

To  
S. Byland  
Physikinstitut  
MNG Rämibühl  
8001 Zürich

*Maturitätsarbeit*

ARCHITECTING  
RICH INTERNET APPLICATIONS

Daniel Gașienica  
March 2, 2005

## Abstract

Today, RICH INTERNET APPLICATIONS (RIA) are the tip of the evolution of web deployed applications. They are the next step in the process of overcoming the boundaries posed by the web as we know it. RIA make use of a rich client, which enables developers to create more responsive, intuitive, and accessible user interfaces than ever before.

In the first part of this paper the fundamentals of RICH INTERNET APPLICATIONS are described: their architecture, underlying technologies, and programming techniques used in their development. In the second part, the process and experiences of building PHYRE, a *real world* RIA, are portrayed. PHYRE's task is to give to the physics department of the MNG Rämibühl access to its catalog of experiments, which can be carried out at the school. PHYRE is based upon the Macromedia Flash Player acting as rich client and otherwise makes use of free and open source technologies. Relevant issues from all parts of the development of a RICH INTERNET APPLICATION are discussed, such as database modeling, dealing with hierarchical data, user authentication & access control, and applying design patterns in RICH INTERNET APPLICATIONS, just to name a few. As the title suggests, one of the main topics of this paper is the architecture of RICH INTERNET APPLICATIONS which is discussed using PHYRE as an example.

Additionally, the complete source of PHYRE, the RICH INTERNET APPLICATION discussed in this paper, is made available to the reader.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Who Should (and Shouldn't) Read This Paper . . . . .	4
1.2	The Evolution of Web Applications . . . . .	5
1.3	What Is a RICH INTERNET APPLICATION? . . . . .	5
1.4	Finding the Right Task for an RIA . . . . .	6
<b>I</b>	<b>Part One: Theoretical Background</b>	<b>7</b>
<b>2</b>	<b>PHYRE: Physics Rämibühl Experiments</b>	<b>7</b>
2.1	Features . . . . .	7
2.2	Specifications . . . . .	7
2.2.1	Functional Requirements . . . . .	8
2.2.2	Business Rules . . . . .	8
2.2.3	Technical Requirements . . . . .	9
2.2.4	User Requirements . . . . .	10
<b>3</b>	<b>Technologies</b>	<b>11</b>
3.1	Introduction . . . . .	11
3.2	Macromedia Flash . . . . .	11
3.3	AMFPHP . . . . .	11
3.4	MySQL . . . . .	13
<b>4</b>	<b>Programming Techniques</b>	<b>14</b>
4.1	Introduction . . . . .	14
4.2	OOP: Object-Oriented Programming . . . . .	14
4.3	Design Patterns . . . . .	15
4.4	Refactoring . . . . .	15
<b>5</b>	<b>Three-Tiered Application Architecture</b>	<b>16</b>
5.1	Introduction . . . . .	16
5.2	Data Tier . . . . .	16
5.3	Business Tier . . . . .	16
5.4	Presentation Tier . . . . .	17
5.5	Benefits . . . . .	17

<b>II Part Two: Building PHYRE</b>	<b>18</b>
<b>6 PHYRE's Catalog Database</b>	<b>18</b>
6.1 The Entity-Relationship Model . . . . .	18
6.2 Relationships & Business Rules . . . . .	19
<b>7 Hierarchical Data &amp; Recursion</b>	<b>24</b>
<b>8 User Authentication &amp; Access Control</b>	<b>28</b>
8.1 Introduction . . . . .	28
8.2 Sensitive Data & MD5 . . . . .	28
8.3 Implementation . . . . .	30
8.3.1 Prerequisites . . . . .	30
8.3.2 Login . . . . .	31
8.3.3 Logout . . . . .	37
8.4 Conclusions . . . . .	37
8.5 Code Listings . . . . .	38
<b>9 Design Patterns in PHYRE</b>	<b>40</b>
9.1 Singleton . . . . .	40
9.2 Proxy . . . . .	41
9.3 Further Patterns Found in PHYRE . . . . .	43
<b>10 UI Design: PHYRE's Single-Screen Interface</b>	<b>44</b>
<b>11 Closing Thoughts</b>	<b>48</b>
<b>12 Acknowledgements</b>	<b>50</b>
<b>A Additional Resources</b>	<b>51</b>
<b>B Tools</b>	<b>52</b>
<b>C Source Code</b>	<b>52</b>
<b>References</b>	<b>53</b>

## Foreword

Since I was little I have carried around the passion to *create*. When I was a young boy this appeared in the sheer endless number of LEGO models I built. In my adolescence I ran across the fascinating world of computers. I made my first steps as a mere consumer playing video games and surfing the internet, although I soon began to ask myself where all those web sites I browsed came from. At that time, I was lucky to have a new classmate who already had experience in designing web sites and who actually built and maintained a *Dragonball*<sup>1</sup> fan web site. I asked him to lend me an introductory book on the topic. The next day already, I came home with a beginner's book on HTML, turned on my computer, opened *Notepad*<sup>2</sup> and started to write my first markup. Those early experiences were responsible that I kept on learning and did not stop until this very day. Looking back, the results from then would now probably make me close my eyes as well as my web browser. Feeling limited and often annoyed by the comparatively little possibilities that plain HTML offered, one day I came across a tool called Flash. Flash promised to overcome many of the boundaries posed by conventional web design. Between 2000, the year I started to experiment with Flash 4, and now, a lot changed: the tool comes in its seventh version called Flash MX 2004 and my experience has grown constantly with the release of new versions.

As the time arose to find a suitable topic to write my *Maturitätsarbeit* about, it was somehow obvious that I should choose a topic that had to do with building web sites. But as it turned out I was not interested in discussing something I wasn't able to investigate and elaborate any further than I already did. As disturbing as this situation was, I allowed myself to virtually consider a completely different topic. After a lengthy conversation with my brother, I got back on track and knew that I had to find a topic related to building web sites, even so, one that would give me the chance to expand my knowledge. Having read many articles about it, the term RICH INTERNET APPLICATIONS (RIA) suddenly came to my mind. The topic offered so many aspects to delve into as well as the opportunity to actually plan, build, and deploy an RIA. This very opportunity was eventually the determining factor for my choice. I am not only excited to have found a topic to which I could apply the skills I have collected over the years but also to face new challenges and, most importantly, gather a lot of practical experience. Today, the challenge of building RICH INTERNET APPLICATIONS is not much different for me than the one of building models with LEGO when I was little — just the tools, approaches, and results are others.

— Daniel Gąsienica

---

<sup>1</sup>Japanese comic book

<sup>2</sup>basic text editor which ships with Microsoft Windows

# 1 Introduction

## 1.1 Who Should (and Shouldn't) Read This Paper

If you can answer any of the following statements with *yes*, you should definitely consider reading this paper.

- You are interested in knowing how certain things really work.
- You want to see in what direction the WWW is going.
- You want to find out more about the underlying architecture of web applications.
- You are building conventional web applications and want to find out about the advantages of RICH INTERNET APPLICATIONS.
- You are developing for the client-side and want to see how the server-side works.
- You are good at writing server-side applications and want to know what possibilities there are to create *rich* user interfaces to those applications.
- You already know what a RICH INTERNET APPLICATION is and want to analyze the complete source of a *real world* RIA.

On the other hand, do not read this paper if you are looking for a step by step tutorial on how to build a RICH INTERNET APPLICATION or if you expect this paper to be a beginner's guide to programming.

I hope you will enjoy reading this paper as much as I did writing it!

---

*Important Note: In order to fully understand the contents of this paper it is helpful to have experience working with Macromedia Flash and Flash Remoting. Comprehension of the code examples and the source code requires advanced skills in different programming languages such as ActionScript 2.0 and PHP as well as a certain understanding of relational database management systems (RDBMS) and the structured query language (SQL).*

## 1.2 The Evolution of Web Applications

The World Wide Web's inventor, Sir Tim Berners-Lee, once mentioned that *"the original idea of the web was that it should be a collaborative space where you can communicate through sharing information."* [1]. At that time, in the early 1990s, hardly anyone had anticipated what followed. The use of the WWW in its early days had another background than today. The purpose of the early WWW was to display plain information with references to other web pages, called *hyperlinks*. The revolutionary idea behind the WWW was that anyone could publish his findings at no charge and make them available to everyone. As soon as personal computers with an internet connection were available to many households across the world, nothing was able to stop the rapid progression of the *information age*. When businesses started to recognize the potential of the web deployment model, they began to use HTML not just for their web sites but also to create interfaces to their applications. Most web applications did not offer the same usability and user experience as their desktop counterparts because not only did HTML provide a limited set of user interface controls, but it also lacked a client-side data model. During the past years the technologies needed to use the web as a deployment platform for applications matured and were standardized. However while the back-end took part in the evolution, the presentation layer did not. Traditional web applications are still written in HTML 4.0, a standard adopted in 1998, and in no way designed for the purpose of providing sophisticated, responsive and intuitive user interfaces. The growing web application niche is where Macromedia, a company developing software for the creation of digital content, claims a share with its solutions for RICH INTERNET APPLICATIONS.

## 1.3 What Is a RICH INTERNET APPLICATION?

Macromedia Flash MX, a digital content authoring tool, and at the same time Flash Player 6, its client-side plug-in, were released in early 2002 and promised to provide a platform for a new generation of web applications called RICH INTERNET APPLICATIONS. The term RICH INTERNET APPLICATIONS (RIA) was introduced by Jeremy Allaire, the founder of Allaire Inc., a web company that later merged with Macromedia, and is discussed in detail in the whitepaper *Macromedia Flash MX: A Next-Generation Rich Client* [2].

Recapitulatory, it states that *"RICH INTERNET APPLICATIONS combine the functionality of desktop software applications with the broad reach and low-cost deployment of web applications – resulting in significantly more intuitive, responsive, and effective user experiences."* [3].

Good examples of RIA are scarce (another reason for me to build one) but the number is growing steadily. RIA are found where the kind of in-

teraction is needed which conventional web applications cannot offer. A superb example for an RIA is the "car configurator" application designed for the US web site of the car manufacturer MINI [4]. RICH INTERNET APPLICATIONS not always have to be as complex as the latter: E\*TRADE developed a light-weight RIA that enables users of their web site to look into stock information without the need of time-consuming page refreshes which also generate large traffic for the server, and thus additional avoidable cost [5]. The E\*TRADE RIA is also a good example for the fact that RIA do not have to be stand-alone applications which take up the entire screen but can also be incorporated into any static web site to perform only certain, adequate tasks.

Even though this paper discusses solely RICH INTERNET APPLICATIONS that are based upon the Flash Player<sup>3</sup> acting as rich client, it is worth mentioning that there are other, competing technologies available or in work. The future will definitely bring forward more RIA, or similar application deployment models, most prominently Microsoft with their next version of Windows, code-named *Longhorn*, which is expected in late 2006. Stay tuned!

#### 1.4 Finding the Right Task for an RIA

The most important aspect to me when I learn something new, is to have examples that illustrate the issues and bear a reference to applications in the *real world*. Primarily, the goal of my *Maturitätsarbeit* is to plan, build and deploy a RICH INTERNET APPLICATION. This paper's purpose is guide the reader through that process and to illustrate, explain, and document some of the challenges faced within it. To make it more interesting and challenging for me, I was looking for an idea for an RIA that would address a real world problem. This will allow readers to apply the *techniques* and *architecture* discussed in this paper to their own projects. Since I will disclose the complete source of this project, readers get the chance not just to see one of the many already available *sample* RIA, but an actual application that was built to handle a *real world* task.

After talking to my tutor, Mr. Byland, we soon found a suitable task for me. He introduced me to the experiments catalog of the *Physics Institute Rämibühl*. Basically, it was a handful of binders with data sheets listing and describing the experiments which the physics teachers can carry out at our school. Even if the catalog served its purpose, we both saw that there was plenty of room for improvement, so I accepted the challenge of turning it into an RIA. PHYRE is the name I chose for the RIA. It is the acronym for *Physics Rämibühl Experiments* and is to be pronounced like *fire*.

---

<sup>3</sup>At the date of publication, the current version is Flash Player 7

## Part I

# Theoretical Background

## 2 PHYRE: Physics Rämibühl Experiments

### 2.1 Features

Naturally, PHYRE has to perform all the tasks the original catalog does. Nevertheless, there are many features I have in mind that will make the catalog more accessible and flexible:

- As the name RICH INTERNET APPLICATION implies, PHYRE will run online, this itself brings one dramatic advantage over the previous catalog: *ubiquitous availability* from any internet enabled device.
- *Data redundancy* is avoided by using a single, centralized data source, a database that holds all the information about the experiments.
- By assigning *meta keywords* to all the experiments the catalog will be *fully searchable*. This will eliminate the problem of not finding certain experiments because of the very specific names they have.
- The data sheet of each experiment in PHYRE will make all the necessary details available *at a glance*: each component needed for the set-up, the settings, attached illustrations, photos and schematic diagrams.
- The RIA's *single-screen interface*, which is discussed in section 10, makes it easier for users to find what they need more quickly.
- *User authentication* allows me to incorporate an *administration mode* into the very same application, so anyone who has logged in can edit, create, and delete experiments.
- Running the client in the Flash Player, hence having a persistent data model, avoids otherwise necessary page refreshes to update data in the application.

### 2.2 Specifications

Meeting with the people responsible for the original catalog and the future users of the RIA, Mr. Byland and two physics assistants, we discuss how the specifications for the application should look like. The specifications include *functional* and *technical requirements*. Before meeting, both parties collect some thoughts and expectations for the new catalog. The school's representatives work out the *functional requirements*, i.e. the tasks

the RIA should be able to handle, since they will eventually be the users of the new system and have experienced the advantages and disadvantages of the previous set-up. Since I possess the technical know-how, my thoughts are mainly concerned with the *technical requirements* of the RIA and the transformation of the *functional requirements* into *business rules*, which are both discussed in the next sections.

### 2.2.1 Functional Requirements

The people involved at the school want the following features, called *functional requirements*, to be incorporated into PHYRE. They expect that users working with PHYRE should be able to ...

- ... browse the catalog for experiments.
- ... find experiments by using a built-in full-text search.
- ... view information on experiments, including associated files, components and their storage location.
- ... add, update, and delete experiments.
- ... add, update, and delete components.
- ... upload, update, and delete files.
- ... link components and files to experiments.

### 2.2.2 Business Rules

"[Business Rules are] *the laws, regulations, policies, and procedures that are encoded into a computer system. Also known as business logic.*" [6]

The essential *business rules* for PHYRE were derived from the original experiments catalog and the *functional requirements* as follows:

- I The catalog is made up of different categories.
- II Categories can be nested.
- III Categories hold experiments.
- IV Experiments are assigned to not more than one category.
- V Experiments have a name and description.
- VI Experiments contain components and files.
- VII Components and files can be shared among several experiments.
- VIII Components have an ID that defines their storage location.

### 2.2.3 Technical Requirements

Contrary to web sites, RICH INTERNET APPLICATIONS need a *rich* client to run on. For PHYRE I chose the Macromedia Flash Player to act as the rich client. One of the goals I set myself, is to build the application to be open to extension. This is achieved by a *tier-, component- and object-oriented* approach in the design of the application architecture, which is discussed in section 5. PHYRE makes use of the following computer languages<sup>4</sup>:

**AS 2.0** *ActionScript 2.0* is used to control the behavior of Flash movies<sup>5</sup> and is, similarly to JavaScript, based upon ECMAScript<sup>6</sup>. AS 2.0 was introduced with the seventh version of Macromedia Flash. It matured dramatically since ActionScript 1.0 and now supports static typing, which is a huge step forward for the debugging process, and offers the most important features that make it an *object-oriented language*. For example, it now supports *classes, interfaces, inheritance* and *polymorphism* – aspects that reduced the redundancy of code and made PHYRE have a much clearer structure.

**PHP 5** *Hypertext Preprocessor*. Handles the communication between application front-end and back-end. The Flash Remoting service, mentioned in section 8.1, is a PHP class that contains all the methods to handle access to the MySQL database.

**HTML** *Hypertext Markup Language*. The WWW is made up of web sites formatted in HTML. In an RIA the rich client takes over the role of the presentation tier, which is discussed in section 5.4. Therefore HTML was used only in the deployment of PHYRE. The page in which the application is embedded is formatted with HTML.

**SQL** The *Structured Query Language* (pronounced *sequel*) is used to fetch, insert, update, and delete data from the catalog database.

A very positive side-effect of running PHYRE in the Flash Player is its platform independent *look & feel*. This is a huge advantage over RICH INTERNET APPLICATIONS using DHTML<sup>7</sup> where cross-browser compatibility is very hard to achieve. Usually, the cost of using non-standard technologies is their limited availability. In the case of the Flash Player, this

---

<sup>4</sup>Some of these are not considered *real* programming languages, the reason why the term *computer languages* is used.

<sup>5</sup>Flash files (\*.swf) are called *movies* because of the origin of Flash as vector animation tool and the timeline based development approach.

<sup>6</sup>"ECMA International is an industry association founded in 1961 and dedicated to the standardization of Information and Communication Technology (ICT) Systems." [7]

<sup>7</sup>Dynamic HTML is an integration of JavaScript, Cascading Style Sheets (CSS), and the Document Object Model (DOM). It is used to enable more interactivity in otherwise static HTML pages.

aspect is not very disconcerting because its market penetration, as of September 2004, is over 80% for version 7 and for older versions even as high as 98.2% [8]. This makes it the world's most pervasive software platform. Furthermore, the browser plug-in is available for all major operating systems such as Microsoft Windows, Mac OS, Linux, and Solaris. From the server's point of view, PHYRE needs to run on a machine which supports PHP and MySQL. However with its tiered architecture, which is discussed on page 16, it could be ported to other systems with a comparatively small effort.

#### 2.2.4 User Requirements

After having tested PHYRE on different machines and browsers, the following requirements were set:

##### Minimum requirements

- Flash Player 7.0.14
- 1024 x 768 or higher screen resolution
- Pentium III 1 GHz or equivalent

##### Recommended requirements

- Flash Player 7.0.14 or higher
- 1280 x 1024 or higher screen resolution
- Pentium IV 2 GHz or higher

These requirements are purposely set this high, due to the fact that Flash Player 7 is not very fast at executing large amounts of code, which RIA like PHYRE contain. First insights into the next Flash Player generation, code-named *Maelstrom*, look very promising, although the following statements are yet to be confirmed by the actual release and independent tests of the next generation Flash Player.

*Maelstrom's* performance when executing ActionScript was increased dramatically, as well as its performance during the animation of complex visual arrangements. The latter was achieved through a technique called *bitmap caching*, which allows the next Flash Player to render unchanging vectors as bitmaps [9]. A demonstration of *Maelstrom* at a major conference showed a framerate increase of *over 800%* in one case [10].

## 3 Technologies

### 3.1 Introduction

One of the goals while planning PHYRE was to use free, open source, and community-supported software as an alternative to commercial solutions whenever possible. The first reason is, that a student's budget for software licenses is limited. The second reason is, to show what possibilities there are to build RIA with no need for expensive, proprietary technologies, if one has the necessary know-how.

Obviously, it is not always possible to find an open source and free equivalent for a certain technology, that also satisfies the requirements. While planning PHYRE, this became clear in the case of Flash. At the time of writing, there simply is no viable alternative to build sophisticated user interfaces. Since I have used Flash for many years I know its strengths and also its weaknesses, but I also know that it can live up to the expectations and meet the requirements better than any other solution known to me. All other components of the system are open source and free, starting with the database, which runs with MySQL, AMFPHP is used as connector between database and user interface, while the whole application runs on an Apache server.

The next sections give a deeper insight into these technologies.

### 3.2 Macromedia Flash

At the time of writing, the current version is Flash MX 2004, which is the seventh version in the line of the software. Macromedia Flash has come a long way from just a vector animation tool to a sophisticated and powerful tool for application developers as well as designers. Macromedia is targeting the market of self-contained web applications that enrich the user experience by using Flash to create more dynamic, flexible and intuitive user interfaces than other web technologies like HTML would allow.

### 3.3 AMFPHP

A database driven application needs a way to exchange data between database and front-end as efficiently as possible. AMFPHP is a technology written in PHP<sup>8</sup> to emulate the capabilities of a software originally developed by Macromedia, called Flash Remoting. Flash Remoting, unveiled in 2001, is a proprietary, highly efficient technology to connect Flash applications to the server. The benefits when choosing Flash Remoting over other technologies are performance and seamless integration with Flash. It uses a binary message format, designed for the ActionScript object model, called

---

<sup>8</sup>Hypertext Preprocessor

Action Message Format (hence the AMF in AMFPHP). The usual way of exchanging data in Flash, is to send it through HTTP<sup>9</sup> using either of the two built-in functions **getURL** or **loadVariables**. Data exchange with those two functions does not allow the transmission of complex data types like arrays or objects and does not offer a suitable solution for debugging. With the advent of Flash 5, XML<sup>10</sup> became another option for developers to handle data exchange. The problem of complex data types was solved because XML allows data to be structured. Although XML solves many of the previous problems, data still has to be parsed and formatted in order to be sent and received. For little amount of data to be sent, XML needs a big amount of markup to structure a file. Handling large XML structures can fail because of the following two factors:

### **I Bandwidth**

Although the number of broadband connections is continuously growing, the markup to describe the structure of a large XML document can quickly cause undesirable latency for the user until it is loaded completely.

### **II Client-Side Performance**

One of Flash's downsides is the lack of support for specific hardware acceleration. Parsing large XML structures can cause the client to slow down or idle at best, or at worst, the client's CPU usage would skyrocket and eventually the client machine would crash.

To address those two issues, Macromedia released Flash Remoting. Because of its binary approach to data encoding, it allows complex data and custom objects to be transmitted very efficiently. One of the few reasons not to use Flash Remoting is its price tag. By propagating the technology by means of a lower price, hosting companies could have offered Flash Remoting together with even their lower-priced plans and made Flash Remoting the standard solution for data exchange with Flash. While Macromedia missed out on their big chance, some members of the open source community recognized the potential of Flash Remoting, hence reverse-engineered the Action Message Format and ported Flash Remoting to other languages than the already available versions for Java, .NET and ColdFusion. AMF-PHP was the first version of Flash Remoting released on a different platform than the three original ones; others followed. At the beginning, AMF-PHP was simply an experiment to make Flash Remoting run on PHP, but as soon as more developers saw that it was working rather well, even in the beta version, they started to jump on the bandwagon. After some hesitation because of legal issues concerning Macromedia's role as inventor of the

---

<sup>9</sup>Hypertext Transfer Protocol

<sup>10</sup>eXtensible Markup Language

Action Message Format, everything became clear when the developers of AMFPHP got in touch with the company and their efforts were endorsed.

Sometime later, Macromedia published an introductory article on leveraging Flash Remoting through AMFPHP in the developer's section of their web site, which marked the beginning of the legal coexistence of AMFPHP and Flash Remoting [11].

### **3.4 MySQL**

Since PHYRE is hosted on a Linux server running Apache and uses some sort of PHP technology (which AMFPHP is), running the database with MySQL was a logical consequence. *"MySQL has become the most popular open source database and the fastest growing database in the industry"*, its own web site claims [12]. MySQL is the standard database choice for non-commercial projects: It offers seamless integration with PHP, is very fast, and is supported by a large community.

## 4 Programming Techniques

### 4.1 Introduction

This section provides a short insight into the different programming techniques and concepts used in the development of PHYRE. Discussing all the details of the techniques mentioned here, would go beyond the scope of this paper. However, this section gives a short introduction to the most important concepts and techniques, while Appendix A provides additional resources which should answer further questions.

### 4.2 OOP: Object–Oriented Programming

While working on larger software projects, developers may encounter several problems. One of them is, that the dependency of inherently independent parts of a system increases as the system’s complexity grows, and therefore the flexibility of the overall system decreases. These circumstances benefit the introduction of bugs<sup>11</sup> into the application and make it harder to maintain and extend it. Object–oriented programming (OOP) is a concept which is well-known and established amongst developers who work with high level languages<sup>12</sup> such as Java, C++ or C# but fairly new to most Flash developers and the ActionScript language. Colin Moock, author of many acclaimed Flash books, describes OOP as *“a different approach to programming, intended to solve some of the development and maintenance problems commonly associated with large, procedural programs. OOP is designed to make complex applications more manageable by breaking them down into self-contained, interacting modules.”* [14].

Without the use of the OO programming concept I would not have been able to develop and test a RICH INTERNET APPLICATION all by myself and in such a short period of time.

---

<sup>11</sup>A programming error that causes a program to work poorly, produce incorrect results, or crash. A bug is different from a glitch, which refers to a hardware problem and not a software problem.

*Note: The term “bug” was coined when a real insect damaged one of the circuits of the first electronic digital computer, the ENIAC.* [13]

<sup>12</sup>A high-level programming language is a programming language that is more user-friendly, to some extent platform-independent, and abstract from low-level computer processor operations such as memory accesses. [18]

### 4.3 Design Patterns

*"Designing object-oriented software is hard, and designing reusable object-oriented software is even harder,"* is what the authors of *Design Patterns*, by the community fondly given the name of *Gang of Four*, observed rightfully [15]. With their seminal book named after their findings, the first time in history someone wrote down a collection of undocumented concepts of conversant software developers called design patterns.

Christopher Alexander, architect and father of the Pattern Language movement in computer science, says:

*"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"*<sup>13</sup> [16].

Today, design patterns play an important role in modern software architectures.

### 4.4 Refactoring

*"Refactoring is changing code structure without changing its function."* [17]

Martin Fowler was first to write a book about *refactoring*, namely *Refactoring – Improving the Design of Existing Code* [19]. The refactoring process helps to improve maintainability, flexibility, and readability of code. During the development of PHYRE I *refactored* many classes in order to add new functionality to them more easily. To fully understand and apply the concepts of refactoring, I warmly recommend Martin Fowler's book.

---

<sup>13</sup>found in the *Design Patterns* book [15]

## 5 Three-Tiered Application Architecture

### 5.1 Introduction

Applications today are typically designed in a *tiered* fashion. Each tier has a certain function. Web deployed applications are usually split up in three tiers. Figure 1 illustrates the structure of PHYRE's three-tiered application architecture:

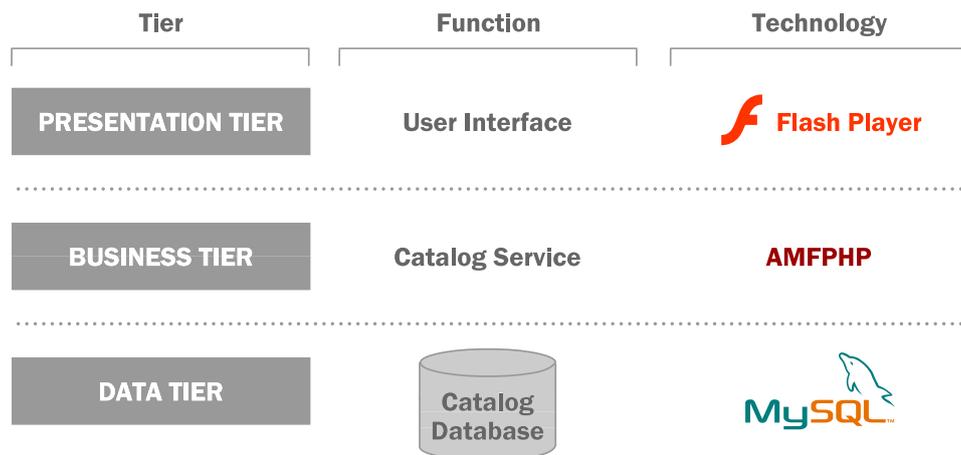


Figure 1: PHYRE's Three-tiered Application Architecture

### 5.2 Data Tier

The *data tier*<sup>14</sup> is responsible for storing all the data the application needs to access. In the case of PHYRE, the data tier consists of a single MySQL database, in other cases it could also be a web-service or any other data source. PHYRE's database stores all the catalog's data and organizes it in a meaningful way. The database's design is defined by the entity-relationship model, which is discussed in section 6.

### 5.3 Business Tier

The *business tier*'s<sup>15</sup> duty is to give the client access to the data tier in order to enable it to retrieve, update and insert data. Although data's integrity and validity is checked in all tiers, it is done mainly in the business tier. Data is received from different places but in order to get stored in the data tier it must pass the business tier, so validation is best done there. For example,

<sup>14</sup>In other environments known as *integration tier*.

<sup>15</sup>Also referred to as *middle tier*.

before an application stores certain data it must make sure its format is correct, such as date formats, which get to the business tier in all kind of forms. Business rules are applied in the business tier, such as the fact that an experiment belongs to not more than one category and a certain storage location cannot exist more than once.

#### **5.4 Presentation Tier**

PHYRE's user interface is located in the *presentation tier*<sup>16</sup>. This tier plays a vital role because it makes the connection between the user and the underlying application logic. It must guide the user to the information she or he needs and present it in a concise form.

#### **5.5 Benefits**

The benefits of structuring an application in tiers becomes obvious when underlying technologies have to be changed. Imagine the application needs to move its database to a different server, and therefore has to use another vendor's database product. If data storage was contained in the rest of the application, the whole application would have to be adjusted. This process costs valuable time and possibly breaks the application. Because the application is designed in tiers, all that has to be done is an adjustment to the business tier in order to support the new database's specific functionality—no changes whatsoever have to be done to the presentation layer because it is independent from the data tier and communicates with it solely through the business tier.

---

<sup>16</sup>Also known as *client tier*.

## Part II

# Building PHYRE

With PHYRE, I built my first RICH INTERNET APPLICATION and therefore took it as an opportunity to experiment with different development techniques and to learn about the process behind building an RIA. This part of the paper presents the most interesting problems I encountered (also the ones that caused the biggest headaches) and the solutions I come up with. Read through it and take it as an initiative to look further into PHYRE's architecture and to learn from it.

## 6 PHYRE's Catalog Database

*"The first step in creating and using a database is to establish its structure" [20].*

This section provides an insight into design considerations during the modeling process and the necessary basics to analyze and comprehend PHYRE's underlying database model. The following steps refer to the accompanying database diagram, which is called *entity-relationship model* (ERM).

### 6.1 The Entity-Relationship Model

The process of modeling a database is based upon the *functional requirements* and *business rules* which were earlier agreed on (Section 2.2.2). Either with paper and pen or with a database modeling tool the catalog's different tables and the relationships between them are laid out.

First, I started out with a *categories* table which stores the catalog's categories. Determining the attributes that define a category is an essential step. I came up with the following attributes which are followed by their respective data types<sup>17</sup> written in capital letters:

**category\_id INTEGER**

The *primary key* (PK) of the *categories* table. It gives every category a unique identity. A certain PK does not exist more than once in a given table, even after deleting and inserting new data records.

**name VARCHAR**

The name of the category that will be used for display.

**description TEXT**

Explains the category's purpose closer. By default, the description is not displayed in the application.

---

<sup>17</sup>The article *"MySQL Database Design"* lists MySQL's supported *data types* [20].

**sortorder SMALLINT**

Defines the order of appearance of a certain category. Categories are displayed in alphabetical order if this attribute is missing.

Looking at the ERM, you can see a few more attributes that the *categories* table has, such as **modified\_by\_user\_id**, **created\_by\_user\_id**, **date\_modified**, and **date\_created**. While examining other database tables, you will encounter these attributes again. Their purpose is to *internally* keep track of changes done to the data in the catalog database. When a new category is added to the catalog, its date of creation is saved, as well as the ID of the user who added it. Likewise when a category's attributes are changed, its modification date and again the ID of the user who modified it are saved. These attributes enable the user to not only sort experiments by their name but also by their dates of creation or modification. This way, a user can quickly find the oldest experiment of a certain category or the most recently updated one.

A special attribute of the *categories* table is **parent\_category\_id**, to which section 7 is dedicated.

## 6.2 Relationships & Business Rules

The *categories* and consequently the first table is now completely set up. As from now, PHYRE can store its own set of categories. Allowing PHYRE to store experiments and defining their affiliation to a specific category is the next step: PHYRE gets an *experiments* table with the following attributes:

**experiment\_id INTEGER**

PK and an experiments unique identifier.

**category\_id INTEGER**

Foreign Key (FK) of the *categories* table. Used to link an experiment to a certain category. For further explanation please read on.

**name VARCHAR**

The name of the experiment that will be used for display.

**description TEXT**

The description can be used to specify the necessary settings that have to be adjusted prior to the demonstration of the experiment. Otherwise, it is also used to refer to illustrations or photos that show how to set up and carry out the experiment.

**quantity SMALLINT**

The total quantity of the experiment. For future uses like the integration of a check-out system for experiments.

**quantity\_available SMALLINT**

The currently available quantity of the experiment. For future use only (see **quantity** attribute.)

**note TEXT**

Further notes for the experiment that are not displayed in the application.

**metainfo TEXT**

Comma separated list of keywords: used by the full-text search module.

All other attributes of the *experiments* table, as seen on the ERM, like **date\_modified**, **date\_created** etc., are explained in section 6.1.

According to *Business Rule IV*, experiments are assigned to at least but not more than one category:

*"How do I apply Business Rule IV in terms of database modeling?"*

In order to apply *Business Rule IV*, I set up a relationship between the *experiments* and the *categories* table. Table 1 shows the three different types of relationships that exist in RDBMS<sup>18</sup>:

**one-to-one relationship (1:1)**

is probably the least common of the three, where a primary key value matches only one (or no) record. These relationships are almost always forced by business rules and seldom flow naturally from the actual data.

**one-to-many relationship (1:n)**

is the most common relationship, in which the primary key value matches none, one, or many records in a related table.

**many-to-many relationship (n:m)**

occurs when both tables contain records that are related to more than one record. [MySQL] doesn't directly support a many-to-many relationship, so you must create a third table: an *associate* [or *assignment*] table. It contains a primary key and a foreign key to each of the data tables. After breaking down the many-to-many relationship, you have two one-to-many relationships between the associate [or assignment] table and the two data tables.

Table 1: Overview: Relationship Types [21].

Since each experiment belongs to one category but each category can hold several experiments, the needed relationship between these two tables is a one-to-many relationship (1:n). After having figured out the relationship I need, I add the attribute **category\_id** to the *experiments* table. When PHYRE's catalog accommodates a new experiment, it inserts the primary key of the category it belongs to into the **category\_id** field.

---

<sup>18</sup>Relational Database Management Systems

This way, an experiment knows to which category it belongs. On the other hand, the *categories* table has no reference to any experiments whatsoever, since such a redundancy should be avoided whenever possible. Hence the code needed to fetch all experiments that belong to a certain category is fairly simple and looks like this:

```

1 function getExperimentsByCategoryID($p_id) {
2   $result = mysql_query("SELECT experiments.* FROM experiments
3     WHERE experiments.category_id = ". $p_id);
4   return $result;
5 }

```

It is important to know how the technical denotation of relationships in ER diagrams, for example in the one that comes with this paper, looks like. There are a number of notations used: Figure 2 shows the three different relationships I discussed with their respective symbols in *Crow's Foot* notation:

### "Crow's Foot" Notation of Relationships in ERM

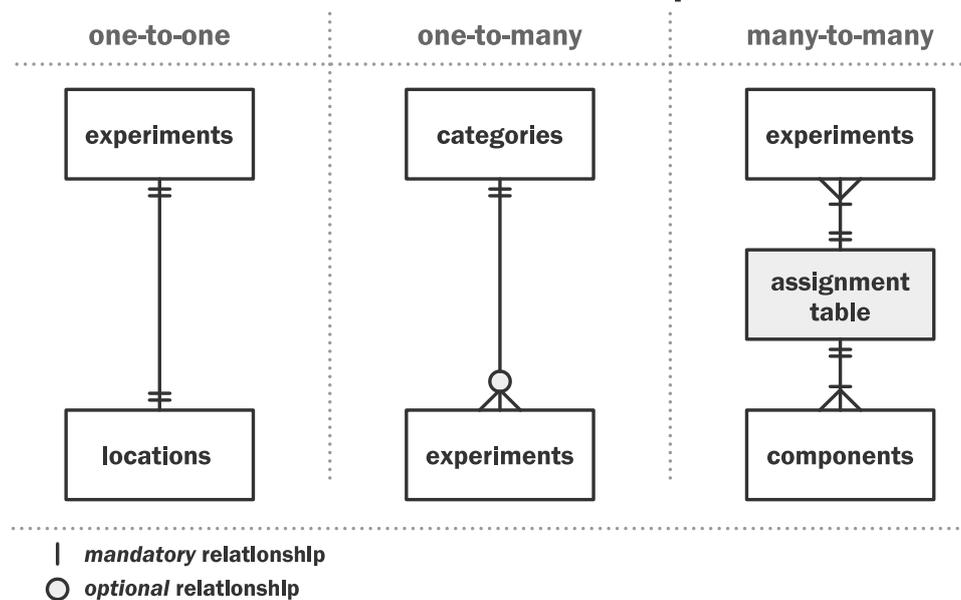


Figure 2: Relationships in *Crow's Foot* Notation

An example for a many-to-many relationship is the linkage of experiments and components, since one experiment is made up of different components, and at the same time, a certain component can be part of many experiments (Business Rule VI & VII). In MySQL, this relationship is set up by using an assignment table, for example *experiments\_components\_assign* in PHYRE's ERM, which actually breaks the many-to-many relationship into two separate one-to-many relationships. For an explanation of this kind of relationship, please refer to Table 1 on page 21.

Many-to-many relationships are handled similarly to the other two relationships. The only thing that changes, is the code to query the database. For example, in order to get all components that are used in an experiment, you have to use an SQL **JOIN** statement (line 2–6), which is not explained in detail, so please refer to Appendix A for additional resources.

Enough said, the following is the function to fetch all components which are used in a certain experiment:

---

#### AMFPHP Service — **ch.gasi.phyre.Catalog**

---

```
1 function getComponentsByExperimentID($p_id) {
2   $result = mysql_query("SELECT c.* FROM components c
3     LEFT JOIN experiments_components_assign
4     eca USING (component_id) LEFT JOIN
5     experiments e USING (experiment_id)
6     WHERE e.experiment_id = ". $p_id);
7   $i = 0;
8
9   while($component = mysql_fetch_object($result)) {
10    $components[$i] = $component;
11    $location_id = $component->location_id;
12    $components[$i]->location =
13      $this->getLocationPath($location_id);
14    $i++;
15  }
16  return $components;
17 }
```

---

With the information about databases and data relationships given in this section, you should be sufficiently savvy to explore PHYRE's data tier and its ERM on your own. Recall PHYRE's *Business Rules* from section 2.2.2 and try to find out how they are implemented by looking at the entity-relationship model. The next section is dedicated to the special treatment of hierarchical data in an application.

## 7 Hierarchical Data & Recursion

After having understood *Business Rule II*, which states that categories can be nested, the following question probably comes up:

*“How do I store hierarchical data, such as nested categories, in a RDBMS<sup>19</sup> like MySQL?”*

The solution is really straight-forward: I add a new attribute to the *categories* table called **parent\_category\_id** in which I can save a category’s reference to its parent category. Categories at the top of the hierarchy simply have a default value of 0 in their **parent\_category\_id** field.

Retrieving the categories from the database is done quickly, but parsing it into a hierarchical tree for display in the presentation tier is a little trickier. At first, the problem appears to be easy to solve: the categories are fetched from the catalog database and with a loop they are checked for possible sub-categories that belong to them. After that, these categories are nested appropriately. Until here, everything seems fine, but with this approach you only get the first and the second level of the hierarchy. No problem, the algorithm gets another loop to check sub-categories for having sub-sub-categories. Now you get three levels deep into the hierarchy. Then you start asking yourself:

*“Does that mean that for every level I want to get deeper into the hierarchy I need another loop with the exact same code that checks for child categories?”*

Something cannot be right—you look at your code and you know: *This code smells*<sup>20</sup>. The method for solving such quite frequently occurring problems is *recursion*. In terms of programming, a function that calls itself during its execution is called a *recursive* function. Applying recursion to our previous problem, the following changes: after checking a category for children we do not write another loop to fetch those children and again check those for child categories themselves, instead the function calls itself again to perform this very task. If the condition returns false, the function simply takes the next category and performs the same tasks until it is completely through the whole hierarchy.

Page 26 shows an excerpt from the code of PHYRE’s AMFPHP service. The **buildTree** method (line 21–34) is a recursive function. After having fetched the categories from the database, it builds an XML tree; XML is the format that Flash’s Tree component needs to display the hierarchical data. The **if** construct on line 30 checks for categories that have the current

---

<sup>19</sup>Relational Database Management System

<sup>20</sup>Amusing expression used in Martin Fowler’s book *Refactoring* [19].

category as parent. If it finds any, the function calls itself (line 31) in order to start the same procedure over again.

In retrospect, the **buildTree** method turned out to be small and simple. Still, I spent a lot of time getting this little function to run. First, the nesting of the categories XML file was incomplete since I worked with a procedural function before using the described recursive approach.

After I rewrote it to a recursive function, the function's call to itself failed over and over again. As usually, it turned out to be a simple problem which was related to my just freshly acquired, and therefore limited PHP knowledge. The catalog service is a PHP class, and therefore each function, according to object-orientation, is a method of this class. Contrary to ActionScript 2.0, where it is optional, PHP insists on the **\$this** keyword (line 14 & 31) in order to call a method of a class within itself.

The previous little problem is a typical example for the fact that no matter how many times you read the documentation, there is no guarantee that you actually know its contents until you use part of it, and preferably fail while doing so. From that point on, you will never do it wrong again, guaranteed.

```
1 function getCategories() {
2   $doc = new DomDocument();
3
4   // Document properties
5   $doc->encoding = "UTF-8";
6   $doc->standalone = true;
7
8   // Create an empty element
9   $root = $doc->createElement("categories");
10
11  // Append root element
12  $doc->appendChild($root);
13
14  $this->buildTree($root, 0, $doc);
15  $output = $doc->saveXML();
16
17  // Retrieve and print the document
18  return $output;
19 }
20
21 function buildTree($p_node, $p_id, $p_doc) {
22   $result = mysql_query("SELECT categories.* FROM categories
23     WHERE categories.parent_category_id = ". $p_id);
24   while($row = mysql_fetch_object($result)) {
25     $element = $p_doc->createElement("item");
26     $element->setAttribute("label", $row->name);
27     $element->setAttribute("id", $row->category_id);
28     $p_node->appendChild($element);
29
30     if (mysql_num_rows($result)) {
31       $this->buildTree($element, $row->category_id, $p_doc);
32     }
33   }
34 }
```

Figure 3 shows the XML structure which is returned by the **buildTree** function discussed before, on top and the visual representation in the user interface on bottom:

## Categories XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<categories>
  <item label="Elektrische Ströme" id="4"/>
  <item label="Magnetismus" id="5">
    <item label="Magnetostatik" id="41"/>
    <item label="Magnetfelder el. Ströme" id="42"/>
    <item label="Lorentzkraft: Kraft auf geladene Teilchen" id="43"/>
  </item>
  <item label="Geometrische Optik" id="6"/>
  <item label="Schwingungen" id="7">
    <item label="Schwingungen allgemein" id="40"/>
  </item>
  <item label="Wellen" id="8"/>
  <item label="Atom- und Quantenphysik" id="9"/>
  <item label="Kernphysik" id="10"/>
  <item label="Relativität" id="11"/>
</categories>
```

---

## Categories Tree View

- 📁 Elektrische Ströme
- ▼ 📁 Magnetismus
  - 📁 Magnetostatik
  - 📁 Magnetfelder el. Ströme
  - 📁 Lorentzkraft: Kraft auf geladene Teilchen
- 📁 Geometrische Optik
- ▼ 📁 Schwingungen
  - 📁 Schwingungen allgemein
- 📁 Wellen
- 📁 Atom- und Quantenphysik
- 📁 Kernphysik
- 📁 Relativität

Figure 3: XML and Tree View of PHYRE's categories

## 8 User Authentication & Access Control

### 8.1 Introduction

PHYRE is not only built to enable users to view information that resides inside the catalog, it is also supposed to give users a way to modify or insert new data into the database.

*How can users modify the underlying data of PHYRE's catalog?*

I integrated an administration module directly into PHYRE's user interface. Again, as you can see in the case of PHYRE, this is a big advantage of RICH INTERNET APPLICATIONS over conventional web applications, since you do not have to create a separate page for the purpose of administration. Obviously within the RIA, the administration is a separate module, although user can access it without having to leave the main application.

This section is concerned with two topics, namely *user authentication & access control*: To ensure a system's underlying information's integrity and veracity you have to be able to control which users get access to it and what rights are given to them.

Users of PHYRE that are authorized to insert or modify information, namely physics teachers and perhaps other school personnel, have to get a user account that will be stored in the database. After being registered, a user can log himself into the application by entering his unique user name and password. During the authentication process the application retrieves the user's login data from a login form in the user interface and verifies it against the data stored in the database.

### 8.2 Sensitive Data & MD5

*What are MD5 and secure hashes and how are they relevant to a system's user authentication mechanisms?*

If you are familiar with the answers to this question please go ahead and carry on reading on page 30. If you have not previously dealt with user authentication or software security in general, it is helpful to read the following introduction to MD5:

First, let me tell why you should care about your applications security: In my opinion, which is shared by many other application developers, a well thought-out system should never store or transmit its sensitive data as plain text. Storing a user's password as plain text means that it can be read easily by anybody with access to the database, including yourself, which is a clear violation of the user's privacy.

You might wonder what other choice you have, since a stored password has to be compared to the password the user enters every time she or he

wants to log into your application, and therefore has to be available as plain text.

There is an option, and this is where the Message-Digest algorithm 5 (MD5), designed by Ronald Rivest in 1991, comes into play. For a discussion of the inner workings of MD5 please refer to the sources listed in Appendix A; this paper discusses only the aspects of MD5 that are relevant to this section's topic.

MD5 provides exactly the kind of functionality we need to protect sensitive data while still being able to work with it. Contrary to general perception, MD5 is not an encryption algorithm but rather a one-way hash function which, in simple terms, calculates a fixed-length, unique string, called secure hash, from any given input. This definition of MD5 should answer the often posed question of its reversibility: MD5 is not reversible (unless you have a hundred computers working on it day and night!) That is, it is close to impossible to find the originally entered value if you only have its corresponding MD5 hash.

Probably many of us know the situation where you forget the password to a service like an e-mail or a shopping account. Often times, there is an option to let the provider of the service send you an e-mail with a new password. Maybe, or maybe not, you have wondered why some providers send new passwords instead of the original ones. If you receive a new password, it is very likely that the provider did not even store your actual password in the first place, but only its unique hash, which was computed by MD5 or a similar algorithm. That is, next time you get a new password, be thankful, since someone decided to respect your privacy!

## 8.3 Implementation

PHYRE's implementation of the authentication mechanism is described on the following pages. Please read the steps carefully and refer to the actual code in PHYRE whenever needed.

### 8.3.1 Prerequisites

The database has a *users* table where all the authorized users of the RIA are stored. The three essential attributes for the authentication mechanism are:

**user\_id INTEGER**

The *primary key* (PK) of the *user* table. It assigns a unique ID number to every user.

**username VARCHAR**

Every user has a user name that is unique across the system, which ensures a smooth authentication process.

**userpass VARCHAR**

A user's personal password, which is not stored as plain text but rather as a secure hash calculated by the previously discussed MD5 algorithm.

Before a user can log into the RICH INTERNET APPLICATION, she or he has to have a user account. When a new user account is created, its corresponding password is stored as an MD5 hash in the *users* table.

### 8.3.2 Login

Having set up the *users* table as a necessary prerequisite for PHYRE's user authentication mechanism, let us look at what happens behind the scenes when a user wants to log into the application:

*Note: At the time of writing, AMFPHP 1.0 was not available, although close to release. Therefore the code examples and the process of user authentication refer to the beta version AMFPHP 0.9.0b which is used in PHYRE.*

- Each time a user tries to enter PHYRE's restricted area but has not been authenticated yet, the application shows a window with a login form that asks the user to enter his user name and password as seen in Figure 4.

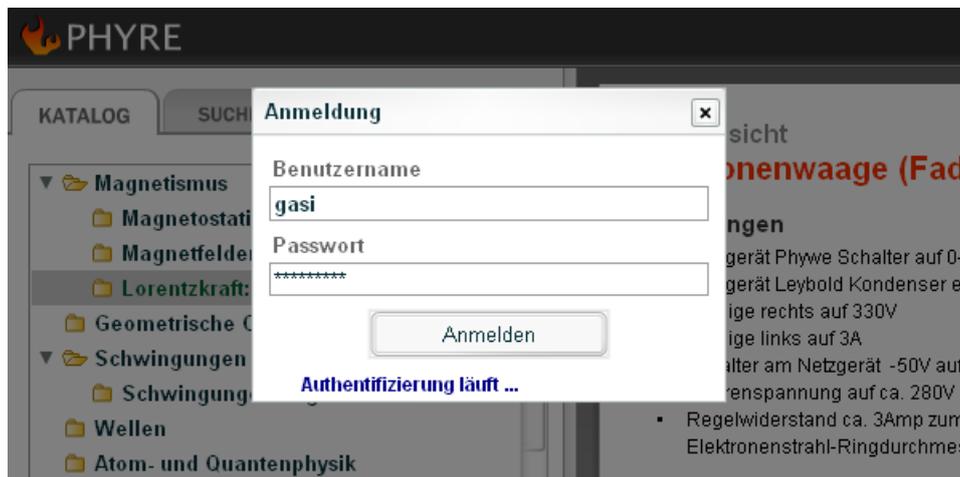


Figure 4: Login screen in PHYRE

- When the user presses the login button, his user name and password are passed on as parameters to the **doLogin** method of the **Catalog** class (line 1 & 2), whose code is listed on page 38.
- The **doLogin** method takes the user's password and calculates a secure hash using the MD5 algorithm (line 6).
- On the client, the user name and password are passed on to the **setCredentials** method of the **Service** class (line 8). The **Service** class is specific to Flash Remoting, and is responsible for creating a connection to the AMFPHP service on the server-side. Its **setCredentials** method is used to set a user name and password for a certain connection. Figure 5 shows how a **setCredentials** method call causes Flash

Remoting to add a credentials header to the connection. This allows authorized users to invoke actions, for example inserting new data into the catalog, which are not granted to unauthorized users.

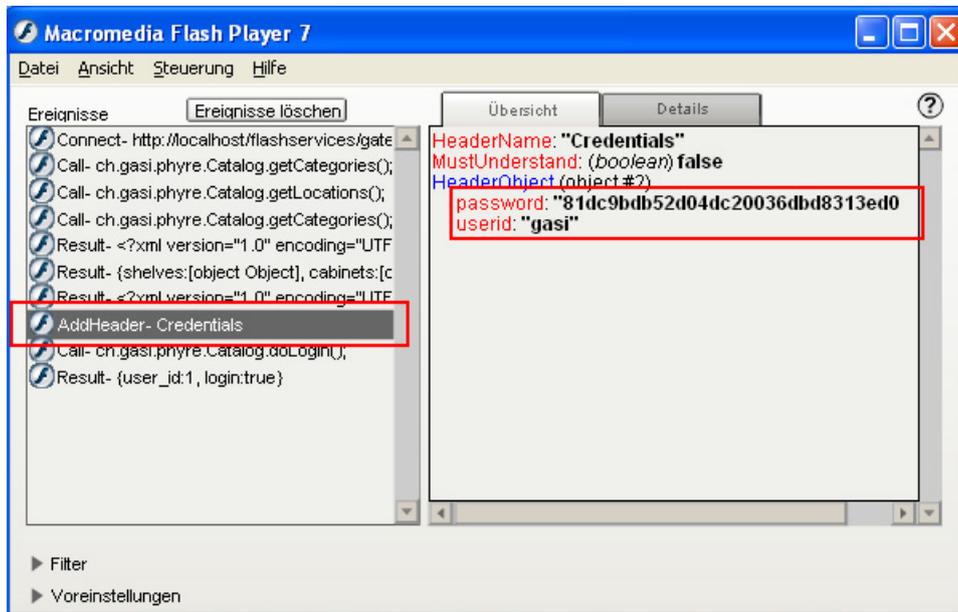


Figure 5: The effects of the `setCredentials` method call as seen in the NetConnection Debugger, the debugging tool for Flash Remoting connections.

- Without going into much detail, let us take a look on how certain methods of PHYRE's AMFPHP service are secured for access to authorized users only:

```

1 "addExperiment" => array(
2     "description" => "Adds a new experiment
3         to the catalog",
4     "access" => "remote",
5     "roles" => "admin",
6     "arguments" => array(     "p_category_id",
7                                 "p_user_id",
8                                 "p_name",
9                                 "p_description",
10                                "p_metainfo" )
11 )

```

This is an excerpt from the catalog service's method table which lists all its methods and properties. Because the `addExperiment` method is

used to add new experiments to the catalog, it is definitely one of the methods that should be protected against access from unauthorized users.

Besides a general **description** (line 2, page 32) of the method, which can be helpful in the debugging process, the method table defines its **access** property (line 4), which has to be set to **remote** in order to be accessed from a client, and a list of arguments it takes (line 6–10). From a security’s point of view, the most important attribute it defines, is the **roles** attribute (line 5). The **roles** attribute, as seen in the complete source of the catalog service, is only defined for methods with certain access restrictions. In this case, the **roles** attribute has the value **admin**, which means, that only users of the group **admin** can access this method. This is how we can prevent abuse through unauthorized users.

- Going back to the situation where the **doLogin** method, which is listed on page 38, is called on the client, you maybe noticed that the method not only calls **setCredentials** on the connection (line 8) but also **doLogin** on the service itself (line 11).

The reason for the latter call is, that after setting the credentials for a connection, you have to call a method on the service that is accessible only if the client is authorized, in order to check if the authentication of the user succeeded. When the secured **doLogin** method is called, it processes the information of the connection’s credentials header in the service’s **\_authenticate** method.

The **\_authenticate** method is a function you have to provide in your service in order to allow AMFPHP to internally authenticate users. A listing of it is found on page 39.

- The **\_authenticate** method takes the user name and password from the credentials header and queries the database (line 9–13) for such a user. If the user exists and its password’s hash matches the one stored in the database (line 15), the method stores the user’s ID in a session, so that it can be retrieved by other methods of the class at any time later<sup>21</sup>, and returns the group the user belongs to. If no such user was found, or any other problem occurred during the process, **\_authenticate** returns **false**.

---

<sup>21</sup>The workaround with the session variable is needed because PHP is stateless.

*Note: Maybe you noticed the line in the `_authenticate` method on page 39 that is commented out. Why was it written in the first place if its not used anymore?*

*Because Flash does not feature a built-in MD5 function, PHYRE calculated the hash in the business tier with the MD5 function provided by PHP (line 8). This approach did not satisfy me because passwords were being transmitted from the client to the server unprotected. After some searching, I eventually found an ActionScript 2.0 class that provides MD5 functionality. Having implemented it on the client, passwords are now safe as soon as the user presses the login button and do not have to be processed on the server. Therefore the commented out line (line 8), which calculated the password's hash, is not needed anymore and left in the source only for the sake of this note.*

- If the user has been authenticated, she or he now belongs to the **admin** group. Recall how we previously set the **roles** attribute for the **doLogin** method and defined that the only group allowed to execute it, is the **admin** group (line 5, page 32).

At this point, **doLogin** returns the user' ID, which is retrieved from the session variable (line 26, page 39) previously set by the **\_authenticate** method, as well as a status variable (line 25) that verifies that the user is authorized to make changes to the catalog. Being authenticated, the user can now access the administration PHYRE' module (Figure 6).



Figure 6: PHYRE's Administration Module

- If the system doesn't know a user with the supplied user data, the `doLogin` method throws an error that is caught on the client and displayed as seen in Figure 7.

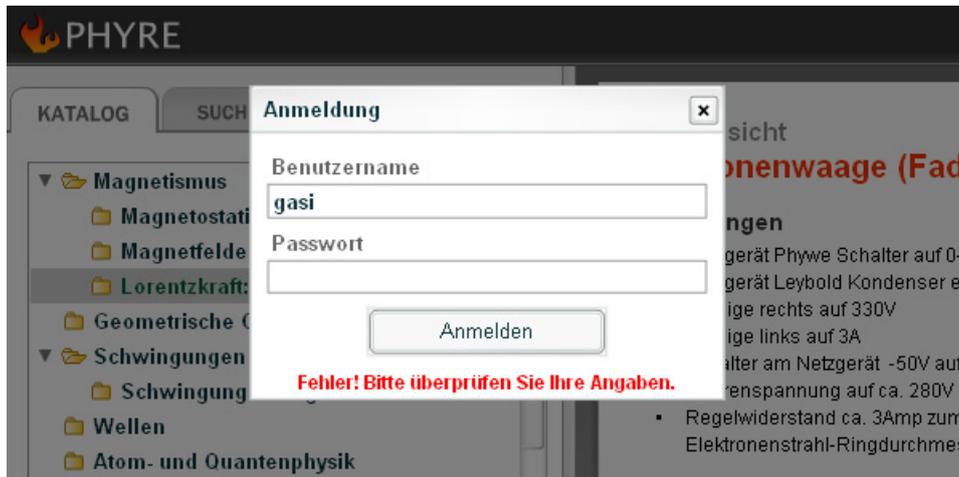


Figure 7: Error message displayed after a failed authentication attempt.

### 8.3.3 Logout

Often times not mentioned, but for the sake of completeness, here is a description of how to implement a logout mechanism in an application. After having read the previous pages you should understand how the login mechanism works. The logout mechanism actually works similarly: it resets the connection's credentials header by calling `setCredentials` with two empty strings as arguments (line 24, page 38). At the same time, it calls the `doLogout` method on the service which logs out the user on the server (line 26). The server-side logout call is found in line 32, on page 39, in the `doLogout` method.

## 8.4 Conclusions

I hope this sections showed that security in RICH INTERNET APPLICATIONS deserves more attention as more and more users will use such applications in the future. As soon as sensitive data like credit card information and social security numbers are being processed in an RIA, it should implement a well thought-out security concept that makes sure that the data is safe from third at any time.

This section outlined the pillars of such a security concept. PHYRE's implementation can be used as an example for basic user authentication and access control mechanisms.

Having come this far, it is easy to extend the system, for example to support user groups that grant certain rights only to members of a certain group. Consider having a `superadmin` group whose members can create new user accounts or delete existing ones. I am sure you can think of even more possibilities ...

---

*Important Note: The herein described approach is not meant to be the ultimate security solution for an RIA but merely makes sure that sensitive data like passwords are not revealed as plain text. Passwords and their respective hashes can still be intercepted during transmission and possibly be used to make an application authenticate even unauthorized users. In order to better secure your application, please consider using a Secure Socket Layer (SSL) connection for the transmission of sensitive data.*

## 8.5 Code Listings

### ActionScript 2.0 — ch.gasi.phyre.Catalog

---

```
1 public function doLogin(p_username:String,  
2                          p_userpass:String) {  
3  
4     trace("#Catalog# doLogin()");  
5     var m_username:String = p_username;  
6     var m_userpass:String = MD5.calculate(p_userpass);  
7     var conn = m_catalogService.connection;  
8     conn.setCredentials(p_username,  
9                        p_userpass);  
10  
11     m_pendingLogin = m_catalogService.doLogin();  
12     m_pendingLogin.responder =  
13         new RelayResponder( this,  
14                             "onLoginSuccess",  
15                             "onLoginFailed");  
16 }  
17  
18  
19 public function doLogout(Void) {  
20  
21     trace("#Catalog# doLogout()");  
22  
23     var conn = m_catalogService.connection;  
24     conn.setCredentials("", "");  
25  
26     m_pendingLogout = m_catalogService.doLogout();  
27     m_pendingLogout.responder =  
28         new RelayResponder( this,  
29                             "onLogoutSuccess",  
30                             "onLogoutFailed");  
31 }
```

```
1 /*
2 * This function will authenticate the client
3 * before it returns the value of method call
4 */
5 function _authenticate($p_username, $p_userpass){
6     $m_username = trim($p_username);
7     $m_userpass = trim($p_userpass);
8     //$m_userpass = md5(trim($p_userpass));
9     $m_query = mysql_query("SELECT users.* FROM users
10                             WHERE users.username
11                             = '$m_username' AND
12                             users.userpass =
13                             '$m_userpass'");
14
15     if(mysql_num_rows($m_query) == 1) {
16         $user = mysql_fetch_object($m_query);
17         $_SESSION["user_id"] = $user->user_id;
18         return "admin";
19     } else {
20         return false;
21     }
22 }
23
24 function doLogin() {
25     $result["login"] = true;
26     $result["user_id"] = $_SESSION["user_id"];
27
28     return $result;
29 }
30
31 function doLogout() {
32     Authenticate::logout();
33     return true;
34 }
```

## 9 Design Patterns in PHYRE

This section discusses where design patterns are found in PHYRE and shortly presents the underlying thoughts and reasons for their use.

Because this section's topic has already filled many books, this paper cannot provide the theoretical background which is probably needed to understand the following situations in which design patterns are applied. If you want, or need to know more about design patterns than discussed in this paper, please refer to Appendix A for additional resources.

While reading the following cases, it might be helpful to use the architecture diagram of PHYRE, which comes with this paper, as a visual aid. Also remember to take a look at the source code from time to time to see the actual implementation of the presented patterns.

### 9.1 Singleton

The intent of the Singleton pattern as described in the design patterns catalog:

*Ensure a class only has one instance,  
and provide a global point of access to it. [15]*

Besides the fact that it is generally known that the overuse of singletons is bad, this pattern turned out to be helpful in the case of PHYRE [22].

During the development of PHYRE, it came out that certain classes need a global point of access to its main object, namely the **Application** object. The **Application** object is instantiated at the very beginning and is responsible for creating all other objects, for example visual objects like the components of the user interface as well as non-visual objects like the catalog object.

Since the **Application** class features the following two characteristics, which are also described in the pattern's intent, I decided to implement it as a singleton:

#### Single Instance

The **Application** object starts the whole application, and therefore should only be instantiated once at the beginning. Multiple instances of the **Application** object would possibly interfere with each other and break the application.

#### Global Point of Access

Other objects in PHYRE should be able to access the **Application** object from any point in the application.

The characteristics of a singleton's implementation are seen in the **Application** class source: the constructor is declared **private** and the **getInstance** method provides the only access to the object.

## 9.2 Proxy

Computer users are used to elastic interfaces<sup>22</sup> from their desktop software and the web sites they are browsing. Flash has always supported lossless scaling of its movies because they are, similarly to SVG<sup>23</sup>, made up of vectors and not pixels, whereas the latter cannot be scaled without a visible loss of quality. But only since its sixth version, Flash MX, it also offers the possibility to make Flash movies behave just like web sites or desktop applications in terms of elastic interfaces.

Although I could have dedicated a whole section to the particularly interesting topic of elastic interfaces, this little introduction was necessary because the pattern discussed in this section is closely tied to the implementation of PHYRE's elastic user interface.

In order to react to changes to the size of a movie, developers have to make use of Flash's built-in **Stage** class.

The problem with the **Stage** class is, that it does not provide all the functionality often needed for RICH INTERNET APPLICATIONS. For example, I generally do not want an application to resize even further when the browser window becomes too small because its layout will be so dense that it would not be usable anymore. I also want to control this minimum size parameter whenever necessary.

Additionally, it is tedious to set certain parameters, like the scaling mode of the stage, every time I develop an application like PHYRE, which should not scale but rather feature an elastic interface.

In order to accomplish these goals, I first wrote complex algorithms in the **Application** class itself, which checked the size of the browser window and then reacted accordingly.

Feeling that the solution I came up with was not ideal, I looked through the design patterns catalog and found a better solution in the form of a pattern, namely the *Proxy* pattern.

---

<sup>22</sup>User interfaces that rearrange their layout, instead of being scaled, according to the dimensions of the window they run in.

<sup>23</sup>Scalable Vector Graphics

The intent of the Proxy pattern as found in the design patterns catalog:

*Provide a surrogate or placeholder for another object to control access to it. [15]*

I wrote a class, named **Canvas**, through which other objects in PHYRE could access all the information they usually retrieved from the built-in **Stage** class, such as the width of the movie, as well as make use of the extra functionality I described.

*What are the advantages of using a Proxy?*

One of the advantages of using a proxy is described using the **width** getter of the **Canvas** class as example:

```
1 public static function get width():Number {
2     var minWidth:Number = Canvas.getMinimumWidth();
3     var maxWidth:Number = Canvas.getMaximumWidth();
4     var w:Number;
5
6     if (Stage.width <= minWidth) {
7         w = minWidth;
8     } else if (Stage.width >= maxWidth){
9         w = maxWidth;
10    } else {
11        w = Stage.width;
12    }
13
14    return w;
15 }
```

As you can see, the **width** getter first retrieves the minimum and maximum widths (line 2 & 3) we want our application resize to and then checks if the actual stage width lies within these boundaries (line 6–9). If not, the function returns the previously defined maximum or minimum value.

Before using the Proxy pattern, I implemented a similar function in every object that had to react to changes of the stage's width. Here is where the principles of refactoring should be applied: If you write the exact same logic more than twice, something is not right and should be changed.

When I was in this situation, I wrote the **Canvas** class acting as proxy to the **Stage** object, deleted the redundant code that was scattered among several classes, and moved it into the **Canvas** class where it actually belongs to. *Why?* An object that needs to know the width of the stage should simply request it, without knowing if the returned value is the actual stage

width or the previously defined minimum width we want our application to have.

The default properties for applications featuring an elastic interface are set in the constructor of the **Canvas** class. This saves time and code for every application I build.

The preceding example shows how the process of applying a pattern generally looks like: You find yourself in a situation that is unsatisfying, one that might lead to a dead end in terms of maintainability and extensibility of your application, and you feel that there probably is a better solution for your problem. You look through a design patterns catalog and many times you find a pattern that helped other developers in similar situations, although it has to be adjusted in order to fit your needs.

This necessary adjustment is the reason why design patterns describe only *approaches* for solving often recurring problems and are *not actual implementations* that can be used out of the box.

### 9.3 Further Patterns Found in PHYRE

The following is a list of a few other useful design patterns which are found in the architecture of PHYRE:

**Command Pattern** This pattern is used to encapsulate requests as an object. PHYRE uses classes designed after this pattern to perform certain actions like logging out a user or switching views in the detail view. To see examples of this pattern, please look into the **ch.gasi.phyre.commands** package.

**Business Delegate** This pattern is used to to hide the implementation details of the business tier. PHYRE's **Catalog** class on the client-side, which is written in ActionScript 2.0, acts as a business delegate. It reduces coupling between the presentation tier and the business tier.

**Model–View–Controller (MVC)** This pattern is actually a composite pattern, meaning that it itself is made up of other patterns, for example the *Observer* pattern. It is used in PHYRE to separate presentation logic from business logic and data. PHYRE actually features a variation of the MVC pattern. Since reusability of PHYRE's classes is not a primary goal, I decided to merge the controller into the view in order to keep the architecture simple, for the sake of clarity, and to keep the number of classes at a minimum. Microsoft uses an own name for this variant of the MVC pattern, called the *Document–View* pattern [23].

## 10 UI Design: PHYRE's Single-Screen Interface

The last section of this paper talks about PHYRE's user interface.

Contrary to conventional web applications, RICH INTERNET APPLICATIONS do not rely on page refreshes in order to display newly requested data. This means, that an RIA can display all relevant information on a single screen, hence *single-screen interfaces*, and is able to update certain parts of it separately and only when requested.

I took advantage of this fact and designed PHYRE's user interface after the following two principles, which seem especially important to me:

I Display all relevant information *at a glance*.

II Let the user get to the information she or he needs *as quickly as possible*.

You can see the translation of the first principle in the following two drafts of PHYRE's user interface, which were designed for the purpose of demonstration to the future users of the application:



Figure 8: Draft of a horizontal layout for PHYRE's user interface

# PHYRE



Figure 9: Draft of a vertical layout for PHYRE's user interface

Even in those very early drafts, you can see that users can choose from a list of experiments and then view the corresponding details on the same screen, without having to undergo a complete page refresh.

To see the second principle applied in PHYRE, please refer to Figure 10 and 11, while you are reading the following description.

First, let us reproduce the steps that are necessary to get to an information when we are *browsing* the experiments catalog with PHYRE (Figure 10):

- 1.) Select a category.
- 2.) Choose an experiment from the corresponding category.
- 3.) View the experiment's information.

Now let us look at the same when we are *searching* the catalog (Figure 11):

- 1.) Enter a keyword and press search.
- 2.) Choose an experiment from the search results.
- 3.) View the experiment's information.

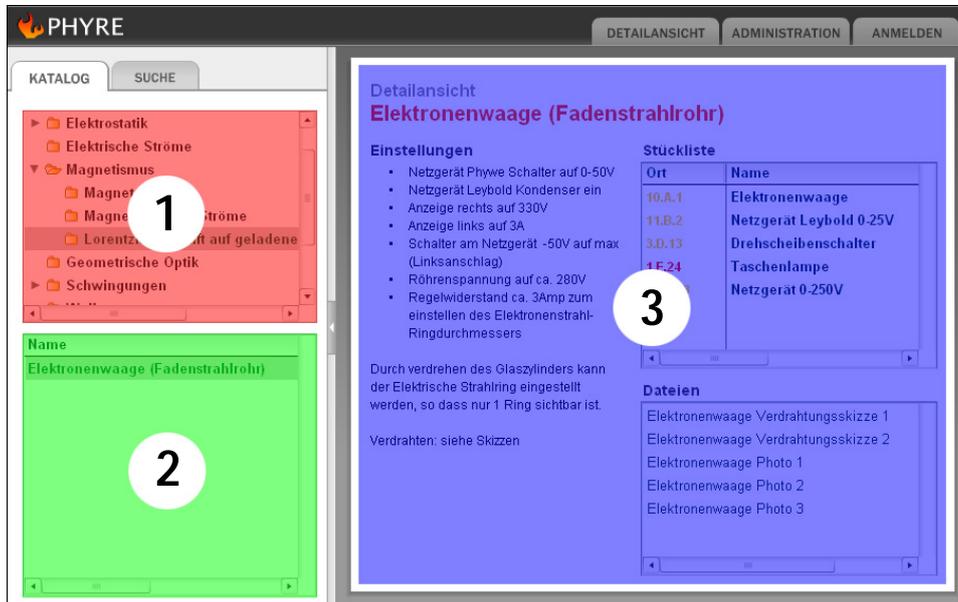


Figure 10: Browsing PHYRE's catalog

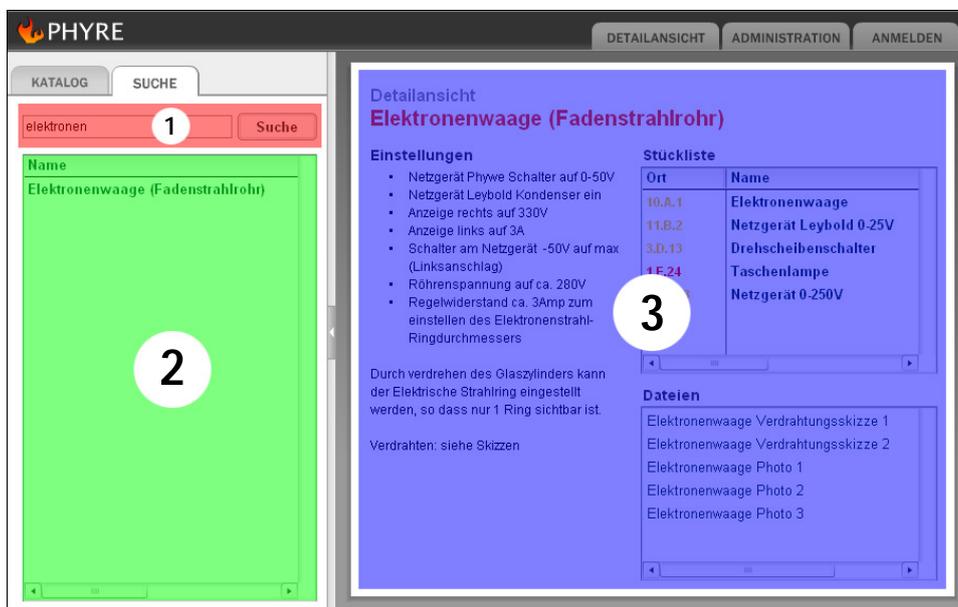


Figure 11: Searching PHYRE's catalog

As seen in the previous two situations, where only three steps are necessary to get to a certain information, and especially when using PHYRE in real life, its user interface design turns out to be very accessible and intuitive. It enables the user to see all the relevant information *at a glance* and at the same time, all the information can be accessed with a minimum amount of steps necessary.

While using PHYRE, you should pay attention to the little details of its user interface, like the possibility of resizing or even hiding the sidebar placed on the left, which gives you more space for viewing the requested information.

Designing good user interfaces means to put oneself in the position of the users that will eventually use the application and to understand their needs. With this in mind you can design user interfaces that enable users to get their job done quicker and be more efficient without having to undergo a long-lasting learning process.

## 11 Closing Thoughts

This paper has reached its end, but my journey as application developer will definitely continue. Over the past six months, during which I extensively dealt with the topics and issues surrounding the fascinating world of RICH INTERNET APPLICATIONS, I learned more than I would have ever thought before. A small part of my experiences is reflected in this paper, an even bigger part in the architecture of PHYRE and the thousands of lines of code of which it is made up, while the rest remains a truly personal experience.

Speaking of PHYRE, I can proudly say that I achieved all the goals I set for myself personally, and at the same time, I accomplished to implement nearly all the features that are found in the specifications.

As always, not everything works out as planned. For example, this was the case with the file upload module for PHYRE. Due to Macromedia Flash Player's lack of support for file uploads and the non-existence of worthy alternatives, except for a few hacks and workarounds, I unfortunately had to abandon the idea of making file uploads possible directly in PHYRE itself.

Other features that I had in my mind, for example allowing users to print the data sheets that are displayed in the application, were not implemented due to the limited time I was given to finish my project.

The preceding setbacks do not actually bother me since the beautiful thing about PHYRE is, that its well thought-out and open architecture allows me to implement these features easily at any time later.

Since developing applications is not a scientific process, I cannot uncover the definite steps which are necessary to build a successful RICH INTERNET APPLICATION. Nevertheless, at this point I will mention six lessons learned from the time I spent working on my project:

### **Prepare Yourself**

Before sitting in front of my computer and starting to work on the project, I read a lot of books, articles, and relevant discussions in forums that are all related to the topic I chose. At the same time, I talked to the involved people from the school and elaborated the specifications for the application.

The bottom line is, having come this far, I can say that this intensive time of preparation definitely saves you a lot of time in the long term, since you truly know where your goals lie and what has to be done in order to get there ...

### **Plan Ahead**

From the development's point of view, I started out by modeling the database upon which the catalog is based. On the entity-relationship diagram, which comes with this paper, you can see that there are a

few tables represented which are not actually used in this first version of PHYRE. Even so, I decided to include them, since they will probably be used in a future version of PHYRE.

Always make sure your application's architecture is open to extension, since there will always come a point where something has to be added or changed. While spending time on thinking about the future of your system, do not lose focus on the upcoming tasks ...

### **Take One Step at a Time**

After having defined PHYRE's architecture and database model, I began building the shell of its user interface. During this process, I added the first feature, namely displaying the different categories of the catalog. This feature included an interaction of all three tiers. It required the programming of the logic in the business tier which is responsible for fetching all categories from the database. On the other hand, I had to develop a visual component whose task it was to display those requested categories in a hierarchical tree.

Each one of PHYRE's components and modules was first built separately in the form of a self-contained element and then integrated into the main application. This approach improves the chance of finding and eliminating bugs, as they appear isolated from the main application.

By taking one step at a time, as described above, you will accomplish your goals faster and more efficiently ...

### **Be Courageous**

When you realize that you could have done something in a better way, have the courage to go back and to do it over again. I ensure you that this will save you from a lot of trouble in the long run ...

### **Test, Test, Test**

Once you have implemented a particular module or component, you should test it thoroughly, to make sure it behaves the way you planned it, and that it does not break or interfere with other parts your system ...

### **Look Back and Learn**

The very last and somehow most frustrating lesson I learned is, that once you have finished your project, you look back and see so many things you could have done so much better with the knowledge you acquired during the time of its emergence.

Thinking about it, it is nice to know that the next project will bring forward new challenges since you already have learned how to tackle the old ones ...

## 12 Acknowledgements

I would not have come this far, if it were not for certain people who supported me throughout the entire time during which I was working on this project.

At this point, I would like to express my deepest gratitude to the following people: Mr. Byland, my tutor and physics teacher, the person who gave me the idea for my PHYRE, for accompanying me the whole time and for all his valuable advice. The Flash community, for sharing all its knowledge. The authors of the books I read, for preparing me and making me understand. Cindy, Greg, Ian, Eric & Owen Lamb, who allowed me to experience the incredible American way of life, for teaching me the American language. The people who make me smile and feel good each and every day. Heinz, my mentor, for believing in me. Janick, my best friend, who introduced me to the fascinating world of computers, for inspiring me and always listening to me patiently whenever I have a problem. Thomas, my brother, who is the reason for the fact that this paper is not about something completely else, for doing an excellent job of proofreading. My parents, to whom I owe my existence, for the moral support during this intensive time. To Julia, *my girl*, for loving me the way I am.

— *Daniel Gąsienica*  
*Uitikon Waldegg, Switzerland*  
*March 2005*

## A Additional Resources

This page lists resources that were of great value before and during the time I was working on this paper. These additional resources provide further information to the topics discussed in this paper.

### SQL & Databases

---

*SQL Pocket Guide* by Jonathan Gennick, ISBN: 0-596-00512-1

*PHP & MySQL* by Johann-Christian Hanke, ISBN: 87-91364-36-1

*PHP professionell* by W. J. Gilmore, ISBN: 3-89842-159-7

*MySQL Table Joins* by W. J. Gilmore,

<http://www.devshed.com/c/a/MySQL/MySQL-Table-Joins>

### ActionScript & OOP

---

*ActionScript for Flash MX* by Colin Moock, ISBN: 0-596-00396-X

*Essential ActionScript 2.0* by Colin Moock, ISBN: 0-596-00652-7

### Design Patterns & Refactoring

---

*Design Patterns by the Gang of Four*, ISBN: 0-201-63361-2

*Head First Design Patterns*, O'Reilly, ISBN: 0-596-00712-4

*Introducing Swing Architecture*, Sun Microsystems

[http://java.sun.com/products/jfc/tsc/articles/getting\\_started/getting\\_started2.html](http://java.sun.com/products/jfc/tsc/articles/getting_started/getting_started2.html)

*How to use Model-View-Controller (MVC)* by Steve Burbeck, Ph.D.

<http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>

*Refactoring* by Martin Fowler, ISBN: 0-201-48567-2

### Miscellaneous

---

*MD5*, Wikipedia

<http://en.wikipedia.org/wiki/MD5>

*Benefits of Using the n-Tiered Approach for Web Applications*, Macromedia

<http://www.macromedia.com/devnet/mx/coldfusion/articles/ntier.html>

## **B Tools**

**Flash Authoring IDE:** Macromedia Flash MX 2004 Professional  
<http://www.macromedia.com/software/flash/>

**ActionScript Editor:** SE|PY ActionScript Editor  
<http://www.sourceforge.net/projects/sepy/>

**PHP Editor:** Macromedia Dreamweaver MX 2004  
<http://www.macromedia.com/software/dreamweaver/>

**Database Modeling Tool:** DBDesigner 4  
<http://www.fabforce.net/>

**Server-Side Testing Environment:** ApacheFriends XAMPP 1.4.6  
<http://www.apachefriends.org/>

## **C Source Code**

For a printed version of PHYRE's source code please refer to the separate booklet titled: PHYRE: THE SOURCE.

## References

- [1] BBC (2003). *Web's inventor gets a knighthood*. Accessed 15th October 2004.  
<http://news.bbc.co.uk/1/hi/technology/3357073.stm>
- [2] Macromedia (2002). Allaire, Jeremy. *Macromedia Flash MX: A Next-Generation Rich Client*. Accessed 8th December 2004.  
<http://download.macromedia.com/pub/flash/whitepapers/richclient.pdf>
- [3] Macromedia. *Rich Internet Applications*. Accessed 8th December 2004.  
[http://www.macromedia.com/resources/business/rich\\_internet\\_apps/](http://www.macromedia.com/resources/business/rich_internet_apps/)
- [4] MINI USA. *Car Configurator RIA*. Accessed 8th December 2004.  
<http://www.miniusa.com/link/buildyourown/minicooper/>
- [5] E\*TRADE. *Stock Market RIA*. Accessed 8th December 2004.  
<https://us.etrade.com/e/t/invest>
- [6] Microsoft (2000). *Glossary*. Accessed 15th October 2004.  
<http://www.microsoft.com/windows2000/en/server/iis/default.asp?url=/windows2000/en/server/iis/htm/core/iigloss.htm>
- [7] ECMA. *Standard ECMA-262*. Accessed 10th February 2005.  
<http://www.ecma-international.org/publications/standards/Ecma-262.htm>
- [8] Macromedia (2004). *Macromedia Flash Player: Version Penetration*. Accessed 10th February 2005.  
[http://www.macromedia.com/software/player\\_census/flashplayer/](http://www.macromedia.com/software/player_census/flashplayer/)
- [9] Macromedia (2004). *The Maelstrom in Your Future*. Accessed 15th February 2005.  
[http://www.macromedia.com/devnet/logged\\_in/wanbar\\_maelstrom.html](http://www.macromedia.com/devnet/logged_in/wanbar_maelstrom.html)
- [10] Moock, Colin (2004). *Nextgen Flash Player Demo in Tokyo*. Accessed 15th February 2005.  
<http://www.moock.org/blog/archives/000146.html>
- [11] Macromedia (2004). *Connecting Macromedia Flash and PHP*. Accessed 18th February 2005.  
<http://www.macromedia.com/devnet/mx/flash/articles/amfphp.html>

- [12] MySQL. *MySQL Product Page*. Accessed October 2004.  
<http://www.mysql.com/products/>
- [13] About. *Bug*. Accessed October 2004.  
<http://pcsupport.about.com/cs/support101/g/bug.htm>
- [14] Moock, Colin (2004). *Essential ActionScript 2.0*. California: O'Reilly Media, Inc.
- [15] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of reusable object-oriented software*. Boston: Addison-Wesley.
- [16] Alexander, C., Ishikawa S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. (1977). *A Pattern Language*. New York: Oxford University Press.
- [17] Cunningham & Cunningham, Inc. (2003). *XP Refactoring FAQ*. Accessed 17th February 2005.  
<http://c2.com/cgi/wiki?XpRefactoringFaq>
- [18] Wikipedia – The Free Encyclopedia. *High-level programming language*. Accessed 10th January 2005.  
[http://en.wikipedia.org/wiki/High\\_level\\_language](http://en.wikipedia.org/wiki/High_level_language)
- [19] Fowler, Martin (1999). *Refactoring – Improving the Design of Existing Code*. Boston: Addison-Wesley.
- [20] Ullman, Larry (2003). *MySQL Database Design*. Accessed 15th September 2004.  
<http://www.peachpit.com/articles/article.asp?p=30885>
- [21] CNET Networks, Inc. (2005). *TechRepublic – Accommodating a many-to-many relationship in Access*. Accessed 17th February 2005.  
<http://techrepublic.com.com/5102-6270-5285168.html>
- [22] IBM (2001). Rainsberger, J.B. *Use your singletons wisely*. Accessed 18th February 2005.  
<http://www-106.ibm.com/developerworks/webservices/library/co-single.html>
- [23] Microsoft (2005). *Model-View-Controller*. Accessed 18th February 2005.  
<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/desmvc.asp>