Technical Report

urn:nbn:de:gbv:830–tubdok–10843

# Java Transactional Domain Programmers Guide V 0.2

Holger Machens
Institute of Telematics
Hamburg University of Technology
Hamburg, Germany

machens@tuhh.de

February 2011

Institute of Telematics
Hamburg University of Technology
TUHH

# Table of Contents

Chapter 1

# Introduction

Transactional memory is promoted as a promising solution to ease parallel programming by solving common problems with deadlocks and scalability. Unfortunately, transactional memory introduces its own problems for programmers due to its rollback and retry behaviour such as inconsistencies when switching between shared protected and local unprotected use of data (privatization [SMDS07]), as well as problems with irrevocable operations such as I/O or blocking in transactions for example to wait for a condition.

One solution to this dilemma is the use of a programming model which prevents the programmer from those problems. The Institute of Telematics of the Hamburg University of Technology analyses the use of a generic programming model [MT11] conforming to this requirement and especially considering transaction-enabled data models to be separated in modules (e.g. libraries) without making demands on the application using that data model (such as instrumentation of each data access).

Java Transactional Domain (JTD) is an adaption of this generic programming model to Java. It consists of a set of programming rules for Java and a prototypical toolchain supporting the validation and automated source code instrumentation. The instrumentation tool and runtime library consist of a modified version of the AtomJava STM system [HG06]. The modified AtomJava compiler and runtime library supports even privatization and local use of data [ATLM+06] to demonstrate automated optimizations as well as manual optimization by the use of lower level APIs for expert programmers.

This document contains the programmers guide providing information to install and use the JTD toolchain.

Chapter 2

# Disclaimer

The Java Transactional Domain Development Toolkit is a research prototype in beta stage. The source is published under GPL v3 [Fou07]. Thus, the Institute of Telematics and the Hamburg University of Technology does not provide any warranty to anything anyhow related to the JTD and the JTD toolchain.

Chapter 3

# Installation

## 3.1 Unix

### Preconditions

You have to install the following software (if not already installed):

- Eclipse 3.6.1 with Java Development Toolkit (JDT) plug-in installed.
- Sun Java 1.6.
- Apache Ant 1.7.

### Installation Procedure

1. Download `jtd-2.0.0.tgz` if not already done.
2. Extract `jtd-2.0.0.tgz` to a directory of your choice.
   `> tar xzf jtd-2.0.0.tgz`
   This will create a directory with the name JTD which we will call *installation root directory* from now on.
3. Enter *installation root directory*: `> cd JTD`
4. Open `./install.sh` script and customize the variables as explained there.
5. Start the build process by typing: `> ./install.sh`. This automatically gathers required packages atomjava 0.1 and polyglot 1.3.4 from directory `thirdparty`, applies all patches and runs a build using `ant`. After that you have:

- The plugin readily build and zipped into `de.tuhh.jtd.dt_2.0.0.zip`
- All sources gathered and patched in the installation directory.

6. Unzip the file `de.tuhh.jtd.jar_2.0.0.zip` to your Eclipse plugin directory `$ECLIPSE_HOME/plugins`.

7. (Re-)Start your Eclipse instance with `eclipse -clean`.

## 3.2 Windows

### Preconditions

You have to install the following software (if not already installed):

- Eclipse 3.6.1 with Java Development Toolkit (JTD) plug-in installed.
- Sun Java 1.6.
- Apache Ant 1.7.
- Cygwin to execute the build script (using bash). Cygwin is no longer needed when the installation is done.

### Installation Procedure

1. Download `jtd-2.0.0.tgz` if not already done.

2. Extract `jtd-2.0.0.tgz` to a directory of your choice.
   `> tar xzf jtd-2.0.0.tgz`
   This will create a directory with the name JTD which we will call *installation root directory* from now on.

3. Enter *installation root directory*: `> cd JTD`

4. Open `./install.sh` script in an editor and customize the variables as explained there.

5. Start the build process by typing: `> ./install.sh`. This automatically gathers required packages atomjava 0.1 and polyglot 1.3.4 from directory `thirdparty`, applies all patches and runs a build using `ant`. After that you have:
   - The plugin readily build and zipped into `de.tuhh.jtd.dt_2.0.0.zip`
   - All sources gathered and patched in the installation directory.

6. Unzip the file `de.tuhh.jtd.jar_2.0.0.zip` to your Eclipse plugin directory `$ECLIPSE_HOME/plugins`.

7. (Re-)Start your Eclipse instance with `eclipse -clean`.

## 3.3 Manual Installation

### Preconditions

- Eclipse 3.6.1 with Java Development Toolkit (JTD) plug-in installed.
- Sun Java 1.6.
- Apache Ant 1.7.
- Some adequate replacement for the Unix tool `patch`.

### Installation Procedure

You will need the following software to be installed:

1. Download `jtd-2.0.0.tgz` if not already done.
2. Download Polyglot version 1.3.4 from `http://www.cs.cornell.edu/projects/polyglot/`.
3. Download AtomJava version 0.1 (`atomjava0.1.tar.gz`)from `http://wasp.cs.washington.edu/wasp_atomjava.html`.
4. Download the patch `atomjava0.1-polyglot-1.3.4-src-patch` from `http://wasp.cs.washington.edu/wasp_atomjava.html`.
5. Extract `jtd-2.0.0.tgz` to a directory of your choice.
   `> tar xzf jtd-2.0.0.tgz`
   This will create a directory with the name JTD which we will call *installation root directory* from now on.
6. Extract `polyglot-1.3.4-src.tar.gz`. This creates directory `polyglot-1.3.4-src`.
7. Apply the patch `atomjava0.1-polyglot-1.3.4-src-patch` to the directory `polyglot-1.3.4-src`.

8. Extract `atomjava0.1.tar.gz` in the directory `polyglot-1.3.4-src/src/polyglot/ext`. This result in a new directory `polyglot-1.3.4-src/src/polyglot/ext/atomjava`.

9. Apply the patch `JTD/polyglot-1.3.4-patch` to the file `polyglot-1.3.4-src/build.xml`.

10. Cleanup the polyglot build directory by running
    `> ant clobber`.

11. Remove the following files and directories from the directory `polyglot-1.3.4-src/src`:
    - `polyglot/ext/atomjava/runtime/AMain.java`
    - `polyglot/ext/atomjava/runtime/AThread.java`
    - `polyglot/ext/atomjava/runtime/wrapper/java/io/.#PrintStream.java.1.3`
    - `polyglot/ext/atomjava/visit/CurrentThreadSharer.java`
    - `ppg/test`

12. Apply the patch `JTD/JTD-0.1-atomjava-0.1-src-patch` to the directory `polyglot-1.3.4-src/src`.

13. Copy all files from `polyglot-1.3.4-src/src` to `JTD/atomjava`. At this stage, you have created the complete build environment for the JTD Eclipse plugin. You can even import the project located in `JTD` into Eclipse if you have the Plugin Development Environment installed.

14. Build the plugin by calling `ant zip.plugin`. This creates a zip file named `de.tuhh.jtd.jar_2.0.0.zip` in the directory `JTD`.

15. Unzip the file `de.tuhh.jtd.jar_2.0.0.zip` to your Eclipse plugin directory `$ECLIPSE_HOME/plugins`.

16. (Re-)Start your Eclipse instance with `eclipse -clean`.

## 3.4 Check Installation

To check if the installation of the JTD Plug-in was successful open the overview of installed plug-ins via `Help -> About Eclipse SDK -> Installation Details`

`-> Plug-ins`. You should find a plug-in called Java Transactional Domain Development Toolkit (see Figure 3.1).

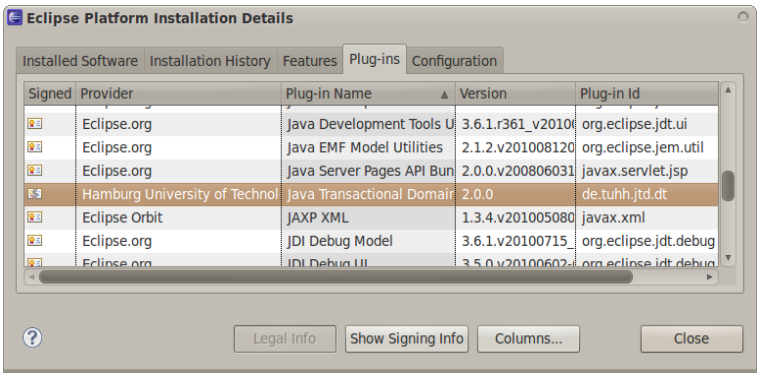


**Figure 3.1:** Expected entry in overview of installed plug-ins

Chapter 4

# Programming Transactional Domains

The concept of a transactional domain is to create data models which are protected by transactions and separated from the remaining software to prevent errors. In means of Java Transactional Domain a transactional domain consists of the set of all transactional objects. The transactional objects in JTD guarantee protection and deadlock freedom in concurrent use by multiple threads through the use of Transactional Memory. Declared transactional objects are automatically instrumented by the instrumentation tool of the JTD toolchain. The programmer declares transactional objects simply by inheritance of the interface `de.tuhh.jtd.runtime.Transactional` (see Listing 4.1).

```
1  package org.acme.dining.philosophers;
2  import de.tuhh.jtd.runtime.Transactional;
3
4  public class Philosopher implements Transactional {
5
6    public void eat() {
7      getForks();
8      eating();
9      putForks();
10   }
11   public void think() {
12     // wise thinking ...
13   }
14
15   // ...
16 }
```

**Listing 4.1:** Declaration of transactional objects

To provide the guaranteed protection the programmer must follow a given set of programming rules given in Section 4.2. The conformance to this programming rules is automatically checked by the validation tool of the JTD toolchain. Violations of rules are displayed as errors in the Java editor.

## 4.1 Project Setup

To develop Java Transactional Domains you can choose one of two ways:

- **Create a JTD project:** Just select the menu entry `File-> New->Project` and choose the Java Transactional Domain Project wizard (Figure 4.1) to create your new project.
- **Enhance an existing Java project:** Select the Java project in the Project Explorer and choose the entry "Add/Remove Java Transactional Domain Nature" from the context menu (right click).

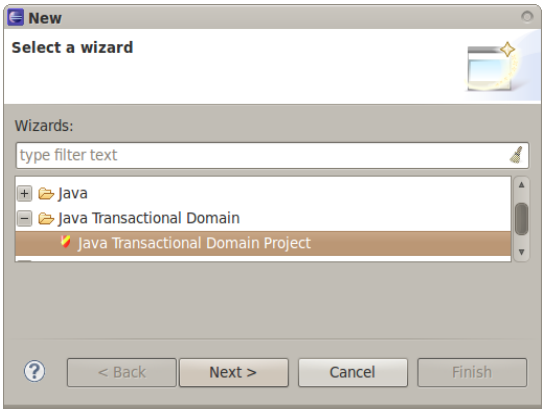


**Figure 4.1:** Project creation wizard

***Important: Set Compiler compliance level to Java 1.4!*** because AtomJava does not support levels higher or equal to 1.5.

Either way for each created project (say *master project*) a so-called *slave project* is created which is used by JTD to instrument and compile transactional objects (see Figure 4.2). The original source files (those you are editing during development) will remain in the *master project* and will never be modified by the JTD system.

Files containing transactional classes or interfaces are indicated by a red and yellow plate on its icon in the project explorer. The *master project* automatically validates those classes and interfaces declared to be transactional and transfers them to the slave project to get instrumented versions of them.

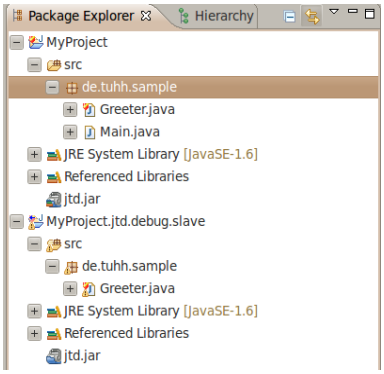


**Figure 4.2:** Master and slave project in the project explorer

## 4.2 Programming Rules

As mentioned in the introduction of this chapter transactional domains must conform to the set of programming rules provided in this section. The set of programming rules is automatically checked prior to each build in any project with the JTD nature (see Section 4.1).

### Rules

***The following list contains even rules for Java 1.5 but the current version of JTD supports only Java 1.4 due to restrictions in AtomJava!***

1. Instance variables of transactional classes must be private unless they are final.

2. Instance variables of transactional classes must be of primitive type, of type String, or references on transactional objects.

3. Any code of a transactional class runs inside a transaction. This includes methods, constructors, instance or class variable initializations and instance or class initialisation blocks.

4. Code of transactional classes does not access any non-transactional objects. Prohibited access includes method calls, constructor calls, direct variable access or class initialisation.

5. Parameters of transactional methods must be of primitive type, of type String or transactional.

6. Arrays as types of parameters or return values of transactional methods are prohibited.

7. Base classes/interfaces as well as derived classes/interfaces of transactional classes/interfaces must be transactional as well.

8. Type arguments of transactional generic classes must be either of type String or transactional.

9. Use of thread suspension (e.g. `wait`, `notify`, `synchronized`) or `native` methods in transactional classes is prohibited.

*Rule 1* grants access to member variables of transactional objects to its own code only which implicitly starts a transaction (see rule 3). *Rule 2* prevents a programmer from temporary saving and accessing non-transactional data inside the transactional domain which does not support recovery. Primitive types are allowed because in Java primitives are returned by value and strings are constants. This makes modifications of both kinds of internal data impossible. *Rule 3* guarantees, that transactional data is accessed by transactions only. *Rule 4* prevents from breaking up strong isolation by getting access to non-transactional data inside a transactional method. *Rule 5* prevents from sharing non-transactional data with a transaction. Parameters provided to a method might be modified in the transaction and therefore need to support transactions as well. For parameters of primitive type, Java has a call-by-value semantics. Thus, the method is working on its own copy and needs no concurrency control. Strings are immutable and need no concurrency control either. *Rule 6* restricts the use of arrays because they do not support transactions. Arrays have to be replaced by a transactional wrapper class provided by the STM and supporting access to an underlying array. *Rule 7* prevents a programmer from calling non-transactional methods of the base class or sub-class when expecting guaranteed consistency. It also allows to declare any implementation of an interface to be transactional. Methods of the common super class Object are overridden by transactional methods of the STM system or prohibited when final. *Rule 8* restricts the use of type arguments for generic transactional classes. To fulfil this requirement, all type arguments of a transactional generic class declaration must have at least one transactional type bound (e.g. `<T extends Transactional>`) or a type bound of type String. *Rule 9* in conjunction with *Rule 4* finally prevents use of blocking and probably irreversible methods.

## 4.3 Privatization Support

This is an example of an API for experienced programmers supporting optimization of applications that use transactional memory. This API supports to switch off transactional behaviour of transactional objects. This slightly improves the performance of transactional objects.

Privatization aims at reducing the computation effort for transactional objects by turning off transactional behaviour. A transactional object may be transformed into a privatized object for one thread by *privatization* and reversed into a shared transactional object by *publication*. A privatized object is accessed by the owner thread only and therefore will not run transactions on its own. Access to privatized objects by other threads is treated as an error. If a privatized object is accessed in a transaction it must behave transactional to support a possible rollback. Thus, it always attaches to running transactions of the same thread.

The API provides the following methods:

- **PrivatizationService.privatize(object)**: Privatize an object for a given thread.
- **PrivatizationService.publish(object)**: Publish a privatized object (i.e. make it shared).
- **PrivatizationService.isShared(object)**: Check if the object is not privatized.
- **PrivatizationService.isPrivatized(object)**: Check if the object is privatized by the calling thread.

All those methods are transactional to support multiple actions at once such as privatizing/publishing multiple objects.

## 4.4 Running and Debugging

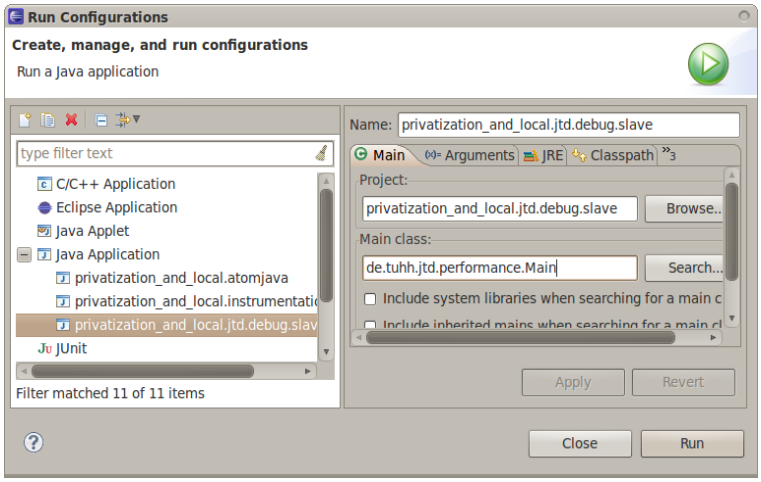


**Figure 4.3:** Launch configuration

The beta version of JTD does not support automatic copying of generated binaries in the *master project*. Therefore you need to run your java application from the slave project. The pathes in the *slave project* are automatically setup to support it. You just need to create a launch

configuration (*Run –> Run Configurations ...*) which points to the *slave project* and to the class containing the main method (which will be located in the *master project*, cf. Figure 4.3). You can even use an existing launch configuration which is configured for the *master project* and point it to the *slave project*.

The same launch configuration can be used to debug your application. The debugger will step through the instrumented version of your transactional objects which might be a bit confusing if you are not planning to modify the STM system. This is another issue to be addressed in some release version of JTD.

## 4.5 Export

If you are planning to export the generated binaries then be aware of the project structure. Assuming you have compiled classes in *master* and *slave project* you can safely start by copying all class files from the master project to a temporary location and then copy all class files from the slave project to the same location. The latter step overrides the classes from the *master project* with the instrumented class files from the *slave project*.

When bundeling the exported class files in a jar file the archive can be used without knowledge of the transactions running inside.

# Bibliography


[ATLM+06] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 26–37, New York, USA, 2006. ACM.

[Fou07] Free Software Foundation. *GNU GENERAL PUBLIC LICENSE*. Free Software Foundation, 2007. `http://licenses/gpl-3.0.txt`.

[HG06] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *MSPC '06: Proceedings Workshop on Memory System Performance and Correctness*, pages 82–91, New York, USA, 2006. ACM.

[MT11] Holger Machens and Volker Turau. Avoiding publication and privatization problems on software transactional memory. In *Proceedings of the Kommunikation in Verteilten Systemen 2011*, Open Access Series in Informatics (OASICS). Dagstuhl Publishing, 2011.

[SMDS07] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. In *PODC '07: Proceedings 26$^{th}$ Annual ACM Symp. on Principles of Distributed Computing*, pages 338–339, New York, USA, 2007. ACM.