

Java version DHSVM

Fan Min

*School of Computer Science and Engineering,
University of Electronic Science and Technology of China, Chengdu 610054, China
College of Computer Sciences,
University of Vermont, Burlington, Vermont 05403, USA.*

Abstract

DHSVM is a very important benchmark used by people from hydrology. This document shows how it is translated into a java version. The translation approaches are explained, and the difficulties are indicated. Moreover, possible bugs and possible introduction of bugs are pointed out. The purposes of this document is to share my ideas and discuss some issues with people in this society.

Key words: Hydrology, DHSVM, c, Java

1 Introduction

DHSVM is an excellent and famous benchmark developed by distinguish people in the Washington University. It helped researchers and application developers to solve hydrology related problems.

Currently, only the ANSI C version is available. With the development of object oriented programming (OOP), there is a need of an advanced version. I choose Java for many reasons, including easier to handle large systems, easier to find bugs, easier to use, easier to write GUI, etc., as will be discussed in more detail in Subsection 2.1. A C version user may argue that Java is rather slow, however, this is not too bad since we have powerful computers.

Email address: minfan@uestc.edu.cn, fmin@cems.uvm.edu (Fan Min).
URL: <http://cems.uvm.edu/fmin> (Fan Min).

2 Motivations

2.1 Advantages of Java

Java has been popular in the past decade, and many features make it suitable for a new version of DHSVM, some are listed below:

- (1) Easier to handle large systems. The source code of DHSVM is approximately 1M, which is fairly large. Dividing the system into a number of packages and then classes makes it easier to deal with.
- (2) Easier to pick out bugs. When using C, a developer may worry about too much about pointer operations and space allocation/releasing. Because they are where bugs most likely hide. While using Java, we do not worry about them any more. Moreover, Java is more strict on logic issues. For example, the boolean type is provided.
- (3) Easier to use. C version users might have a thorough understanding of the whole system before using it. While Java version users are not required to know too much about the detail. Also, a new user may find it is easier to start with a Java version. In fact, this is the most important reason for me to translate it.
- (4) Easier to write/read documents. Documents of DHSVM is very good, and comments in programs make them easy to understand. However, we can do even better with Java. Javadoc provides a good mechanism for extracting document from the comments in the program. Naturally, during the program writing process, respective documents should be deliberately written.
- (5) Easier to write graphics user interfaces (GUIs) documents. Menus and dialogs are easy to written, other graphics requirements are easy to meet.

3 Translation process

The translation process is not complex. The key issue is how to reduce the possibility of bug introduction. This section explains the whole complex. I use neither top-down nor bottom-up approaches. Instead, I follow my own steps of translation. However, this process is not one pass, that is, I have to go back frequently to programs I wrote earlier and modify them accordingly.

3.1 Classes

To identify classes, the most straightforward way is to begin with structures. The file `data.h` in DHSVM provides most data structures used in the system. So for each data structure, an independent public class is written. Each field of the structure corresponds with a member in the class.

3.2 Constants and global variables

There are some constants defined in `constants.h`. I define them as “public static final” in `dhsvm.common.Constants.java`. The naming policies in both C and Java are the same, e.g., `VON_KARMAN`. More constants are given in `settings.h`. I also include them in `dhsvm.common.Constants.java`.

A more complex issue is introduced by “enum KEYS” in `settings.h`. Since enumerate type of Java is different from C, I use constants instead.

Some “extern constants” are also defined in `constants.h`. However, they are not real constants. They are initialized at the beginning of each running, read from a file. Therefore I put them to `common.Globals.java`, and their naming policy is different. For example, `laiSnowMultiplier` instead of `LALSNOW_MULTIPLIER`.

Other constants are put to the class that they are mostly pertinent. For example, `SECPMIN` (second per minute) is put into `dhsvm.util.calendar.HydroDate`;

3.3 Methods

Each function in the C version naturally corresponds with a method in the Java version. The key issue is where they should be put. I explain through some examples as follows:

- (1) Simple functions/methods. By “simple” I mean that only simple data types (`int`, `double`, ...) are involved. Such functions are claimed as *public static* and could be defined in either `dhsvm.common.SimpleMethods.java` or `dhsvm.functions.Functions.java`. I choose the former. Another case is to deal with macros. For example, `#define INBASIN(x) ((x) != OUTSIDEBASIN)` is implemented as `public static boolean inBasin(int paraX)`.
- (2) One data structure/class related functions/methods. For example, `computeDayOfYear()` is only related to variables in `dhsvm.common.HydroDate`, and it is put into the class. Another good example is `CalcAerodynamic`, in the C version there are 9 parameters, while in the java version no

parameter is needed.

- (3) Multiple data structure/class related functions/methods. For example, `SnowInterception()` has 31 parameters in the C version. But I find while invoking it in `MassEnergyBalance()` (note that it is the only place invoking it), more often modified parameters are variables in `dhsvm.precipitation.PrecipitationPixel`, therefore I put it into this class, and these parameters should no longer be transferred. Other parameters are transferred by the object it belongs to if it is modified. For example, `LocalVeg` instead of `LocalVeg-¿Tcanopy` (see `MassEnergyBalance.c` lines 241-251) is used in Java (see `PrecipitationPixel.java` method `snowInterception()`). In this way the number of parameters is deduced to 11, quite easier to handle.

As long as member variables of an object are involved, there is a need to protect them from modifying outside the class. So in a matured Java version, I will define most members to be `private` instead of `public`. But before that, I should get the current version run correctly.

3.4 Packages

Now there are many packages, some packages only contain one class. In the near future, I will manage to put them into less packages.

3.5 Naming

I follow the naming rule set of Java. That is, the rule set used in Java Developer's Kit (JDK). Some examples are given in Table 1.

Table 1
Naming rules

Description	Examples
Constants	DELTAT, LEAF_DRIP_DIA, G
Classes	Constant, SoilPixel
Member variables	temperature, soilType, dt
Methods	sensibleHeatFlux, surfaceEnergyBalance
Parameters	paraTemperature, paraSoilType
Temporary variables	tempTemperature, tempSoilType

To distinguish among member variables, parameters and temporary variables (variables used in the scope of a method), I add a “para” before a parameter, e.g., `paraStartDate`, and a “temp” before a temporary variable,

In most cases, I use whole words rather than abbreviations. However, when the physical/mathematical meaning is obvious, I will use the notation. For example, G stands for gravity, which is a physical notation, dt stands for delta time which is a mathematical notation.

3.6 Documents

In my opinion, writing program is just like writing a paper or dissertation. Since it is firstly for people to read, and then to the computer. In fact, I have not run the C version until now. What I have done is to read the C source file and translate! It sounds not so reasonable, but with good documents of the C version, I am making it. As One objective of this what I want to do is to write more detailed documents.

4 Difficulties

During the translation process I encountered many difficulties, some are partly solved, while other should need help from people in the society.

- (1) File IO. Until now file input/output is still my headache. The file `tableio.lex` is rather hard for me to comprehend. I need a pure C version, or at least some exemplary input/out data files such that I can write programs to cope with them.
- (2) Long parameter list. I have partly solved this problem, as discussed in Subsection 3.3. However, for a more comprehensive version, I need to further reduce the list size.

5 Possible bug report

This section lists some possible bugs of the C version. I will consult Bart and other authors for these issue. After that, I will write the answers.

5.1 Not found methods

There are many files using function `Read2DMatrix`, but I cannot find its implementation. Only `Read2DMatrixBin` (in `FileIOBin.c`) and `Read2DMatrixNetCDF` (in `FileIONetCDF.c`) were found.

5.2 Useless variables/parameters

Table 2 lists some variables/parameters that are claimed while never used. Now I remove them from the Java version.

Table 2

Useless variables

No.	Position	Unused variable/parameters
1	InterceptionStorage.c	float * Height
2	InitNewMonth.c, function InitNewMonth	TOPOPIX **TopoMap, RADCLASSPIX **RadMap, IN
3	RouteRoad.c, function RouteRoad	SOILTABLE * SType
4	DHSVMChannel.c, function RouteChannel	SOILTABLE * SType This is due to RouteRoad

5.3 Useless functions/method

Table 3

Useless functions/methods

No.	Position	Unused variable/parameters
1	CalcTotalWater.c	calcTotalWater

5.4 Suspectable logic

Table 4 lists them.

Table 4

Suspectable logic

No.	File	Line Code	Question
1	CalcTopoIndex.c	147 if (INBASIN(TopoMap[coarsei][coarsej].Mask))	Does it mean Mask > 0?
2	InitTables.c	577 if (impervious)	impervious is a float, is it good?

5.5 Suspectable error

VarID.c, line 187, there should be a 0 before the end of the line.

6 Conclusions

References

- [1] B. Nijssen, P. Storck, DHSVM 3.0, Washington University.