Thesis Submission Form

Save File as "Your_Lastname_YourFirstName.doc" and send to your thesis supervisor.

Author (1) (Last Name, First Name, Initials)	Hauck, Kevin, KH
Author (2) (If required)	–
Author (3) (If required)	–
Class	E.g. PSYC 4105 COSC 4325
Abstract	See thesis
Notes	E.g. Includes figures. E.g. Includes documentation on computer disc. See thesis
Archiving, Access, and Use License/ Terms	The author(s) has given permission for this thesis to be consulted as a regular part of the ***Arthur A. Wishart Library collection*** and also to reproduce all or parts of it, for scholarly research only, in compliance with the Canadian Copyright Act. This digital edition is released under Creative Commons Attribution-Noncommercial-No Derivative Works 2.5 Canada License. You are free to share — to copy, distribute and transmit the work under the following conditions: Attribution. You must attribute the work in the manner specified by the author(s) or licensor(s) (but not in any way that suggests that they endorse you or your use of the work). Noncommercial. You may not use this work for commercial purposes. No Derivative Works. You may not alter, transform, or build upon this work. For any reuse or distribution, you must make clear to others the license terms of this work. Any of the above conditions can be waived if you get permission from the copyright holder(s). The author's moral rights are retained in this license.
Date Work Created	Mar. 22/2012
Date Submitted to Library	April. 10/2012
	By completing and submitting this document the author(s) indicate that he or she has read, understood and accepted the terms of the license above.

Software Testing and Development on the Open Source Integrated Library System Evergreen

by

Kevin Hauck

A thesis
presented to Algoma University
in fulfillment of the
thesis requirement for the degree of
Bachelor of Science
in
Computer Science

Advisor: Dr. Simon Xu

Sault Ste. Marie, Ontario, Canada, 2012

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Libraries using software to manage their operations have specific requirements since, in most cases, library operations can be quite extensive. The software required to manage a library must be well written and reliable. The Open Source Integrated Library System Evergreen has garnered much attention and development but lacks an adequate testing suite. Due to the intense nature of Evergreen's requirements, varying from server interaction, database management, and communication amongst applications, the entailing source code is vast. To reduce the errors during development and usage, it is wise to understand the requirement of testing, specifically, White Box Testing. This study will engage in the use of creating White Box Tests and then using coverage reports to compare the tests against the full Evergreen software suite. In doing so, we will find that given Evergreen's modular nature; many of the tests written overlap other modules, causing indirect testing. This result is useful because indirectly tested modules can be removed from the test plan, and thus an exhaustive test suite is not required.

Keywords: *Evergreen, OpenSRF, Integrated Library Systems, Open Source Software, Project Conifer, Perl, White Box Testing, CPAN, Code Coverage*

Acknowledgements

I wish to extend my thanks to my advisor, Dr. Simon Xu, for his assistance through this exciting work. I also wish to thank Robin Isard and Richard Scott from the Wishart Library for their assistance and expertise with the Evergreen system.

Table of Contents

AUTHOR'S DECLARATION ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Figures vi

Chapter 1 Introduction and Background Information 1

1.1 Integrated Library Systems 1

1.2 Open Source Software 2

1.3 Open Source Integrated Library Systems 3

1.4 Evergreen 4

1.5 Technical Details of Evergreen 6

1.6 Perl and Testing 7

1.6.1 Perl – Background Information 7

1.6.2 Testing with Perl 8

1.7 Objectives 8

1.8 Motivation 9

Chapter 2 Methodology and Techniques 10

2.1 Getting Started 10

2.1.1 Installation of OpenSRF and Evergreen 10

2.1.2 Evergreen's Structure 13

2.2 White box testing 15

2.3 Perl Unit Tests 16

2.3.1 Testing using the Test::More module 16

2.3.2 Unit Tests 19

2.4 Coverage Essentials 20

2.4.1 What is Coverage? 20

2.4.2 Devel::Cover and Application 21

2.5 Creating Tests 23

2.6 Alternate Approaches 24

2.6.1 Black Box Testing 24

2.6.2 Why are we not using black box testing? 25

Chapter 3 Analysis 26

3.1 Effectiveness 26

3.2 Results 26

3.3 Discussion 27

Chapter 4 Conclusions 28

Chapter 5 Future Work 29

Appendix A Source Code 30

Appendix B Test::More Synopsis 48

Appendix C Devel::Cover Synopsis 49

Appendix D Coverage Reports 57

Coverage Summary 501

List of Figures

Figure 1 5
Figure 2 11
Figure 3 12
Figure 4 13
Figure 5 14
Figure 6 14
Figure 7 17
Figure 8 17
Figure 9 18
Figure 10 18
Figure 11 21
Figure 12 22
Figure 13. 25

Chapter 1
Introduction and Background Information

1.1 Integrated Library Systems

Libraries in academic institutions are required to manage a variety of tasks and it is important to realize that many of the concepts from the information technology field can be applied to library science, maintenance, and upkeep. These concepts include, database and resource management, given that libraries manage large amounts of information. Modern libraries are fortunate to have software to manage many of their tasks; such tasks include cataloguing, executing acquisitions, and circulation. The software used to manage the above is known as an Integrated Library System, in which functionality is combined with database access and manipulation. The idea, development, and usage of Integrated Library Systems have caused much discussion among professionals in fields such as information technology and library science. The inherit nature and purpose of a library is to provide vast information; therefore, one of the main tools required by an Integrated Library system is a database. The science behind databases and database management systems relies heavily on many concepts found in computer science and information technology that one can see that the word integrated is not only used to describe the systems' interaction with library services, but also with many ideas in computer science. Although Integrated Library Systems are built to serve libraries, it is indeed still software and is therefore maintained using a structure programming language.

1.2 Open Source Software

Open source software has grown extensively in popularity with a variety of software available ranging from operating systems, basic applications, and advanced multi-faceted systems. The most interesting notion when looking into the development of software in the open source community is that the software being created and maintained is comparable to many popular proprietary titles. For example, Debian, one a popular open source operating system has 29,000 different packages available for free on its website. Development of open source software is largely done in a community setting, as in many cases there is no real financial motivation. Regardless of any work contributed to the software, the end product will always be publicly available. However, in many cases of development, open source software does turn a profit through selling support and consulting contracts. This is a viable and popular option as the costs of developing and supporting open source software can be considerably less than even the purchasing price for many proprietary systems.

From the notion of public availability, open source software benefits greatly from the possibility of the actual users, who, with the proper motivation, become co-developers [1]. The more popular cases of open source software have larger user bases and therefore a larger possible set of extended developers. Given that open source allows it users' access to the source code, there is the possibility of some users providing development. End-users that have access to the information and source code of the software that they are using are more likely not only to find problems with software but also, indeed, correct it themselves or provide detailed information about the problem. The idea of rapid releasing was introduced during the development of Linux and many of the individual distributions that were created [1]. Rapid releasing is important to open source software as it allows many of the problems that are easily corrected to be implemented quickly and effectively. With the arrival of Linux and rapid releasing, many of the users of the system were able to quickly provide critical analysis about the quality of the software and offer valuable debugging [1].

1.3 Open Source Integrated Library Systems

Integrated Library Systems vary greatly in the functions and operations that they provide. The requirements and specifications that an individual institution requires of its system also vary. A proven approach to procuring an integrated library system for an institution is to use already written and functioning open source software and modify the existing routines, or provide additions to meet the desired functionality. In support of this Yang and Hofmann proclaim:

> Open source has been the center of attention in the library world for the past several years. Koha and Evergreen are the two major open-source integrated library systems (ILSs), and they continue to grow in maturity and popularity. [11]

Open source systems are accepted in library settings as they allow for the "opportunity to add features to a system that will take years for a proprietary vendor to develop" [11]. Having this in-house ability to optimize software to the requirements that an institution needs is required in libraries since in most cases an individual library has different needs than others. An advantage of choosing open source over proprietary when procuring an integrated library system is that it is free to use and try. In cases where it does not meet the needs of the library, or is unfeasible to modify, the library can choose not to pursue it.

1.4 Evergreen

Evergreen is an Open Source Integrated Library System with initial development by "a small in-house team using open source technologies, [providing] a significantly lower cost than the commercial options" [2]. Given these features, one can see that Evergreen certainly fits well in a library setting, and more specifically a university library, with consideration to how much budgeting and financial management is required to run such institutions. Evergreen was created by the Georgia Public Library Service and has since grown in popularity and functionality. The Georgia Public Library Service's development and implementation of Evergreen "serves its large consortial group of libraries across the state" [2]. The Georgia Public Library Service created the Public Network for Electronic Services (PINES) Consortium involving "250 public libraries in 44 systems [with] over 1.3 million active users and includes nearly eight million items" [3]. This sense of a community and togetherness is what drives success and further development of the Evergreen system and has proven to be effective. With so many different institutions involved and many users with possibly varying backgrounds, the extent of debugging has no bounds. Since many of the users of the system are involved in the consortium and also help in the development this gives them a greater appreciation and understanding of the software. The Public Network for Electronic Services (PINES)

Consortium "provides library automation for public libraries statewide at no cost to the members" [3]. With no real financial binding to get started with the system, the Evergreen system that the PINES consortium delivers provides users and libraries little cause to argue against using the system regardless of providing means to contribute to it. Individuals at libraries who are not involved in the development process can still provide excellent debugging information through their day-to-day usage. Figure 1, found below shows a basic interface that one would use to perform a search.

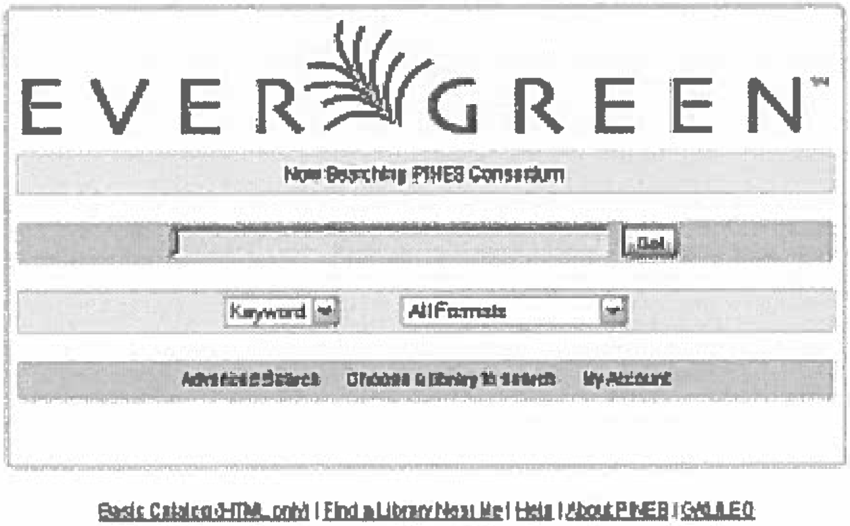

EVERGREEN™
Now Searching PINES Consortium
Go!
Keyword
All Formats
Advanced Search
Choose a Library to Search
My Account
Basic Catalog (HTML only) | Find a Library Near Me | Help | About PINES | GALILEO

Figure 1[11]: A basic representation of an interface that Evergreen provides

1.5 Technical Details of Evergreen

Evergreen is a multi-tiered software suite that uses an Apache web server built on top of a framework called OpenSRF. OpenSRF which stands for The Open Service Request Framework "is an inter-application message passing architecture built on XMPP [the Extensive Messaging and Presence Protocol] " [2], which means that OpenSRF provides a means of communication for software components and applications. Not only is Evergreen open source, but it draws on the strengths of many other open source technologies [2]. The open source technologies that Evergreen takes advantage of include Apache, OpenSRF, a PostgreSQL database, and uses coding from languages such as Perl and C [2]. The interfaces' overlaying technology makes use of XML User Interface Language (XUL), for the client's application, similar to that of Firefox [2]. Evergreen is directly built on top of and utilizes OpenSRF, Weber notes that "messaging is an important part of application frameworks, allowing different processes to exchange information", this highly scalable and robust framework is what Evergreen needs [2]. OpenSRF's basis is from "a decentralized XML technology originally intended for messaging between people" known as Jabber; "Jabber's scalable design also [makes] it ideal for use as an application messaging framework" [2].

1.6 Perl and Testing

Perl and testing pair well together; we will explore this topic further including understanding the many benefits of using Perl and, specifically, the benefits of using Perl to write tests.

1.6.1 Perl – Background Information

Perl is an exceptional programming language that has a variety of conventional uses. With a large set of available modules through the Comprehensive Perl Archive Network, (CPAN), many of the problems one would use perl to solve are already solved. Perl has been accepted as a scripting language and its uses are very practical. Perl programs, "when executed, are compiled into an intermediate representation without creating an immediate file and then interpreted" [4]. This provides Perl an exceptional advantage over most languages since Perl programs are executable by calling their source file. Perl also uses "automatic memory management and [has access to] large free libraries" [4]. As mentioned above, the free library that Perl uses is CPAN, where many modules are importable and well documented. When comparing Perl to other programming languages it certainly exceeds many expectations. Fourment and Gillings found when writing their article, "A Comparison of Common Programming Languages Used in Bioinformatics", that when compared to another scripting language, Python, "Perl clearly outperformed Python for I/O operations" [4]. However, the true strength of Perl is its distributable nature, ease of use, and it's inherent ability to conquer a variety of tasks given that it is a scripting language.

1.6.2 Testing with Perl

Perl has access to community written modules through CPAN, and, of these modules, the most important ones for testing are Test::More, Test::Simple, and Test::Harness. With these modules developers can write concise tests using a variety of assertions to test almost anything written in Perl. After tests have been written, another useful module from CPAN is the Devel::Cover module, which allows the written tests to be profiled against the source code to show code coverage. The need for code coverage is important because if there are crucial areas of code that have not been tested and are possibly problematic, many errors can easily arise. When testing code from a different Perl file, more specifically a Perl module, one can use the same technique as accessing the testing modules: *Module::PerlModule.* Once a module is imported using the 'use' command the subroutines within that module are callable, and one can pass parameters and check the returned results. The assertions provided by Test::More can be found in Appendix B.

1.7 Objectives

The objective of this study is to determine an effective approach for creating a testing suite for an open source project. More specifically, the open source project being used to create a test suite will be the integrated library system Evergreen. We will explore and discover the structure of Evergreen and how its modules are used in conjunction with one another. The structure of Evergreen is what is in question when we are testing since our intent is to create a testing suite using white box testing techniques. Once tests have been written, the next step will be to evaluate the effectiveness of the tests, and that can be accomplished by showing test coverage. The unique process of writing tests, then showing coverage to reveal the remainder of untested code, will be our strategy and approach.

1.8 Motivation

The motivation to create a useful test suite for Evergreen is that because it is such an expansive, modular suite there are many possibilities that errors can arise during development and use. With consideration that the current testing suite for Evergreen is minimal, this is an area that requires attention. Given Evergreen's open source nature, and development model, testing is not generally regarded as high a priority as other tasks during development. Armour notes that testing is "a knowledge acquisition activity rather than simply a post hoc quality assurance process" [10], proving that the usefulness of testing is not only to provide quality assurance but to give a greater understanding of the software. So it is certain that making a test suite not only increases the quality of the source code structure, but also allows it to be better understood. Considering that white box testing is done with regards to the actual code, and not functionality, difficulty certainly increases in that one cannot rely on abstraction for tests. Further consideration of the expansive modular nature of Evergreen also makes this a worthwhile endeavor as it does indeed have more code than a handful of unit cases can test

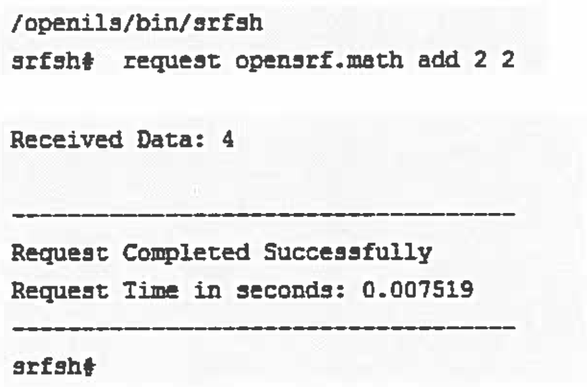
Chapter 2
Methodology and Techniques

2.1 Getting Started

There are a variety of prerequisites before one can start exploring Evergreen's code structure and inner workings. The basic requirements for getting started include having a Linux operating system installed, Debian or Ubuntu, install a PostgreSQL database, and configure an Apache web server.

2.1.1 Installation of OpenSRF and Evergreen

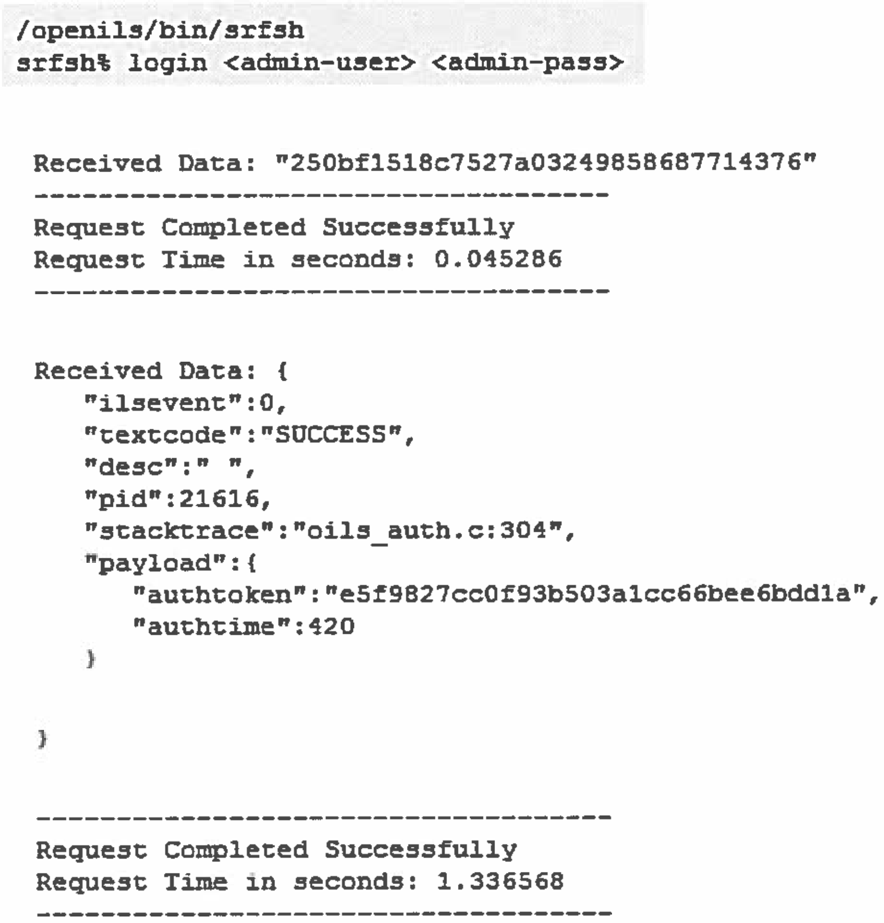
To begin the process of generating a test suite one must first have access to the software's source code and be able to execute it. Getting started with Evergreen is a tedious and lengthy process because to get a full installation with a functioning server there are a variety of steps involved. It is important to have a full installation as the developer and tester can be ensured that their work is not undermined and that the work they do will actually be useful. Since Evergreen is built on top of OpenSRF, OpenSRF must be installed and functional first. The current version available for OpenSRF is OpenSRF 2.0.1 and is available via GIT repository or by a direct zipped download [12]. The installation requirement of OpenSRF, and thus Evergreen, for development is that the operating system required is a Linux distribution, namely, either Debian or Ubuntu. The operating system that is used to conduct this study is Ubuntu-Lucid. After installation and compilation of OpenSRF, OpenSRF requires the user to set up separate, private and public domains, specifically "jabber domains to separate services into public and private realms" [13]. Once domains are setup, they "need two Jabber users to manage the OpenSRF communication", namely, a router and an opensrf user [13]. OpenSRF is a great framework that allows it to be tested without Evergreen built on top of it, namely, one can request some of its more basic functions such as its math service. Figure 2 below explicitly shows the basic operation of OpenSRF's math service, it computes the value of '2+2'.



```
/openils/bin/srfsh
srfsh#  request opensrf.math add 2 2

Received Data: 4


------------------------------------
Request Completed Successfully
Request Time in seconds: 0.007519
------------------------------------
srfsh#
```

Figure 2 [13]: A screenshot of the usage of OpenSRF's math service

After confirmation of the above operation, one can begin the installation process for Evergreen. One must first install and compile Evergreen, once completed, an Apache web server can be configured, and finally, a PostgreSQL 9.0 server [14]. When configuring the PostgresSQL 9.0 server, one must also assign a name for the database, users, admin users, passwords, hostnames, and ports that will be associated with the database [14]. Once the above is completed, one can create or port the database to a remote server [14]. After configuration, Evergreen can be started, and the most basic test to ensure its correctness is to test to see if one can login with the admin user. Figure 3 directly shows the process below.



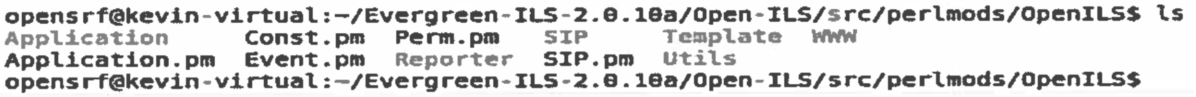
```
/openils/bin/srfsh
srfsh% login <admin-user> <admin-pass>
```

```
Received Data: "250bf1518c7527a03249858687714376"
------------------------------------
Request Completed Successfully
Request Time in seconds: 0.045286
------------------------------------


Received Data: {
   "ilsevent":0,
   "textcode":"SUCCESS",
   "desc":" ",
   "pid":21616,
   "stacktrace":"oils_auth.c:304",
   "payload":{
      "authtoken":"e5f9827cc0f93b503a1cc66bee6bdd1a",
      "authtime":420
   }

}

------------------------------------
Request Completed Successfully
Request Time in seconds: 1.336568
------------------------------------
```

Figure 3 [14]: A synopsis of an attempt to login and make a connection to a fully installed distribution of Evergreen

2.1.2 Evergreen's Structure

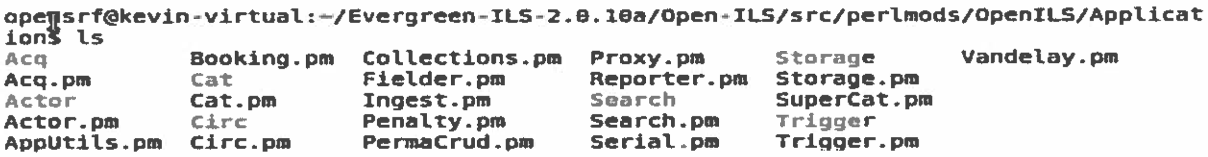
The Evergreen suite is made up of many modules, more specifically perl modules. Each module has a specific use and varies amongst the more general categories: Application, Utilities, Reporter, WWW, and SIP. See Figure 4 for a brief look at the directory structure and arrangement of the perl modules. The modules have extension .pm and within each of the directories more specific perl modules can be found.



```
opensrf@kevin-virtual:~/Evergreen-ILS-2.0.10a/Open-ILS/src/perlmods/OpenILS$ ls
Application      Const.pm  Perm.pm    SIP     Template  WWW
Application.pm   Event.pm  Reporter   SIP.pm  Utils
opensrf@kevin-virtual:~/Evergreen-ILS-2.0.10a/Open-ILS/src/perlmods/OpenILS$
```

Figure 4: Directory structure showing the composition of modules in the OpenILS directory

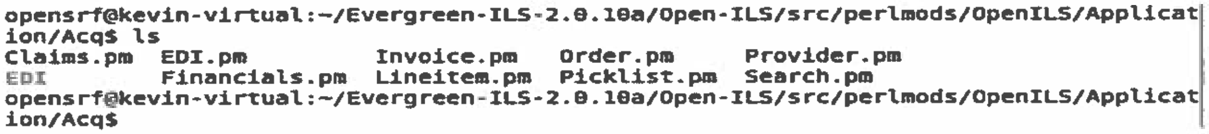
The perl modules found under the Application directory are function specific and are split up further into individual categories varying from acquisitions, actor, catalogues, circulation, search and storage. Below, in Figure 5, we can see the directory structure as well as some additional perl modules, which act as the driving parent of the subdirectories. Some miscellaneous perl modules are also found in this directory, namely modules such as Booking.pm, Collections.pm, and Serial.pm. For example, Booking.pm handles the bookings of resources.



```
opensrf@kevin-virtual:~/Evergreen-ILS-2.0.10a/Open-ILS/src/perlmods/OpenILS/Applicat
ion$ ls
Acq          Booking.pm   Collections.pm   Proxy.pm      Storage       Vandelay.pm
Acq.pm       Cat          Fielder.pm       Reporter.pm   Storage.pm
Actor        Cat.pm       Ingest.pm        Search        SuperCat.pm
Actor.pm     Circ         Penalty.pm       Search.pm     Trigger
AppUtils.pm  Circ.pm      PermaCrud.pm     Serial.pm     Trigger.pm
```

Figure 5: Directory structure showing the composition of modules in the OpenILS subdirectory Application

See Figure 6 below for the structure of Acq for acquisitions. Within this directory we find modules regarding functions in more basic form that are used to perform acquisitions, namely, Claims.pm, Financials.pm, Invoice.pm, LineItem.pm, Order.pm, Picklist.pm, Provider.pm, and Search.pm.



```
opensrf@kevin-virtual:~/Evergreen-ILS-2.0.10a/Open-ILS/src/perlmods/OpenILS/Applicat
ion/Acq$ ls
Claims.pm  EDI.pm         Invoice.pm   Order.pm     Provider.pm
EDI        Financials.pm  Lineitem.pm  Picklist.pm  Search.pm
opensrf@kevin-virtual:~/Evergreen-ILS-2.0.10a/Open-ILS/src/perlmods/OpenILS/Applicat
ion/Acq$
```

Figure 6: Directory structure showing the composition of modules in the OpenILS/Application subdirectory Acq (acquisitions) – thus the path is OpenILS/Application/Acq

Careful examination of individual modules and sub-modules reveal that the majority of modules are composed of subroutines. Understanding this composition and structure is important as we need this information to plan and construct a test suite. We can test the modules by calling the subroutines within them. In cases where parameters are required by a subroutine, we can vary the input we send as parameters to check for different outputs; it satisfies branch, condition, and decision path testing.

2.2 White box testing

White box testing is a fundamental testing technique with a variety of applications. Haller explicitly notes the importance of white box testing in that it "is an important part of every software testing and quality assurance strategy" [5]. The objective of white box testing is to test the structure of software: it does not test functionality, thus revealing any inaccuracies with the structure of the code. White box testing can be summarized in that it "analyzes the source code to identify possible execution paths and parameter sets for the invocation to ensure that the intended execution path is taken" [5]. This approach is useful as it ensures the soundness of the structure and internal workings of the software. Such techniques as path, branch, condition, and decision testing test the actual code, to ensure that source code is reachable, specific conditions results cause exits as proposed, and the internals of decision structures are reachable. McCabe as cited in Williams explains that successful path testing provides "a means for ensuring that all independent paths through a code module have been tested" [15]. So given a variety of different executable paths, we want to ensure that any of the available routes are tested for better coverage. White box testing is a useful approach for testing Evergreen because if any additions or changes occur to the code base, white box testing can be used to test and document any possibilities of faults or errors in existing code caused by shortcomings in updated code.

2.3 Perl Unit Tests

Dividing a planned test suite up into Unit tests is a good approach as it maintains modularity and reduces the risk of error. In the subsequent sections, we will explore the usefulness of Perl's testing abilities and assistive modules, as well as the concept of unit testing.

2.3.1 Testing using the Test::More module

Perl, being a scripting language, allows a variety of different ways to execute perl code and useful ways of naming perl files given their use. For example, Perl not only understands files ending in '.pl', but also files ending in '.pm', and more specifically for this situation, '.t'. More explicitly, '.pm', is for Perl modules, to be used in conjunction with other '.pm' files to create a modular code structure where individual purposes can be coded into single '.pm' files. For writing test cases, developers can use the '.t' file extension and Perl will understand it. As mentioned previously, the most profound and well-used testing module is Test::More from CPAN. As Test::More is derived from the module Test::Simple, we will examine Test::Simple first. The basic structure of a test written in Perl using Test::More can be seen below in Figure 7.

```
use Test::More;

... run your tests ...
```

Figure 7 [6]: A basic synopsis of the structure of a test case, an import statement, followed by code and assertions.

The simplicity of testing in Perl is easy to see because all one needs to start testing is to ensure access to the testing module, Test::More, clear access to the source code to be tested, and begin testing using the assertions provided by Test::More. See Appendix B for a brief look at how the assertions provided by Test::More can be used. Once a developer has a set of test cases, for example, TEST1.t, TEST2.t, and TEST3.t, the developer is able to call the individual test files by using the command perl, as shown in Figure 8.

```
perl TEST1.t
perl TEST2.t
perl TEST3.t
```

Figure 8: An example of how one would run .t test files using the perl command

There is an excellent tool that allows a developer to call many test cases at once, which, when dealing with many tests divided into individual test files, proves to be much more efficient and useful than calling each test file individually. If all of the tests files are placed into a single directory, from the parent directory, the 'prove' command can be used. The synopsis for the prove command can be found in Figure 9.

```
prove [options] [files/directories]
```

Figure 9: The structure of the command 'prove' and how to use it given options and directory structure.

Using the prove command, one can call all of the test files within a directory at once and as they run, you can see the results of each test, results being whether the tests failed or passed.

The provisions provided to show the results of a test are also simplistic and easy to read. The testing modules provided on CPAN use a unique counting technique to show the number of written tests, and the result for each one. An example from the module Test::Simple is shown in Figure 10.

```
1..5
ok 1 - new() works
ok 2 - Title() get
ok 3 - Director() get
not ok 4 - Rating() get
#   Failed test 'Rating() get'
#   in t/film.t at line 14.
ok 5 - NumExplodingSheep() get
# Looks like you failed 1 tests of 5
```

Figure 10 [7]: Visualization of the result that is shown after a test case is run, explicit information about the result of the test is visible.

From Figure 10, we can see line by line exactly what is going on. The first line shows the numbering of the individual tests within the test file, 1..5, which lets us know that there are assertions 1 through 5. The remainder of the output is descriptive information regarding the results of the test. The strength of the testing module is clearly noted in Figure 10 when an error does occur. We have information regarding the failure, including which subroutine, the test file, and at which line in the source code, this concise and well documented information is what one needs for debugging. For larger test suites, the last line is tremendously useful as it allows the tester to see how many of the tests have failed.

2.3.2 Unit Tests

When software is created using modular design, many features can be easier to represent and understand. A useful testing technique when software is composed of many distinct modules is unit testing. Unit testing allows for concise testing of individual units, or modules of source code. In the case that a module is dependent on other modules, drivers and stubs can be used to simulate the desired effects for those modules. The simplicity of unit testing, by testing one single unit of concise source code, reduces the possibility of error and increases readability.

2.4 Coverage Essentials

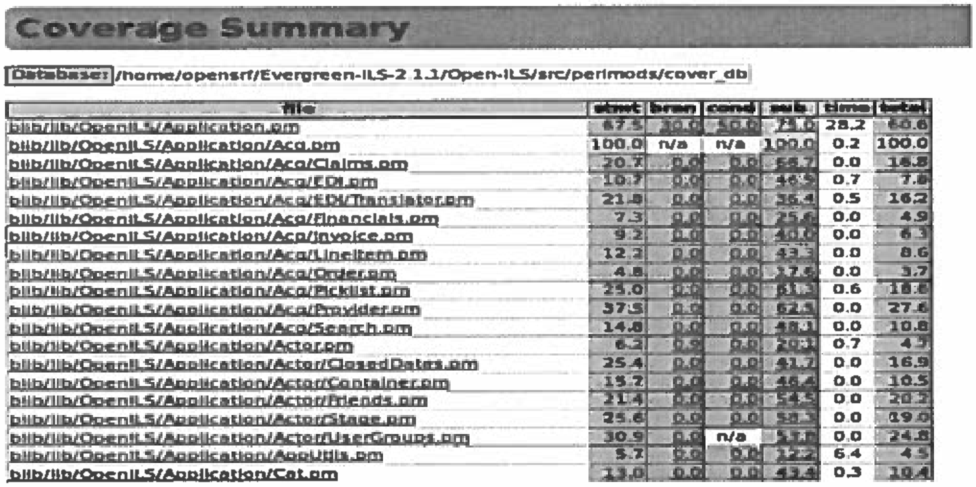
Testing without the evaluation of the tests can certainly limit the effectiveness of the tests in question. One must consider what tools are available to evaluate and gauge the usefulness and thoroughness of a given test suite. It can be difficult to imagine a solution to a question such as, "*how does one test a test?*" However, when using white box tests, the effectiveness of a test suite can be evaluated by determining its coverage.

2.4.1 What is Coverage?

When using white box testing, the idea is to test the code structure, and an important evaluation tool is to check coverage. It is noted by Inoue and Yamada that "testing-coverage is one of the important measures to evaluate the quality of testing and tested software products" [8]. In essence, coverage is the measure which is used to show how much of the source code is evaluated and tested by a given set of tests. This is useful because it allows one to see what areas have and have not been tested, allowing reasonable judgement and evaluation of the current state of a test suite. It also allows one to see any holes in test suites and the determination of any areas that desperately need testing. Inoue and Yamada summarize the coverage testing techniques as statement coverage, branch coverage, and path coverage: statement coverage is the measured result of individual statements executed, thus being an expression of some sort: branch coverage is the measured result of branches taken, which is measureable from decision structures, and looping mechanisms; finally, path coverage is the measured result of "all distinct program paths that have been executed at least once by the tests-cases" [8]. As this study will involve creating white box tests, it is natural to have coverage reports to show the effectiveness and thoroughness of those tests.

2.4.2 Devel::Cover and Application

Devel::Cover, like Test::More, is another perl module that is included in CPAN. A developer is free to download and use Devel::Cover as they please. Devel::Cover is used to provide "code coverage metrics for Perl" [9]. The synopsis for Devel::Cover is also included in Appendix C. Devel::Cover evaluates a test suite against a source code base that returns coverage information in an html file explicitly showing statement, branch, condition, and subroutine coverage. The module also gives useful statistics regarding the current coverage of tests. Figure 11 gives a brief look at the main interface for accessing the reports and gives a generalized description using statistics for the current coverage.



Coverage Summary

Database: /home/opensrf/Evergreen-ILS-2 1.1/Open-ILS/src/perlmods/cover_db

file	stmt	bran	cond	sub	time	total
blib/lib/OpenILS/Application.pm	67.5	30.0	50.0	75.0	28.2	60.6
blib/lib/OpenILS/Application/Acq.pm	100.0	n/a	n/a	100.0	0.2	100.0
blib/lib/OpenILS/Application/Acq/Claims.pm	20.7	0.0	0.0	58.7	0.0	16.8
blib/lib/OpenILS/Application/Acq/EDI.pm	10.7	0.0	0.0	46.5	0.7	7.8
blib/lib/OpenILS/Application/Acq/EDI/Translator.pm	21.8	0.0	0.0	36.4	0.5	16.2
blib/lib/OpenILS/Application/Acq/Financials.pm	7.3	0.0	0.0	25.6	0.0	4.9
blib/lib/OpenILS/Application/Acq/Invoice.pm	9.2	0.0	0.0	40.0	0.0	6.3
blib/lib/OpenILS/Application/Acq/Lineitem.pm	12.2	0.0	0.0	43.3	0.0	8.6
blib/lib/OpenILS/Application/Acq/Order.pm	4.8	0.0	0.0	17.6	0.0	3.7
blib/lib/OpenILS/Application/Acq/Picklist.pm	25.0	0.0	0.0	61.5	0.6	18.6
blib/lib/OpenILS/Application/Acq/Provider.pm	37.5	0.0	0.0	62.5	0.0	27.6
blib/lib/OpenILS/Application/Acq/Search.pm	14.8	0.0	0.0	48.1	0.0	10.8
blib/lib/OpenILS/Application/Actor.pm	6.2	0.0	0.0	20.1	0.7	4.3
blib/lib/OpenILS/Application/Actor/ClosedDates.pm	25.4	0.0	0.0	41.7	0.0	16.9
blib/lib/OpenILS/Application/Actor/Container.pm	15.2	0.0	0.0	46.4	0.0	10.5
blib/lib/OpenILS/Application/Actor/Friends.pm	21.4	0.0	0.0	54.5	0.0	20.2
blib/lib/OpenILS/Application/Actor/Stage.pm	25.6	0.0	0.0	58.3	0.0	19.0
blib/lib/OpenILS/Application/Actor/UserGroups.pm	30.9	0.0	n/a	53.8	0.0	24.8
blib/lib/OpenILS/Application/AppUtils.pm	5.7	0.0	0.0	12.2	6.4	4.5
blib/lib/OpenILS/Application/Cat.pm	13.0	0.0	0.0	43.4	0.3	10.4

Figure 11: A screenshot of the resulting compilation of the test suite against the software suite created using the module Devel::Cover

Figure 11 is a visualization of the perl modules that have been run through Devel::Cover, this list of perl modules in the entire suite is much longer. However, from Figure 11, one can see the functionality and usefulness of Devel::Cover. The conciseness and readability is evident for the perl modules in question and from which directory they originate. The remainder of Figure 11 explicitly allows one to see the intuitive nature of Devel::Cover as it shows the meaningful coverage statistics, using percentages, regarding each of the modules listed. A brief example is shown below in Figure 12 regarding the results for Application.pm:

67.5% statement coverage

30.0% branch coverage

50.0% condition coverage

75% subroutine coverage

Figure 12: Coverage percentages for Application.pm

Therefore, it is easy to see that in comparison to many of the other perl modules listed in Figure 11, excluding Acq.pm, Application.pm has better coverage than any other module shown. The use of the tool Devel::Cover provides a unique way to calculate and determine which areas of the software have been evaluated by our tests, and which areas require attention. During the usage of Devel::Cover, testing becomes easier because there are situations where tests may be written and given the modular nature of projects, such as this one, that one test may indirectly test many other areas unbeknownst to the developer. This is indeed a nice additive feature; therefore, explicitly testing everything that is visible to the developer can now in a sense be excused.

2.5 Creating Tests

With the introduction of Devel::Cover, we can use its effective reports to determine what has been tested thus far. The current Evergreen distributions include a small functioning test suite that uses the Test::More module and the assertion 'use_ok' to verify usability and reachability of each of the '*.pm' modules. The evaluation of the tests with Devel::Cover creates an excellent representation of current coverage. To create more tests, one can use the reports given by Devel::Cover to determine what needs testing, and what has already been tested in addition to what has already been indirectly tested. The process of creating tests is twofold in that a developer should create tests, but after each test is created, compile it with other tests in the test suite and use Devel::Cover to compare current coverage standings. This technique is very useful as it reduces the load of exhaustive white box testing. This means that we do not have to write a test for every statement, branch, conditions statement, and hopefully some subroutines. This approach is effective and saves much time.

Given the modular nature of the Evergreen suite, many of the overlaying modules that control a variety of the sub-modules, which exist in lower subdirectories, often indirectly test many of the paths and coverage requirements of those lower modules. With this in mind, much of the testing done on Evergreen can be done indirectly by examining current coverage. When a test is written for a specific module and that module calls another distinct module, it is reliable and worthwhile to check the coverage of the called module to check indirect testing. In the case that the coverage report indicates that the called module has been tested, then we can reduce our exhaustive testing load by noting that specific tests do not need to be written for the indirectly tested module. Using this technique and approach to writing tests will severely reduce the amount of subsequent planning and creation of tests for the majority of the suite.

2.6 Alternate Approaches

An alternate approach to this study would be to use a different testing technique, namely, black box testing. We will explore black box testing and the reasons why it is not very applicable in the upcoming sub-section.

2.6.1 Black Box Testing

This study involves the use of white box testing because it uses coverage to evaluate the effectiveness of the tests created. While, black box testing tests the functional requirements of the software it does not return any useful information about the quality or structure of the source code. Black box testing tests the functional requirements and one can use a variety of techniques to do this. It depends heavily on abstraction and little is known as of the internal workings of the source code, other than input requirements and desired output. Figure 13 below shows a representation of the basis of a black box test.

Input
Black Box
Output

Figure 13 [16]: A representation of the basis of a black box testing. The 'Black Box' is where the code is and has abstraction, thus, we do not see the code.

2.6.2 Why are we not using black box testing?

Black box testing is not useful for this study because although it may be able to test functional requirements, it gives no feedback regarding the success or failure of individual statements, execution paths, condition statements, branching, or decision structures. Since Evergreen's source code is vast, testing structure is more of a necessity than functional testing because it allows developers to find source code errors with relative ease using a white box testing suite.

Chapter 3
Analysis

3.1 Effectiveness

With a variety of white box tests written and coverage reports to evaluate the tests using statistical analysis in the form of percentage of the code base tested in each perl module, we can evaluate the effectiveness of this process. The approach that was used to create tests was that of 'write and check', which means, write a test, and then run the coverage reports to evaluate current coverage. This proved effective because given the modular nature of Evergreen, one test written can often, and often did, indirectly provide coverage of other dependent modules. This is effective because once we write a test and realize that the test has accomplished tests that we had planned for in the future, there was no longer a need to write additional tests to meet the initial plan.

3.2 Results

From Appendix D, the results from the coverage reports show a variety of different areas of the code base that have testing coverage. With consideration to how large Evergreen is, there is, and generally always will be more to test, given the nature of testing. However, with percentages of coverage obtained shown from the coverage reports in Appendix D, a test suite like this can provide useful and meaningful information to developers as further development occurs on the system. Using the prove command to run all of the tests at once provides great automation for checking to see if any updates have caused system instability or problems in unexpected areas. In short, if we change a subroutine in module 'A' and module 'M' calls module 'A', and in the case that this was overlooked, we may have errors and not know why.

3.3 Discussion

Evergreen is a nicely constructed software suite and it is divided into many modules and each module is nicely divided into distinct subroutines. When software is written like this, it certainly makes it easier to create test suites. The multi-tiered nature and inter-dependence requirements through the use of an Apache webserver, PostgreSQL 9.0 database, and continuous communication and requests using OpenSRF, a variety of errors can easily occur. With a functioning test suite, identifying and mending any errors becomes less tedious. Many of the automation techniques and abilities in Perl certainly make this process simpler when dealing with a large software suite.

Chapter 4
Conclusions

The outcome of this study and work has been worthwhile in that it allowed much to be learned about open source technologies, testing processes, and many of the unique features provided by Linux and Perl. Evergreen is an important software suite, and many different libraries in a variety of locations are dependent on it. Understanding its structure and implementations is a valuable endeavor and creating test suites is definitely one of the most important development needs. In situations, like now for example, where Evergreen is live and it is currently being used, it is important to note that testing bridges the gap between developers and end-users, it allows developers to catch errors after any updates, rather than unfortunate situations such as errors occurring during end-user usage. The approach enhanced in this study can be exploited elsewhere as well as a possible guideline for writing tests. The write then compare coverage approach showed that written tests can exceed expectations in that there is the possibility of indirect testing which lightens an exhaustive testing load.

Chapter 5
Future Work

Future endeavors will certainly include rewriting and improving existing test modules, as well as creating new modules as the development of Evergreen continues and more functionality is added. A software suite or system can always use improvement in some fashion and testing is a great method for reducing the overhead and risks associated with debugging and recovery from regressive development. One can appreciate the effectiveness in cost and development capabilities using a consortium model to produce software, in that it draws on the resources of individuals from the possibility of different fields and different perspectives. White box testing is not the only way to test software, so there is always an avenue to continue testing the Evergreen system; with testing implemented as modular unit tests the risk of error decreases as individual tests grow to be more simplistic, and the combined effort of the tests is what truly brings the desired results. As work continues the most important notion in developing test cases is to keep the tests modular and design each test to analyze very specific entities to maintain effectiveness.

Appendix A

Source Code

Test cases:

Test suite includes additions as well as existing test cases

00-OpenILS.t

```
#!usr/bin/perl

use Test::More tests => 4;

BEGIN {
        use_ok( 'OpenILS' );
}

use_ok( 'OpenILS::Const' );
use_ok( 'OpenILS::Event' );
use_ok( 'OpenILS::Perm' );

diag( "Testing OpenILS $OpenILS::VERSION, Perl $], $^X" );
```

01-OpenILS-Application.t

```
#!usr/bin/perl

use Test::More tests => 13;

BEGIN {
        use_ok( 'OpenILS::Application' );
}
```

```
use_ok( 'OpenILS::Application::AppUtils' );
use_ok( 'OpenILS::Application::Booking' );
use_ok( 'OpenILS::Application::Collections' );
use_ok( 'OpenILS::Application::Fielder' );
use_ok( 'OpenILS::Application::Ingest' );
use_ok( 'OpenILS::Application::Penalty' );
use_ok( 'OpenILS::Application::PermaCrud' );
use_ok( 'OpenILS::Application::Reporter' );
use_ok( 'OpenILS::Application::ResolverResolver' );
use_ok( 'OpenILS::Application::Serial' );
use_ok( 'OpenILS::Application::SuperCat' );
use_ok( 'OpenILS::Application::Vandelay' );
```

02-OpenILS-Application-Acq.t

```
#!usr/bin/perl

use Test::More tests => 11;

BEGIN {
        use_ok( 'OpenILS::Application::Acq' );
}

use_ok( 'OpenILS::Application::Acq::Claims ');
use_ok( 'OpenILS::Application::Acq::EDI ');
use_ok( 'OpenILS::Application::Acq::EDI ');
use_ok( 'OpenILS::Application::Acq::Financials ');
use_ok( 'OpenILS::Application::Acq::Invoice ');
use_ok( 'OpenILS::Application::Acq::Lineitem ');
use_ok( 'OpenILS::Application::Acq::Order ');
use_ok( 'OpenILS::Application::Acq::Picklist ');
use_ok( 'OpenILS::Application::Acq::Provider ');
use_ok( 'OpenILS::Application::Acq::Search ');
```

03-OpenILS-Application-Actor.t

```
#!usr/bin/perl

use Test::More tests => 6;

BEGIN {
        use_ok( 'OpenILS::Application::Actor' );
}

use_ok( 'OpenILS::Application::Actor::ClosedDates' );
use_ok( 'OpenILS::Application::Actor::Container' );
use_ok( 'OpenILS::Application::Actor::Friends' );
use_ok( 'OpenILS::Application::Actor::Stage' );
use_ok( 'OpenILS::Application::Actor::UserGroups' );
```

04-OpenILS-Application-Cat.t

```
#!usr/bin/perl

use Test::More tests => 6;

BEGIN {
        use_ok( 'OpenILS::Application::Cat' );
}
```

```
use_ok( 'OpenILS::Application::Cat::AssetCommon' );
use_ok( 'OpenILS::Application::Cat::AuthCommon' );
use_ok( 'OpenILS::Application::Cat::Authority' );
use_ok( 'OpenILS::Application::Cat::BibCommon' );
use_ok( 'OpenILS::Application::Cat::Merge' );
```

05-OpenILS-Application-Circ.t

```
#!usr/bin/perl

use Test::More tests => 13;

BEGIN {
        use_ok( 'OpenILS::Application::Circ' );
}

use_ok( 'OpenILS::Application::Circ::CircCommon' );
use_ok( 'OpenILS::Application::Circ::Circulate' );
use_ok( 'OpenILS::Application::Circ::CopyLocations' );
use_ok( 'OpenILS::Application::Circ::CreditCard' );
use_ok( 'OpenILS::Application::Circ::HoldNotify' );
use_ok( 'OpenILS::Application::Circ::Holds' );
use_ok( 'OpenILS::Application::Circ::Money' );
use_ok( 'OpenILS::Application::Circ::NonCat' );
use_ok( 'OpenILS::Application::Circ::ScriptBuilder' );
use_ok( 'OpenILS::Application::Circ::StatCat' );
use_ok( 'OpenILS::Application::Circ::Survey' );
use_ok( 'OpenILS::Application::Circ::Transit' );
```

06 –OpenILS-Application-Search.t

```
#!usr/bin/perl

use Test::More tests => 8;

BEGIN {
        use_ok( 'OpenILS::Application::Search' );
}

use_ok( 'OpenILS::Application::Search::AddedContent' );
use_ok( 'OpenILS::Application::Search::Authority' );
use_ok( 'OpenILS::Application::Search::Biblio' );
use_ok( 'OpenILS::Application::Search::CNBrowse' );
use_ok( 'OpenILS::Application::Search::Serial' );
use_ok( 'OpenILS::Application::Search::Z3950' );
use_ok( 'OpenILS::Application::Search::Zips' );
```

07-OpenILS-Application-Storage.t

```
#!usr/bin/perl

use Test::More tests => 3;

BEGIN {
        use_ok( 'OpenILS::Application::Storage' );
}

use_ok( 'OpenILS::Application::Storage::FTS' );
use_ok( 'OpenILS::Application::Storage::QueryParser' );
```

08-OpenILS-Application-Storage-CDBI.t

```
#!usr/bin/perl

use Test::More tests => 13;

BEGIN {
        use_ok( 'OpenILS::Application::Storage::CDBI' );
}

use_ok( 'OpenILS::Application::Storage::CDBI::action' );
use_ok( 'OpenILS::Application::Storage::CDBI::actor' );
use_ok( 'OpenILS::Application::Storage::CDBI::asset' );
use_ok( 'OpenILS::Application::Storage::CDBI::authority' );
use_ok( 'OpenILS::Application::Storage::CDBI::biblio' );
use_ok( 'OpenILS::Application::Storage::CDBI::booking' );
use_ok( 'OpenILS::Application::Storage::CDBI::config' );
use_ok( 'OpenILS::Application::Storage::CDBI::container' );
use_ok( 'OpenILS::Application::Storage::CDBI::metabib' );
use_ok( 'OpenILS::Application::Storage::CDBI::money' );
use_ok( 'OpenILS::Application::Storage::CDBI::permission' );
use_ok( 'OpenILS::Application::Storage::CDBI::serial' );
```

09-OpenILS-Application-Storage-Driver.t

```
#!usr/bin/perl
```

```
use Test::More tests => 3;

use_ok( 'OpenILS::Application::Storage::Driver::Pg::cdbi' );
use_ok( 'OpenILS::Application::Storage::Driver::Pg::fts' );
use_ok( 'OpenILS::Application::Storage::Driver::Pg::QueryParser' );

# These modules are not meant to be loaded as a normal Perl module
# use_ok( 'OpenILS::Application::Storage::Driver::Pg' );
# use_ok( 'OpenILS::Application::Storage::Driver::Pg::dbi' );
# use_ok( 'OpenILS::Application::Storage::Driver::Pg::storage' );
```

10-OpenILS-Application-Storage-Publisher.t

```
#!usr/bin/perl

use Test::More tests => 11;

BEGIN {
	use_ok( 'OpenILS::Application::Storage::Publisher' );
}

use_ok( 'OpenILS::Application::Storage::Publisher::action' );
use_ok( 'OpenILS::Application::Storage::Publisher::actor' );
use_ok( 'OpenILS::Application::Storage::Publisher::asset' );
use_ok( 'OpenILS::Application::Storage::Publisher::authority' );
use_ok( 'OpenILS::Application::Storage::Publisher::biblio' );
use_ok( 'OpenILS::Application::Storage::Publisher::config' );
use_ok( 'OpenILS::Application::Storage::Publisher::container' );
use_ok( 'OpenILS::Application::Storage::Publisher::metabib' );
use_ok( 'OpenILS::Application::Storage::Publisher::money' );
use_ok( 'OpenILS::Application::Storage::Publisher::permission' );
```

11-OpenILS-Reporter.t

```
#!usr/bin/perl

use Test::More tests => 2;

use_ok( 'OpenILS::Reporter::Proxy' );
use_ok( 'OpenILS::Reporter::SQLBuilder' );
```

12-OpenILS-SIP.t

```
#!usr/bin/perl

use Test::More tests => 8;

BEGIN {
        use_ok( 'OpenILS::SIP' );
}

use_ok( 'OpenILS::SIP::Item' );
use_ok( 'OpenILS::SIP::Msg' );
use_ok( 'OpenILS::SIP::Patron' );
use_ok( 'OpenILS::SIP::Transaction' );
use_ok( 'OpenILS::SIP::Transaction::Checkin' );
use_ok( 'OpenILS::SIP::Transaction::Checkout' );
use_ok( 'OpenILS::SIP::Transaction::Renew' );
```

13-OpenILS-Template.t

```
#!usr/bin/perl

use Test::More tests => 3;

use_ok( 'OpenILS::Template::Plugin::Unicode' );
use_ok( 'OpenILS::Template::Plugin::WebSession' );
use_ok( 'OpenILS::Template::Plugin::WebUtils' );
```

14-OpenILS-Utils.t

```
#!usr/bin/perl

use Test::More tests => 20;

use_ok( 'OpenILS::Utils::Configure' );
use_ok( 'OpenILS::Utils::Cronscript' );
use_ok( 'OpenILS::Utils::CStoreEditor' );
use_ok( 'OpenILS::Utils::Editor' );
use_ok( 'OpenILS::Utils::Fieldmapper' );
use_ok( 'OpenILS::Utils::ISBN' );
use_ok( 'OpenILS::Utils::Lockfile' );
use_ok( 'OpenILS::Utils::MFHDParser' );
use_ok( 'OpenILS::Utils::MFHD' );
use_ok( 'OpenILS::Utils::ModsParser' );
use_ok( 'OpenILS::Utils::Normalize' );
use_ok( 'OpenILS::Utils::OfflineStore' );
use_ok( 'OpenILS::Utils::Penalty' );
use_ok( 'OpenILS::Utils::PermitHold' );
use_ok( 'OpenILS::Utils::RemoteAccount' );
use_ok( 'OpenILS::Utils::ScriptRunner' );
use_ok( 'OpenILS::Utils::SpiderMonkey' );
use_ok( 'OpenILS::Utils::ZClient' );
```

```
# LP 800269 - Test MFHD holdings for records that only contain a caption field
my $co_marc = MARC::Record->new();
$co_marc->append_fields(
   MARC::Field->new('853',",",
      '8' => '1',
      'a' => 'v.',
      'b' => '[no.]',
   )
);
my $co_mfhd = MFHD->new($co_marc);

my @comp_holdings = $co_mfhd->get_compressed_holdings($co_mfhd->field('853'));
is(@comp_holdings, 0, "Compressed holdings for an MFHD record that only has a caption");

my @decomp_holdings = $co_mfhd->get_decompressed_holdings($co_mfhd->field('853'));
is(@decomp_holdings, 0, "Decompressed holdings for an MFHD record that only has a caption");
```

15-OpenILS-WWW.t

```
#!usr/bin/perl

use Test::More tests => 10;

use_ok( 'OpenILS::WWW::BadDebt' );
use_ok( 'OpenILS::WWW::EGWeb' );
use_ok( 'OpenILS::WWW::Exporter' );
use_ok( 'OpenILS::WWW::IDL2js' );
use_ok( 'OpenILS::WWW::PasswordReset' );
use_ok( 'OpenILS::WWW::Proxy' );
use_ok( 'OpenILS::WWW::Redirect' );
use_ok( 'OpenILS::WWW::TemplateBatchBibUpdate' );
use_ok( 'OpenILS::WWW::Vandelay' );
use_ok( 'OpenILS::WWW::XMLRPCGateway' );
```

16-OpenILS-WWW-AddedContent.t

```
#!usr/bin/perl

use Test::More tests => 5;

BEGIN {
        use_ok( 'OpenILS::WWW::AddedContent' );
}

use_ok( 'OpenILS::WWW::AddedContent::Amazon' );
use_ok( 'OpenILS::WWW::AddedContent::ContentCafe' );
use_ok( 'OpenILS::WWW::AddedContent::OpenLibrary' );
use_ok( 'OpenILS::WWW::AddedContent::Syndetic' );
```

17-OpenILS-WWW-Reporter.t

```
#!usr/bin/perl

use Test::More tests => 2;

BEGIN {
        use_ok( 'OpenILS::WWW::Reporter' );
}
use_ok( 'OpenILS::WWW::Reporter::transforms' );
```

18-OpenILS-WWW-SuperCat.t

```
#!usr/bin/perl

use Test::More tests => 2;

BEGIN {
        use_ok( 'OpenILS::WWW::SuperCat' );
}
use_ok( 'OpenILS::WWW::SuperCat::Feed' );
```

19-1-1-OpenILS-Application-Acq-Claims-Unit.t

```
#!usr/bin/perl

use Test::More qw(no_plan) ;
use OpenILS::Application::Acq::Claims;
use OpenILS::Utils::Cronscript;

my %defaults = (

     'min=i'     => 0,        # keys are Getopt::Long style options
     'max=i'     => 999,      # values are default values
     'user=s'    => 'admin',
     'password=s' => '',
     'nolockfile' => 1,
);

my $core = OpenILS::Utils::Cronscript->new(\%defaults);
my $opts = $core->MyGetOptions();
$core->bootstrap;
```

```
my @subs = (claim_ready_items, claim_item, claim_lineitem_detail,
                    get_claim_voucher_by_lid, );

use_ok( 'OpenILS::Application::Acq::Claims', @subs);
OpenILS::Application::Acq::Claims::claim_item();
```

19-1-2-OpenILS-Applicatioin-Acq-Financials.t

```
#!usr/bin/perl

use Test::More qw(no_plan) ;
use OpenILS::Application::Acq::Financials;
use OpenILS::Utils::Cronscript;

my %defaults = (

      'min=i'     => 0,        # keys are Getopt::Long style options
      'max=i'     => 999,      # values are default values
      'user=s'    => 'admin',
      'password=s' => '',
      'nolockfile' => 1,
);

my $core = OpenILS::Utils::Cronscript->new(\%defaults);
my $opts = $core->MyGetOptions();
$core->bootstrap;

my @subs = (create_funding_source, delete_funding_source );
use_ok( 'OpenILS::Application::Acq::Claims', @subs);
OpenILS::Application::Acq::Claims::claim_item();
```

19-1-OpenILS-Application-Unit.t

```
#!perl -T

use Test::More qw(no_plan) ;
use OpenILS::Application::Acq;

my @subs = (new);
use_ok( 'OpenILS::Perm', @subs);
```

19-OpenILS-Application-Unit.t

```
#!usr/bin/perl

use Test::More qw(no_plan) ;
use OpenSRF::System;
use OpenILS::Application;
use OpenSRF::Application;
use OpenSRF::Utils::SettingsClient;
use OpenILS::Utils::Cronscript

my %defaults = (

            'min=i'      => 0,          # keys are Getopt::Long style options
        'max=i'      => 999,        # values are default values
        'user=s'     => 'admin',
        'password=s' => ",
        'nolockfile' => 1,
);

my $core = OpenILS::Utils::Cronscript->new(\%defaults);
my $opts = $core->MyGetOptions();
$core->bootstrap;

my @subs = (ils_version, get_idl_file, register_method, authoritative_wrapper);
use_ok ( 'OpenILS::Application' , @subs);


is( OpenILS::Application::ils_version(), '2-1-1', "Testing version");
my $idl = OpenILS::Application::get_idl_file();
```

20-OpenILS-Const-Unit.t

```
#!usr/bin/perl

use Test::More qw(no_plan) ;
use OpenILS::Const;

my @subs = (econst);
use_ok( 'OpenILS::Const');
```

21-OpenILS-Event-Unit.t

```
#!usr/bin/perl

use Test::More qw(no_plan) ;
use OpenILS::Event;
use OpenILS::Utils::Cronscript;

my %defaults = (
        'min=i'      => 0,          # keys are Getopt::Long style options
        'max=i'      => 999,        # values are default values
        'user=s'     => 'admin',
        'password=s' => '',
        'nolockfile' => 1,
);

my $core = OpenILS::Utils::Cronscript->new(\%defaults);
my $opts = $core->MyGetOptions();
$core->bootstrap;

my @subs = (new, _load_events);
use_ok( 'OpenILS::Event', @subs);
```

22-OpenILS-Perm-Unit.t

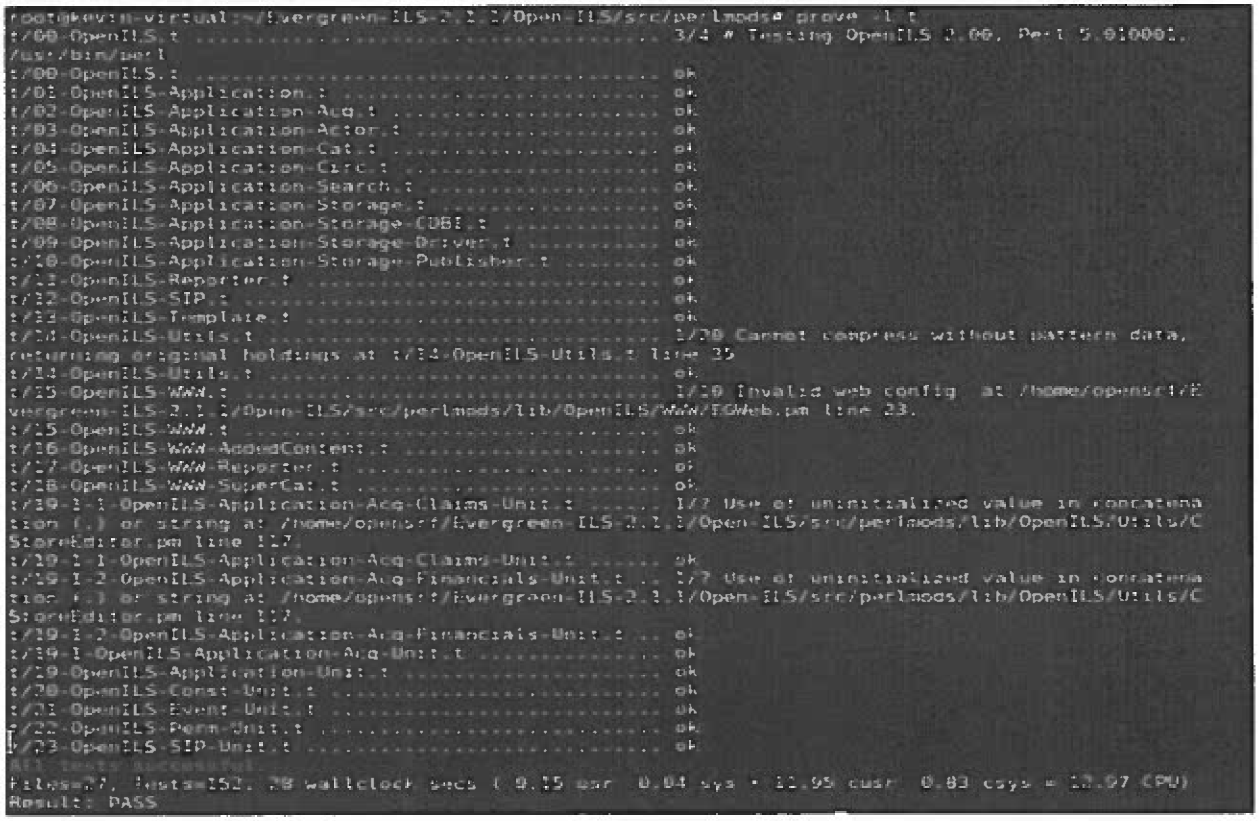
```
#!usr/bin/perl

use Test::More qw(no_plan) ;
use OpenILS::Perm;

my @subs = (new);
use_ok( 'OpenILS::Perm', @subs);
```

23-OpenILS-SIP-Unit.t

```
#!usr/bin/perl

use Test::More qw(no_plan) ;
use OpenILS::SIP;
```

```
my @subs = (new, fetch_session, verify_session,
            editor, config, get_option_value,
            make_editor, clean_text, shortname_from_id,
            patron_barcode_from_id, format_date, login,
            find_patron, find_item, institution, institution_id,
            supports, check_inst_id, to_bool, checkout_ok,
            checkin_ok, renew_ok, status_update_ok,
            offline_ok, checkout, checkin, end_patron_session,
            renew, );

use_ok( 'OpenILS::SIP', @subs);
```

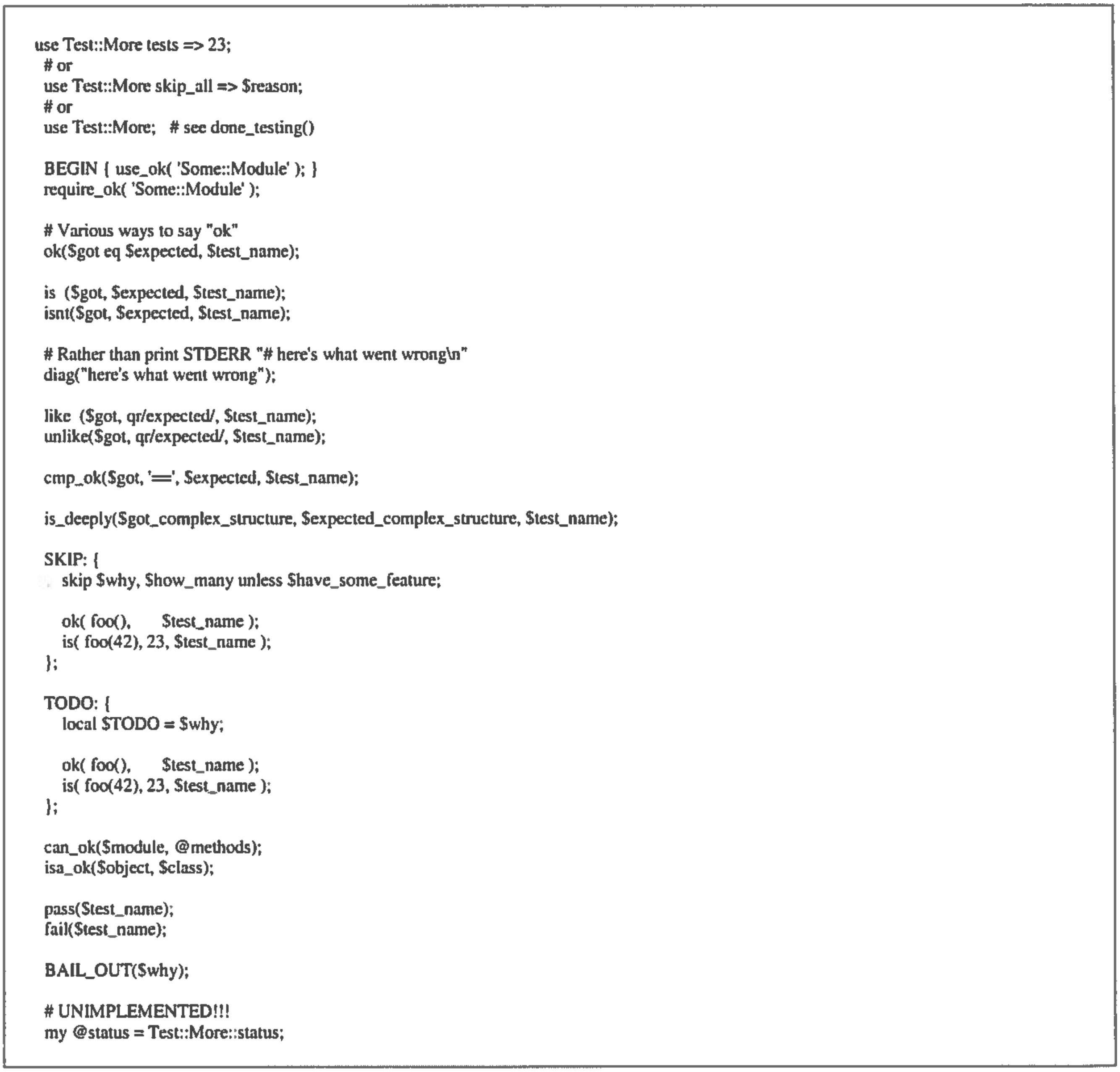
Results of of tests:



```
root@kevin-virtual:~/Evergreen-ILS-2.1.1/Open-ILS/src/perlmods# prove -l t
t/00-OpenILS.t ...................................... 3/4 # Testing OpenILS 2.00, Perl 5.010001,
/usr/bin/perl
t/00-OpenILS.t ...................................... ok
t/01-OpenILS-Application.t .......................... ok
t/02-OpenILS-Application-Acq.t ...................... ok
t/03-OpenILS-Application-Actor.t .................... ok
t/04-OpenILS-Application-Cat.t ...................... ok
t/05-OpenILS-Application-Circ.t ..................... ok
t/06-OpenILS-Application-Search.t ................... ok
t/07-OpenILS-Application-Storage.t .................. ok
t/08-OpenILS-Application-Storage-CDBI.t ............. ok
t/09-OpenILS-Application-Storage-Driver.t ........... ok
t/10-OpenILS-Application-Storage-Publisher.t ........ ok
t/11-OpenILS-Reporter.t ............................. ok
t/12-OpenILS-SIP.t .................................. ok
t/13-OpenILS-Template.t ............................. ok
t/14-OpenILS-Utils.t ................................ 1/20 Cannot compress without pattern data,
returning original holdings at t/14-OpenILS-Utils.t line 35
t/14-OpenILS-Utils.t ................................ ok
t/15-OpenILS-WWW.t .................................. 1/10 Invalid web config  at /home/opensrf/E
vergreen-ILS-2.1.1/Open-ILS/src/perlmods/lib/OpenILS/WWW/EGWeb.pm line 23.
t/15-OpenILS-WWW.t .................................. ok
t/16-OpenILS-WWW-AddedContent.t ..................... ok
t/17-OpenILS-WWW-Reporter.t ......................... ok
t/18-OpenILS-WWW-SuperCat.t ......................... ok
t/19-1-1-OpenILS-Application-Acq-Claims-Unit.t ...... 1/7 Use of uninitialized value in concatena
tion (.) or string at /home/opensrf/Evergreen-ILS-2.1.1/Open-ILS/src/perlmods/lib/OpenILS/Utils/C
StoreEditor.pm line 117.
t/19-1-1-OpenILS-Application-Acq-Claims-Unit.t ...... ok
t/19-1-2-OpenILS-Application-Acq-Financials-Unit.t .. 1/7 Use of uninitialized value in concatena
tion (.) or string at /home/opensrf/Evergreen-ILS-2.1.1/Open-ILS/src/perlmods/lib/OpenILS/Utils/C
StoreEditor.pm line 117.
t/19-1-2-OpenILS-Application-Acq-Financials-Unit.t .. ok
t/19-1-OpenILS-Application-Acq-Unit.t ............... ok
t/19-OpenILS-Application-Unit.t ..................... ok
t/20-OpenILS-Const-Unit.t ........................... ok
t/21-OpenILS-Event-Unit.t ........................... ok
t/22-OpenILS-Perm-Unit.t ............................ ok
t/23-OpenILS-SIP-Unit.t ............................. ok
All tests successful.
Files=27, Tests=152, 28 wallclock secs ( 0.15 usr  0.04 sys + 11.95 cusr  0.83 csys = 12.97 CPU)
Result: PASS
```

Appendix B

Test::More Synopsis



```
use Test::More tests => 23;
 # or
 use Test::More skip_all => $reason;
 # or
 use Test::More;   # see done_testing()

 BEGIN { use_ok( 'Some::Module' ); }
 require_ok( 'Some::Module' );

 # Various ways to say "ok"
 ok($got eq $expected, $test_name);

 is  ($got, $expected, $test_name);
 isnt($got, $expected, $test_name);

 # Rather than print STDERR "# here's what went wrong\n"
 diag("here's what went wrong");

 like  ($got, qr/expected/, $test_name);
 unlike($got, qr/expected/, $test_name);

 cmp_ok($got, '==', $expected, $test_name);

 is_deeply($got_complex_structure, $expected_complex_structure, $test_name);

 SKIP: {
    skip $why, $how_many unless $have_some_feature;

    ok( foo(),       $test_name );
    is( foo(42), 23, $test_name );
 };

 TODO: {
    local $TODO = $why;

    ok( foo(),       $test_name );
    is( foo(42), 23, $test_name );
 };

 can_ok($module, @methods);
 isa_ok($object, $class);

 pass($test_name);
 fail($test_name);

 BAIL_OUT($why);

 # UNIMPLEMENTED!!!
 my @status = Test::More::status;
```

CPAN Synopsis – Test::More

Appendix C

Devel::Cover Synopsis

To get coverage for an uninstalled module:

```
cover -test
```

or

```
cover -delete
HARNESS_PERL_SWITCHES=-MDevel::Cover make test
cover
```

To get coverage for an uninstalled module which uses Module::Build (0.26 or later):

```
./Build testcover
```

If the module does not use the t/*.t framework:

```
PERL5OPT=-MDevel::Cover make test
```

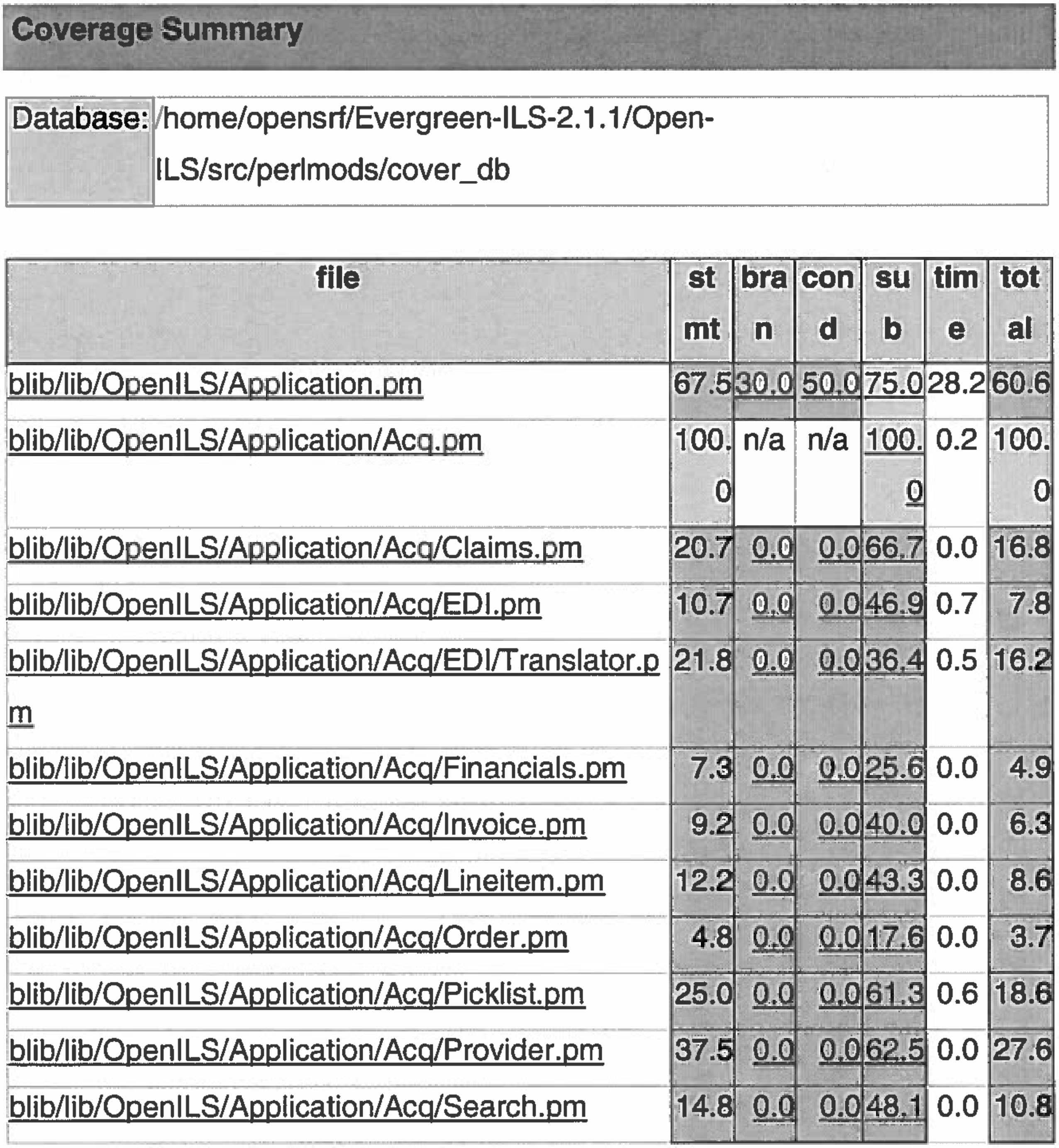
If you want to get coverage for a program:

```
perl -MDevel::Cover yourprog args
cover

perl -MDevel::Cover=-db,cover_db,-coverage,statement,time yourprog args
```

CPAN Synopsis – Devel::Cover

Appendix D

Coverage Reports



Coverage Summary

Database:	/home/opensrf/Evergreen-ILS-2.1.1/Open-ILS/src/perlmods/cover_db

file	stmt	bran	cond	sub	time	total
blib/lib/OpenILS/Application.pm	67.5	30.0	50.0	75.0	28.2	60.6
blib/lib/OpenILS/Application/Acq.pm	100.0	n/a	n/a	100.0	0.2	100.0
blib/lib/OpenILS/Application/Acq/Claims.pm	20.7	0.0	0.0	66.7	0.0	16.8
blib/lib/OpenILS/Application/Acq/EDI.pm	10.7	0.0	0.0	46.9	0.7	7.8
blib/lib/OpenILS/Application/Acq/EDI/Translator.pm	21.8	0.0	0.0	36.4	0.5	16.2
blib/lib/OpenILS/Application/Acq/Financials.pm	7.3	0.0	0.0	25.6	0.0	4.9
blib/lib/OpenILS/Application/Acq/Invoice.pm	9.2	0.0	0.0	40.0	0.0	6.3
blib/lib/OpenILS/Application/Acq/Lineitem.pm	12.2	0.0	0.0	43.3	0.0	8.6
blib/lib/OpenILS/Application/Acq/Order.pm	4.8	0.0	0.0	17.6	0.0	3.7
blib/lib/OpenILS/Application/Acq/Picklist.pm	25.0	0.0	0.0	61.3	0.6	18.6
blib/lib/OpenILS/Application/Acq/Provider.pm	37.5	0.0	0.0	62.5	0.0	27.6
blib/lib/OpenILS/Application/Acq/Search.pm	14.8	0.0	0.0	48.1	0.0	10.8

blib/lib/OpenILS/Application/Actor.pm	6.2	0.9	0.0	20.1	0.7	4.7
blib/lib/OpenILS/Application/Actor/ClosedDates.pm	25.4	0.0	0.0	41.7	0.0	16.9
blib/lib/OpenILS/Application/Actor/Container.pm	15.7	0.0	0.0	46.4	0.0	10.5
blib/lib/OpenILS/Application/Actor/Friends.pm	21.4	0.0	0.0	54.5	0.0	20.2
blib/lib/OpenILS/Application/Actor/Stage.pm	25.6	0.0	0.0	58.3	0.0	19.0
blib/lib/OpenILS/Application/Actor/UserGroups.pm	30.9	0.0	n/a	53.8	0.0	24.8
blib/lib/OpenILS/Application/AppUtils.pm	5.7	0.0	0.0	12.2	6.4	4.5
blib/lib/OpenILS/Application/Cat.pm	13.0	0.0	0.0	43.4	0.3	10.4
blib/lib/OpenILS/Application/Cat/AssetCommon.pm	13.1	0.0	0.0	44.8	0.1	8.9
blib/lib/OpenILS/Application/Cat/AuthCommon.pm	54.0	0.0	n/a	75.0	0.0	52.9
blib/lib/OpenILS/Application/Cat/Authority.pm	30.6	0.0	0.0	58.8	0.0	25.5
blib/lib/OpenILS/Application/Cat/BibCommon.pm	13.6	0.0	0.0	42.9	0.0	10.8
blib/lib/OpenILS/Application/Cat/Merge.pm	19.1	0.0	0.0	71.4	0.1	17.3
blib/lib/OpenILS/Application/Circ.pm	13.8	0.0	0.0	44.6	0.3	10.8
blib/lib/OpenILS/Application/Circ/CircCommon.pm	42.3	0.0	0.0	76.9	0.1	37.7
blib/lib/OpenILS/Application/Circ/Circulate.pm	5.9	0.0	0.0	27.0	0.2	4.3
blib/lib/OpenILS/Application/Circ/CopyLocations.pm	29.7	0.0	0.0	56.2	0.0	23.1
blib/lib/OpenILS/Application/Circ/CreditCard.pm	31.2	0.0	0.0	61.9	0.1	23.0
blib/lib/OpenILS/Application/Circ/HoldNotify.pm	24.0	0.0	0.0	60.0	0.2	21.2

blib/lib/OpenILS/Application/Circ/Holds.pm	5.4	0.0	0.0	26.5	0.1	4.0
blib/lib/OpenILS/Application/Circ/Money.pm	8.4	0.0	0.0	36.7	0.0	6.3
blib/lib/OpenILS/Application/Circ/NonCat.pm	32.8	0.0	0.0	61.9	0.0	25.9
blib/lib/OpenILS/Application/Circ/ScriptBuilder.pm	15.8	0.0	0.0	40.0	0.1	12.5
blib/lib/OpenILS/Application/Circ/StatCat.pm	7.1	0.0	n/a	25.9	0.0	6.5
blib/lib/OpenILS/Application/Circ/Survey.pm	15.0	0.0	0.0	37.5	0.0	12.7
blib/lib/OpenILS/Application/Circ/Transit.pm	22.7	0.0	0.0	60.9	0.1	17.3
blib/lib/OpenILS/Application/Search.pm	71.4	0.0	0.0	87.0	0.9	67.2
blib/lib/OpenILS/Application/Search/AddedContent.pm	69.2	n/a	n/a	42.9	5.1	60.0
blib/lib/OpenILS/Application/Search/Authority.pm	23.1	0.0	0.0	61.9	0.0	21.1
blib/lib/OpenILS/Application/Search/Biblio.pm	6.9	0.0	0.0	24.4	0.0	5.5
blib/lib/OpenILS/Application/Search/CNBrowse.pm	44.4	0.0	n/a	72.7	0.0	39.5
blib/lib/OpenILS/Application/Search/Serial.pm	46.7	0.0	0.0	90.5	0.0	41.5
blib/lib/OpenILS/Application/Search/Z3950.pm	21.5	0.0	0.0	57.1	0.5	18.7
blib/lib/OpenILS/Application/Search/Zips.pm	60.0	0.0	0.0	80.0	0.0	56.1
blib/lib/OpenILS/Application/Storage/CDBI.pm	20.2	1.2	1.8	54.3	3.8	17.0
blib/lib/OpenILS/Application/Storage/CDBI/action.pm	100.0	n/a	n/a	100.0	0.0	100.0
blib/lib/OpenILS/Application/Storage/CDBI/actor.pm	100.0	n/a	n/a	100.0	0.0	100.0
blib/lib/OpenILS/Application/Storage/CDBI/asset.pm	100.0	n/a	n/a	100.0	0.0	100.0

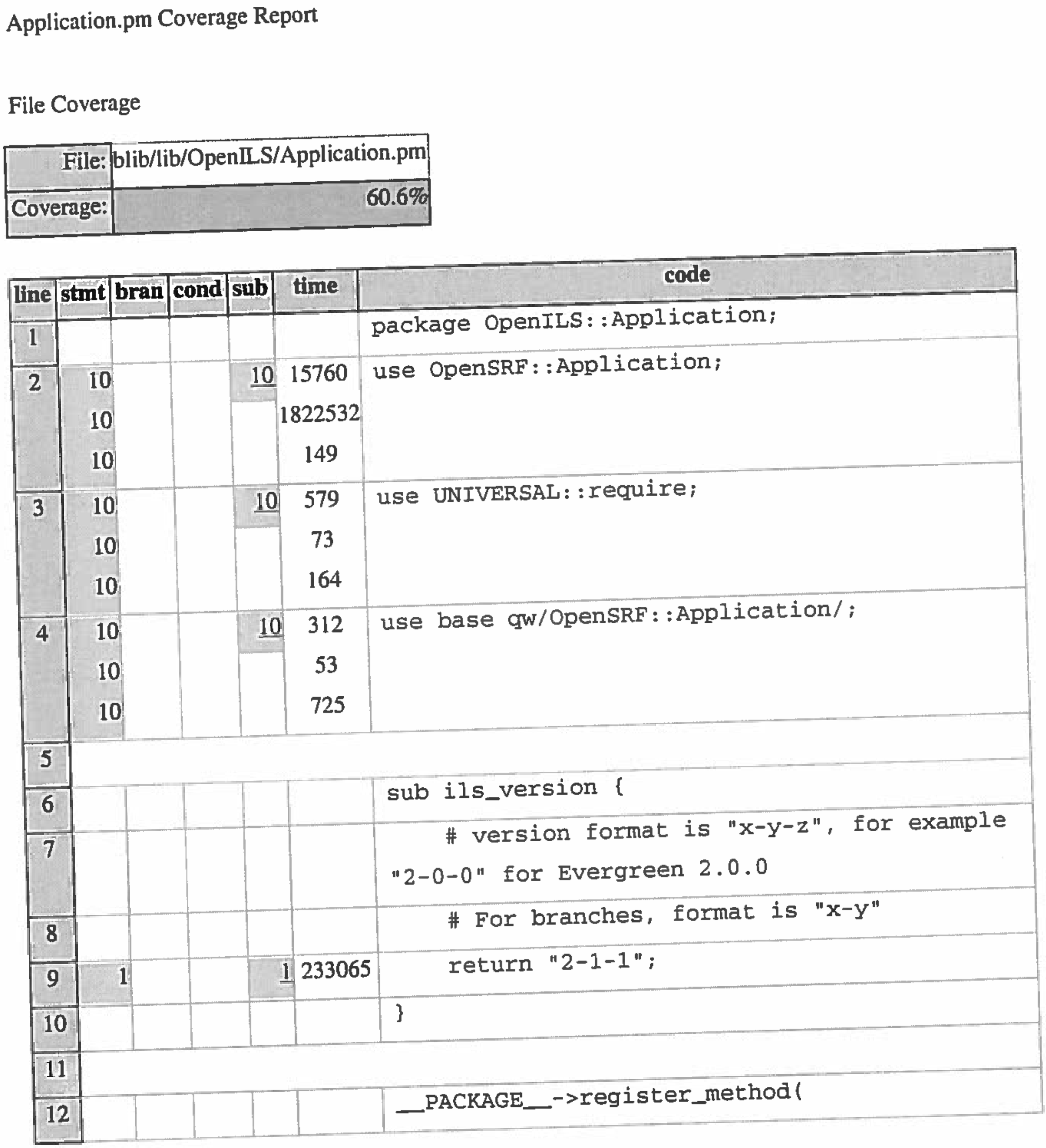
blib/lib/OpenILS/Application/Storage/CDBI/authority.pm	100.0	n/a	n/a	100.0	0.0	100.0
blib/lib/OpenILS/Application/Storage/CDBI/biblio.pm	100.0	n/a	n/a	100.0	0.0	100.0
blib/lib/OpenILS/Application/Storage/CDBI/booking.pm	100.0	n/a	n/a	100.0	0.0	100.0
blib/lib/OpenILS/Application/Storage/CDBI/config.pm	100.0	n/a	n/a	100.0	0.0	100.0
blib/lib/OpenILS/Application/Storage/CDBI/container.pm	100.0	n/a	n/a	100.0	0.0	100.0
blib/lib/OpenILS/Application/Storage/CDBI/metabib.pm	100.0	n/a	n/a	100.0	0.0	100.0
blib/lib/OpenILS/Application/Storage/CDBI/money.pm	100.0	n/a	n/a	100.0	0.0	100.0
blib/lib/OpenILS/Application/Storage/CDBI/permission.pm	100.0	n/a	n/a	100.0	0.0	100.0
blib/lib/OpenILS/Application/Storage/CDBI/serial.pm	100.0	n/a	n/a	100.0	0.0	100.0
blib/lib/OpenILS/Application/Storage/Driver/Pg/QueryParser.pm	7.9	0.0	0.0	24.1	0.1	6.4
blib/lib/OpenILS/Application/Storage/Driver/Pg/cdbi.pm	25.0	25.0	16.7	40.0	1.7	25.8
blib/lib/OpenILS/Application/Storage/Driver/Pg/fts.pm	10.3	0.0	0.0	40.0	3.2	11.1

blib/lib/OpenILS/Application/Storage/QueryParser.pm	8.4	2.8	6.3	11.1	0.1	7.2
blib/lib/OpenILS/Const.pm	96.0	n/a	n/a	87.5	1.6	93.9
blib/lib/OpenILS/Event.pm	28.3	0.0	0.0	71.4	3.6	21.3
blib/lib/OpenILS/Perm.pm	88.9	n/a	n/a	88.9	0.9	88.9
blib/lib/OpenILS/Reporter/Proxy.pm	42.7	0.0	0.0	73.3	4.7	35.5
blib/lib/OpenILS/Reporter/SQLBuilder.pm	6.6	0.0	0.0	9.7	0.1	5.1
blib/lib/OpenILS/SIP.pm	23.8	0.0	0.0	40.4	0.4	20.3
blib/lib/OpenILS/SIP/Item.pm	17.3	0.0	0.0	31.6	0.0	13.1
blib/lib/OpenILS/SIP/Msg.pm	96.0	n/a	n/a	87.5	0.0	93.9
blib/lib/OpenILS/SIP/Patron.pm	10.4	0.0	0.0	16.9	0.0	8.3
blib/lib/OpenILS/SIP/Transaction.pm	50.0	0.0	n/a	66.7	0.0	47.1
blib/lib/OpenILS/SIP/Transaction/Checkin.pm	27.0	0.0	0.0	73.3	0.0	20.2
blib/lib/OpenILS/SIP/Transaction/Checkout.pm	33.8	0.0	0.0	80.0	0.0	32.0
blib/lib/OpenILS/SIP/Transaction/Renew.pm	48.6	0.0	n/a	75.0	0.0	51.1
blib/lib/OpenILS/Template/Plugin/Unicode.pm	15.8	n/a	n/a	14.3	1.3	15.4
blib/lib/OpenILS/Template/Plugin/WebSession.pm	65.6	0.0	0.0	70.0	0.1	57.1
blib/lib/OpenILS/Template/Plugin/WebUtils.pm	63.8	n/a	0.0	66.7	0.0	61.5
blib/lib/OpenILS/Utils/CStoreEditor.pm	14.0	0.0	0.0	26.0	0.6	10.3
blib/lib/OpenILS/Utils/Editor.pm	22.9	0.0	0.0	36.7	0.1	17.0
blib/lib/OpenILS/Utils/Fieldmapper.pm	15.4	1.6	0.0	26.2	3.7	13.4
blib/lib/OpenILS/Utils/MFHD.pm	8.9	0.0	0.0	37.0	0.1	7.8
blib/lib/OpenILS/Utils/MFHD/Caption.pm	5.1	0.0	0.0	18.8	0.0	3.9

blib/lib/OpenILS/Utils/MFHD/Date.pm	10.0	0.0	0.0	28.0	0.0	7.9
blib/lib/OpenILS/Utils/MFHD/Holding.pm	5.0	0.0	0.0	21.9	0.0	4.2
blib/lib/OpenILS/Utils/MFHDParser.pm	17.1	0.0	n/a	43.5	0.0	16.1
blib/lib/OpenILS/Utils/ModsParser.pm	13.2	0.0	0.0	50.0	0.8	12.1
blib/lib/OpenILS/Utils/Penalty.pm	41.9	0.0	0.0	80.0	0.1	39.3
blib/lib/OpenILS/Utils/PermitHold.pm	29.3	0.0	0.0	75.0	0.0	21.9
blib/lib/OpenILS/Utils/RemoteAccount.pm	8.6	0.0	0.0	27.5	0.4	6.5
blib/lib/OpenILS/Utils/ScriptRunner.pm	7.9	0.0	0.0	16.1	1.7	6.6
blib/lib/OpenILS/Utils/ZClient.pm	8.8	0.0	n/a	7.1	0.0	6.8
blib/lib/OpenILS/WWW/AddedContent.pm	39.6	0.0	0.0	60.0	4.6	32.4
blib/lib/OpenILS/WWW/AddedContent/Amazon.pm	49.1	0.0	0.0	50.0	3.9	43.9
blib/lib/OpenILS/WWW/AddedContent/ContentCafe.pm	16.7	0.0	0.0	22.0	0.0	15.0
blib/lib/OpenILS/WWW/AddedContent/OpenLibrary.pm	17.4	0.0	0.0	36.4	0.0	14.9
blib/lib/OpenILS/WWW/AddedContent/Syndetic.pm	20.8	0.0	0.0	25.0	0.0	19.0
blib/lib/OpenILS/WWW/BadDebt.pm	41.1	0.0	0.0	78.6	4.7	36.6
blib/lib/OpenILS/WWW/EGWeb.pm	24.0	2.3	6.2	54.2	0.4	22.0
blib/lib/OpenILS/WWW/Exporter.pm	35.9	0.0	0.0	77.8	1.3	28.5
blib/lib/OpenILS/WWW/IDL2js.pm	45.6	0.0	0.0	60.0	0.0	38.9
blib/lib/OpenILS/WWW/PasswordReset.pm	44.7	0.0	0.0	80.0	0.0	36.7
blib/lib/OpenILS/WWW/Proxy.pm	35.1	0.0	0.0	68.8	0.0	27.8

blib/lib/OpenILS/WWW/Redirect.pm	43.6	0.0	0.0	75.0	0.4	36.6
blib/lib/OpenILS/WWW/Reporter.pm	56.9	0.0	0.0	76.0	3.7	50.7
blib/lib/OpenILS/WWW/SuperCat.pm	8.6	0.0	0.0	49.1	5.0	7.0
blib/lib/OpenILS/WWW/SuperCat/Feed.pm	22.9	0.0	0.0	35.4	0.1	19.3
blib/lib/OpenILS/WWW/TemplateBatchBibUpdate.pm	46.6	0.0	0.0	84.8	0.0	44.0
blib/lib/OpenILS/WWW/Vandelay.pm	61.1	0.0	0.0	90.0	0.0	52.5
blib/lib/OpenILS/WWW/XMLRPCGateway.pm	47.5	0.0	0.0	82.1	0.2	43.7
Total	16.5	0.2	0.7	41.7	100.0	13.0

Appendix E
Coverage Summary



Application.pm Coverage Report

File Coverage

File:	blib/lib/OpenILS/Application.pm
Coverage:	60.6%

line	stmt	bran	cond	sub	time	code
1						package OpenILS::Application;
2	10 10 10			10	15760 1822532 149	use OpenSRF::Application;
3	10 10 10			10	579 73 164	use UNIVERSAL::require;
4	10 10 10			10	312 53 725	use base qw/OpenSRF::Application/;
5						
6						sub ils_version {
7						# version format is "x-y-z", for example "2-0-0" for Evergreen 2.0.0
8						# For branches, format is "x-y"
9	1			1	233065	return "2-1-1";
10						}
11						
12						__PACKAGE__->register_method(

13						api_name => 'opensrf.open-ils.system.ils_version',
14						api_level => 1,
15						method => 'ils_version',
16						);
17						
18						__PACKAGE__->register_method(
19						api_name => 'opensrf.open-ils.fetch_idl.file',
20						api_level => 1,
21						method => 'get_idl_file',
22						);
23						sub get_idl_file {
24	10 10 10			10	902 53 108	use OpenSRF::Utils::SettingsClient;
25	0			0	0	return OpenSRF::Utils::SettingsClient->new->config_value('IDL');
26						}
27						
28						sub register_method {
29	844			844	2092	my $class = shift;
30	844				2947	my %args = @_;
31	844				2998	my %dup_args = %args;
32						
33	844		33		4305	$class = ref($class) || $class;
34						
35	844		50		2250	$args{package} ||= $class;
36	844				8308	__PACKAGE__->SUPER::register_method(

						%args);
37						
38	844	100	67		64430	if (exists($dup_args{authoritative}) and $dup_args{authoritative}) {
39	157				630	(my $name = $dup_args{api_name}) =~ s/$/.authoritative/o;
40	157	50			378	if ($name ne $dup_args{api_name}) {
41	157				307	$dup_args{real_api_name} = $dup_args{api_name};
42	157				363	$dup_args{method} = 'authoritative_wrapper';
43	157				293	$dup_args{api_name} = $name;
44	157				345	$dup_args{package} = __PACKAGE__;
45	157				1653	__PACKAGE__->SUPER::register_method(%dup_args);
46						}
47						}
48						}
49						
50						sub authoritative_wrapper {
51						
52	0	0		0		if (!$OpenILS::Utils::CStoreEditor::_loaded) {
53	0	0				die "Couldn't load OpenILS::Utils::CStoreEditor!" unless 'OpenILS::Utils::CStoreEditor'->use;
54						}
55						

56	0					my $self = shift;
57	0					my $client = shift;
58	0					my @args = @_;
59						
60	0					my $method = $self->method_lookup($self->{real_api_name});
61	0	0				die unless $method;
62						
63	0					local $OpenILS::Utils::CStoreEditor::always_xact = 1;
64						
65	0 0					$client->respond($_) for ($method->run(@args));
66						
67	0					OpenILS::Utils::CStoreEditor->flush_forced_xacts();
68						
69	0					return undef;
70						}
71						
72						1;
73						

Bibliography

[1] Raymond, Eric. "The Cathedral And The Bazaar." Knowledge, Technology & Policy 12.3 (1999): 23. Academic Search Complete. Web. 14 Mar. 2012.

[2] Weber, Jonathan. "Evergreen: Your Homegrown Ils." Library Journal 131.20 (2006): 38-41. Academic Search Complete. Web. 14 Mar. 2012.

[3] Kenney, Brian. "Georgia's 250 PINES Libraries To Create An ILS Their Way." Library Journal 129.14 (2004): 28. Academic Search Complete. Web. 14 Mar. 2012.

[4] Fourment, Mathieu, and Michael R. Gillings. "A Comparison Of Common Programming Languages Used In Bioinformatics." BMC Bioinformatics 9.(2008): 1-9.Academic Search Complete. Web. 17 Mar. 2012.

[5] Haller. "White-Box Testing for Database-Driven Applications: A Requirements Analysis." In Proceedings of the Second International Workshop on Testing Database Systems (DBTest '09). ACM, Article 13 , 6 pages. DOI=10.1145/1594156.1594172 Web. 17 Mar. 2012

[6] Schwern, Michael G. "Test::More." *CPAN*. Version 0.98. N.p., 2008. Web. 17 Mar. 2012

[7] Schwern, Michael G. "Test::Simple." *CPAN*. Version 0.98. N.p., 2008. Web. 17 Mar. 2012

[8] INOUE, SHINJI, and SHIGERU YAMADA. "Testing-Coverage Dependent Software Reliability Growth Modeling." International Journal Of Reliability, Quality & Safety Engineering 11.4 (2004): 303-312. Academic Search Complete. Web. 19 Mar. 2012.

[9] Johnson, Paul. "Devel::Cover." *CPAN*. Version 0.82. N.p., 2011. Web. 17 Mar. 2012

[10] Phillip G., Armour. "The Business Of Software Testing: Failing To Succeed." Communications Of The ACM 54.10 (2011): 30-31. Academic Search Complete. Web. 20 Mar. 2012.

[11] Yang, Sharon Q., and Melissa A. Hofmann. "The Next Generation Library Catalog: A Comparative Study Of The Opacs Of Koha, Evergreen, And Voyager." Information Technology & Libraries 29.3 (2010): 141-150. Academic Search Complete. Web. 20 Mar. 2012.

[12] Members of the Evergreen Project. "OpenSRF." *Evergreen*. Georgia Public Library Service, n.d. Web. 21 Mar. 2012

[13]Members of the Evergreen Project. "Installing OpenSRF 2.0." *Evergreen*. Georgia Public Library Service, 28 Sep. 2011. Web. 21 Mar. 2012.

[14] Members of the Evergreen Project. "Readme for Evergreen 2.1.1." *Evergreen*. Georgia Public Library Service, 16 Nov 2011. Web 21 Mar. 2012

[15] Williams, Laurie. "White-Box Testing." *Open Seminar*. Open Seminar Project, n.d. Web 21 Mar. 2012

[16] Guru 99 Team. "Black Box Testing." *Guru 99*.

Guru 99 Team. n.d. Web 21 Mar. 2012