BTech Project End of Semester Report

Shelley Lowe
4950257
slow056

2011

Abstract

This report explains the work done up to date for the final year BTech project at Kiwiplan. The main goal of the project is to extend an existing Distributed Service Profiler to integrate the database layer to intercept queries made to a database.

It will cover the project and problem description and a short overview of the company Kiwiplan. The profiler is used by developers working for Kiwiplan to help them analyse and find bottlenecks in their code. The profiler currently used by Kiwiplan developers only analyses service calls made between various services. Queries to the database are not specified, even though the time taken for those queries are included in the information returned. This makes it difficult to find problem areas and information about database calls without developers manually going through the log files.

Background information on the distributed service profiler is also given. This will explain the current capabilities of the profiler and why it needs to be improved. Database calls are a very important part of the services so getting information on these queries is crutial to help developers create more efficient code.

The main part of this report will describe the research and work I have done since the beginning of this project. It has been mostly research to help understand the current profiler and find what tools I can use to complete this project. Research into Hibernate, Interceptors and logging has been done and explained in this report.

There is also some information included about a Database Changes Logger, another program created by a BTech student in the past which is used to intercept calls made to the database and log any changes the user made. This is somewhat similar to what this project requires, so research into this changes logger is helpful in finding what techniques would help with extending the Distributed Service Profiler.

The other big part of this report describes the actual parts I have implemented and tested. This included a simple Interceptor which I tested on a simple MySql database and the start of implementing the actual logging Interceptor that will be integrated with the profiler.

Finally, there will be a section describing the work planned to do next. This will just be an overview of the main tasks to do, not a full list of the things to be completed.

Contents

Chapter 1

Introduction

The Bachelor of Technology (IT) degree is a four year Honours degree, with selection of courses mainly from Computer Science and Information Systems. In the final year, the students do a year-long project 8-10 hours a week. The project is compulsory, and carries the weight of two taught University courses.

This report will describe the project carried out by me, Shelley Lowe, at the company Kiwiplan, for my final year of BTech in 2011.

1.1 The Project

My final year BTech project will be carried out at a company called Kiwiplan. This involved me going into the company (located in East Tamaki) once a week as well as doing my own work at home. It will be a database based project to be implemented in Java. The final solution will be an extension to an application Kiwiplan currently uses to analyse their code.

Chapter 2

Project Description

This chapter describes the project and the company Kiwiplan.

2.1 Project Goal

To extend the capabilities of the Java distributed performance profiling tool by adding integration with the Hibernate ORM and the database layer. The new profiling tool should intercept calls made by the services to the database and display this information in the window. The information extracted from the database queries should be displayed as an extension to the call tree created by the current profiler.

2.2 Problem Description

Kiwiplan currently uses a Distributed Service Profiler to help analyse their code. This profiler can check calls and communication on a higher level to analyse the communication between the various services. An example of this communication is when a particular service calls another service, which then queries the database. Currently, developers must manually check the queries by going through the log files, similar to how they used to go through the code before using the profiler. The plan is to add database integration and automate the process to help Kiwiplan be more efficient in creating new solutions.

2.3 Company Information

With over 30 years of expertise in developing software for the corrugating and packaging industry, Kiwiplan provides systems that deliver the high standard required for enterprise-wide, fully automatic integrated systems for Web-enabled supply chain management, sales order management, scheduling and planning, manufacturing, and inventory control.

Kiwiplan first developed over three decades ago by a visionary management and engineering team in a corrugated box plant, Kiwiplan's sophisticated software systems reflect the intimate knowledge of the packaging industry in every menu and function. Kiwiplan

systems exhibit special characteristics and options that only industry participants would be aware of.

With a true global presence, Kiwiplan continues to be the world's premier leading provider of software for the corrugated and packaging industry [1].

2.4 Motivation

Most of the services developed by Kiwiplan must communicate with each other and with the database. Some service methods take a lot of time to process, where the majority of these times are queries to the database. Queries to the database are not specified, even though the time taken for those queries are included in the service method information returned. This is typically the bottleneck in the system so Kiwiplan developers would like to know which methods are taking the longest so they can try to fix these problem areas.

Since the current service profiler only displays the process time under the service method with no information on the database calls, developers must manually check the database logs. Checking where and what these database calls are is important to help developers find the bottlenecks and improve their code.

Without the database integration, the developers can only see which service methods are taking the longest. Any number of factors could cause this delay in computation time. If this information could be extended to display information on database queries, it would help the developers narrow down the problem areas and find out whether the long runtime is due to the actual service method or because of queries to the database.

Chapter 3

Distributed Service Profiler Information

This chapter describes the Distributed Service Profiler currently used by developers in Kiwiplan.

3.1 Background Information

The initial Distributed Service Profiler was created by another BTech student a few years ago. It connects to different services by connecting with specified port numbers. Any methods called by the services to communicate with each other are intercepted and the method information displayed in the profiler window.

The calls made by the various services are stored into nodes and a call tree is built up. This is displayed in the window to show which calls are initiated by another method. Each node displays the information on the method call. This information includes the method name, the average time taken for the method to be completed, the argument and return bytes and the number of hits (number of times this method was called). The time shown on the nodes is the average time taken to process, this is calculated from the total number of hits and time taken.

Leaf nodes are typically where database queries are made and the majority of the time for some longer method calls are due to queries to the database. These nodes are usually where the time taken to process is much higher. The methods which take much longer than others is what the developers are interested in as these usually indicate the problem areas of the system. However, since no information is given on the actual database calls, it is hard for developers to improve the code without going into the database query logs.

An example of the distributed service profiler running is shown in (Figure 3.1).

Each service is represented with a different colour. Options on the right of the window allow the user to connect to different services by entering the desired port number. The call tree is built as calls made between services are captured and each service method node is coloured in the colour of that particular service. This colour is dependent on the connections on the right and extra connections can be added.

3.2 Related Work

A profiling tool is a program typically used in software engineering or computer science for optimization tasks to run a performance analysis on an application. A profile of the program's dynamic behaviour under a variety of inputs is presented by the profiler and represents the program's behaviour from invocation to termination [2]. Profiling is important for understanding program behaviour. It can be a statistical summary of the events caused by the application, a stream of recorded events or an ongoing interaction with the virtual machine manager.

JProfiler is an award-winning all-in-one Java profiler. JProfiler's intuitive GUI helps you find performance bottlenecks, pin down memory leaks and resolve threading issues [3]. It is a commercial Java profiling tool developed by ej-techonologies and can be used as a standalone application or an Eclipse plugin to anaylse Java code.

Most profilers (like the JProfiler) typically analyse the performance of applications to find bottlenecks and memory leaks for that particular program. They usually analyse method calls, memory usage, runtime and frequency of calls and plenty of other issues to test a program's performance from start to termination.

The Distributed Service Profiler used by Kiwiplan analyses performance on a higher level to check service calls made between programs, not within.

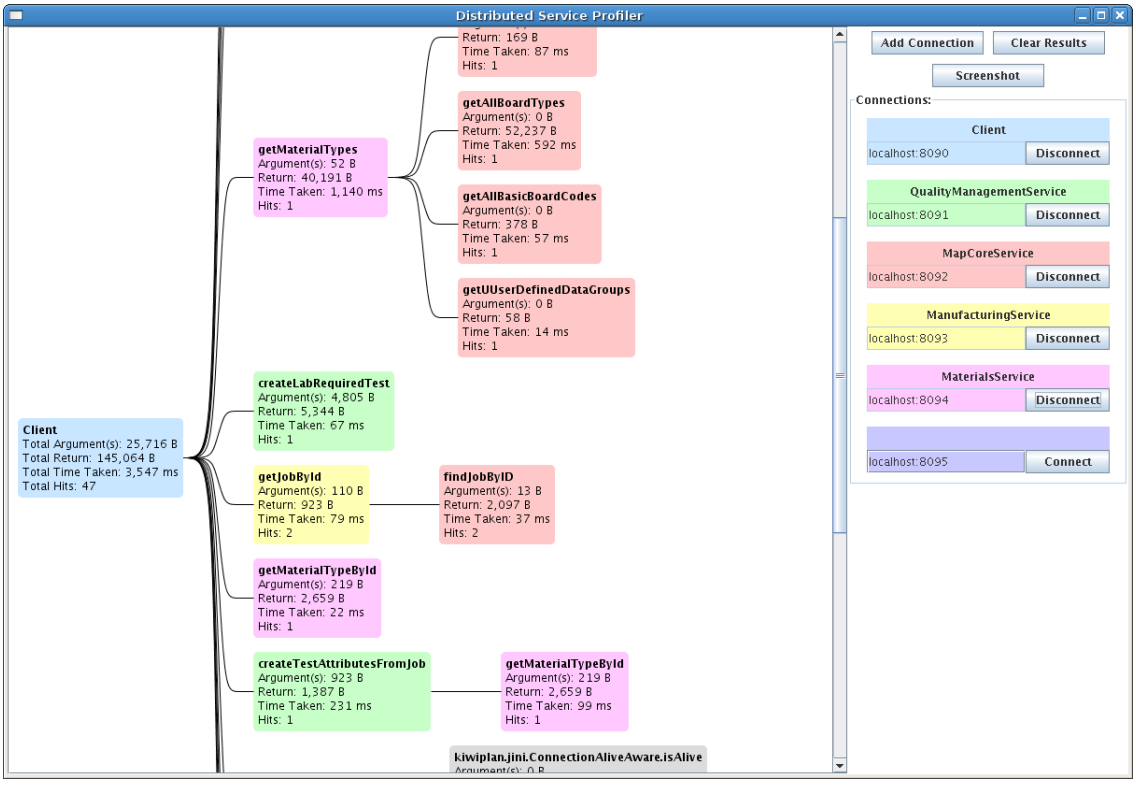

Figure 3.1: Distributed Service Profiler Screenshot

Chapter 4

Development Plan

This chapter describes the development plan I created which includes specifications and details of the project.

4.1 Solution Overview

The final solution should be an extension of the current profiler which can analyse queries made to the database and display the details on screen. The current profiler intercepts calls made between services and shows the details of each method call. This needs to be extended to calls made by the methods to the database and display their details as well as this seems to be where most processing time is spent.

4.2 Architecture

The current profiler will be extended and will use an interceptor to log calls made to the database. This information can then be displayed in the profiler. A Hibernate interceptor can be used to get the information from the database queries. This can be an extension of the EmptyInterceptor so only necessary methods are implemented. The profiler will need to configure the environment to use the Interceptor. The work done by the interceptor should be done on the server side. Database query information will be passed back and displayed in the call tree by the profiler.

4.3 User Interaction

The GUI of the new profiler will be the same as the current profiler. It will have options to connect to different services and listen for calls made between the services and display the details on screen. It will also display calls made to the database from the services in the same window as an extension to the current call tree.

Queries to the database will contain different information compared to calls made between services. There could be extra information on the tables, attributes, values and other

information. This extra information could be shown on the side of the window, in the space under the different connections. There will still be a node for each query which will be included in the call tree. These database query nodes will include a button which allows the user to click to show the extra details on the side of the page.

This is better than including all the information in the call tree as it could result in the tree becoming too large and database query nodes taking up too much space. This will keep the tree looking tidy and allow the user to choose which query they want to look at.

Another idea to display the extra information was to allow the user to select a node they want to see and that node would expand to display any extra information. However, expanding the selected node would mean the call tree would need to be re-computed and redrawn each time a node is selected.

Scrolling may be enabled in the different frames if the information can't all fit into the given space. A mock-up of the new GUI is shown in (Figure 4.1) and (Figure 4.2) which shows how the window will change once a database query node is selected.

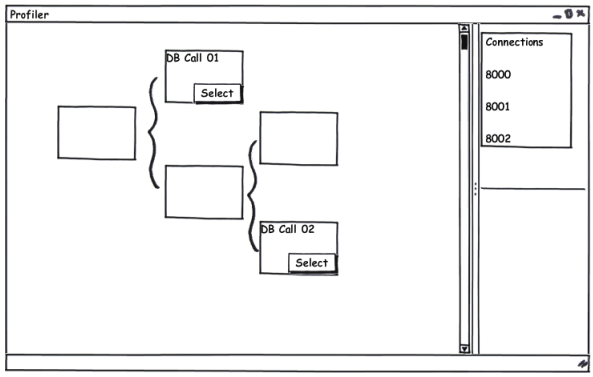

Figure 4.1: GUI Mock-Up Before Selecting Node

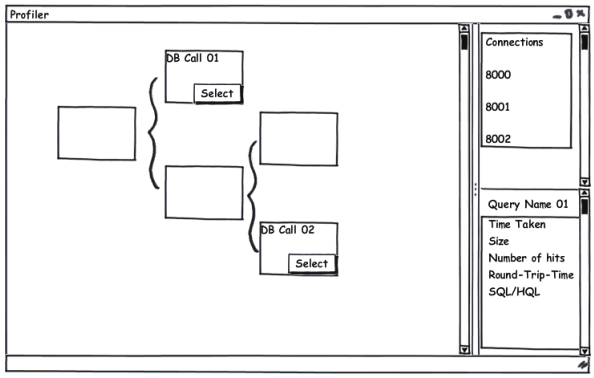

Figure 4.2: GUI Mock-Up After Selecting Node

4.4 Extra Optional Specifications

These are some extra specifications which could be included to improve the existing profiler. They are optional so will be added if I have time after creating the main database query interceptor.

Change current information in call nodes to display Average Time (currently Time Taken) and Total Time
Highlight the largest argument bytes for each call - show top 3
For long argument lists, show the largest argument (most important)
For database calls, display name, time, size, number of hits, round-trip-time, SQL/HQL
Order nodes by time - could try ordering nodes or changing the tree edges of the longest three to a different colour
Search nodes in call tree by method name

4.5 Project Plan

This is an approximate timeline for this project. I don't expect to follow it exactly and have been adding tasks to it as the project progressed. It is simply a guide to help stay on track and have a goal of what to accomplish.

Week 1-7: Planning and Research
- How to intercept calls made from service to database
- How the current profiler builds up the call tree
- How to configure the interceptor to work with the profiler

Week 7-16: Working Prototype and Research
- 7-9: Configure session to use interceptor to listen for database queries.
  - Implement extension of EmptyInterceptor class.
  - Application-scoped or session-scoped
  - Test on a table with simple queries
  - Insert, update, delete, etc
- 9-11: How to log the SQL and HQL
  - Look at Hibernate events
  - Test with some more complex queries
  - Getting the useful information out of the queries
- 11-12: Implement LoggingInterceptor for the current profiler
- 12-14: Implement LoggerManager to create and manage the interceptor
  - Grab required information from the database calls
  - Look at current profiler and decide the best way to store this
- 14-16: Extend current profiler to display information from database queries into window.
  - Create class to store information from database query
  - Query method name, time taken to process, hits etc
  - Display information as node on the call tree

Week 16-30: Completed Solution
    Fix issues that prototype had

Chapter 5

Research So Far

This chapter describes the research I have done for this project up to date. Each section includes research on different areas and topics which helped me to understand the project and get an idea of how to carry out the final implementation.

5.1 Distributed Service Profiler

The first step of the project involved me looking into the source code of the existing profiler to find out how it works. This is essential as I will need to eventually extend this to include the database call interceptor.

The profiler used a NotificationListener from the javax.management package. This listener is notified when a JMX (Java Management Extensions) notification occurs. The profiler then takes this Notification object to extract the message from it. The message is then split so information can be extracted. This information includes the method name, the service name, argument bytes, return bytes and the time taken. The hit count increments every time the same process is called.

All of this information is passed to the MethodCallContainer class which helps store this information for each method call. The ResultsPanel class can then grab the information from each call to create a method call node and paint this component onto the profiler window.

To keep track of which service calls which method, a method setInvokingIndex(i) is used to set which connection invoked that method, This is essential in building up the method call tree.

5.2 Hibernate

Historically, Hibernate facilitated the storage and retrieval of Java domain objects via Object/Relational Mapping. Today, Hibernate is a collection of related projects enabling developers to utilize POJO-style domain models in their applications in ways extending well beyond Object/Relational Mapping [4].

Hibernate is free and provides a framework to map Java classes to database tables. This means the object-oriented classes of a Java program can be mapped to traditional relational

model databases. This makes it possible to map Java data types to SQL data types. Hibernate can also be used for database queries by generating the SQL code. This makes it easier for developers as they dont have to manually change the data types and can easily create applications which are supported with SQL databases.

In the case of the services created by Kiwiplan, Hibernate also helps to create query strings when database calls are made and translate these to traditional MySQL queries.

5.3 Hibernate Interceptor

The Hibernate Interceptor is an interface in Hibernate which can be used in applications to react to certain events occurring inside Hibernate. This is the main tool I looked into and is what I will be using to extend the Distributed Service Profiler as it can be used to intercept calls made to a database.

The Hibernate Interceptor can be configured with an application and a database to intercept calls made to that database by the application. There can be two types of interceptors, global or session-scoped interceptors. Global interceptors are application-scoped. This means that when an application has several database sessions, the interceptor will affect objects in all sessions. Session-scoped interceptors, like the name indicates, are configured for each individual database session so they will only affect objects associated with that particular session.

Once an interceptor is configured and created, it will be invoked every time a insert, delete, update is made to the database. Methods in the interceptor can be modified to do different things based on the query. For example, if a user tries to insert an object into the database, the interceptor could be created such that it will check the properties to be saved to see whether they are valid. Another example could be for updates to the database, a log could be kept so each update is recorded to keep track of changes in the database.

There are three interfaces related to Interceptors are available in Hibernate, the Lifecycle and the Validatable interfaces and the Interceptor from the org.hibernate package [5].

The Lifecycle interface is used to encapsulate an objects phases of its lifecycle. Methods available from the Lifecycle interface include onLoad(), onSave(), onUpdate() and onDelete(). The Validatable interface only has a validate() method which is called during save operations to check the validity of the state of the object. The final Interceptor interface is what I am interested in.

5.4 EmptyInterceptor

The Interceptor interface is part of the org.hibernate package and includes over 15 methods. The EmptyInterceptor class implements the Interceptor interface and it typically what developers extend to implement their own customized interceptor. This allows the developer to implement only the necessary methods. Methods implemented in the interceptor are automatically invoked when a particular event occurs in the application.

A full list of the methods available from the Interceptor interface can be found on the Interceptor API page [6]. The methods I will be using include the onSave(), onDelete(), on-

Load(), onFlushDirty(), afterTransactionBegin() and afterTransactionCompletion() methods. The onSave(), onDelete(), onLoad() and onFlushDirty() methods are invoked when query to the database saves an object, deletes an object, reads an object data and updates an object respectively. The afterTransactionBegin() and afterTransactionCompletion() methods are invoked when a transaction begins and finishes so I will be using these methods to find the overall transaction time.

I know in Java there is a method System.currentTimeMillis() which return the current system time in milliseconds. This can be used in the afterTransactionBegin() and afterTransactionCompletion() methods and the difference taken to get the transaction time in milliseconds.

For a Hibernate Interceptor to work there must be at least two xml files to configure the interceptor. A hibernate.cfg.xml file is needed to configure the session factory with properties. These properties include the connection driver, url, username and password and other properties which can set the logged SQL to a formatted output. This file will also specify the mapping files which are used to map the Java classes to the database tables. This mapping file will be named [mappingClassName].hbm.xml and specify the attribute names and types.

I chose to use this to create my final solution as it is fairly straightforward to implement and can easily intercept database calls. It will take some work to integrate it with the current profiler but using the Hibernate Interceptor will make it easier to take care of the query interception part of the project. I will be extending the EmptyInterceptor class as I will only need a few specific methods and this will make it more flexible for me to choose which methods to include and which to ignore.

5.5 Logging

I also looked at how HQL (Hibernate query language) and SQL could be logged. I found something called SLF4J (Simple Logging Facade for Java) which serves as a simple facade or abstraction for various logging frameworks [7]. SLF4J is what Hibernate uses for logging SQL. However, it should have a binding framework such as Logback or Log4J.

I compared the difference between Logback and Log4J to see which binding I should use. Logback is basically the new version of Log4J and it seems pretty straightforward to use. I initially decided to use Logback with my interceptor. However, Kiwiplan typically uses Log4J so I decided to use Log4J for my final logging interceptor to keep things consistent.

Log4j is a tool used by developers to output log statements so that problem areas can be located. Logging behaviour can be controlled by using a configuration file and may be assigned to levels. The possible levels are TRACE, DEBUG, INFO, WARN, ERROR and FATAL. Since these levels are ordered, if the logging level is set to INFO, then log requests at or above this level (INFO, WARN, ERROR and FATAL) will be enabled and the log statements printed. Levels below the set logging level (in this case, TRACE and DEBUG) will be disabled and log statements of that level are not printed.

I am currently unsure about logging for HQL, it is hard to find sources which describe how HQL can be logged. Most articles I found were for SQL logging. I will need to do

further research for this.

5.6 Hibernate Query Language (HQL)

I have used SQL queries before but I had no previous experience with HQL so I had to do some research on that. HQL is similar to SQL but is fully object-oriented and understands notions like inheritance, polymorphism and association [8]. It is used to write queries which are similar to SQL queries but for Hibernate objects. I found that HQL is much easier to use than SQL. A simple example of HQL is ”from Contact”. This short query will return all instances of the class Contact. Compared to SQL, where the same query would need to be ”Select * from Contact”.

HQL also has a feature called criteria, which can be used to set restrictions to a query to select specific rows or attributes from a table. This makes it similar to object-oriented programming and can help make code neater when using long and complex queries which would require many ”where” and join conditions if traditional SQL was used.

5.7 Java MBean

MBeans are managed beans, Java objects that represent resources to be managed, and have a management interface [9]. MBeans represents objects or resources in Java and implement getter and setter methods for attributes and properties of that object. The special management interface associated with an MBean allows access to attributes, operations that can be invoked, notifications that can be emitted and constructors for that object. There are also four types of MBeans - Standard, Dynamic, Open and Model.

MBeans can be used to invoke operations on the application being monitored, as well as receive notifications about events. This is what the current Distributed Service Profiler does. A property change listener can also be implemented to detect changes made to a property of an object.

5.8 Database Changes Logger

The Database Changes Logger is a tool developed another BTech student a few years ago used to record changes to Hibernate databases used in Kiwiplan services. It uses JMX Mbeans and the Hibernate Interceptor to intercept database queries and is similar to what I need to create to extend to the Distributed Service Profiler.

The interceptor in this changes logger implements onSave(), onDelete() and onFlushDirty() and writes the changes to a log. The logging level can be changed to log a different amount of detail for each query (NONE, BASIC and FULL). This changes logger also uses LoggerManager class which manages the interceptor and creates it.

This changes logger will use configurations files to configure Hibernate if they are provided, otherwise default values are used. The logging levels to choose from are NONE, BASIC and FULL. If NONE is used, nothing will be logged. BASIC means all save and

delete operations will log a single line with the username, action, objects id and the objects toString() method result. Updates are only logged if there were actual changes. At the FULL logging level, all properties will be shown for save and delete operations and for updates, the object will be logged even if there are no changes made to it.

This changes logger uses the service name passed into the LoggingManager to associate each different logging manager with a particular service. This service name is then used to create service properties class which stores the properties for a service and provides methods for retrieving them. This class can then be used by the interceptor.

The LoggingManager class registers itself as an MBean, this means the logging level can be changed after initialisation using a JMX management application. Logged messages will be written to the appropriate log, specified by the configurations in the log4j.properties or log4.xml file.

Chapter 6

Work So Far

This chapter describes what I have actually implemented.

6.1 Simple Hibernate Interceptor

To experiment with a Hibernate Interceptor without having to integrate it with the service profiler, I implemented a simple Interceptor class which extends the EmptyInterceptor. In this Interceptor, I implemented the onSave(), onDelete(), onLoad(), onFlushDirty(), afterTransactionBegin() and afterTransactionCompletion() methods. At first, the onSave(), onDelete(), onLoad(), onFlushDirty() methods would only print out whether it was saving, deleting, updating or loading an object. This was just to let me see whether the interceptor was working and whether the methods were being invoked at the right time.

I also created the hibernate.cfg.xml and mapping file to configure the interceptor and map it to the MySQL table. To test this interceptor, I created a Contact class in Java which has attributes id, first name, last name and email. I also created a main class HibernateTest which configures the session factory and creates an application-scoped Interceptor. In here, I created a few Contact objects and saved, deleted, updated and read them from the database to check if the Interceptor worked.

The hibenate.cfg.xml file contains information on the properties needed to configure the session factory for Hibernate to use. These properties include the connection driver class, connection url, dialect, connection username and password and the names of the mapping files. There can be a lot of other properties to configure the session factory but many of them are not necessary for the program to work. The properties I used in this simple interceptor test were the minimal amount needed to get it working. Once I start to integrate the real interceptor with the Distributed Service Profiler, I will probably need a lot more information. I expect most of this information can be obtained from Kiwiplan as they already use Hibernate for their code.

The same can be said for the mapping files. Since mapping files include information to map the attributes from a Java class to a database table, I would expect I can get this information from Kiwiplan otherwise it would be extremely difficult for me to create one from scratch.

I also added a SLF4J with Logback into this application and tried the different logging levels. I used this in the Interceptor class to print out when a method is invoked as well the entity class (in this case, Contact) and the id.

Up until this point, the afterTransactionBegin() and afterTransactionCompletion() methods were empty. I added a System.currentTimeMillis() in afterTransactionBegin() and stored this in a global variable. When afterTransactionCompletion() is invoked, System.currentTimeMillis() is called again and the difference is taken between the returned value and the stored time from the beginning of the transaction. This is the total transaction time in milliseconds.

Once the Interceptor was working with the saves and deletes, I decided to test it with some more complicated queries. I had only been using session.save(c) or session.delete(c) to test it so far. I used some HQL and SQL queries to select certain contacts from the Contact table. This included using the HQL criteria and setting parameters into the queries to run on the database. I also created traditional SQL query strings. Both were fine in running on the database and the interceptor successfully intercepts the queries and extracts the information from the queries.

6.2 Logging Interceptor for Profiler

It was good that I had created the simple Interceptor for testing as it allowed me to understand how the Hibernate interceptor works and reuse the same format and some of the code from that for the actual LoggingInterceptor class. I also created a LoggerManager class, similar to the one in the database changes logger application, to manage and create the LoggingInterceptor.

The LoggingInterceptor is initialised with a service name and the LoggingInterceptor class has the onSave(), onDelete(), onLoad(), onFlushDirty(), afterTransactionBegin() and afterTransactionCompletion() methods.

I also included methods in the LoggingInterceptor which are called when the onSave(), onDelete(), onLoad() and onFlushDirty() methods are invoked. These methods take in the parameters passed into the methods which have information on the entity being passed to the database for saving, deleting, reading or updating. This information can be used in interceptors to validate or change the incoming information before passing it on. I simply take these parameters to format them and print out the entity being changed and its properties.

Currently, the LoggingManager class only takes in a service name and creates a LoggingInterceptor associated with that service name. I will be adding more details into this class as I integrate it with the current profiler.

Chapter 7

Future Work

This chapter describes what I work I plan to do next.

7.1 Logging Interceptor for Profiler

I will need to continue implementing the LoggingInterceptor. With the main database query interception mostly done, I will need to extract the useful information from these queries and store them so that they can be displayed on the profiler window. This will need to be set up so that the Distributed Service Profiler can use this information to draw the call nodes and include it in the call tree. Since information from these database calls will be different than the typical service calls, I may need to implement a new class which extends the one used in the profiler to make it more appropriate for the database queries.

I will also need to find out how to combine this interceptor with the profiler. My previous tests with the simple interceptor were done with a simple test class which creates the new SessionFactory and configures the interceptor. Since the Distributed Service Profiler is a lot more complex, I will need to find out how I can use that to configure the sessions and use the LoggingInterceptor. To make the interceptor work, the hibernate.cfg.xml and mapping files are also needed, so I will need to get the right files used by Kiwiplan to configure my interceptor.

Since the nodes required to show the database calls will need to be different to include a button, I will need to incorporate that into the profiler. I will need to look at how the nodes are created and painted in the Distributed Service Profiler and use that format to keep things consistent.

7.2 Distributed Service Profiler Source Code

The next stage of this project will require a lot of time spent looking at the source code of the Distributed Service Profiler. I had read over some of the classes in the research phase in the beginning of the project. However, that was just to get a general idea of how the profiler runs and what it does. I will need to look at the smaller details if I am going to integrate my code with the profiler.

The main parts I will need to look at are how the service method call information is stored, how the call tree is built up and how the nodes of the call tree are created and painted. I will need to make changes to the code so that the new database query nodes can be drawn as part of the call tree.

Once I have my interceptor fully working and integrated with the Distributed Service Profiler, I can start looking at how I can improve the actual profiler with the extra specifications included in the development plan.

7.3 Maven

Kiwiplan uses a tool called Maven which is used to build and manage Java projects. I will need to learn how to use this and use it to finish the extended profiler.

7.4 Conclusion

There is still a lot of work to be done for this project. The next phase of the project will include a lot more coding and less research than this first semester. I have made a lot of progress this semester and have a much clearer idea of how to continue and complete the project.

Bibliography

[1] "Kiwiplan — contact info." http://www.kiwiplan.com/site_about/index.cfm?abbr=en, 2011.

[2] "Profiling (computer programming)." http://en.wikipedia.org/wiki/Profiling_(computer_programming), 2011.

[3] "Java profiler - jprofiler." http://www.ej-technologies.com/products/jprofiler/overview.html, 2011.

[4] "Hibernate - jboss community." http://www.hibernate.org/, 2011.

[5] Raja, "Introduction to interceptors in hibernate orm framework." http://www.javabeat.net/articles/9-interceptors-in-hibernate-orm-framework-an-introducti-1.html, 2011.

[6] "Interceptor (hibernate api documentation)." http://www.dil.univ-mrs.fr/~massat/docs/hibernate-3.1/api/org/hibernate/Interceptor.html, 2011.

[7] "Simple logging facade for java (slf4j)." http://www.slf4j.org/, 2011.

[8] "Hql: The hibernate query language." http://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html, 2011.

[9] "Overview of monitoring and management." http://download.oracle.com/javase/1.5.0/docs/guide/management/overview.html, 2011.