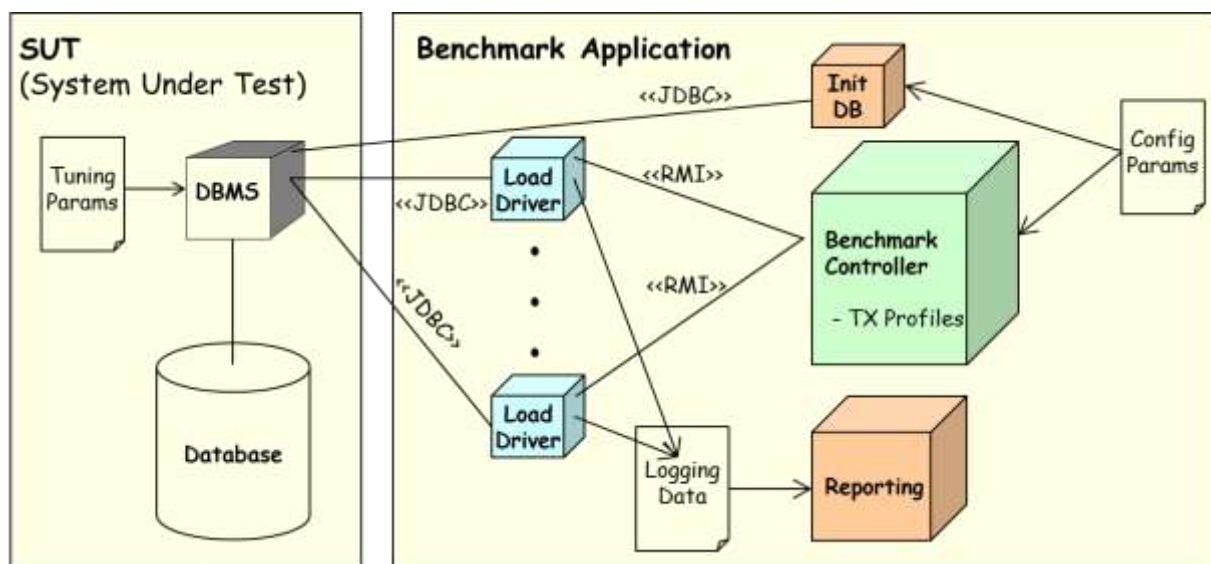


Ein Java-Framework zum Datenbank-Benchmarking



Inhaltsverzeichnis

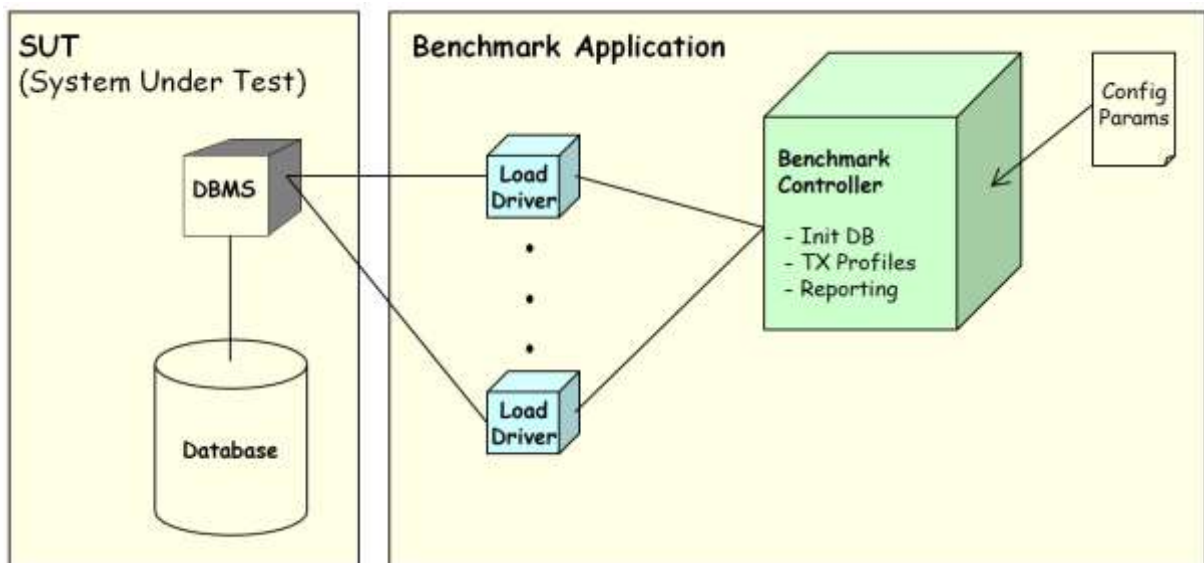
Einführung	3
Architekturüberblick zum Framework	6
Grundlegende Klassen und Interfaces	6
Verteiltes System mittels Java Remote Method Invocation	8
Konfigurationseinstellungen	9
Initialisierung der Benchmark-Datenbank	11
Reporting und Logging	12
Überblick über die Implementierung.....	15
Paket Benchmark.....	15
Paket LoadDriver	15
Paket Transactions	16
Paket Statistics	16
Paket Util	17
Installation und Einsatz des Frameworks.....	18
Check-Out als Eclipse-Projekte.....	18
Anpassungen an konkreten Benchmark und konkretes DBMS.....	21
Durchführung eines Benchmark-Laufs.....	21
Zusammenfassung und Ausblick	24
Anhang	25

Allgemeine Einführung

Die Begriffe „**Datenbank-Benchmarking**“ bzw. „Datenbank-Benchmark“ (kurz: DB-Benchmark) bezeichnen Verfahren zur Leistungsbewertung von Datenbankmanagementsystemen (kurz: DBMSs) mit Hilfe eines Vergleichs zu messender Leistungsmerkmale unter genau vorgegebenen Last-situationen.

Zur einfachen und wiederholten Benchmark-Durchführung werden solche DB-Benchmarks selbst als Anwendungssysteme (d.h. als DB-Benchmark-*Programme*) realisiert, die die wesentlichen Aspekte des Benchmarkings automatisieren (bspw. Erzeugung einer zugrunde liegenden initialen Benchmark-Datenbank, Lasterzeugung, Messung der Leistungsmerkmale, Benchmark-Auswertung usw.). Dieses Dokument beschreibt ein Framework, das die Entwicklung und Implementierung solcher DB-Benchmark-Programme für die Programmiersprache Java unterstützen und vereinfachen soll.

Wenn man von den jeweils sehr spezifischen Details konkreter Benchmark-Verfahren einmal absieht, dann sind solche DB-Benchmarks in der Regel strukturell gleichartig oder zumindest ähnlich aufgebaut. Noch sehr allgemein und auf einem hohen Abstraktionsniveau kann dieser Aufbau eines DB-Benchmarks durch das folgende UML-Verteilungsdiagramm beschrieben werden:



Links dargestellt ist das zu untersuchende bzw. zu bewertende Datenbankmanagementsystem, das in diesem Zusammenhang auch „**System under Test**“ (kurz: SUT) genannt wird. Es läuft in der Regel auf einem eigenen Rechner und besitzt eine zugehörige initiale Benchmark-Datenbank zu einem durch den Benchmark genau vorgegebenen Datenbankschema. Die Größe der zu nutzenden initialen Datenbank (mit im Allgemeinen künstlich generierten Anwendungsdaten!) kann üblicherweise durch einen wählbaren Skalierungsfaktor für jede Benchmark-Durchführung individuell konfiguriert werden.

Auf der rechten Seite ist das Benchmark-Anwendungssystem als *verteiltes* System dargestellt. Die Last wird durch einen oder mehrere „**Load Driver**“ generiert, die gleichzeitig bzw. nebenläufig

entsprechende Transaktionen (kurz: TXs) als Lastoperationen auf dem Datenbankmanagementsystem ausführen. Die Koordination und Steuerung in diesem verteilten System übernimmt ein zentraler „**Benchmark Controller**“, der über entsprechende Konfigurationsparameter eingestellt werden kann. Diese Konfigurationsparameter bestimmen u.a. die Größe der initialen Benchmark-Datenbank, die Anzahl der Load Driver, ggf. ihre Verteilung auf verfügbare Rechner sowie die durchzuführenden Transaktionen und ihre jeweiligen Gewichtungen.

Zur Auswertung (engl. **Reporting**) einer Benchmark-Durchführung müssen vorgegebene Leistungsmerkmale gemessen werden. Bspw. ist häufig die durchschnittliche Laufzeit der unterschiedlichen Transaktionstypen zu ermitteln oder die absolute oder durchschnittliche Anzahl durchgeführter Transaktionen innerhalb einer vorgegebenen Zeitspanne. Diese Messungen werden üblicherweise durch das Benchmark-Anwendungssystem selbst durchgeführt, d.h. die Messwerte werden als Leistungsmerkmale *clientseitig* durch die Load Driver oder den Benchmark Controller und *nicht serverseitig* direkt im DBMS ermittelt.

In der Diagrammdarstellung sind die einzelnen Bestandteile der Benchmark-Anwendung jeweils auf eigene „Knotenrechner“ verteilt. Dies stellt eine *logische* Verteilung (engl. **Deployment**) dar, d.h. bei einer konkreten Benchmark-Durchführung können durchaus mehrere Bestandteile und damit mehrere logische Knoten auf einem physisch vorhandenen Rechner zusammengefasst werden¹.

Auch wenn sich unterschiedliche Benchmark-Verfahren im Detail stark voneinander unterscheiden, so führt doch der skizzierte gleichartige Aufbau dazu, dass Benchmark-Programme immer auch gleichartige oder ähnliche Bestandteile zur Realisierung der folgenden gemeinsamen Aspekte enthalten:

- Kommunikation im verteilten System
- Koordination durch den zentralen Benchmark Controller
- Lastgenerierung durch verteilte Load Driver
- clientseitige Messung und Auswertung interessierender Leistungsmerkmale
- Konfiguration einer Benchmark-Durchführung über einstellbare Parameter.

Statt diese Aspekte für jedes konkrete Benchmark-Verfahren jeweils von Grund auf neu zu implementieren, erscheint es sinnvoller zu sein, ein **Benchmarking-Framework** zu entwickeln, das diese gemeinsamen Bestandteile Benchmark- und DBMS-übergreifend sehr allgemein, erweiterbar und konfigurierbar zur Verfügung stellt. Ein solches Framework soll dann den Erstellungsaufwand für einen konkreten Benchmark und ein konkretes DBMS jeweils erheblich reduzieren.

¹ Im Extremfall können SUT und Benchmark-Anwendungssystem sogar komplett auf *ein und demselben* Rechner laufen, auch wenn dadurch natürlich die Lastgenerierung das Leistungsverhalten des DBMS direkt beeinflusst, was bei einer wirklichen Benchmark-Messung definitiv nicht wünschenswert erscheint. Ein solch triviales Deployment vereinfacht aber möglicherweise die Entwicklungsphase einer Benchmark-Anwendung und wird deshalb während der Entwicklung tatsächlich häufig genutzt!

Das im Folgenden beschriebene Java-Framework wurde am Softwaretechniklabor der Westfälischen Hochschule am Campus Bocholt entwickelt¹ und wird dort insbesondere im Rahmen des Praktikums zur Lehrveranstaltung „Datenbanken und Informationssysteme“ eingesetzt.

Das vorliegende Dokument soll den Einsatz des Frameworks erleichtern. Es ist folgendermaßen aufgebaut: Im nächsten Kapitel wird zunächst ein noch sehr grober Überblick über die gesamte Framework-Architektur gegeben. Anschließend werden einzelne Komponenten (entsprechend der in der Implementierung gewählten Java-Paketstruktur) jeweils im Detail erläutert! Den Abschluss bilden ausführliche Hinweise zur Installation und zum Einsatz des Frameworks. Im Anhang befindet sich zudem eine Javadoc-Dokumentation zum Framework.

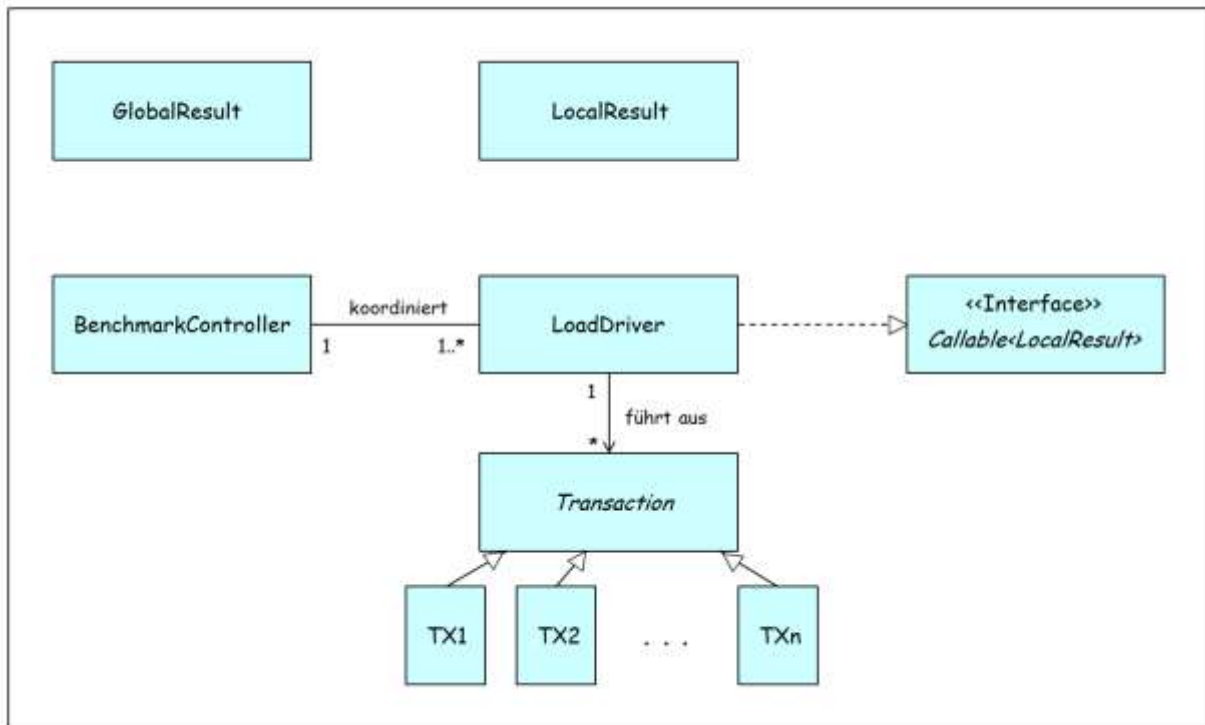
¹ Die Entwicklung, die im Wesentlichen durch frühere Studenten (Tobias Theophile, Max Lübbering, Dennis Effing) durchgeführt wurde, ist noch nicht komplett abgeschlossen! Hinweise auf Fehler, gewünschte Änderungen oder sinnvolle Erweiterungen sind immer willkommen und sollten bitte per Mail an bernhard.convent@w-hs.de gesendet werden.

Architekturüberblick zum Framework

Das Framework implementiert ein allgemeines, erweiterbares und konfigurierbares Benchmark-Anwendungssystem und konkretisiert damit viele der noch sehr allgemeinen Vorgaben aus dem obigen Verteilungsdiagramm.

Grundlegende Klassen und Interfaces

Um einen ersten Einstieg zu ermöglichen, zeigen wir zunächst die wichtigsten Klassen zusammen mit ihren Assoziationen innerhalb eines UML-Klassendiagramms:



Im Zentrum des Klassendiagramms stehen die beiden Klassen **BenchmarkController** und **LoadDriver**. Die **Klasse BenchmarkController** realisiert den zentralen Benchmark Controller aus dem Verteilungsdiagramm (siehe Kapitel „Allgemeine Einführung“, Seite 3), der einen oder mehrere Load Driver koordiniert. Diese Load Driver werden durch den Benchmark Controller zu Beginn einer Benchmark-Durchführung über den Start eines Benchmark-Laufs informiert und nach dessen Beendigung wieder gestoppt, und sie liefern am Ende jeweils aggregierte Messergebnisse als Objekt der **Hilfsklasse LocalResult** an den Benchmark Controller zurück. Die erhaltenen LocalResults werden vom Benchmark Controller zu einem Objekt der **Hilfsklasse GlobalResult** zusammengefasst und im Sinne eines noch sehr groben Reportings als Gesamtergebnis ausgegeben.

Die **Klasse LoadDriver** realisiert einzelne Load Driver, die nebenläufig in einem eigenen Thread ablaufen sollen. Dazu implementiert die Klasse das **Interface Callable** aus dem Paket

java.util.concurrent der Java-Standardbibliothek¹ mit der Klasse `LocalResult` als generischen Parameter für den Ergebnistyp. Jedes `LoadDriver`-Objekt wählt zur Benchmark-Durchführung (entsprechend den vorgegebenen Gewichtungen aus dem konfigurierten TX-Profil) einzelne Transaktionen aus einer Menge vorhandener Lasttransaktionen aus, führt diese nacheinander auf dem DBMS aus und erfasst und protokolliert die zugehörigen Laufzeiten in einem `LocalResult`-Objekt.

Die **abstrakte Klasse `Transaction`** kapselt einzelne Transaktionstypen aus dem Lastprofil und bietet dazu eine einheitliche Schnittstelle für die benötigten Datenbankzugriffe an. Diese Schnittstelle enthält einerseits allgemeine und *Benchmark-übergreifend* einsetzbare Operationen bspw. zum Starten, zum erfolgreichen Beenden und auch zum Rollback einer Transaktion. Diese Operationen sind alle innerhalb der `Transaction`-Klasse unter Nutzung des JDBC-APIs bereits vollständig ausprogrammiert. Andererseits enthält die Klasse auch eine abstrakte Methode `run()`, die eine *Benchmark-spezifische* Lasttransaktion realisieren soll und die deshalb in konkreten Unterklassen noch ausprogrammiert werden muss.

Diese Ausprogrammierung der eigentlichen Lasttransaktionen erfolgt demnach innerhalb einer Menge von konkreten **Klassen `TX1`, ..., `TXn`** als abgeleitete und *selbst zu implementierende* Unterklassen von `Transaction`². Gleichzeitig können in diesen Unterklassen bei Bedarf die ererbten und bereits ausprogrammierten Methoden von `Transaction` überschrieben werden, um die allgemeine JDBC-Implementierung aus `Transaction` an ein konkret gewähltes DBMS anzupassen unter Nutzung evtl. vorhandener DBMS-Spezifika. D.h. die Unterklassen von `Transaction` stellen die vorgesehene Möglichkeit im Framework dar, um an *einer lokalen Stelle* sowohl die Benchmark-spezifischen Lasttransaktionstypen zu realisieren als auch notwendige DBMS-spezifische Anpassungen vorzunehmen³!

¹ Zu den Grundlagen der Java-Thread-Programmierung siehe ins offizielle Java-Tutorial <http://docs.oracle.com/javase/tutorial/essential/concurrency/> oder in entsprechende Java-Lehrbücher wie bspw. J. Hettel, M.T. Tran, Nebenläufige Programmierung mit Java, dpunkt.verlag, 2016.

² Die Bezeichner dieser Unterklassen sind übrigens frei wählbar und werden nicht vom Framework vorgegeben! Häufig wird auch von der Spezialisierung **`WeightedTransaction`** (statt von der Klasse `Transaction` selbst) abgeleitet. Diese Spezialisierung verwaltet die Gewichtungen der einzelnen Transaktionstypen gleich mit.

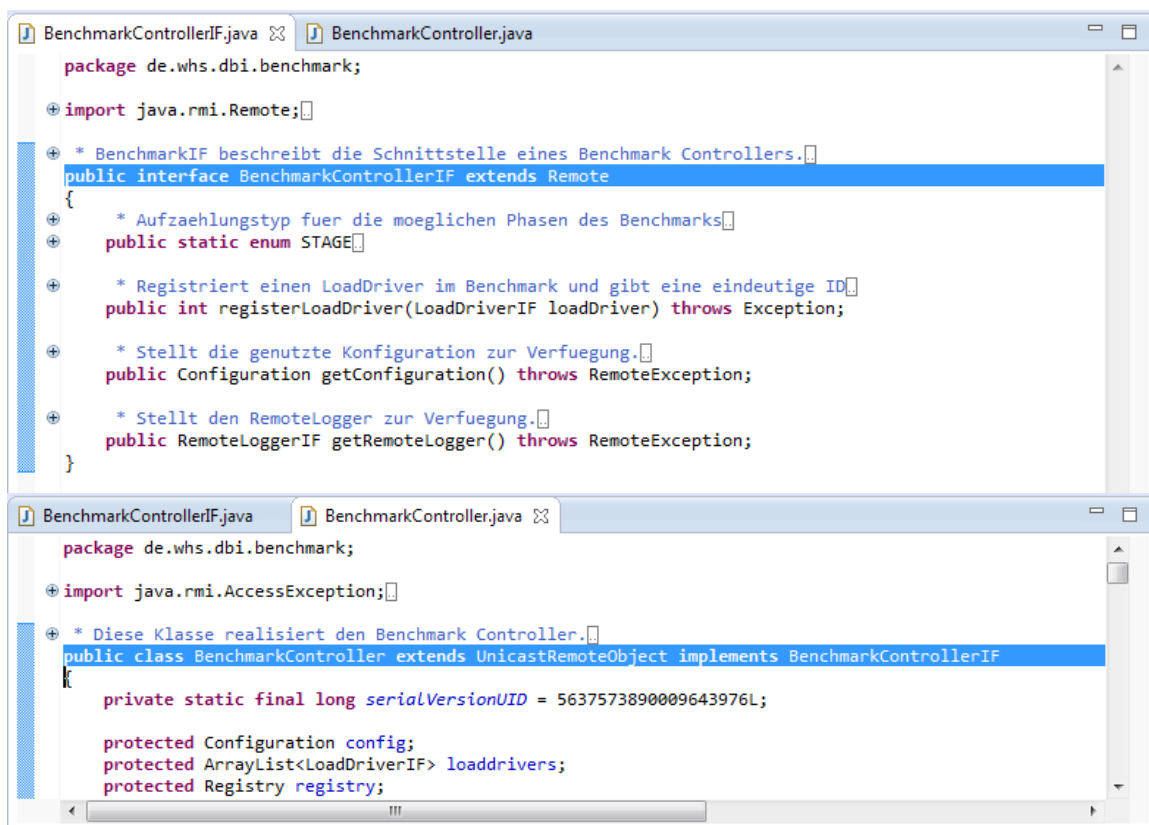
³ Alle anderen Klassen des Frameworks bleiben im Idealfall bei der Realisierung eines konkreten DB-Benchmarks für ein spezielles DBMS unverändert! Dieser idealisierte Anspruch wird realistischerweise nicht immer völlig durchzuhalten sein (vgl. auch die später einzuführende Klasse `InitMyDatabase` im Abschnitt „Initialisierung der Benchmark-Datenbank“). Aber dennoch soll zumindest der Hauptanteil notwendiger Anpassungen und Erweiterungen an konkrete Benchmarks oder spezielle DBMSs definitiv *lokal* innerhalb der Unterklassen von `Transaction` erfolgen!

Verteiltes System mittels Java Remote Method Invocation

Schon das UML-Verteilungsdiagramm aus dem Einführungskapitel stellte die Benchmark-Anwendung als verteiltes System dar, das aus mehreren, potenziell auf unterschiedlichen Knotenrechnern laufenden Komponenten besteht, die miteinander kommunizieren und gegenseitig bei Bedarf die Dienste der jeweils anderen Komponenten nutzen.

Zur Realisierung bietet Java dazu mit der **Remote Method Invocation (RMI)**¹ eine eigene Variante für sogenannte „Remote Procedure Calls“ an. Diese ermöglicht eine einfache Kommunikation zwischen verteilten Java-Objekten durch Methodenaufrufe über die Grenzen der jeweils eigenen Java Virtual Machine hinweg bei weitgehender Transparenz der Verteilung für den Programmierer. Das ermöglicht eine einfache Programmierung auf einem sehr hohen Abstraktionsniveau.

Die **Schnittstelle einer RMI-fähigen Klasse** wird dazu als Java Interface festgelegt, wobei dieses Interface von `java.rmi.Remote` abgeleitet sein muss. Die konkrete **Realisierung einer RMI-fähigen Klasse** implementiert das jeweilige Remote-Interface und ist dabei selbst von der RMI-Systemklasse `java.rmi.server.UnicastRemoteObject` abgeleitet. In unserem Benchmark-Framework gilt dies bspw. für die beiden Klassen `BenchmarkController` und `LoadDriver`, die natürlich RMI-fähig sein müssen. Für den Benchmark Controller und dessen Klasse `BenchmarkController` wird dies beispielhaft durch den folgenden Quelltextauszug demonstriert.



```
package de.whs.dbi.benchmark;

import java.rmi.Remote;

/* BenchmarkIF beschreibt die Schnittstelle eines Benchmark Controllers. */
public interface BenchmarkControllerIF extends Remote
{
    /* Aufzählungstyp fuer die moeglichen Phasen des Benchmarks */
    public static enum STAGE

    /* Registriert einen LoadDriver im Benchmark und gibt eine eindeutige ID */
    public int registerLoadDriver(LoadDriverIF loadDriver) throws Exception;

    /* Stellt die genutzte Konfiguration zur Verfuegung. */
    public Configuration getConfiguration() throws RemoteException;

    /* Stellt den RemoteLogger zur Verfuegung. */
    public RemoteLoggerIF getRemoteLogger() throws RemoteException;
}

package de.whs.dbi.benchmark;

import java.rmi.AccessException;

/* Diese Klasse realisiert den Benchmark Controller. */
public class BenchmarkController extends UnicastRemoteObject implements BenchmarkControllerIF
{
    private static final long serialVersionUID = 5637573890009643976L;

    protected Configuration config;
    protected ArrayList<LoadDriverIF> loaddrivers;
    protected Registry registry;
```

¹ Zu den Grundlagen der Java Remote Method Invocation siehe ins offizielle Java-Tutorial <http://docs.oracle.com/javase/tutorial/rmi/> oder wieder in entsprechende Java-Lehrbücher für Fortgeschrittene.

Um eine Ortstransparenz bei der Programmierung zu ermöglichen, bietet RMI mit der **RMI-Registry** einen einfachen Namensdienst¹ an, bei dem RMI-fähige Objekte zur Laufzeit unter einem symbolischen Namen registriert werden können (engl. **Bind**). Anschließend können diese RMI-Objekte (im Sinne der Ortstransparenz unabhängig von der genauen Kenntnis ihres tatsächlichen Ortes!) nur mit Hilfe ihres Namens gesucht und gefunden werden (engl. **Lookup**).

Im Benchmark-Framework startet der Benchmark Controller auch die RMI-Registry (und zwar auf dessen lokalem Rechner) und registriert sich dort selbst. Alle anderen RMI-Objekte wie bspw. die Load Driver greifen mittels Lookup zunächst auf das registrierte BenchmarkController-Objekt zu und erhalten vom ihm alle weiteren benötigten Informationen. Damit dies möglich wird, benötigen alle RMI-Komponenten Kenntnis der genauen **Adresse der RMI-Registry**. Diese ist daher allen Komponenten als Konfigurationseinstellung zur Verfügung zu stellen.

Konfigurationseinstellungen

Variable Eigenschaften eines konkreten Benchmark-Laufs (wie bspw. die Adresse der RMI-Registry oder die genaue Anzahl zu nutzender Load Driver) sollten über **einstellbare Konfigurationsparameter** festlegbar sein. In einem verteilten System, wie es durch das Benchmark-Framework realisiert wird, existieren in der Regel eine Reihe von globalen Konfigurationsparametern, die von *zahlreichen* verteilten Komponenten des Systems benötigt werden.

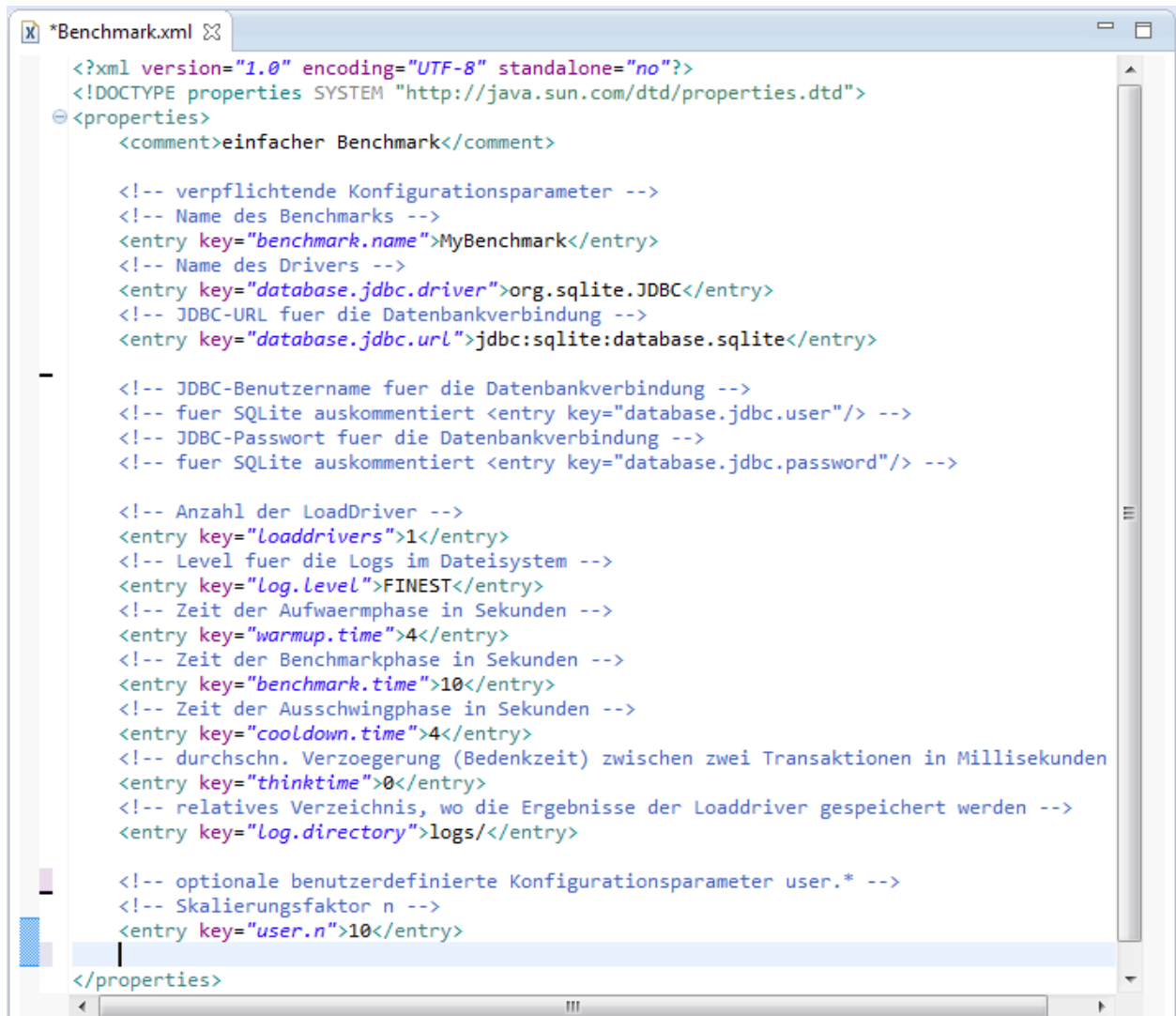
Will man sicherstellen, dass alle Komponenten konsistent *dieselben* Konfigurationsparameter nutzen, so erscheint es sinnvoll, dass diese Parameter möglichst nur an einer Stelle festgelegt und vorgehalten werden, um dann als „**RMI-Konfigurationsobjekte**“ an alle Komponenten verteilt zu werden. Im Benchmark-Framework bietet sich dafür direkt der zentrale Benchmark Controller an, der Konfigurationsparameter einlesen und im System zur Verfügung stellen kann.

Die Konfigurationseinstellungen werden im Benchmark-Framework über die **Klasse Configuration** bereitgestellt, die alle Einstellungen in einem einzelnen Objekt repräsentiert. Für die Eingabe wird die vollständige Konfiguration in drei Teile (Benchmark, Transactions und Communication) aufgeteilt und in separaten Dateien im XML-Format der Java-Properties² gespeichert. Diese Dateien müssen zur Laufzeit dem Benchmark Controller zum Einlesen zur Verfügung stehen!

Der Hauptteil der Konfiguration befindet sich in einer **Datei Benchmark.xml**. Dort werden alle relevanten allgemeinen Informationen festgehalten, die für die Ausführung des Benchmarks benötigt werden. Die folgende Abbildung zeigt eine typische Eingabe mit Beispielwerten zusammen mit der genutzten Syntax:

¹ Zum Konzept eines Namensdienstes in verteilten Systemen siehe bspw. A.S. Tanenbaum, M. van Steen, Distributed Systems – Principles and Paradigms, 2nd ed., Prentice Hall, 2006.

² Siehe java.util.Properties aus dem Java-Standard-API.



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <comment>einfacher Benchmark</comment>

  <!-- verpflichtende Konfigurationsparameter -->
  <!-- Name des Benchmarks -->
  <entry key="benchmark.name">MyBenchmark</entry>
  <!-- Name des Drivers -->
  <entry key="database.jdbc.driver">org.sqlite.JDBC</entry>
  <!-- JDBC-URL fuer die Datenbankverbindung -->
  <entry key="database.jdbc.url">jdbc:sqlite:database.sqlite</entry>

  <!-- JDBC-Benutzername fuer die Datenbankverbindung -->
  <!-- fuer SQLite auskommentiert <entry key="database.jdbc.user"/> -->
  <!-- JDBC-Passwort fuer die Datenbankverbindung -->
  <!-- fuer SQLite auskommentiert <entry key="database.jdbc.password"/> -->

  <!-- Anzahl der LoadDriver -->
  <entry key="loaddrivers">1</entry>
  <!-- Level fuer die Logs im Dateisystem -->
  <entry key="log.level">FINEST</entry>
  <!-- Zeit der Aufwaermphase in Sekunden -->
  <entry key="warmup.time">4</entry>
  <!-- Zeit der Benchmarkphase in Sekunden -->
  <entry key="benchmark.time">10</entry>
  <!-- Zeit der Ausschwingphase in Sekunden -->
  <entry key="cooldown.time">4</entry>
  <!-- durchschn. Verzoegerung (Bedenkzeit) zwischen zwei Transaktionen in Millisekunden -->
  <entry key="thinktime">0</entry>
  <!-- relatives Verzeichnis, wo die Ergebnisse der Loaddriver gespeichert werden -->
  <entry key="log.directory">logs</entry>

  <!-- optionale benutzerdefinierte Konfigurationsparameter user.* -->
  <!-- Skalierungsfaktor n -->
  <entry key="user.n">10</entry>
</properties>
```

Eine weitere **Konfigurationsdatei Transactions.xml** legt das genaue Lastprofil eines Benchmark-Laufs fest. Beide Konfigurationen werden zentral beim Benchmark Controller verwaltet, d.h. sie werden zur Laufzeit vom Benchmark Controller an die einzelnen Load Driver übermittelt und müssen dadurch nur an einer Stelle angepasst und verwaltet werden. Auch hierzu zeigen wir eine einfache Template-Datei mit nur zwei Transaktionen mit jeweils unterschiedlichen relativen Gewichten im Lastprofil:



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <comment>Transactions</comment>

  <!-- Aufzaehlung aller Transaktionen des Lastprofils -->
  <!-- angegeben wird jeweils der qualifizierte Klassenname und
  eine relative Gewichtung innerhalb des Lastprofils -->
  <entry key="benchmark.mybenchmark.SelectTx">10</entry>
  <entry key="benchmark.mybenchmark.UpdateTx">1</entry>
</properties>
```

Die dritte und letzte **Konfigurationsdatei Communication.xml** wird von *allen* Komponenten benötigt, da dort die Adresse der RMI-Registry festgelegt wird. Auch hier zeigen wir der Vollständigkeit halber eine Template-Datei mit entsprechenden Standardwerten für eine lokale Registry:



Initialisierung der Benchmark-Datenbank

Entsprechend dem UML-Verteilungsdiagramm aus dem Einführungskapitel ist der Aufbau der initialen Benchmark-Datenbank eine Aufgabe des Benchmark Controllers, die dieser – genauso wie die Transaktionslast bei einer Benchmark-Durchführung – prinzipiell an *mehrere nebenläufige* Load Driver verteilen kann! Im konkreten Framework wurde diese (für das Benchmarking nicht ganz so wichtige!) Aufgabe aus der Klasse für den Load Driver „herausgelöst“ und einfacher (und durch einen Einzel-Thread auch etwas weniger allgemein!) direkt realisiert.

Eine eigene **Klasse InitMyDatabase** bietet in ihrer Main-Methode eine Grundstruktur für einen einfachen Aufbau der Benchmark-Datenbank mit entsprechend *auszuprogrammierenden* Hilfsmethoden an (für den Verbindungsauf- und -abbau, für die Datenbankschemadefinition, für eine rudimentäre Zeitmessung usw.). Diese Klasse dient wieder als einfaches „Template“ und ist daher bei der Framework-Nutzung jeweils DBMS- und Benchmark-spezifisch zu vervollständigen und zu erweitern. Alle Hilfsmethoden für die Einzelschritte bei der Initialisierung der Benchmark-Datenbank sind als „static“ deklarierte Klassenmethoden. Die folgende Abbildung zeigt die vorgegebene Main-Methode der Klasse InitMyDatabase, die diese Hilfsmethoden in einer passenden Reihenfolge nacheinander aufruft.

```
*InitMyDatabase.java
public class InitMyDatabase
{
    protected static Connection connection;
    protected static Configuration config;

    protected static void openConnection() throws Exception
    protected static void closeConnection() throws SQLException
    protected static void dropTables() throws SQLException
    protected static void createTables() throws SQLException
    protected static void tuneDatabase() throws SQLException
    protected static void insertRows() throws SQLException

    public static void main(String[] args)
    {
        try
        {
            System.out.println("Initialisierung der Datenbank:\n");

            // Lädt die Konfiguration
            config = new Configuration();
            config.loadBenchmarkConfiguration();

            openConnection();
            System.out.println("Löschen vorhandener Relationen ...");
            dropTables();
            System.out.println("Anlegen des Datenbankschemas ...");
            createTables();
            System.out.println("Durchführung von Tuning-Einstellungen ...");
            tuneDatabase();
            System.out.println("Einfügen der Tupel ...\n");

            // Merken des Start-Zeitpunktes der Initialisierung
            long duration = System.currentTimeMillis();

            insertRows();

            // Berechnung der Zeitdauer der Initialisierung
            duration = System.currentTimeMillis() - duration;

            closeConnection();

            System.out.println("Einfuegedauer: " + duration / 1000 + " Sekunden\n");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Der Quelltextauszug zeigt, dass diese Klasse primär für die Datenbankschemadefinition und die Füllung der Tabellen zuständig sein soll. Evtl. zu nutzende DBMS-spezifische **Tuning-Einstellungen** können innerhalb der `tuneDatabase()`-Hilfsmethode (oder auch zusätzlich direkt am DBMS!) eingegeben werden.

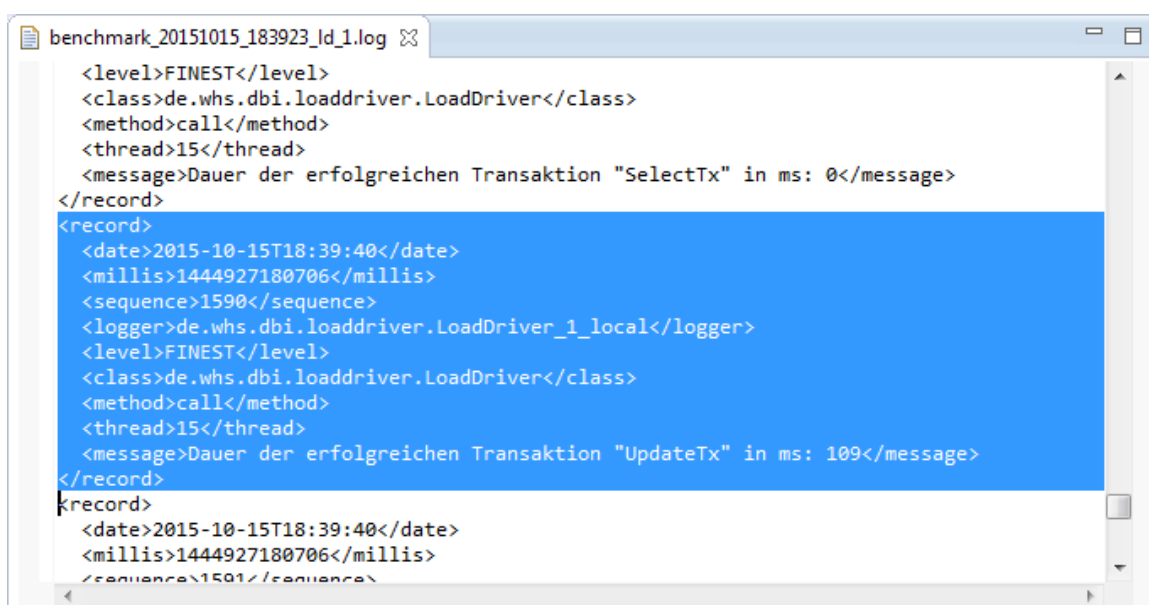
Reporting und Logging

Auch die Auswertung einer Benchmark-Durchführung, das sogenannte Reporting, ist nach dem einführenden UML-Verteilungsdiagramm zunächst eine direkte Aufgabe des Benchmark Controllers.

Ein erstes **grobes Reporting in der Form statistischer Kennzahlen** (wie bspw. die Bestimmung der Anzahl durchgeführter Transaktionen oder die Bestimmung der durchschnittlichen Dauer eines jeden Transaktionstyps) kann tatsächlich leicht bereits im Benchmark Controller erfolgen, indem hier

entsprechende Zwischenergebnisse der einzelnen Load Driver zusammengefasst werden, wobei diese Zwischenergebnisse dem Benchmark Controller per RMI übermittelt werden¹.

Im Gegensatz dazu erscheint ein **detailliertes Reporting** aber nur dann möglich, wenn die Load Driver zur Benchmark-Laufzeit *alle* ausführlichen Einzelmessergebnisse für eine *spätere* Auswertung mitprotokollieren (engl. to log) und als Datei sichern. Das Framework nutzt dazu den für Java standardisierten **Logging-Ansatz** aus dem Paket `java.util.logging`², sodass unterschiedliche Log Level von „FINEST“ bis „SEVERE“ zur Verfügung stehen, mit denen man Umfang und Detaillierungsgrad der Protokollierung je nach Bedarf einfach konfigurieren kann. Zudem bietet sich so auch die Nutzung des **XML-Standards** als Format für die Protokolldatei an. Die folgende Abbildung zeigt einen beispielhaften Auszug einer typischen Log-Datei mit einem markierten Messergebnis zu einer durchgeführten Einzeltransaktion:



```
<level>FINEST</level>
<class>de.whs.dbi.loaddriver.LoadDriver</class>
<method>call</method>
<thread>15</thread>
<message>Dauer der erfolgreichen Transaktion "SelectTx" in ms: 0</message>
</record>
<record>
  <date>2015-10-15T18:39:40</date>
  <millis>1444927180706</millis>
  <sequence>1590</sequence>
  <logger>de.whs.dbi.loaddriver.LoadDriver_1_local</logger>
  <level>FINEST</level>
  <class>de.whs.dbi.loaddriver.LoadDriver</class>
  <method>call</method>
  <thread>15</thread>
  <message>Dauer der erfolgreichen Transaktion "UpdateTx" in ms: 109</message>
</record>
<record>
  <date>2015-10-15T18:39:40</date>
  <millis>1444927180706</millis>
  <sequence>1591</sequence>
```

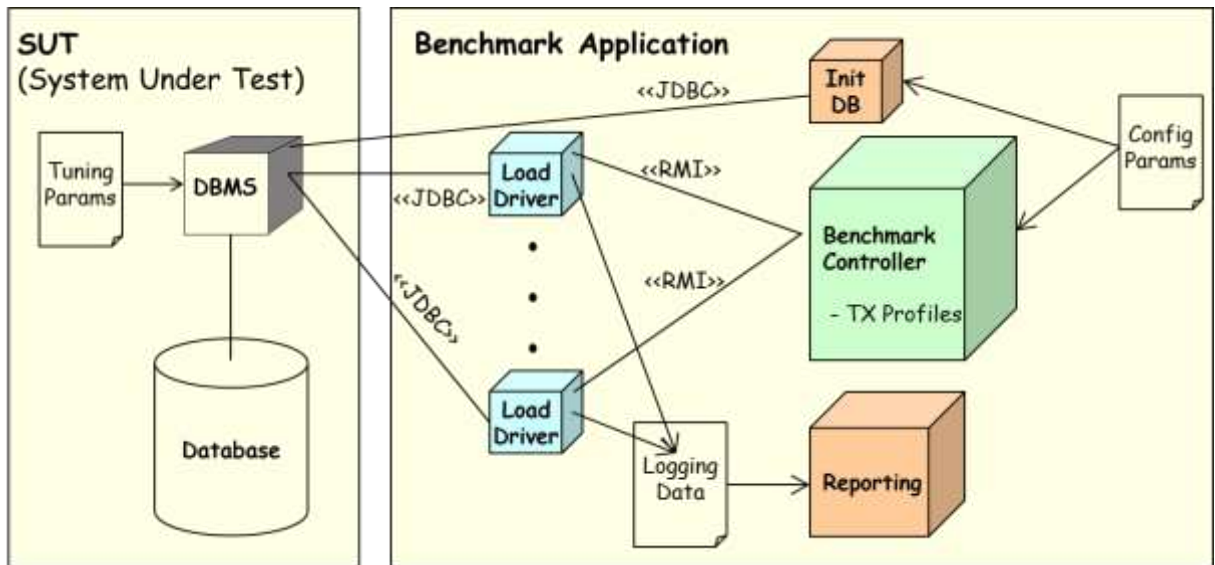
Das XML-Format wurde u.a. gewählt, um unter Nutzung entsprechender XML-Frameworks die Analyse und Auswertung solcher Log-Dateien möglichst einfach zu gestalten. Der Benchmark Controller sammelt dazu die Log-Dateien aller Load Driver einer Benchmark-Durchführung in einem dafür vorgesehenen Log-Verzeichnis. Für die detaillierte Auswertung ist dann eine eigene Reporting-Komponente vorgesehen³.

Damit kann der grobe Architekturüberblick zum Framework noch einmal durch ein angepasstes und leicht detaillierteres UML-Verteilungsdiagramm zusammengefasst und abgeschlossen werden:

¹ Siehe LocalResult-Objekte im Abschnitt „Grundlegende Klassen und Interfaces“!

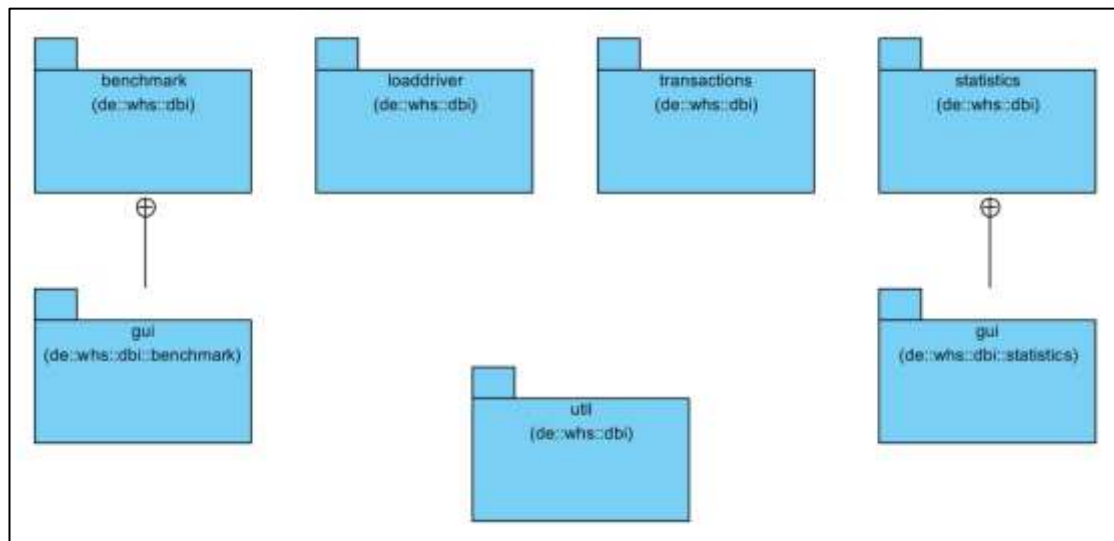
² Siehe bspw. <https://docs.oracle.com/javase/8/docs/technotes/guides/logging/> !

³ ..., die aber derzeit leider noch nicht vollständig realisiert ist.



Überblick über die Implementierung

Dem obigen Architekturüberblick entsprechend erfolgt die Java-Implementierung innerhalb der folgenden Paketstruktur:



Das **benchmark-Package** enthält alle Programmteile zur Realisierung der Klassen Benchmark-Controller und InitMyDatabase, während das **loaddriver-Package** alles aufnehmen soll, das zur Implementierung der Load Driver benötigt wird. Das **transactions-Package** enthält die abstrakte Klasse Transaction und könnte auch die selbst implementierten Unterklassen aufnehmen. Zusätzlich existiert ein eigenes **statistics-Package**, das das grobe Reporting und die detaillierte Auswertung der erzeugten XML-log-Dateien realisiert, sowie ein weiteres **util-Package**, das paketübergreifend genutzte Hilfsklassen und -Interfaces aufnimmt.

Die derzeitige Realisierung besitzt nur eine einfache ASCII-Benutzeroberfläche. Vorgesehen sind aber bereits zwei Unterpakete **benchmark.gui-Package** und **statistics.gui-Package**, die zu beiden Bereichen eine in Java Swing¹ oder JavaFX realisierte eigene Benutzeroberfläche anbieten soll.

Im Folgenden sollen die wichtigsten Punkte aus den einzelnen Paketen überblicksartig zusammengefasst werden.

Paket Benchmark

Das Paket **benchmark** enthält die Klassen, welche sich um die eigentliche Durchführung des Benchmarks kümmern. Die Klasse **BenchmarkController** ist verantwortlich für die Verwaltung der einzelnen **LoadDriver** sowie für das Laden der Konfiguration des Benchmarks. Außerdem steuert der **BenchmarkController** die Abfolge der einzelnen Phasen des Benchmarks. Der **BenchmarkController** implementiert das Interface **BenchmarkControllerIF**, welches wiederum das Interface **Remote** aus dem RMI-Paket implementiert. Der **BenchmarkController** besitzt außerdem eine **main**-Methode, die den kompletten Benchmark durchführt.

¹ Siehe bspw. <https://docs.oracle.com/javase/tutorial/uiswing/> !

Zur Nachvollziehbarkeit und zur späteren Analyse eines Benchmarks verwendet der **BenchmarkController** die Klasse **LogFileServer**. Diese ist dafür zuständig, zum Ende eines Benchmarks die Log-Dateien der einzelnen **LoadDriver** einzusammeln und zur Auswertung zur Verfügung zu stellen.

Paket LoadDriver

Im Paket **LoadDriver** sind ausschließlich die Klasse **LoadDriver** sowie das zugehörige RMI-Interface **LoadDriverIF** enthalten. Die Klasse enthält eine **main**-Methode, über welche ein **LoadDriver** gestartet wird. Über den Konstruktor registriert sich der **LoadDriver** beim **BenchmarkController**. Des Weiteren wird die JDBC-Verbindung vorbereitet und konfiguriert. Hierbei wird „auto commit“ abgeschaltet und das Transaktionsisolutionslevel fest auf „serializable“ gesetzt.

Der **LoadDriver** selbst ist ansonsten eher passiv. Das heißt, dass er selbst keine Aktionen selbstständig initiiert, sondern ausschließlich vom **BenchmarkController** dazu angestoßen werden muss. Über die **call**-Methode wird die Folge von Transaktionen gestartet. Der **BenchmarkController** ist nur für das Aufrufen zuständig, nicht jedoch für die Details der Transaktionsdurchführung. Der **LoadDriver** wählt eine zufällige Transaktion aus, führt diese aus und pflegt die Statistiken sowie die Logs. Nach Abschluss einer jeden Transaktion wird eine über die Konfiguration einstellbare Zeitspanne gewartet.

Die **LoadDriver**-Klasse ausschließlich mit der im Folgenden beschriebenen Klasse **WeightedTransaction**.

Paket Transactions

Das Paket **transactions** stellt die abstrakten Klassen **Transaction** und **WeightedTransaction** bereit. **Transaction** enthält Methodenschnittstellen für grundlegende Funktionalitäten einer Transaktion. **WeightedTransaction** erweitert diese Klasse dann um eine zusätzliche Gewichtung. Von **WeightedTransaction** werden einzelne Transaktionsimplementierungen abgeleitet. Für einen eigenen Transaktionstyp muss in jedem Fall die **run**-Methode implementiert werden. In dieser werden die einzelnen SQL-Anweisungen des Transaktionstyps detailliert ausprogrammiert. Um den Benchmark nicht zu verfälschen, sollte *in keinem Fall* selbst eine JDBC-Verbindung geöffnet werden. Die Verbindung wird einer jeden Transaktion bereits im Konstruktor übergeben und ist bereits vorbereitet.

Paket Statistics

Die bisher implementierten Reportings beschränken sich auf die einfache Auswertung und Ausgabe durch den **BenchmarkController** selbst, ohne die XML-Protokolldateien direkt auszuwerten. Das Paket realisiert u. a. die Klassen **GlobalResult** und **LocalResult** sowie eine Hilfsklasse **TXResult** für aggregierte Messergebnisse eines einzelnen Transaktionstyps.

Paket Util

Im Paket **util** existiert eine **RemoteLogger**-Klasse. Diese wird im **BenchmarkController** verwendet, um Logs von Loadriven über RMI empfangen zu können. Wird eine Nachricht empfangen, so wird diese über einen lokalen Logger wiederum geloggt, damit die Nachricht in der Ausgabe des Benchmark-Controllers sichtbar wird.

Das Paket enthält zudem eine **Hilfsklasse ParameterGenerator**, die häufig direkt genutzt werden kann, um benötigte Zufallswerte für unterschiedliche Datentypen zu erzeugen. Diese Klasse kann bei Bedarf Benchmark-spezifisch durch Ableitung erweitert und angepasst werden.

Installation und Einsatz des Frameworks

Das Framework wird in der Form zweier Eclipse-Projekte zur Verfügung gestellt, die mittels SVN-Check-Out in den eigenen Workspace geladen und bearbeitet werden können. Dies erlaubt jedem Entwickler Einblick in den kompletten Quelltext des Frameworks, und so kann er bei Bedarf auch dessen Funktionsweise und Programmierung im Detail studieren und ggf. Anpassungen und Ergänzungen vornehmen.

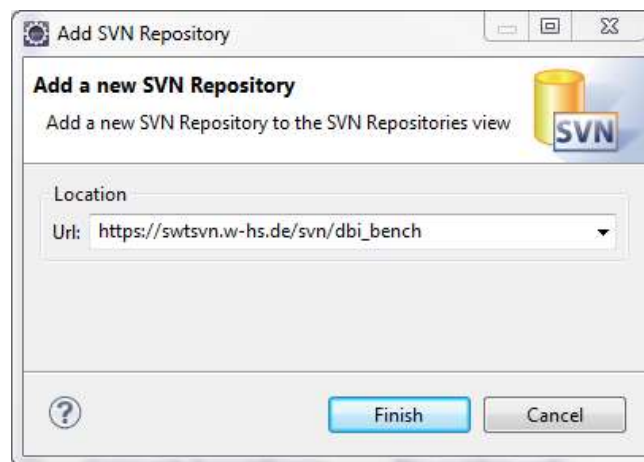
Die folgenden Erläuterungen setzen voraus, dass zur Installation und Nutzung des Frameworks bereits

- ein aktuelles **Java SDK**,
- eine aktuelle **Eclipse-Entwicklungsumgebung** mit einem **Plugin für Subversion (SVN)** als Werkzeug zum Konfigurationsmanagement und zur Versionskontrolle sowie
- ein installiertes **JDBC-unterstützendes Datenbankmanagementsystem**

vorhanden sind.

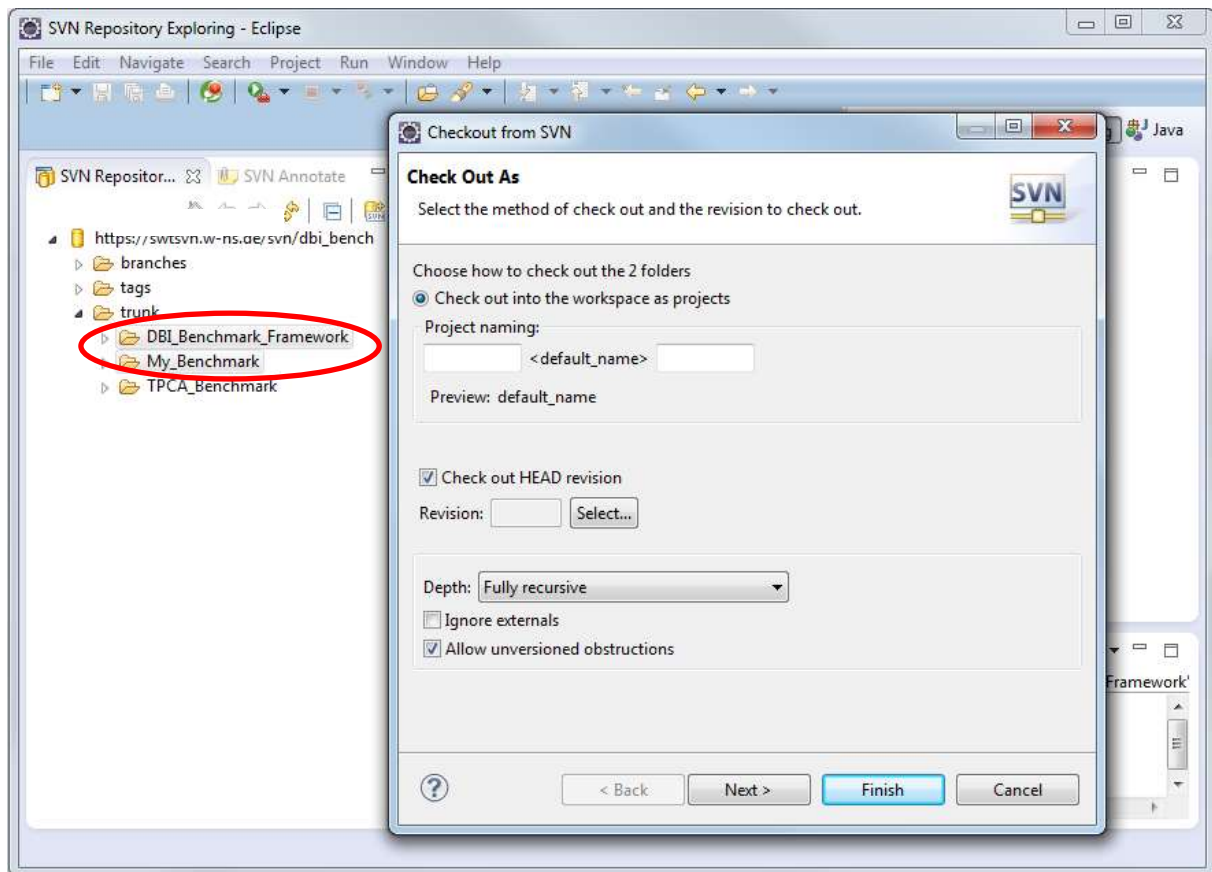
Check-Out als Eclipse-Projekte

Die Adresse des **SVN-Repositorys** lautet https://swtsvn.w-hs.de/svn/dbi_bench und muss in der SVN-Repository-Exploring-Perspektive beim Hinzufügen eines neuen Repositorys genutzt werden¹:



Anschließend können die beiden **Projekte „DBI_Benchmark_Framework“ und „My_Benchmark“** aus dem SVN-trunk-Bereich ausgewählt werden, um sie per Check-Out in den eigenen Workspace zu laden. Das Projekt „DBI_Benchmark_Framework“ enthält das komplette Framework zum DB-Benchmarking, das im Wesentlichen *unverändert* bleiben soll. Das zweite Projekt stellt ein bereits *lauffähiges* „Template-Projekt“ dar mit allen notwendigen Eclipse-Einstellungen für die Entwicklung eines konkreten Benchmarks auf der Basis eines konkreten DBMS. Dieses Projekt *muss angepasst und ergänzt* werden - in diesem Projekt findet also die eigentliche Benchmark-Entwicklung statt.

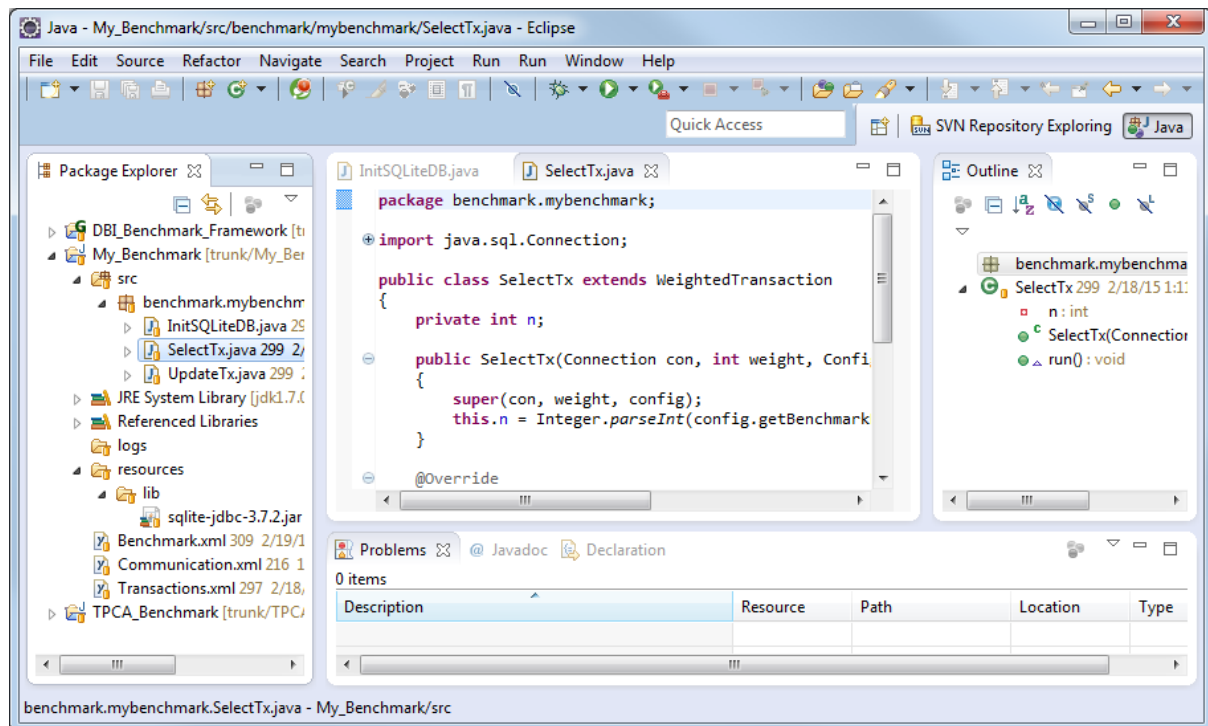
¹ Zur Authentifizierung nutze man einfach die bereits bekannten SVN-Zugangsdaten aus der Softwaretechnik-Lehrveranstaltung oder „bench/bench“.



Nach dem Check-Out¹ besitzt man im lokalen Workspace die beiden gewünschten Projekte mit der folgenden **Verzeichnisstruktur im Template-Projekt My_Benchmark**²:

¹ Bitte trennen Sie die Projekte direkt wieder mittels „Disconnect“ vom angegebenen SVN-Repository und übertragen Sie sie ggf. in ein eigenes Repository. In der Datenbanklehrveranstaltung werden dazu bei Bedarf weitere Repositories zur Verfügung gestellt. Details dazu im Praktikum!

² Die Detailstruktur im eigentlichen Framework-Projekt wird im Folgenden nicht weiter erläutert. Dies bleibt dem interessierten Leser als Aufgabe zum Selbststudium überlassen!



Das **src-Verzeichnis** enthält im Java-Package `benchmark.mybenchmark` Klassen als Ausprogrammierung einzelner **Lasttransaktionen** (hier bspw. `SelectTx`)¹. Weiterhin enthält das Java-Package die **Klasse `InitSQLiteDB`**², die zur Initialisierung der aufzubauenden Benchmark-Datenbank vorgesehen ist (hier für das konkrete DBMS SQLite).

Das **resources/lib-Verzeichnis** enthält einzubindende Java-Archivdateien (Jar-Dateien) bspw. für benötigte JDBC-Treiber (hier bspw. den JDBC-Treiber für SQLite). Das **logs-Verzeichnis** ist zur Aufnahme der XML-Protokolldateien des RemoteLoggers vorgesehen³.

Zusätzlich gibt es im Projekt die drei Konfigurationsdateien⁴

- **Benchmark.xml** für allgemeine Benchmark-Konfigurationsparameter
- **Transactions.xml** zur Angabe der Lasttransaktionen mitsamt ihrer Gewichtung
- **Communication.xml** zur Festlegung der zu nutzenden RMI-Registry

mit entsprechenden Standardeinstellungen, die durch den Entwickler angepasst und ergänzt werden können.

¹ Siehe Abschnitt „Grundlegende Klassen und Interfaces“!

² Vgl. Abschnitt „Initialisierung der Benchmark-Datenbank“!

³ Vgl. auch Abschnitt „Reporting“!

⁴ Vgl. auch Abschnitt „Konfigurationseinstellungen“!

Anpassungen an konkreten Benchmark und konkretes DBMS

Die Realisierung eines Benchmarks mithilfe des Frameworks besteht im Wesentlichen aus der Abarbeitung der folgenden **Anpassungsschritte**¹ innerhalb des Eclipse-Projektes My_Benchmark:

1. Ablage des einzusetzenden **JDBC-Treibers** als jar-Datei im resources/lib-Verzeichnis
2. Einbindung des JDBC-Treibers im **Java Build Path** des Projektes
3. Ausprogrammierung der Klasse InitMyDatabase zur DBMS-spezifischen **Initialisierung der Benchmark-Datenbank**
4. Ausprogrammierung der einzelnen **Lasttransaktionen** des Benchmarks jeweils als parameterlose public-void-run()-Methode einer Unterklasse der Framework-Klasse Transaction²
5. Überprüfung, evtl. Ergänzung und Festlegung der **Konfigurationsparameter** in den drei XML-Konfigurationsdateien.

Das Eclipse-Projekt My_Benchmark enthält für alle Anpassungsschritte bereits beispielhaft gewählte Voreinstellungen, die einen trivialen **Beispiel-Benchmark** für das „Embedded DBMS“ SQLite³ realisieren. Neben der Demonstration der einzelnen Anpassungsschritte kann dieser einfache Benchmark auch sehr gut zur Überprüfung der Korrektheit der eigenen Installation genutzt werden. Dazu führt man diesen Benchmark wie im nächsten Abschnitt gezeigt durch und überzeugt sich so von der Lauffähigkeit der Installation.

Durchführung eines Benchmark-Laufs

Eine **Benchmark-Messung** kann direkt aus Eclipse heraus gestartet werden⁴. Dazu ist nach erfolgter Einstellung der Konfigurationsparameter der Eclipse-Workspace auf die entsprechenden Rechner für den Benchmark Controller bzw. für die Load Driver zu kopieren. Der Start⁵ erfolgt dann sehr einfach über das Run-Icon aus dem Eclipse-Menü mit vordefinierten Konfigurationen (engl. Eclipse Launch Configurations).

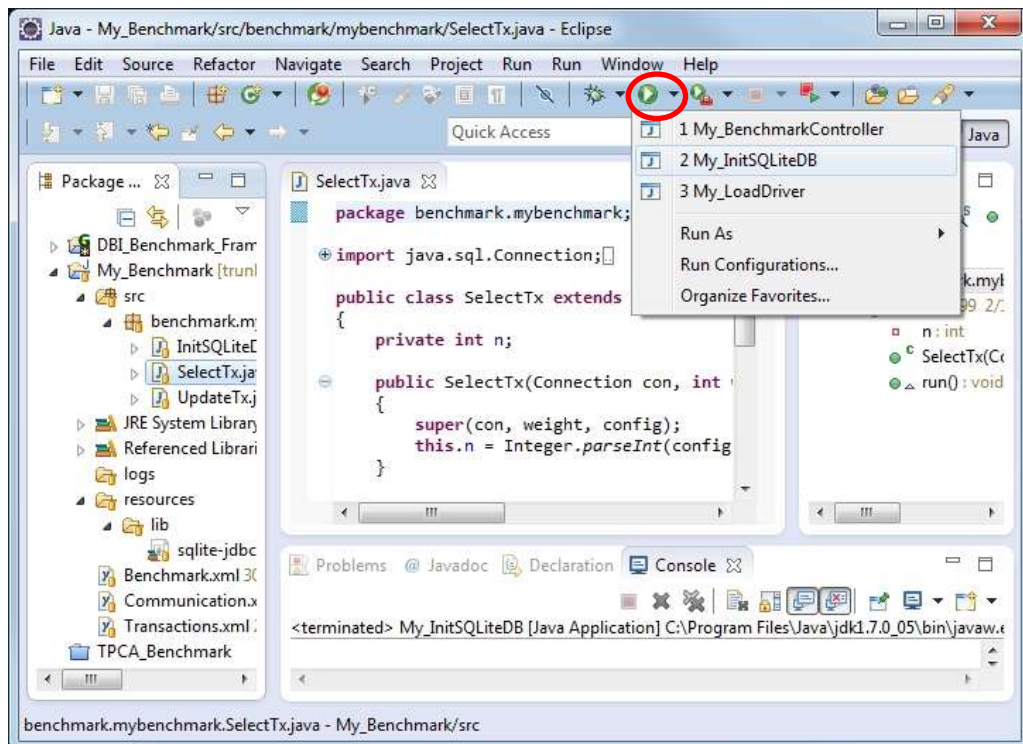
¹ Die Schritte 3-5 werden in der Regel nicht rein sequentiell sondern eher in mehreren Iterationen durchlaufen, um jeweils die bei der Implementierung benötigten Konfigurationsparameter direkt in den entsprechenden XML-Konfigurationsdateien einzugeben oder anzupassen.

² *Achtung:* Die Methoden werden ohne JDBC-Commit programmiert, da das Commit vom Framework automatisch gesetzt wird! Häufig wird auch die bereits erwähnte Spezialisierung **WeightedTransaction** anstelle der allgemeineren Klasse Transaction genutzt, um die Lasttransaktionsklassen davon abzuleiten.

³ Siehe <http://www.sqlite.org>!

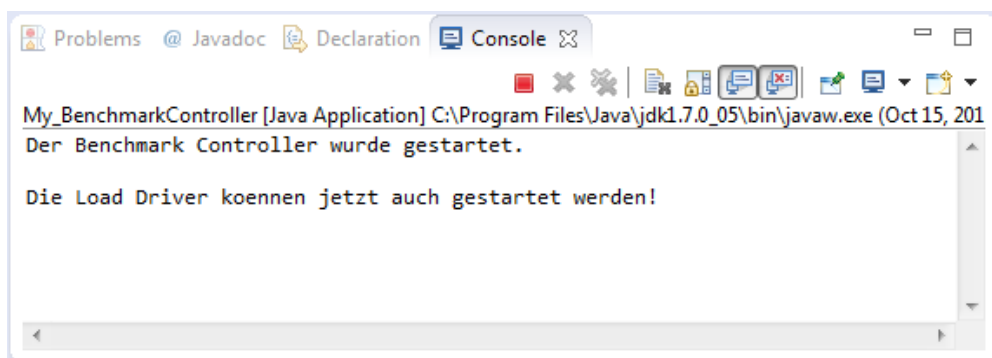
⁴ Dies stellt sicherlich die einfachste Aufrufmöglichkeit dar. Natürlich kann das Projekt aber auch als Java Archive (jar-Datei) exportiert werden, um dann direkt und unabhängig von der Eclipse-Entwicklungsumgebung gestartet zu werden.

⁵ Vor dem Start sollte man auf jeden Fall sicherstellen, dass das DBMS auch **Remote-Verbindungen** zulässt (und nicht als reines Entwickler-DBMS mit einer ausschließlichen Unterstützung nur lokaler Verbindungen installiert wurde). Außerdem ist es ratsam, die Einstellungen einer evtl. aktivierten **Firewall** zu überprüfen und ggf. anzupassen!



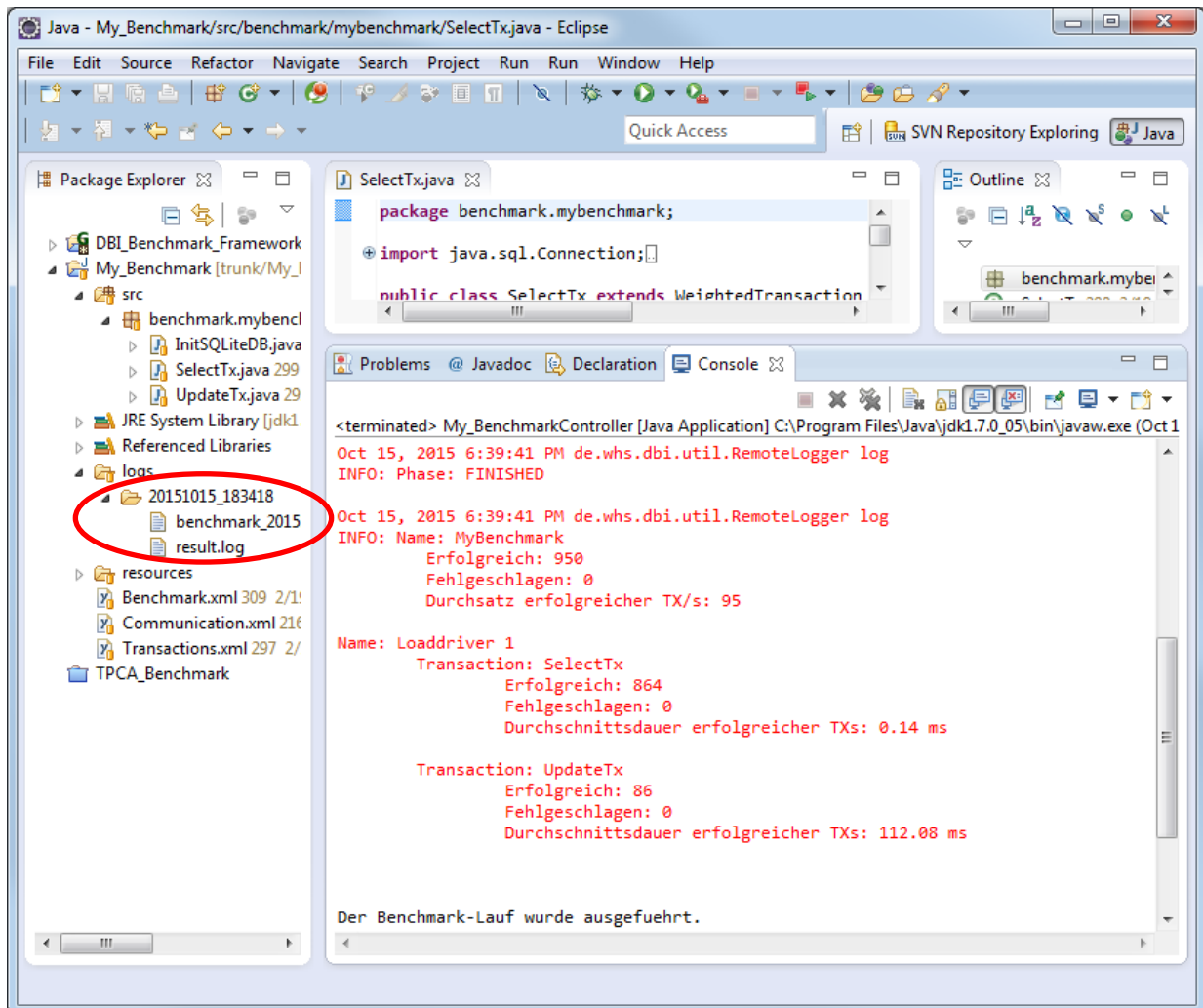
Falls die **Initialisierung der Benchmark-Datenbank** noch nicht durchgeführt wurde, so kann sie über einen eigenen Eintrag (im obigen Bild "My_InitSQLiteDB") einmalig initialisiert werden.

Zum **Starten des Benchmark Controllers** wählt man anschließend die Run-Konfiguration "My_BenchmarkController" aus. Der Benchmark Controller startet, gibt einige Kontrollmeldungen auf der Konsole aus und wartet dann auf die in den Konfigurationsparametern festgelegte Anzahl benötigter Load Driver, die danach einzeln auf ihrem jeweiligen Rechner gestartet werden müssen.



Das **Starten der Load Driver** erfolgt völlig analog über die Run-Konfiguration "My_LoadDriver". Die einzelnen Load Driver melden sich direkt nach dem Start selbstständig beim Benchmark Controller an, und sobald der Controller die in den Konfigurationsparametern festgelegte Anzahl erkannt hat, beginnt der eigentliche Benchmark-Lauf. Dazu gibt der Controller auch wieder entsprechende Kontrollmeldungen direkt auf der Konsole aus. Nach Beendigung des Benchmark-Laufs wird eine Zusammenfassung der erzielten **Messergebnisse** auf der Konsole ausgegeben. Zusätzlich befindet

sich dann die zugehörige XML-Protokolldatei bereits im logs-Verzeichnis. Diese Protokolldatei kann bei Bedarf mit dem Ziel einer ausführlichen Detailanalyse der Ergebnisse ausgewertet werden¹. Das folgende Bild zeigt die Konsolenausgabe bei einem Benchmark-Lauf mit nur einem Load Driver:



¹ Der Leser sollte sich von der Lauffähigkeit seiner Installation überzeugen und dazu den Beispiel-Benchmark für das Embedded DBMS SQLite tatsächlich einmal durchführen! Auch ein Blick in die erzeugten XML-Protokolldateien erscheint dabei ratsam.

Zusammenfassung und Ausblick

Nach einer generellen Motivation einer Framework-Nutzung bei der Implementierung von Datenbank-Benchmarks wurde in diesem Dokument ein konkretes, Java-basiertes Framework zum DB-Benchmarking beschrieben.

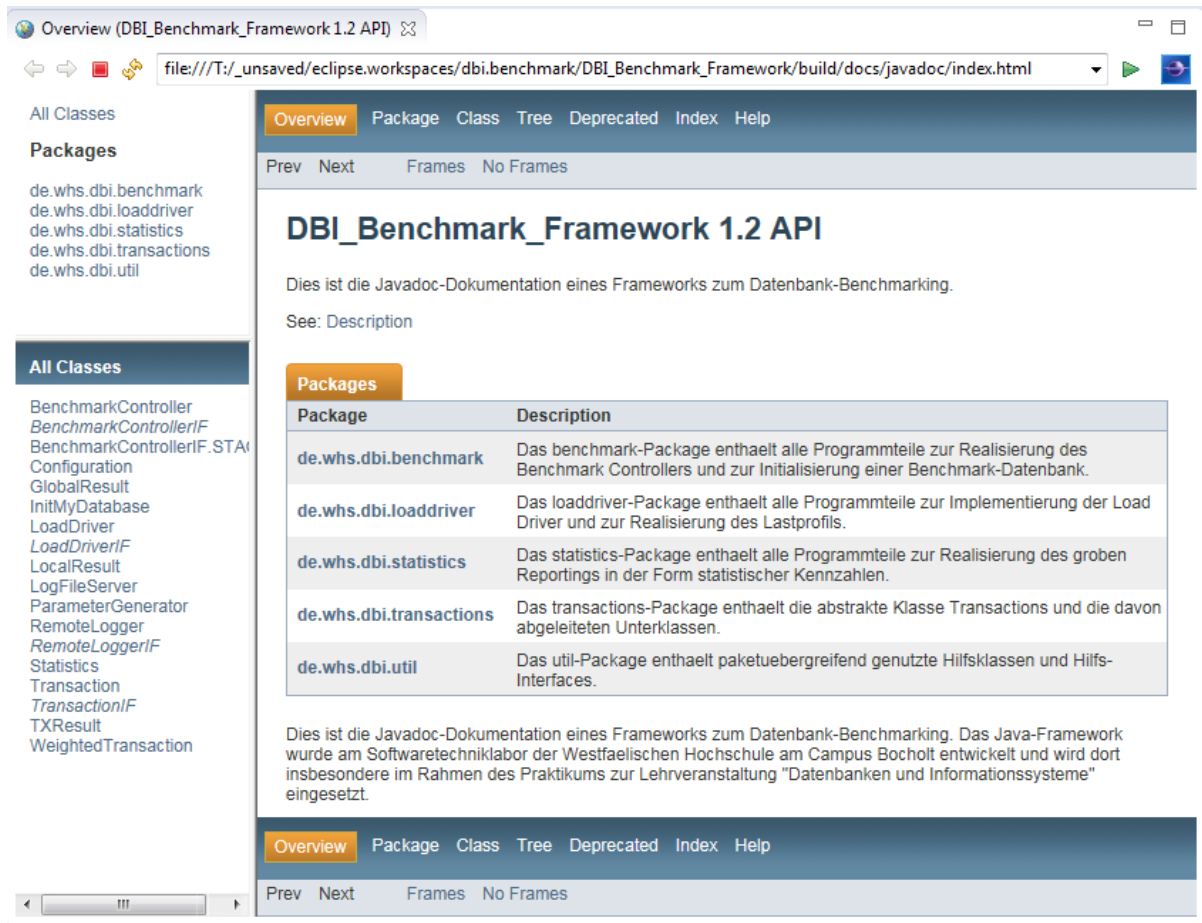
Zunächst wurde die zugrunde liegende Architektur überblicksartig dargestellt, um so die wichtigsten Komponenten des Frameworks und deren Zusammenspiel kennenzulernen. Anschließend wurden einzelne Details der Implementierung des Frameworks entlang der gewählten Java-Paketstruktur erläutert. Detaillierte Schilderungen zur Installation und zum Einsatz des Frameworks vervollständigten diese einführende Beschreibung des Frameworks und sollen eine einfache direkte Nutzung ermöglichen.

Die Entwicklung des Frameworks ist noch nicht vollständig abgeschlossen. Bspw. fehlen noch GUI-Komponenten für die einzelnen Bestandteile, und auch ein detailliertes Reporting basierend auf der Auswertung der erzeugten XML-Protokolldateien ist bisher nicht realisiert. Dies soll zukünftig aber noch erfolgen, und wir wiederholen an dieser Stelle noch einmal unsere Aussage, dass Hinweise auf gefundene Fehler, gewünschte Änderungen oder sinnvolle Erweiterungen in den Programmen oder in diesem Dokument jederzeit sehr willkommen sind¹!

¹ bspw. per Mail an bernhard.convent@w-hs.de

Anhang

Das Eclipse-Projekt DBI_Benchmark_Framework ermöglicht eine einfache Erstellung üblicher **Javadoc-API-Beschreibungen**, die bei der Programmierung sicherlich hilfreich sind:



Die Erstellung erfolgt entweder unter dem Eclipse-Menüpunkt „Project-->Generate Javadoc ...“ oder einfach mithilfe des enthaltenen **Gradle-Buildfiles** build.gradle.