JavaSpaces NetBeans — a Linda Workbench for Distributed Programming Course

Magdalena Dukielska
Google Inc.
magdalenad@google.com *

Jacek Sroka
University of Warsaw
sroka@mimuw.edu.pl

ABSTRACT
In this paper we introduce the JavaSpaces NetBeans IDE (JSN) which integrates the JavaSpaces technology, an implementation of Linda principles in Java, with the NetBeans[1] IDE. JSN is a didactic tool for practical assignments during distributed programming courses. It hides advanced aspects of JavaSpaces configuration and lets students focus on interprocess coordination. An important component of JSN is a distributed debugger which can help to make concurrent programming classes easier to understand and more compelling.

Categories and Subject Descriptors
K.3.1 [**Computer Uses in Education**]: Tools, Instructional Technologies, Active Learning

General Terms
Design, Experimentation, Human Factors

Keywords
distributed programming, Linda, JavaSpaces, integrated development environment, NetBeans, distributed debugger

1. INTRODUCTION
During her first years of studies every computer science student learns the principles of distributed programming, which is usually based on discussing solutions of classical problems as the dining philosophers. Skills obtained during those classes are in a high demand by the industry, where coordination and communication between processes are used on a daily basis to solve problems arising from the scale of contemporary processing tasks.

Yet, for many students distributed programming is difficult. Their imagination is put to a test when they are required to solve tricky problems by writing algorithms on paper and analyzing them in their minds. Nonetheless, there are many possible curriculum improvements which can mitigate these initial difficulties. The most obvious one is the choice of a readable and powerful formalism best suited to discussed problems. Linda [8] is a coordination language typically described during the distributed programming course which allows students to represent most problems in a very elegant way and which can facilitate their first steps into the concurrent world.

Even though some implementations of Linda's tuplespace model exist for popular programming languages, like JavaSpaces [7] for the Java language, they are rarely used by students. The main reason is the amount of additional work that has to be dedicated to setting up, managing and connecting to a tuplespace. Unfortunately, there is not enough time to practice such administrative tasks during the class because of the number of other formalisms, languages and ideas included in the distributed programming curriculum, like CSP, Ada, semaphores and monitors. Furthermore, there is a lack of supplementary tools which could make it easier for students to debug or test their distributed algorithms.

In this paper we describe JavaSpaces NetBeans IDE (JSN) [2] — a JavaSpaces integrated programming environment. JSN provides a convenient way for instructors to introduce students into practical concurrent programming in the Linda tuplespace model. The tool hides complicated configuration issues and lets students concentrate on designing coordination protocols. Moreover, JSN includes a distributed debugger which allows students to easily test their programs and to discover particular synchronization issues.

2. LINDA AND JAVASPACES
In this section we provide a brief introduction to the Linda coordination language and JavaSpaces — its Java implementation.

2.1 Linda
Linda [8] was developed by David Gelernter on Yale University in mid-1980s. It is a flexible and elegant model for parallel programming obtained by adding synchronized operations on a so-called tuplespace to the base programming language like Java or C++. The whole model is built upon notions of a *tuple* and *tuplespace*.

*This work was done at the University of Warsaw.
[1]See `http://netbeans.org/`
[2]See `http://javaspaces-netbeans.googlecode.com`

A tuple is a sequence of data objects of certain types. The sequence of these types is called the tuple's signature. For example, a tuple `("aa", 15, "bb")` has the signature `(string, int, string)`. A tuplespace is an environment for storing tuples — a sort of virtual shared memory — supplied with a kind of a search interface.

In Linda all interprocess communication goes through a tuplespace. Typically, each process uses a tuplespace from which it collects input data and where it deposits results of the computation it performs. A tuplespace has a very simple interface: processes can insert new tuples by using the `out` operation, withdraw an existing tuple matching a given pattern with the `in` operation, or read the contents of a tuple while the tuple itself remains available for other processes with the `read` operation. Both `in` and `read` operations can be either blocking or non-blocking. An associative matching algorithm based on the tuple's contents is used to check whether a particular tuple matches the pattern used by a process for an input operation.

Many classical problems have simple and easy to understand Linda solutions. Below we present a correct implementation of the five dining philosophers problem using four tickets guarding the entrance to the dining room. Each philosopher is represented by a process, while forks and tickets are tuples. Initially, the tuplespace contains 5 fork tuples with indexes 0 to 4 and 4 ticket tuples. The philosopher's code is:

```
void philosopher(int i) {
 while (true) {
  think();
  in("ticket"); in("fork", i); in("fork", (i+1)mod 5);
  eat();
  out("fork", (i+1)mod 5); out("fork", i); out("ticket");
 }
}
```

The major advantage of Linda over other formalisms like message passing which are also presented during distributed programming classes is loose coupling between components of the resulting system. Processes in Linda do not need to establish connections to one another. Communication is asynchronous and the system can be easily extended to conform to changing requirements of the environment in which it is running. In this respect, Linda programs match perfectly what is expected from real-world applications developed by companies like Google. A more detailed description of Linda programming model with its history and recent developments can be found in [13, 14].

2.2 JavaSpaces

There are multiple Linda implementations available for many languages like C, C++, Prolog, Python, Scala and Java. For a survey of Java implementations see [14]. One of them is JavaSpaces [3, 5, 7] created by Sun Microsystems.

Tuples in JavaSpaces are Java objects. Their signatures are classes that have a no-argument constructor and public fields of serializable types. Such requirements are implications of the tuple matching algorithm used in JavaSpaces. For example, a simple JavaSpaces implementation of a fork tuple for use by philosophers is:

```
public class Fork implements net.jini.core.entry.Entry {
  public Integer id;
  public Fork(Integer id) { this.id = id; }
}
```

The pattern provided by a process as an argument to an input operation is also an object of some tuple class like `Fork`. A tuple matches the pattern, if values of its public fields are equal to those specified in the pattern. A `null` value in the pattern is treated as a wildcard matching any value of a real tuple. For example, the following code inserts a `Fork` tuple with id 3 and then withdraws another `Fork` tuple with id 4 from the tuplespace:

```
JavaSpace space = ...  // here tuplespace is initialized
Fork tuple = new Fork(3);
// tuple will remain in the space forever
space.write(tuple, null, Lease.FOREVER);

Fork template = new Fork(4);
// wait if there is no matching tuple
Fork result =
  (Fork)space.take(template, null, Long.MAX_VALUE);
```

3. PROJECT OBJECTIVES

In this section we present three main objectives we wanted to achieve with the JSN project.

Our main goal was to create a Linda based programming environment that would allow students to learn by creating and testing working examples without being bothered with administrative, low-level or technology-oriented details. Such an environment was needed because using practical Linda implementations takes lots of introductory effort. This is caused by the fact that most implementations focus on building sophisticated commercial systems and require expert knowledge of their underlying principles and often many other fields as well. JavaSpaces is affected by these issues too. One of the reasons is that Jini technology [5, 11], on which JavaSpaces is based, aims at solving multiple network problems, and using it takes experience in other Java technologies like RMI. Before a tuplespace is started several other Jini-specific services need to be set up with appropriate parameters which makes the whole configuration cumbersome. Besides, even the simplest application requires a large amount of boilerplate code before a reference to a usable tuplespace is obtained, see [9] for some examples.

On the other hand, there exist many programming environments and tools for distributed programming courses like [1, 2, 12]. There were earlier attempts to provide classroom tools using Pascal-based Linda [10]. However, students are not familiar with these tools and need to invest additional time into getting used to them, even though they are moderately simple as they are designed for education. Unfortunately, for many students learning how to use such tools is not interesting. Due to the limitations of the didactics technology, they do not profit during their further studies and professional career from mastering educational tools. This is why the second objective of the JSN project was to stay close to the current professional trends and design patterns to make JSN more interesting for students. For that reason JSN: (1) uses Java instead of Pascal as the base language, (2) does not extend or modify the base language, but uses an existing and mature JavaSpaces library, (3) is based on an existing and popular NetBeans IDE, and (4) uses Java annotations and the Dependency Injection design pattern.

Our final objective is related to a fact that, although a large number of distributed algorithms has concise implementations, understanding what happens when these algorithms are executed concurrently is difficult. The standard approach is to extend the implementation with some logging routines and analyze traces from the distributed execution. Yet, this by itself is laborious and requires extra effort to enforce interesting interleaving of operations. At the same time we support the viewpoint presented in [4] that in many cases debugging an incorrect code and explaining the misconceptions in an argument has a very high didactic value. For that reason, we decided to include in JSN a distributed debugger allowing students to simulate selected system behaviors, so that it is easy for them to 'play with examples'.

4. THE JAVASPACES NETBEANS PROJECT

In this section we provide a concise overview of JSN's features.

4.1 Annotations

One of the biggest barriers for students when using JavaSpaces is complex and tedious tuplespace configuration. JSN deals with this problem with help of Java 5.0 annotations. Annotations in Java offer a simple and extensible syntax for adding metadata to classes, methods, variables, parameters and packages[3]. The main uses of annotation in JSN include: (1) providing declarative access to tuplespaces (with the popular Dependency Injection [6] design pattern), (2) performing necessary environment configuration, and (3) controlling the debugging support. The major annotations offered by JSN are:

- `@Space(name="a")` – if placed on a field of type `JavaSpace`, the field will be initialized with a reference to a tuplespace with the specified name when it is accessed for the first time;
- `@JavaSpaces(policy="let.all", spaces={"a","b"})` – placed on the main method of a program, performs the environment configuration, e.g., sets all necessary security properties for specified tuplespaces;
- `@JavaSpacesDebug` – placed on the main method, turns on/off debugging support, which we describe in more detail in Section 4.6.

An example of a complete implementation of `Philosopher` class using the annotations is:

```
@JavaSpaces
public class Philosopher {
  @Space(name="phil") private static JavaSpace space;
  private Ticket ticket;
  private Fork leftFork;
  private Fork rightFork;

  public Philosopher(int id, int num) {
    ticket = new Ticket();
    leftFork = new Fork(id);
    rightFork = new Fork((id + 1) % num)
  }

  private void eat() {...}
  private void think() {...}
  public void work() throws Exception {
    while (true) {
      think();
      space.take(ticket, null, Long.MAX_VALUE);
      space.take(leftFork, null, Long.MAX_VALUE);
      space.take(rightFork, null, Long.MAX_VALUE);
      eat();
      space.write(rightFork, null, Lease.FOREVER);
      space.write(leftFork, null, Lease.FOREVER);
      space.write(ticket, null, Lease.FOREVER);
    }
  }
```

[3]Java 5.0 annotations are just metadata added to the code. Their contract is realized in JSN with the use of an aspect-oriented programming language AspectJ.

4.2 Integrated JavaSpaces server

JSN fully integrates a JavaSpaces server[4] and allows to control its operations from the IDE. Using the server does not require any special installation or configuration because all necessary libraries and files are already included in JSN. JavaSpaces server control is based on a default NetBeans interface for web and application servers (see SERVICES tab in Figure 1). This way students gain experience in operating the popular NetBeans IDE. This is also a major facilitation for students who have already used this mechanism to control other servers available in NetBeans by default like Tomcat.

Blitz is presented as a new node named BUNDLED BLITZ in the SERVICES tab. The administrative operations like starting the server, creating or deleting a tuplespace and clearing tuplespace's contents are available through the context menu of the node.

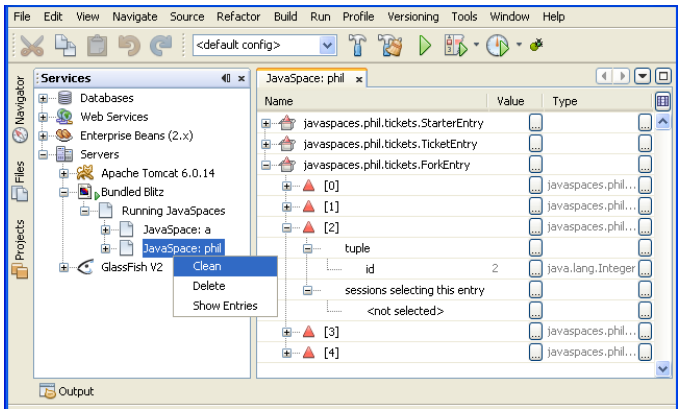

Figure 1: Browsing one of tuplespaces of the bundled Blitz server

4.3 Tuple browser

Once a tuplespace is started, students can use the *tuple browser* to inspect its contents (see Figure 1) which is presented as a tree. At the top level there is one node for each tuple signature. Expanding the signature's node reveals the list of individual tuples of the given type. Tuples can be further expanded to inspect their values. The tuple browser makes it easy to track the contents of a tuplespace which is not available in JavaSpaces by default. It also helps students to observe the impact that each process has on a tuplespace and how different processes communicate and cooperate.

[4]We have chosen Blitz — an open source project lead by Dan Creswell, see `http://www.dancres.org/blitz`.

4.4 Project template

Together with the JSN distribution students obtain example projects and a template for creating their own projects. The example projects allow students to quickly get acquainted with the basic features of JSN projects. The template references all Blitz- and JavaSpaces-specific libraries which may be needed when doing assignments. Furthermore, it contains a modified build file which is responsible for compilation and execution of the application. An AspectJ compiler is used instead of the standard one to make sure that JSN aspects used to implement JSN annotations (see Section 4.1) are integrated correctly with the student's code. Additionally, the project template module facilitates execution of JavaSpaces projects. For each project a RUN ON BLITZ action is provided which automatically starts the Blitz server and all tuplespaces necessary for the project, if they are not running yet. Afterward, the code is compiled and executed. This is a convenience versus having to first start the Blitz server, then the required tuplespaces, and finally compiling and running the code.

4.5 Graphical signatures editor

As discussed in Section 2.2, a JavaSpaces tuple is a class which has to conform to several rules. JSN includes a tuple signatures editor that allows students to visually define tuples from the very first day of classes. The correct code is generated by JSN which takes care that all requirements enforced by JavaSpaces are fulfilled.

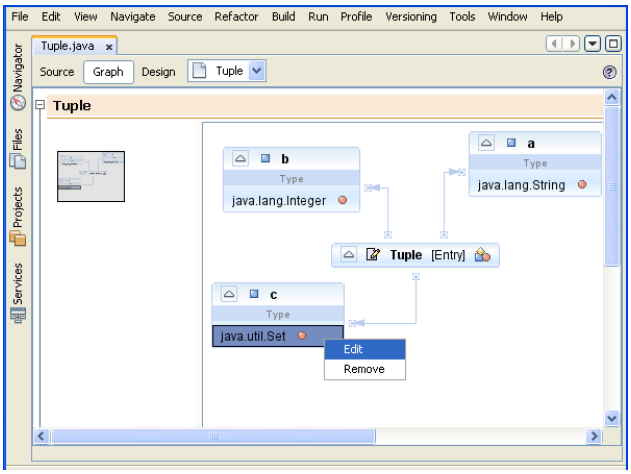

Figure 2: Graph view of the tuple editor

The editor offers following views: *source* – a default NetBeans view of Java source code with standard features like syntax completion, *design* – a table listing all fields with actions to add, edit or remove them, and *graph* – a readable graph (see Figure 2) with the same functionality as the form view. All three views are automatically synchronized, so that changes in one of them are propagated to the other two. Besides, when the file is being saved, the editor enhances the code to comply with JavaSpaces requirements for tuples, e.g., a no-argument constructor can be generated or field's type can be changed from `int` to `Integer`.

The form and graph views of the signatures editor allow students to concentrate on the optimal design of the fields that are necessary to solve a given problem. In fact, all the code can be automatically generated by the editor. Moreover, if students work with example projects or extend projects that are partially implemented, they can use the graphical views to quickly acquaint themselves with the signatures.

4.6 Distributed debugger

One of the most important features of JSN is its distributed debugger. It not only makes testing programs that use tuplespaces easier, but also it allows students to simulate arbitrary sequences of tuplespace operations. In contrast to a standard NetBeans debugger the JSN debugger was designed for managing multiple concurrent processes that communicate with one another through a tuplespace.

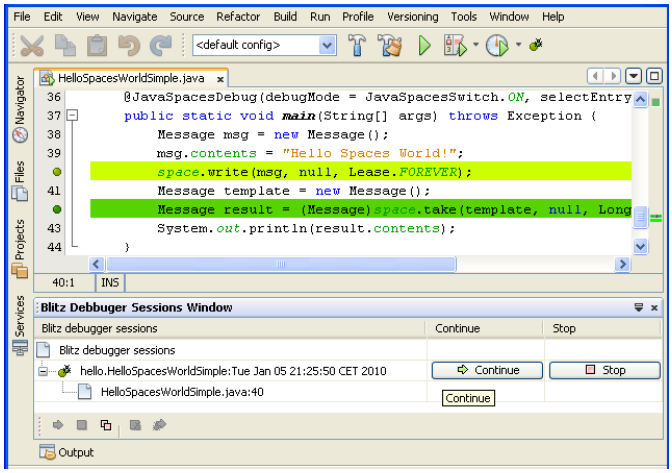

Figure 3: Example debugging session

A session with the JSN debugger typically starts with setting breakpoints on selected tuplespace operations. When the execution reaches such an operation the process will be suspended until the user resumes it. The described breakpoints are parallel to standard debugging facilities of NetBeans, which can still be used. A special view presents JSN breakpoints and allows users to manage them, e.g., enable, disable or remove.

The debugged code should contain a `@JavaSpacesDebug` annotation on the main method of the program. Selecting DEBUG ON BLITZ action from the context menu starts a new JSN debugging session. Each started process automatically obtains a unique identifier. The current state of all running processes is visible in the *Blitz Debugger Sessions Window* (see Figure 3). It shows whether a given process is running and if not, on which breakpoint it was stopped. Once the execution of a process is suspended by a breakpoint, a CONTINUE button appears in the sessions window which lets to resume the process. By resuming processes in a given order different sequences of synchronization operations can be simulated.

4.6.1 Select tuple mechanism

As there may be many different tuples matching a pattern with wildcards, the choice of a specific one for an input operation can have important consequences for the system's execution. During normal operation matching tuples get selected in a nondeterministic way. If a process is stopped on an input operation, the JSN debugger lets the user make this

choice. This is achieved by the integration with the tuple browser (see Section 4.3) and allows to guide execution of the system into desired scenarios. Such functionality can be used for testing of the correctness of a solution by enforcing the most inconvenient selections.

4.6.2 *Record and replay mechanisms*

To make it even easier for instructors to demonstrate classical concurrent problems, JSN allows for recording and later replaying of sequences of synchronization operations. An instructor can prepare example implementations of some problems and record particularly interesting sequences of operations to be examined by students during the class. A recorded execution can be edited by hand later on. An example replay of a previously saved debugging session is presented in Figure 4.

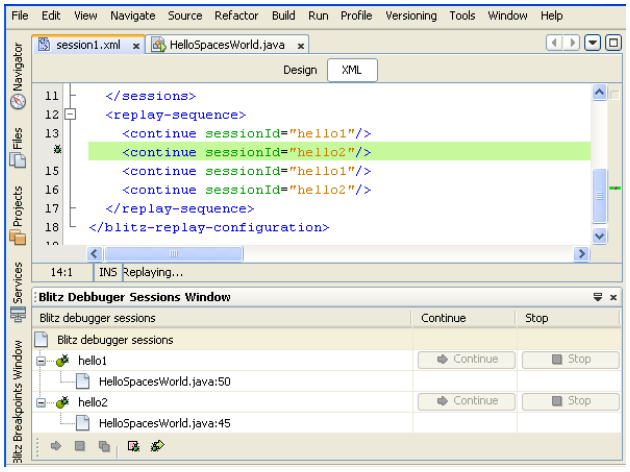

Figure 4: Replaying a saved debugging session

In an example use case for this feature, the instructor can prepare an incorrect solution for the dining philosophers problem which does not use tickets and is therefore susceptible to a deadlock. Then, the sequence of tuplespace operations leading to the deadlock – in this case all philosophers taking the left fork first — can be simulated during the class. When the simulation is finished, students can observe that no philosopher is able to continue its execution. In a similar way starvation can be demonstrated. This functionality can also be used in assignments where students are supposed to find an error in a given implementation and to record the sequence of operations leading to it.

5. FURTHER RESEARCH

The main directions we intend to focus our attention on while continuing the JSN project include: (1) extending the record and replay mechanism with recording of particular tuple selection choices (2) adding new visualization views and animations, e.g., a visualization of topology of the distributed system with animation of traveling messages, (3) designing of formal verification and model checking procedures based on exploration of the possible state space, and (4) developing a visualization for the forthcoming formal verification mechanism, possibly based on a Petri net. Finally, JSN needs to be constantly developed to follow improvements and enhancements of the NetBeans IDE itself and its underlying framework — the NetBeans Platform.

6. ACKNOWLEDGMENTS

The development of JavaSpaces NetBeans was partially sponsored by Sun Microsystems as a part of NetBeans Innovators Grant process. The authors are also grateful to the NetBeans Dream Team members for their support.

7. REFERENCES

[1] M. Ben-Ari. A suite of tools for teaching concurrency. In *ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*, pages 251–251, New York, NY, USA, 2004. ACM.

[2] M. Ben-Ari and S. Silverman. Dplab: an environment for distributed programming. In *ITiCSE '99: Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, pages 91–94, New York, NY, USA, 1999. ACM.

[3] P. Bishop and N. Warren. *JavaSpaces in Practice*. Pearson Education, Essex, UK, UK, 2002.

[4] A. Fekete. Using counter-examples in the data structures course. In *ACE '03: Proceedings of the fifth Australasian conference on Computing education*, pages 179–186, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[5] R. Flenner. *Jini and JavaSpaces Application Development*. Sams, Indianapolis, IN, USA, 2001.

[6] M. Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, 2004.

[7] E. Freeman, K. Arnold, and S. Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[8] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[9] S. Hupfer. The nuts and bolts of compiling and running javaspaces programs. SDN Online Article `http://java.sun.com/developer/technicalArticles/jini/javaspaces`, 2000.

[10] C. McDonald. Teaching concurrency with joyce and linda. In *SIGCSE '92: Proceedings of the twenty-third SIGCSE technical symposium on Computer science education*, pages 46–52, New York, NY, USA, 1992. ACM.

[11] J. Newmarch. *Foundations of Jini 2 Programming*. Apress, Berkely, CA, USA, 2006.

[12] R. Oechsle and T. Gottwald. Disaster (distributed algorithms simulation terrain): a platform for the implementation of distributed algorithms. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 44–48, New York, NY, USA, 2005. ACM.

[13] G. Wells. Coordination languages: Back to the future with linda. 2005.

[14] G. C. Wells, A. G. Chalmers, and P. G. Clayton. Linda implementations in Java for concurrent systems: Research Articles. *Concurr. Comput. : Pract. Exper.*, 16(10):1005–1022, 2004.