ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Interaktive Systeme

Development of an iPhone business application

Diplomarbeit an der Universität Ulm

Vorgelegt von:
Andreas Robecke
andreas.robecke@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert
Prof. Dr. Peter Dadam

Betreuer:
Rüdiger Pryss

2011

Fassung February 3, 2011

Satz: PDF-LaTeX 2$_\varepsilon$

Contents

1 Introduction

Smart-phones conquered the world in the past years and it seems like they have become an indispensable tool for many people to alleviate their everyday life. By today they can compete with the CPU power and memory of the personal computers built not even ten years ago [37]. Effective batteries allow to use the CPU power and perform tasks like playing music and videos or running a mobile internet connection for hours. Techniques like UMTS and not at least the increasing coverage of WLAN access points provide mobile fast internet connections and enable mobile client applications to communicate with servers anywhere. Often a GPS unit in conjunction with a respective framework allows for localisation and navigation functionalities within applications. Cameras and 3D accelerometer can serve as additional input interfaces. The smart-phones have proven to be a applicable platform for many everyday life applications. One of the most advanced smart-phones is being produced by Apple: the iPhone. By now there are hundreds of thousand iPhone applications available. More and more companies decide to take advantage of Apple‘s mobile platform and develop their own iPhone applications in order to support their employees in daily tasks (See [8] for examples). In this diploma thesis we want to analyse and evaluate the paradigm of iPhone development. Therefore we will develop an iPhone application. Along the procedure of development we want to get a feeling for all the aspects involved in development. The iPhone comes with an outstanding multi touch screen as the main user input interface. This is an important difference to usual computer input interfaces which influences development heavily. The operating system was being designed to perform well with limited resources and requires to adopt programming in terms of efficiency. Apple provides a multiplicity of frameworks for development but also restricts developers and applications by certain policies and guidelines. In this diploma thesis we want to learn about all these things. Our findings shall serve as a basic framework for further investigations on developing business applications for Apple‘s mobile devices. By developing our own iPhone application we want to get a feeling for the performance of the device and furthermore generate code which can be reused and investigated for future projects. In order to achieve all this, we have addressed ourselves the task of developing an iPhone application for the calculation of scholarly indexes. This idea seems to be an adequate challenge as it will include parsing, processing and the storage of data. The key feature of our application is to retrieve data from Google Scholar in order to calculate the h- and g-Indexes of scholars. Hereby we want all necessary tasks, from data retrieval to index calculations, to be performed on the device. The two indexes aim to be a measurement of the productivity and

impact of the work of a scholar. They are both based on the amount of times the top most cited publications are being cited of other publications. Before we present the technical parts of the project we want to give a short introduction about the mentioned indexes, their background and relevance.

1.1 The H-Index

The h-index was defined by J. E. Hirsch in his paper “An index to quantify an individual‘s scientific research output” [42] in 2005 as the number of papers with citation numbers $>= h$. For an illustration see figure 1.1 below.

Assuming a scholar has published 22 papers. When we order the papers based on their citation counts, we receive a table like the one to the right.

The h-index is the number of the best cited publications which have been cited h times at least.

For this example the h-index therefore is 9 as there are 9 publications with at least 9 citations. The 10th paper only has 8 citations and thus does not contribute to the h-index in this case.

h	Papers
1	100 Citations
2	21 Citations
3	17 Citations
4	11 Citations
5	10 Citations
6	10 Citations
7	9 Citations
8	9 Citations
9	9 Citations
10	8 Citations
11	7 Citations
12	6 Citations
13	5 Citations
14	4 Citations
15	4 Citations
16	3 Citations
17	2 Citations
18	2 Citations
19	1 Citations
20	1 Citations
21	0 Citations
22	0 Citations

Figure 1.1: H-index calculation example

Hirsch claims the h-index to be a useful index to characterize the scientific output of a researcher by combining quality with quantity and avoiding the disadvantages of the other bibliometric indicators like the following:

1. Total number of papers (N_p):
   - Advantage: Measures productivity.
   - Disadvantage: Does not measure the impact of papers.
2. Total number of citations $(N_{c,\ tot})$:
   - Advantage: Measures total impact.
   - Disadvantage: Hard to find and may be inflated by a small amount of highly cited papers which may not be representative for the individual if he/she is a co-author.
3. Citations per paper $(i.e.\ ratio\ of\ N_{c,tot}\ to\ N_p)$:
   - Advantage: Allows to compare scientists of different ages.
   - Disadvantage: Hard to find, rewards low productivity and disadvantages high productivity.
4. Number of "significant papers", defined as the number of papers with $> y$ citations:
   - Advantage: Eliminates the disadvantages of all criteria above, gives an idea of broad and sustained impact.
   - Disadvantage: Randomly favours or disfavours individuals as y is arbitrary. Y would have to be adjusted for different levels of seniority.
5. Number of citations to each of the q most-cited papers:
   - Advantage: Overcomes many of the disadvantages of the criteria above.
   - Disadvantage: Not a single value and thus insufficient for comparison,q is arbitrary which again would favour or disfavours individuals randomly.

Although the h-index aims to measure the broad impact of an individual's work by respecting quality as well as quantity, here are some aspects which have to be considered among others.

1. Different fields will have h values typically to the field. Thus the comparison of scientists of different fields does not make much sense.
2. The h-index is bounded by the total amount of publications. A scholar who published five articles can only achieve an h-index of five even if these publications are highly cited and would be the most significant articles in the whole field of research.
3. The amount of co-authors of a paper in not being considered.
4. Context is critical:
   a) Citations can be made in a negative context but still would contribute to a better h-index

b) Many citations are used simply to flesh out a paper's introduction and thus have no real significance to the work.

c) Well-established researchers and projects are cited disproportionately more often than those less known.

5. Manipulation possibilities: Systematic self citing can increase someone's h-index. The same thing applies for indirect self citations which is the case when a co-author self-cites a publication. One author could help another to increase his h-index by citing publications which need few more citations to contribute to the h-index.

1.2 The G-Index

The g-index was introduced by Leo Egghe in his paper "Theory and practice of the g-index" [39] in 2006 as an improvement to the h-index. The g-index is defined as the unique number such that the g most cited articles received (together) at least g^2 citations. Thus the g-index inherits all the good properties of the h-index and in addition better takes into account the top most cited articles. The h-index is robust in the sense that it is insensitive to lowly cited papers as well as outstanding highly cited papers. Egghe claims latter to be a drawback as the evolution of the most cited papers is not being taken into account. Once a paper is selected to belong to the top h papers, the paper does not influence the calculation of h-index in subsequent years even if it doubles its number of citations. Therefore Egghe introduced the g-index to better take into account outstanding highly cited publications over time. For an example see figure 1.2.

Assuming a scholar has published 22 papers. When we order the papers based on their citation counts, we receive a table like the one to the right.

The g-index is defined as the number such that the g most cited articles together received at least g^2 citations

For this example the g index therefore is 15 as the 15 best cited papers together have a citation count of 230. But the 16 best cited papers do not have a citation count of 256 or more.

g	Papers	g^2	∑ citations
1	100 Citations	1	100
2	21 Citations	4	121
3	17 Citations	9	138
4	11 Citations	16	149
5	10 Citations	25	159
6	10 Citations	36	169
7	9 Citations	49	178
8	9 Citations	64	187
9	9 Citations	81	196
10	8 Citations	100	204
11	7 Citations	121	211
12	6 Citations	144	217
13	5 Citations	169	222
14	4 Citations	196	226
15	4 Citations	225	230
16	3 Citations	256	233
17	2 Citations	289	235
18	2 Citations	324	237
19	1 Citations	361	238
20	1 Citations	400	239
21	0 Citations	441	239
22	0 Citations	484	239

Figure 1.2: G-index calculation example

1.3 Acceptance of the indexes

The acceptance, especially of the h-index has increased in the years since its introduction in 2005. The Wired Magazine published the article "The Genius Index: One Scientist's Crusade to Rewrite Reputation Rules" [40] in June 2009. The article states: "*In its nearly four years of life, the relatively simple, flexible h-index has become the most talked-about metric in the very hot science of rating scientists and their research,...Schools and labs use such ratings to help them make grants, bestow tenure, award bonuses, and hire postdocs. In fact, similar statistical approaches have become standard practice in Internet search algorithms...*". Public lists of the h-indexes of scholars of specific fields [46] [47] further suggest the broad acceptance of the h-index. Even though the g-index tries to eliminate one of the weaknesses of the h-index [39], which even Hirsch acknowledged according to the mentioned article in the Wired Magazine [40], it does not seem to have the same popularity as the h-index yet.

1.4 Google Scholar as the data source

In order to calculate the h- and g-index, our application needs a data source from which it can retrieve the citation counts of the publications of the scholar of interest. Comprehensive data sources are the “ISI Web of Science” [4], which is provided by Thomson Reuters, Elsevier‘s “Scopus” [3] and Google‘s “Google Scholar” [1]. There are several scientific publications which discuss and compare these sources, serving as data sources for calculating h-indexes, intensively. The calculation results definitely vary by the data source being used and there are different opinions, based on several criteria, on which data source serves best for such calculations. We neither object to discuss the differences of these data sources nor the topic of which data source serves best for the calculation of the indexes.

However, for a better understanding and an imagination of the impact of these differences we will mention a few.

1. The different sources have different coverage in different fields. [35]
2. The coverage of non English publications differs [35]
3. In contrast to WoS and Scopus, which index citations mainly from journal articles and conference papers, citations found through GS come from many different types of documents, including journal articles, conference papers, doctoral dissertations, master’s theses, technical reports, research reports, chapters, and books, among others [45]
4. Google Scholar sometimes includes non-scholarly sources (e.g., course reading lists), phantom or false citations [45]
5. Google‘s coverage of sources is not clear as they never published anything about it
6. Google Scholar‘s processing is fully machine based and occasionally makes mistakes i.e. cannot adopt details like publication year, author,... or even double counts citations [41]
7. Web of Science and Scopus both administer data manually

Obviously those differences between the data sources have to be considered as they directly influence the h-indexes and their reliability. One may imagine that a particular source results in higher h-indexes if it includes more sources than another. Again another might provide lower h-indexes in a particular field. The h- and g-index on the other hand are quite robust to minor data faults by their definitions and may compensate some of those aspects. In her paper "Which h-index? – A comparison of WoS, Scopus and Google Scholar" [34] Judit Bar-Ilan calculated and compared the h-indexes of highly cited Israeli researchers using the three mentioned data sources. In her conclusion she writes "*The findings show that it matters which citation tool is used to compute the h-index of scientists*". Table 1.1 shows

a selection of the h-indexes and the variations based on the data sources.

Researcher	**Subject Area**	**WoS**	**Scopus**	**GS**
Alexander, Gideon	Physics	32	30	20
Alon, Noga	Mathematics, Computer Science	14	17	27
Beeri, Catriel	Computer Science	3	3	8
Ciechanover, Aaron	Biology & Biochemistry	33	34	30
Dolev, Daniel	Computer Science	5	7	18
Mikenberg, Giora	Physics	31	10	4
Netzer, Hagai	Space Sciences	28	28	18
Oren, Moshe	Molecular Biology & Genetics	47	49	38
Peleg, David	Computer Science	8	11	21
Shainberg, Isaac	Ecology/Environment	8	10	9

Table 1.1: H-indexes according to Web of Science, Scopus and Google Scholar

Anne-Wil Harzing does not doubt that all these data sources have their own limitations but also states in her publication "Google Scholar - a new data source for citation analysis" [41] that she believes: "*In most cases, Google Scholar presents a more complete picture of an academic's impact than the Thomson ISI Web of Science*". Due to the fact that Google Scholar is the only comprehensive data source which is available freely, we will use it as the data source in our application. This also has the advantage that the results can be reproduced by anyone.

2 iPhone Development Introduction

During this diploma thesis we will often refer to technologies, methods and classes of different types. To make it easier for you to distinguish between custom code which we have implemented or technologies already provided through Objective C or by the frameworks we have used, we will mark methods, classes, frameworks and specific technologies in different colours. The *- DARK BLUE COLOUR -* shows that we are talking about something specific to our application and usually means that we have implemented or created the class or method or whatever we are talking about ourselves. The *- GENERAL TECHNOLOGY COLOUR -* stands for technologies, classes and methods provided either by one of the frameworks we used or specific to Objective C.

When we started developing the current iPhone OS and SDK versions were 3.X. During the development the new version of the previously called iPhone OS was introduced as iOS (4.0). When the iOS came out, the new SDK version also was renamed in iOS SDK. Even though we started developing in version 3.2, we switched to the new SDK when it came out. However, in the following chapters we will refer to all versions of the iPhone - OS and - SDK as iOS and iOS SDK. In cases where it is relevant, we will provide the specific versions. Sometimes we talk about iPhone applications or apps. When we do so, we mean iOS applications which run on the iPhone but also on the iPodTouch and possibly on the iPad.

2.1 XCode, Interface Builder and the iOS SDK

Before you can start developing an iPhone application, it is necessary to install XCode and the iOS SDK on your Mac. Your Mac has to be Intel-based and needs to run Snow Leopard. The iOS SDK is not designed to run on other systems which makes the Mac indispensable for serious development. XCode and the iOS SDK can be downloaded for free after registering as an Apple developer, which is also for free.

2.1.1 XCode

XCode is Apple's IDE. It ships with built-in project templates, a graphical debugger, the iPhone Simulator, Interface Builder and Instruments.

iPhone Simulator

The iPhone Simulator enables you to test your applications directly on your desktop without connecting an actual device.

Interface Builder

Interface Builder is a tool for developing user interfaces graphically. It allows you to build a complete user interface via drag and drop and then connect the individual components to objects and methods in your code with so called *IBOutlets* and *IBActions*.

Instruments

Instruments provides a set of tools for inspecting memory usage, disk activity, network activity, and graphic performance. Instruments is very helpful for tracking down memory leaks and locating critical areas of memory usage.

2.1.2 iOS SDK

On top of the iOS Kernel there are four layers of services on which iOS applications can be built. Figure 2.1 shows the layer hierarchy.

These layers provide different interfaces and technologies on different levels of abstraction. The higher-level frameworks provide infrastructures for implementing standard system behaviour. Lower-level frameworks usually are suitable for implementing custom behaviour which is not provided by higher levels. The starting point for building a new application usually is the Cocoa Touch layer and the *UIKit* in particular. The frameworks at these layer provide the fundamental infrastructure for an application whereat most technologies are based on Objective-C. The UIKit framework provides the visual infrastructure for an application, including classes for windows, views, controls and controllers for handling those objects.

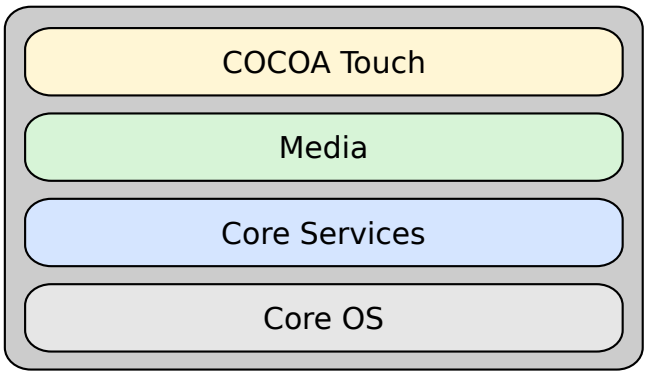

Figure 2.1: iOS layers

Moving down from the Cocoa Touch layer to the next layer, the Media layer provides technologies to support drawing, audio and video. The interfaces on this layer are either based on Objective-C or C. A C-based technology on this layer for example is OpenGL. An Objective-C based technology on the other hand is Core Animation, which is a framework for animating user interfaces.

The two lower-level layers, Core Services and Core OS contain the fundamental interfaces for iOS. Most interfaces on this layer are C-based. A Core OS layer-specific technology for example is the database technology SQLite.

2.2 iPhone Limitations

Developing for a mobile platform such as the iPhone, one has to consider several restrictions. Due to platform limitations such as limited amount of memory and CPU power, there is no garbage collection in iOS. This limitation directly affects development and is the reason why the developer is responsible for retaining and releasing objects in memory. If an application takes up too much memory, iOS simply forces it to quit. Applications running on iOS, live in so called sandboxes, which is a security concept and restricts applications in terms of data access. Applications only can access the data in their own sandbox. It is not possible to access data from other applications or certain folders like the iTunes library. The lack of physical input devices and the tiny screen might restrict user interaction to some extend. On the other hand the iPhone offers technologies such as multi touch, an accelerometer which detects device orientation and an on-screen keyboard. These technologies in com-

bination with the provided frameworks such as the UIKit enable the developer to build rich user interaction despite restrictions.

2.3 Objective-C and MVC

The iOS SDK is designed around supporting two programming paradigms. Object oriented programming and the Model View Controller (MVC) design pattern. Objective-C is the object oriented programming language which is used for the implementation of iOS applications. It is a superset of ANSI C that has been extended with certain syntactical and semantic features derived from Smalltalk to support object oriented programming. Due to the fact that Objective-C is based on a foundation of ANSI C, it is possible to mix straight C with Objective-C. Inheritance becomes an important feature for hierarchically structuring the behaviour of specific classes in the frameworks such as the UIKit framework. Many classes provided by the frameworks, are shipped with the SDK, inherit from multiple parent classes (Not multiple inheritance!). The *UIButton* for example inherits from the classes *UIControl* and *UIView*, thus acting as a *UIView* as well as a *UIControl*. The complete inheritance hierarchy of the *UIButton* class actually is *NSObject* : *NSResponder* : *UIView* : *UIControl*. If you want to built an application for iOS, you have to be familiar with Objective-C. Apple's document "Introduction to The Objective-C Programming Language" [25] is a good starting point to learn Objective-C.

2.3.1 Memory Management

When you are programming an application for iOS, you will have to manually take care of the memory management as there is no garbage collection available for iOS. This means you have to manually allocate and free the memory which is allocated by your objects, as soon as you do not need them any longer. If you do not take care of your memory properly, your application will leak memory and you risk it to get terminated by the system at some point. Therefore memory management is an important issue you have to deal with. In Cocoa you will use a reference counting system to take care of an object's life cycle 2.1. When you allocate an object it will have a reference count of one. You can manipulate the reference count by using the instance methods *retain* and *release*. If you retain an object the reference will be incremented by one, if you release it, the reference count will be decremented by one 2.2. As soon as the reference count is zero the object will be destroyed automatically and the allocated memory freed. Another fundamental aspect is the object ownership mechanism by which you can specify when to release an object. If you create or retain an object, using one of the methods *alloc, new, copy* or *retain*, you own it. An object thus can have multiple owners. If you own an object, you are responsible for releasing it

later. If you do not own an object on the other hand, you are not allowed to release it. So if you release an object at some point in your code, you have to make sure that you are the owner at this point. Of course the object can still have other owners. But you do not need to care about existing other ownerships when you release an object. You simply have to release an object for each time you retained it and you release it as soon as you do not need it any more. Apart from the release method you can send an *autorelease* message to an object, to declare that you do not want to own the object beyond the scope in which you sent the message. This means when you send an *autorelease* message to an object, its retain count will automatically be decremented by one at some point in the future. Let's assume you create an object in a method to return it to the caller. In this case the method does not want to take responsibility for the created object even though it has created it. Therefore the method will call an *autorelease* message to the object before returning it. The caller then has to make sure he acquires ownership for this object by sending a retain message to it. Listing 2.6 is an example of latter. If the caller does not do so, it will be destroyed automatically at some point in the future. Maybe before the caller expects this. Some classes provide so called *convenience constructors*, which return autoreleased objects. Method names usually indicate if they retain objects by containing one of the words *alloc, new or copy* in the method name. If they do not, the created object probably will get autoreleased. Another fundamental concept you should be familiar with, are pointers. If you create an object in Objective-C you assign it to a pointer. A reassignment of the pointer might cause a memory leak as you loose track of the object originally assigned to your pointer. Listing 2.5 is an example of a pointer reassignment causing a memory leak. Moreover you need to know that there are methods which retain objects (Compare listing 2.3 and 2.4). Other methods in turn create autoreleased objects. Compare listing 2.7. Thus, you always have to make sure whether the object will be autoreleased or if you are the owner and therefore have to release it at some point. XCode integrates the tool *Instruments* which allows you to scan for memory leaks. You can check for memory leaks by selecting Run –> Run with Performance Tools –> Leaks in XCode. Another tip is to "build and analyse" your application (Build –> Build and Analyze). XCode will then mark the lines of code which potentially might cause memory leaks.

Listing 2.1: Object Lifecycle

```
1 NSString *hello = [[NSString alloc] initWithString:@"Hello!"];
2 //reference count is 1
3 [hello release];
4 /*
5 calling release on the string
6 reference count is 0
7 object will be destroyed
```

```
and memory can be freed
*/
```

Listing 2.2: Retain Message

```
NSString *hello = [[NSString alloc] initWithString@"Hello!"];
//reference count is 1
[hello retain];
//reference count is 2
[hello release];
//reference count is 1
[hello release];
//reference count is 0
```

Listing 2.3: Some methods call retain on objects

```
NSNumber *number = [[NSNumber alloc] initWithInt:84];
//number reference count is 1
NSMutableArray *arr = [[NSMutableArray alloc]init];
//arr reference count is 1
[arr addObject:number];
/*
addObject calls retain on number!
number reference count is 2
*/
[number release];
//number reference count is 1
[arr release];
/*
number reference count is 0
arr reference count is 0
*/
```

Listing 2.4: Another example of a method calling retain on an object

```
UIView *viewX = [[UIView alloc]initWithFrame:CGRectMake(100, 100,
    100, 100)];
//viewX reference count is 1
viewX.backgroundColor = [UIColor redColor];
[self.view addSubview:viewX];
/*
addSubview calls retain on the viewX object
```

```
viewX reference count is 2
*/
[viewX release];
/*
viewX reference count is 1
will be 0 when the parent view will be released
*/
```

Listing 2.5: Reassignment of a pointer

```
NSNumber *number = [[NSNumber alloc] initWithInt:84];
//original number (84) object reference count is 1
number = [[NSNumber alloc] initWithInt:85];
/*
new number (85) object reference count is 1
reassignment of the number pointer:
you lost the pointer to the original number object
and therefore caused a memory leak!
*/
```

Listing 2.6: Autoreleasing

```
/*
When you create an object in a method and you
intend to return it, you have to autorelease it.
The caller on the other hand should retain it if necessary
*/
- (NSString*)sayHello{

        NSString *hello = [[NSString alloc] initWithString:@"Hello
            !"];
        return [hello autorelease];
}
...
NSString *helloString = [self sayHello];
[helloString retain];
//To make sure the object will be kept alive
...
[helloString release];
//Call release when you do not need it any more
```

Listing 2.7: Autoreleasing

```
/*
Some methods construct autoreleased objects
If the method name does not contain alloc, new or copy
but creates an object, it probably will be autoreleased
*/
NSNumber *number = [NSNumber numberWithInt:84];
/*
If you call release on an object which was already
autoreleased your application will crash
*/
```

2.3.2 MVC on the iPhone

Even though the iOS SDK is build around the MVC design pattern, the MVC pattern in iOS does not always match the theory of the fundamental MVC pattern.

Views in an iPhone application are based on the class *UIView*. Nearly all user interface classes are being derived from this class. Each iOS application usually contains one *UIWindow* object, which is a special *UIView* object providing the root for all other *UIView* objects. A view can be displayed by adding it to the applications window object or by adding it to another view by using the *addSubview:* method. A complete user interface can be seen as a tree of sub views. In addition to the *UIView* objects, *UIViewController* objects play a key role in managing views. Even though you can develop an application without using any *UIViewController* class, in most cases you should not do that, as they have many built-in functionality for managing user interfaces. They can provide so much behaviour and functionality expected of an iPhone application that it would be a huge waste of time to implement respective controller classes along the MVC architecture yourself. Especially if you are new to iOS development, building an application without using view controllers, simply is not an option you should consider at all. View controllers are responsible for laying out the items on your screen and thus do not exclusively act as a controllers in the sense of the MVC pattern. Usually one view controller is associated with one view object. This view object usually contains a hierarchy of sub views whereat these sub views can be manipulated by the view controller as well. View controllers take responsibility for rotating the view depending on the device orientation and resizing views to fit in the boundaries defined by special UI elements such as tool bars and navigation bars. The *UIViewControllers* classes exist to make it easy to manage UI elements and build applications that conform to Apples design guidelines. In addition to the base class *UIViewController*, there are special *UIView-*

Controller classes for special purposes. The *TableViewController* for example is a controller specially designed for displaying data in the form of a table. The *UINavigationController* and the *UITabBarController* both are controllers which are designed to navigate between other view controllers. The navigation controller can be used to navigate between different levels in a stack of view controllers. Similarly a tab bar controller can manage multiple distinct view controllers allowing the users to switch between those by tapping the corresponding tab. By combining different *UIViewController* classes, it is possible to build complex layouts. Each view controller then manages a specific part of the layout or just takes responsibility for switching between controllers. In our application for example, we combined a tab bar controller with three navigation controllers whereat all three manage a stack of miscellaneous view controllers. By now you might be able to imagine of how much functionality you would relinquish if you would not use the provided *UIViewControllers* classes. It will definitely be clear after you read the whole diploma thesis. The iOS SDK provides many classes for implementing controllers and views, but it does not provide any model specific templates. So it is up to you to implement callback methods and delegate protocols (See 2.3.3) to support the required functionality.

In the following, we will be talking a lot about view controllers, views and other classes and objects. Thereby we sometimes use the class name even though we refer to an object. Furthermore we will refer to controller objects of the class *UIViewController* as view controllers. *UITableViewController* objects are being named table view controllers and so on. Similarly we would refer to a custom controller object of the class *SuperSpecialCustomViewController* as super special custom view controller. It might seem a bit confusing now, but it should all be clear in the context.

2.3.3 Communication between objects

Controllers, models and views all are represented by objects designed to fulfil a specific role in an application. Each of those objects contributes a limited set of behaviours to the application. In order to get a task done, these objects might need to communicate with each other at runtime. Therefore Cocoa provides several patterns by which objects can talk to each other. The most important mechanisms in iOS are delegation, target-action mechanism and notifications.

Delegation

Delegation is a way of handing over the responsibility for responding to some kind of event. It is used by UIKit classes in order to hand over responsibility for responding to user interaction. A delegate object has to act in behalf of the object originally encountered the event.

Therefore the delegating class holds a property usually called delegate. Furthermore it declares a so called protocol. The protocol defines one or more methods the delegate object has to implement. The delegating object itself does not implement these methods. A good example is the *UITableView* class. It does not have a built-in way of responding to a tap on a row. Instead it requires a delegate object which has to implement the delegate methods such as *tableView: didSelectRowAtIndexPath:* which then implement the response to the user interaction. Listing 2.8 shows an example of our application, where we use delegation for the parser object to communicate with a view controller. The parser class defines the *ParserDelegate* protocol as illustrated by listing 2.8. Figure 2.2 illustrates the delegation pattern in general.

Listing 2.8: Parser.h

```
1  //  Parser.h
2
3  @protocol ParserDelegate
4  @required
5  - (void)requestHTMLFrom:(NSString *)url;
6  - (void)presentResults;
7  - (void)insertQuery;
8  - (void)setExpLastPage:(int)lastPage;
9  @end
10
11 @interface Parser : NSObject {
12         id <ParserDelegate> delegate;
13         ...
14 }
15
16 @property(nonatomic, assign) id <ParserDelegate> delegate;
17 ...
```

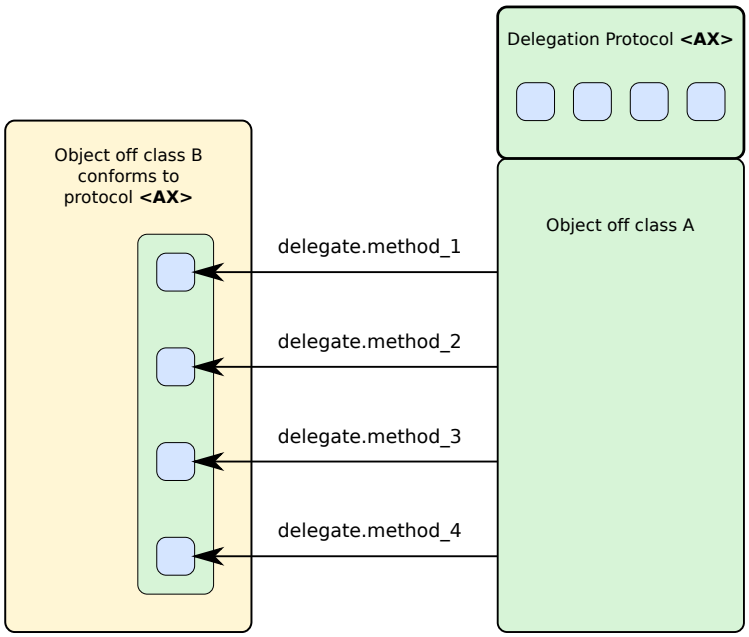

Figure 2.2: Delegation pattern in general

The parser object holds a property to the delegate, in our case an instance of the *SearchViewController* class. The search view controller conforms to the *ParserDelegate* protocol and implements the required methods (*requestHTMLFrom:, presentResults, ...*). In section 5.1.5 we will introduce the whole search flow. Hereby, the communication of the parser object and the search view controller is an important aspect. For an illustration see figure 5.6 in the "Implementation" chapter.

Each iOS application has to implement at least one delegate, the application delegate which adopts the *UIApplicationDelegate* protocol. This is crucial to any application as it must respond to application-launch, application-quit, low-memory, and other messages from the application object.

Almost identical to delegates there are data sources. Unlike being a delegate control of the user interface, it is a delegate control of data. A data source is a reference held by *UIView* objects such as *UITableView* that require a data source from which they retrieve the data they present on the screen. Most of the time, but not necessarily, that is the same object as the delegate. Also similar to the delegate the data source must implement one or more methods of a protocol to supply the view with the data to be displayed. In iOS applications, the view controllers typically act as delegate as well as data source of their corresponding views. This is another iOS variance of the MVC pattern as models are usually responsible

for data encapsulation. Of course data source can be used in a model class either.

The Target-Action Mechanism

The target-actions mechanism is a lower-level way of redirecting user interactions. Especially controls like *UIButtons* use this mechanism to communicate specific user events to other objects. In the UIKit, the children of the *UIControl* class almost exclusively define most of the target-action mechanism for iOS. To set up a control to send an action message to the target, you have to associate both, the target and action with a control event. Listing 2.9 shows an example of the target-action mechanism with a *UIButton*. The button (*subjectAreasBtn*) sets the target to self, the action to *@selector(displaySubjectAreas*) and the control event to *UIControlEventTouchUpInside*. The target has to implement an appropriate respond in its selector. If it does not do that and the action-message is sent to the target object anyway, the application will crash at runtime due to an undefined method call.

Listing 2.9: Target-Action Mechanism with UIButton

```
1 [subjectAreasBtn addTarget:self action:@selector(
     displaySubjectAreas:) forControlEvents:
     UIControlEventTouchUpInside];
```

An action message must have a unique signature. The method it invokes is of the type void and has a single argument of the type id, named sender by convention. The sender identifies the control which sent the action message but is not really required to be specified. Listing 2.10 shows the corresponding method which will be invoked.

Listing 2.10: Target-Action Mechanism - Method

```
1 - (void)displaySubjectAreas:(id)sender{
2 /*
3 If you have defined the (id)sender in your method signature
4 you can then access it within your method
5 i.e. to make decisions based on the senders type
6 */
7 NSLog(@"%@", sender);
8 ...
9 }
```

Notifications

In addition to delegates and target-actions Cocoa offers another way of object communication. Notifications do not implement a tight coupling between the communicating objects and furthermore allow to broadcast notifications in contrast to delegation and action-messages. With notification one object can inform multiple other objects about status changes. Therefore the interested objects have to register with a notification center and then start observing for notification messages.

2.4 User Interface Design

In section 2.1.1 we introduced Interface Builder and said that you can use it to build your user interfaces graphically. In fact most project templates provided by XCode come with a set of Interface Builder files and Apple suggests to build your user interfaces with it. On the other hand it is not indispensable to use Interface Builder as you can create your user interfaces programmatically in a view controllers *loadView* method.

2.4.1 More on Interface Builder

Interface Builder provides a library of user interface elements (See figure 2.3) and some special objects, such as *UIViewController* objects, which you can add to your user interface via drag and drop. See figure 2.4.

Figure 2.3: Interface Builder Library - Elements you can add to your interfaces

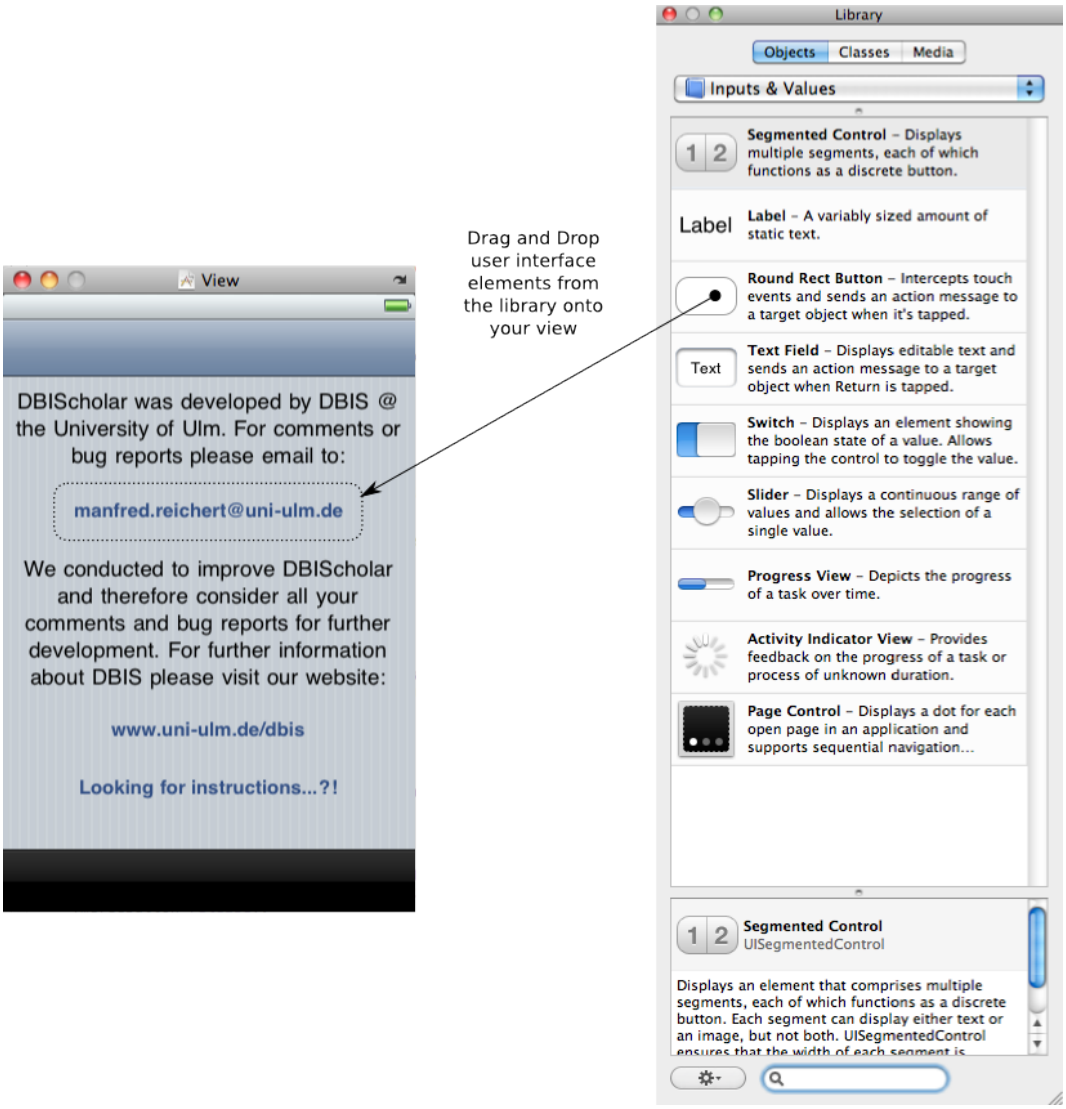

Figure 2.4: Interface Builder - Building UI via Drag and Drop

When you save a user interface created in Interface Builder, it creates a *XIB* file which XCode automatically converts to a *NIB* file at build time so that it can be deployed with your application. This means during development you will create and edit XIB files whereat your application later will use NIB files. XIB files are XML based and provide some advantages for development. NIB files on the other hand are archives. For further information check out [6]. When a NIB file is loaded into memory, the contained objects will be unarchived and instantiated. Each NIB file contains one *File's Owner* object which provides the link

between your hand written code and the objects created from the NIB file. It is responsible for loading and managing the NIB file and the contained objects. Figure 2.5 illustrates an example of our application. It shows the *AboutViewController.xib* main window with all the objects contained in this XIB file. You can see the *File's Owner* object is of the class *AboutViewController*. This means, when we create an instance of the *AboutViewController* class in our application, the about view controller's NIB file and thus the contained user interface will be loaded automatically. Because view controllers often have a corresponding XIB file, XCode offers to create a corresponding XIB file when you subclass *UIViewController* from the provided template. When you choose to do so, it automatically assigns your controller class to the *File's Owner* object in the XIB file.

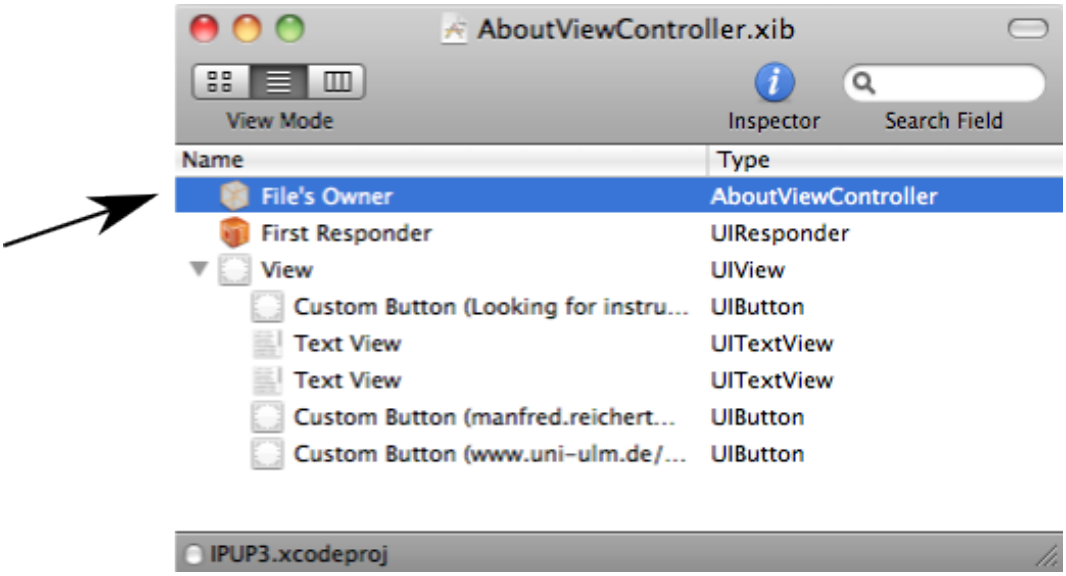

Figure 2.5: AboutViewController as the *File's Owner* object of the AboutViewController.xib

Often you need to access the user interface elements you have added to your XIB file in your code. Or you want one of the buttons you have added to you user interface with Interface Builder to trigger a method in your code. To connect the objects that you have added to your XIB file and your code, there are *IBOutlets* and *IBActions*.

IBOutlets

With *IBOutlets* you can mark properties in your code simply by placing the *IBOutlets* keyword in the property declaration. In the *SearchViewController* class for example, we have defined the instance variable **go* of the type *UIButton* to connect with a button in the XIB

file. Therefore we added a property declaration and the *IBOutlets*. See listing 2.11. When you then open Interface Builder, it looks for *IBOutlets* and allows you to connect the instance variables in your code with objects in the XIB file of the same type. Your application then can access and manipulate these objects during runtime.

Listing 2.11: Declaration of an IBOutlet

```
1  #import <UIKit/UIKit.h>
2  ...
3
4  @interface SearchViewController : UIViewController <
       UITextFieldDelegate, ParserDelegate> {
5  ...
6  UIButton *go;
7  ...
8  }
9  ...
10 @property(nonatomic, retain) IBOutlet UIButton *go;
11 ...
```

There are several ways how you can connect your *IBOutlets* with your code in Interface Builder. It usually involves dragging something somewhere. In the Inspector window for example, you can see the *IBOutlet* of an object. To connect your *IBOutlets* you can click and drag them on the corresponding user interface elements of your XIB's main window as illustrated in figure 2.6.

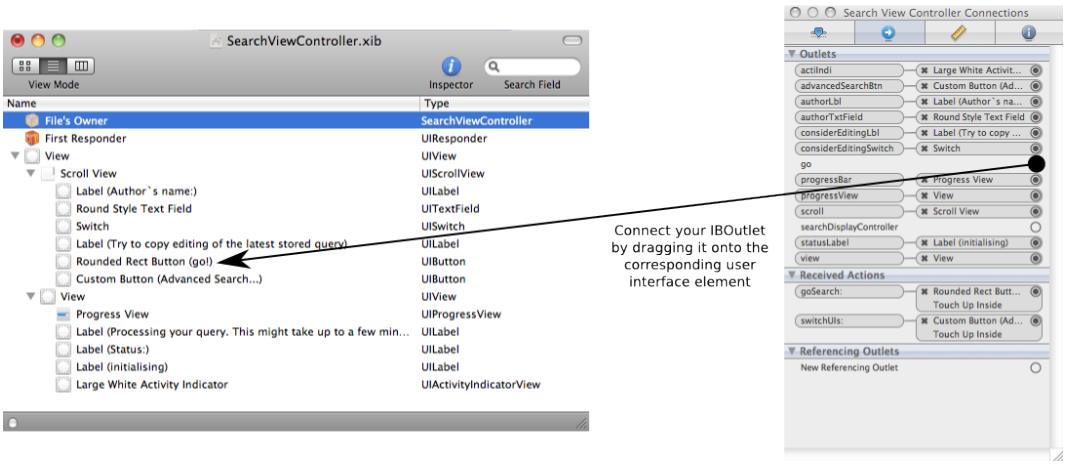

Figure 2.6: Connecting an IBOutlet with the corresponding user interface element

IBActions

IBActions are methods in your code which can be triggered by objects, typically controls loaded from the NIB file. Therefore you mark your methods with the return type *IBAction* to tell Interface Builder to treat it like a target action. See target-action mechanism in section 2.3.3. Usually the method takes one argument *sender* of the type *id* which is a reference to the object sending the action message. The return type *IBAction* is the same as void. If you have an *UIButton* in your user interface which you have created in Interface Builder, you can connect its *UIControlEventTouchUpInside* event with an *IBAction* defined in your view controller's code. In our *AboutViewController* class we have defined three methods. To connect these with controls of the corresponding XIB file, we marked these with the return type *IBAction* as illustrated in listing 2.12. The connection of *IBActions* with corresponding methods works similar to the connection of *IBOutlets* with user interface elements. When you select your target object and open the Inspector window in Interface Builder you can see the *IBActions* you have defined in your code. Again you can click and drag them onto a control in your XIB file. When you do so, Interface Builder will list all events of the control. You can then choose to connect an event with your *IBAction*. See figure 2.7.

Listing 2.12: Declaration of IBActions in code

```
1  #import <UIKit/UIKit.h>
2  ...
3
4  @interface AboutViewController : UIViewController <
       MFMailComposeViewControllerDelegate> {
5
6  }
7
8  - (IBAction)presentInstrcutionsVC;
9  - (IBAction)emailUs;
10 - (IBAction)visitDBISWebsite;
11 ...
```

Buttons in general usually are being connected to the methods they trigger either programmatically (See target-action mechanism in section 2.3.3) or through a connection to an *IBAction* in their XIB file. Note that neither Interface Builder nor XCode will warn you about non-connected *IBOutlets* and *IBActions*. So if you forget to make a connection your application will probably crash at some point or just not behave like you expect it to. Also be careful, when you rename *IBActions* or *IBOutlets*, you have to reconnect them in Interface

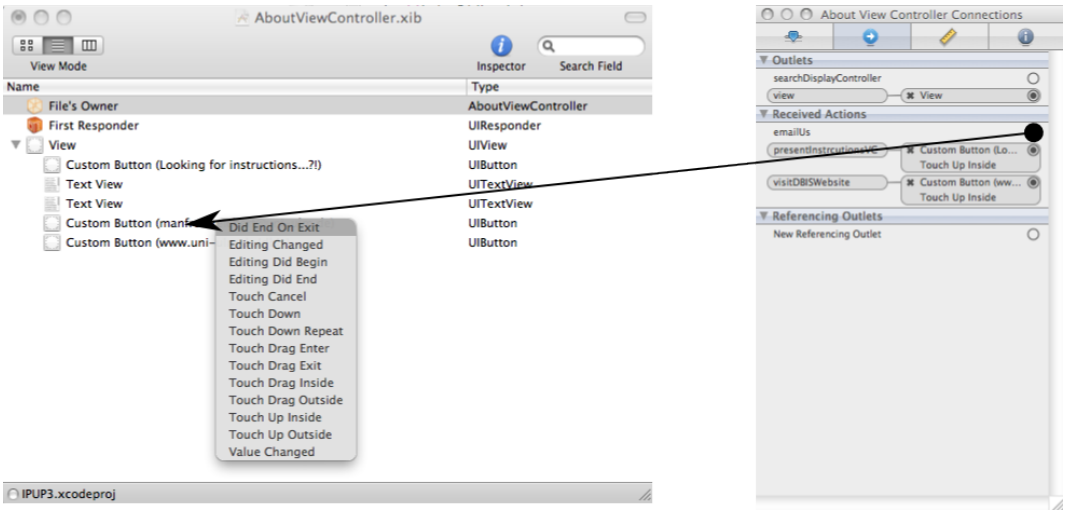

Figure 2.7: Connecting an IBAction with a control of your XIB

Builder. Otherwise you will cause an error too.

Interface Builder primarily simplifies building user interfaces and saves writing code. It is important to know that the entire file and referenced frameworks or code must be loaded into memory so that individual objects can be instantiated. Therefore it is important to keep NIB files small in order to keep memory usage low.

2.4.2 Coding user interfaces vs. Interface Builder

You do not have to use Interface Builder to build your user interfaces. If you decide not to use Interface Builder you will have to write the corresponding code yourself. Depending on the complexity of your user interface this can be quite a lot. Laying out user interfaces and setting all the necessary attributes requires to write several lines of code. The *loadView* method of your controller is where you would put this code. Listing 2.13 shows an example where we simply added an *UILabel* and an *UIButton* to a *UIView* in an view controller. Figure 2.8 shows the resulting user interface. In comparison if you would take advantage of Interface Builder instead, you would not have to write a single line of code and still could adjust your user interface objects programmatically in the *viewDidLoad* method of the view controller. However, there are lively discussions on the internet about NIB files slowing down performance as they are expensive to load. We did not make any experience with NIB files slowing down performance significantly in our application. But it also is not subject of this diploma thesis to compare the performance of the different methodologies of loading

user interfaces. On the other hand one might imagine that thoughtlessly assembling all user interfaces in a single XIB file might slow down performance as all elements are being loaded in memory at once.

Listing 2.13: Coding user interfaces

```
#import "HandMadeViewController.h"

@implementation HandMadeViewController

- (void)loadView {

        // Implement loadView to create a view hierarchy
           programmatically, without using a NIB.

        CGRect rectFrame = [UIScreen mainScreen].applicationFrame;
        UIView *view = [[UIView alloc] initWithFrame:rectFrame];
        view.backgroundColor = [UIColor grayColor];
        self.view = view;

        UIButton *button = [UIButton buttonWithType:
           UIButtonTypeRoundedRect];
        [button setTitle:@"A Button" forState:UIControlStateNormal
           ];
        button.frame = CGRectMake(80, 210, 160, 40);
        [self.view addSubview:button];
        [button release];


        UILabel *label = [[UILabel alloc]initWithFrame:CGRectMake
           (60, 300, 200, 100)];
        label.text = @"This is a label...";
        label.backgroundColor = [UIColor grayColor];
        label.textAlignment = UITextAlignmentCenter;
        label.textColor = [UIColor blackColor];
        label.font = [UIFont systemFontOfSize:20];
        [self.view addSubview:label];
        [label release];

}
```

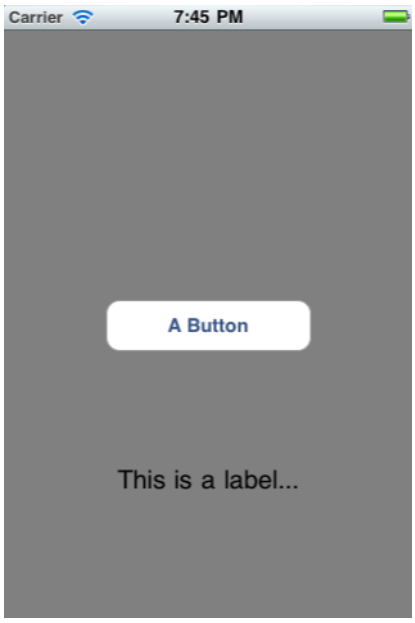

Figure 2.8: The user interface generated by the code of listing 2.13

On the following pages we will refer to navigation bars and tool bars. Just that you know what we are talking about, in the interfaces of view controllers the navigation bar (*UINavigationBar*) always is on top whereas the tool bar (*UIToolbar*) is located at the bottom of the interface. For an illustration see figure 2.9 below.

2.4.3 Human Interface Guidelines

Before you start developing an application you should have a look at Apple's *Human Interface Guidelines* [33]. The documents basically describe how you should design your user interfaces to achieve a rich user experience in your application. iOS and the frameworks with which you will implement your application provide several standard techniques how to implement user interaction. When you are building an application for iOS you should not forget that you are designing for a multi touch screen and respective user interfaces and interactions will have to be different to what you are used to from desktop applications. Moreover iOS users expect a consistent behaviour not only application wide, but also across iOS applications. This has the advantage that users are able to operate any iOS application quickly. Apple provides several techniques and principles to support you in designing user interfaces. Animating your user interactions and interfaces can be crucial in order to achieve rich usability. You also should be aware of the fact that violating Apple's Human Interface Guidelines might be a reason for your application not to be accepted for

Figure 2.9: Navigation Bar and Tool Bar

the distribution in the App Store. But we will talk about the *Review Guidelines* in the next section.

2.5 Review Guidelines

If you are developing an iPhone application, you may want to offer it in the App Store where you can find hundreds of thousands apps by today. You need to know that not every app actually makes it into the App Store. Apple reviews each app which is being submitted for distribution. There are quite a few restrictions, not only in terms of how you have to build and design your app, but also in terms of the content and the functionality it provides. In the *Review Guidelines* and the *iOS SDK Agreement*, which you have to agree to in order to use the iOS SDK, you can find what is allowed and what is not. These terms also apply to third party libraries which you have included in your project. So be careful with third party libraries as they can cause your app to get rejected. In order to read the Apple's Review Guidelines, you need to register as a developer. The guideline's introduction states: "... We don't need any more Fart apps. If your app doesn't do something useful or provide some form of lasting entertainment, it may not be accepted. ... We will reject Apps for any content or behavior that we believe is over the line. What line, you ask? Well, as a Supreme Court Justice once said, "I'll know it when I see it". And we think that you will also know it when

you cross it. To give you some sense of what might cause an app to be rejected we have listed a few restrictions of the current Review Guidelines below.

- Apps that crash will be rejected
- Apps that use non-public APIs will be rejected
- Apps that download code in any way or form will be rejected
- Apps that are "beta", "demo", "trial", or "test" versions will be rejected
- Apps that duplicate apps already in the App Store may be rejected, particularly if there are many of them
- Apps that are not very useful or do not provide any lasting entertainment value may be rejected
- Apps that do not notify and obtain user consent before collecting, transmitting, or using location data will be rejected
- Apps that are primarily marketing materials or advertisements will be rejected
- Apps must comply with all terms and conditions explained in the Apple iPhone Human Interface Guidelines and the Apple iPad Human Interface Guidelines
- Apps that do not use system provided items, such as buttons and icons, correctly and as described in the Apple iPhone Human Interface Guidelines and the Apple iPad Human Interface Guidelines may be rejected
- Apps containing pornographic material, defined by Webster's Dictionary as "explicit descriptions or displays of sexual organs or activities intended to stimulate erotic rather than aesthetic or emotional feelings", will be rejected
- Apps containing references or commentary about a religious, cultural or ethnic group that are defamatory, offensive, mean-spirited or likely to expose the targeted group to harm or violence will be rejected
- Apps that include the ability to make donations to recognized charitable organizations must be free

As we said, these are just some of the restrictions, there are many more. We even made an experience with the restrictions ourselves as we knowingly used a non-public API in our application and tried to submit it for distribution (See 7.2).

2.6 Test your app on the device

At the beginning of this chapter we told you about the iPhone Simulator 2.1.1. The simulator is great because it allows you to investigate iPhone development without actually having

a device. It also is practical during development for testing and debugging features quickly. But if you are serious about developing an app for the distribution in the App Store, you have to get a device. The cheapest option is the iPod Touch as the primarily difference to the iPhone is that it does not support the mobile communication technologies. For many apps, an iPod Touch will be sufficient for testing, for others it won't. That depends on the functionality you are planning to develop. However a real device is indispensable when it comes to testing your app. You can get the real "look and feel" of your application only on the device. Even more important you have to make sure the app in general performs well on the device. This is not assured as the simulator in some cases behaves differently to the device. Unfortunately the device is not the only thing necessary for testing. You also have to be enrolled in one of the iOS developer programs. If you are studying or teaching at a university you might have the possibility to enrol for the *iOS Developer University Program*. It is free and allows you to test your apps on iPads, iPhones, and iPods. It does not allow to distribute your apps in the App Store though. If you are planning to do so, you will have to enrol in one of two programs, either into the *Standard Individual Program* or the *Standard Company Program*. Either one will cost you a fee of $99 USD per year. For more information about the developer programs check out Apple's Support Center [11]. When you have a device and you are enrolled in one of the developer programs, there are still some steps required before you finally can execute your code on your device. When you log into the iOS Development Center with your developer account, you will find many resources about the required steps for testing and app on a device. Many developers on the internet complain about having problems with setting up the device for testing. In order to run your application on a device you need to sign it with a valid signing identity. To sign your application for development you need a private key, iPhone Developer Certificate, and a Development Provisioning Profile. The private key is necessary so that XCode can sign your app binaries. The Developer Certificate is a electronic document that associates your digital identity with information including your name, email address, or business. Provisioning profiles let XCode know which certificate/private key pair to use to sign your application and let the device know how to verify the applications installed on them. When you are signed in at the iOS Developer Center and you go to the *Provisioning Portal*, click on *Certificates* and then select the *How To* tab [31], you will find the required steps for setting up your device for testing. As you will see there it is also possible to transfer your private key to other Macs, so that you can work from multiple systems. We actually did this once and can confirm that it worked for us. But before you can install your application, you have to register your device for development. Therefore you can go back to the "*Home*" area of the Provisioning Portal, launch the provisioning assistant and follow along the steps. Another option is to open your *Organizer* in XCode (*Window –> Organizer*). If you have connected a device, it should automatically appear in the *Devices* section. When you right click and choose "*Add device for development*" the provisioning profile should be generated and automatically installed on your device. If everything is set up the device should be marked with

a green button and you should be able to see a provisioning profile named *Team Provisioning Profile* for your device. Furthermore you should be able to see your provisioning profile matching your certificate in your applications info.plist file in the *Build* Tab under the *Code Signing Identity* section. You can find the info.plist file in XCode under targets, it is named like your application. Just leave the selection to automatic selection and you should be able to run your application on the device by selecting *Device* from the big button on the top left in XCode before you hit *Build and Run*. An issue which you still could stumble across is if you do not have the right version of iOS on your device. Also you might have to change your bundle identifier. The bundle identifier is a unique identifier for your application. In my experience this should not cause any problems during the development phase. But if you get such an error you should know the bundle identifier can be specified in the *Properties* tab of your info.plist file. By convention the bundle ID should be named something like *com.companyName.appName (top-level Internet domain.companyName.appName)* and is being defined based on the AppID you are using for provisioning. If you experience any kind of problem you should be able to solve them by reading the respective documents in the provisioning portal carefully.

3 Requirements

In this chapter we describe all the requirements we determined for our application. We defined most of the requirements before we started with developing of course. However, some functionality and thus requirements emerged during the process of implementation.

3.1 Requirements defined before implementation

3.1.1 Calculation of h- and g-indexes

The key functionality of our application is the calculation of h- and g-indexes based on the data we retrieve from Google Scholar. In the following, we sometimes will refer to these two indexes as scholarly indexes.

3.1.2 Manage search results

Display the calculated indexes and the publications which the results are based on

After the calculation we need to display the results in a comprehensible way. The user needs to be able to check the publication his or her results are based on.

Function to edit results by deleting and restoring publications

Search results can contain publications from not requested authors, i.e. when he has the same name as the requested author. Therefore the user must be able to manually edit his/her results by deleting, merging and restoring publications. This means when one publication is being deleted, the results have to be recalculated, because the deleted publication does not contribute to the indexes any longer. Restoring must behave conversely. Merging is being explained in section 3.1.3 below.

Manage stored search results: save, display and delete

We need functionality for managing search results. The search results not only contain the indexes but all publications the indexes are based on. We want to be able to store and delete the indexes and edit the corresponding set of publications at any time. Again editing the set of publications refers to selecting, deselecting and merging publications for the calculations of the results.

Functionality to email the results

The application should have functionality to email those results in the form of a pdf document.

3.1.3 Merging publications

Google Scholar sometimes delivers the very same publication as two or even more different publications, i.e. due to variations in the title or the same title but different publication types or versions. This might distort the results. A functionality which allows to merge multiple publications and hence their citation counts is required to fix this issue.

3.2 Requirements emerged during the implementation process

3.2.1 Graph Feature

For an author of a publication it might be interesting to see how his publications are evolving over time. Therefore we conducted to implement a graph feature as it is possible to retrieve citation counts per year.

Functionality to email the graph

It would be nice to be able to email the evolution graph as a pdf or image.

3.2.2 Comparison Feature

The scholarly indexes provide some kind of measurement of the success and impact of an author. But the indexes are not very useful until they are mapped to some kind of scale. Thus it would be useful to compare someone's results with others of the same area of research. For comparison, we thought it would be best to display two results in the same hierarchical context. This feature also allows to analyse changes or even the evolution of a scholarly index as it is possible to limit search requests to specific time intervals. Another option is to submit a search, save the results and submit the exact same search at some point in the future again. If you then compare the results of both, you can see if the indexes increased or not. You will even see if the citation counts of specific publications increased.

3.2.3 Remember Feature

If you request the same search once in a while, maybe to check if there are new citations which effect the indexes, you would have to edit the search results each time. Therefore a feature would be nice which can handle the editing of the search results for you based on the latest previously submitted search and the corresponding undertaken editing.

4 Architecture

4.1 Controller Hierarchy

Chapter 3 describes the different features that we initially defined for our application. Before we could start developing we had to discuss how to structure these features in our application. As a result we identified two main functionalities which should be accessible right away, the calculation of the two indexes as the key feature and some kind of a results manager. Furthermore we would have to include instructions and information about the application which eventually resulted in three dividable parts. With that in mind, we could think about how to realize this in our application and found the *UITabBarController* suitable for sectioning our functionalities.

4.1.1 UITabBarController Class

The *UITabBarController* class is a template for a special controller. A controller for switching between other view controllers. In the *UITabBarController* class reference [5] Apple says that this class is not intended for sub-classing. Instead you should use instances of it as-is to present an interface that allows the user to choose between different modes of operation. A tab bar at the bottom of the controller's interface allows to select one of the provided modes. Each tab is associated with a specific view controller. When the user selects a tab, the tab bar controller presents the root view of the corresponding view controller. All previous views are being replaced, even if the tab was previously selected. With the *selectedViewController* property you can choose the controller which is being displayed initially. By conforming to the *UITabBarControllerDelegate*, objects can be notified about the interactions of the tab bar controller.

XCode provides a tab bar application template. It was the starting point for our application. It includes three XIB files of which the *MainWindow.xib* was the important one to us. It contains a tab bar controller serving as the root controller of our application. The template automatically adds the view of the tab bar to the application's window. We removed all useless parts like the two other XIB files and the *FirstViewController* object. In Interface Builder we added three controllers of the type *UINavigationController* (Compare section

4.1.2) to the tab bar controller, each one serving as the root controller of one tab. All four controllers are being instantiated from the *NIB* file, not requiring us to write a single line of code. Each navigation controller in turn, holds a custom root view controller. In our application these are controllers of the types *SearchViewController*, *HistoryTableViewController* and the *AboutViewController* as you can see in figure 4.1. Figure 4.2 further illustrates that each of the mentioned controller manages a specific part of the user interface.

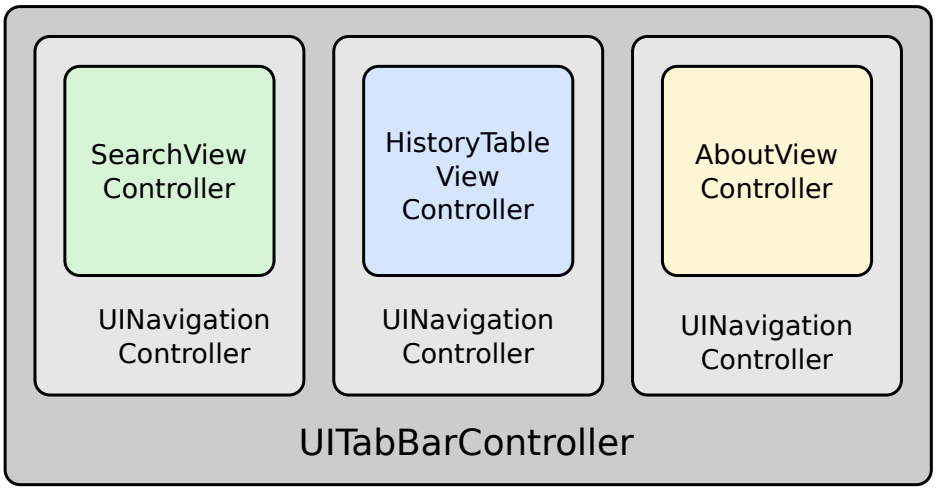

Figure 4.1: UITabBarController with one UINavigationController per tab

4.1.2 UINavigationController Class

Similar to the *UITabBarController* class the *UINavigationController* class implements a special view controller for switching between other view controllers. Again (Compare section 4.1.1) you are not supposed to subclass it but use unmodified instances of it for presenting a hierarchical user interface. A navigation controller therefore keeps track of the hierarchy in a navigation stack and provides a back button for moving back in the navigation hierarchy. To modify the stack, and therefore navigate in your application, the navigation controller provides two methods, *pushViewController: animated:* and *popViewControllerAnimated:*. When you push your custom view controller on the stack, its view will be displayed and the navigation controls will be updated. The *popViewControllerAnimated:* method is used to navigate back in the hierarchy. This functionality usually is provided by the back button in the navigation bar of the navigation controller. The navigation controller provides the *UINavigationControllerDelegate* protocol to notify other objects about its navigation.

Figure 4.3, 4.4 and 4.5 show the complete navigation structure of the three navigation controllers managed by the tab bar controller. The three figures represent the entire navigation

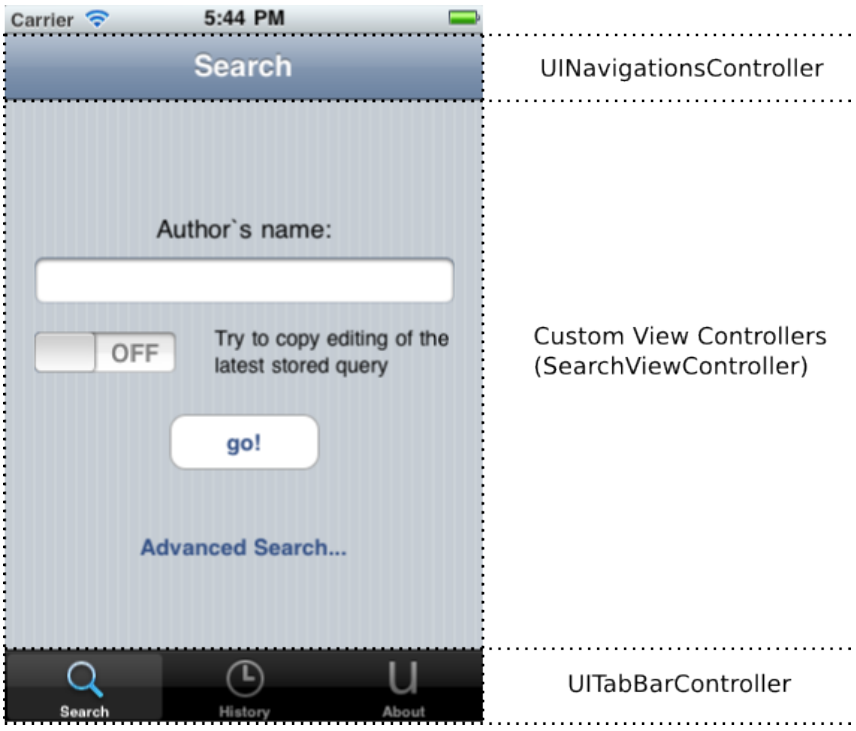

Figure 4.2: The user interface managed by multiple controllers

flow of the application. There are of course other controllers and classes. But all controllers which are part of the navigation tree have corresponding graphical user interfaces which they present at the respective points of the application flow.

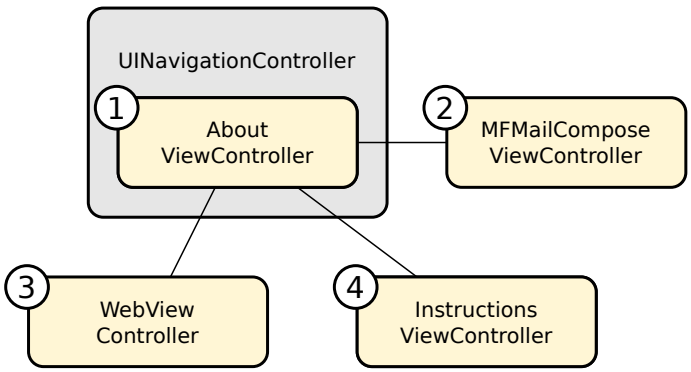

Figure 4.5: Navigation-tree of the UINavigationController of the right tab

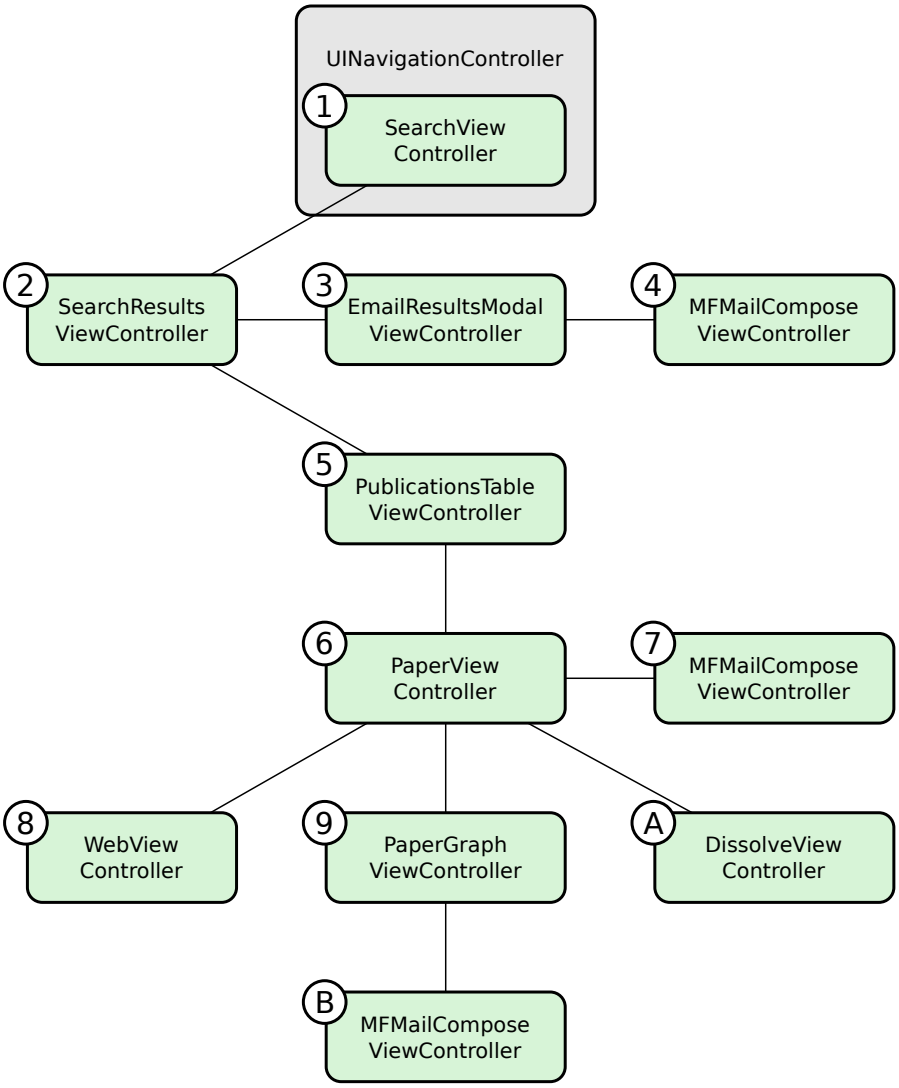

Figure 4.3: Navigation-tree of the UINavigationController of the left tab

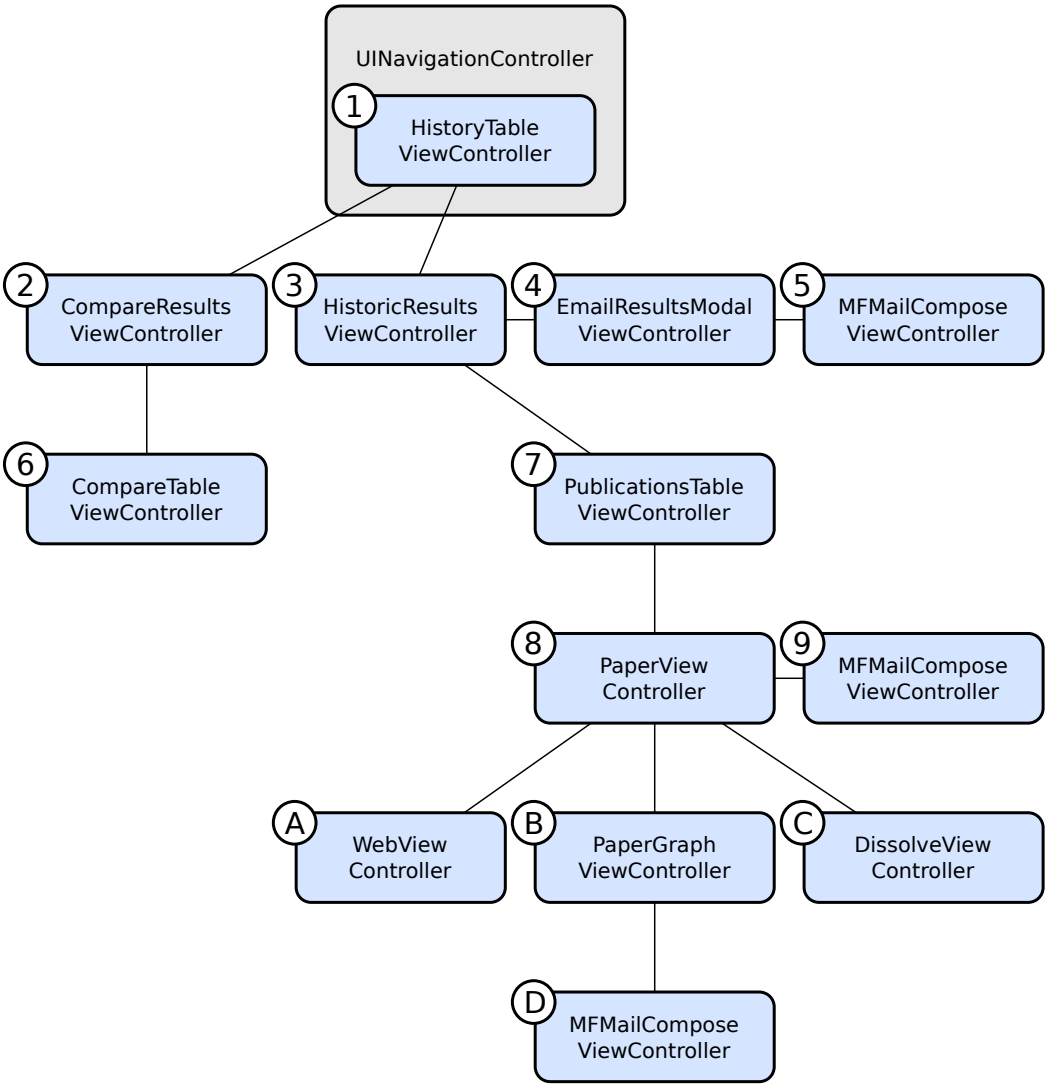

Figure 4.4: Navigation-tree of the UINavigationController of the tab in the middle

4.1.3 Custom controllers loaded through the MainWindow NIB

Our *MainWindow.xib* file contains a controller of the class *HistoryTableViewController*. It inherits from the class *UITableViewController*, which is a special controller for presenting data in the form of a table. It does not have a corresponding XIB file as the user interface is loaded programmatically. We will talk about the *UITableViewController* class in detail later. The *SearchViewController* class as well as the *AboutViewController* class both have their own XIB files which contain the corresponding user interfaces.

4.2 Custom Controller Inheritance

All controllers in our application inherit from the *UIViewController* class provided by the UIKit framework. Because the custom controllers of our application share properties and functionalities it makes sense to structure them into an inheritance-hierarchy like the controllers of the UIKit framework (Compare section 2.3). Therefore code can be reused and is easier to maintain. Figure 4.6 illustrates the following explanation.

4.2.1 UIKit Controllers

The yellow controllers of the types *UIViewController* and *UITableViewController* both are included in the UIKit framework. All controllers inherit from the class *UIViewController*.

4.2.2 Super Controllers

The red controllers are custom controllers providing different levels of functionality which other controllers may implement through inheritance.

BasicViewController

The *BasicViewController* class includes a reference to the database controller and an id to the database entry it manages. Furthermore it provides two basic methods. One to get the current year and another to display an alert message to the user.

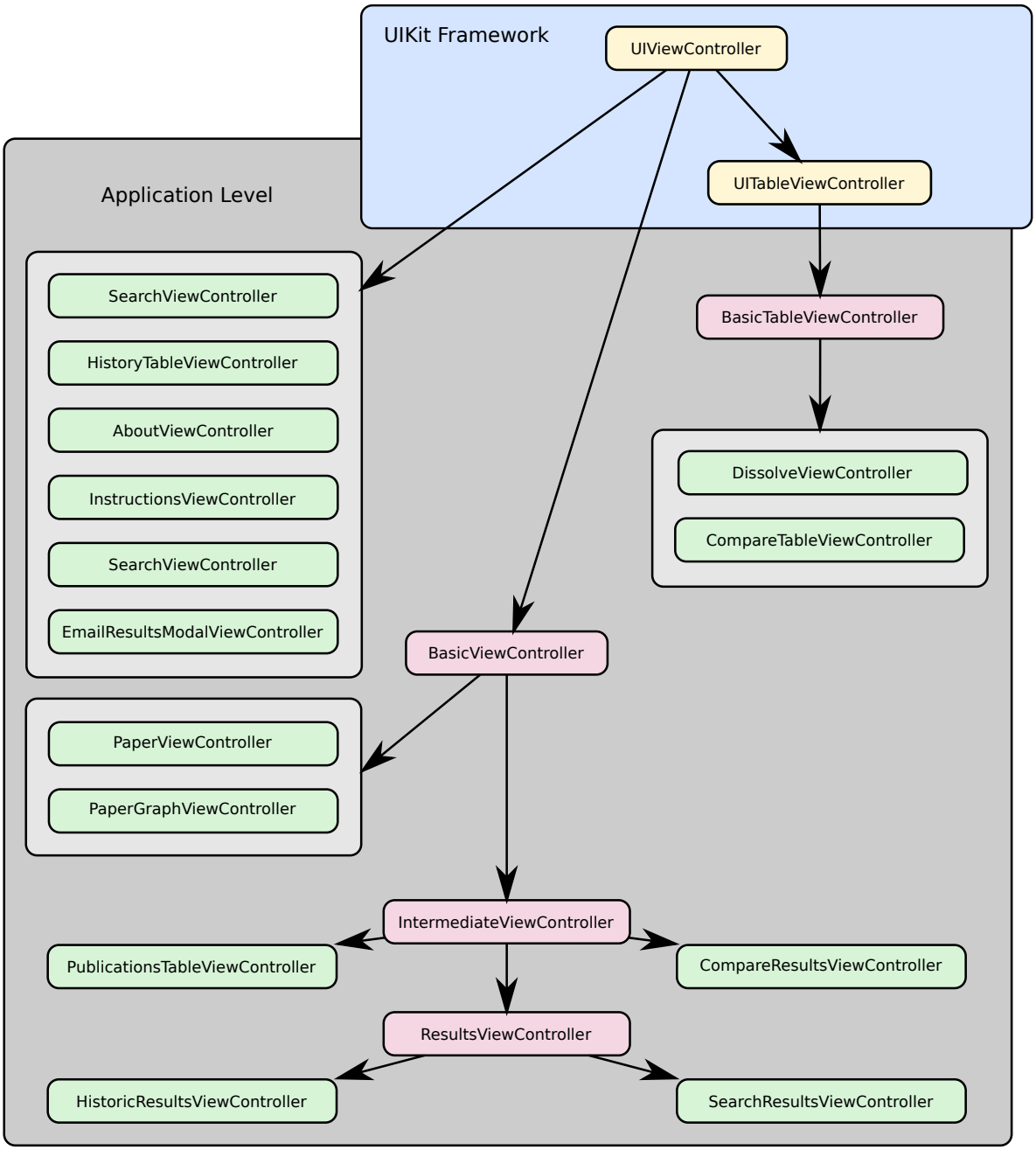

Figure 4.6: Custom Controller Inheritance Hierarchy

BasicTableViewController

The *BasicTableViewController* class only provides a reference to the database controller and an id to the database entry it manages. It inherits from the class *UITableViewController* and therefore primarily is for displaying data in the form of a table.

IntermediateViewController

The *IntermediateViewController* class extends the functionality of the *BasicViewController* class by two methods for the calculations of the two scholarly indexes and further references for holding a set of database entries and the two indexes. Additionally it provides a special view for communicating that it is busy.

ResultsViewController

The *ResultsViewController* class extends the functionality of the *IntermediateViewController* class by a several properties and methods, all to display and manage results. Another characteristic is the ability to send email.

4.2.3 Custom Controllers

The green controllers are custom view controllers which inherit their basic functionality from one of the provided controller classes and extend their functionality in order to fulfil specific tasks. These controllers actually get instantiated in code.

4.3 Parsing and Database Functionality

4.3.1 Parsing

All parsing functionality is being encapsulated by the *Parser* class. The only controller object which needs to communicate with the parser object is the *SearchViewController* object as it serves the data to be parsed. The communication between both objects is being implemented through delegation (Compare section 2.3.3). The parser object also has a reference to a *SQLiteController* object which is instantiate by the search view controller and only passed on to the parser object when it is being created. More on this in the next section 4.3.2. For a visualization see figure 4.7.

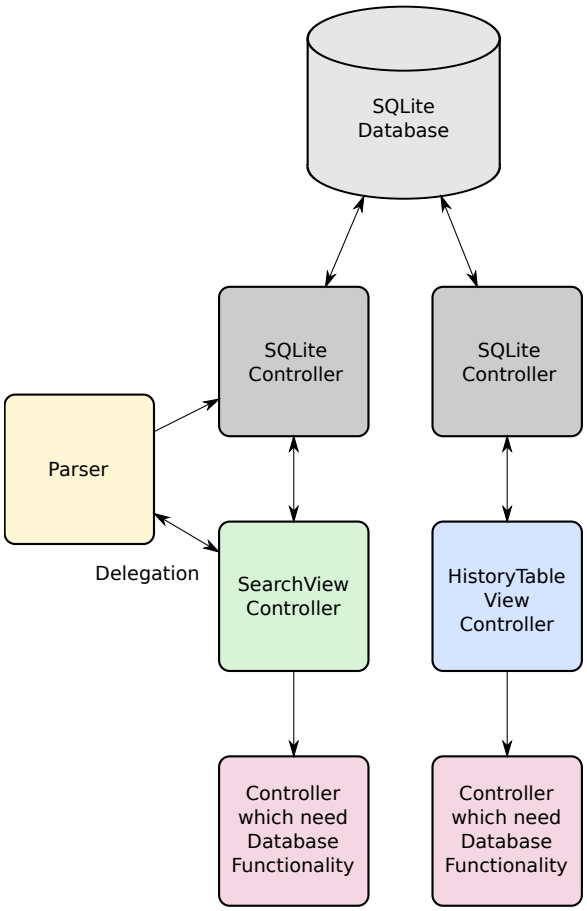

Figure 4.7: Database Access

4.3.2 Database Access

There are two controllers which actually instantiate a *SQLiteController* object in order to access the database: the search view controller and a controller of the class *HistoryTableViewController*. Both controllers then pass on their reference to other objects when these are being created and require database functionality. See figure 4.7. None of our controllers belonging to the third tab of our tab bar controller needs to access the database. Alternative data persistence technologies are being discussed in section 5.2.6. Section 5.2.7 explains the integration of the SQLite library.

In our database we want to store the results of a search. Therefore we store the search parameters in a table called *queries*. In order to store the publications of a search we create a new *results* table for each time the user submits a search. The relation between query

and results table is based on the auto incremented ID field of the query entries, simply by naming a corresponding results table *results_queryID*. Figure 4.8 illustrates the binding between the query entries and the results tables. Figure 4.9 and 4.10 show all fields of the corresponding tables.

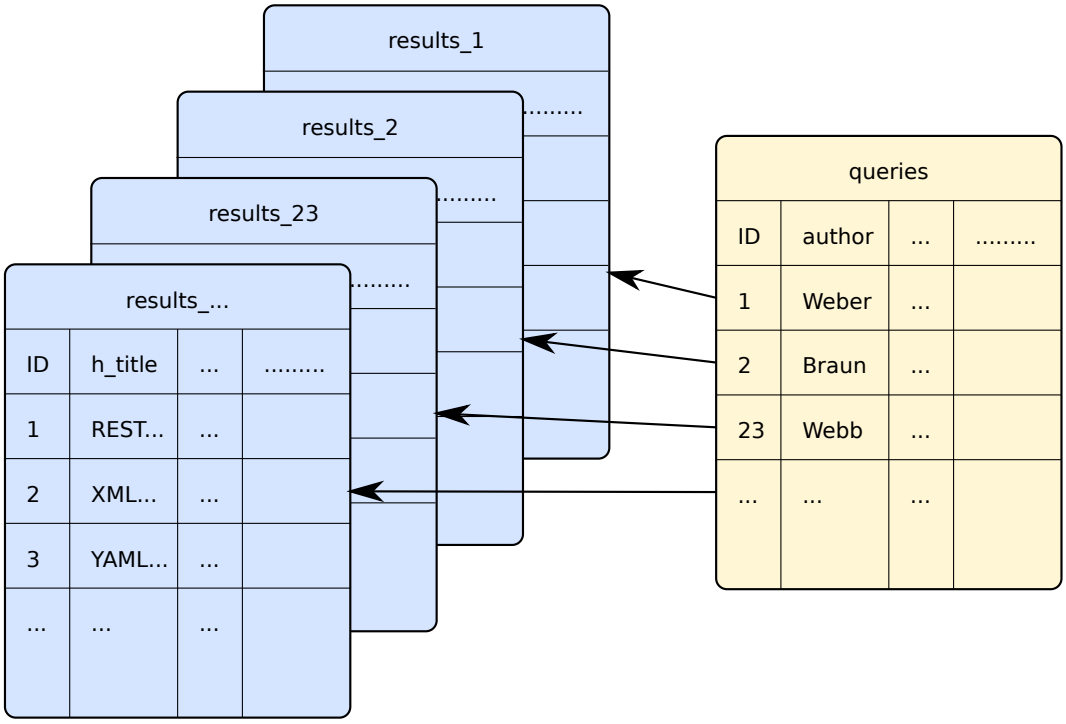

Figure 4.8: Table relations

Column ID	Name	Type	Not Null	Default Value	Primary Key
0	id	INTEGER	0		1
1	author	VARCHAR(100)	0		0
2	waotw	VARCHAR(100)	0	'NotSet'	0
3	wtep	VARCHAR(100)	0	'NotSet'	0
4	walootw	VARCHAR(100)	0	'NotSet'	0
5	wtw	VARCHAR(100)	0	'NotSet'	0
6	ylo	VARCHAR(10)	0	'NotSet'	0
7	yhi	VARCHAR(10)	0	'NotSet'	0
8	each	INTEGER	0	1	0
9	bio	INTEGER	0	0	0
10	bus	INTEGER	0	0	0
11	chm	INTEGER	0	0	0
12	eng	INTEGER	0	0	0
13	med	INTEGER	0	0	0
14	phy	INTEGER	0	0	0
15	soc	INTEGER	0	0	0
16	url	VARCHAR(200)	0	'NotSet'	0
17	keep	INTEGER	0	0	0
18	created_at	VARCHAR(20)	0	'NotSet'	0

Figure 4.9: Queries table structure

Column ID	Name	Type	Not Null	Default Value	Primary Key
0	id	INTEGER	0		1
1	h_title	VARCHAR(100)	0	'NotSet'	0
2	h_type	VARCHAR(10)	0	'NotSet'	0
3	h_link	VARCHAR(100)	0	'NotSet'	0
4	h_href	VARCHAR(100)	0	'NotSet'	0
5	d_title	VARCHAR(100)	0	'NotSet'	0
6	d_link	VARCHAR(50)	0	'NotSet'	0
7	d_href	VARCHAR(100)	0	'NotSet'	0
8	d_type	VARCHAR(100)	0	'NotSet'	0
9	cites	INTEGER	0	0	0
10	cites_href	VARCHAR(100)	0	'NotSet'	0
11	authors	VARCHAR(100)	0	'NotSet'	0
12	publication	VARCHAR(100)	0	'NotSet'	0
13	publisher	VARCHAR(100)	0	'NotSet'	0
14	year	VARCHAR(10)	0	'NotSet'	0
15	rejected	INTEGER	0	0	0
16	merge_count	INTEGER	0	0	0
17	merged_with	INTEGER	0	0	0
18	description	VARCHAR(300)	0		0

Figure 4.10: Results table structure

4.3.3 Outsourcing parsing to a server

As an alternative to parsing the data on the device, we could parse it on a webserver. Of course a server could parse data much faster than our mobile devices and implement a central caching mechanism so that data would be available rapidly. The described scenario typically would be implemented using a webservice architecture. Our application, as the webservice client, would have to implement functionality for consuming the webservice provided by the server. Unfortunately the iOS SDK does not provide a special framework for consuming webservices and we would have to implement this functionality ourselves using a compatible XML parser. There are other reasons why we did not choose to use a

webservice for data processing. The first and most crucial reason is that we wanted to learn about the software and hardware potentials of the device and thus constituted to perform all tasks on the device right from the start (See chapter 1). Secondly we would have had to face the problem of concealing the identity of the server because Google Scholar restricts the amount of requests in a certain amount of time based on the IP address. Depending on how many users would try to use the application, and thus the webservice, our server would get blocked quickly without respective "concealing" technique. There might be solutions to the challenge of building such a "concealing" mechanism. But on the other hand it would require a server to run all the time which also would be more expensive. The risk of getting blocked by Google also applies to the concept of performing all tasks on the device as one simply can use the features which involve requesting data from Google Scholar too much. Fortunately the blocking of an IP will be removed after a few hours. As introduced later in section 5.2.7, the application tries to prevent the user from getting blocked quickly by using caching mechanisms.

5 Implementation

5.1 Calculation of h- and g-indexes

5.1.1 Development of a Google Scholar API

In order to calculate the h- and g-indexes of a scholar we need to get the necessary data first. The data in this case means an initially unknown amount of citation counts of the best cited publications of the scholar of interest. In some cases all citation counts of all publications might be required for the calculations. As we have mentioned earlier, we chose Google Scholar to serve as our data source. Unfortunately Google Scholar does not provide an API for their services. But Google provides an advanced search form at `http://scholar.google.com/advanced_scholar_search?hl=en&as_sdt=2000` (See figure 5.1) which allows querying exclusively for publications of a certain scholar by specifying the name of the scholar as the author in the form. Figure 5.1 shows a screen shot of the advanced Google Scholar search from. Submitting the search form, the input form data is being encoded in the URL. For example if we specify the following search values

With the exact phrase:	"http"
Author:	"Roy Fielding"
Subject areas:	Engineering, Computer Science, and Mathematics

and submit the form, the resulting URL therefore is:

```
http://scholar.google.com/scholar?as_q=&num=10&btnG=Search+Scholar&
as_epq=HTTP &as_oq=&as_eq=&as_occt=any& as_sauthors=Roy+Fielding &as_
publication=&as_ylo=&as_yhi=&as_sdt=1& as_subj=eng &as_sdts=5&hl=en
```

This means it is possible to directly retrieve the requested search results in HTML format, assuming the URL therefore is known. In order to retrieve specific data it is necessary that

we are able to construct the corresponding URL independently from the Google Scholar form. Hence, the first step towards retrieving and processing Google Scholar data in an iPhone Application was to analyse and understand the URL encoding structure of the advanced Google Scholar search form.

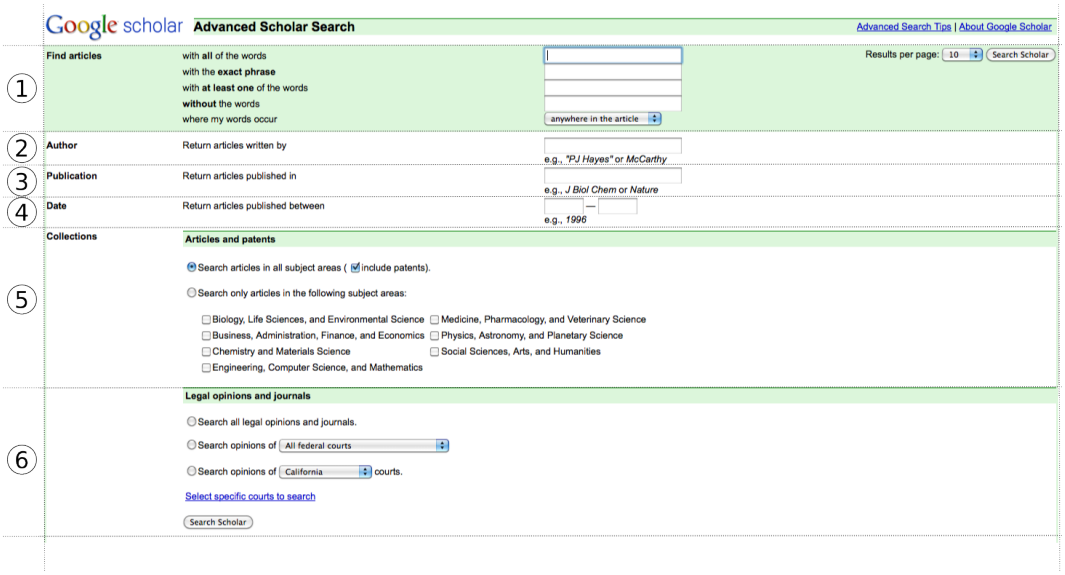

Figure 5.1: Google Scholar Advanced Search Form

All input field values can be found assigned to the corresponding variables in the URL, after submitting the form. In addition, the URL encodes the language in which the results will be displayed in, as well as some other variable value combinations which could not be classified definitely.

Form Field Name	URL Variable	Value Type
with all the words	as_q	Text
with the exact phrase	as_epq	Text
with at least one of the words	as_oq	Text
without the words	as_eq	Text
where my words occur	as_occt	Text
Results per page	num	10, 20, 30, 50, 100

Table 5.1: Find Articles - See corresponding section (1) in figure 5.1

Form Field Name	URL Variable	Value Type
Author	as_sauthors	Text

Table 5.2: Author - See corresponding section (2) in figure 5.1

Form Field Name	URL Variable	Value Type
Publication	as_publication	Text

Table 5.3: Publication - See corresponding section (3) in figure 5.1

Form Field Name	URL Variable	Value Type
Date	as_ylo, as_yhi	year (From - To)

Table 5.4: Date - See corresponding section (4) in figure 5.1

all possible search types (See table 5.8)	as_sdt	1, 1., 2, 3, 4
Articles and Patents		
Form Field Name	**URL Variable**	**Value Type**
Search articles in all subject areas	as_sdt	1.
Include Patents	as_sdtp	on
Search only articles in the following subject areas:	as_sdtp	1
Biology, Life Science and Environmental Science	as_subj	bio
Business, Administration, Finance and Economics	as_subj	bus
Chemistry and Material Science	as_subj	chm
Engineering, Computer Science and Mathematics	as_subj	eng
Medicine, Pharmacology, and Veterinary Science	as_subj	med
Physics, Astronomy, and Planetary Science	as_subj	phy
Social Sciences, Arts, and Humanities	as_subj	soc
Legal opinions and journals		
Form Field Name	**URL Variable**	**Value Type**
Search all legal opinions and journals	as_sdt	2
Search only US federal court opinions	as_sdt	3
Search only court opinions from the following states	as_sdt	4

Table 5.5: Collections - See corresponding section (5) in figure 5.1

Form Field Name	**URL Variable**	**Value Type**
Language	hl	en (English), de (German), ...

Table 5.6: Language Encoding

Explanation	URL Variable	Value Type
Usually representing the search type; assigned values differ from the structure	as_sdt	2000, 2001, ...
US State seems to be set to California (value 5) as default, even though the US state selection is not included in the search as we always search for articles and only one search type can be selected at the same time	as_sdts	5

Table 5.7: Unclassified Variable Value Combinations:

In the collections section (5) of the advanced search form, it is possible to define a search type. By default the search type is set to *Search articles in all subject areas*. Search types cannot be combined. Table 5.8 contains all search types and the corresponding values.

Search Type	Variable Value
Search articles in all subject areas	1.
Search only articles in the following subject areas	1 (with multiple additional as_subj)
Search all legal opinions and journals	2
Search only US federal court opinions	3
Search only court opinions from the following states	4

Table 5.8: The variable "as_sdt" usually defines the search type

After the search results was submitted all results are being displayed in the browser (Section (2) in figure 5.2). In section (1) of figure 5.2 on the right side, you can see the amount of results discovered by your search. Even though a search can retrieve much more than a thousand results, Google only allows to view 1000 results. If we choose to display 100 results per page, we receive a maximum of 10 result pages. This is more then enough for the calculation of the h-index and also should be sufficient for the calculation of the g-index as it is as good as impossible that a scholar will achieve a g-index of more than one thousand. For example the best rated chemist "Whitesides, G. M.", according to the article "Hirsch index ranks top chemists" [46] by Richard Van Noorden has an approximate g-index of 257

based on a rough calculation.

The results page provides the ability to limit the results (Section (1) in figure 5.2) and navigate through all results via the typical Google navigation (Section (3) in figure 5.2). Following one of these links sometimes had an assignment of the values “2000”, “2001”, ... to the variable “as_sdt” as a side effect. These values could not be classified. Also their absence did not seem to affect the search results in any way. Another special variable is the “as_sdts” variable. It was classified as holding the value for the US states. But these exclusively occur in the form when searching for "opinions from a court in a specific US state”. It seemed always to be set to the default value 5, standing for the state of California.

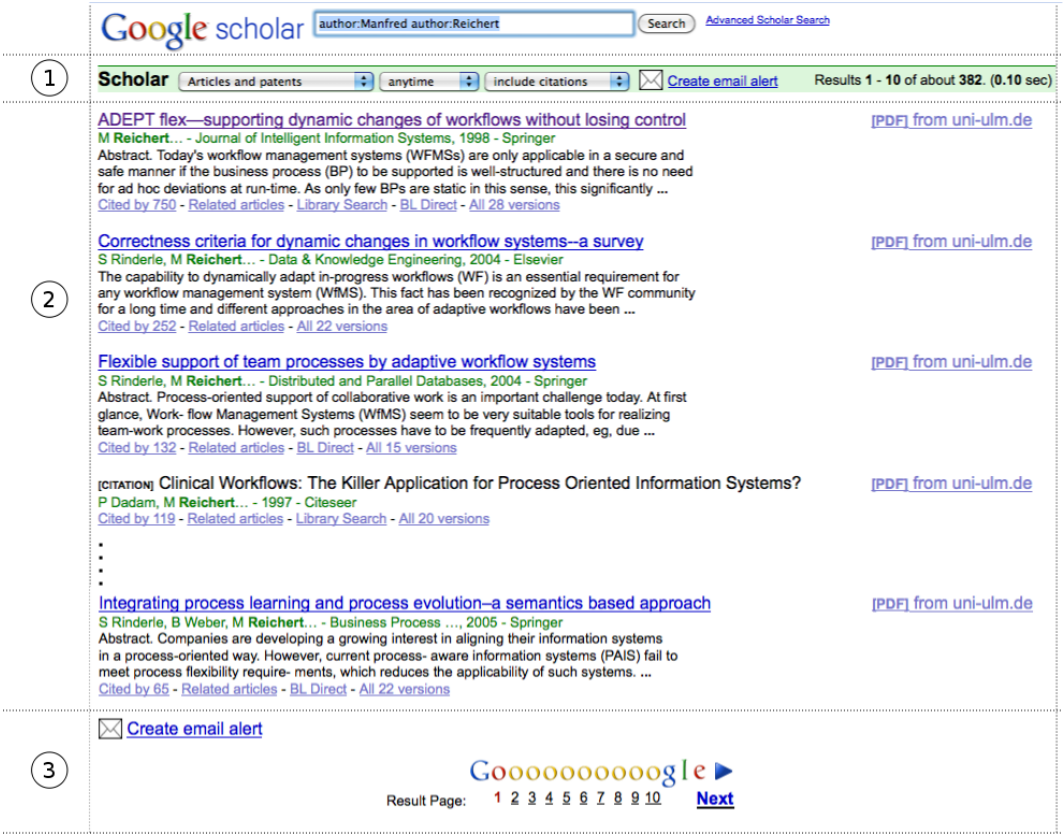

Figure 5.2: Google Scholar Search Results

One variable and multiple values

In all the text fields, it is possible to enter multiple values separated by spaces. In the URL these values are being combined using a "+" operator.

The variable "as_sdts" as well as the variable "as_subj" can appear multiple times in the same URL as it is possible to select multiple US states and multiple subject areas in the same form.

With the gained knowledge about the encoding structure, it is possible to directly retrieve the same search results (in HTML format) as with the advanced Google Scholar form, by constructing the corresponding URL and requesting it via HTTP. As a result we are now able to retrieve any search results in our iPhone application, simply by using *NSURLConnection* combined with *NSMutableURLRequest* and the corresponding *NSURL*. These classes are part of the *Foundation framework* which itself is part of iOS SDK. The three classes are part of the so called *URL loading system* [18], which contains classes and protocols for interacting with URLs and communicating with servers using standard internet protocols.

5.1.2 Analysis of the HTML document structure for parsing

Now that we are able to collect the necessary HTML data, we need to extract the actual values in which we are interested in from the HTML. Thus we need a parser which is able to parse HTML data. There are several parsers available which can be used in an iPhone application. Apple provides the sample application *XMLPerformance* [19] in the iOS Reference Library which explores two approaches to parsing XML in iOS. The *Readme.txt* file [20] of the project states: "*The iPhone SDK provides two APIs for parsing XML. At the Objective C level, NSXMLParser implements an event-driven approach... The other API in the SDK, the C library "libxml2", has a similar approach known as SAX (Simple API for XML).*" Furthermore there are several third party libraries available which can be integrated into iPhone applications. Amongst others, there are TouchXML, TBXML and KissXML.

In order to choose an appropriate parser for the application, the HTML structure of the Google Scholar result pages had to be studied before. The easiest way to do that simply is to submit an advanced search in a browser and copy and paste the resulting HTML in an appropriate editor. Doing that, the first thing that strikes someone, is that the HTML is very messy. Quotation marks around class tags are missing, the DOM structure is incomplete and the HTML document does not even contain a DOCTYPE declaration. In summary the

HTML is far from being valid and definitely should not, and most likely cannot be parsed in its original format. By examining single results, a comprehensible structure can be found. Searching for “Manfred Reichert” as the author in the advanced Google Scholar Search form, we can extract the HTML DIV containing the values of the first publication result "Adept flex-supporting dynamic changes of workflows without losing control" as illustrated by listing 5.1. Figure 5.3 shows the corresponding result displayed by the browser.

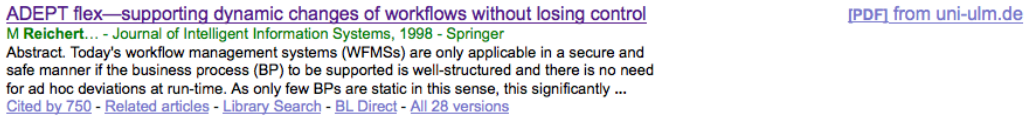
ADEPT flex—supporting dynamic changes of workflows without losing control [PDF] from uni-ulm.de
M Reichert... - Journal of Intelligent Information Systems, 1998 - Springer
Abstract. Today's workflow management systems (WFMSs) are only applicable in a secure and safe manner if the business process (BP) to be supported is well-structured and there is no need for ad hoc deviations at run-time. As only few BPs are static in this sense, this significantly ...
Cited by 750 - Related articles - Library Search - BL Direct - All 28 versions

Figure 5.3: Google Scholar Search Results

Listing 5.1: HTML code of a result

```
<div class=gs_r>
        <div class=gs_rt>
                <h3>
                        <a href="http://www.springerlink.com/index
                            /W852302K53V81700.pdf" class=yC0>ADEPT
                            flex-supporting dynamic changes of
                            workflows without losing control
                        </a>
                </h3>
        </div>
        <span class="gs_ggs gs_fl">
                <a href="http://citeseerx.ist.psu.edu/viewdoc/
                    download?doi=10.1.1.38.8630&amp;rep=rep1&amp;
                    type=pdf" class=yC1>
                        <span class=gs_ctg2>[PDF]</span> from psu.
                            edu
                </a>
        </span>
        <font size=-1>
                <span class=gs_a>M <b>Reichert</b>&hellip; -
                    Journal of Intelligent Information Systems,
                    1998 - Springer</span>
                <br>
                Abstract. Today&#39;s workflow management systems
                    (WFMSs) are only applicable in a secure and
```

```
17                  <br>
18                  safe manner if the business process (BP) to be
                        supported is well-structured and there is no
                        need
19                  <br>
20                  for ad hoc deviations at run-time. As only few BPs
                         are static in this sense, this significantly <
                        b>...</b>
21                  <br>
22                  <span class=gs_fl>
23                          <a href="/scholar?cites
                                =13863410738913124073&amp;as_sdt=2005&
                                amp;sciodt=2000&amp;hl=en">Cited by 733
                                </a>
24                          - <a href="/scholar?q=related:6Z5AlJ26ZMAJ
                                :scholar.google.com/&amp;hl=en&amp;
                                as_sdt=2000">Related articles</a>
25                          - <a href="http://www.worldcat.org/oclc
                                /312690337" class=yC2>Library Search</a
                                >
26                          - <a href="http://direct.bl.uk/research/1A
                                /63/RN042324139.html?source=
                                googlescholar" class=yC3>BL Direct</a>
27                      - <a href="/scholar?cluster
                            =13863410738913124073&amp;hl=en&amp;as_sdt
                            =2000">All 28 versions</a>
28                  </span>
29          </font>
30  </div>
```

In order to parse the HTML, we have to analyse the structure of the HTML, so that we know where in the DOM tree, we can retrieve which values. To gain this knowledge, an analysis of several search result pages had to be carried out. As a result, listing 5.2 shows the basic structure of a single publication-result. The invalid HTML code was being corrected.

Listing 5.2: Pseudo HTML Structure

```
<div class="gs_r">
        <div class="gs_rt">
                <h3>
                <!-- 1 -->
                        <span class="gs_ctc">Link-type ([BOOK],[
                            PDF],...)</span>
                        <!-- 2 -->
                        <a href="url to the publication resource"
                            class="yC0">publication name</a>
                </h3>
        </div>
        <span class="gs_ggs gs_fl">
                <!-- 3 -->
                <a href="url for download" class"yC1">download-
                    resource
                        <span class="gs_ctg2">download-link-type
                            ([PDF],[HTML],[PS],[DOC],...)</span>
                </a>
        </span>
        <font size="-1">
                <!-- 4 -->
                <span class="gs_a">Authors - Publications, Year -
                    Publisher</span>
                <!-- 5 -->
                Short preview of the publication
                <!-- 6 -->
                <span class="gs_fl">
                        <a href="url to all results which cite
                            this publication">Cited by #</a>
                        <a href="url to related articles">Related
                            articles</a>
                        <a href="url for executing a library
                            search">Library Search</a>
                        <a href="url to british library direct">BL
                             Direct</a>
                        <a href="url to all versions of the
                            publication">All # versions</a>
                </span>
```

```
29          </font>
30  </div>
```

The structure above contains all relevant elements with their relevant pseudo attributes and pseudo values. Figure 5.4 illustrates the mapping from the visual browser representation of a result to the code of listing 5.2. You may have noticed that there is no visual representation for the code snippet marked with $<! --1-- >$. This is because not all HTML elements appear in each result, making parsing even more complex.

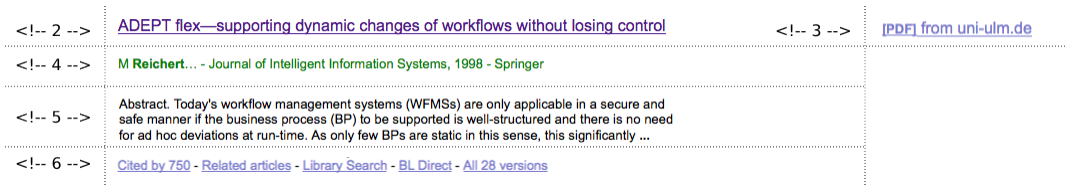

Figure 5.4: Visual representation

The following set of rules about the appearance of the HTML elements, supplements the basic structure.

Rules of document structure

1. Each result is wrapped in a DIV of the class “gs_r”
2. Each result includes a DIV with the class “gs_rt” and the inner H3 element
   a) The H3 element includes either one of the elements A, SPAN or both
      i. If both elements are included, the class of the SPAN is “gs_ctc”
      ii. If the a element does not exist, the class of the SPAN is “gs_ctu”
3. The SPAN with the classes “gs_ggs” “gs_fl” is optional
   a) If the SPAN exists, it contains an A element and a SPAN element
   b) The A element is of the class “yC. . . “
      i. In the “yC. . . ” class, the dots are being replaced by a counter.
      ii. The counter counts all A elements of the whole document with the classes “yC. . . ” starting with zero, counting hexadecimal
   c) The inner SPAN element is of the class “gs_ctg2”

4. Each result includes the font element (its attribute size is set to -1)
    a) The font element includes a SPAN with the class “gs_a”
        i. The SPAN element includes authors, publication, publication year and publisher in the format: authors - publication, year – publisher
        ii. At least authors is included, all others are optional
    b) The font element optionally includes a SPAN with the class “gs_fl”
        i. The SPAN at least one of seven A elements, the most important is the “Cited by #” A element, as it contains the amount of publications which cite this publication and the link to those. The other links are: Related articles, Library Search, BL Direct, All Versions, View as . . . , Cached
    c) The font element optionally contains a short preview of the publication as its content

With the dynamic results structure in mind, we can now concentrate on choosing an appropriate parser for our application.

5.1.3 Choosing a parser

Tree based vs. event based parsing

The first thing we have to consider for choosing a parser are the two common but fundamentally different approaches of parsing, tree based parsing and event based parsing.

Tree based parsing approaches map the XML document into an internal tree structure and allow the application to navigate through that tree. The most popular tree based API for XML and HTML documents, is the DOM API (Document Object Model) specified by the DOM working group at the W3C [50].

Event based parsing approaches report parsing events to the application through callback methods and thereby do not need to build an internal tree structure. Instead the application needs to implement handlers to deal with the different events such as the start or the end of an element. The most common tree based API is the SAX API (Simple API for XML) [48].

Both APIs have got advantages and drawbacks. DOM parsers usually keep the whole document tree structure in memory and thus usually are slower and consume more memory than SAX parsers. This needs to be considered especially for parsing large documents.

The internal tree structure on the other hand makes accessing elements easy. Some DOM parsers even allow modifying the structure of the tree in memory like adding or deleting elements. In some cases it is inefficient to build a tree structure in memory. It might not make a lot of sense to just map the data into a new structure and discard the tree subsequently. Retrieving specific data with a SAX parser in turn can become very difficult depending on the nested level and the complexity of the document structure.

At this point we know that using a SAX parser might, and probably will be difficult due to the complexity and nested level of our HTML structure. Choosing a DOM parser instead might result in slow performance and memory issues especially due to the restricted resources on a mobile platform. Another point to consider for development is that we do not have any influence on the HTML structure. If it changes, it is very likely that the parsing functionality of the application would have to be adjusted accordingly. This means the final implementation of the parser should be easy to maintain and still flexible in concern of small changes.

To finally make a decision we will implement and test the different approaches in terms of speed, maintainability and flexibility. For the following performance tests we have used an iPod Touch 8GB device of the second generation. The installed iOS version was 3.2. To actually determine the performance of certain code sections we measure the execution times in milliseconds with the data type *NSTimeInterval*. At the very beginning when we started developing our application, we used the iPhone SDK in the version 3.2 for development and the corresponding version of Apple's iPhone OS which was not named iOS yet. Short after we started developing, iOS 4.0 was released and we then used the new SDK package to develop for iOS 4.0.

As the representative of the event based parsers, we will test NSXMLParser as it implements an event driven approach similar to the SAX API. It is included in the iPhone SDK, easy to use for simple tasks and written in Objective C. Because NSXMLParser does not build a tree structure of the document, one challenge will be to implement some functionality to keep track of the document structure.

TouchXML will serve as the representative of the tree based parsers. The project description [49] states: "*TouchXML is a lightweight replacement for Cocoa's NSXML* cluster of classes. It is based on the commonly available Open Source libxml2 library.*" As mentioned by the "Readme.txt" file of Apples XMLPerformance project, the libxml2 library is a C Library available in the iPhone SDK. TouchXML provides comfortable functions for navigating the xml tree and even allows XPath queries [51]. Thus it should be uncomplicated to access

the HTML elements and retrieve the relevant data whereas building the tree might steal some time and could result in slow parsing.

Table 5.9: Performance Comparison of TouchXML and NSXM

Search Value	TouchXML	NSXML	Performance
Weber	37,23	33,65	overall processing time in sec.
	9,07	6,94	overall parse time in sec.
	1.389.831	1.390.115	overall bytes.
	153.234	200.305	bytes per sec.
Braun	36,83	36,45	overall processing time in sec.
	9,20	9,27	overall parse time in sec.
	1.418.426	1.418.142	overall bytes.
	154.177	152.982	bytes per sec.
Reichert	31,62	32,64	overall processing time in sec.
	8,97	9,30	overall parse time in sec.
	1.382.759	1.382.475	overall bytes.
	154.154	148.653	bytes per sec.
Dadam	10,95	9,99	overall processing time in sec.
	3,20	3,05	overall parse time in sec.
	405.121	405.121	overall bytes.
	126.600	132.827	bytes per sec.
Van der Aalst	32,46	30,39	overall processing time in sec.
	8,88	8,95	overall parse time in sec.
	1.169.849	1.169.567	overall bytes.
	131.740	130.678	bytes per sec.

Table 5.9: Performance Comparison of TouchXML and NSXM

Search Value	TouchXML	NSXML	Performance
Brown	35,8	36,27	overall processing time in sec.
	9,45	9,21	overall parse time in sec.
	1.400.466	1.395.537	overall bytes.
	148.197	151.524	bytes per sec.
Peter+Dadam	6,93	6,90	overall processing time in sec.
	1,91	2,00	overall parse time in sec.
	276.697	276.981	overall bytes.
	144.868	138.491	bytes per sec.
Manfred+Reichert	11,98	10,29	overall processing time in sec.
	3,48	3,46	overall parse time in sec.
	478.406	478.406	overall bytes.
	137.473	138.268	bytes per sec.
Michael+Weber	33,44	33,65	overall processing time in sec.
	9,38	9,16	overall parse time in sec.
	1.409.578	1.409.578	overall bytes.
	150.275	153.884	bytes per sec.
	144.524	149.735	average bytes per sec.

Table 5.9 shows the results of the comparison of the TouchXML implementation with the NSXML implementation. The overall processing time includes generating the query URL from the form, downloading the HTML, preprocessing the raw HTML for parsing and the actual parsing itself. One might notice that the overall amount of bytes, being processed for the same search, vary. The reason therefore is that the returned HTML in deed varies. Not just attributes, like HREF attributes containing hash keys, but also minor content parts differ from search to search. This issue does not influence the comparison of the parsers

in anyway as we calculate how long it took the parser to parse the given amount of data. But it should be kept in mind during development.

The table shows that the TouchXML implementation and the NSXML implementation behave very similar in terms of performance. Both parser implementations handle about the same amount of bytes per second and both take up a contingent of the overall processing time of less than 30%. This can be traced back to the fact that in the NSXML parser implementation, tracking the nested document structure does take some time. It also leads to less maintainable code. The TouchXML implementation in contrast supports XPath queries which makes accessing data easy. It results in more flexibility in case of changes and better maintainability as expected.

In both implementations the actual parsing time is less than 30%. This means the impact of parsing in concern of speed is limited and we could reduce the processing time by using a faster parser at maximum of less than 30%. To be able to reduce the processing time we would need a very fast parser. TBXML is another third party library parser. The project home page [36] states: "*TBXML is a light-weight XML document parser written in Objective-C designed for use on Apple iPad, iPhone & iPod Touch devices. TBXML aims to provide the fastest possible XML parsing whilst utilising the fewest resources. This requirement for absolute efficiency is achieved at the expense of XML validation and modification.*" Due to the fact that we preprocess the HTML before we parse it and meanwhile make sure it is valid, we can set validation aside and try to implement the parsing with the TBXML library.

Table 5.10: Performance Comparison of TouchXML and TBXML

Search Value	TouchXML	TBXML	Performance
Weber	37,23	27,87	overall processing time in sec.
	9,07	1,71	overall parse time in sec.
	1.389.831	1.389.831	overall bytes.
	153.234	812.767	bytes per sec.
Braun	36,83	29,54	overall processing time in sec.
	9,20	1,74	overall parse time in sec.
	1.418.426	1.418.142	overall bytes.
	154.177	815.024	bytes per sec.

Table 5.10: Performance Comparison of TouchXML and TBXML

Search Value	TouchXML	TBXML	Performance
Reichert	31,62	28,92	overall processing time in sec.
	8,97	1,74	overall parse time in sec.
	1.382.759	1.382.473	overall bytes.
	154.154	794.524	bytes per sec.
Dadam	10,95	7,98	overall processing time in sec.
	3,20	0,61	overall parse time in sec.
	405.121	405.689	overall bytes.
	126.600	665.064	bytes per sec.
Van der Aalst	32,46	25,33	overall processing time in sec.
	8,88	1,63	overall parse time in sec.
	1.169.849	1.169.567	overall bytes.
	131.740	717.526	bytes per sec.
Brown	35,8	27,85	overall processing time in sec.
	9,45	1,84	overall parse time in sec.
	1.400.466	1.396.105	overall bytes.
	148.197	758.753	bytes per sec.
Peter+Dadam	6,93	5,15	overall processing time in sec.
	1,91	0,36	overall parse time in sec.
	276.697	276.697	overall bytes.
	144.868	768.603	bytes per sec.
Manfred+Reichert	11,98	8,21	overall processing time in sec.
	3,48	0,65	overall parse time in sec.
	478.406	478.406	overall bytes.
	137.473	736.009	bytes per sec.

Table 5.10: Performance Comparison of TouchXML and TBXML

Search Value	TouchXML	TBXML	Performance
Michael+Weber	33,44	27,51	overall processing time in sec.
	9,38	1,74	overall parse time in sec.
	1.409.578	1.409.578	overall bytes.
	150.275	810.102	bytes per sec.
	144.524	764.264	average bytes per sec.

As you can see in table 5.10, the TBXML implementation beats the two others in concern of speed. In fact compared to the two others it is more than five times faster (needs only about 18,77% of the time). Unfortunately it is not possible to access content in specifically nested structures with the TBXML parser. For example it is not possible to retrieve the content “but inaccessible here” as demonstrated by listing 5.3

Listing 5.3: TBXML html parsing problem

```
<div>
        accessible here
        <p>
        and accessible here
        </p>
        but inaccessible here
</div>
```

According to the project page [36] , one of the goals is:

- XML files conforming to the W3C XML spec 1.0 should be passable.

The above structure does not violate the W3C XML Specification 1.0. But there is no way to access the “but inaccessible here” content. Because there is no straightforward way of

retrieving such nested content we cannot use TBXML in our project. Even though the parser is really fast compared to the two alternatives and accessing the elements and attributes is almost as easy as with XPath queries, accessing such nested content would require an unpleasing workaround and might cause troubles at some point. This being said, we chose to use TouchXML in our project. It supports XPath queries and thus makes accessing elements and attributes comfortable and maintainable. Using TouchXML for parsing, it takes up about 30% of the overall processing time which actually results in a maximum of about ten seconds exclusively for parsing the HTML of one search query. This is based on the fact that Google returns only a maximum amount of 1000 results which we choose to receive spread on a maximum of 10 HTML documents with 100 results each. (Compare 5.1.1) This means if we are using TouchXML instead of TBXML we have to accept that the application needs about eight additional seconds for parsing the maximum of 10 HTML documents.

5.1.4 Calculation Algorithms

H-Index Calculation

The h-index is defined as the amount h of publications with a citation count $>= h$. This leads to the following basic algorithm.

Listing 5.4: H-Index Calculation Algorithm

```
1   //First we have to order all publications descending based
        on the citation counts
2   allPublications.orderDesc();
3
4   int citationCount = 0;
5
6   for(int i = 0; i < allPublications.length; i++){
7           citationCount = allPublications[i].citationCount;
8           //i publications with at least i citations
9           if(citationCount < i+1){
10                  break;
11          }
12  }
13  return i;
```

Similarly to the h-index the g-index is defined as the unique number such that the g most cited articles together received at least g^2 citations. Thus the algorithm can be retrieved by

a simple transformation to the algorithm for calculating the h-index.

Listing 5.5: G-Index Calculation Algorithm

```
1         //First we have to order all publications descending based
              on the citation counts
2         allPublications.orderDesc();
3
4         int citationCount = 0;
5
6         for(int i = 0; i < allPublications.length; i++){
7                 //i papers with at least i citations
8                 citationCount += allPublications[i].citationCount;
9                 if(citationCount < (i+1)*(i+1)){
10                        break;
11                }
12        }
13        return i;
```

For both calculations we retrieve all relevant publications of a search from the database in descending order based on their citation counts.

5.1.5 The Search Flow

The search view controller plays a major role in our application as this is the controller which implements all parts of the data retrieval and processing. Just like many other controllers classes of our application, the *SearchViewController* class is a standard *UIViewController* class. It is loaded directly when the application is launched. It contains the interface to Google Scholar and functionality to request and clean the retrieved HTML for subsequent parsing. The search view controller instantiates the parser object in its *viewDidLoad* method. During data retrieval it instructs the parser object and acts on commands of the parser as it is its delegate and conforms to the *ParserDelegate* protocol. Figure 5.6 visualizes an overview of the search flow starting with the user entering data in the search view controller's form. All delegation messages the parser sends to its delegate are coloured in red. The search view controller's default user interface is a simple search form, as illustrated in figure 5.5 . It can be extended by pressing "*Advanced Search*" and "*Limit Subject Areas*" subsequently in order to use the full functionality of the implemented Google Scholar API. Figure 5.7 shows a clipping of the advanced search form. It contains almost all fields of Google's form. For comparison see 5.1. During the data retrieval, thus

as soon as the user pressed the "*go*" button, the search view controller displays a view, visualizing the progress of the search. Figure 5.7 shows a screen-shot of latter. When the search flow was completed, the search view controller creates an *SearchResultsViewController* object and pushes it on the navigation stack of the underlying navigation controller to present the search results. We will talk about the *SearchResultsViewController* class in the next section. The *SearchViewController* class has a corresponding XIB file which contains the default user interface elements as well as the view visualizing the process of the search. When a user taps the "*Advanced Search...*" button the user interface is being adjusted and extended by several elements programmatically. You may wonder what the switch "*Try to editing copy of the latest stored query*" does. It will be explained in the last section of the implementation 5.7.

Figure 5.5: SearchViewController's default interface - See (1) in fig. 4.3

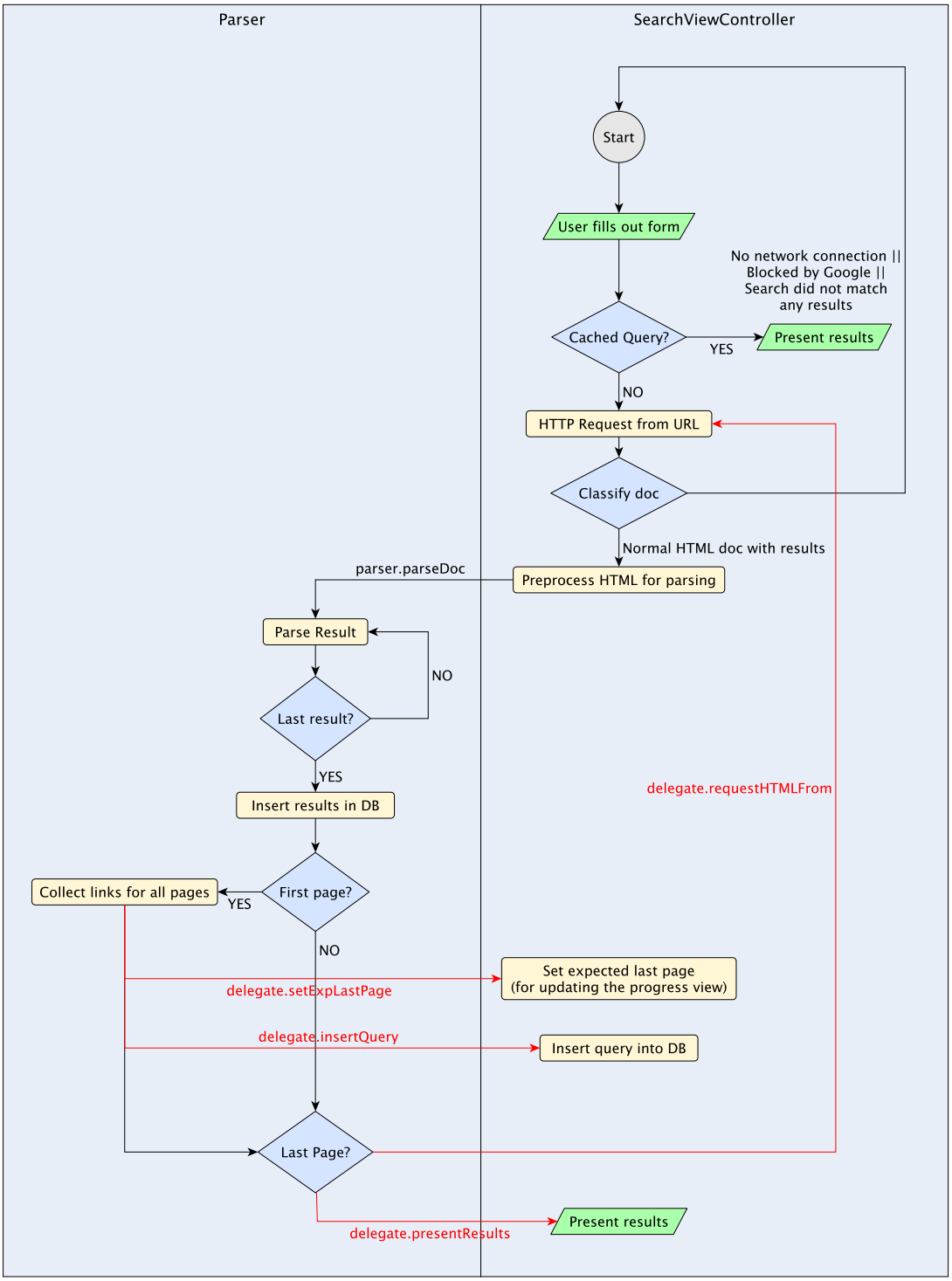

Figure 5.6: Search Flow

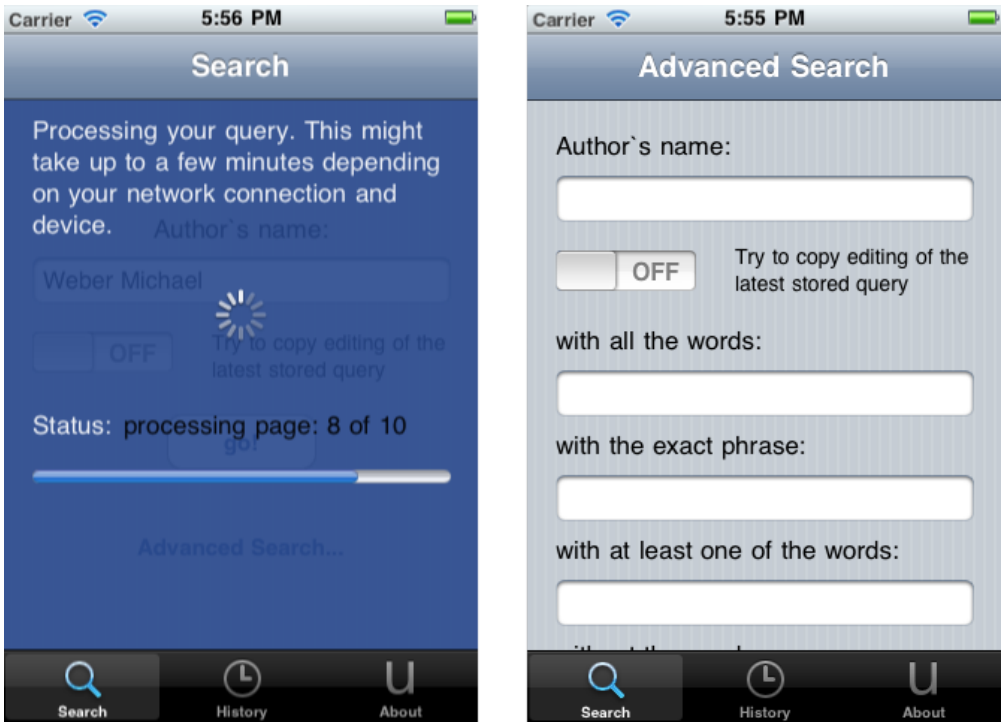

Figure 5.7: SearchViewController's progress view & advanced search form

5.2 Manage search results

5.2.1 Displaying the calculated indexes

The search results view controller is the controller which is responsible for presenting the two scholarly indexes as illustrated by figure 5.8. The class *SearchResultsViewController* inherits from the *ResultsViewController* class which provides most functionality. An common user interface element is the tool bar which can be found in many view controller interfaces. The class *UIViewController* provides the method *setToolBarItems: animated:*. Note that you can only use this method if your controller is managed by a *UINavigationController* object as this is the class providing the tool bar. See "UIViewController Class Reference" [26] for further information. If you do not have a underlying navigation controller, you can add a tool bar either programmatically as a sub view in the view controller's *viewDidLoad* method or simply with Interface Builder in your XIB file. The most interesting feature of the *ResultsViewController* class probably is the functionality to email the results which we will talk about later. The *SearchResultsViewController* class itself is small and al-

most exclusively implements its user interface. Therefore it has a NIB file which contains a *UIScrollView* for adding the user interface elements after the NIB was loaded. Alternatively we could have resigned to use a XIB file here and add the scroll-view programmatically as the XIB exclusively contains the scroll-view.

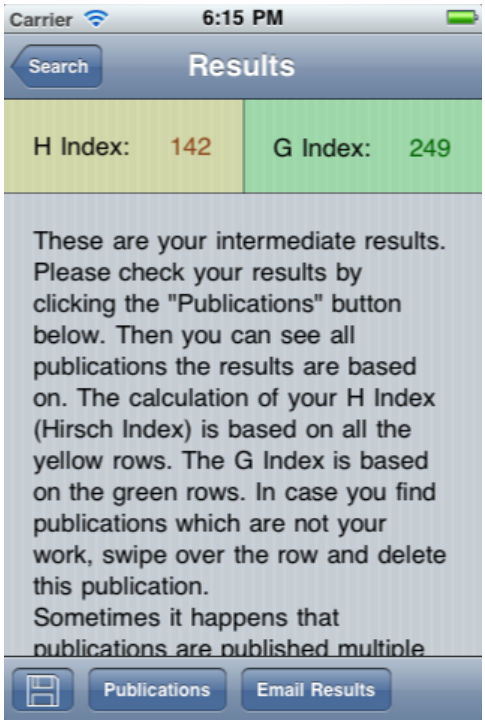

Figure 5.8: SearchResultsViewController's user interface - See (2) in fig. 4.3

5.2.2 Emailing results

The *ResultsViewController* class is the super class of the *SearchResultsViewController* class. In order to provide the functionality to email results, it conforms to two protocols, the *MFMailComposeViewControllerDelegate* protocol which is provided by the *MessageUI* framework and the custom protocol *EmailResultsDelegate* which we defined in the *EmailResultsModalViewController* class. Latter is for the communication of a modal view controller object of the class *EmailResultsModalViewController* and a *ResultsViewController* controller object. The MessageUI framework provides the controller class *MFMailComposeViewController* which allows to implement functionality to send email (as well as text messages) from within your application easily. To do so, you have to import the MessageUI framework and the *MFMailComposeViewController* class in your controller so that you can

create an instance of it (or one of *MFMessageComposeViewController* for texting). The class provides several methods to set up an email. Amongst others attributes, it allows to define an email subject (*setSubject:*), the message-body (*setMessageBody: isHTML:*) and attachments (*addAttachmentData: mimeType: fileName:*). After you have checked whether the device's email client is set up properly, simply by calling the mail compose view controller's *canSendMail* method, you can present the mail compose view controller's email interface, with the same techniques you would present the view of any other controller. Figure 5.9 shows a screenshot of it on the right side. The search results view controller presents the email interface modally. This is a special technique for presenting a controller's view which does not fit into the current navigation flow. But when the user taps the "*Email Results*" button on the tool bar of the *SearchResultsViewController*, the email interface is not being displayed right away. At first the search results view controller presents the interface of an *EmailResultsModalViewController* object, as illustrated by the left screenshot of figure 5.9. It is also presented modally and offers the ability to limit the amount of publications which will be emailed.

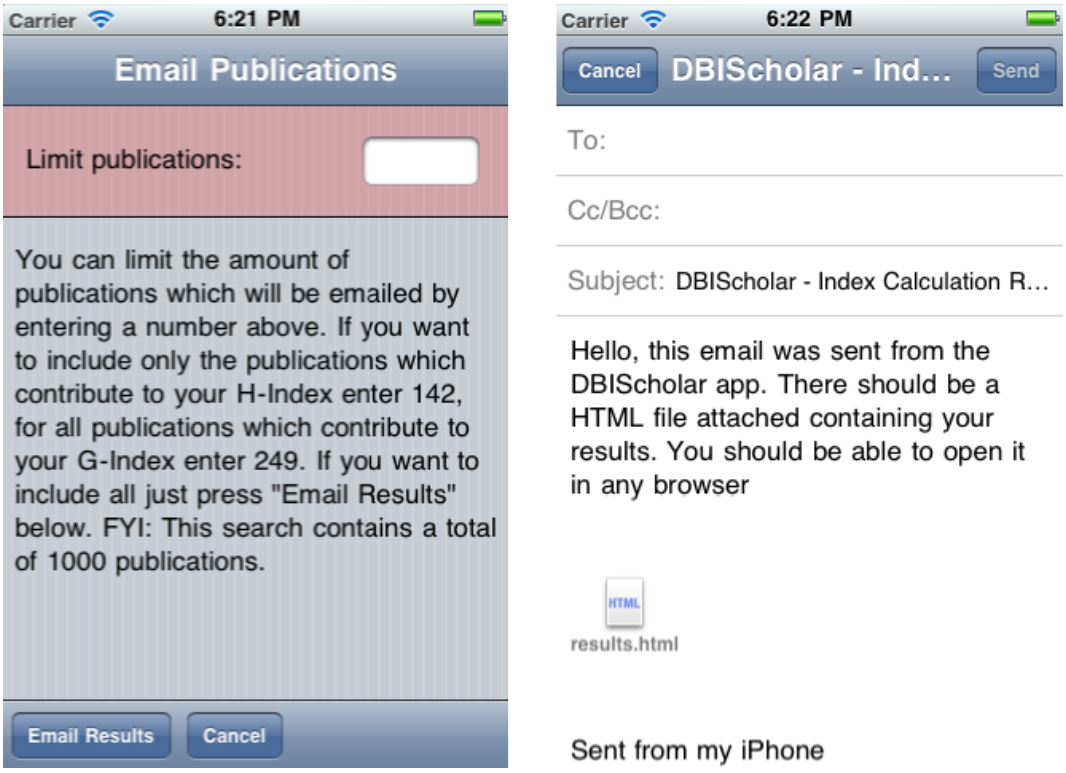

Figure 5.9: **Left:** EmailResultsModalViewController's user interface , **Right:** MFMailComposeViewController's user interface

Modal View Controllers

The so called modal view controllers are not of a special kind of controllers. It sounds like they are controllers which inherit from the "ModalViewController" class or something like that. But that is not the case. The expression modal view controllers refers to presenting any kind of *UIViewController* object in a modal manner. Presenting a modal view controller makes sense when you have to interrupt the current workflow of your application. For example to gather information from the user or temporarily present information to the user. The *UIViewController* class provides the method *presentModalViewController: animated:* to present a view controller modally. Thus any view controller can present another modally. Even modally presented view controllers can present other view controllers modally. Furthermore you can choose between several transition style for presenting a modal view controller by setting the property *@property(nonatomic, assign) UIModalTransitionStyle modalTransitionStyle* of a view controller to the specific *UIModalTransitionStyle*. The transition should differ from the typical navigation transition for a better usability. Usually the controller which presents another modally, is also responsible for dismissing the modally presented view controller by calling the method *dismissModalViewControllerAnimated:*. Again delegation is the default communication concept between the two controllers. (Compare section 2.3.3). The modal view controller defines a delegation protocol to which the presenting view controller has to conform.

Using the example of the *EmailResultsModalViewController* class, the delegation protocol is the *EmailResultsDelegate* protocol to which *ResultsViewControllers* object conforms. The *EmailResultsDelegate* protocol is being defined like described in listing 5.6.

Listing 5.6: EmailResultsDelegate protocol

```
1 @protocol EmailResultsDelegate
2 @required
3 - (void)presentMailVC:(int)withLimit;
4 - (void)cancelEmailResults;
5 @end
```

In the *cancelEmailResults* method, a results view controller simply dismisses the *EmailResultsModalViewController* object with the mentioned method. In the *presentMailVC: withLimit:* method it additionally creates a *MFMailComposeViewController* object to present it subsequently. In this case we first dismiss the current modal view controller (*EmailResultsModalViewController* object) to present another view controller *MFMailComposeViewController* object modally. As an alternative we could make the first modal view controller

present the second view controller modally. This would enable us to dismiss both controllers by dismissing the root modal view controller. For more information see [10].

PDF Support

The requirements define the functionality to email the search results in the form of a PDF-document. Apple's *Quartz 2D* Programming Guide contains the document "PDF Document Creation, Viewing, and Transforming" [12]. Amongst other topics it describes how to create a PDF document using the Quartz 2D drawing engine. Although we quickly managed to make use of the Quartz engine to create PDF documents containing our defined content, we had to find out that the PDF documents could not be displayed on most non-Apple devices. After some searching on the web we found that many others experienced similar problems. Apparently this is due to a bug which does not include the font into the PDF document correctly. Unfortunately we neither could find an official document about this bug nor could we set up a quick solution for the problem ourselves and thus decided to use the HTML format instead. One might argue that we could have used a third party PDF library to realize PDF support, we are aware of this but we deliberately decided against the integration of a third party library as the alternative of using HTML does not have any drawbacks and can be realized with standard Objective-C file handling technologies.

File Management

To create the search results HTML document, we firstly write the HTML code to a *NSMutableString* object. Secondly we convert it to a normal *NSString* object. Before we can write the file to a directory in our sandbox, we have to determine the path to the destination folder. Due to the fact that we create the HTML documents only temporarily to send them via email, the /tmp folder is the right place to store these files. The only available alternative would be the /Documents directory as we will not be able to create files and directories inside the application bundle on the device. To create the path to the /tmp directory we use the method *NSHomeDirectory()* to return the path to the home directory. We then have to append @"tmp/filename.html" to specify the complete path to the file in the /tmp folder. Lastly we write our string to a file by using the method *writeToFile: atomically: encoding: error:* which is available for *NSString* objects.

5.2.3 Displaying the publications of a search

A user can check all the publications, the two indexes are based on by clicking the "*Publications*" button of the search results view controller's tool bar (See earlier 5.8). By doing so,

an instance of the *PublicationsTableViewController* is being pushed on the stack which displays the publications in an *UITableView* object as demonstrated by figure 5.10. In conjunction with the publications table view controller, the table view enables the user to navigate through a list of all publications. The publications are sorted in descending order based on their citation counts and all publications contributing to the two indexes are marked in the corresponding colours yellow and green. As you will see later, the *PublicationsViewController* class also contains functionality for editing the results.

Figure 5.10: PublicationsTableViewController's user interface

The *UITableView* class is a subclass of *UIScrollView* which is part of the UIKit framework and is meant for displaying and editing data in the form of a table. The view's table has a single column and allows vertical scrolling only. The rows are represented by objects of the class *UITableViewCell*, which are special views for actually presenting the data. The *UITableViewCell* class contains some built in sub views such as *textLabel*, *imageView* and methods for displaying your content and handling cell interactions like cell selection and highlighting. For further information about the class see [23]. If the standard properties and methods are not sufficient for your needs, you can subclass *UITableViewCell* to implement your custom functionality. We will provide some more details about the *UITableViewCell*

class in the next section. But first we will introduce the container which class which holds the cell objects, the *UITableView* class. You can instantiate a *UITableViewCell* object with one of two styles, either *UITableViewStylePlain* or *UITableViewStyleGrouped*. The style assigns a certain visual appearance to the way the table view presents its sections and rows. See figure 5.11 for examples. A table view is subdivided into one or more sections which can be identified by an index. Each section contains its own rows of which each can be identified by another index within the section. Table views which are of the style *UITableViewStylePlain* can have an additional index which appears as a list of items (such as the numbers 1, 101, 201, 301 on the right side of the table view in figure 5.11). By clicking on one item, the table view jumps to the corresponding section. In order to display data and to interact with an *UITableView* object, there must be a delegate and a data source object. Typically this is a *UIViewController* object which conforms to the *UITableViewDelegate* protocol and acts as the *UITableViewDataSource* object. Many methods defined by the *UITableViewDelegate* and the *UITableViewDataSource* protocols take *NSIndexPath* objects as parameters. An index path represents a path to a node in a tree of nested arrays. The index path arguments of the two table view protocol methods contain two indexes, the section index and the row index and thus represent the path to a specific row (*UITableViewCell*). For further information about *UITableViewDataSource* see [15]. For more information about the *UITableViewDelegate* protocol check out [16]. The instance of *UITableView* which belongs to our *PublicationsTableViewController* object can contain up to 1000 rows. Therefore one might think it has 1000 *UITableViewCell* objects to display all rows. Due to the fact that this would be very ineffective and probably would not work with the restricted resources of the iPhone, the *UITableView* object caches its *UITableViewCell* objects. It has only as many objects as are actually visible on the screen and reuses these objects to display the entire data set. The responsible method for reusing the cell objects is the table view's *dequeueReusableCellWithIdentifier:* method. For detailed information check out [22].

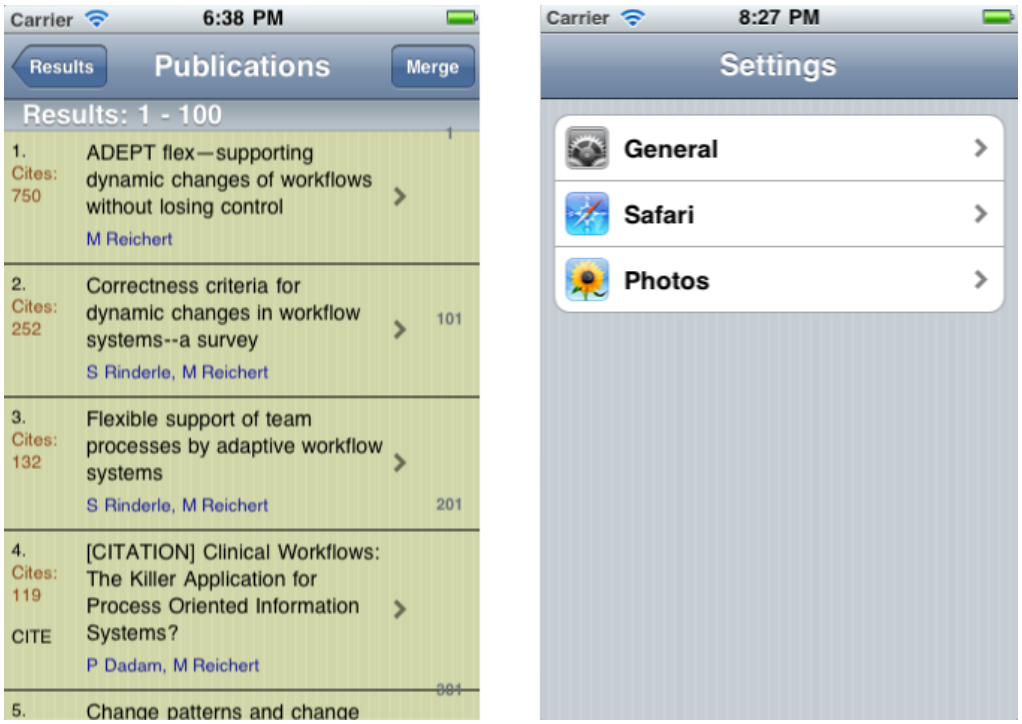

Figure 5.11: Left: UITableViewStylePlain - Right: UITableViewStyleGrouped

UITableViewCell

As mentioned before the *UITableViewCell* class is a special *UIView* class for presenting and managing data in the table of an *UITableView* object. The class provides multiple view properties (*textLabel, detailedTextLabel, imageView, contentView, backgroundView,...*) which let you build the visual appearance of the cell. When you create a cell you can choose to implement a predefined cell style similar to when you create a table view. A cell style positions the sub views of the cell in a certain manner and can be seen as a template for presenting data in a certain layout. In many situations the predefined styles and views will be sufficient to present your data. If they are not, you have two options to extend your cells. You can simply add your custom sub views to the content view of a cell. The *CompareTableViewController* class in our application implements its cells this way. If you decide to add custom views to the cell's content view, you have to think about how you can reuse your sub views when you make use of a reuse identifier. And you always should reuse your cells. You might have to adjust the heights of the cell and the layout of the sub views for each cell when your content is of dynamic size. The second option is to subclass *UITableViewCell*. In our application we did this to present the rows of the publications table view

controller by our custom class *Cell*. Each object of the class *Cell* presents one publication in a *PublicationsTableViewController* object. Therefore it contains six labels (*cites, title, authors, ...*) which can be set through corresponding methods (*setTitle:, setCites:, setAuthors:, ...*). Another method allows to set the colours in which the different elements are being displayed. Furthermore our cell automatically calculates the heights of its sub views. Thus we can determine the necessary cell height through the properties *rightSectionCurrentHeight* and *leftSectionCurrentHeight* and adjust the cells in the delegate's *tableView: heightForRowAtIndexPath:* method.

The *UITableView* class not only handles the data visualization but also enables the user to add and edit data presented by the cells. This is why the *UITableViewCell* class has many "editing" properties, which define the behaviour of the cell when it is in editable state. A cell enters the editable state when the table view calls the *setEditing: animated:* method on the cell with editing set to *YES*. Therefore the editing style can be set through the *editingStyle* property. The editing style defines special controls (*deletion controls, insertion controls, reordering controls*) that the table view's delegate has assigned to each row in the *tableView: editingStyleForRowAtIndexPath:* method. As the names of the controls indicate, they allow to trigger methods to delete, insert and reorder cells. For detailed information check out the chapters "Inserting and Deleting Rows and Sections" and "Managing the Reordering of Rows" in Apple's "UITableView Programming Guide" [14]. In our application we neither make use of the editing style (as we exclusively need to delete publications) nor any of the deletion-, insertion- and reordering controls. We will talk about how the *PublicationsTableViewController* class implements its functionality to delete publications in the next section.

5.2.4 Editing results

The user must be able to delete publications which are part of the results but do not belong to the requested author or more generally speaking to the requested search. Deleting a publication results in an automatic recalculation of the two scholar indexes. For inserting, editing and deleting rows the *UITableViewDataSource* protocol provides the method: *tableView: commitEditingStyle: forRowAtIndexPath:*. When this method is implemented by the data source, the table view automatically provides a swipe-to-delete feature which allows to trigger this method by swiping over the cell and pressing the appearing "*Delete*" button. See figure 5.12. Doing so calls the methods with the *UITableViewCellEditingStyle* argument set to *UITableViewCellEditingStyleDelete*. In the case of the *PublicationsTableViewController* class, we exclusively make use of this method to delete publications and thus do not need to differentiate between multiple *UITableViewCellEditingStyles*. To actually delete a publication we call a database method inside this function, which marks the result as deleted

and reloads the table view to reorder the rows. Deleted results appear in an additional "*Rejects*" section at the bottom of the table view and are coloured in red. See the screenshot on the right side of figure 5.12. Deleting a publication in the rejects section in the same manner, will restore the publication.

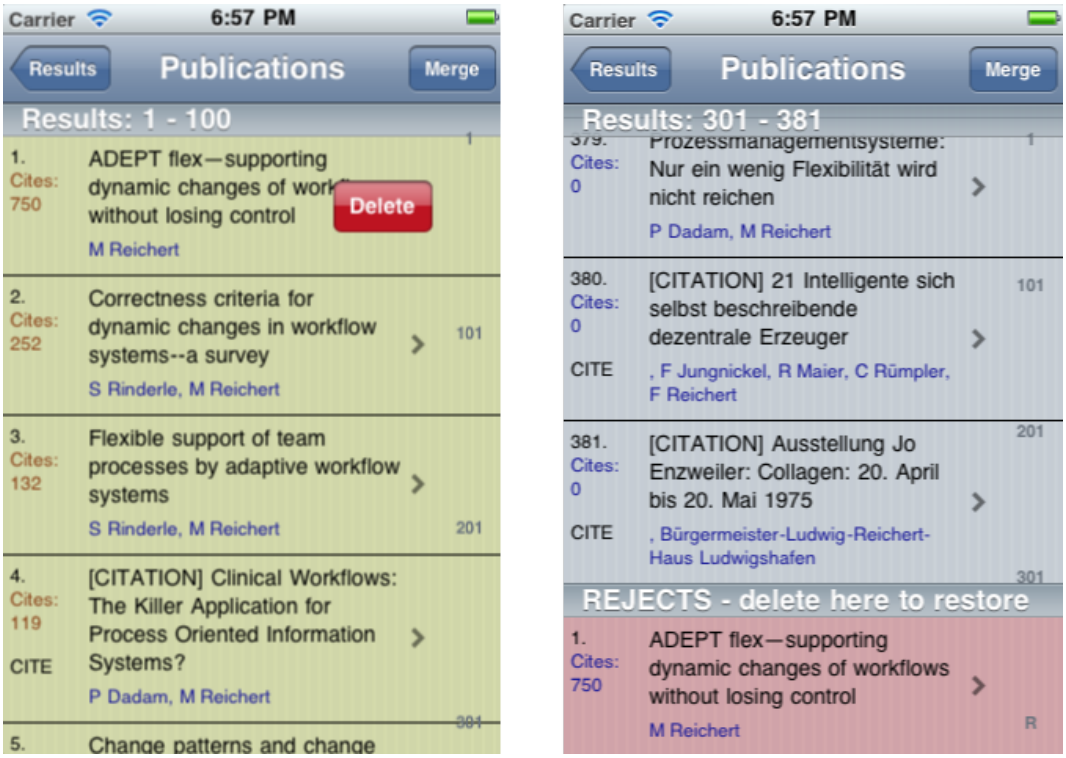

Figure 5.12: PublicationsTableViewController's: Deleting publications & Rejects section

5.2.5 Displaying a publication

When you tap one of the rows of the publications table view controller, a *PaperViewController* object is being created and presents its user interface. Figure 5.13 shows a corresponding user interface. It contains further information about a publication. Additionally it provides up to four functions. If Google could find a link to the actual publication you can use the very left button of the tool bar to browse the URL. The second feature is the "*Graph*" feature. By clicking the corresponding button, a graph which visualizes the evolution of the citations over time, will be calculated and displayed for the respective publication. We will talk about it later as this is one of the features which emerged during the development. In case the publication is merged with others, the third button allows to navigate to a controller

with which you can dissolve the merges. It will be explained in section 5.3.3. Last but not least, if Google found a link to the publication, you can email this link simply by clicking the respective button in the tool bar.

Figure 5.13: PaperViewController's user interface

The *PaperViewController* class inherits from our *BasicViewController* class. Its corresponding XIB file contains a *UIScrollView* to which we dynamically add several *UILables* in the controller's *viewDidLoad* method to display the publication details. Dynamically means we first calculate the size of the text, add the label and adjust the height of the scroll-view if necessary. Furthermore we have added a tool bar with four buttons to trigger the functions mentioned above. By now you know that most of the application is made up of *UIViewControllers* objects and their corresponding *UIView* objects which contain the user interface the controllers present. The *PaperViewController* class is another example of a typical view controller. In order to send email, the controller conforms to the *MFMailComposeViewControllerDelegate* protocol and imports the *MessageUI* header and the *MFMailComposeViewController* header file. We already talked about emailing within an application in section 5.2.2. The methodology here pretty much is the same. For further information you can check out Apple's MessageUI Framework Reference [9].

In some situations we would like to display a web document in our application. In case the Google search engine could retrieve a link to the web resource where we might be able to read the actual publication, this is one of those situations. In order to embed web content within your application the UIKit framework provides the class *UIWebView*. After you have created a *UIWebView* object you can use the *loadRequest:* method to connect to a given URL. The *UIWebView* class actually allows to implement standard browser functionality within your application. The methods *stopLoading*, *reload*, *goBack* and *goForward* allow to stop, reload the requests and navigate through the web. The class defines the delegation protocol *UIWebViewDelegate*. It enables the delegate to react on the events *webView: shouldStartLoadWithRequest: navigationType:, webViewDidStartLoad:, webViewDidFinishLoad:, webView:didFailLoadWithError:*. But there is even more. A *UIWebView* object can automatically detect phone numbers, http links, email address defined by it's *dataDetectorTypes* property in order to provide respective functionality. For further information check out the UIWebView Class Reference [17].

As I said, we would like to display web content in several situations. Therefore we built the *WebViewController* class. It is a usual view controller which serves as the delegate for its own *UIWebView* object. The corresponding XIB file contains the *UIWebView* object, a tool bar to support the browser functionality and a *UIActivityIndicatorView* object to indicate when the controller is busy loading. Figure 5.14 shows an example of the web view controller's user interface.

To actually see a *WebViewController* object in action click on the very left button of the paper view controller's tool bar. The *WebViewController* object will load the publication's respective web document. Not all publications have a respective URL and for others you might need an account in order to read them. However, many of the publications can be read directly within the application.

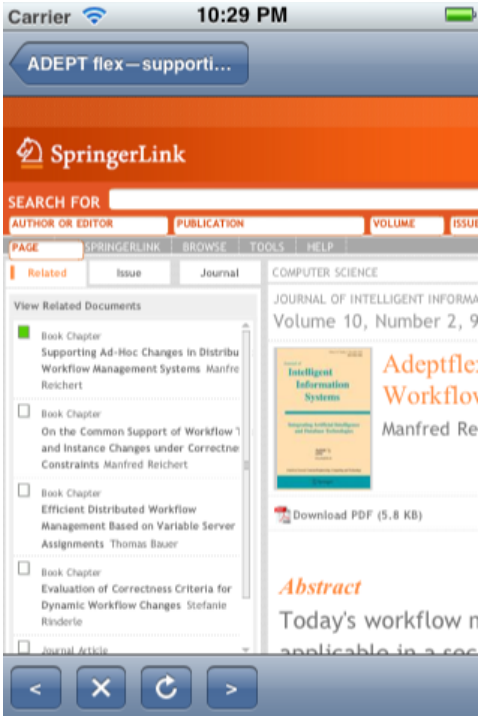

Figure 5.14: Example of a WebViewController user interface

5.2.6 Data persistence on the iPhone

After submitting a specific search, we want to be able to maintain our results. This means we have to store the data we retrieved during parsing on our device at some point. The most common ways of saving data to the iPhone are Property Lists (Plists), Object Archiving, Core Data and SQLite. Depending on what task you want to accomplish, it is important to know about a basic security concept of iOS: Each application only can access its reserved portion of the file system, the so called sandbox. Any other parts of the file system cannot be accessed. In particular all of the mentioned concepts therefore usually store data in the */Documents* folder inside the application's sandbox. An alternative for temporary data is the /tmp folder. The following sections will give a short overview of the concepts mentioned above.

Property Lists

Property lists provide a primitive way to store and create serialized objects. Although any object can be serialized only a limited number of types can be stored using a property list.

Some types are for primitive values and others are containers which can hold those values: *NSString, NSData, NSDate, NSNumber, NSArray, NSDictionary*. Property Lists are convenient for persisting small amounts of primitive data. For more complex data hierarchies, custom objects and large amounts of data (more than a few hundred kilobytes), property lists will quickly become insufficient.

Object Archiving

Object Archiving can be seen as the next higher level from Property Lists. As the name already indicates, object archiving provides a convenient way for storing objects of an arbitrary class in an archive. It allows you not only to archive a single object but complex webs of interrelated objects. These webs are called object graphs. In an archive arbitrarily complex webs of interrelated objects can be stored at which the identity of every object in the graph and all its relationships to other objects are being preserved. Objects therefore have to conform to the *NSCoding* protocol. An example of object archives are the NIB files created by Interface Builder to archive user interface objects (Compare chapter 2.4). For further information about object archiving check out the "Archives and Serializations Programming Guide" [21].

SQLite

iOS ships with SQLite, a lightweight reliable and powerful relational database which directly reads and writes to ordinary disk files. One big advantages of SQLite library is that it can handle relative large amounts of data (up to one gigabyte and more) efficiently. We can define data structures independently from corresponding model objects and perform basic SQL queries on these. Another advantage of SQLite is that the database format is cross platform.

Core Data

Core Data is a fully featured persistence framework which helps you manage models in the sense of the model-view-controller design pattern. It implements intelligent caching mechanisms and allows you to keep a subset of your model objects in memory at any time which can be crucial to memory management. Model objects managed by Core Data can be seen as a representation of a record in a database table. Amongst other things, Core Data can give automatic support for common tasks like save, restore undo and redo and for maintaining reciprocal relationships between objects. The Core Data framework is based

on the built in SQLite library.

Table 5.11 gives an overview of the different technologies.

Property Lists	- Small amounts of data (less than a few hundred KB) - No custom objects - I.E. to save user settings
Object Archiving	- Any objects conforming to NSCoding protocol - Stores the objects not just values - Complex object graphs can be archived
SQLite	- Relational Database - C API - Can handle large amounts of data (up to GB)
Core Data	- Framework with high level abstraction - No Database queries necessary - Especially useful to manage model objects

Table 5.11: The different possibilities of data persistence

5.2.7 Caching and Saving

In our application we have to differentiate between caching and saving at first. Obviously it makes sense to have the ability to save results, simply for later use. Caching on the other hand is necessary to protect the user from resubmitting the same queries. This is important because Google only allows to request a certain amount of data in a certain amount of time based on an IP address. By using a caching mechanism we try to prevent the user from quickly getting blocked by Google. Secondly we must have the ability to manage large amounts of data as a single search result with all the contained publications may allocate about 800KB on the device and the user is in charge of which search results he wants to keep. It is a design decision whether we want to map our data to corresponding models (in terms of the MVC pattern) or not. Because of the straightforward structure of the result data, we can stick to basic data structures instead of mapping our data to custom models.

For full flexibility we eventually decided to use SQLite over Core Data and all other options. SQLite provides basic date-time functionality and thus not only serves for saving but also for caching. As we know, Objective C is a superset of ANSI C. Therefore it is possible to mix C and Objective C within your project. The SQLite API for instance is a C API. It is not included in XCode projects by default. You can do so by right clicking your *Frameworks* folder in XCode, *Add –> Existing Frameworks...* and then choose *libsqlite3.dylib*. To actually use it in one of your Objective C classes you have to import the *sqlite.h*. We will not cover the SQLite API in this diploma thesis. The documentation can be found at `http://www.sqlite.org/`. In our application all database methods are bundled into the custom class *SqliteController* which handles database interaction through the SQLite API. The two controller of the types *SearchViewController* and *HistoryTableViewController* are the only controllers which actually instantiate this class. Compare with the "Architecture" chapter 4. These two controllers then pass on pointers of their *SqliteController* objects to other controllers. It simply would be inefficient to create a sqlite controller for each controller which needs to interact with the database. In fact it would be even more efficient to instantiate the *SqliteController* class only once in the *AppDelegate* and then pass on the pointer to the database controller from there. There are probably different strategies on how to achieve this but it definitely is something to consider when your application implements some kind of database controller.

Caching Mechanism

To implement our caching mechanism we make use of the date-time functionality provided by SQLite. Each time when a controller of the class *SqliteController* is being instantiated by one of our two root controllers, the sqlite controller checks for cached data which is older than twelve hours. In case there are older queries and respective publications, they are being deleted. To distinguish between cached and stored data, we simply set the *keep* integer field of the query entries to the respective boolean values. SQLite does not provide a separate boolean storage class (Compare queries table structure 4.9).

5.2.8 Manage stored results

The *HistoryTableViewController* object of our application is the controller which displays search results which were being stored or are still in the cache. Like the *PublicationsTableViewController* objects, the *HistoryTableViewController* object displays its data in a table view. To differentiate between the search results, each cell displays the name of the author and the date of when the search was being submitted. To distinguish between cached and stored results the history table view controller displays either a transparent floppy disk or a coloured floppy disk as illustrated by figure 5.15.

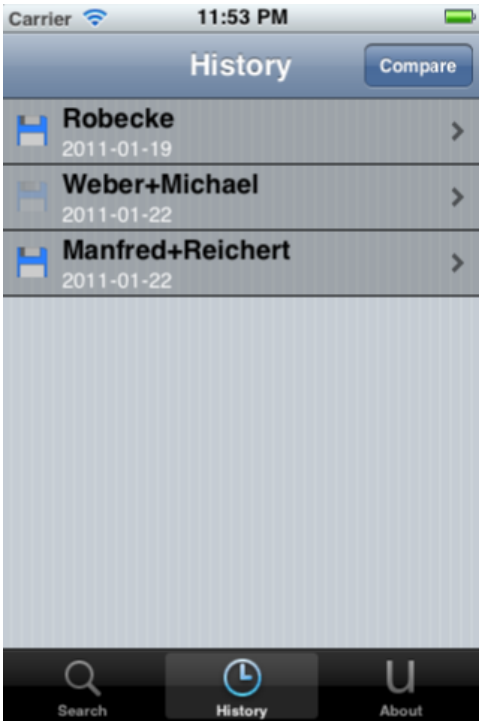

Figure 5.15: HistoryTableViewController's user interface

Unlike our *PublicationsTableViewController* class, the *HistoryTableViewController* class does not inherit from any of our custom controller classes. This has the advantage that we can make it inherit from the *UITableViewController* class provided by the UIKit framework. The *UITableViewController* class is a template for a special controller which presents data in its predefined *UITableView*. The controller object thereby automatically serves as the data source and as the delegate object of its table view. This mainly saves writing code by automating some procedures we had to take care of ourselves when we implemented the *PublicationsTableViewController* class. For instance we do not need to create a *UITableView* as the controller's view already is of this type. Furthermore the data source and delegate get hooked up to the controller object automatically when we create it. Generally speaking, when you want to display data in a table view, you should subclass *UITableViewController*. If this is not possible you can still implement a table view, data source and delegate independently.

Another difference between the *PublicationsTableViewController* class and the *HistoryTableViewController* class are the cell objects which they use to display their data. While the *PublicationsTableViewController* class makes use of our custom class *Cell*, the *His-*

toryTableViewController class uses a standard version of the *UITableViewCell* class. To create one of the standard versions of the class *UITableViewCell*, you initialize it with a predefined style using the method *initWithStyle: reuseIdentifier:*. In this case we have used the style *UITableViewCellStyleSubtitle*. It provides a left-aligned label across the top and a left-aligned label below. In order to display the floppy disk icon, we simply add the icon to the cell's *imageView*. Figure 5.16 illustrates the layout of the three view properties. (Also compare section 5.2.3)

Figure 5.16: UITableViewCellStyleSubtitle - UITableViewCell out of the box

The *HistoryTableViewController* object, just like the *SearchViewController* object is being presented by a *UINavigationController* object (See chapter 4 for details). When the user taps on of the rows, an instance of *HistoricResultsViewController* is being created and being pushed on the stack of the underlying navigation controller to present a more detailed view of the search query. The *HistoricResultsViewController* class is almost identical to the *SearchResultsViewController* class. The only difference to the *SearchResultsViewController* class is the user interface. Instead of an explanation of what to do next, the historic results view controller displays the details of the search query. Both controllers inherit their functionality from their super class *ResultsViewController* which was being explained in section 5.2.1 and the following. Therefore the functionalities of showing publications and emailing results are exactly the same.

You might have noticed the "*Compare*" button in the history table view controller's user interface. We will talk about it later.

5.3 Merging publications

Occasionally it happens that the very same publication is listed multiple times in the very same search result. This might happen when the same publication was published under different titles. In such a case the Google Scholar search engine might not be able to determine that these titles refer to the same publication and thus index two publications instead of one. Furthermore it even can happen that Google finds multiple publications with the

exact same title for the same author. This is confusing because the same author would probably not write two different publications and entitle them identically. Obviously these issues can distort the results. Unlike the Google Scholar search engine, a person might notice that the very same publication is listed twice. Maybe the user can even imagine why this happens. However, if the very same publication appears multiple times, one may use the "*Merge*" button in the navigation bar of the publications view controller's user interface to merge multiple publications. (See figure 5.10) After the user tapped the "*Merge*" button, the table view swaps into what we call "*Merge Mode*". In "*Merge Mode*" one can select the publications which should be merged. The interface then displays a tool bar with the "*Merge Selected Publications*" button to merge the selected publications. Figure 5.17 shows a screenshot of the publications view controller's interface in "*Merge Mode*" on the left side. The feature allows to merge multiple publications at once and even nested merging is possible. Back in normal mode, merged publications are being marked with a red "*Merge* #amount of other publications merged with this publication" as illustrated by figure 5.17 on the right side. The publication with the most citations will be kept as the representative publication. If there are multiple, an arbitrary publication will serve as the representative one. All others will disappear from the table view. Of course it is possible to dissolve merges. Therefore the merged publication has to be tapped. In the appearing view, the bottom bar contains a "*Dissolve*" button. Tapping it shows all publications which are currently merged with this publication, again in a table view. To delete a merge you simply delete the publication here the same way you would delete publications in the publications view controller's user interface with the swipe and delete feature (Compare section 5.2.4).

Figure 5.17: Merge Mode & Publication with one merged publication

5.3.1 Merge Mode

The implementation of the "*Merge Mode*" is simple. Basically we toggle a flag which indicates whether merge mode is set or not. Depending on the flag we change the appearance of the cells by using different *UITableViewAccessory* types and reloading the table view. Usually a cell's accessory type is set to *UITableViewCellAccessoryDisclosureIndicator* (1) which indicates that the cell can be tapped to display the next level in the navigation flow. When the "*Merge*" button gets tapped, we toggle the flag and reload the table view. In the data source's method which returns the cells (*tableView: cellForRowAtIndexPath:*), we toggle the accessory type accordingly. In "*Merge Mode*" we use the accessory type *UITableViewCellAccessoryNone* (2) so that it is clear that the usual function of the cell is disabled temporarily. For the selection and deselection of the cells, we makes use of the same principle. In the delegate's method, which gets called when the user taps a cell (*tableView: didSelectRowAtIndexPath:*), we toggle between the accessory types *UITableViewCellAccessoryNone* and *UITableViewCellAccessoryCheckmark* (3). Latter indicates that a cell was being selected by displaying a check mark. The deselection of the cell in

turn, causes the cell to switch back and display the accessory type *UITableViewCellAccessoryNone*. Figure 5.18 shows the three accessory types.

Figure 5.18: UITableViewCell Accessory Types

5.3.2 Merging

The purpose of merging is to merge the citation counts of multiple publications into a single representative publication count. Therefore we define the publication with the highest citation count as the "*Master*". If there are multiple we select an arbitrary one to become the "*Master*". For all other publications which were selected for being merged with the master publication, we change the database field *merged_with* to the master's ID and add the citation counts to the master's *cites* field. Additionally we increment the master's database field *merge_count* for each merged publication by one so that we know how many publications are being merged with the master.

5.3.3 Dissolving

In order to dissolve the merges of a publication, you first have to navigate to the next higher level in the navigation flow. Again, this is implemented with the usual methodology to push the next controller onto the navigation stack and triggered by tapping a row of the publications view controller's table view. The subsequently presented user interface belongs to a controller of the type *PaperViewController* which we introduced in section 5.2.5. By pressing the "*Dissolve*" button of the paper view controller's tool bar, an instance of the *DissolveViewController* is being created and pushed on the stack. The *DissolveViewController*

class inherits from our *BasicTableViewController* class, which simply is an implementation of the class *UITableViewController* holding a reference to the database controller and an ID to the database entry it manages. Thus the *DissolveViewController* class basically is another standard *UITableViewController* implementation which shows the merged publications as illustrated by figure 5.19. The user can dissolve a merge by deleting a publication the same way publications can be deleted in the publications table view controller (Compare section 5.2.4). The implementation of this functionality thus also is the same. The only difference is the database method it triggers to dissolve a merge.

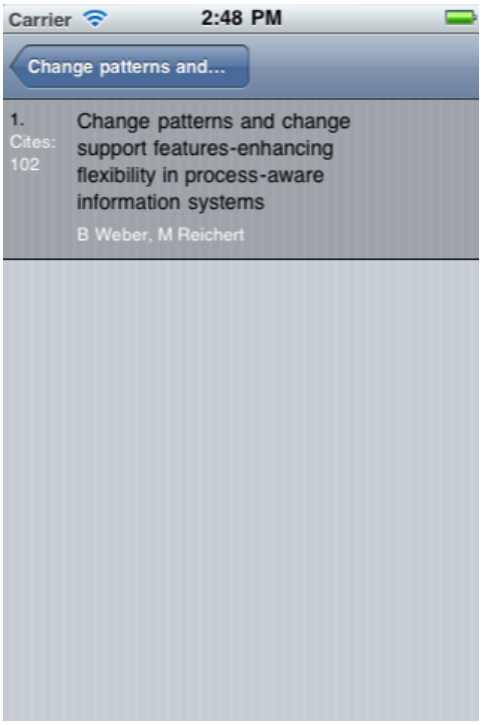

Figure 5.19: DissolveViewController contains the merged publications of a publication

5.4 Graph Feature

The idea of the graph feature is, that it might be interesting to know about the evolution of the citation counts of a particular publication. For most publications, which are contained in the result of a search, Google provides a link to display all the publications which cite this publication (See the "Cited by #" link in section $<!--6-->$ of fig. 5.4). Furthermore it is possible to limit all publications, which cited the publication of interest, to those which were

published in a particular year (Compare section (1) of figure 5.2). Assuming the publication years are known, this means that we can assign the amount of publications which cited the publication of interest to a specific year. Given these data, we can draw a graph visualizing the evolution of the citation counts over time. Fortunately most publication years are given and we can draw more or less meaningful graphs in most cases. To provide a measure of the actual accuracy of the graph, we quote the percentage of how many publications could be assigned to it.

When you tap the "*Graph*" button of the paper view controller's tool bar, a *PaperGraphViewController* object will be instantiated and start with the collection of the publication counts for each year. The *PaperGraphViewController* object does not make use of the *Parser* class. It requests HTML documents jut like the *SearchViewController* class but retrieves the data of interest using regular expressions. Therefore we use the *RegexKitLite* framework [2]. It extends the standard *NSString* class, provided by the foundation framework, by advanced regular expression functionality. To use it in your project download and include the classes and simply add *-licucore* to your *Other linker Flags* in the *Build* tab of your info.plist file. Similar to the *SearchViewController* object the *PaperGraphViewController* object initially presents a view visualizing the progress of the data collection. As soon as the data is available a bar graph is being drawn and presented to the user as illustrated by figure 5.20. The user then has the option to switch to a full screen overview mode (See figure 5.21), email an image of the graph (See figure 5.23), display the amount of publications which could be allocated in the graph (See figure 5.22) and navigate back to the paper view controller of course. The four options are being provided by four buttons on top of the initially presented graph view.

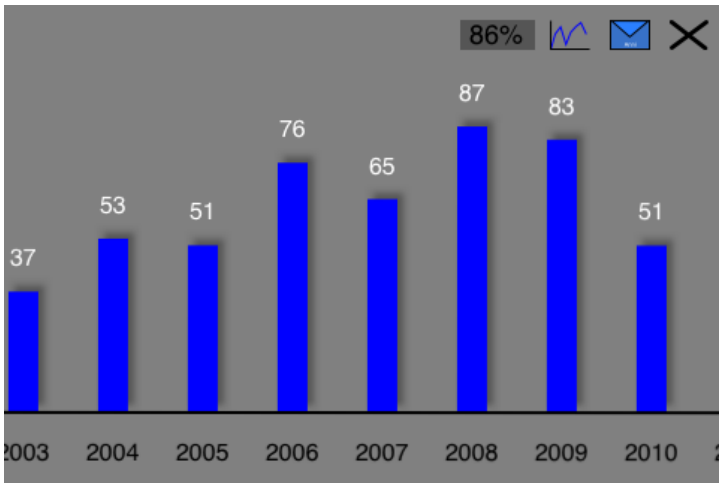

Figure 5.20: PaperGraphViewController's default graph view

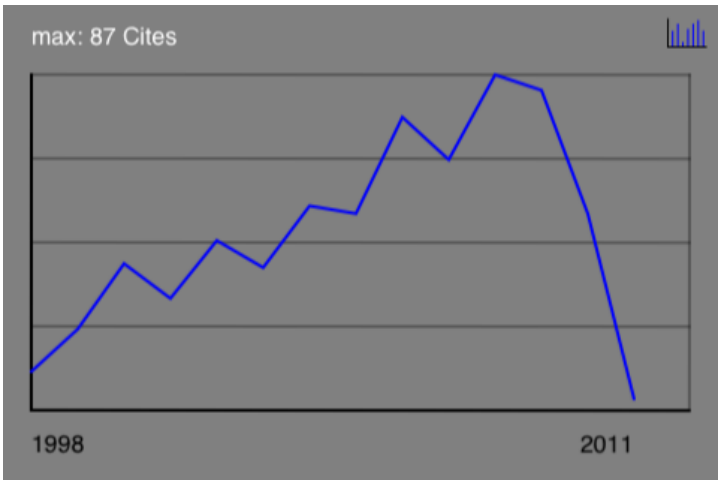

Figure 5.21: PaperGraphViewController: Graph overview

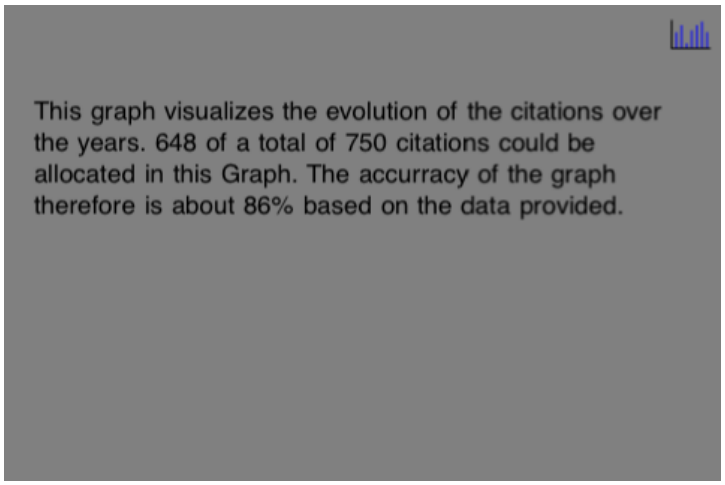

Figure 5.22: PaperGraphViewController: Citation allocation explained

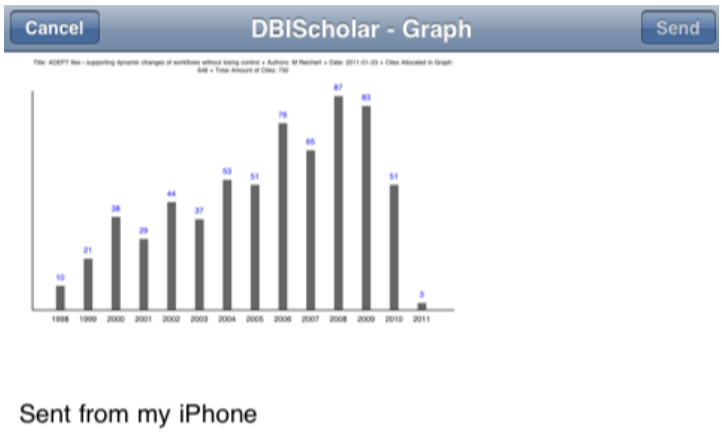

Figure 5.23: Email an image of the graph

5.4.1 Displaying large Images

Before we could start with implementing the actual drawing of the bar graph, we had to think about how to accomplish two important aspects. On the one hand, we had to find a way of presenting graphs of arbitrary size on a screen with a resolution of 320x480 pixels (IPhone and iPodTouch of the 2nd and 3rd generation). Thereby we wanted the graph to show the evolution of the citation counts but also the exact values. On the other hand it is not possible to display images of arbitrary size on the iPhone and the iPodTouch as memory is limited. Both aspects are based on similar constraints. The problem of displaying large images on the iPhone is not novel at all. The basic approach is to chop the image into multiple smaller tiles and use a *UIScrollView* object to scroll around and partially display the image. By using a *UIScrollView* object, zooming in and out also can be achieved. Indeed that is what we first tried. We initially displayed an overview of the graph and enabled to zoom in, to make the individual values such as citation counts and years readable. Quickly we figured that one has to do a lot of scrolling and zooming to read the individual values as well as for getting back to display the overview. In order to achieve both, displaying an overview as well as the details in the graph, our final approach was to use two different perspectives. This not only solved the illustration problem but also lead to a basic approach of how to implement the detailed perspective in which the actual values of the citation counts are being displayed. More on this in section 5.4.3.

5.4.2 Quartz 2D

We already mentioned the *Quartz 2D* drawing engine in the context of creating PDF documents in section 5.2.2. This is no coincidence at all. In order to draw with Quartz a graphics context is being required. This is an opaque data type which encapsulates the information Quartz uses to draw to an output device. It can be thought of as some kind of drawing destination which not only contains the drawing parameters but also device specific characteristics. The device can be a window, an image or a PDF file for instance. This has the advantage that you can draw the same sequences of Quartz drawing routines to different devices simply by using different graphic contexts. As you know, we decided to create HTML files instead of PDF documents eventually. But when we initially created PDFs, we used a PDF graphics context to create our documents. Likewise we used a window graphics context, to draw our graph on the screen, and a bitmap graphics context to create a PNG image in order to email it.

5.4.3 Graph drawing strategy

Like we said, it is not possible to display images of arbitrary size. Also this does not make much sense as the screen resolution is 320 x 480 pixels (Device of third generation) and we can not show images of larger size anyway. For our detailed bar graph perspective we came up with the idea that we could use the device in landscape mode and adjust the citation values to the screen height of 320 pixels so that horizontal scrolling would be sufficient to show graphs which are broader than the screen. This results in an easy to use interface and omits the necessity of zooming and scrolling for citation counts and years as we can add the values in readable size to the graph. To overcome the problem of drawing large graphs, we use three tiles of 240 x 320 pixels and a *UIScrollView* object to draw the detailed graphs. The tiles hereby change their positions in the scroll-view based on the scroll-view's offset from its origin and draw their content based on their position. Figure 5.24 illustrates this approach. In the application an instance of *PaperGraphViewController* holds the scroll-view with the three tiles which draw the entire bar graph. The tiles are of the class *GraphTile*. It basically is a *UIView* which draws its content based on its frame, more precisely on the position of its frame in the scroll-view. To rearrange the tiles we overwrite the *setFrame:* method. The paper graph view controller conforms to the scroll-view's *UIScrollViewDelegate* protocol and responds on scrolling with rearranging the tiles respectively by using the mentioned *setFrame:* method.

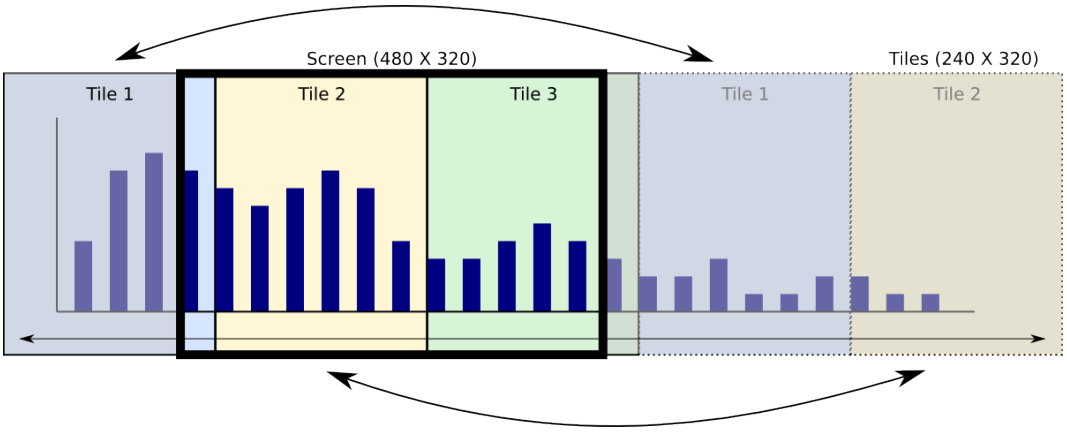

Figure 5.24: Relocating tiles which draw their content based on their position

Once a graphics context is available, the parameters for painting can be set and the drawing routines like *CGContextAddLineToPoint* being defined. To draw something on the screen, you have to make an *UIView* object implement the method *drawRect:*, in which you can then perform the actual drawing. The *UIView* object automatically creates a graphic context for the current drawing environment which you can obtain in the *drawRect:* method by calling *UIGraphicsGetCurrentContext*.

5.4.4 Creating an image file

To create an image, you have to create a *CGBitmapContextCreate* first. Therefore you need to supply a pointer to the memory where you want the drawing to be rendered. But first you have to calculate the size of the memory block which is required for your image and allocate it using the *malloc* method. In our case we have used 8 bit for each component R, G, B and Alpha in the RGB color space, which results in 32 bit per pixel. The necessary memory block therefore must be $imageWidth * imageHeight * 4$ bytes large. There are plenty of drawing routines and transformations functions available. You can draw and combine paths, gradients, shadows, patterns, layers and transformations. It would go beyond the scope of this diploma thesis to explain them here. Also it very much depends on what you want to accomplish. To find out which routines and transformations you need for what you want to draw, you should check out Apple's Quartz 2D Programming Guide [13].

5.4.5 Email the image

The process of emailing the graph image very much is the same as emailing the HTML results as discussed in section 5.2.2. After we drew the image, we can save it to the /tmp folder and then attach it to our email. Hence the *PaperGraphViewController* class not only conforms to the *UIScrollViewDelegate* protocol but also the *MFMailComposeViewControllerDelegate* protocol in order to implement the email functionality like described before.

5.5 Comparison Feature

The comparison feature, as the name suggests, is a feature for the comparison of two search results. In the navigation bar of the history table view controller there is the "*Compare*" button (Left screenshot of figure 5.25). When you tap it, a tool bar with the button "*Compare Checked Publications*" will appear and you can select two of the search results as illustrated by the right screenshot of figure 5.25. When you tap the button in the tool bar both scholarly indexes for both scholars are being calculated and presented by the interface of a *ComparisonResultsViewController* object which is illustrated by the left screenshot of figure 5.26. The presented view allows to compare the two indexes and the search details. For a better distinction between the tow search results, the interface displays all results related to the first scholar in brown and yellow and all results related to the second scholar in green. The more interesting part is the *CompareTableViewController* object which displays the publications of both searches in the same table. In order to differentiate between which publication belongs to which search, the compare table view controller's table view also uses brown and yellow to display the publications of the first search and two shades of green for the publications of the second search. Unfortunately we cannot be sure if it is possible to identify publications distinctly. Initially we thought that publications do have a unique title. But we had to find out that this is not always the case. For distinct identifications we would have to know whether it is possible to identify results distinctly or whether it can happen that there are two results with the exact same values and thus are indistinguishable. Fortunately, all this only effects us slightly. The fact that there can be publications with the same titles does not have any impact on the calculations of the two indexes at all. If Google accidentally provides two publications instead of one, it is up to the user to clean the data by deleting one (Compare 5.2.4) or merging both (Compare 5.3). Therefore we assume that the publication titles within the results of one search are unique and identify the publications for the comparison based on their titles. When we can find one publication in both search results, we display the corresponding row in the neutral blue colour. Moreover we split the left section of our rows in top and bottom. In the top half, we show citation- and merge count of the first scholar's publication. In the bottom half we display the respective details of the second scholar's publication whereas the citation count is being

displayed in relation to the first publications count. So you might see something like $+3$, -1 or $+-0$ if the citation count is the same for both publications. For a visual illustration see the right screenshot of figure 5.26. The comparison feature can be helpful to find out why an author has better indexes than another. It also allows to compare the results and publications of the same author at different points in time. In the advanced search form the user can limit publication years. On the other hand one could perform a search today for a comparison with the exact same search in two month from now. However, the results of the feature are not that easy to understand and you should remember that the colours green and brown stand for two different searches, instead of two different indexes, like they did in the interfaces of the *SearchResultsViewController* and the *HistoricResultsViewController* objects.

Figure 5.25: HistroyTableViewController: Select two publications for comparison

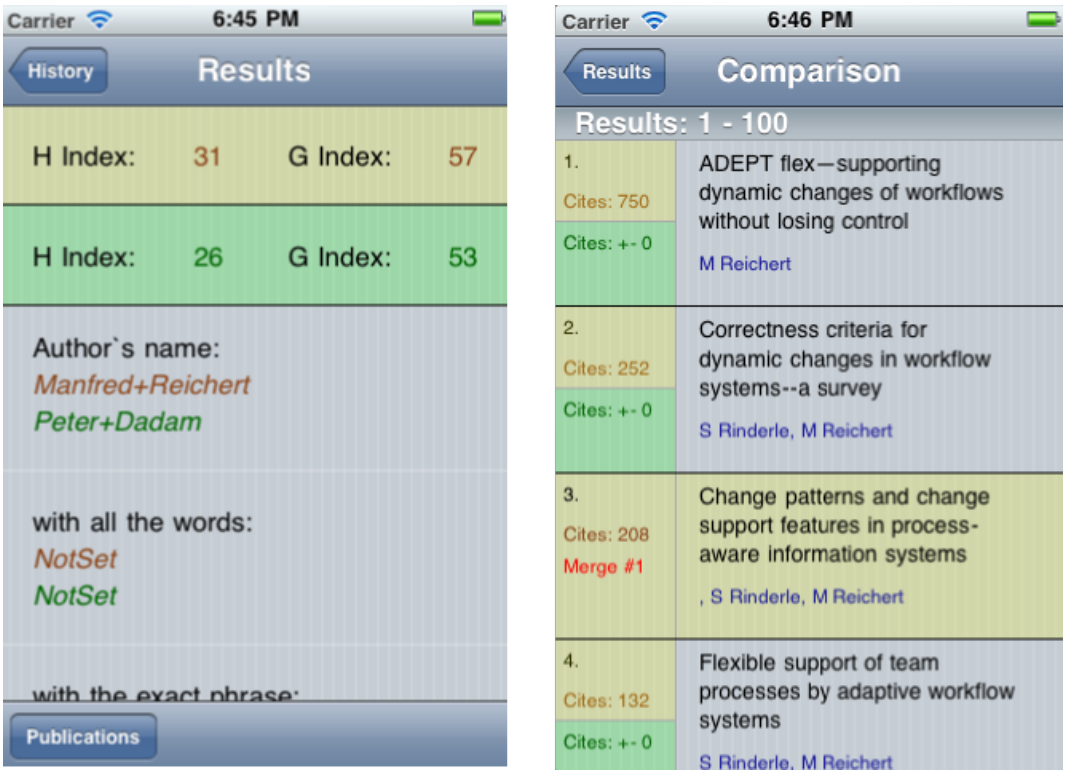

Figure 5.26: HistroyTableViewController: Select two publications for comparison

We have already introduced all the techniques which we have used to implement the comparison feature. The selection of the two publication in the history table view controller's interface is the same approach that we have used in order to select publication for merging 5.3.1. The *ComparisonResultsViewController* class inherits from the *IntermediateViewController* class and implements a user interface to show the two results at once. By now we have talked about the *UITableViewController* class quite a lot. The *CompareTableViewController* class is just another example as it inherits from our *BasicTableViewController* class.

5.6 About and Instructions

At this point we have completed the explanation of the required features. In the "Architecture" chapter 4 we explained the basic structure of our application and said that it consists of a *UITabBarController* object with three tabs. You may have noticed that all custom controllers we described in the "Implementation" chapter so far, are located in the left or in the

middle tab. The right one is still missing. The right tab does not contain any more serious features. Its function simply is to present some information about the application and instructions about all our features. A controller of the type *AboutViewController* is the root controller of the third tabs navigation controller. Figure 5.27 shows the user interface of the *AboutViewController* class. It has a corresponding XIB file containing two *UITextViews* and three buttons. The buttons all are connected to respective *IBActions* in the controller. The first button is represented by an email address. By tapping it, once again an instance of *MFMailComposeViewController* is being created and presented modally. The second button is our institute's web address. By tapping it, a *WebViewController* object is being presented in order to load our institutes website. By tapping the third button, an instance of the *InstructionsViewController* is being presented. The *InstructionsViewController* class does not inherit from the *WebViewController* class but implements similar functionality. As the name suggest, the instructions view controller's interface presents the instructions as illustrated by figure 5.28. Therefore the corresponding XIB file contains the exact same user interface elements as the interface of the *WebViewController* class. Most important, the *UIWebView*. Additionally the tool bar holds the "*Instructions*" button. The reason why the *InstructionsViewController* class is so similar to the *WebViewController* class is because the instructions view controller presents a local website which is being stored in the application bundle and contains the instructions of all features. The additional "*Instructions*" button serves as a "*Home*" button which provides functionality to always return to the initially presented instructions site.

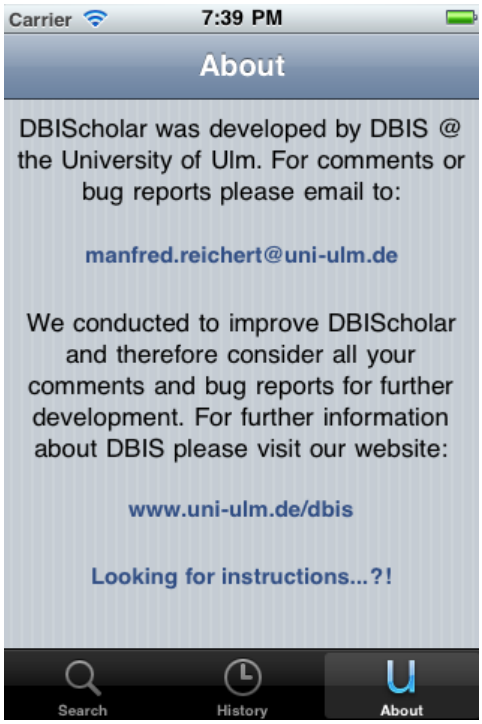

Figure 5.27: AboutViewController's user interface

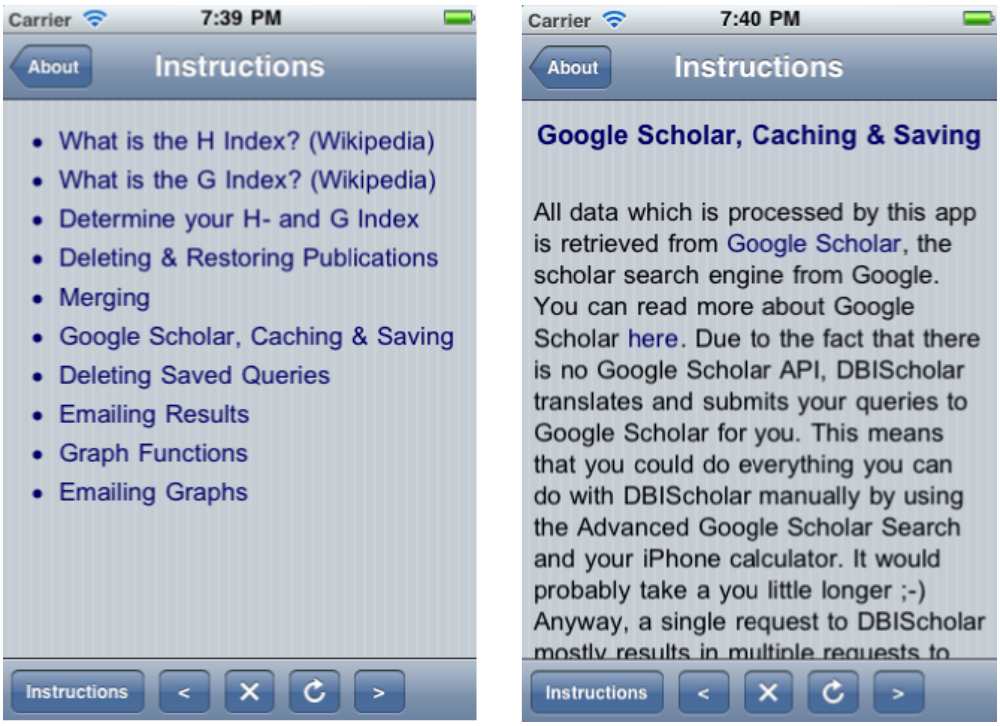

Figure 5.28: InstructionsViewController's user interface

5.7 Remember Feature

Last but not least we will talk about the "*Try to editing copy of the latest stored query*" switch in the search view controller's interface (See figure 5.5) and its functionality. Switches are of the class *UISwitch* which implements functionality identical to a real switch: you can turn it on and off. However, let us assume you have submitted a search sometime in the past, deleted all publications which did not match your request, merged all publications which belong together and then saved the results. Now you would like to do the same search again, maybe to check if the indexes or specific citation counts increased. Then you would have to query the same search and perform the exact same editing on the result like you did last time. The function behind the "*Try to editing copy of the latest stored query*" switch is to automate this. If the "*Try to editing copy of the latest stored query*" switch is turned on before you search, the application checks if there are saved results which belong to an equal search request. If this is the case, the application automatically tries to perform the exact same editing on your new search results after retrieving them like you did on the

latest saved results of the respective equal search.

Due to the fact that we are not sure whether the Google results allow to be identified distinctly, we assume that the titles of publications usually are unique. We know that there are cases in which this does not apply. We even discovered cases where the title of the exact same publication varied from one search to another. But we assume that these cases are very rare and only emerge due to poor performance of the Google Scholar search engine or based on disadvantageously entitling of publications. Thus we (Compare 5.3 and 5.5) use the title as an identifier for a specific publication. This again means we have to do a lot of string comparison here. Based on the matchings of the strings, database functions are being triggered to perform deletions and merging of the new publications with the same titles. Due to the identification problems this feature can not guarantee to reproduce all the editing of the latest stored results accurately.

5.8 iOS Frameworks

In the appendix of Apple's iOS Technology Overview there is a nice table [29] which gives an overview of all iOS frameworks, a short explanation for each and a link for further resources. The table 5.12 lists most of these frameworks and the corresponding descriptions. All frameworks which we actually used for development are marked in blue. Note, that all frameworks coloured in green are available since iOS version 4.0 or above. Therefore they were not yet available, when we started development. However, the tables point out the spectrum of available technologies and possibilities that iOS development has reached by today.

Table 5.12: Frameworks used for development

Framework	**Description**
Accelerate.framework	Contains accelerated math and DSP functions
AddressBook.framework	Contains functions for accessing the user's contacts database directly.
AddressBookUI.framework	Contains classes for displaying the system-defined people picker and editor interfaces.
AssetsLibrary.framework	Contains classes for accessing the user's photos and videos.

Table 5.12: Frameworks used for development

Framework	Description
AudioToolbox.framework	Contains the interfaces for handling audio stream data and for playing and recording audio.
AudioUnit.framework	Contains the interfaces for loading and using audio units.
AVFoundation.framework	Contains Objective-C interfaces for playing and recording audio and video.
CFNetwork.framework	Contains interfaces for accessing the network via Wi-Fi and cellular radios.
CoreAudio.framework	Provides the data types used throughout Core Audio.
CoreData.framework	Contains interfaces for managing your application's data model.
CoreFoundation.framework	Provides fundamental software services, including abstractions for common data types, string utilities, collection utilities, resource management, and preferences.
CoreGraphics.framework	Contains the interfaces for Quartz 2D.
CoreLocation.framework	Contains the interfaces for determining the user's location.
CoreMedia.framework	Contains low-level routines for manipulating audio and video.
CoreMIDI.framework	Contains low-level routines for handling MIDI data.
CoreMotion.framework	Contains interfaces for accessing accelerometer and gyro data.
CoreTelephony.framework	Contains routines for accessing telephony-related information.
CoreText.framework	Contains a text layout and rendering engine.

Table 5.12: Frameworks used for development

Framework	Description
EventKit.framework	Contains interfaces for accessing a user's calendar event data.
EventKitUI.framework	Contains classes for displaying the standard system calendar interfaces.
ExternalAccessory.framework	Contains interfaces for communicating with attached hardware accessories.
Foundation.framework	Contains interfaces for managing strings, collections, and other low-level data types
GameKit.framework	Contains the interfaces for managing peer-to-peer connectivity.
iAd.framework	Contains classes for displaying advertisements in your application.
ImageIO.framework	Contains classes for reading and writing image data.
MapKit.framework	Contains classes for embedding a map interface into your application and for reverse-geocoding coordinates.
MediaPlayer.framework	Contains interfaces for playing full-screen video.
MessageUI.framework	Contains interfaces for composing and queuing email messages.
OpenGLES.framework	Contains the interfaces for OpenGL ES, which is an embedded version of the OpenGL cross-platform 2D and 3D graphics rendering library.
QuartzCore.framework	Contains the Core Animation interfaces.
QuickLook.framework	Contains interfaces for previewing files.
Security.framework	Contains interfaces for managing certificates, public and private keys, and trust policies.

Table 5.12: Frameworks used for development

Framework	**Description**
StoreKit.framework	Contains interfaces for handling the financial transactions associated with in-app purchases.
SystemConfiguration.framework	Contains interfaces for determining the network configuration of a device.
UIKit.framework	Contains classes and methods for the iOS application user interface layer.

6 Bug

There is one disappointing aspect we have to talk about. There is a bug in our application. After a few weeks of development we discovered that our app sometimes gets stuck. It does not crash, it just does not respond any more. We figured that it has got something to do with our table view controllers. Funny thing though, when the device is being shook, the app suddenly responses again. It seems that the bug has got something to do with animating the table view when the respective table controller is being pushed and popped on or off the navigation stack. It seems not to occur when either the animation is not being applied in the *pushViewController: animated:* method of the underlying navigation controller or when we relinquish on the data source's *sectionIndexTitlesForTableView:* method for navigating within the table. Initially we thought of course that this is our fault and we might have messed up memory management at some point. Thus we searched for the bug for quite a while but could not find anything. Our next step was to break down everything to a simple example in order to reproduce the bug. Fortunately we could achieve this with a XCode application template and a few additional lines of code. At this point it seemed unlikely that we have messed up something and everything pointed to the fact that this could be an iOS bug. Fortunately the Apple's developer program provides technical support. In the Standard Developer Program, you can submit two TSIs (Technical Support Incident) each year [32]. After a few more weeks of putting the bug aside we submitted a TSI reporting the issue. The Apple technician who responded on our TSI analysed that it seems to be an iOS bug. He proposed to rewrite a part of the code so that the bug would not occur. Unfortunately that did not work. By now we do not have a solution for the problem yet. In the appendix A.1 you can read our TSI and Apple's response.

7 App Store Submission

When your application is ready for the submission to the App Store, you will initially have to go through a similar procedure than the one of preparing your application to run on a device for development purposes. Unfortunately this time you will have to perform even more steps. I can understand when developers experience problems with submitting an app for distribution in the App Store the first time as it requires a lot of steps before you can actually upload your application. The provisioning portal on the other hand provides documents which should give you the answers to all of your questions concerning App submission. For the distribution you have to create a distribution certificate and the corresponding private key. The steps are similar to setting up the provisioning profile for test purposes. This time you have to create an *App Store Distribution Provisioning Profile* in order to build your app for distribution via the App Store. Moreover you have to configure your application for the distribution. All necessary steps can be found in the provisioning portal under the *Distribution* section [27]. When you have finally built your application for distribution, you are still not quite done yet. You now have to log in at iTunes Connect (https://itunesconnect.apple.com/) with your App ID and add a new Application. Therefore you need to provide a description, screen shots, icons and a URL to your app's website. When all this is done, you are ready to upload your binary via Application Loader. Application Loader comes with iOS SDK 3.2 and later and allows you to upload your binary.

7.1 Distribution Alternatives

The App Store is not the exclusive way of distributing apps. We have already mentioned three different Developer Programs in section 2.6: the *Standard Individual Program*, the *Standard Company Program* and the *iOS Developer University Program*. If you are enrolled into either the *Standard Individual Program* or the *Standard Company Program* you can distribute your apps via the App Store. Moreover you have the ability to install your application on up to 100 devices via Ad Hoc distribution. More information about Ad Hoc distribution can be found in the iOS Provisioning Portal of the iOS Developer Center [30]. If you plan on developing an app for a company and therefore need to distribute your app on more than 100 devices, there is a developer program especially for that: the *iOS Developer*

Enterprise Program. It costs $299 per year and exclusively allows to distribute apps via Ad Hoc but not via the App Store. To enrol into the *iOS Developer Enterprise Program* a *DUNS Number* [38] is required. For further information see the iOS Developer Program [28].

7.2 Rejected

When you have uploaded your app you can check the status of the review in iTunes Connect [7]. For our application the review process took about ten days. But that probably depends on the complexity, functionality and content of your app and on how busy they are reviewing other apps by the time you submit yours. We have mentioned some of the reasons which cause apps to be rejected earlier in section 2.5 in the chapter "iPhone Development Introduction". We knowingly used a non-public API, the method *addTextFieldWithValue: label:* of the class *UIAlertView* to add a text field to the alert view in order to limit the amount of publications being emailed with the results. The current implementation of this functionality was explained back in section 5.2.2 in the "Implementation" chapter. However everything worked well with the initial implementation and one could limit the amount of publications being sent with the results by typing in a number in the text field of the appearing alert view. But when we uploaded our app for the review, it got rejected. We received an email pointing out the respective rule of the Review Guidelines: *2.5 Apps that use non-public APIs will be rejected*. Thus we replaced the alert view simply with a view controller which we present modally to ask the user whether he would like to limit the amount of publications being sent with the results. We uploaded the app once again and after about 8 days, the app was being accepted by Apple and finally was available in the App Store. Based on this experience you can see that Apple is serious about the Review Guidelines. There are quite many restrictions and you better check them before you invest your time in development.

8 Conclusion

With the development of our application we could proof that the iOS provides some basic technologies for the development of mobile business applications for the iPhone. We could meet most of the requirements which we defined for our application. The development frameworks provide enough technologies which allow to port lightweight business applications on the iPhone. The platform is ready for thin client applications and it is possible to outsource partial functionality from a desktop computer onto the iPhone. On the other hand developing for the iPhone is more complex than developing an application for a usual personal computer. You have to deal with memory management and restricted resources on one side and with building a rich user interface for a small multi touch screen on the other. Apple's frameworks and guidelines assist the developer in designing a rich user interface but cannot compensate all constraints which come with the small screen and the nature of the device such as having to hold it during typing. Of course the device has several advantages such as mobility, which a computer simply cannot provide. And the small touch screen interface even might be the better solution in some scenarios compared to the usual computer input interfaces. In order to gain more knowledge about the usability of the device, it would be necessary to analyse user performance for several tasks of different complexity in comparison with the same tasks on a computer. Even though there are several development frameworks available in the iOS SDK, the technology outline by far cannot compete with the possibilities of software development for personal computers. Building a complex application requires to seriously investigate feasibility in general. In case the basic conditions are being satisfied, the development still will take an adequate time.

A Appendix

A.1 TSI

A.1.1 Our TSI

Hello,

we seem to have a strange bug in our application,

UITableView - sectionIndexTitlesForTableView seems to cause the application not to respond on touches. When it gets stuck and one shakes the device, it responds again.

Attached code has to be installed on a device! In simulator the bug only appears occasionally.

Steps to reproduce:

To reproduce the status where the app is not responding to touches, you have to press the initially presented button in the app, scroll down and subsequently navigate back (to the root view controller presenting the button) while the table view still is animating the scrolling

and then repeat these steps a few times.

Eventually, after a few times, the app will not respond after the table view was being presented.

When you then shake the device the app will respond again.

If it just will not happen restart your device and try again. I promise at some point it will get stuck. I can make a video if necessary.

I have experienced this situation in an app I am developing currently. It did not respond in a view controller which presents another view controller containing a table view. So the bug appears in a different situation. But the reason should be the same. I figured out that the method sectionIndexTitlesForTableView seems to have something to do with it. If I do not implement it, it works just fine.

I could reproduce the bug on multiple devices (iPodTouch 2nd gen, 4th gen, iPhone 4th gen) as well as with multiple SDKs (iOS 4.0, iOS 4.1 and I am quite sure also in iOS 3.2)

Thanks a lot for checking it out.

Cheers, Andreas Robecke

A.1.2 Apple's response

Hello,

Thank you for your inquiry to Apple Worldwide Developer Technical Support. I am responding to let you know that I have received your request for technical assistance.

I have been able to reproduce the problem. I believe there is race condition somewhere due to the animation firing and the releasing of the table view controller when the back button is pressed. I have not confirmed if "sectionIndexTitlesForTableView" is the smoking gun for the cause of the problem.

This is a very strange bug indeed. I would suggest not allocating, releasing, and reallocating this view controller each time you navigate. This may be the heart of the problem. I would suggest creating your TableView instance in viewDidLoad inside your RootViewController and use that instance when pushing and popping. This will keep the retain count at 1 and never cause the release to occur while switching back and forth. I've tested it a bit and couldn't reproduce the problem.

I would consider this a bug in iOS and you can file one at "http://bugreporter.apple.com".

Best regards,
Keith Mortensen
Apple Developer Technical Support

Bibliography

[1] *Google Scholar*. http://scholar.google.com/

[2] *RegexKitLite*. http://regexkit.sourceforge.net/RegexKitLite/

[3] *Scopus*. http://www.scopus.com/

[4] *Web of Science*. http://isiwebofknowledge.com/

[5] *UI Tab Bar Controller Class Reference*. http://developer.apple.com/library/ios/#documentation/uikit/reference/UITabBarController_Class/Reference/Reference.html. Version: 2009

[6] *Nib File Management*. http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/IB_UserGuide/BuildingaNibFile/BuildingaNibFile.html#//apple_ref/doc/uid/TP40005344-CH11-SW1. Version: 2010

[7] https://itunesconnect.apple.com/

[8] APPLE: *iPhone in Business*. http://www.apple.com/iphone/business/profiles/

[9] APPLE: *Message UI Framework Reference*. http://developer.apple.com/library/ios/#documentation/MessageUI/Reference/MessageUI_Framework_Reference/_index.html

[10] APPLE: *Modal View Controllers*. http://developer.apple.com/library/ios/#featuredarticles/ViewControllerPGforiPhoneOS/ModalViewControllers/ModalViewControllers.html

[11] APPLE: *Program Enrollment*. http://developer.apple.com/support/ios/enrollment.html

[12] APPLE: *Quartz 2D Programming Guide*. http://developer.apple.com/library/ios/#documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/dq_pdf/dq_pdf.html#//apple_ref/doc/uid/TP30001066-CH214-TPXREF101

[13] APPLE: *Quartz 2D Programming Guide*. http://developer.apple.com/library/ios/#documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/Introduction/Introduction.html

[14] APPLE: *Table View Programming Guide for iOS*. http://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/TableView_iPhone/AboutTableViewsiPhone/AboutTableViewsiPhone.html#//apple_ref/doc/uid/TP40007451

[15] APPLE: *UITableViewDataSource Protocol Reference*. http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableViewDataSource_Protocol/Reference/Reference.html

[16] APPLE: *UITableViewDelegate Protocol Reference*. http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableViewDelegate_Protocol/Reference/Reference.html

[17] APPLE: *UIWebView Class Reference*. http://developer.apple.com/library/ios/#documentation/uikit/reference/UIWebView_Class/Reference/Reference.html

[18] APPLE: *URL Loading System Programming Guide*. https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/URLLoadingSystem/URLLoadingSystem.html

[19] APPLE: *XMLPerformance*. http://developer.apple.com/library/ios/#samplecode/XMLPerformance/Introduction/Intro.html

[20] APPLE: *XMLPerformance ReadMe.txt*. http://developer.apple.com/library/ios/#samplecode/XMLPerformance/Listings/ReadMe_txt.html#//apple_ref/doc/uid/DTS40008094-ReadMe_txt-DontLinkElementID_23

[21] APPLE: *Archives and Serializations Programming Guide*. https://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/Archiving/Archiving.html. Version: 2010

[22] APPLE: *UITableView Class Reference*. http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableView_Class/Reference/Reference.html#//apple_ref/occ/instm/UITableView/dequeueReusableCellWithIdentifier:. Version: 2010

[23] APPLE: *UITableViewCell Class Reference*. http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableViewCell_Class/Reference/Reference.html#//apple_ref/occ/cl/UITableViewCell. Version: 2010

[24] APPLE: *Memory Management Programming Guide*. http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/MemoryMgmt/Articles/mmRules.html. Version: 2010-06-24

[25] APPLE: *Introduction to The Objective-C Programming Language*. http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html. Version: 2010-07-13

[26] APPLE: *UIViewController Class Reference*. http://developer.apple.com/library/ios/#documentation/uikit/reference/UIViewController_Class/Reference/Reference.html. Version: 2010-10-13

[27] APPLE: *Distribution*. http://developer.apple.com/ios/manage/distribution/index.action. Version: 2011

[28] APPLE: *iOS Developer Enterprise Program*. http://developer.apple.com/programs/ios/enterprise/. Version: 2011

[29] APPLE: *iOS Frameworks*. http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSFrameworks/iPhoneOSFrameworks.html#//apple_ref/doc/uid/TP40007898-CH6-SW3. Version: 2011

[30] APPLE: *iOS Provisioning Portal*. https://developer.apple.com/ios/manage/overview/index.action. Version: 2011

[31] APPLE: *Obtaining your iOS Development Certificate*. https://developer.apple.com/ios/manage/certificates/team/howto.action. Version: 2011

[32] APPLE: *Technical Support*. http://developer.apple.com/support/resources/technical-support.html. Version: 2011

[33] APPLE: *iOS Human Interface Guidelines*. https://developer.apple.com/library/ios/#documentation/UserExperience/Conceptual/MobileHIG/Introduction/Introduction.html. Version: 2011-01-03

[34] BAR-ILAN, JUDIT: Which h-index? – A comparison of WoS, Scopus and Google Scholar. In: *Scientometrics, Vol. 74, No. 2 (2008) 257–271* (2007)

[35] BOSMAN, Jeroen: Scopus reviewed and compared. (June 2006)

[36] BRADLEY, Tom: *TBXML*. http://www.tbxml.co.uk/TBXML/TBXML_Free.html. Version: 20th May 2010

[37] COLLECTION.DE cpu: *CPUs by Production Year*. http://www.cpu-collection.de/. Version: 2011

[38] DNBGERMANY: *D&B*. http://www.dnbgermany.de/

[39] EGGHE, LEO: Theory and practise of the g-index. In: *Scientometrics* (2006)

[40] GUGLIOTTA, Guy: The Genius Index: One Scientist's Crusade to Rewrite Reputation Rules. In: *Wired Magazine* (17.06.2009). http://www.wired.com/culture/geekipedia/magazine/17-06/mf_impactfactor

[41] HARZING, Prof. Anne-Wil: Google Scholar - a new data source for citation analysis. (2007)

[42] HIRSCH, J E.: An index to quantify an individual's scientific output. In: *Proc. Natl. Acad. Sci. U. S. A.* 46 (2005), Aug, Nr. physics/0508025, S. 16569

[43] MAHER ALI, PhD: *iPhone SDK 3 Programming*. A John Wiley and Sons, Ltd, Publication, 2009

[44] MARK, Dave ; LAMARCHE, Jeff: *Beginning iPhone 3 Development: Exploring the iPhone SDK*. Apress, 2009

[45] MEHO, Lokman I. ; YANG, Kiduk: A New Era in Citation and Bibliometric Analyses: Web of Science, Scopus, and Google Scholar. In: *Journal of the American Society for Information Science and Technology* (2006)

[46] NOORDEN, Richard V.: Hirsch index ranks top chemists. In: *Royal Society of Chemistry* (23 April 2007). http://www.rsc.org/chemistryworld/news/2007/april/23040701.asp

[47] PALSBERG, Jens: The h Index for Computer Science. (2011). http://www.cs.ucla.edu/~palsberg/h-number.html

[48] SAX: *Official website for SAX*. http://sax.sourceforge.net/

[49] TOUCHXML: *TouchXML*. https://github.com/TouchCode/TouchXML

[50] W3C: *Document Object Model (DOM)*. http://www.w3.org/DOM/

[51] W3C: *XPath Language*. http://www.w3.org/TR/xpath20/

Name: Andreas Robecke Matrikelnummer: 541382

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Andreas Robecke