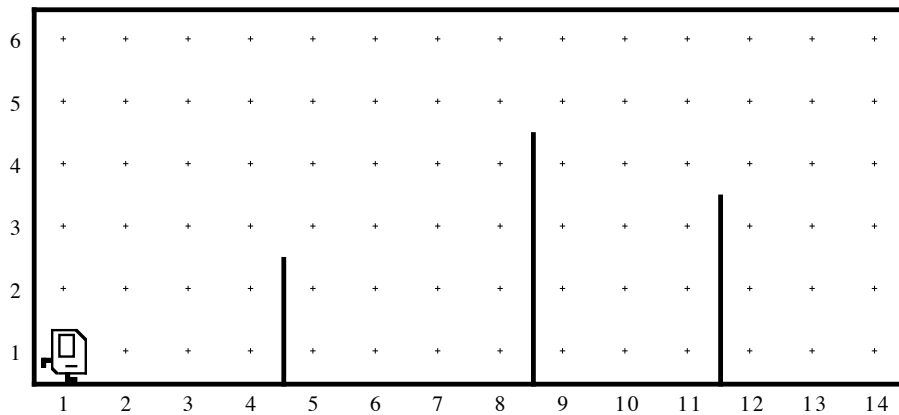


An Example of Stepwise Refinement

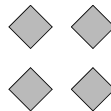
“sweet spring is your
time is my time is our
time for springtime is lovetime
and viva sweet love”

—e. e. cummings

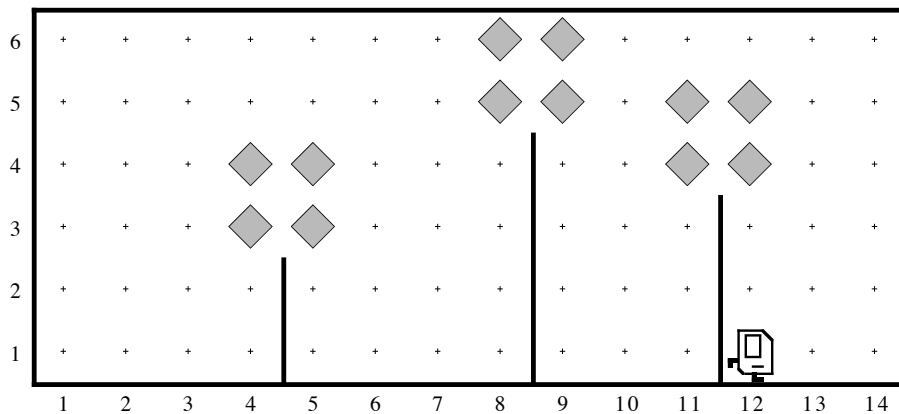
Now that we’re in spring quarter, Karel wants to celebrate the new season by helping add leaves to the trees in its world. The program we’ll develop today starts out with a winter world that looks something like this:



In this sample world, the vertical wall sections represent barren tree trunks. In an attempt to add some semblance of life back to the world, Karel has taken on the mission of adorning the barren trees with a new set of leaves represented by beepers. Karel’s plan is to climb each of the trees and adorn the top of each tree with a cluster of four leaves arranged in a square like this:



Thus, when Karel is done, the scene will look like this:



As in most Karel problems, the situation that Karel faces need not match exactly the one shown in the diagram. There may be more trees; Karel simply continues the process until there are no beepers left in the beeper bag. The trees may also be of different heights or spaced differently than the ones shown in the diagram. Your task is to design a program that is general enough to solve any such problem, subject to the following assumptions:

- Karel starts at the origin facing east, somewhere west of the first tree.
- The trees are always separated by at least two corners, so that the leaves at the top don't interfere with one another.
- The trees always end at least two corners below the top, so that the leaf cluster will not run into the top wall.
- Karel has just enough beepers to outfit all of the trees. The original number of beepers must therefore be four times the number of trees.
- Karel should finish facing east at the bottom of the last tree.

Think hard about what the parts of this program are and how you could break it down into simpler subproblems. What if there were only one tree? How does that simplify the problem, and how can you use the one-tree solution to help solve the more general case?

The solutions will be available on the web as Handout #11A.

Stepwise Refinement

Eric Roberts
CS 106A
April 6, 2012

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

Built-in Karel commands: <code>move();</code> <code>turnLeft();</code> <code>putBeeper();</code> <code>pickBeeper();</code>	Conditional statements: <pre>if (condition) { statements executed if condition is true }</pre> <pre>if (condition) { statements executed if condition is true } else { statements executed if condition is false }</pre>
Karel program structure: <pre>/* * Comments may be included anywhere in * the program between a slash-star and * the corresponding star-slash characters. */ import stanford.karel.*; /* Definition of the new class */ public class name extends Karel { public void run() { statements in the body of the method } definitions of private methods }</pre>	Iterative statements: <pre>for (int i = 0; i < count; i++) { statements to be repeated }</pre> <pre>while (condition) { statements to be repeated }</pre>
	Method definition: <pre>private void name () { statements in the method body }</pre>

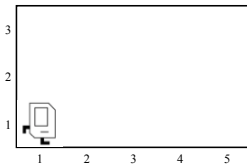
Review: The Karel Language

Karel condition names: <code>frontIsClear()</code> <code>frontIsBlocked()</code> <code>leftIsClear()</code> <code>leftIsBlocked()</code> <code>rightIsClear()</code> <code>rightIsBlocked()</code> <code>beepersPresent()</code> <code>noBeepersPresent()</code> <code>beepersInBag()</code> <code>noBeepersInBag()</code> <code>facingNorth()</code> <code>notFacingNorth()</code> <code>facingEast()</code> <code>notFacingEast()</code> <code>facingSouth()</code> <code>notFacingSouth()</code> <code>facingWest()</code> <code>notFacingWest()</code>	New commands in the SuperKarel class: <code>turnRight();</code> <code>turnAround();</code> <code>paintCorner(color);</code> New conditions in the SuperKarel class: <code>random();</code> <code>random(p);</code> <code>cornerColorIs(color)</code>
--	---

Exercise: Creating a Beeper Line

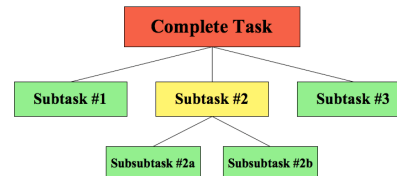
- Write a method `putBeeperLine` that adds one beeper to every intersection up to the next wall.
- Your method should operate correctly no matter how far Karel is from the wall or what direction Karel is facing.
- Consider, for example, the following main program:

```
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*.

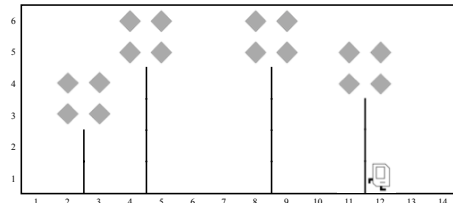


Criteria for Choosing a Decomposition

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.

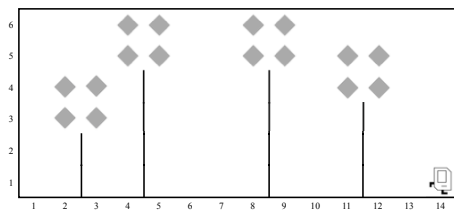
Exercise: Banishing Winter

- 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:



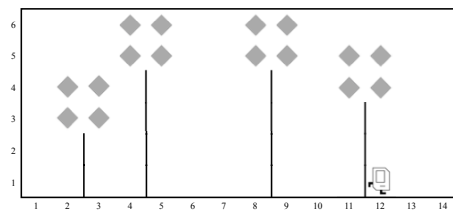
Understanding the Problem

- 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.