Stepwise Refinement

Eric Roberts
CS 106A
January 8, 2010

Outline
1. Review Karel statement forms
2. Exercise: The `putBeeperLine` method
3. Stepwise refinement
4. Exercise: The `BanishWinter` program
5. Preconditions and postconditions
6. Story time: Past Karel Contest winners

Review: The Karel Language

<table>
<thead>
<tr>
<th>Built-in Karel commands:</th>
<th>Conditional statements:</th>
</tr>
</thead>
<tbody>
<tr>
<td>move();</td>
<td>if (condition) {</td>
</tr>
<tr>
<td>turnLeft();</td>
<td>statements executed if condition is true</td>
</tr>
<tr>
<td>putBeeper();</td>
<td>} else if (condition) {</td>
</tr>
<tr>
<td>printBeeper();</td>
<td>statements executed if condition is false</td>
</tr>
<tr>
<td>import statement.karel;</td>
<td>while (condition) {</td>
</tr>
<tr>
<td>/* Definition of the new class */</td>
<td>statements to be repeated</td>
</tr>
<tr>
<td>public class now extends Karel {</td>
<td>}</td>
</tr>
<tr>
<td>public void run() {</td>
<td>Method definition:</td>
</tr>
<tr>
<td>statements in the body of the method }</td>
<td>private void name() {</td>
</tr>
<tr>
<td>definition of private methods }</td>
<td>statements in the method body</td>
</tr>
</tbody>
</table>

Exercise: Creating a Beeper Line

- Write a method `putBeeperLine` that adds one beeper to every intersection up to the next wall.
- Your method should operate correctly no matter how far Karel is from the wall or what direction Karel is facing.
- Consider, for example, the following main program:

```
public void run() {
    putBeeperLine();
    turnLeft();
    putBeeperLine();
}
```

Stepwise Refinement

- The most effective way to solve a complex problem is to break it down into successively simpler subproblems.
- You start by breaking the whole task down into simpler parts.
- Some of those tasks may themselves need subdivision.
- This process is called stepwise refinement or decomposition.

Complete Task

- Subtask #1
- Subtask #2
- Subtask #3
- Subtask #1a
- Subtask #1b
Criteria for Choosing a Decomposition

1. *The proposed steps should be easy to explain.* One indication that you have succeeded is being able to find simple names.

2. *The steps should be as general as possible.* Programming tools get reused all the time. If your methods perform general tasks, they are much easier to reuse.

3. *The steps should make sense at the level of abstraction at which they are used.* If you have a method that does the right job but whose name doesn’t make sense in the context of the problem, it is probably worth defining a new method that calls the old one.

Exercise: Banishing Winter

- In this problem (which is described in detail in Handout #11), Karel is supposed to usher in springtime by placing bundles of leaves at the top of each “tree” in the world.
- Given this initial world, the final state should look like this:

Understanding the Problem

- One of the first things you need to do given a problem of this sort is to make sure you understand all the details.
- According to the handout, Karel stops when it runs out of beepers. Why couldn’t it just stop at the end of 1st Street?

The Top-Level Decomposition

- You can break this program down into two tasks that are executed repeatedly:
  - Find the next tree.
  - Decorate that tree with leaves.

Preconditions and Postconditions

- Many of the bugs that you are likely to have come from being careless about the conditions under which you use a particular method.
- As an example, it would be easy to forget the `turnLeft` call at the end of the `addLeavesToTree` method.
- To reduce the likelihood of such errors, it is useful to define pre- and postconditions for every method you write.
  - A *precondition* specifies something that must be true before a method is called.
  - A *postcondition* specifies something that must be true after the method call returns.