ZJLib Tutorial

By Moritz Eysholdt

Table of Contents

Introduction

The library ZJLib allows to create Java5-applications based on a specification in Z-Notation in a straightforward way. It provides data types and operations defined in the Z-Notation and tools to quickly create a Command Line Interface (CLI) for Java-Z-Code.

ZJLib is open source and currently hosted at Google Code[1].

This document illustrates how to convert code in Z-Notation to Java code by comparing fragments of both. The complete code can be found at[2].

To read this document, you should have a solid understanding of Java 5 including generics and nested classes. Knowledge about the Z-Notation is also helpful.

In the following Pages the Z-code will be shown first, followed by the Java-Code and some explanations.

1 http://code.google.com/p/zjlib/
2 http://zjlib.googlecode.com/svn/trunk/zjlib/examples/assignment2/

Primitive Datatypes

[*PLAYER*, *NAME*]

```
public class Assignment2 {

	public static class PLAYER extends ZPrimitive {
	}
```

For this example, everything will be member of the class Assignment2. For every primitive data type in Z we create a empty class that inherits ZPrimotive. The type ZPrimitive contains a name-attribute and it is ensured that this name is unique for all instances of a single type. This uniqueness is required to refer to single objects just by identifying them via their name. As an descendant of ZPrimitive already has a name, we don't need to create al class for the Z-Primitive NAME.

Statespaces

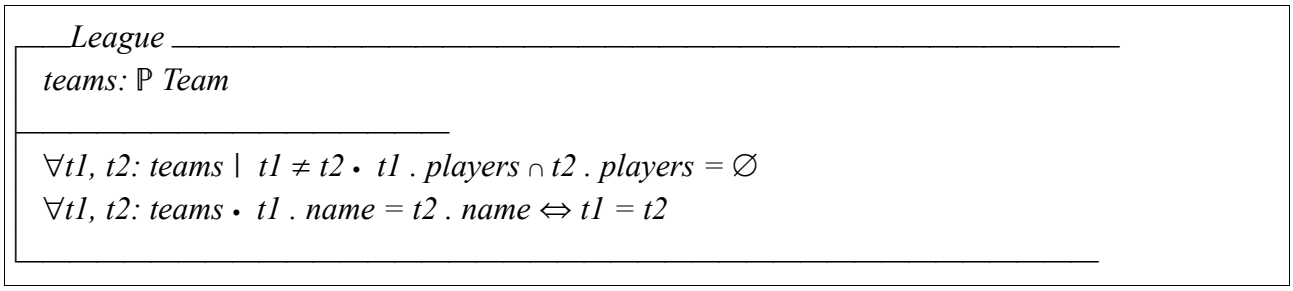
Team

$$players : \mathbb{P}\ PLAYER$$
$$name : NAME$$

```
	public static class Team extends ZPrimitive {
		public ZSet<PLAYER> players = Z.Set();

		public String toString() {
			return getName() + " player:" + players;
		}
	}
```

Every Z-Statespace becomes a Java-Class and every variable inside this state space becomes a member variable in Java. As descendants of ZPrimitive already have a name-attribute, we don't have to care about that anymore. To model sets and relationships ZJLib provides classes like ZSet and ZRelation. The class Z is the factory for all Z-related objects and provides a convenient syntax to create them.

Statespaces with state invariants

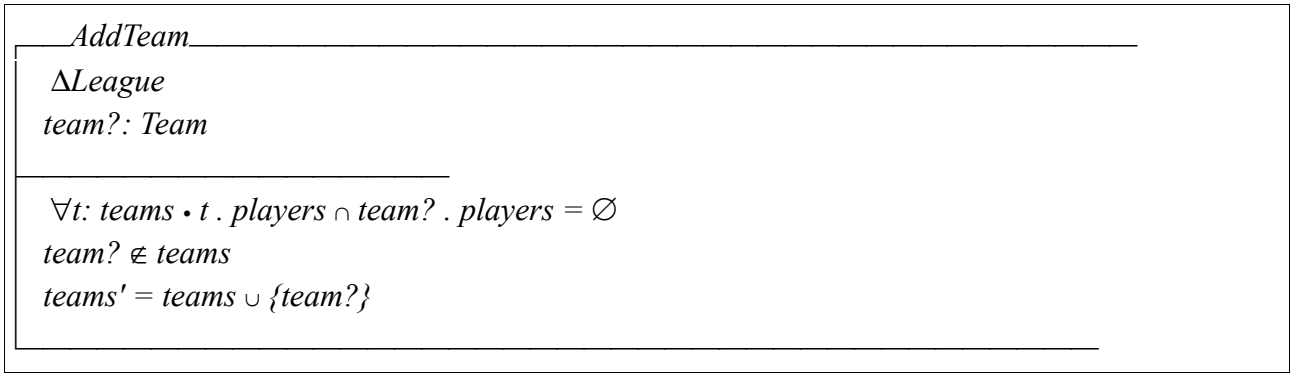



```
	public static class League {
		public ZSet<Team> teams = Z.Set();

		public boolean checkInvariants() {
			assert Z.forAll(teams, teams, new ZPredicate2<Team, Team>() {
				public boolean condition(Team t1, Team t2) {
					return !t1.equals(t2);
				}

				public boolean predicate(Team t1, Team t2) {
					return t1.players.intersection(t2.players).isEmpty();
				}
			});
			assert Z.forAll(teams, teams, new ZPredicate2<Team, Team>() {
				public boolean predicate(Team t1, Team t2) {
					return t1.getName().equals(t2.getName()) == (t1 ==
t2);
				}
			});
			return true;
		}
```

For the state invariants we create a method checkInvariants() (you choose the name). Every predicate is prefixed by Java's "assert" statement. This way predicates are only evaluated when assertions are enabled[3] in Java. The Z.forAll method iterated over all items contained in the supplied sets, in this case team and team. So the methods of the ZPredicate2 class are evaluated for all possible combinations of teams. Z.forAll returns true, if the predicate-method returns true for all cases and the predicate-method is only evaluated if the condition-method returns true. The checkInvariants-method has the return type boolean, so itself can be a used inside an assert-statement.

3 To enable Assertions in Java, start the JVM with the parameter "-ea"

Operations




```
public void addTeam(final Team team) {
	/* preparation */
	assert checkInvariants();
	ZSet<Team> teamsp = teams.clone();

	/* preconditions */
	assert Z.forAll(teams, new ZPredicate1<Team>() {
		public boolean predicate(Team t) {
			return t.players.intersection(team.players).isEmpty();
		}
	});
	assert !teams.isMember(team);

	/* implementation */
	teamsp.add(team);

	/* postconditions */
	assert teamsp.equals(teams.union(Z.Set(team)));

	/* finish */
	teams = teamsp;
	assert checkInvariants();
}
```

The implementation of every operation follows the following pattern:

1. Preparation: Make sure that the state invariants are assured and copy all variables that are about to be modified by this operation. We need to copy this variables to make be able to compare the original and the modified variable in the postconditions.
2. Preconditions: Check all preconditions via assert-statements.
3. Implementation: Do whatever this operation is supposed to do.
4. Postcondition: Check all postconditions via assert-statements. Post conditions usually compare the modified variables with the unmodified ones to ensure that the implantation has has done it's job correctly.
5. Finish: Copy the modified variables back to the the Statespace and check if all state invariants are still fulfilled.

Commandline-based User Interface

```
	public static void main(String[] args) {
		ZShellUI ui = new ZShellUI();
		League league = new League();
		createTestData(league);
		ui.setZClass(league);
		ui.runShell();
	}
```

Yes, that is really everything you have to implement for a Commandline-based user interface (CLI). Create a instance of ZShellUI, tell it which class to inspect and call runShell(). The CLI looks like this:

```
Commands for Class League:
  Viewable Variables:
    v0: ZSet teams
  Executable Operations:
    o0: checkInvariants
    o1: addTeam <Team>
    o2: deleteTeam <Team>
    o3: deletePlayer <PLAYER>
    o4: getPlayerCountForTeam <String>
    o5: getPlayerCount
  Other Commands:
    h: Print this help
    e: Print last exception
     q: Quit
```

The class ZShellUI uses Java-reflection to determine the public variables and methods implemented by the class that is supplied to setZClass(). Typing the command “v0” will list all items of the variable “teams”. Operations can be called by “o<number>” or the full name of the operation.

Further valid commands are:

```
addTeam "My Team"
deleteTeam "My Team"
o2 Thomas
o4 Lions
getPlayerCount
```

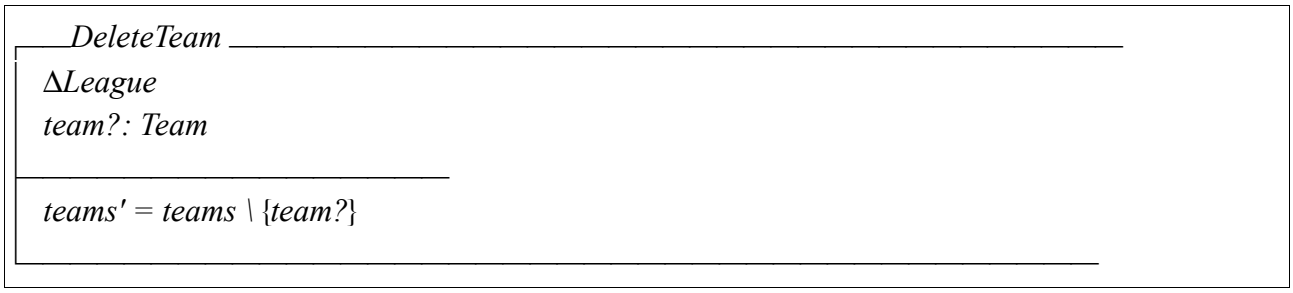
Create Testdata

```
	private static void createTestData(League l) {
		Team t1 = ZPrimitive.getInstance(Team.class, "Sharks");
		Team t2 = ZPrimitive.getInstance(Team.class, "Lions");
		Team t3 = ZPrimitive.getInstance(Team.class, "Bears");
		t1.players.add(ZPrimitive.getInstance(PLAYER.class, "Fritz Meyer"));
		t1.players.add(ZPrimitive.getInstance(PLAYER.class, "Max Mustermann"));
		t1.players.add(ZPrimitive.getInstance(PLAYER.class, "John Doe"));
		t2.players.add(ZPrimitive.getInstance(PLAYER.class, "Fritz"));
		t2.players.add(ZPrimitive.getInstance(PLAYER.class, "Hans"));
		t3.players.add(ZPrimitive.getInstance(PLAYER.class, "Thomas"));
		t3.players.add(ZPrimitive.getInstance(PLAYER.class, "Michael"));
		l.teams.add(t1);
		l.teams.add(t2);
		l.teams.add(t3);
	}
```

Creating testdata might look like this. It is important to notice that descendants of ZPrimitive are only instantiated by ZPrimitive.getInstance(<Class>, <Name>). This makes sure that for every name only one instance of an object exists.

Further Example Operations

deleteTeam




```
public void deleteTeam(Team team) {
        /* preparation */
        assert checkInvariants();
        ZSet<Team> teamsp = teams.clone();

        /* preconditions */
        /* none */

        /* implementation */
        teamsp.remove(team);

        /* postconditions */
        assert teamsp.equals(teams.substract((Z.Set(team))));

        /* finish */
        teams = teamsp;
        assert checkInvariants();
}
```

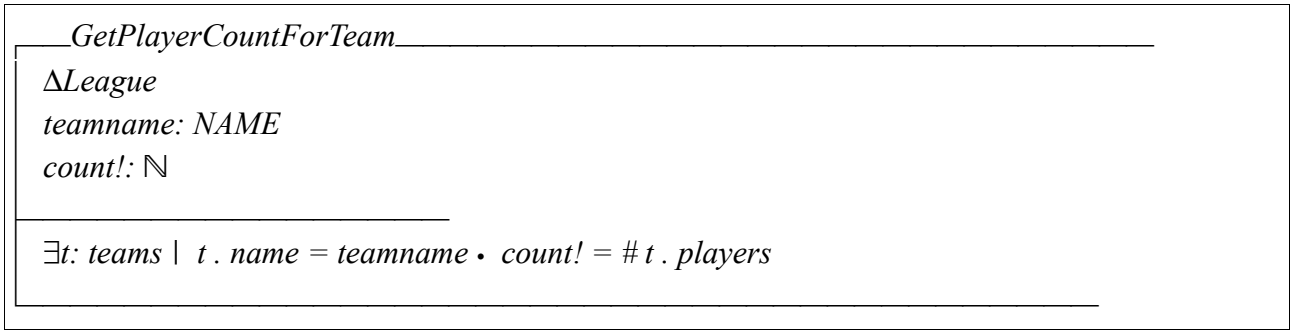
deletePlayer

DeletePlayer

$$
\begin{array}{l}
\Delta League \\
player?: PLAYER \\
\hline
\exists t: teams \mid player? \in t.players \\
\quad \bullet\ teams' \\
\qquad = teams \setminus \{t\} \\
\qquad \cup \{ n: Team \\
\qquad\quad \mid n.name = t.name \\
\qquad\quad \land n.players = t.players \setminus \{player?\} \}
\end{array}
$$

```
public void deletePlayer(final PLAYER player) {
	/* preparation */
	assert checkInvariants();
	final ZSet<Team> teamsp = teams.clone();

	/* preconditions */
	/* none */

	/* implementation */
	for (Team t : teamsp)
		if (t.players.isMember(player))
			t.players.remove(player);

	/* postconditions */
	Z.exists(teams, new ZPredicate1<Team>() {
		public boolean condition(Team t) {
			return t.players.isMember(player);
		}

		public boolean predicate(Team t) {
			Team n = ZPrimitive.getAnonymousInstance(Team.class, t
					.getName());
			n.players = t.players.difference(Z.Set(player));
			return teamsp.equals(teams.difference(Z.Set(t).union(
					Z.Set(n))));
		}
	});

	/* finish */
	teams = teamsp;
	assert checkInvariants();
}
```

getPlayerCountForTeam




```
public int getPlayerCountForTeam(final String teamname) {
	/* preparation */
	assert checkInvariants();
	final int count;

	/* preconditions */
	/* none */

	/* implementation */
	count = ZPrimitive.getInstance(Team.class, teamname).players
			.cardinality();

	/* postconditions */
	assert Z.exists(teams, new ZPredicate1<Team>() {
		public boolean condition(Team t) {
			return t.getName().equals(teamname);
		}

		public boolean predicate(Team t) {
			return count == t.players.cardinality();
		}
	});

	/* finish */
	assert checkInvariants();
	return count;
}
```

getPlayerCount

$$
\begin{array}{|l}
\textit{GetPlayerCount} \\
\hline
\Delta League \\
count!: \mathbb{N} \\
\hline
count! = \#\{\ p: PLAYER;\ t: teams \mid p \in t\ .\ players \bullet p\ \}
\end{array}
$$

```
public int getPlayerCount() {
	/* preparation */
	final int count;
	assert checkInvariants();
	/* preconditions */
	/* none */

	/* implementation */
	int c = 0;
	for (Team t : teams)
		c += t.players.cardinality();
	count = c;

	/* postconditions */
	assert count == Z.Set(teams, new ZSelect1<Team, PLAYER>() {
		public ZSet<PLAYER> select(Team t) {
			return t.players;
		}
	}).cardinality();

	/* finish */
	assert checkInvariants();
	return count;
}
```