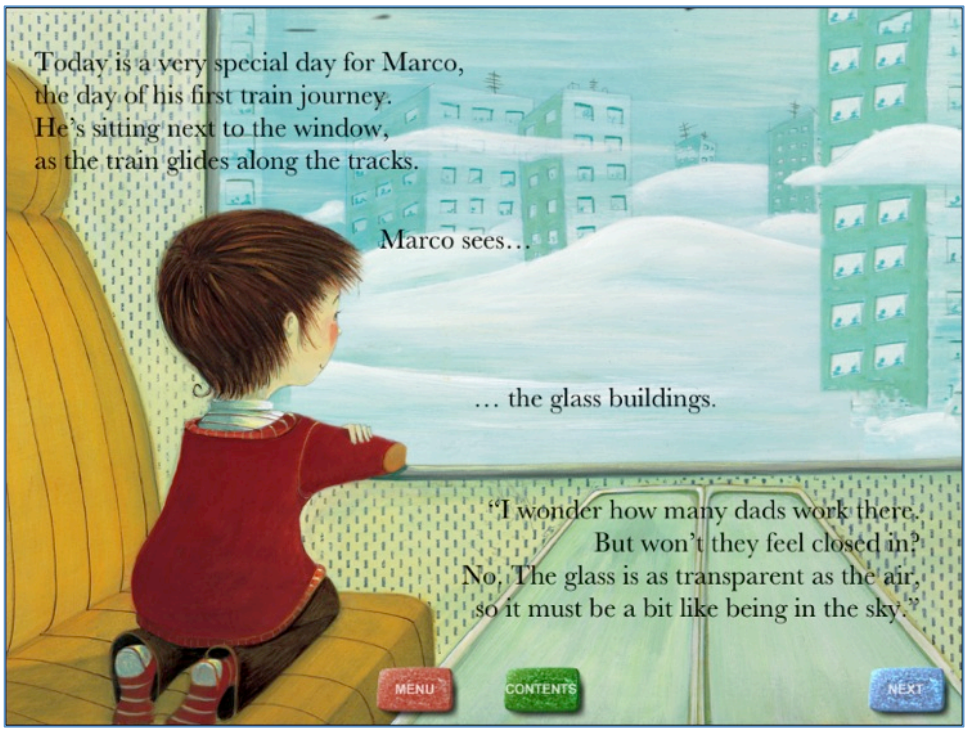

FIGURE 4.3 - FIRST PAGE

2. **CONTENTS:** tapping on this button brings us to the index page;

FIGURE 4.4 - INDEX

3. **PLAY:** this button lets the user start to play; there are two games to play with: the first one is a puzzle with images taken from the book,

and the second is a drawing without colours, that the user has to fill with his fingers.

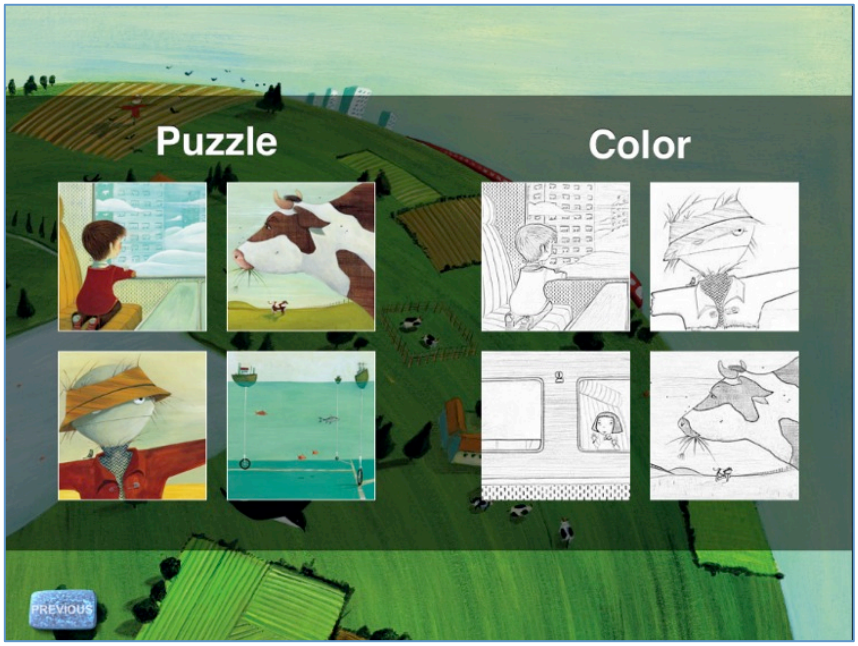

FIGURE 4.5 - GAME MENU

4. **OPTIONS:** this section permits to activate the recording mode on each page, to enable recorded voice instead of the default application voice, and lets switch from Italian language to English.

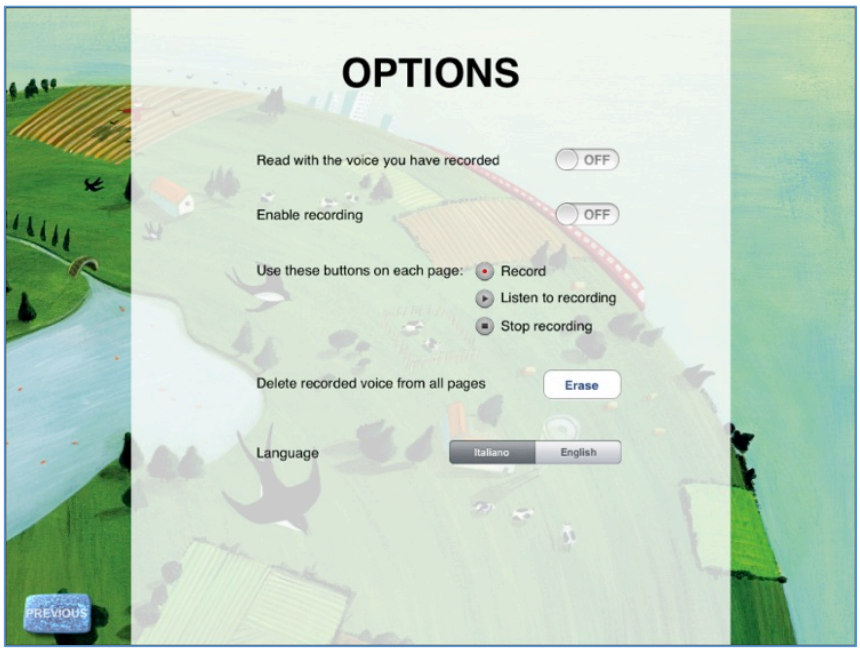

FIGURE 4.6 - OPTIONS MENU

5. **CREDITS:** this section contains credits and contacts.

FIGURE 4.7 - CREDITS PAGE

4.2.2 ARCHITECTURE

As already said the application is a mix of Cocoa frameworks and Sparrow: in fact the launching phase is the same of a normal Cocoa Touch application and the first initializations are done in the view delegate methods in the application delegate class[9]. By the way, we report here the classes of the Model, of the View and of the Controller:

- **MODEL:**

CLASS	SUBCLASS OF
PageSprite	SPSprite
Page1Sprite	PageSprite
Page3Sprite	PageSprite

[9] The application delegate is the most important class used at application launching. It deals with controlling the main window and application initializing.

…	…
Page12Sprite	PageSprite

Each of these classes is an extension of the Sparrow class SPSprite and it is used to group elements of a particular page. For example, if page *i* contains two images and a sound, they will be embedded in class Page*i*Sprite (more specifically, in the `init` method of the class there will be code to load images and the sound). The first class (PageSprite) is the base class that extends SPSprite, and all the others, in turn, inherit from it. This solution was adopted because every page has some text and sounds. Therefore, the base class is responsible for loading texts and sounds (from a very simple and basic plist [10] file). Other classes extend the base behaviour adding the particular objects that we want to show in the related page. Note that some pages are made only of texts, sounds and of a background image, therefore it was not necessary extending the base class PageSprite to represent them. This is why in the table above we did not enumerate completely the sprites from the first to the last.

- **VIEW:**
  - IntroView
  - MenuView
  - ReadView
  - ButtonsView
  - IndexView
  - SettingsView
  - AboutView
  - GameView
  - JigsawView
  - PaintView
  - SparrowView

  These classes are the xib files constructed in the Interface Builder, for each of which there is a Controller. The last one is a subclass of SPView, a Sparrow object used as a bridge from Cocoa to Sparrow: inside SPViews it is possible to render Sparrow objects like images, text, and stuff like that.

- **CONTROLLER:**

[10] For more details about property lists, see Appendix.

- **IntroViewController**: it deals with showing an index and bringing the user to the right page when he taps on a particular miniature;
- **MenuViewControlle**r: it's the main menu controller, and it deals with pushing other controllers on its navigation controller;
- **ReadViewController**: it is responsible for starting the first page of the book, containing the title. The first page does not use Sparrow. The "Next" button of this page is bound to an action, which adds an instace of SparrowView class in the ReadView. SparrowView is a subclass of SPView, the base Sparrow view class used to render the contents of a page[11]. SPView and the Game class are instantiated in the application delegate class.
- **ButtonsViewController:** this controller is responsible for changing pages of the book; more specifically, it notifies the stage[12] telling it to update the screen with the next or previous page. It also controls recording buttons.
- **IndexViewController:** it controls the index and allocates the right pageSprite when the user wants to go to a specific page.
- **SettingsViewController:** it controls the options menu.
- **AboutViewController:** it is the controller of the credits view. It does nothing but popping itself from the navigation controller when clicking on the "previous" button.
- **GameViewController:** it deals with showing the game menu, it lets the user choose which game he wants to play with and it starts the chosen one.
- **JigsawViewController**: it deals with the puzzle game; it acts in synergy with PieceView, BackgroundView and JigSawHandler as a single MVC.
- **PaintViewController:** controller used to manage the paint game
- **Game:** this class is an extension of the SPStage Sparrow class, and it used as the container of Sparrow objects, such as images, texts and everything that has to be rendered on the screen through Sparrow. It receives notifications[13] that tell it to start i-th page and it uses the model part of the application to load the proper page. The Game class is not a real controller, in the sense that it does not extend the UIViewController base class, and it does not use mechanisms used in a classical controller. Nevertheless it behaves like a controller for the SparrowView.

---

[11] For a more detailed explanation of Sparrow, please visit http://wiki.sparrow-framework.org/start

[12] In order to discover what is the stage, see "Game" class in this list.

[13] For more details about notifications, see Appendix ("Communicating with objects in Cocoa" paragraph).

- **BBChissàObjectStore:** it is a singleton object that keeps the current page numbers synchronized with the ButtonsViewController and the Game class.

All the classes except the last two are quite ordinary controller classes. They embed view control logic, like implementations of actions, setting up of the views, and communicate with the underlying model. Communications between the three architecture components (Model, View and Controller) are done via notifications or via the actions scheme.

In the following page we can see a graphical representation of the architecture of the application.

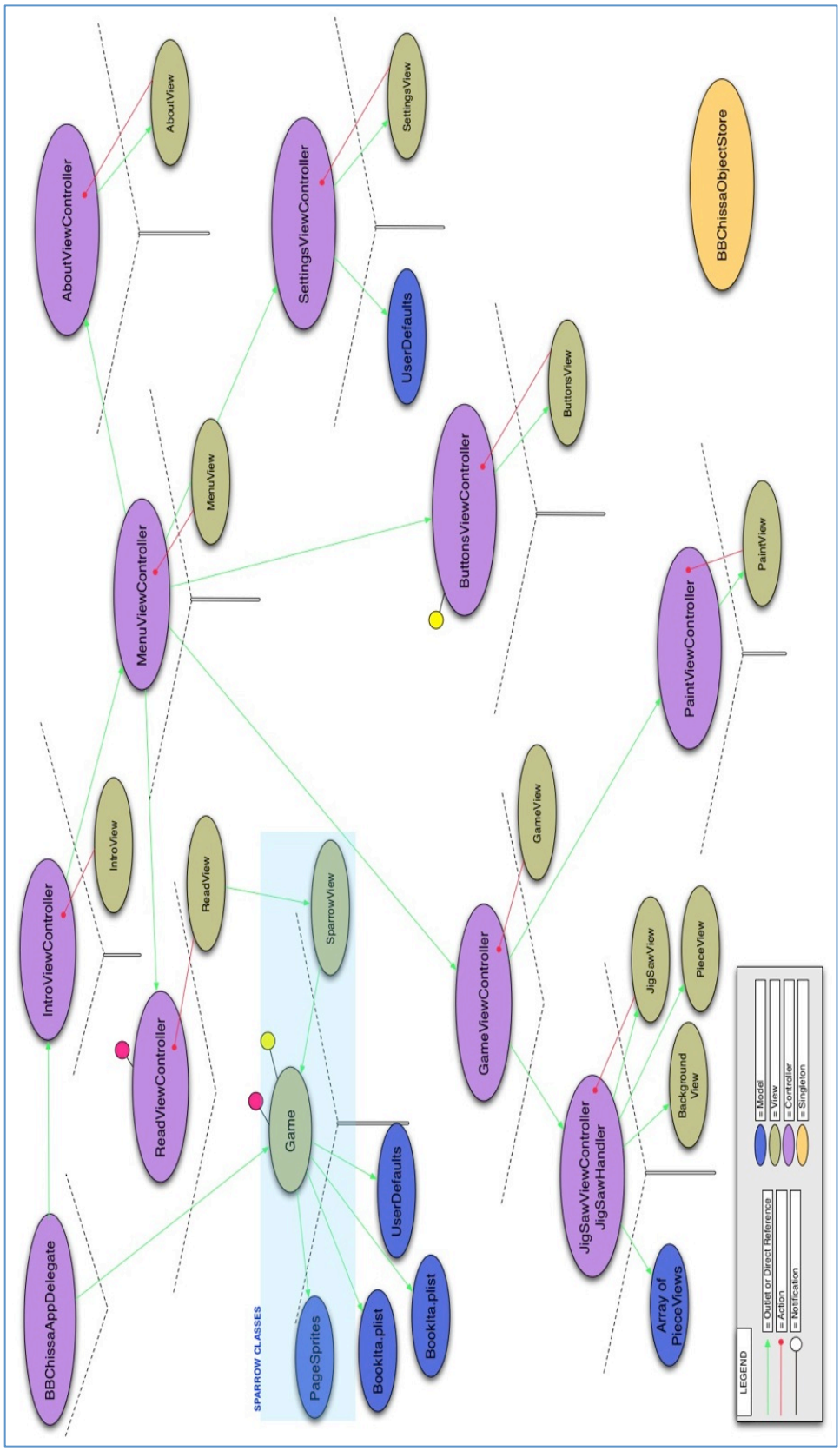

FIGURE 4.8 - MVC ARCHITECTURE OF CHISSÀ

4.3 WHY A SOFTWARE FRAMEWORK?

In the analysis phase Altera developers found two possible patterns to develop the application:

1. Hardcoding[14] every feature of the book (as in "Chissà").
2. Constructing a framework to make common functions, algorithms and data structures reusable when developing other books and using an external file to represent features.

It soon became clear that the first solution would have been faster to realize at the beginning, but it would not have been scalable. In fact, embedding every feature of the book in code is easy to do at the beginning, but it does not let the programmer reuse the code written in the previous book for the next one. The idea this first solution is based on is to design a class for every page of the book. As we have seen in the last paragraph, each class embeds data about almost every feature of the page, such as texts, sounds, images, animations, and information about the triggering of an element by another one (in reality we have seen that text and sounds are loaded from a basic plist, but the very first concept did not provide this feature). It becomes clear that this kind of code is definitely not reusable: every page of a book is different from another one, and so are the classes that codify the pages. In this solution every book is a new application, and, building a new one, results in a new design phase of every page, followed by rewriting them from scratch. Obviously this solution is fast for the first book, but it does not permit to exploit the common aspects of every interactive book to build the following ones in a faster way. In a long lasting perspective this solution becomes a total failure.

The second solution uses a different approach: it starts from the fact that every interactive book has common features:

- They are all made of texts, images, and sounds.
- The animation system is always the same.
- Objects can be dragged around the screen in every book.

Taking account of these common features, we can conclude it is possible to construct a set of reusable classes representing the application: the most important idea is to use a file to represent text, images, sounds and so on. In this way, the application itself will not embed data anymore, but it will load information about the book from this file. Thus, we can also construct a framework that will contain the needed set of functions, data structures and logic to read data in the file and to show contents of the interactive book. This way of designing the application has a big advantage: when the program will

[14] Hard coding (also, hard-coding or hardcoding) refers to the software development practice of embedding what may, perhaps only in retrospect, be regarded as input or configuration data directly into the source code of a program or other executable object, or fixed formatting of the data, instead of obtaining that data from external sources or generating data or formatting in the program itself with the given input (Hard-coding).

be completed, modifying the data file to create a new book from scratch will be enough, saving a very large amount of time and effort. This kind of design pattern has also an important drawback: the amount of work and the difficulty level to develop the application and the framework used by the application are much larger than the time to develop a book with the first pattern. The framework needs to be as much general as possible and it has to embrace every kind of situation that can occur in the design of an interactive book. For example, while with the first pattern is easy to program the starting of a text after a sound (roughly speaking, an event listener on the sound object is enough), this feature becomes hard to represent in a data file; moreover, the application needs to be able to read from the file that text *x* has to start after sound *y*. We will see in chapter four how these problems are solved, but right now we can appreciate that there is a major difference between the two approaches.

Being aware of all of these considerations, Altera team decided to choose the first approach: the necessity to produce a book in about three weeks[15] forced them to take the ”easier but not scalable” way (at least for the first phase of release of the books). They also decided that a framework would have to be developed. Implementing an application that, using the framework, would reproduce Chissà in the same manner of the first generation of applications was the final goal. This would have made possible to manage most of the features of interactive books. More specifically, we decided that the general capabilities that the framework should have provided would have been:

- Inserting images and texts at a particular position, with a specific width, height and degree of transparency.
- Inserting sounds.
- Inserting movies.
- Animating images and texts.
- Triggering whatsoever element by some other one.
- Specifying a transition from a page to the following.
- Make some objects draggable.
- Make some objects react when tapped by the user.
- Letting the user play with the puzzle or the filling game presented in 4.2.1.
- Showing credits to the user.
- Choosing the language of the book.
- Letting the user record his voice and listen to it.

[15] Before publishing the books on the App Store they developed it to show it to a technology fair.

5 SOFTWARE FRAMEWORK

In this chapter we will explain the framework created to make the development of an interactive book easier and we will talk about the application that uses the framework. We start describing the design of the tasks of the application using the framework, then we go on describing every phase of the design of the framework, from the simplest features to the most complicated, and then we conclude describing the coding solutions used to make the framework and the application as fast as possible.

5.1 THE APPLICATION AND THE FRAMEWORK DESIGN

The first topic was finding a pattern to build books in a faster way. We started from the idea that every interactive book has aspects in common: for example all of them show images, texts, they play sounds and so on. Therefore, we started thinking that these common features could have been represented in a structured way by using a markup language like XML. Thanks to XML language we can store data and give data a meaning at the same time. For example, if we want to describe the position of an element, we can store the coordinates of the position in a file of the element and make them meaningful for the application by inserting appropriate tags. Thus, it became clear that we could represent the elements of a page with an XML file. Therefore, there will be an XML file for each book, and we will have to group some common classes together in order to read, interpret and use the elements represented with the markup language. This set of classes will make up the so-called framework. We designed the framework in order to be used by the particular application that will implement the current interactive book.

The next problem was to distinguish what should have been done by the framework, and what by the application. The main idea was to delegate to the framework all the common aspects of the several interactive books and to the application the particular aspects of them. When creating a new book, we only need to customize the application, because the framework will have already implemented the general aspects of it; in my opinion this is the most powerful aspect of the strategy. In addition, the XML file will specify the appearance and the behaviour of the scenes of the book. Therefore, connecting the new application to the framework and using its classes will be enough to create a new interactive book. The framework will be used to read the description of the book from the new XML file constructed ad hoc, using the editor or by simply editing its XML structure.

The following problem was to define the general aspects of every interactive book. Note that this strategy creates some constraints for what we can represent with the framework, and therefore for what we can represent in a fast way (i.e. without writing objective-c code). These constraints are strictly related to the design of the XML file. Think about inserting a triangle shape in a page: we have to create a mechanism to represent it in the XML file and a way to interpret it the framework. Therefore, to extend the power of the XML file, we should extend the capabilities of the framework at the same time.

We decided that the data file should have represented all aspects presented in 4.3, and the framework should have been able to read, interpret and transfer them to the screen of the iPad, delegating to the application the implementation of all the other custom features of the particular interactive book. In the next paragraph we will show the design of the representation of the features.

5.2 THE DATA MODEL DESIGN

The design of the data model of the framework, i.e. the way to represent data the application will load, was the most important aspect considered at the beginning. "Data" in this context means everything needed to represent on the screen the right behaviour of the book; in this sense we do not only mean, for example, the images and texts of a page, but also the order in which these objects appear on the screen and with which timing they remain on the screen. In such a kind of context, the problem is to find a meaningful way to represent data that will be used by the over standing application, and at the same time to create a scalable system (i.e. a system that can be extended in order to represent some new features in the future).

The first problem was solved using property lists[16]. Although this kind of representation seems to be very restrictive, it can in fact represent a very large variety of objects, and it is general enough for our purpose. Property lists can be easily stored in the file system using an XML representation, and they are often used when amount of data does not exceed a few hundreds of kilobytes. Therefore, they are the most proper technology to represent and store the application data. The idea using a property list is based on is representing the book as an array of pages, where each page is a dictionary. Every key-value pair of the dictionary has to represent a property of the page. Thus, there must be an entry that lists the elements of the page (see figure 14). In this case we decided to identify the property with the string *"elements"* (the key of the entry) and the corresponding value with an array. Obviously, the array has to contain the description of each element of the page. In this phase we decided to start differentiating the simpler objects in the scene of a page; the candidates where:

- Images
- Texts
- Sounds
- Movies
- Objects animations

[16] For a description of property lists, see Appendix.

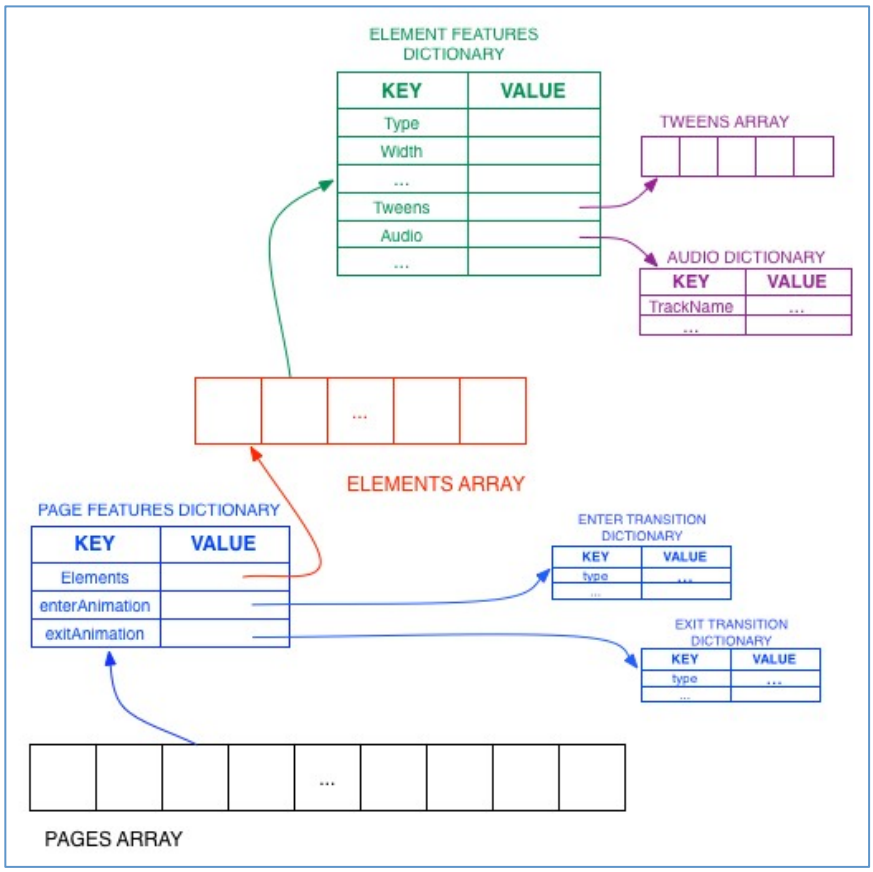

FIGURE 5.1 - GRAPHICAL REPRESENTATION OF THE DATA MODEL

We also decided to describe if a page has to appear or disappear with some animation; thus, we inserted two other keys in the page features dictionary:

- *enterAnimation*: the correspondent value is a dictionary that describes the animation with which the page appears:

KEY	VALUE	VALUE TYPE
type	interpolation type	string
duration	duration of the animation	integer
animation	type of animation	string

- *exitAnimation*: the correspondent value is a dictionary that describes the animation with which the page disappears:

KEY	VALUE	VALUE TYPE
type	interpolation type	string
duration	duration of the animation	integer

animation	type of animation	string

We will see in the next paragraph the strategy used to describe and represent the elements of the page.

5.2.1 REPRESENTING ELEMENTS

Recall that basically we have an array of pages, where each page is represented by a dictionary (see figure 14), and the dictionary has an entry with key “elements” and an array as the corresponding value. Following this philosophy, we need to find something that identifies each kind of object on the page. For example, concerning images, the solution is quite simple: we can represent them with their file name in the file system. Basically an image does not need anything else to be represented. We delegate to the framework the problem of using the file name to show the image on the screen. We could decide to represent the image with its binary data (property lists allows to embed Data objects), but this would be a waste of space: images can have sizes of several megabytes, while a file name is only a few bytes long.

The next issue of the example is adding the image representation just found (or, in general, the representation of any other kind of object) to the elements array. An image is an element itself, so we could think to add the file name in the first position of the array. This solution is not smart, because the application could possibly find different kinds of elements in the array. Thus, we need to find a flexible solution to let the application interpret different kinds of elements. The trick is to use a dictionary to identify the properties of each element, as we did to represent the properties of a page. In other words, we use a dictionary to represent a particular element. The element dictionary must have general entries that refer to the common properties of different kinds of elements. For example images and texts must have dimensions, and at a least two variables to identify their position on the screen. On the other hand, sounds do not have this kind of properties, so finding a uniform representation for deeply different objects is a serious problem. The solution adopted was inserting an entry in the element dictionary to distinguish the *type* of the element. The application will use at first this entry to acquire knowledge about the element nature. The element dictionary will then list all the possible properties of the several kinds of objects, letting us to use a homogenous representation for all the objects. For example every element dictionary will have a *height* entry, although it is a sound. Distinguishing the type of element and simply ignore properties that do not make sense for the specific type of element will be a task of the application. This is the key aspect to make the application scalable. In fact, in the future we could decide to represent a new type of element, with new kinds of properties. Thanks to the generality of the representation, things will continue working because the application will simply ignore properties that do not make sense. Writing some code to interpret the new properties will be enough to manage them. We will see in the implementation section how we achieved this strategy.

Based on these considerations, we managed to decide a basic set of common properties for each element:

KEY	VALUE	VALUE TYPE
type	the type of element	string
x	the horizontal coordinate	integer
y	the vertical coordinate	integer
width	the width of the element	integer
height	the height of the element	integer
alpha	the transparency of the element	integer
name	the identifier of the element	string

Obviously the second, the third, the fourth, the fifth and the sixth property do not make sense for a sound, but we decided to group theme here, because they make sense for all the other kinds of objects. Nevertheless, putting the properties in the specific group of a type of element would not have made any difference: this choice has no effect on the implementation; grouping properties was made only for clearer explanation.

REPRESENTING IMAGES

As we said, all we need to represent an image is the file name (in addition to the basic properties). Thus, the key-value pairs will be:

KEY	VALUE	VALUE TYPE
type	“image”	string
fileName	the name of the file representing the image	string

REPRESENTING SOUNDS

In order to represent sounds we need to know the file name and, optionally, the time offset between the appearing of the page and the start of the sound itself. Offset property allows us to provide a delay to the sound. To represent this particular kind of element we created an entry with key *audio* and a dictionary as value:

KEY	VALUE	VALUE TYPE
type	“audio”	string
audio	a dictionary	dictionary

In the following table we can see the contents of the audio dictionary:

KEY	VALUE	VALUE TYPE
trackName	the file name of the audio track	string
offset	the delay	integer

REPRESENTING TEXTS

Representing texts is quite simple: we do not need to know any file name, because we can embed text directly in the property list (as a String property). Thus, we need to store information about the text itself (in addition to the common properties listed previously), the name, the colour and the size of the font:

KEY	VALUE	VALUE TYPE
type	"text"	string
text	the text	string
fontColor	the colour of the font, represent in hexadecimal format	string
fontName	the name of the font	string
fontSize	the size of the font	integer

REPRESENTING MOVIES

Movies are, in the Sparrow slang, are sequences of images (called frames) reproduced with a constant rate and possibly with sounds. They are constructed in sparrow as instances of SPMovieClip. The constructor of this class receives a series of textures, that will be used as frames. The frames are constructed from a texture atlas. A texture atlas is an xml file that describes the position and the dimension of the textures that have to be taken from a file. Usually this file is an image that contains all the textures that will make up the frames. All we need to do after constructing the atlas and the SPMovieClip instance is sending the play: message to the instance in order to start the movie. After that, adding the instance to the stage juggler[17] will start playing the sequence of images on the screen. Based on this pattern, the element dictionary has been provided with the following entries:

KEY	VALUE	VALUE TYPE
type	"movie"	string

[17] In the SPSprite class and in the SPStage class there is a particular object, called SPJuggler, that plays movies as soon as they are added to it. See figure 5.4 for more information about the rendering system of Sparrow.

frames	array of frames	array

Every element of the frames array is a dictionary with the following entry:

KEY	VALUE	VALUE TYPE
name	the filename of the texture	string

We decide to avoid using a texture atlas.

REPRESENTING ANIMATIONS

Animations are represented taking account of the behaviour of Sparrow. With Sparrow, we can animate every numeric property of an object, such as the width of an image, or the alpha value of a text. All we need to do is creating a tween object[18], specifying the target object to animate, and the final values of the target properties (a tween can animate more than one property of the same object). After constructing the tween we have to add it to the stage juggler[19] and we are done.

Therefore, if we decide that a particular object must be animated, the data model will have to describe a tween, specifying the target element (i.e. the object to animate), the target properties and their final values. Coming back to our property list, we designed a particular entry in the element dictionary to list all the tweens of an element (it should be enough using one tween for all the animated properties, but we wanted to give more freedom to the framework). The entry' value is an array of dictionaries, where each dictionary represents a particular tween:

KEY	VALUE	VALUE TYPE
tweens	array of dictionaries	array

In the following table we can see the contents of a tween dictionary:

KEY	VALUE	VALUE TYPE
name	identifier of the tween	string
delay	offset between the appearing of the page and the start of the animation	integer

[18] Tweens are objects that make animation possible. When the programmer wants to animate a numeric property of an object, he creates an instance of SPTween and he passes to its constructor the object whose property we want to animate, the name of the property, the duration and type of animation.

[19] The juggler also plays animations as soon as tweens are added to it.

properties	dictionary of properties to animate	dictionary
loopType	loop animation type	string
removeFromStageWhenFinished	remove from stage flag	boolean

We give a brief explanation of every property of the tween dictionary:

- **Name:** this property permits to identify the tween for the current element.
- **Delay:** we have already seen this property for sounds, under the name *offset.*
- **Properties:** we need another dictionary to be able to distinguish properties of the Sparrow objects to animate. Thus, keys are exactly the names of the properties of Sparrow objects. For example, SPImage has a property named *alpha* to keep information about the transparency. If we want to animate the alpha property of an image, then we add an entry in this dictionary with key "alpha"[20]. The final value of the animated property will be paired with the related key in the entry.
- **LoopType:** Sparrow provides three animation loop types; the first one is a no loop, the second one is a loop from the initial value to the final value of the property; the third type is a loop from the initial value to the final one and then back from the final to the initial one.
- **RemoveFromStageWhenFinished:** this flag tells the application if the current element has to be removed from the stage when the animation stops.

5.3 THE FRAMEWORK AND APPLICATION'S MVC DESIGN

5.3.1 MODEL

After deciding the base objects and logic that the book should have, the next topic was designing a set of classes to read the property list from the file and loading the pages into a runtime structure. In other words, we started designing the application model. Given the fact that the contents of the property list file and the array of pages are common aspects of every interactive book application, we decided to put this part of the project into the framework. In this way every time we create a new interactive book, the application will "ask" the framework for an array containing the pages with all the needed elements inside them.

[20] we use quotes to refer to a string

To achieve this goal, we started from the elements of the property list and we decided to wrap the properties of each element in a class that would have represented the element itself. After that we decided to wrap the elements in a class that would have represented a page. The last step was creating a class to represent the book, that would have read the property list file and created the structure of the pages of the book. We also designed a special class for representing tweens. Tweens are those particular objects used by Sparrow to implement animations of other Sparrow objects. This class embeds every property of the tween dictionary explained in the last paragraph. Summarizing, the model classes designed for the framework are the following:

1. **AKBook**: class designed to represent the book, and to keep a reference to an array containing the pages of the book
2. **AKPage**: class designed to represent a page, and to keep a reference to the elements of a page
3. **AKElement**: class designed to represent an element of the page, and all its properties
4. **AKTween**: class designed to represent a tween and its properties.

Note that the "AK" prefix stands for AppKid, the name of the framework. The properties tracked by AKPage, AKElement and AKTween are the same of the properties we discussed about in the property list section, so we do not list them once again.

5.3.2 CONTROLLER

After designing the model of the application, we came to the bigger topic of the design phase: constructing a system to interpret the elements of a book and to translate them in objects or actions on the screen. We decided to take inspiration from the first generation of applications built for interactive books and we kept its main structure. We can see it in the next page.

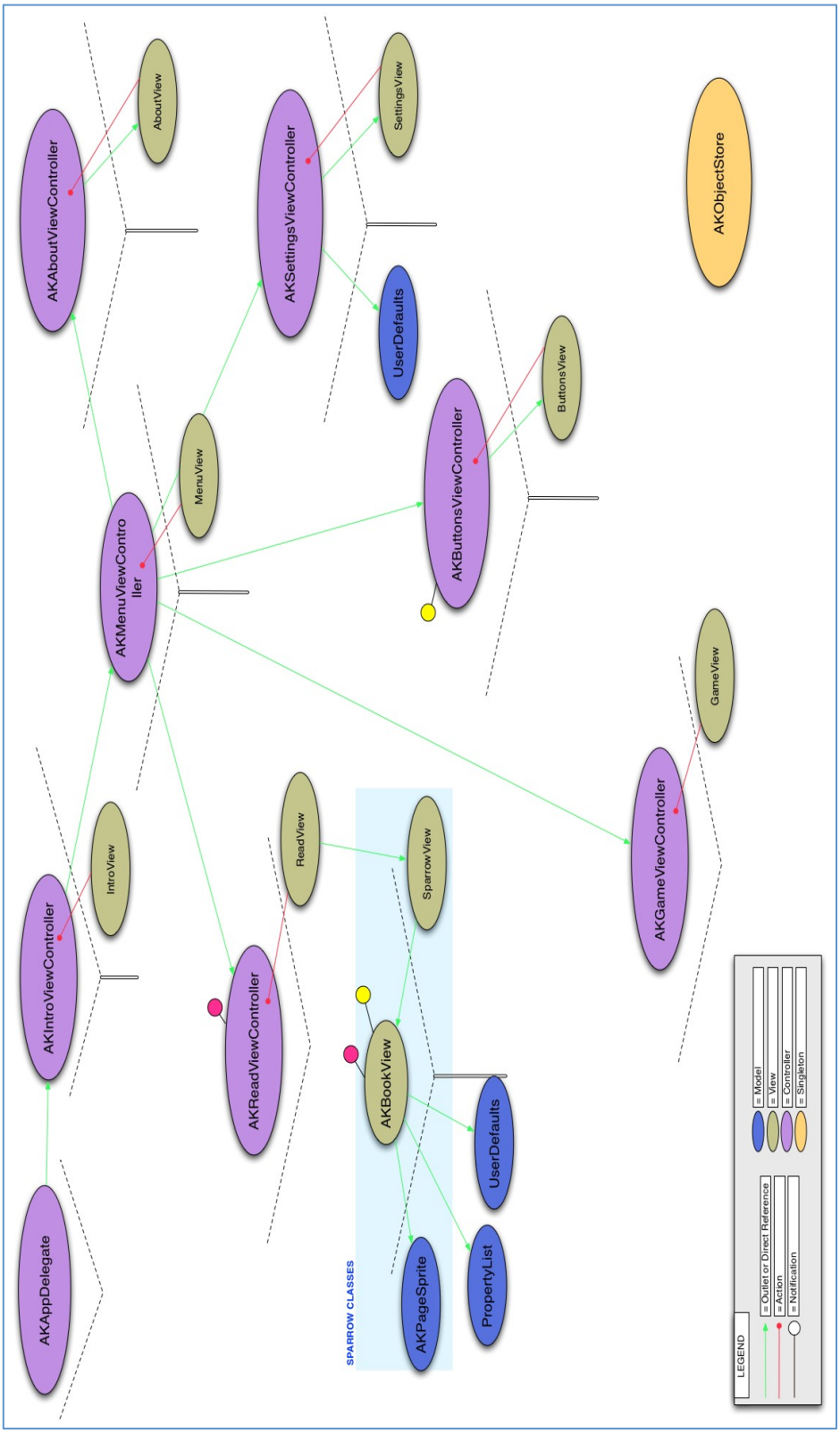

FIGURE 5.2 - MVC ARCHITECTURE OF APPKID

As we have already seen in chapter 4, the application is basically a set of MVCs linked together by direct reference or using outlets. The same idea was used in this context: the application starts with the MenuViewController, which shows

on the screen a menu made of several buttons. Clicking on a particular button forces the application to instantiate a new MVC and to push the related controller on the navigationController[21].

FIGURE 5.3 - THE MAIN MENU OF THE APPLICATION

For example, as we can see in figure 13, clicking on the "Gioca" button will push the GameViewController on the navigationController, as the first generation of interactive book application does. So far we only designed the ReadViewController, the other controllers are designed only in the logic to come back to the main menu. Designing and developing the other controllers and related sections (game, settings, index, etc) is a future goal. The key aspect is that all the controllers used in the first generation are related to common aspects of the books. In fact, every book has to have an index, it has to provide logic to record voice and to set some options (choosing the language for example, and stuff like that). In addition, every book could have some games, and, in particular, the two games mentioned in chapter 4 are totally reusable for every book: they are both designed to take an image as input, and they use it to construct the puzzle or the filling game. Thus, it becomes clear that all these controllers have to be put in the framework.

The problem now is that we have a general and reusable framework, but we have not talked about the application yet. Obviously a framework is a simple set of classes, so it cannot be executed on an iPad. It is necessary to create an application that uses the framework to show the contents of the book on the screen. We have also to keep in mind that the application should

[21] Pushing a controller on the navigation controller causes the appearance of the related view and the disappearance of the previous view in an animated manner. UINavigationController is a particular controller used by Cocoa Touch to achieve this behaviour.

allow implementing the particular aspects of a single interactive book. In this context inheritance helped us very much. As a matter of fact, we decided that the application should have been an extension of the framework. More specifically, all the controller classes of the application should have been basically subclasses of the related controller class of the framework. These extensions basically do not provide more functionalities than the framework does, but in the future someone could want to give a particular function to a book. To do this he will be obliged to write some methods in the subclasses of the application, and this code would overwrite the basic methods of the various controllers. In this way when the programmer wants to create a new interactive book, he will create subclasses for the controllers placed in the framework. If he does not want to add custom behaviour, he will simply edit the property list file in order to create the interactive book and then link the application against the framework, without overwriting methods. Thanks to inheritance, when the application will start, the methods of the superclasses will be invoked, because the subclasses will not have overwritten them. Thus, the framework classes will be used to read the property list, to process data about pages and to render graphic objects on the screen. We give here a list of the controllers designed to be in the framework (i.e. all the controllers available):

- **FRAMEWORK**
  - AKMenuViewController
  - AKGameViewController
  - AKAboutViewController
  - AKSettingsViewController
  - AKRecordViewController
  - AKIndexViewController
  - AKReadViewController
  - AKButtonsViewController
  - AKBookAppDelegate

By the way, this is only a design discussion: as we said, we put our strengths on designing at first a set of classes in order to reproduce the interactive pages, reading them from the property list, and we did not focus on the index, on the game and stuff like that. In other words, we concentrated on the design of the ReadViewController, delaying to the future the other functionalities. Although the skeleton of the application is quite the same of the first generation, the ReadViewController is deeply different from the first one, and it is now placed in the framework.

DESIGN OF THE READVIEWCONTROLLER

At this point, our concept of the interactive book is the following:

1. The program starts showing the main menu, corresponding to the push of the MenuViewController on the navigationController.
2. After clicking on the “Leggi” button, the AKReadViewController is pushed on the top of the navigationController.

In the framework we changed the structure of the ReadViewController: we decided that the SPStage class extension (called AKBookView) would be instantiated every time the ReadViewController is pushed on the navigationController. In this way every time the ReadViewController is popped off from the navigationController it gets deallocated (i.e. when the user taps on the button used to return to the main menu). At this point, after loading its own view, the AKReadViewController uses the sparrowView[22] created in the correspondent ReadViewController.xib file to initialize the stage. The stage has the same goal of the stage of the first generation of applications: rendering the pages.

In the framework there is a great difference: when the user wants to go to the next page, the stage uses information from the property list to render the new page. We know that Sparrow uses an SPSprite instance to construct a page: we can add images, texts, sounds, animations and so on; adding the SPSprite instance to the stage forces it to be rendered on the screen. When the stage has to start a new page (there is a special method in the stage class to do this), it creates a new AKPageSprite instance (called pageSprite) and it adds the sprite as a child of itself (see figure).

[22] The sparrowView is an instance of the SPView object, which is the UIView object that Sparrow renders its content into.

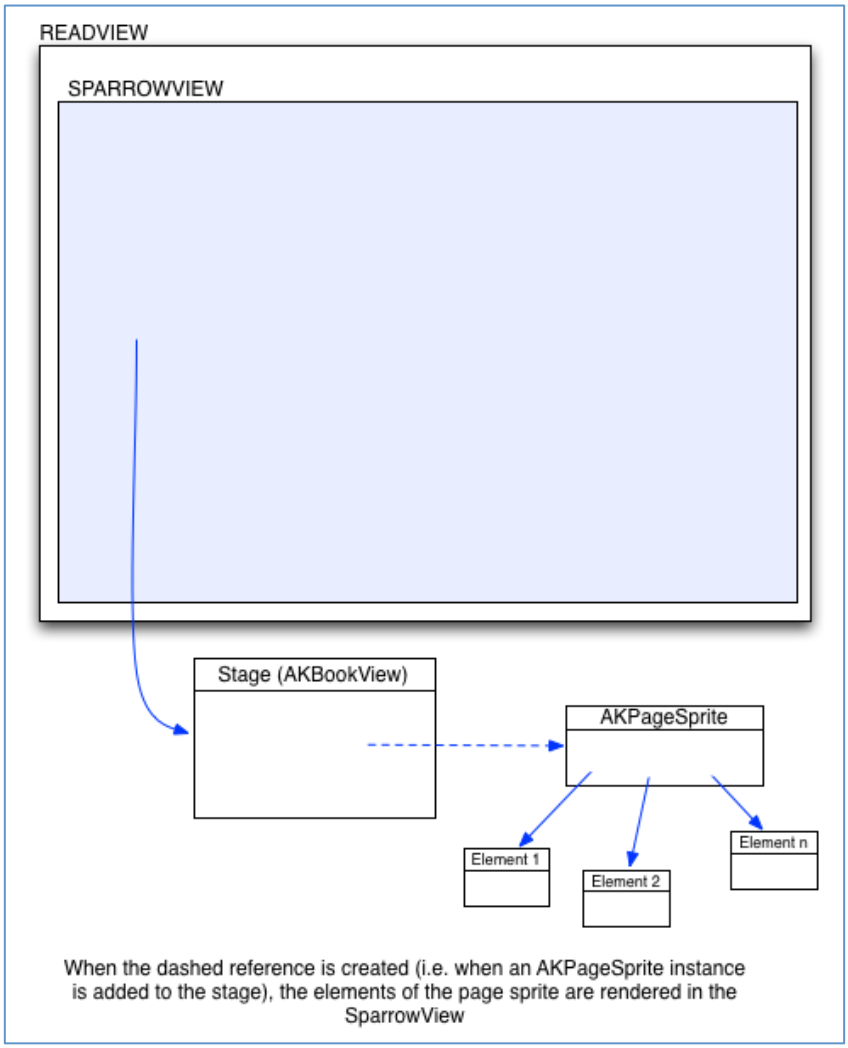

FIGURE 5.4 - VIEW CONTROLLING OF READ VIEW CONTROLLER

In fact the way a pageSprite is added to the stage depends in our design by its *enterAnimation* value in the page features dictionary. So far we have only designed a slide effect in order to make the page appear and disappear.
Apart that, constructing a pageSprite by reading information from the property list is the most important issue. To achieve this goal we designed two algorithms, the first to read the property list and construct the array of pages (which will be implemented in the application delegate), and the second to extract the needed information from the array and to create an AKPageSprite instance for every page (which will be implemented in the PageSprite init method).

CONSTRUCTING THE PROPERTY LIST AT RUNTIME

```
start;
pageDicts <- read the property list from file;
pages <- new array
for (i <- 0, i < pagedDicts.size, i++) {
        page <- pageDicts[i];
        akPage <- new instance of AKPage;
        for (j <- 0, j < page.elements.size, j++) {
                akElement <- create new instance of AKElement using
                        page.elements[j];
                add akElement to akPage;
        }
        pages[i] = akPage;
}
return pages;
end;
```

ALGORITHM 5.1 – CONSTRUCTING THE ARRAY OF PAGES

As we can see, the algorithm reads the property list from the file and puts the pages (read as dictionaries, see chapter 4) in an array. Then the algorithm starts to scan every position of the array to read the elements of a particular page. In particular, it uses the j-th dictionary of the page to construct the j-th element of the page. A page is allocated as an instance of AKPage before starting to check the elements of the page itself.

CREATING A PAGESPRITE FOR A PAGE

```
start;
elements <- current elements of a page;
pageSprite <- new empty page sprite;
for (i <- 0, i < elements.size, i++) {
    currentElement <- elements[i];
    if (currentElement.type == image) {
        spImage <- create Sparrow image instance using  filename;
        set position;
        set dimensions;
        set alpha;
        set name;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        add spImage to pageSprite;
    } else if (currentElement.type == text) {
        spText <- create Sparrow text;
        set position;
        set dimensions;
        set alpha;
        set name;
        set font size;
        set font name;
        set font color;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        add spText to pageSprite
    } else if (currentElement.type == audio) {
        spAudio <- create Sparrow audio instance using trackName
        set position;
        set dimensions;
        set alpha;
        set name;
        set font size;
        set font name;
        set font color;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        add spAudio to pageSprite;
    } else if (currentElement.type == movie) {
        spMovie <- create Sparrow movie instance using frames read from
            the property list
        set position;
        set dimensions;
        set alpha;
        set name;
        start playing movie;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        add spMovie to pageSprite;
    }
}
end;
```

ALGORITHM 5.2 – CREATING A PAGESPRITE

The algorithm is quite simple: it checks all the elements instantiated from the property list and tracked in the AKPage class and it constructs the proper Sparrow element for each object. If some element needs animations, some tweens are created and added to the pageSprite's juggler.
By using this basic algorithm, the ReadViewController can render on the sparrowView the pages of the book, showing animated images and texts on the screen and playing sounds. These are not the only capabilities of the framework, and in 5.4 we will see the design of the extensions to this basic behaviour.

5.3.3 VIEW

All the xib files created in the Interface Builder represent the view. We imagined that the visual appearance of a menu or of a section could not be the

same for all the books. Therefore, we decided to put the xib files in the application. This gives the possibility to the author of an interactive book to customize every menu or section wit his own preferences, without modifying the underlying framework. Obviously, the possibility to extend the controller classes of the framework lets the author modifying also the behaviour and the logic behind the appearance of a section. For example, if the author wants to add another game in the book besides the two games descripted, he could add some objects to the xib file of the GameViewController, and add the code for the new game in the subclass of AKGameViewController, extending its functionalities. We can see that this is quite a scalable strategy in order to customize the application.

The views are designed to be funny and suitable for kids. Thus, they are quite simple and with a restricted set of interactive items. In this phase we decided to recreate the book Chissà with the framework and using the plist[23] file, so the concept of the pages of the book is the same of Chissà.

5.4 EXTENSION OF THE FRAMEWORK DESIGN

After designing the architecture of the framework and of the application, and how they interact together creating a scalable system, we focused on extending the capabilities of the framework, in order to achieve all the goals presented in 5.1. We have realized that representing images, texts, sounds and movies is quite simple with a property list. Unfortunately, representing objects that have to be rendered exactly after others, or animations that have to start after a particular event, is not as simple. We will see in 5.4.1 how this problem was solved.

Another important topic was giving the capability to the framework to distinguish if some object may react to a tap done by the user: for example, in the first page of Chissà, if the user taps on the head of the child, it will start laughing. The idea is to cook up a set of properties in the property list file to tell if an object has to react to a tap of the user. This is an important aspect, because it is the soul of the book's interactivity. We will talk about it in 5.4.2
The following topic we are going to talk about is the capability of managing the dragging of objects around the screen. This is a very hard feature, and it has not been implemented yet. We will give an idea of the solution in 5.4.3.

5.4.1 DESIGNING THE TRIGGERING SYSTEM

As we said, we have to face the problem of giving objects the opportunity to be rendered when some animation stops, or when it starts. In the same way, it would be very nice to create the possibility to start an animation after another one, or stuff like that. Therefore, we have to create a system to synchronize animations and objects.

A simple solution would be setting the time delay of the appearance of the object. In other words this would mean specifying *when* something has to

[23] "plist" stands for property list

happen. This is quite uncomfortable because we have to estimate the exact timing for each object and for each animation. Moreover, changing the script of the page would mean computing all the timings from scratch, leading to a huge loss of time. For example, if we decide that in an already constructed page there must be a new text that appears before any other object, this will force us to compute a new timing for each object. In the first generation of interactive books we exploited the power of Sparrow in order to synchronize objects and tweens. Sparrow in fact provides an event-driven development environment, which is very useful for this kind of situation: every kind of action done on an object generates an event. For example, when an animation ends, the tween in charge for the animation generates an event, notifying that the animation ended. This is quite useful, because we can schedule a particular action exactly at the end of the animation by adding an event listener to the tween. The event listener can specify the method to be invoked when receiving a particular event, and the object to invoke the method on[24]. For example, we can decide to add a text exactly at the end of the animation of an image. Therefore, writing code to do this in the target method would be enough. In Sparrow we have the following types of events:

- **SP_EVENT_TYPE_TRIGGERED:** a button was triggered
- **SP_EVENT_TYPE_ADDED:** a display object was added to a container
- **SP_EVENT_TYPE_ADDED_TO_STAGE:** a display object was added to a container that is connected to the stage
- **SP_EVENT_TYPE_REMOVED:** a display object was removed from a container
- **SP_EVENT_TYPE_REMOVED_FROM_STAGE:** a display object lost its connection to the stage
- **SP_EVENT_TYPE_ENTER_FRAME:** a new frame of an animation is displayed
- **SP_EVENT_TYPE_TOUCH:** a touch event occurred
- **SP_EVENT_TYPE_TWEEN_STARTED / UPDATED / COMPLETED:** a tween changed its state (shown in the animation section)
- **SP_EVENT_TYPE_SOUND_COMPLETED:** a sound finishes.

By the way, it is possible to create custom events too. We did not use custom events though in our framework. The events just listed specify very meaningful moments in which we want to decide to do something.

From the important aspects we have just talked about, it is clear that we can exploit the Sparrow event system to create a triggering scheme. This is more comfortable than the timing delay system described previously. If we want an image appearing after a sound, we will say that the sound triggers the image. If in the future we want to change this behaviour, we only have to modify it, without changing the time delay of any other object in the page.

[24] Thanks to Objective-C we can specify which methods we want to invoke, or better, which messages we want to send, and the destination object of the message. Objective-C is a highly dynamic programming language and lets the programmer sending messages, or invoking methods, also on objects that do not implement that particular method.

Nevertheless, there is a big problem: Sparrow mechanism forces us to create an object and then to add an event listener to that object. When the object generates a particular event, a particular method is invoked and some actions are done. In the property list we cannot specify what actions to do when an event is dispatched by an object, we can only describe elements and their properties. Therefore, in the property list file we can only describe the triggering object and the triggered object. It will be a task of the framework adding the event listener to the right object; the framework will also have to select the right method when the triggering object dispatches the event, and this method will have to force the triggered object doing something (like simply appearing, or starting an animation).

EXTENDING THE PROPERTY LIST FEATURES

In order to extend the property list with this feature, in a first time we decided to insert a list in the element's features dictionary of a triggering object, containing all the triggered objects. This solution is impossible to realize: when creating the triggering object in the pageSprite we are not sure if the triggered object has been already created (usually in the plist we describe objects of the page starting from the ones that are not triggered by anything). In this situation it would be easy to add a listener to the triggering object, but it would be impossible to put the triggered object in a queue of pending elements, because we would not be sure that the object has already been instantiated.

As a solution, we decided to create a dictionary for each object, with information about the object that triggers the current one. In this way, when constructing the pageSprite, we can add the object in a queue only if it has a triggering object; adding the listener to the triggering object is still possible, because we are sure that it has already been instantiated. Remembering that each property of an element of the page is a key-value entry in the element dictionary, we added an entry with key *trigger* and a dictionary as value (the dictionary we have just talked about):

KEY	VALUE	VALUE TYPE
trigger	dictionary describing the triggering object	dictionary

We decided to represent the following properties in the triggering object dictionary:

KEY	VALUE	VALUE TYPE
triggeringItem	name of the triggering object	string
triggeringItemType	the type of the triggering object	dictionary
triggerIsATween	flag used to tell if the trigger is a tween associated with the object	boolean

triggeringTweenNumber	identifier of the triggering tween, if the trigger is a tween	integer
triggerIsATouch	flag used to tell if the trigger is a touch event[25]	boolean
triggerWhen	if the trigger is a tween, this entry specifies at which point of the animation the object has to be triggerd	string

EXTENDING THE FRAMEWORK TO MANAGE TRIGGERS

After designing the data model of the triggering system, we had to plan how to extend Sparrow, because for our implementation, the event dispatched by the triggering object had to carry also an identifier of the triggering object. Sparrow does not provide this feature, so we will see later how it was extended.

In order to extend our framework, we started modifying the algorithm used to construct the pageSprite, distinguishing the objects to add without any kind of trigger, and those with trigger:

[25] For more details, see 5.4.2 .

```
start;
elements <- current elements of a page;
pageSprite <- new empty page sprite;
for (i <- 0, i < elements.size, i++) {
    currentElement <- elements[i];
    if (currentElement.type == image) {
        spImage <- create Sparrow image instance using fileName;
        set position;
        set dimensions;
        set alpha;
        set name;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        if (!currentElement.hasTrigger) {
            add spImage to pageSprite;
        } else {
            triggeringElement <-  findTriggeringElementOf(currentElement);
            addObjectToPendingListWithTriggeringElement(spImage,
                triggeringElement, currentElement);
        }
    } else if (currentElement.type == text) {
        spText <- create Sparrow text;
        set position;
        set dimensions;
        set alpha;
        set name;
        set font size;
        set font name;
        set font color;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        if (!currentElement.hasTrigger) {
            add spText to pageSprite;
        } else {
            triggeringElement <-  findTriggeringElementOf(currentElement);
            addObjectToPendingListWithTriggeringElement(spText,
                triggeringElement, currentElement);
        }
    } else if (currentElement.type == audio) {
        spAudio <- create Sparrow audio instance using trackName
        set position;
        set dimensions;
        set alpha;
        set name;
        set font size;
        set font name;
        set font color;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        if (!currentElement.hasTrigger) {
            add spText to pageSprite;
        } else {
            triggeringElement <-  findTriggeringElementOf(currentElement);
            addObjectToPendingListWithTriggeringElement(spAudio,
                triggeringElement, currentElement);
        }
    } else if (currentElement.type == movie) {
        spMovie <- create Sparrow movie instance using frames read from
            the property list
        set position;
        set dimensions;
        set alpha;
        set name;
        start playing movie;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        if (!currentElement.hasTrigger) {
            add spMovie to pageSprite;
        } else {
            triggeringElement <-  findTriggeringElementOf(currentElement);
            addObjectToPendingListWithTriggeringElement(spMovie,
                triggeringElement, currentElement);
        }
    }
}
end;
```

ALGORITHM 5.3 - CONSTRUCTION OF PAGESPRITE WITH TRIGGERS

As we can see, the main difference is a guard that tells if the current element being constructed has to be triggered or not. In the latter case the algorithm

invokes the function `addObjectToPendingListWithTriggeringElement`, passing as parameters the Sparrow object to add, the triggeringElement retrieved with `findTriggeringElementOf()`, and the element corresponding to the Sparrow object. The pending list is a property of the AKPageSprite class of AppKid framework and it is a dictionary. The keys of the dictionary are the names of the triggering objects, and the related values are arrays of objects that are waiting for the dispatch of an event from the triggering object. The function simply distinguishes what kind of listener to add based on the type of the triggering object:

1. If the triggering object is an audio object, the algorithm adds a listener to the PageSprite's audio channel; the listener has to wait for events of type SP_EVENT_TYPE_SOUND_COMPLETED; after that the algorithm adds the object to be triggered in the pending list, using as key the triggeringItem property of the *trigger* dictionary; the object is added in the array associated with the key in a FIFO mode and then the array is inserted again in the dictionary with the same key.
2. If the triggering object is an image, the only possibility is that the trigger is a tween of the image; therefore the algorithm starts checking if the triggering object is a tween of the image; in this case, the algorithm searches for the tween in a dictionary of the current tweens of each element of the page. The entries of the dictionary are key-value pairs, where each key is the name of the element containing the tweens, and the value is an array of tweens used by the element. Then the algorithm uses the triggeringItem property of the triggering object dictionary as key and the triggeringTweenNumber as the index of the array of tweens obtained from the dictionary. After extracting the right tween, the algorithm adds an event listener to it (listening for events of type SP_EVENT_TYPE_TWEEN_COMPLETED) and adds the object to be triggered to the pending list in the same way seen at point 1.
3. There is also the possibility that the triggering object is an active area. See 5.4.2 for more details.
4. Management of other kinds of triggering objects have not been designed yet.

When the triggering object dispatches the event, the event listener invokes a target function, passing the event as parameter. The event carries out a reference to the triggering object (this is the feature added to Sparrow), so that the algorithm can extract the array of objects waiting for it and add them to the pageSprite. We will see in the implementation section the particular aspects of this algorithm.

5.4.2 DESIGNING INTERACTIVE TOUCH

As we have already said, the core of interactivity is giving the user the opportunity to touch some objects on the screen and receive a reaction by them. In this section we will explain the construction of a system to describe

interactive objects in the property list file and to interpret them in the framework.

EXTENDING THE PROPERTY LIST FEATURES

In order to give the property list the capability to representing interactive touch, we started from the basic idea that there must be an area on the screen able to react to the touch of the user. Therefore, we decided to create a new type of element: the active area element. It represents a rectangular area on the screen and obviously it has to be sensible to touches. At the data model level we created the new active area type and we added a Boolean in the element dictionary to tell if the active area is active:

KEY	VALUE	VALUE TYPE
type	"activearea"	string
isActive	yes if the area is active, no otherwise	boolean

One could think that obviously an active area is always active, so the second property could not be necessary. However, we decided to use this second property because in the future we could decide to make other objects (like images and text) become touchable; thus, this type of property could be useful for this goal. Anyway, the hardest part of the work has been delegated to the framework.

EXTENDING THE FRAMEWORK TO MANAGE ACTIVE AREAS

In order to manage active areas, we extended algorithm 5.3 (used to create an AKPageSprite instance) with a guard that tells if the current element is an active area. In this case the algorithm creates a rectangle (Sparrow makes available a programmatic interface to create geometric shapes) using the dimension properties read from the element dictionary in the property list. Then it checks if the element is active, using the second property listed in the last table and in this case it adds an event listener to the rectangle just created (listening for events of type SP_EVENT_TYPE_TOUCH). After that the rectangle is added to the current pageSprite.

```
start;
elements <- current elements of a page;
pageSprite <- new empty page sprite;
for (i <- 0, i < elements.size, i++) {
    currentElement <- elements[i];
    if (currentElement.type == image) {
        spImage <- create Sparrow image instance using fileName;
        set position;
        set dimensions;
        set alpha;
        set name;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        if (!currentElement.hasTrigger) {
            add spImage to pageSprite;
        } else {
            triggeringElement <-  findTriggeringElementOf(currentElement);
            addObjectToPendingListWithTriggeringElement(spImage,
                triggeringElement, currentElement);
        }
    } else if (currentElement.type == text) {
        spText <- create Sparrow text;
        set position;
        set dimensions;
        set alpha;
        set name;
        set font size;
        set font name;
        set font color;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        if (!currentElement.hasTrigger) {
            add spText to pageSprite;
        } else {
            triggeringElement <-  findTriggeringElementOf(currentElement);
            addObjectToPendingListWithTriggeringElement(spText,
                triggeringElement, currentElement);
        }
    } else if (currentElement.type == audio) {
        spAudio <- create Sparrow audio instance using trackName
        set position;
        set dimensions;
        set alpha;
        set name;
        set font size;
        set font name;
        set font color;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        if (!currentElement.hasTrigger) {
            add spText to pageSprite;
        } else {
            triggeringElement <-  findTriggeringElementOf(currentElement);
            addObjectToPendingListWithTriggeringElement(spAudio,
                triggeringElement, currentElement);
        }
    } else if (currentElement.type == movie) {
        spMovie <- create Sparrow movie instance using frames read from
            the property list
        set position;
        set dimensions;
        set alpha;
        set name;
        start playing movie;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        if (!currentElement.hasTrigger) {
            add spMovie to pageSprite;
        } else {
            triggeringElement <-  findTriggeringElementOf(currentElement);
            addObjectToPendingListWithTriggeringElement(spMovie,
                triggeringElement, currentElement);
        }
    } else if (currentElement.type == activearea) {
        spRectangle <- create Sparrow rectangle instance
        set position;
        set dimensions;
        set alpha;
        set name;
        set color;
        if (currentElement.hasTweens) {
            add tweens for the current instance;
        }
        add spRectangle to pageSprite;
    }
end;
```

ALGORITHM 5.4 - CREATING A PAGESPRITE WITH ACTIVE AREAS

Note that we have not designed triggered active areas yet. This can be a feature to develop in the future. The last part of the algorithm shows the adding of active areas to the pageSprites.

In order to make things work, something has to happen when the user taps on an active area. Therefore, in the property list we added the opportunity to specify an active area as a triggering object. In other words, the *triggeringItemType* of the *trigger* dictionary can be "activearea". The framework has to be able to interpret active areas as triggering objects and for this reason we modified `addObjectToPendingListWithTriggeringElement`, adding a guard that manages the case in which the triggering element is an active area. In this case the object just created is only added to the pending list (in the way we have already seen). There is no need of adding event listeners, because this has been already done in the creation of the active area. When the active area dispatches a touch event, the event listener invokes a target function, passing the event as a parameter. We will see in the implementation section how this function is implemented.

5.4.3 DESIGNING DRAGGING OF OBJECTS

Dragging objects means that the user can touch an object on the screen and move the object around using its finger. This an advanced feature, which gives an entertaining experience to the kid "reading" the book. For example, in the fourth page of Chissà there is a draggable train in the background; in this case the problem of dragging objects was solved in the following way: we decided to add an invisible rectangle over the train in foreground, and made that rectangle reactive to touches. In this way, when the user will touch the invisible, he will implicitly touch the train. The rectangle will start tracking the position of the touch event and it will move itself and the train image consequently.

We planned that the framework should do something similar: the property list should specify if an object is draggable. In that case the framework should add an invisible rectangle over the draggable object and track the touch events' position. The tracking should start when the user starts touching the draggable object and go on while the user moves his finger keeping the object touched. Obviously, tracking the position would have to update the position of the draggable object and of the invisible rectangle too.
We did not design this feature in more detail and we left it to be developed in the future.

5.5 IMPLEMENTATION OF THE FRAMEWORK

In this paragraph we discuss about the special coding solutions employed to implement all the algorithms designed in the previous paragraph. Most of the solutions were presented in the last paragraph and the implementation had not any such important issue that deserves to be mentioned here. This is why we report here only some particular aspects of the coding phase.

5.5.1 EXTENSION OF SPARROW

Sparrow is a powerful framework and lets the programmer use a simpler programmatic interface than Cocoa frameworks, hiding the API of UIKit and Foundation. It is very useful for many reasons: its event handling system is simpler than the Cocoa one; it gives the opportunity to customize an application with several multimedia components, such as sounds, images, texts and so on. Moreover, it lets the programmer create animations without knowing anything of Core Animation framework. In my opinion the most important feature is the possibility to render objects in a very simple way: we have seen that adding an object to the pageSprite is enough to show it on the screen. Cocoa is more complicated: to render objects on the screen we have basically to *draw* them using UIKit or OpenGL.

All these advantages have an important drawback: Sparrow does not provide the same freedom we have when using directly UIKit and Foundation. This is quite natural: in programming there is always a trade-off between the ease of use and the power of a tool. Moreover, Sparrow has been designed to be used directly in an application, and not as part of another framework. In our case, the main problem was using Sparrow to construct the triggering system. As we have already seen, when the triggering object dispatches the event, it needs to carry an identifier of itself. When the pageSprite receives the event, it must be able to know the name of the object that dispatched it because it has to extract the correspondent array of waiting objects (remember that this array is an entry of a dictionary, and the key of this entry is the name of the triggering object).

For all this reasons, we planned and managed to extend Sparrow. Sparrow provides a property called *name* (Sperl, SPDisplayObject Class Reference) to identify its display objects, but natively it does not construct events that carry information about the generating object. Therefore we extended all the classes instantiating objects that can act as triggers for us; after that we wrote some code to make them construct events that keep a reference to the constructing object. We also extended the event class and added two properties: the first represents the triggering object and the second represents the generating tween (if the event has been generated by a tween associated with an object). Thus, when a pageSprite receives the event, it can easily get to know about the triggering object or the triggering tween and use its name to find the list of waiting objects. The name of objects and tweens is set when the pageSprite is initialized and it is read from the property list. For objects, it is in the form "*element*" followed by the number of the element; otherwise it is in the form "*tween*" followed by the number of the tween.

5.5.2 IMPLEMENTATION OF THE MODEL

Recalling 5.3.1, AKElement is the class representing a property list element, and it will be used to construct the relative Sparrow object. This class has a property for each possible key of the element dictionary, and it sets the property using the value of the corresponding key in the dictionary. For

example, for setting the type (image, sound, text, etc.), the constructor of the class searches the value associated to the "type" key in the current dictionary using the `objectForKey:` method of the NSDictionary[26] class. If the element being read has some properties in the property list that do not make sense for that particular type, they simply will be ignored. If some property is missing in the property list, the `objectForKey:` method will simply return nil, because it does not find the key passed as parameter in the dictionary. In this way, properties that are not present in the current element dictionary, will be simply set to nil in the AKElement class.

5.5.3 IMPLEMENTATION OF THE CONTROLLER

In this section we give an explanation of the solution used to implement the algorithms that describe the logic of the controller.

PAGE SPRITE ALGORITHM IMPLEMENTATION

Algorithm 5.2 is implemented in the `init:` method of our SPSprite sublclass, called AKPageSprite. We have to remark some features of the implementation of it:

- In order to detect if the current element is triggered by some other object, the guards check a Boolean property of the element that is set to TRUE only if the element dictionary has a valid *trigger* entry;
- The tweens dictionary mentioned in 5.4.1 is a property of the pageSprite; the array of tweens associated with the object identified by the key in the entry is extended with a new element whenever a new tween is created with that object as target.
- To start playing a movie, after its construction using the frames provided from file, we have to invoke the start method on it and then to add it to the stage.

PAGE TRANSITIONS

An interesting aspect of the implementation of the rendering of a is how the pageSprite is added to the bookView. In fact the way a pageSprite is added on the stage depends by its *enterAnimation* dictionary in the page features dictionary. In the `startOtherPage:` method of the AKBookView class (i.e. the stage) the new pageSprite is created and then a series of guards uses the *animation* property of the *enterAnimation* dictionary to get to know how to animate the appearance of the new page. The implementation of the *slide* effect has been done as follows:

- The new pageSprite is added to the stage, but its coordinates are set to (1024,0). In this way the new pageSprite is not yet visible.

[26] NSDictionary is the Cocoa implementation of the dictionary abstract data type.

- A tween is added to animate the *x* property of the current pageSprite, with final value equal to -1024.
- Another tween is added to the new pageSprite, with final value equal to 0.
- The two tweens are added to the bookView's juggler and the slide animation starts.

The effect is also applied to the disappearance of a page in a similar manner. We thought about other animations for the appearance and disappearance of a page, like the page curl (an effect that simulates the page browsing) or the fade effect. Nevertheless, we have not implemented these effects yet.

TARGET METHOD OF PAGE SPRITE

We saw in 5.4.1 that adding the event listener to a triggering object involves specifying the target method to invoke when the object dispatches the event and the event listener receives it. The method created for this purpose is `addObjectToPageSprite:`. This method receives the event from the listener as an input parameter and starts extracting the triggering object reference from the event. Then it extracts the name of the triggering object and possibly the name of the generating tween. Then it pulls out the list of objects waiting for the triggering one from the so-called pending list. This is done by querying the pending list with the name just found. After that a for loop starts scanning the array of waiting objects and adds them to the pageSprite.

TARGET FUNCTION OF ACTIVE AREA

In 5.4.2 we saw that every time an active area is created, it is added an event listener with a target method named `touchHappened:`. The behaviour is similar to `addObjectToPageSprite:`.

5.5.4 MEMORY MANAGEMENT

The memory management aspect is crucial in Objective-C and especially when we develop applications on a device like iPad, which have a limited amount of memory[27]. For this reason we had to be very careful with the allocation of objects in the framework. The first big mistake was allocating and adding event listeners to triggering objects and active areas without removing them when not useful anymore (i.e. when the pageSprite gets deallocated). We managed to fix this by removing explicitly the event listener of an object in the `dealloc` method of the AKPageSprite class. Another important issue was to get the objects of a pageSprite properly deallocated when the pageSprite is deallocated. These objects have a very complex retain graph, and to achieve our goal we had to overwrite the retain method of every graphic class to check where the retain count of the objects increased. In the overwritten method we inserted a `NSLog()` function invocation to print information of who were

[27] The first generation of iPads have a total amount of RAM equal to 256 MB.

retaining the object under analysis, and this has been a quite tedious work. With the help of some tools to check memory leaks, we managed to fix these problems and to build an application that does not crash due to memory breakdown.

6 EDITOR

In this chapter we will talk about the development of AppKid Editor, the second part of my task in Altera. We start explaining why Altera decided to create an editor of interactive books and then we go on introducing the goals found in the analysis phase of the software. After that we deal with the design phase, and we list all the possible solution hypothesized, explaining why we had to discard some of them. Then we talk about the design of the application structure and about the implementation solutions used.

6.1 MOTIVATIONS

The idea of constructing an editor of interactive books occurred right after the idea of the software framework. There are several reasons why we can think to construct an interactive book with an editor, instead of editing an XML file or programming it from scratch, and they are very clear and simple:

- **Creating interactive books in a faster way**: obviously adding and editing the elements of a page with a visual tool is much more convenient and easier than modifying the XML property list and writing XML code. Just for example, imagine the simple operation of placing a text in a page: with a visual tool we can place it actually where we want; if we write the coordinates of text in the XML file we are forced to compile AppKid application and run it to check if the text has been placed in the right position, leading to a big loss of time.
- **Giving non-programmers the opportunity to develop interactive books:** Altera have the ultimate goal of selling the application to anyone who wants to develop interactive books and who has not any knowledge of programming or XML editing.

6.2 ANALYSIS: TARGET FEATURES

During the analysis phase we listed all the features the application should have when completed. We started from the idea that AppKid Editor should give the same freedom given by programming the interactive book from scratch. Therefore the features of the editor should be:

- Creating several pages and removing them
- Adding and removing images to a page
- Adding and removing texts to a page
- Editing texts
- Selecting multiple images and texts in the page canvas[28]
- Setting position and dimensions of text and images

[28] By canvas we mean the area on the screen where the user can draw objects using the specified application

- Adding sounds
- Specifying animations of objects in a page
- Specifying triggering of objects
- Saving and loading a project
- Generating a property list file according to the objects placed in the page

6.3 DESIGN

The design of the application has been quite difficult, because my knowledge about creating graphical applications was really poor at the beginning of this project. First of all we had to create a *Document-Based Application*[29] for Mac OS X using the template provided by XCode[30]. Thanks to document-based application skeleton, we have the possibility to construct the graphical user interface and the logic of the document part separately from the rest of the application. Our document will be the book we are trying to construct using the editor, and it will be made of several pages. First of all we needed to provide a GUI (and all the other underlying components) implementing all the features listed above. The GUI should have been provided with tools to construct the elements of every page.

The most important problem at the beginning was finding a way to place graphical objects inside a custom NSView[31] and tracking them in the application. After a period of reading up on the various technologies provided by Apple to do this, we synthesized and tried three possibilities:

1. The use of the Core Animation framework and CALayer[32] ("CA" stands for Core Animation) class; we took a cue from the LightTable application example provided by Apple (the first of the Apple example applications analyzed). This application is basically made of a view in which we can add images with a drag-and-drop mechanism. Images are draggable across the view and we can change their dimension by clicking on their corners and moving them. Behind the scenes there is a custom view[33] class that adds a CALayer (after enabling support for

[29] For more details about Document-Based applications, see Appendix.

[30] Xcode is a suite of tools developed by Apple for developing software for Mac OS X and iOS, first released in 2003. The latest stable release is version 4.2.1, which is available on the Mac App Store free of charge for Mac OS X Lion users.

[31] NSView is a class that defines the basic drawing, event-handling, and printing architecture of an application. NSView objects (also know, simply, as view objects or views) are arranged within an NSWindow object, in a nested hierarchy of subviews. A view object claims a rectangular region of its enclosing superview, is responsible for all drawing within that region, and is eligible to receive mouse events occurring in it as well.

[32] CALayers are simply classes representing a rectangle on the screen with visual content. Every UIView contains a root layer that it draws to. We can add CALayers as sublayers of this root layer and we can use them to create some neat effects. The most important aspect is that they are animatable.

[33] The NSView class acts mainly as an abstract superclass; generally you create instances of its subclasses, not of NSView itself. NSView provides the general mechanism for displaying content

CALayer on the view) for each image (we can drag an image file from anywhere to the LightTable's view). We thought about exploiting this mechanism to manage images and text with CALayers. In order to make images react to user events like resizing or dragging, Core Animation layers provide methods to be aware of mouse clicks. Moreover, Core Animation is a very optimized framework with high performance. However, it soon became clear that this approach would have created lots of problems: representing text boxes and editing texts in a CALayer is not the simplest thing to do. Therefore we decided to discard this solution.

2. In the second solution we thought about substituting Core Animation layers with views. Cocoa provides a lot of custom views, like buttons, labels, text fields, and so on. For our purposes, we found two important views: the NSImageView, and the NSTextField. With NSImageView we can easily add an image to a view and with NSTextField we can add text. Moreover, since both these classes extend NSResponder[34] class, they manage mouse and keyboard events more easily than Core Animation layers do. This is an advantage for the programmer, but using this strategy would have led to another failure. In fact the problem was finding a way to manage more than one of these objects at the same time. In particular, in many graphical editors we can draw with the mouse a rectangular area to allow multiple selections of objects. We did not find an easy way to approach this problem with image views and text views, because every view manages mouse events on its own. When drawing the selection area it would have been difficult to make the subviews communicate each other information about the area being drawn. Moreover, smart editors show a mask on graphic objects when the user clicks on them, to communicate him which objects are selected (see figure 6.1). We did not find an easy way to implement this feature: we thought about adding another custom view representing the selection mask whenever the user clicked on a graphical object, but the problem was how to draw the selection handles inside the overlapped custom view and without hiding image or text.

FIGURE 6.1 - EXAMPLE OF OBJECT WITH SELECTION MASK

3. The third solution is basically managing all the graphic objects with custom drawing. In other words this means that everything in the

on the screen and for handling mouse and keyboard events, but its instances lack the ability to actually draw anything. If the application needs to display content or handle mouse and keyboard events in a specific manner, we will need to create a custom subclass of NSView.

[34] NSResponder is the base class used to handle events, and in particular keyboard and mouse events

custom NSView is drawn with AppKit[35] framework resources. This strategy is different from the previous ones, because we do not add objects to our basic custom view (like CALayer or subviews), but we *draw* objects that have to be displayed directly using the drawing method `drawRect:`[36] of NSView. In order to do this, we got ideas from Sketch, an Apple example application that permits drawing geometric shapes such as lines, circles, ovals, but also texts. It also lets the user dragging images in the canvas from elsewhere. Using this solution we managed to plan and design all the features listed before.

6.3.1 MODEL

After deciding which pattern to use in order to construct the application, we went to the real design phase, and we started from the application model. Since the canvas would have had a custom view (subclass of NSView) to draw all of its contents, we needed to find a way to represent objects to draw. Moreover, we had also to store and keep track of the properties of the objects to draw. Therefore we decided to construct a set of classes to represent the objects to be drawn and to store their properties, like position, dimensions, name, and stuff like that. We organized these objects in a hierarchical mode, designing a base class to hold common properties of every kind of object, and then some other subclasses to represent the specific properties of the several objects that would be rendered in the canvas. These classes must also provide logic for drawing the object they represent. Although this feature could seem wrong (model objects should not deal with views at all), this is quite MVC compliant: our application is strongly oriented to graphic drawing, and in this case our data (model part of the application) is represented by position, dimensions of objects, texts shown in the view and stuff like that. Since we have to represent only images and texts, we designed the following classes:

- **ALTGraphic**: this is the base class; it must hold information about position and dimensions. It must provide logic to draw selection handles on the object as well; it will also declare methods to draw its contents in the view, but we delegated these actions to the subclasses. Remembering that the final goal of the editor is generating a property list to deliver to AppKid, this class has also to provide logic to translate its properties in a property list representation.
- **ALTText**: this class extends ALTGraphic and must provide properties to hold information about the text that it represents. Moreover, ALTText class must provide logic to edit text it represents using keyboard events. Obviously this subclass will have to implement logic to convert its properties (first of all its text contents) in a property list representation.

---

[35] The AppKit framework is a set of classes, first developed by NeXT Computer, and now by AppleComputer, for rapidly developing applications. The majority of AppKit classes deal with user interface elements, such as windows, buttons, and menus.

[36] For more details about custom drawing in Cocoa, see Appendix.

- **ALTImage:** this class extends ALTGraphic and represents an image. Therefore it must provide a property to hold the image it represents. It will also provide logic to draw the image in the view and to convert itself into a property list representation.

6.3.2 VIEW

In this phase we isolated the major features the GUI should have:

- A subview representing the current page of the book to create (the so-called canvas), and the space around it. In the interactive book in the iPad there could be objectS not immediately visibile because they have coordinates that place themselves outside the screen. We have to be able to do this also with our editor.
- Buttons to add images.
- Buttons to add texts.
- Buttons to add movies.
- Buttons to add and remove pages and to navigate across them.
- Removing objects.
- An inspector to manage positions and dimensions of objects in the page view.
- An inspector to decide which properties to animate about a particular object.
- An inspector to decide if an object is draggable and the drag area.
- Drag of objects around the page view with the mouse ((to decide their future location in the book page).
- Live sizing of objects in the page view with the mouse.
- Multiple selections of objects in the page view.
- Zooming on the canvas.

In the following picture we can see the concept of the document editing view:

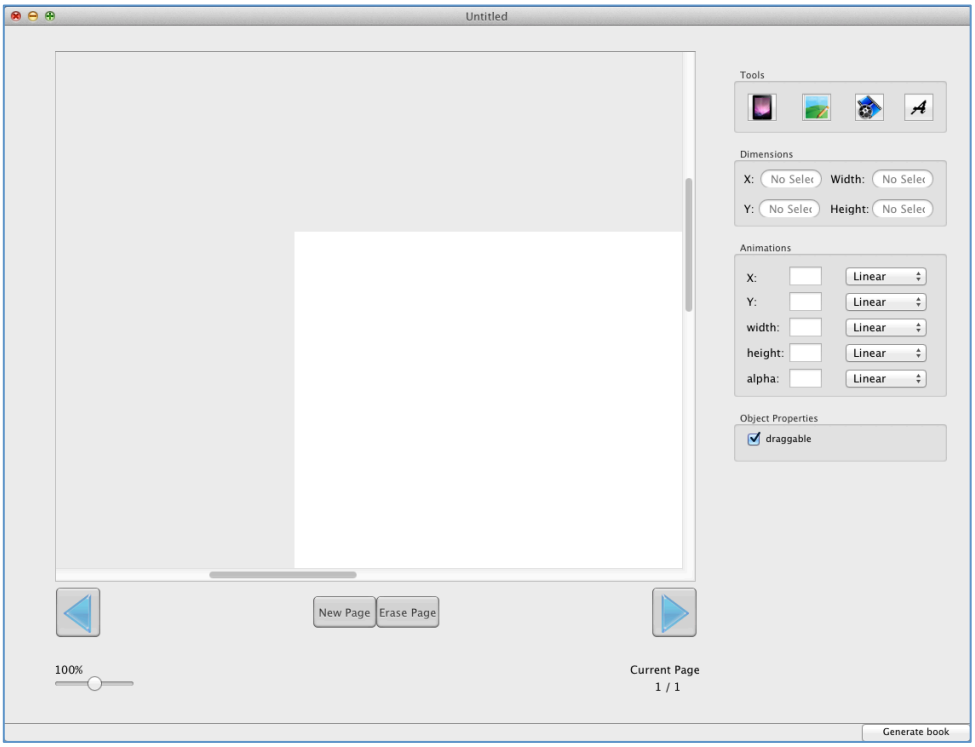

FIGURE 6.2 - CONCEPT OF DOCUMENT VIEW

In the right column we placed tools to add elements, inspectors to control position, dimensions, and animated properties of each object. We also placed a check box to distinguish if an object is draggable or not. We can see the scroll view with the white canvas representing the screen of the iPad inside (with dimensions 1024x768). The grey area represents the zone around the screen and its dimensions are 3072x2304 (three times the dimension of the canvas). Here we can place objects that will be situated outside the screen of the iPad during the execution of the interactive book. The white canvas with the grey area around will be an instance of the custom view class ALTEditorView and will be set as subview of the scroll view. This special view will render the graphic objects on the screen querying them from the model layer of the application. In addition this class is designed to respond to and manage mouse and keyboard events. Note that this is only the design of the GUI. We did not manage to implement all of these features. We will see in 6.4 what we actually implemented.

RENDERING CONTENTS IN THE VIEW

As we said before, the pattern used for displaying graphical objects is not based on *adding* objects (like Core Animation Layers or NSViews), but rather on *drawing* them using their properties. This means that in order to display an image in the white canvas we do not have to create an NSImageView and then add it as a subview of the ALTEditorView. This would not give us enough freedom to implement all the features describe above. The idea is to keep a list of graphic objects currently rendered in the canvas (the model of the application) in the controller tier of the application and to query this list

whenever the view needs to be updated. This pattern is very powerful: the view has no knowledge of the objects it is displaying and it delegates their maintenance to the controller. This is perfectly MVC compliant, and allows keeping the several components of the applications separated, leading to the advantages seen in 3.2.3. Thus, we designed an algorithm to draw the contents of the ALTEditorView (i.e. the white canvas with the grey area around it):

```
start;
graphics[] <- get graphic objects array from the controller;
for (i <- 0, i < graphics.size, i++) {
    current_graphics <- graphics[i];
    if (current_graphics.isSelected) {
        draw graphics in the view with selection handles;
    } else {
        draw graphics in the view without selection handles;
    }
}
end;
```

ALGORITHM 6.1 - RENDERING CONTENTS IN ALTEDITORVIEW

The algorithm starts retrieving the list of objects to be rendered. After that it scans the list and it simply draws the current object; if the current object is selected, then it will be rendered with the selection mask seen in figure 6.1.

HANDLING MOUSE EVENTS

Another interesting problem for the ALTEditorView class is handling mouse and keyboard events in order to manage selections, objects dragging, objects resizing and text editing. The algorithm must be able to manage all the possible situations given by the different types of mouse and keyboard events sent to the ALTEditorView. The following pseudocode illustrates the algorithm, assuming that it receives an event when the user clicks somewere in the graphic view.

```
start;
selection <- NULL;
clickedGraphicIsSelected = FALSE;
editingGraphic = FALSE;
clickedOnHandle = FALSE;
movingGraphics = FALSE;
marqueeSelect = FALSE;
if (event.type == MOUSE_EVENT) {
    initialMouseLocation = event.point;
    if (click_count > 1) {
        graphic <- get graphic under event.clickPoint;
        if (graphic.type == text) {
            start editing graphic;
            editingGraphic = TRUE;
            listen for keyboard events;
        }
}
else {
    graphic <- get graphic under event.clickPoint;
    clickedGraphicIsSelected <- graphic.isSelected;
    if (graphic != null) {
        if (clicked on resizing handle) {
            start resizing graphic;
            clickedOnHandle = TRUE;
            wait for mouse dragged events;
        }
        else {
            if (shift key was pressed) {
                if (clickedGraphicIsSelected == TRUE) {
                    selection.remove(graphic);
                    clickedGraphicIsSelected = FALSE;
                }
                else {
                    selection.add(graphic);
                    clickedGraphicIsSelected = TRUE;
                }
            }
            else {
                selection <- graphic;
                clickedGraphicIsSelected = TRUE;
            }
        }
        if (clickedGraphicIsSelected == TRUE) {
                start moving graphics selection;
                movingGraphics = TRUE;
                wait for mouse dragged events;
        }
    }
    else { // in this case user clicked on no graphics
            start tracking mouse dragging to select multiple objects;
        marqueeSelect = TRUE;
    }
}
end;
```

ALGORITHM 6.2 – ALGORITHM USED TO MANAGE MOUSE AND KEYBOARD INPUT

The algorithm starts initializing some Boolean variables that will be used to distinguish what kind of action is going to be taken in response to the user click. If the number of clicks of the user is more than 1, then this means that he wants to edit something. The only editable object is text, so the algorithm retrieves the graphic object under the click location (if there is one) and it checks if the object is a text. In this case it starts a procedure to edit text using keyboard events. Conversely, if the number of clicks is equal to one, the algorithm retrieves the graphic under the click location (if there is one) and manages different scenarios:

- If there is a graphic and the user clicked on a handle, this means that the graphic has been already selected. In this case the Boolean variable corresponding to a resize action is set to TRUE.
- If the click happened with a shift key pressed, then the user is trying to add or remove an object from the set of current selected objects:
    - If the user clicked on a selected object, then this is deselected, and the corresponding variable is set to FALSE;

    - If the user clicked on a deselected object, then it is added to the current set of selected objects; the corresponding Boolean variable is set to TRUE
- If the user clicked on the graphic without the shift key pressed, then this causes the set of current selected objects to be reinitialized with the current graphic; the corresponding variable is set to TRUE.
- In every case in which the graphic under click location is selected (`clickedGraphicIsSelected` set to TRUE), the algorithm sets the `movingGraphics` variable to TRUE and it prepares itself to move the current selection.
- If there were no graphic under the click location, then the `marqueeSelect` variable is set to true and the algorithm starts drawing the selection box.

The following algorithm is executed whenever the user moves the mouse with the left button pressed (drag action). It represents the second part of algorithm 6.2. In this case it works in response to mouse drag events sent to ALTEditorView instance.

```
start;
if (clickedOnHandle == TRUE) {
    resize graphic using handle and event.point;
    update display;
}
else if (movingGraphics == TRUE) {
    move graphics to point: event.point;
    update display; // invokes algorithm 6.1
}
else if (marqueeSelect == TRUE) {
    figure out a new selection rectangle using initalMouseLocation and
        event.point;
    add all the graphics intersecting the new selection rectangle to
        selection;
    update display; // invokes algorithm 6.1
}
end;
```

ALGORITHM 6.3 - ALGORITHM EXECUTED IN RESPONSE OF MOUSE DRAG EVENT

We can see that, based on the value of the previous Boolean variables, the algorithm does several mutual exclusive actions:

- Resize graphic using mouse dragged events and update display.
- Move graphics using mouse dragged events and update display.
- Continue drawing the selection box, insert in the current selection set graphics that intersect it and update display.

Note that “update display” causes algorithm 6.1 to be used.

The following algorithm is executed when the user releases the left button. It simply resets all the Boolean variables, in order to make things work.

```
start;
editingGraphic = FALSE;
clickedOnHandle = FALSE;
movingGraphics = FALSE;
marqueeSelect = FALSE;
update display; // invokes algorithm 6.1
end;
```

ALGORITHM 6.4 - ALGORITHM EXECUTED IN RESPONSE OF MOUSE UP EVENT

The view reacts to keyboard events in several moments:

1. Arrow keys pressed: in this case the current selected objects are moved in the same direction of the arrow and then the display is updated.
2. Letter keys pressed: if `editingGraphics` variable is set to TRUE, then there is a text object in editing mode. In this case the view uses the keyboard events to construct the text.
3. Delete key pressed: This event deletes the current selected objects and updates display; if there is a text being edited, it deletes the last typed character.

6.3.3 CONTROLLER

For the controller part of the application architecture, we decided to centralize the control logic in a class called ALTDocument. This class is automatically generated by the document-based application template, and can act both as Model and as Controller for the View we designed. In a first time we decided to put the controller logic in a subclass of NSWindowController, but this created lots of problems when linking outlets and actions[37] from the Interface Builder to the NSWindowController's subclass. These problems were caused by the File's Owner specification of the nib file. The File's Owner of the document's editing xib file is the ALTDocument class, which is also delegated to serialization of data when saving a document. Changing the File's Owner to the NSWindowController's subclass would have been permitted to isolate the controller logic, but it caused a bad behaviour when saving a document. Thus, instead of wasting time on looking for the bug inside this solution, we decided to put all the controller logic in the ALTDocument class. Moreover, a look at Sketch's architecture confirmed our design: Sketch centralizes the controller logic in the SKTDocument class (the equivalent of ALTDocument). Summarizing, ALTDocument manages:

- The list of graphic objects of the current page to be rendered in the ALTEditorView (instance of the model of the application architecture);
- The list of pages being constructed and edited (instance of the model of the application architecture);
- The number of pages and the current page number (instance of the model of the application architecture);
- The current tool selected in the GUI.

[37] For more details about communicating with objects in Cocoa, see Appendix.

- The actions triggered when clicked on a button (application logic):
    - Adding an image;
    - Adding a text;
    - Adding a movie;
    - Adding a new page;
    - Removing a page;
    - Going to the next page;
    - Going to the previous page;
    - Generating the XML property list file.

There is another controller class used by the view and linked to the ALTDocument: it is the NSArrayController[38] class. This class, added to the xib file of the document-editing window, is bound[39] to the graphic objects list of the ALTDocument. ALTEditorView is in turn bound to the array controller and uses it to manage the selection of the graphic objects and to draw the selection mask around them.

In the following picture we can see the MVC architecture of the AppKidEditor design.

[38] NSArrayController is a bindings compatible class that manages a collection of objects. Typically the collection is an array, however, if the controller manages a relationship of a managed object the collection may be a set. NSArrayController provides selection management and sorting capabilities.

[39] For more details about bindings, see Appendix ("Communicating with objects in Cooca" paragraph).

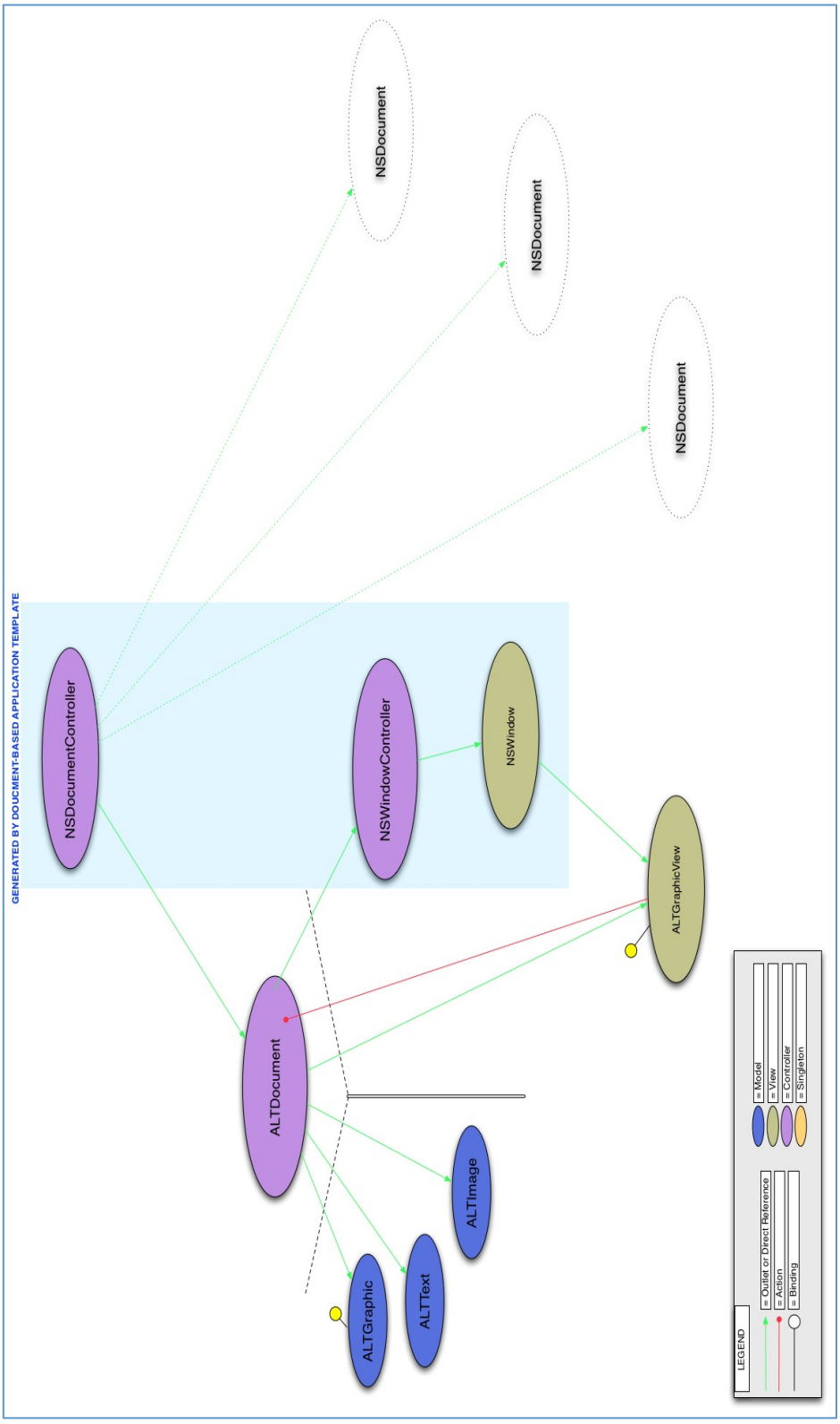

FIGURE 6.3 - MVC ARCHITECTURE OF APPKID EDITOR

6.4 IMPLEMENTATION

In this chapter we give a detailed explanation of all the solutions used to implement the algorithm and the design architecture discussed in the last chapter.

6.4.1 IMPLEMENTATION OF THE MODEL

We implemented the model classes in order to provide them all the properties they need to represent a graphical object. We provided these classes with a method to draw the object they represent in the ALTEditorView, called `drawContentsInView:isBeingCreatedOrEditing`. In this way drawing objects is not an issue of the ALTEditorView: the implementation of the drawing algorithm will invoke this method on the current graphic object, delegating to it the problem of rendering itself. In the following list we will explain the particular solutions used for each model class:

- **ALTGraphic**: in order to hold information about position and dimensions we created a NSRect[40] property called `bounds`, synthesized with an instance variable[41]. We also added four properties, two for the position and two for the dimensions, without synthesizing them with an instance variable. The getters and setters of these properties deal with the NSRect (in other words the getter of the x coordinate property returns the x variable of `bounds`) and some objects in the view are bound to them (as we will see in 6.4.2). We had to use this mechanism because NSRect is not an object and it is not Key-Value Observing compliant[42].
  Moreover, the class provides methods to translate its position, to draw selection handles, to return the selection handle the user clicked on (handles are represented as an enum type), to encode and decode its properties[43].
  A method was also created in order to return a property list compliant representation of the graphic object. The method, called `propertyListRepresentation`, creates a dictionary, and fills it with some entries. Each entry has key equal to a string that describes the name of property, and value equal to the property described by the string. For example, for the width of the graphic object the method constructs an entry with key “width” and value equal to the width of the graphic object, retrieved from the NSRect property.

---

[40] NSRect is a C struct of the Foundation framework representing a rectangle and its position.

[41] Properties are kind of virtual member variables. Synthesizing a property means creating the related getter and setter methods. They can be synthesized automatically (with the @synthesize directive) with real instance variables. In this case the getters returns the instance variable and setters updates the instance variable.

[42] Once again, for more details about Key-Value Observing, see Appendix (“Communicating with objects in Cocoa” paragraph).

[43] For more details about archiving, see Appendix.

At the end of the method, the dictionary is returned. It will be used to create the property list representing the book. In the following table we can see the properties coded in the dictionary.

KEY	VALUE
"x"	bounds.origin.x
"y"	bounds.origin.y
"width"	bounds.size.width
"height"	bounds.size.height

We had to pay particular attention when creating entries with key "x" and "y". These entries must represent the x and y coordinates of the object in the reference system of the white canvas. On the other side, objects track coordinates in the reference system of the grey area that contains the white canvas. This is true because mouse events carry information about position in the ALTEditorView instance[44]. For example placing an image on the upper left corner of the white canvas would have coordinates (1024, 768) in the editorView reference system, assuming that the origin is always in the upper left corner. Obviously we want to deliver to the framework the same object, but having coordinates equal to (0, 0). The last pair of coordinates is expressed in the canvas coordinate system (the same of the iPad screen). Therefore every time we generate entries with "x" and "y" keys we must transform the x and y properties of the object accordingly to this formula:

$$x_{out} = x - workingAreaX$$
$$y_{out} = y - workingAreaY$$

where

$$workingAreaX = \frac{editorViewWidth - iPadScreenWidth}{2}$$
$$workingAreaY = \frac{editorViewHeight - iPadScreenHeight}{2}$$

[44] Recall that editorView, the instance of ALTEditorView, is made of the whole grey area with the white canvas inside it and a click in editorView generates an event with position relative to this big area.

Roughly speaking, *workingAreaX* and *workingAreaY* are the coordinates of the white canvas in the reference system of editorView. Width and height are not affected by the change of reference system, so they do not need to be transformed.

- **ALTImage:** as already said, this class extends ALTGraphic and represents an image. It has a property called `contents`, of type NSImage, which holds data about the image to be rendered on the screen. There is another property, `imageName` used to hold the file name of the image. The "constructor", called `initWithPosition:contents:` initializes the class with a given image.
  The class overwrites `drawContentsInView:isBeingCreatedOrEditing:` method of ALTGraphic. In this method we use the `bounds` property of the superclass (i.e. ALTGraphic) to draw the image in the view passed as parameter (which will be an instance of ALTEditorView) in the right position and with the right dimensions.
  The class also overwrites the methods for encoding and decoding its properties. These methods start invoking the superclass' method (so it starts encoding the superclass' properties) and then encode the properties of its class.
  Also `propertyListRepresentation` has been overwritten. It starts invoking the superclass' method to initialize the dictionary to be returned. After this invocation the dictionary will contain key-value entries for the properties of the superclass. After that the method inserts the following key-value entries, in the same manner seen in ALTGraphic class:

KEY	**VALUE**
"filename"	imageName
"type"	"image"
"alpha"	1

In this implementation we did not gave the possibility to the user to set the transparency of the image, so we created by default an alpha value equal to 1. "fileName" is used by the framework to find the image to be loaded in the pageSprite; "type" is used by the framework to distinguish the type of object to add in the pageSprite, as we have already seen in algorithm 5.2.

- **ALTText**: the class extends ALTGraphic, as ALTImage does. It has a property of type NSTextStorage that holds the text the class has to render on the screen. The class overwrites `drawContentsInView:isBeingCreatedOrEditing:` method to draw the text in the ALTEditorView instance.
  The class also provides methods to create an NSTextView instance when the user wants to edit the text (i.e. when he double-clicks on it). Obviously, the class has methods to encode and decode its properties and it overwrites `propertyListRepresentation` in order to insert the following key-value entries in the dictionary to be returned:

KEY	VALUE
"contents"	contents.mutableString
"type"	"text"
"fontName"	contents.font
"fontSize"	contents.fontSize

  The first entry is used by the framework to load the text in the pageSprite, the last twos are used to adjust the font.

6.4.2 IMPLEMENTATION OF THE VIEW

ALTEditorView implements the algorithms seen in 6.3.2. Our application architecture mixes the classic MVC with the so-called direct view and data model binding. As already mentioned in the design section, there is an array controller in the xib file of the document-editing view (called ALTDocument.xib). The array controller acts as a broker between the ALTDocument class and the ALTGraphic view for the selection management. In fact the array controller has a useful function: it holds a property of type NSIndexSet that is used to track the currently objects selected in the "controlled" array. Moreover it has useful methods to get and set the index set. ALTEditorView instance is bound to the index set of the array controller (this type of binding has been made programmatically, not through interface builder). In this way whenever some new object is selected or deselected by the user, the correspondent method to change the selection indexes of the array controller is invoked from the editorView. Since ALTEditorView instance is bound to the array controller, this method is able to directly modify the selection indexes set of the array controller. Whenever the display is updated,

the ALTEditorView retrieves the index set that is bound to and uses it to draw the selection handles around selected objects.

Bindings are also used to implement the inspectors shown in the right column (see the concept of the document view in figure 6.2) of the document view. For example the text fields of the Dimensions inspector are bound to the properties of the current selected object in the array controller. In this way the text fields show the dimensions of the current selected object, and get updated in real time whenever the user drags or resizes the selected object. With this mechanism setting the position and the dimension from the text field is possible too. If there is more than one object selected, then the fields automatically show the special message "multiple values" instead of the coordinates. However setting the position from the inspector is still possible. The same idea was designed for the Animation inspector, but it has not been implement yet.

6.4.3 IMPLEMENTATION OF THE CONTROLLER

The ALTDocument class implements all the actions described in 6.3.3 as IBActions[45]. It has an outlet that refers to the ALTEdtorView instance, called `editorView`. Moreover, there is a property of type NSMutableArray called `currentPageGraphics`, which lists the graphics of the current page to be rendered on the screen and which the array controller is bound to. There is also another NSMutableArray called `pages`, which lists the pages of the book. The i-th element of pages is an array of graphic objects, representing the graphics of i-th page. In algorithm 6.1 we said that the ALTEditorView retrieves the list of graphical objects from the controller. This is achieved via the delegation pattern[46]. More specifically, the ALTEditorView interface declares a protocol called `ALTEditorViewDelegate` with a method called `graphicsForEditorView`. ALTDocument implements this protocol: `graphicsForEditorView` simply returns the `currentPageGraphics` array. Following this pattern, in the `awakeFromNib` method of ALTDocument, the delegate of `editorView` is set to `self`. In this way when the ALTEditorView instance will need the array of graphic objects, it will simply send the `graphicsForEditorView` message to its delegate to retrieve it. Another important step done in the `awakeFromNib` method is binding the editorView to the selection indexes property of the array controller (ALTDocument instance has an outlet to the array controller).

ALTDocument implements the methods used to archive its model (in order to realize file saving and loading); the two methods are:

- `dataOfType:error:`: this method start archiving the pages array; NSArray has methods to serialize itself and that causes the objects it

[45] For more details about IBActions, see Appendix ("Communicating with objects in Cocoa" paragraph).

[46] For more details about delegation, see Appendix ("Communicating with objects in Cocoa" paragraph).

contains to be serialized. Using the methods depicted in 6.4.1 to encode their properties, the objects contained in each page array are serialized. After this chain reaction, in which every object forces its properties to be serialized, the result is returned by the method as raw data. When the user clicks on the "Save" button, the ALTDocument instance invokes this method to serialize data and then stores it automatically in the file system.

- `readFromData:ofType:error:`: this method is invoked when the user clicks on the "Load" button; in fact it starts deserializing raw data stored in the file system using the file name chosen by the user. The methods mentioned in 6.4.1 to decode objects' properties are invoked to deserialize each object in the pages array. After this process we will have the pages array restored as before saving; The method ends by setting `pages` variable to this recovered array. The method also sets a flag to TRUE, in order to distinguish that the data model has been loaded from file. If this flag is set to TRUE, `awakeFromNib:` method avoids initializing `pages` property.

7 CONCLUSIONS

In this project we designed and implemented a framework in order to create interactive books without programming them from scratch, but rather editing an XML file that describes its geometry, it graphical representation, its animations and the interactions with the user. Moreover, we created also an editor, in order to edit the XML file in a WYSIWYG[47] manner. With the possibilities to speed up the process of creation of interactive books, Altera and Ware's Me company decided to produce a series of interactive books and to sell them on the Apple iOS App Store. Nowadays I am using the framework and the editor to develop two other interactive book, called "Oswald" and "Aliens". Obviously not all the features listed in chapter 4 have been implemented and we decided to use the framework to realize only the already implemented ones. The idea is to hardcode in the interactive book application the features that are not provided by the framework. We will see in the next paragraph what remains to do in order to give to the framework and to the editor the maximum expressiveness.

7.1 FUTURE WORK

As already said there are some features not yet implemented:

- **FRAMEWORK:**
  - Make objects draggable: this is the most important feature to develop in the future.
  - Accomplish the development of the triggering system.
  - Puzzle game.
  - Filling game.
  - Showing credits to the user.
  - Choosing the language of the book.
- **EDITOR**
  - Adding sounds
  - Specifying animations of objects in a page
  - Specifying triggering of objects
  - Specifying draggable objects.

These are the future goals of the work in Altera. We estimated that when these goals will be achieved, we would be able to produce an interactive book in some days instead of several weeks of programming, leading to a huge production speed. Note that with the results already obtained, we are able to create the skeleton of an interactive book (i.e. adding static images and texts) with the editor in few hours. We are also able to insert animations and make

[47] WYSIWYG stands for "What You See Is What You Get". The term is used in computing to describe a system in which content (text and graphics) displayed onscreen during editing appears in a form closely corresponding to its appearance when printed or displayed as a finished product (WYSIWYG).

objects react to user touches in some other few hours editing the XML property list file. This is already an important result, and we estimated that it allows saving something like a week of work. We also estimated that, in order to create an interactive book from scratch, we need about three weeks. Thus, we can save the 33% of time for each book.

The last important phase will be selling the editor and the framework on the App Store, allowing anyone who wants to develop an interactive book to do this in a completely free manner.

8 APPENDIX

8.1 PROPERTY LISTS

A property list is a structured data representation used by Cocoa and Core Foundation as a convenient way to store, organize, and access standard types of data (About property lists). It is colloquially referred to as a "plist." Property lists are used extensively by applications and other software on Mac OS X and iOS.

They are based on an abstraction for expressing simple hierarchies of data. The items of data in a property list are of a limited number of types. Some types are for primitive values and others are for containers of values. The primitive types are:

- Strings
- Numbers
- Binary data
- Dates
- Boolean values.

The containers are:

- Arrays
- Dictionaries

The containers can contain other containers as well as the primitive types. Thus we might have an array of dictionaries, and each dictionary might contain other arrays and dictionaries, as well as the primitive types. A root property-list object is at the top of this hierarchy, and in almost all cases is a dictionary or an array. Note, however, that a root property-list object does not have to be a dictionary or array; for example, we could have a single string, number, or date, and that primitive value by itself can constitute a property list.

From the basic abstraction derives both a static representation of the property-list data and a runtime representation of the property list. The static representation of a property list, which is used for storage, can be either XML or binary data. (The binary version is a more compact form of the XML property list.) In XML, each type is represented by a certain element. The runtime representation of a property list is based on objects corresponding to the abstract types. The objects can be Cocoa or Core Foundation objects. The following table lists the types and their corresponding representations.

Abstract Type	XML element	Cocoa Class
array	`<array>`	NSArray
dictionary	`<dict>`	NSDictionary
string	`<string>`	NSString
data	`<data>`	NSData
date	`<date>`	NSDate

number - integer	<integer>	NSNumber (intValue)
number - floating point	<real>	NSNumber (floatValue)
boolean	<true/> or <false/>	NSNumber

By convention, each Cocoa object listed in the table is called a property-list object. Conceptually, one can think of "property list" as being an abstract superclass of all these classes. If some method or function returns a property list object, then it must be an instance of one of these types, but a priori one may not know which type. If a property-list object is a container (that is, an array or dictionary), all objects contained within it must also be property-list objects. If an array or dictionary contains objects that are not property-list objects, then the hierarchy of data cannot be saved or restored using the various property-list methods and functions. Although NSDictionary objects allow their keys to be objects of any type, if the keys are not string objects, the collections are not property-list objects.

A property list can be stored in one of three different ways: in an XML representation, in a binary format, or in an "old-style" ASCII format. The programmer can serialize property lists in the XML and binary formats. The serialization API with the old-style format is read-only. XML property lists are more portable than the binary alternative and can be manually edited, but binary property lists are much more compact; as a result, they require less memory and can be read and written much faster than XML property lists. In general, if our property list is relatively small, the benefits of XML property lists outweigh the I/O speed and compactness that comes with binary property lists. If there is a large data set, binary property lists are a better solution.

8.2 DOCUMENT-BASED APPLICATIONS

A document-based application is one of the more common types of applications. It provides a framework for generating identically contained but uniquely composed sets of data that can be stored in files. Word processors and spreadsheet applications are two examples of document-based applications. Document-based applications do the following things:

- Create new documents.
- Open existing documents that are stored in files.
- Save documents under user-designated names and locations.
- Revert to saved documents.
- Close documents (usually after prompting the user to save edited documents).
- Print documents and allow the page layout to be modified.
- Represent data of different types internally.
- Monitor and set the document's edited status and validate menu items.
- Manage document windows, including setting the window titles.
- Handle application and window delegation methods (such as when the application terminates).

Three Application Kit classes provide architecture for document-based applications, called the document architecture, that simplifies the work developers must do to implement the features listed above. These classes are NSDocument, NSWindowController, and NSDocumentController. Objects of these classes divide and orchestrate the work of creating, saving, opening, and managing the documents of an application. They are arranged in a tiered one-to-many relationship, as depicted in Figure 1. An application can have only one NSDocumentController, which creates and manages one or more NSDocument objects (one for each New or Open operation). In turn, an NSDocument object creates and manages one or more NSWindowController objects, one for each of the windows displayed for a document. In addition, some of these objects have responsibilities analogous to NSApplication and NSWindow delegates, such as approving major events like closing and quitting.

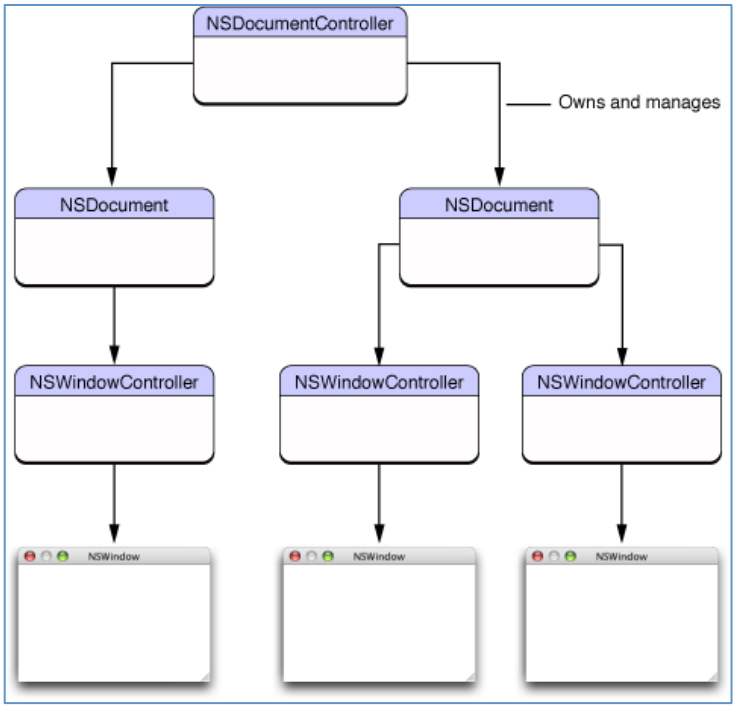

FIGURE 8.1 - ARCHITECTURE OF DOCUMENT-BASED APPLICATIONS

Conceptually, a document is a container for a body of information identified by a name under which it is stored in a disk file. In this sense, however, the document is not the same as the file but is an object in memory that owns and manages the document data. In the context of the Application Kit, a document is an instance of a custom NSDocument subclass that knows how to represent internally, in one or more formats, persistent data that is displayed in windows. A document can read that data from a file and write it to a file. It is also the first-responder target for many menu commands related to documents, such as Save, Revert, and Print. A document manages its window's edited status and is set up to perform undo and redo operations. When a window is closing, the Application Kit first asks the document, before it asks the window delegate, to approve the closing.

8.3 COMMUNICATING WITH OBJECTS IN COCOA

With Cocoa and Objective-C one way of adding the behavior that is specific to our program is through inheritance. We create a subclass of an existing class that either augments the attributes and behavior of an instance of that class or modifies them in some way. But there are other ways of adding the special logic that characterizes our program. There are other mechanisms for reusing and extending the capabilities of Cocoa objects (Communicating with Objects). The relationships between objects in a program exist in more than one dimension. There is the hierarchical structure of inheritance, but objects in a program also exist dynamically, in a network of other objects that must communicate with one another at runtime to get the work of the program done. In a fashion similar to a musician in an orchestra, each object in a program has a role, a limited set of behaviors it contributes to the program. It displays an oval surface that responds to mouse clicks, or it manages a collection of objects, or it coordinates the major events in the life of a window. It does what it is designed to do, and nothing more. But for its contributions to be realized in the program, it must be able to communicate them to other objects. It must be able to send messages to other objects or be able to receive messages from other objects.

Before our object can send a message to another object, it must either have a reference to it or have some delivery mechanism it can rely on. Cocoa gives objects many ways to communicate with each other. These mechanisms and techniques make it possible to construct robust applications efficiently. They range from the simple to the slightly more elaborate, and often are a preferable alternative to subclassing. We can configure them programmatically and sometimes graphically in Interface Builder

8.3.1 OUTLETS

An outlet is an object instance variable - that is, an instance variable of an object that references another object. With outlets, the reference is configured and archived through Interface Builder. The connections between the containing object and its outlets are reestablished every time the containing object is unarchived from its nib file. The containing object holds an outlet as an instance variable with the type qualifier of IBOutlet. For example:

```
@interface AppController : NSObject {
    IBOutlet NSArray *keywords;
}
```

LISTING 8.1 - EXAMPLE OF OUTLET

Because it is an instance variable, an outlet becomes part of an object's encapsulated data. But an outlet is more than a simple instance variable. The connection between an object and its outlets is archived in a nib file; when the nib file is loaded, each connection is unarchived and reestablished, and is thus always available whenever it becomes necessary to send messages to the other object. The type qualifier IBOutlet is a tag applied to an instance-variable

declaration so that the Interface Builder application can recognize the instance variable as an outlet and synchronize the display and connection of it with Xcode. We connect an outlet in Interface Builder, but we must first declare an outlet in the header file of our custom class by tagging the instance variable with the IBOutlet qualifier.

An application typically sets outlet connections between its custom controller objects and objects on the user interface, but they can be made between any objects that can be represented as instances in Interface Builder, even between two custom objects. As with any instance variable, the programmer should be able to justify its inclusion in a class; the more instance variables an object has, the more memory it takes up. If there are other ways to obtain a reference to an object, such as finding it through its index position in a matrix, or through its inclusion as a function parameter, or through use of a tag (an assigned numeric identifier), we should do that instead.

Outlets are a form of object composition, which is a dynamic pattern that requires an object to somehow acquire references to its constituent objects so that it can send messages to them. It typically holds these other objects as instance variables. These variables must be initialized with the appropriate references at some point during the execution of the program.

8.3.2 DELEGATES

A delegate is an object that acts on behalf of, or in coordination with, another object when that object encounters an event in a program. The delegating object is often a responder object—that is, an object inheriting from NSResponder in AppKit or UIResponder in UIKit—that is responding to a user event. The delegate is an object that is delegated control of the user interface for that event, or is at least asked to interpret the event in an application-specific manner.

To better appreciate the value of delegation, it helps to consider an off-the-shelf Cocoa object such as a text field (an instance of NSTextField or UITextField) or a table view (an instance of NSTableView or UITableView). These objects are designed to fulfill a specific role in a generic fashion; a window object in the AppKit framework, for example, responds to mouse manipulations of its controls and handles such things as closing, resizing, and moving the physical window. This restricted and generic behavior necessarily limits what the object can know about how an event affects (or will affect) something elsewhere in the application, especially when the affected behavior is specific to our application. Delegation provides a way for our custom object to communicate application-specific behavior to the off-the-shelf object.

The programming mechanism of delegation gives objects a chance to coordinate their appearance and state with changes occurring elsewhere in a program, changes usually brought about by user actions. More importantly, delegation makes it possible for one object to alter the behavior of another object without the need to inherit from it. The delegate is almost always a custom object, and by definition it incorporates application-specific logic that the generic and delegating object cannot possibly know itself.

HOW DELEGATION WORKS

The design of the delegation mechanism is simple (Figure 5-1). The delegating class has an outlet or property, usually one that is named delegate; if it is an outlet, it includes methods for setting and accessing the value of the outlet. It also declares, without implementing, one or more methods that constitute a formal protocol or an informal protocol. A formal protocol that uses optional methods—a feature of Objective-C 2.0—is the preferred approach, but both kinds of protocols are used by the Cocoa frameworks for delegation.

In the informal protocol approach, the delegating class declares methods on a category of NSObject, and the delegate implements only those methods in which it has an interest in coordinating itself with the delegating object or affecting that object's default behavior. If the delegating class declares a formal protocol, the delegate may choose to implement those methods marked optional, but it must implement the required ones. Delegation follows a common design, illustrated in figure 8.2:

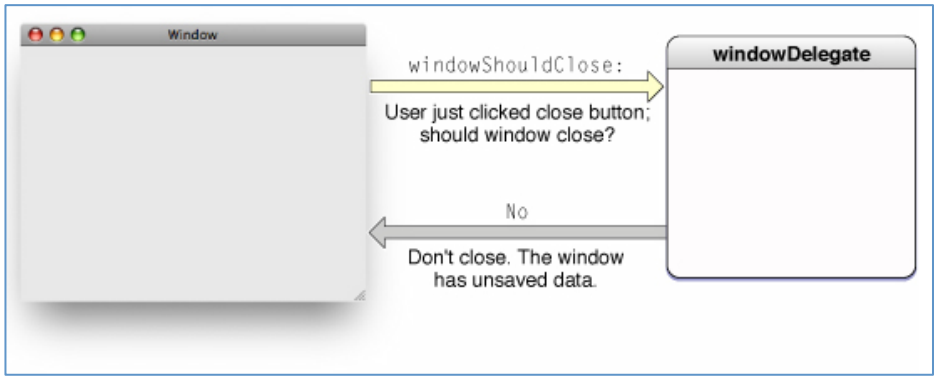

FIGURE 8.2 - THE MECHANISM OF DELEGATION

The methods of the protocol mark significant events handled or anticipated by the delegating object. This object wants either to communicate these events to the delegate or, for impending events, to request input or approval from the delegate. For example, when a user clicks the close button of a window in Mac OS X, the window object sends the `windowShouldClose:` message to its delegate; this gives the delegate the opportunity to veto or defer the closing of the window if, for example, the window has associated data that must be saved (see Figure 8.3).

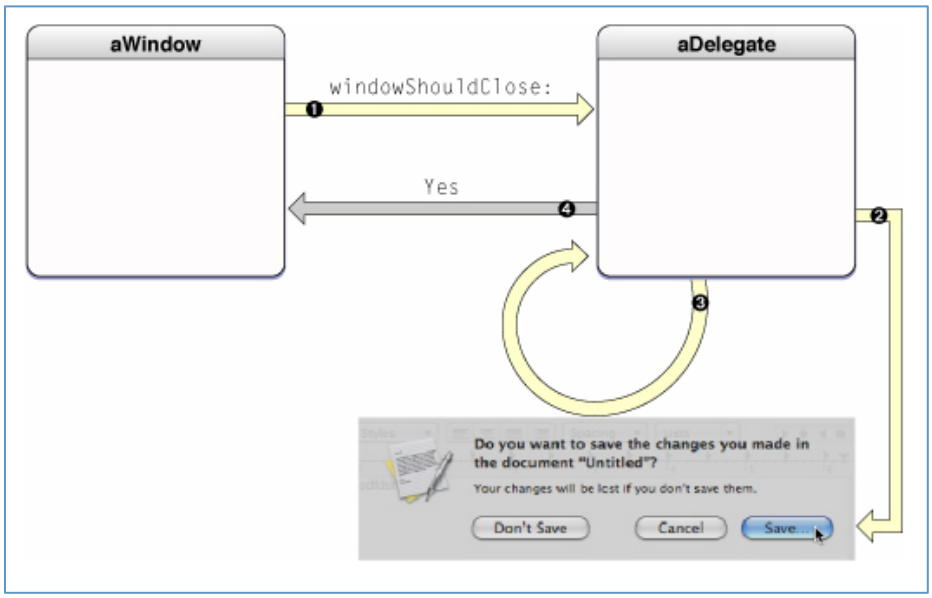

FIGURE 8.3 - A MORE REALISTIC SEQUENCE INVOLVING A DELEGATE

The delegating object sends a message only if the delegate implements the method. It makes this discovery by invoking the NSObject method `respondsToSelector:` in the delegate first.

8.3.3 THE TARGET-ACTION MECHANISM

Although delegation, bindings, and notification are useful for handling certain forms of communication between objects in a program, they are not particularly suitable for the most visible sort of communication. A typical application's user interface consists of a number of graphical objects, and perhaps the most common of these objects are controls. A control is a graphical analog of a real-world or logical device (button, slider, checkboxes, and so on); as with a real-world control, such as a radio tuner, we use it to convey usr intent to some system of which it is a part - that is, an application.
The role of a control on a user interface is simple: it interprets the intent of the user and instructs some other object to carry out that request. When a user acts on the control by, say, clicking it or pressing the Return key, the hardware device generates a raw event. The control accepts the event (as appropriately packaged for Cocoa) and translates it into an instruction that is specific to the application. However, events by themselves don't give much information about the user's intent; they merely tell us that the user clicked a mouse button or pressed a key. So some mechanism must be called upon to provide the translation between event and instruction. This mechanism is called target-action.

Cocoa uses the target-action mechanism for communication between a control and another object. This mechanism allows the control and, in Mac OS X its cell or cells, to encapsulate the information necessary to send an application-specific instruction to the appropriate object. The receiving object - typically an instance of a custom class - is called the target. The action is the

message that the control sends to the target. The object that is interested in the user event - the target - is the one that imparts significance to it, and this significance is usually reflected in the name it gives to the action.

8.3.4 BINDINGS

Bindings are a Cocoa technology that the programmer can use to synchronize the display and storage of data in a Cocoa application created for Mac OS X. They are an important tool in the Cocoa toolbox for enabling communication between objects. The technology is an adaptation of both the Model-View-Controller and object modeling design patternsIt allows us to establish a mediated connection—a binding—between the attribute of a view object that displays a value and a model-object property that stores that value; when a change occurs in the value in one side of the connection, it is automatically reflected in the other. The controller object that mediates the connection provides additional support, including selection management, placeholder values, and sortable tables.

HOW BINDINGS WORK

Bindings arise from the conceptual space defined by the Model-View-Controller (MVC) and object modeling design patterns. An MVC application assigns objects general roles and maintains separation between objects based on these roles. Objects can be view objects, model objects, or controller objects whose roles can be briefly stated as follows:

- View objects display the data of the application.
- Model objects encapsulate and operate on application data. They are typically the persistent objects that our users create and save while an application is running.
- Controller objects mediate the exchange of data between view and model objects and also perform command-and-control services for the application.

All objects, but most importantly model objects, have defining components or characteristics called properties. Properties can be of two sorts: attributes—values such as strings, scalars, and data structures—and relationships to other objects. Relationships can be of two sorts: one-to-one and one-to-many. They can also be bidirectional and reflexive. The objects of an application thus have various relationships with each other, and this web of objects is called an object graph. A property has an identifying name called a key. Using key paths—period-separated sequences of keys—one can traverse the relationships in an object graph to access the attributes of related objects.

The bindings technology makes use of an object graph to establish bindings among the view, model, and controller objects of an application. With bindings we can extend the web of relationships from the object graph of model objects to the controller and view objects of an application. We can establish a binding between an attribute of a view object and a property of a model object (typically through a mediating property of a controller object).

Any change in the displayed attribute value is automatically propagated through the binding to the property where the value is stored. And any internal change in the value of the property is communicated back to the view for display.

For example, Figure 8.4 shows a simplified set of bindings between the displayed values of a slider and a text field (attributes of those view objects) and the number attribute of a model object (MyObject) through the content property of a controller object. With these bindings established, if a user moves the slider, the change in value is applied to the number attribute and communicated back to the text field for display.

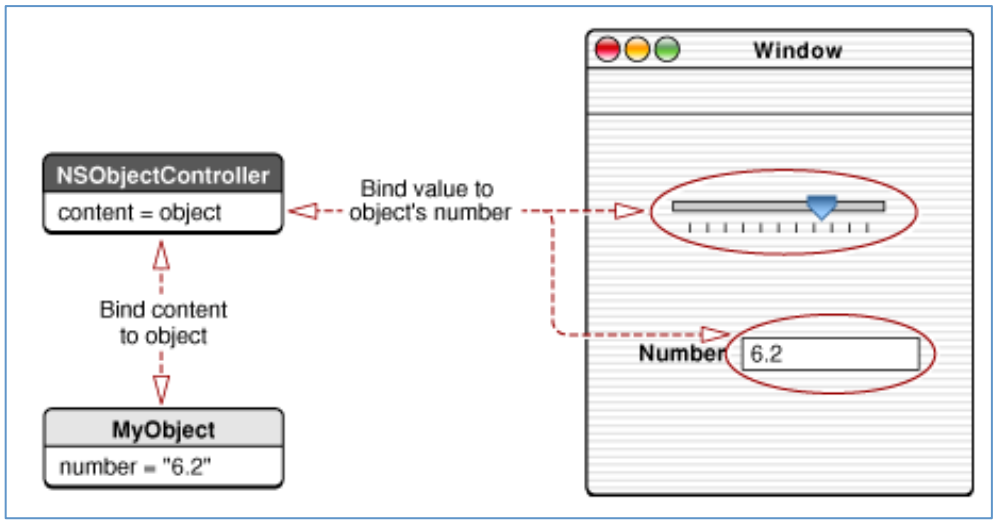

FIGURE 8.4 - BINDINGS BETWEEN VIEW, CONTROLLER, AND MODEL OBJECTS

The implementation of bindings rests on the enabling mechanisms of key-value coding, key-value observing, and key-value binding.

We can establish a binding between any two objects. The only requirement is that the objects comply with the conventions of key-value coding and key-value observing. However, we generally want to establish the binding through a mediating controller because such controller objects offer bindings-related services such as selection management, placeholder values, and the ability to commit or discard pending changes. Mediating controllers are instances of several NSController subclasses; they are available in the Objects & Controllers section of the Interface Builder library. We can also create custom mediating-controller classes to acquire more specialized behavior.

8.3.5 NOTIFICATIONS

The standard way to pass information between objects is message passing—in which one object invokes the method of another object. However, message passing requires that the object sending the message know who the receiver is and what messages it responds to. This requirement is true of delegation messages as well as other types of messages. At times, this tight coupling of

two objects is undesirable—most notably because it would join together what might be two otherwise independent subsystems. And it is impractical because it would require hard-coded connections between many disparate objects in an application.

For cases where standard message passing just will not do, Cocoa offers the broadcast model of notification. By using the notification mechanism, one object can keep other objects informed of what it is doing. In this sense, it is similar to delegation, but the differences are important. The key distinction between delegation and notification is that the former is a one-to-one communication path (between the delegating object and its delegate). But notification is a potentially one-to-many form of communication—it is a broadcast. An object can have only one delegate, but it can have many observers, as the recipients of notification are known. And the object doesn't have to know what those observers are. Any object can observe an event indirectly via notification and adjust its own appearance, behavior, and state in response to the event. Notification is a powerful mechanism for attaining coordination and cohesion in a program.

How the notification mechanism works is conceptually straightforward. A process has an object called a notification center, which acts as a clearing house and broadcast center for notifications. Objects that need to know about an event elsewhere in the application register with the notification center to let it know they want to be notified when that event happens. An example of this is a controller object that needs to know when a pop-up menu choice is made so it can reflect this change in the user interface. When the event does happen, the object that is handling the event posts a notification to the notification center, which then dispatches the notification to all of its observers. Figure 8.5 depicts this mechanism.

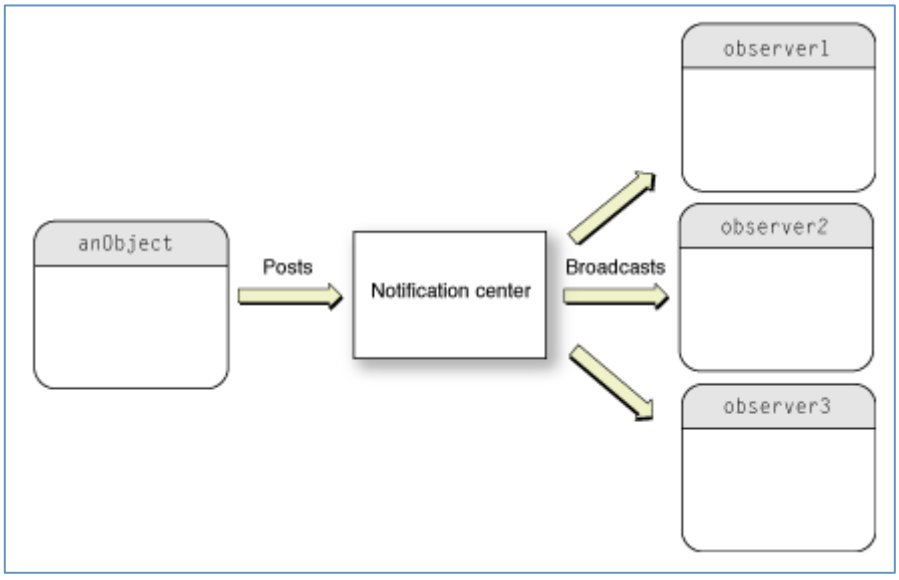

FIGURE 8.5 - POSTING AND BROADCASTING A NOTIFICATION

Any object can post a notification and any object can register itself with the notification center as an observer of a notification. The object posting the

notification, the object that the posting object includes in the notification, and the observer of the notification may all be different objects or the same object. (Having the posting and observing object be the same does have its uses, such as in idle-time processing.) Objects that post notifications need not know anything about the observers. On the other hand, observers need to know at least the notification name and the keys to any dictionary encapsulated by the notification object.

8.4 CUSTOM DRAWING WITH COCOA

Drawing is a fundamental part of most Cocoa applications. If the application uses only standard system controls, then Cocoa does all of the drawing for us. If there is the need to use custom views or controls, though, then it is up to the programmer to create their appearance using drawing commands.
In the following sections we will provide a quick tour of the drawing-related features available in Cocoa.

8.4.1 COCOA DRAWING SUPPORT

The Cocoa drawing environment is available to all applications built on top of the Application Kit framework. This framework defines numerous classes and functions for drawing everything from primitive shapes to complex images and text. Cocoa drawing also relies on some primitive data types found in the Foundation framework (Foundation.framework).

The Cocoa drawing environment is compatible with all of the other drawing technologies in Mac OS X, including Quartz, OpenGL, Core Image, Core Video, Quartz Composer, PDF Kit, and QuickTime. In fact, most Cocoa classes use Quartz extensively in their implementations to do the actual drawing. In cases where we find Cocoa does not have the features we need, it is no problem to integrate other technologies where necessary to achieve the desired effects.
Because it is based on Quartz, the Application Kit framework provides most of the same features found in Quartz, but in an object-oriented wrapper. Among the features supported directly by the Application Kit are the following:

- Path-based drawing (also known as vector-based drawing)
- Image creation, loading and display
- Text layout and display
- PDF creation and display
- Transparency
- Shadows
- Color management
- Transforms
- Printing support
- Anti-aliased rendering
- OpenGL support

Like Quartz, the Cocoa drawing environment takes advantage of graphics hardware wherever possible to accelerate drawing operations. This support is automatic. We do not have to enable it explicitly in our code.

8.4.2 THE PAINTER'S MODEL

Like Quartz, Cocoa drawing uses the painter's model for imaging. In the painter's model, each successive drawing operation applies a layer of "paint" to an output "canvas." As new layers of paint are added, previously painted elements may be obscured (either partially or totally) or modified by the new paint. This model allows us to construct extremely sophisticated images from a small number of powerful primitives.

Figure 8.6 shows how the painter's model works and demonstrates how important drawing order can be when rendering content. In the first result, the wireframe shape on the left is drawn first, followed by the solid shape, obscuring all but the perimeter of the wireframe shape. When the shapes are drawn in the opposite order, the results are very different. Because the wireframe shape has more holes in it, parts of the solid shape show through those holes.

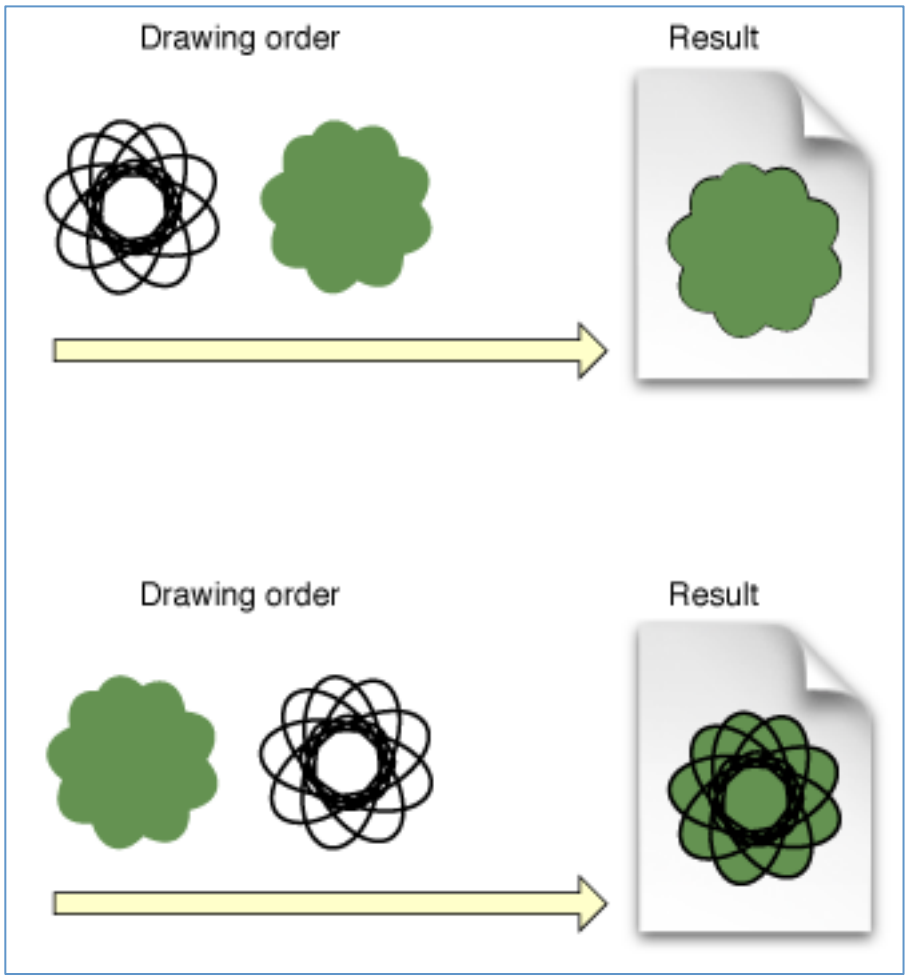

FIGURE 8.6 - THE PAINTERS' MODEL

8.4.3 BASIC DRAWING ELEMENTS

The creation of complex graphics often has a simple beginning. In Cocoa, everything we draw is derived from a set of basic elements that we assemble in our drawing code. These elements are fundamental to all drawing operations and are described in the following sections.

GEOMETRY SUPPORT

Cocoa provides its own data structures for manipulating basic geometric information such as points and rectangles. Cocoa defines the data types listed in the following table. The member fields in each of these data structures are floating-point values.

TYPE	DESCRIPTION
`NSPoint`	A point data type consists of an x and y value. Points specify the coordinates for a rendered element. For example, we use points to define lines, to specify the start of a rectangle, to specify the angle of an arc, and so on.
`NSSize`	A size data type consists of a width and height field. Sizes are used to specify dimensions of a target. For example, a size data type specifies the width and height of a rectangle or ellipse.
`NSRect`	A rectangle data type is a compound structure composed of an origin point and a size. The origin field specifies the location of the rectangle's bottom-left corner in the current coordinate system. The size field specifies the rectangle's height and width relative to the origin point and extending up and to the right. (Note, in flipped coordinate spaces, the origin point is in the upper-left corner and the rectangle's height and width extend down and to the right.)

SHAPE PRIMITIVES

Cocoa provides support for drawing shape primitives with the NSBezierPath class. We can use this class to create the following basic shapes:

- Lines
- Rectangles
- Ovals and circles
- Arcs
- Bezier Arabic Curves

Bezier path objects store vector-based path information, making them compact and resolution independent. We can create paths with any of the simple shapes or combine the basic shapes together to create more complex paths. To render those shapes, we set the drawing attributes for the path and then stroke or fill it to "paint" the path to our view.

IMAGES

Support for images is provided by the NSImage class and its associated image representation classes (NSImageRep and subclasses). The NSImage class contains the basic interface for creating and managing image-related data. The image representation classes provide the infrastructure used by NSImage to

manipulate the underlying image data. Images can be loaded from existing files or created on the fly.
Cocoa supports many different image formats, either directly or indirectly. Some of the formats Cocoa supports directly include the following:

- Bitmap images, including the following image formats:
  - BMP
  - GIF
  - JPEG
  - JPEG 2000
  - PNG
  - TIFF
- Images based on Encapsulated PostScript (EPS) data
- Images based on Portable Document Format (PDF) data
- Images based on PICT data
- Core Image images

Because they support many types of data, we should not think of image objects strictly as bitmaps. Image objects can also store path-based drawing commands found in EPS, PDF, and PICT files. They can render data provided to them by Core Image. They can interpolate image data as needed and render the image at different resolutions as needed.

GRADIENTS

In Mac OS X v10.5 and later, we can use the NSGradient class to create gradient fill patterns.

TEXT

Cocoa provides an advanced text system for drawing everything from simple strings to formatted text flows. Because text layout and rendering using the Cocoa text system is a very complicated process, it is not covered with great detail here[48].

8.4.4 VIEWS AND DRAWING

Nearly all drawing in Cocoa is done inside views. Views are objects that represent a visual portion of a window. Each view object is responsible for displaying some visual content and responding to user events in its visible area. A view may also be responsible for one or more subviews.

The NSView class is the base class for all view-related objects. Cocoa defines several types of views for displaying standard content, including text views, split views, tab views, ruler views, and so on. Cocoa controls are also based on the NSView class and implement interface elements such as buttons, scrollers, tables, and text fields. In addition to the standard views and controls,

[48] For more details, please visit http://developer.apple.com/

we can also create our own custom views. We create custom views in cases where the behavior we are looking for is not provided by any of the standard views. Cocoa notifies our view that it needs to draw itself by sending our view a `drawRect:` message. Our implementation of the `drawRect:` method is where all of our drawing code goes.

By default, window updates occur only in response to user actions. This means that our view's `drawRect:` method is called only when something about our view has changed. For example, Cocoa calls the method when a scrolling action causes a previously hidden part of our view to be exposed. Cocoa also calls it in response to requests from our own code. If the information displayed by the custom view changes, we must tell Cocoa explicitly that we want the appropriate parts of our view updated. We do so by invalidating parts of our view's visible area. Cocoa collects the invalidated regions together and generates appropriate drawRect: messages to redraw the content.

Although there are numerous ways to draw, a basic `drawRect:` method has the following structure:

```
– (void)drawRect(NSRect rect)
{
    // Draw contents here
}
```

LISTING 8.2 - DRAWRECT METHOD STRUCTURE

By the time our `drawRect:` method is called, Cocoa has already locked the drawing focus on our view, saved the graphics state, adjusted the current transform matrix to our view's origin, and adjusted the clipping rectangle to our view's frame. All we have to do is draw our content.

In reality, our `drawRect:` method is often much more complicated. Our own method might use several other objects and methods to handle the actual drawing. We also might need to save and restore the graphics state one or more times. Because this single method is used for all of our view's drawing, it also has to handle several different situations. For example, we might want to do more precise drawing during printing or use heavily optimized code during a live resizing operation. The options are numerous and covered in more detail in the Apple developer website.

8.4.5 COMMON DRAWING TASKS

The following table lists some of the common tasks related to drawing the content of our view and offers advice on how to accomplish those tasks.

TASK	HOW TO ACCOMPLISH
Draw the content of a custom view	Implement a drawRect: method in our custom view. Use our implementation of this method to draw content using paths, images, text, or any other tools available to you in Cocoa, Quartz, or OpenGL.
Update a custom view to reflect changed content	Send a `setNeedsDisplayInRect:` or `setNeedsDisplay:` message to the view. Sending either of these messages marks part or all of the view as invalid and in need of an update. Cocoa responds by sending a drawRect: message to our view during the next update cycle.
Animate some content in a view	Use Core Animation, set up a timer, or use the NSAnimation or NSViewAnimation classes, to generate notifications at a desired frame rate. Upon receiving the timer notification, invalidate part or all of our view to force an update[49].
Draw during a live resize	Use the `inLiveResize` method of NSView to determine if a live resize is happening. If it is, draw as little as possible while ensuring our view has the look you want.

8.5 ARCHIVING

Archives and serializations are two ways in which you can create architecture-independent byte streams of hierarchical data. Byte streams can then be written to a file or transmitted to another process, perhaps over a network. When the byte stream is decoded, the hierarchy is regenerated. Archives provide a detailed record of a collection of interrelated objects and values. Serializations record only the simple hierarchy of property-list values.

8.5.1 OBJECT GRAPHS

Object-oriented applications contain complex webs of interrelated objects. Objects are linked to each other by one object either owning or containing another object or holding a reference to another object to which it sends messages. This web of objects is called an object graph.

Even with very few objects, an application's object graph becomes very entangled with circular references and multiple links to individual objects. Figure 8.7 shows an incomplete object graph for a simple Cocoa application. (Many more connections exist than are shown in this figure.) Consider the window's view hierarchy portion of the object graph. This hierarchy is

[49] For information about Core Animation,, about animating with timers, about using NSAnimation objects, see http://developer.apple.com

described by each view containing a list of all of its immediate subviews. However, views also have links to each other to describe the responder chain and the keyboard focus loop. Views also link to other objects in the application for target-action messages, contextual menus, and much more.

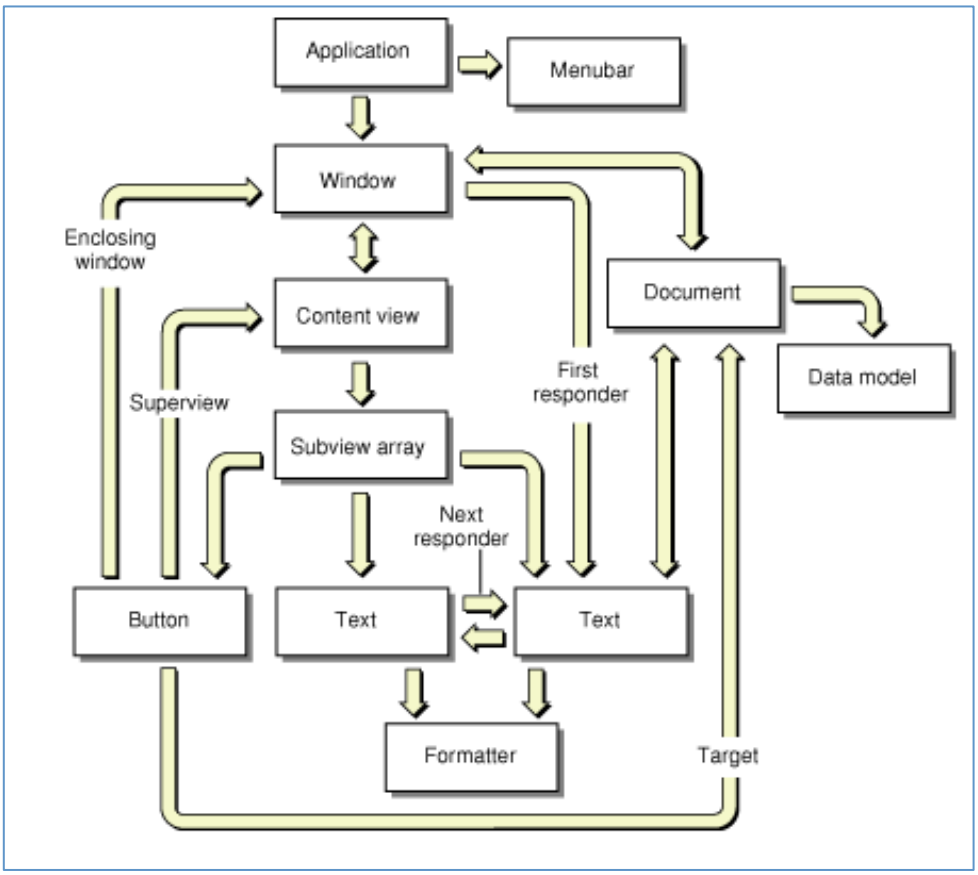

FIGURE 8.7 - PARTIAL OBJECT GRAPH OF AN APPLICATION

There are situations where one may want to convert an object graph, usually just a section of the full object graph in the application, into a form that can be saved to a file or transmitted to another process or machine and then reconstructed. Nib files and property lists are two examples in Mac OS X where object graphs are saved to a file. Nib files are archives that represent the complex relationships within a user interface, such as a window's view hierarchy. Property lists are serializations that store the simple hierarchical relationship of basic value objects.

8.5.2 ARCHIVES

Mac OS X archives store an arbitrarily complex object graph. The archive preserves the identity of every object in the graph and all the relationships it has with all the other objects in the graph. When unarchived, the rebuilt object graph should, with few exceptions, be an exact copy of the original object graph.

Interface Builder uses archives (nib file) to store the objects and relationships that make up a user interface. A Cocoa application loads the nib archive to reconstruct a window, menu, or view that was designed in Interface Builder. Our application can use an archive as the storage medium of our data model. Instead of designing (and maintaining) a special file format for our data, we can leverage Cocoa's archiving infrastructure and store the objects directly into an archive. With minimal effort, we can implement Save and Open in your application. To support archiving, an object must implement the NSCoding protocol, which consists of two methods. One method encodes the object's important instance variables into the archive and the other decodes and restores the instance variables from the archive. All of the Foundation value objects (NSString, NSArray, NSNumber, and so on) and most of the Application Kit user interface objects implement NSCoding and can be put into an archive. Each class's reference document identifies whether they implement NSCoding.

8.5.3 SERIALIZATIONS

Mac OS X serializations store a simple hierarchy of value objects, such as dictionaries, arrays, strings, and binary data. The serialization only preserves the values of the objects and their position in the hierarchy. Multiple references to the same value object might result in multiple objects when deserialized. The mutability of the objects is not maintained.
Property lists are examples of serializations. Application attributes (the Info.plist file) and user preferences are stored as property lists.
Arbitrary objects cannot be serialized. Only instances of NSArray, NSDictionary, NSString, NSDate, NSNumber, and NSData (and some of their subclasses) can be serialized. The contents of array and dictionary objects must also contain only objects of these few classes.

9 BIBLIOGRAPHY

Apple. (n.d.). *The Objective-C Programming Language: Introduction*. (Apple) From Apple Developer: developer.apple.com

Bucanek, J. (2009). *Learning Objective-C for Java Developers.* USA: Apress.

Cocoa Fundamentals Guide. (n.d.). From Mac OS X Developer Library: http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html#//apple_ref/doc/uid/TP40002974-CH3-SW16

Interactive Children's Books. (n.d.). From Wikipedia: http://en.wikipedia.org/wiki/Interactive_children's_book

iOS. (n.d.). From Wikipedia: http://en.wikipedia.org/wiki/IOS

iOS Technology Overview. (n.d.). From iOS Developer Library: http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/IPhoneOSOverview/IPhoneOSOverview.html#//apple_ref/doc/uid/TP40007898-CH4-SW1

Kernel Programming Guide. (n.d.). From Mac OS X Developer Library: http://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KernelProgramming/Architecture/Architecture.html

Mac OS X Technology Overview. (n.d.). From Mac OS X Developer Library: http://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/OSX_Technology_Overview/About/About.html

Model-View-Controller. (n.d.). From Wikipedia: http://en.wikipedia.org/wiki/Model–view–controller

Objective-C. (n.d.). From Wikipedia: http://en.wikipedia.org/wiki/Objective-C

TABLE OF FIGURES

TABLE OF ALGORITHMS

ACKNOWLEDGMENTS

I offer my sincerest gratitude to my supervisor, Dr. Sergio Congiu, for his patience and for his advices.

I am grateful to the members of Altera team, for their guidance and enthusiasm. Without them this thesis would not have been completed or written.

I would like to express the deepest appreciation to Maurizio and Adriana Peruzzo, without whom I would not have been able to get my last degree.

I would like to thank Dr. Alessia Rigon, who helped me with improving my English.

A thank you to my girlfriend Alice Zappa and to anyone who have supported me over the last five years.

Lastly, and most importantly, I wish to thank my family, for their understanding, patience, endless support and never failing faith in me.