Java:

Learning to Program with Robots

Chapter 03: Developing Methods

Chapter Objectives

After studying this chapter, you should be able to:

- Use stepwise refinement to implement long or complex methods.
- Explain the advantages to using stepwise refinement.
- Use pseudocode to help design and reason about methods before code is written.
- Use multiple objects to solve a problem.
- Use inheritance to reduce duplication of code and increase flexibility.
- Explain why some methods should not be available to all clients and how to appropriately hide them.

An algorithm is a finite set of step-by-step instructions that specifies a process of moving from the initial situation to the final situation.

Everyday examples of algorithms:

- From a bottle of shampoo:

  wet hair with warm water

  gently work in the first application of shampoo

  rinse thoroughly and repeat

- From a spool of dental floss:

  wrap dental floss around your middle fingers

  firmly grasp floss with your index fingers

  forming a C-shape, carefully slide the floss up and down between your tooth and gum line

  gently slide the floss in between both sides of your teeth and repeat until finished

3.1: Characteristics of Good Algorithms

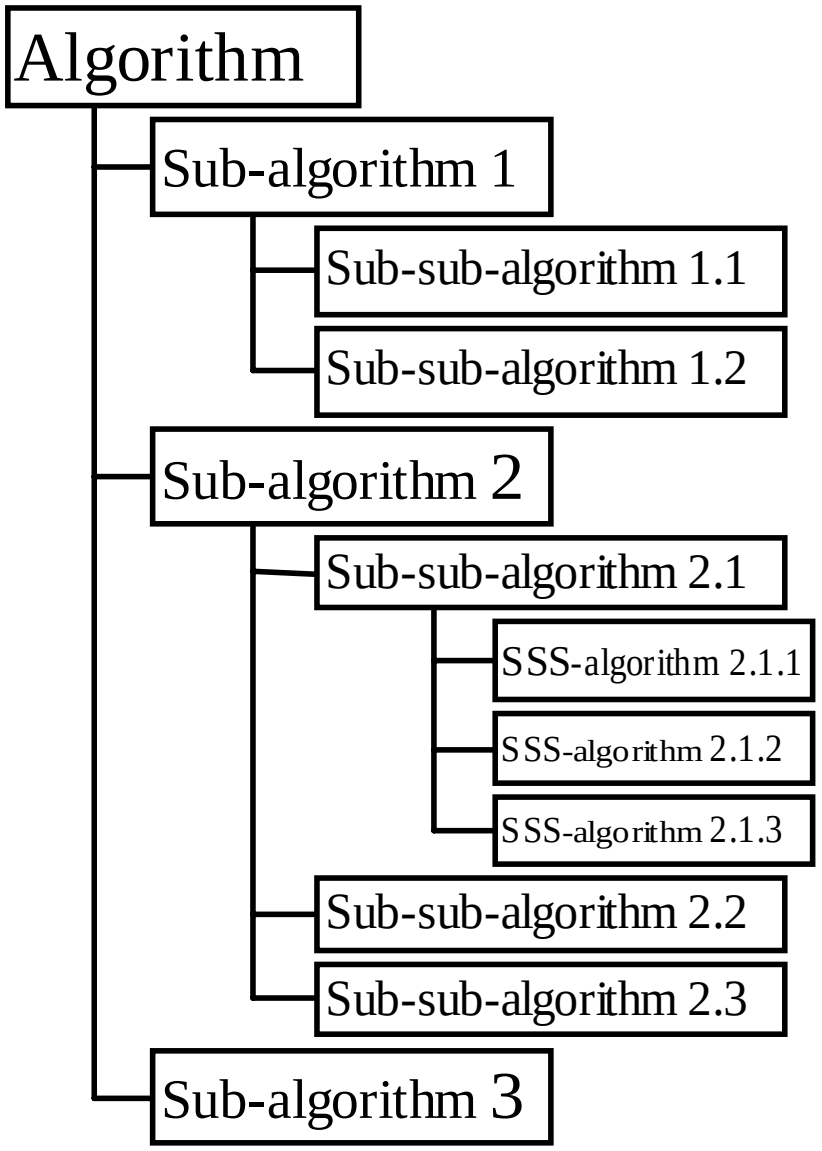
Good algorithms are:

- correct
- easy to read and understand
- easy to debug
- easy to modify to solve variations of the original task
- efficient

A computer program is one way of writing an algorithm so that it is precise enough to be executed by a computer.

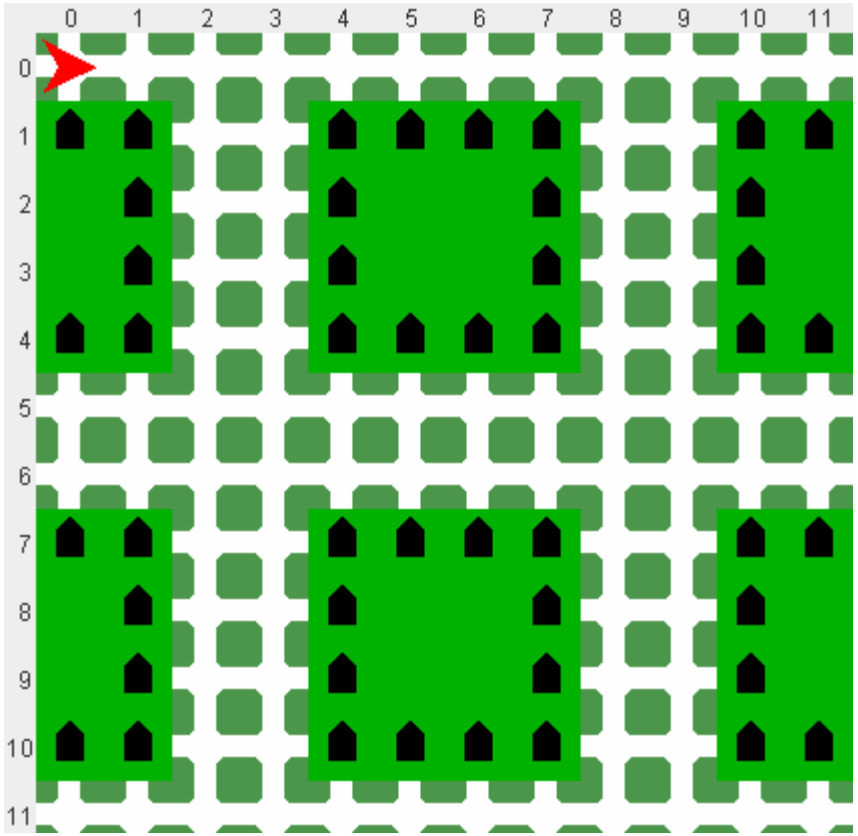
3.2: Stepwise Refinement

- Stepwise refinement is a method of constructing algorithms (and therefore computer programs and the methods they use).
- It decomposes a complex algorithm into smaller, simpler algorithms.
  - Construct sub-algorithms the same way (decompose into smaller, simpler algorithms). Do the same for sub-sub-algorithms.
  - After enough decomposition, the (sub)-algorithms become simple enough to solve using tools that are already available (e.g. **move**, **turnLeft**).

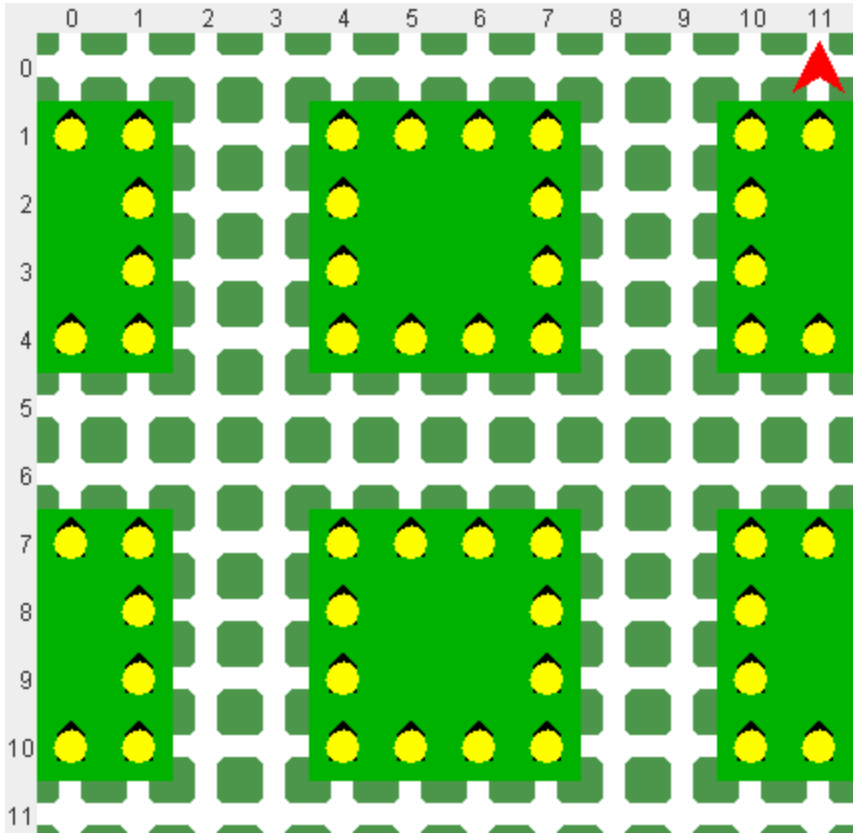


Case Study: Problem Description

You've taken a job delivering flyers for a local advertising agency. A robot to help with the work sure would be nice… The route includes all the houses shown below.



Initial Situation



Final Situation

It is assumed the robot will stay off the green grass as much as possible.

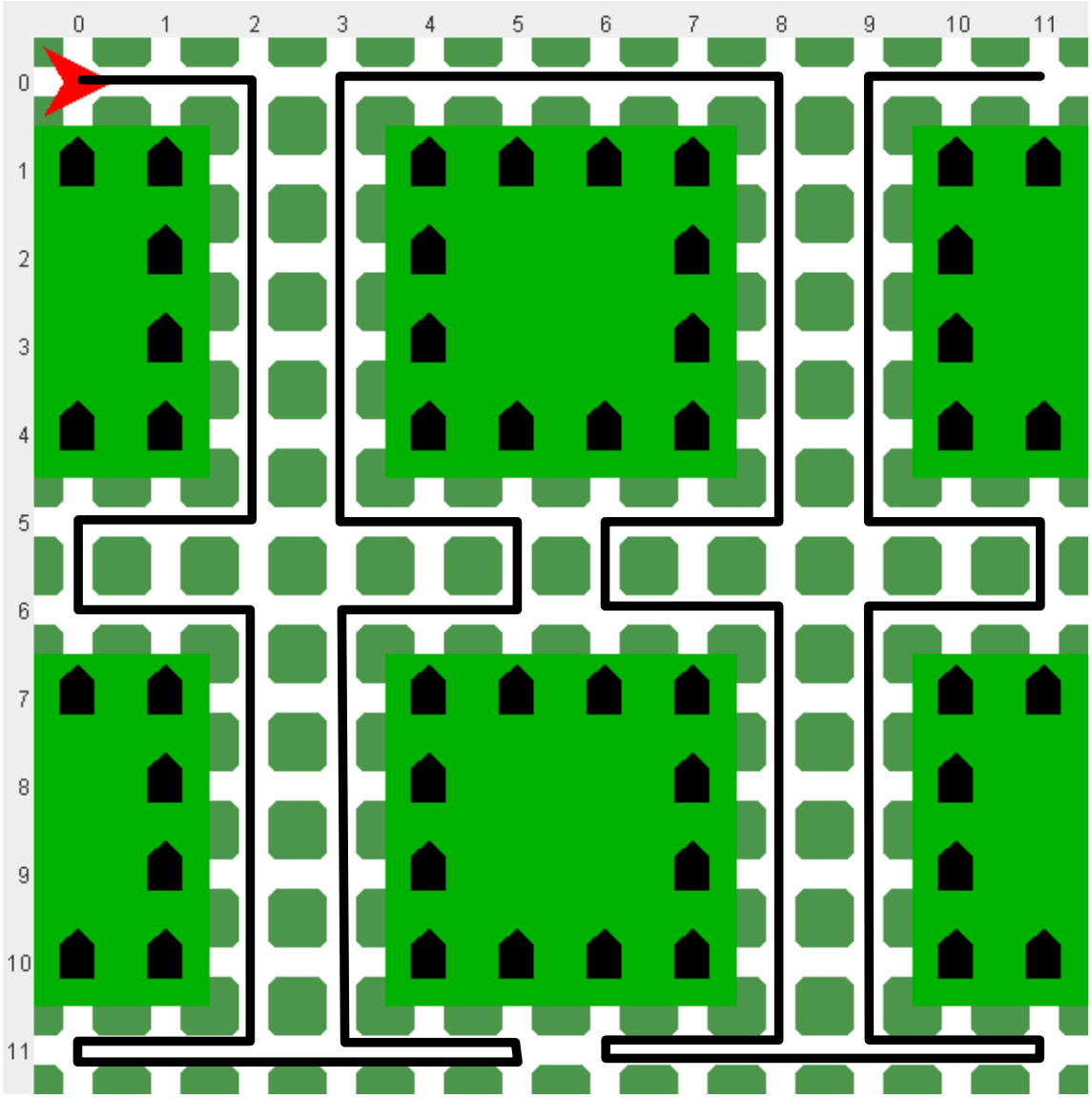
Case Study: Main method

```
import becker.robots.*;


/** Program a robot to deliver flyers.
* @author Byron Weber Becker */
public class DeliverFlyers
{
   public static void main(String[ ] args)
   {
      // Set up the route with the houses.  Create a DeliveryBot to do the work, complete with
      // flyers.  (The Route class extends City and therefore is a kind of City.)
      Route route = new Route();
      DeliveryBot karel = new DeliveryBot(route, 0, 0, Direction.EAST, 48);

      // Instruct the robot to deliver the flyers.
      karel.deliverFlyers();
   }
}
```

Case Study: Overall Strategy

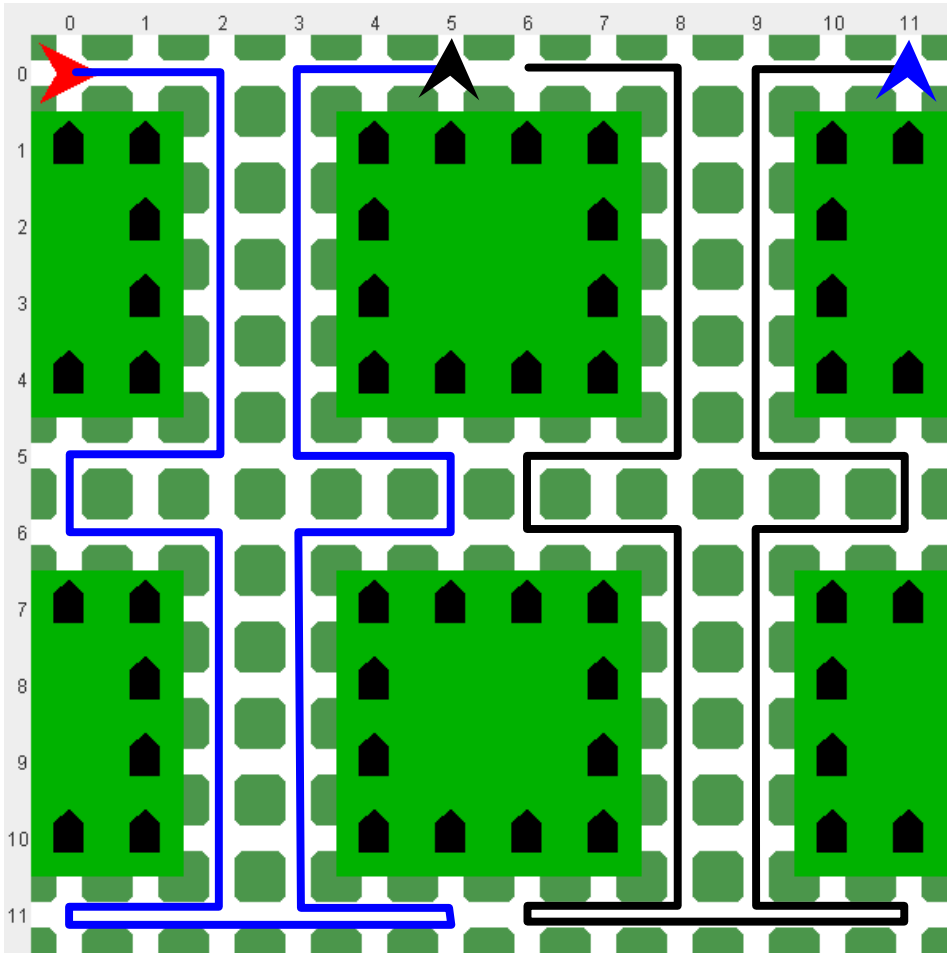
What path should the **DeliveryBot** follow? One option is shown below. Not shown is actually going up to each house to deliver the flyer and then returning to the road.

How can this complex algorithm (**deliverFlyers**) be decomposed into smaller, simpler sub-algorithms?



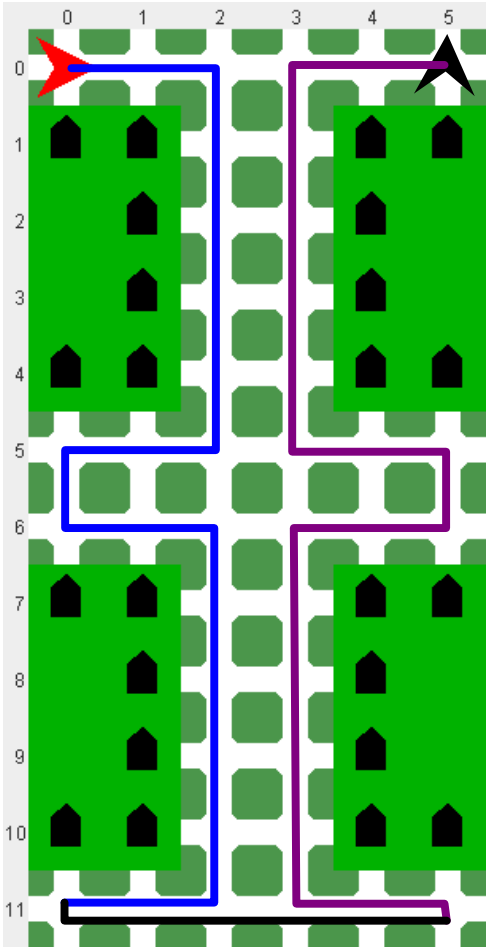
Case Study: Deliver Flyers

```
import becker.robots.*;
/** A robot to deliver flyers on a prescribed route.
 * @author Byron Weber Becker */
public class DeliveryBot extends RobotSE
{  /** Construct a robot to deliver flyers. */
   public DeliveryBot(City aCity, int aStr, int anAve, Direction aDir, int numThings)
   {  super(aCity, aStr, anAve, aDir, numThings);
   }

   /** Deliver flyers to all the houses on
   *   a prescribed route. */
   public void deliverFlyers()
   {  this.deliverOneAvenue();
      this.turnRight();
      this.move();
      this.deliverOneAvenue();
   }

   /**Deliver flyers to one avenue (plus the
   *   side streets).
   public void deliverOneAvenue()
   {  // Stub to permit compilation.
   }
}
```



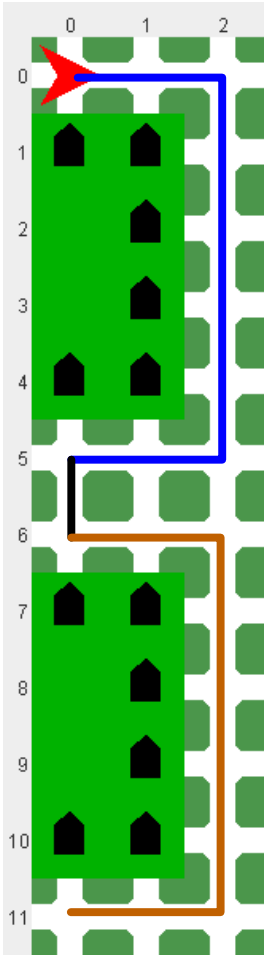
Case Study: Deliver One Avenue

```
import becker.robots.*;
public class DeliveryBot extends RobotSE
{  public DeliveryBot…

   public void deliverFlyers()
   {  this.deliverOneAvenue();
      this.turnRight();
      this.move();
      this.deliverOneAvenue();
   }

   public void deliverOneAvenue()
   {  this.deliverOneSide();
      this.goToOtherSide();
      this.deliverOneSide();
   }

   public void deliverOneSide()
   {
   }

   public void goToOtherSide()…
}
```



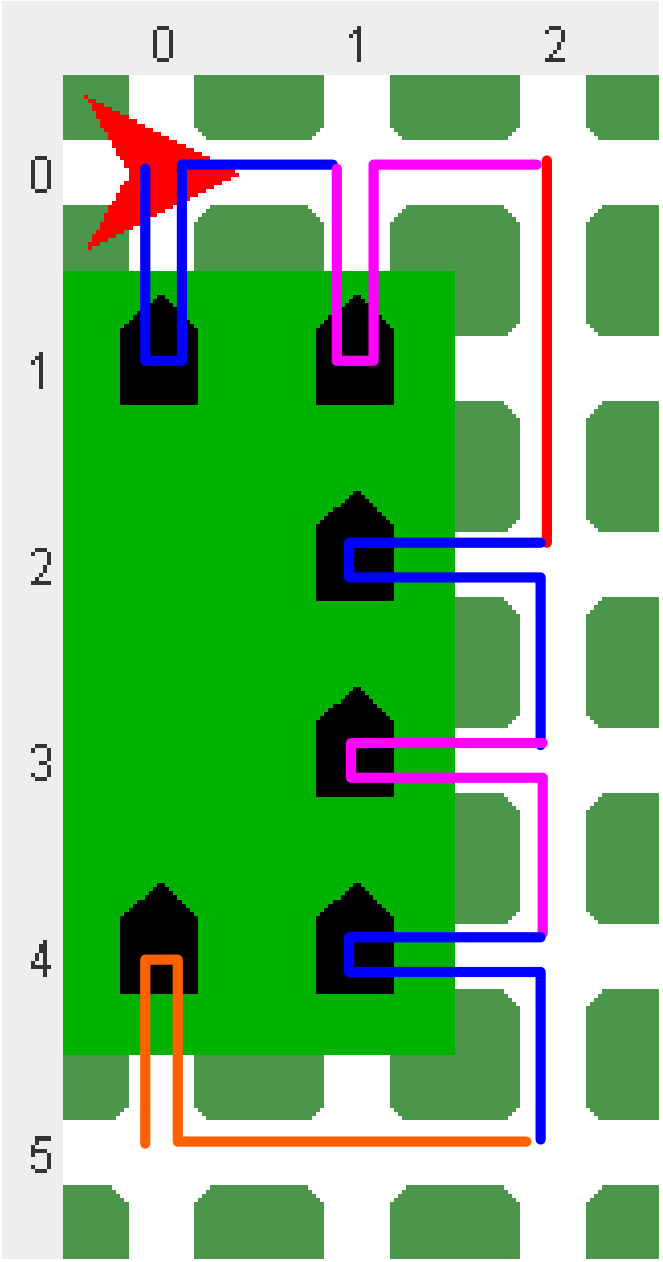
Case Study: Deliver One Side

```
import becker.robots.*;
public class DeliveryBot extends RobotSE
{  public DeliveryBot…
   public void deliverFlyers()…

   public void deliverOneAvenue()
   {  this.deliverOneSide();
      this.goToOtherSide();
      this.deliverOneSide();
   }

   public void deliverOneSide()
   {  this.deliverBlock();
      this.crossStreet();
      this.deliverBlock();
   }

   public void deliverBlock()…
   public void crossStreet()…
   public void goToOtherSide()…
}
```



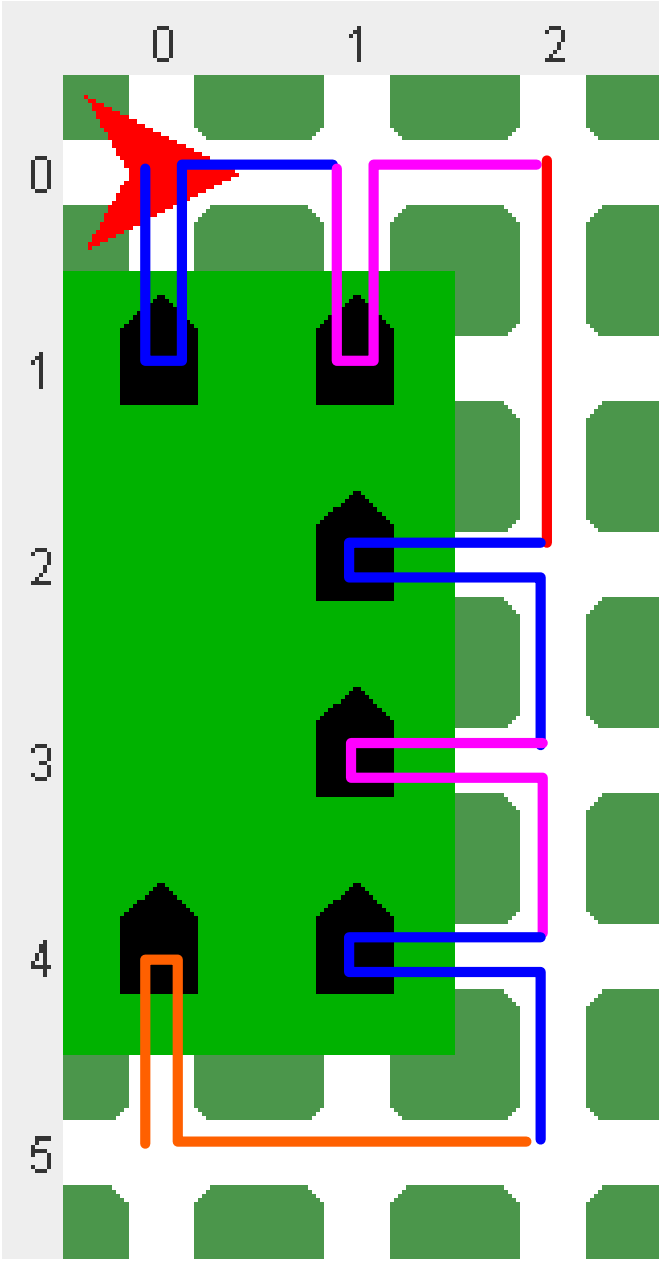
Case Study: Deliver Block

```
import becker.robots.*;
public class DeliveryBot extends RobotSE
{   public DeliveryBot…
    public void deliverFlyers()…
    public void deliverOneAvenue()…

    public void deliverOneSide()
    {   this.deliverBlock();
        this.crossStreet();
        this.deliverBlock();
    }

    public void deliverBlock()
    {  this.deliverHouse();
       this.deliverHouse();
       this.goAroundCorner();
       this.deliverHouse();
       this.deliverHouse();
       this.deliverHouse();
       this.deliverLastHouse();
    }
    public void deliverHouse()…
    public void goAroundCorner()…
    public void deliverLastHouse()…
    public void crossStreet()…
    public void goToOtherSide()…
}
```



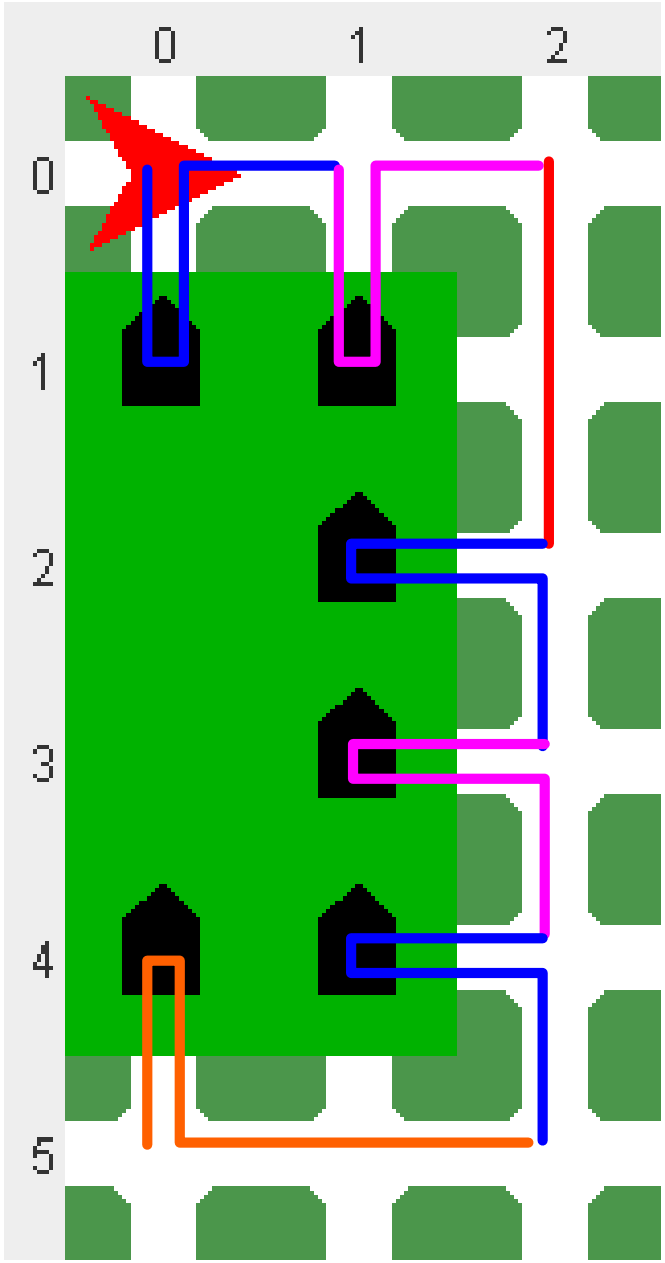
Case Study: Deliver House

```
import becker.robots.*;
public class DeliveryBot extends RobotSE
{   public DeliveryBot…
    public void deliverFlyers()…
    public void deliverOneAvenue()…
    public void deliverOneSide()…

    public void deliverBlock()
    {   this.deliverHouse();
        this.deliverHouse();
        this.goAroundCorner();
        this.deliverHouse();
        this.deliverHouse();
        this.deliverHouse();
        this.deliverLastHouse();
    }

    public void deliverHouse()
    {  this.turnRight();
       this.move();
       this.putThing();
       this.turnAround();
       this.move();
       this.turnRight();
       this.move();
    }
    public void goAroundCorner()…
    public void deliverLastHouse()…
    public void crossStreet()
```



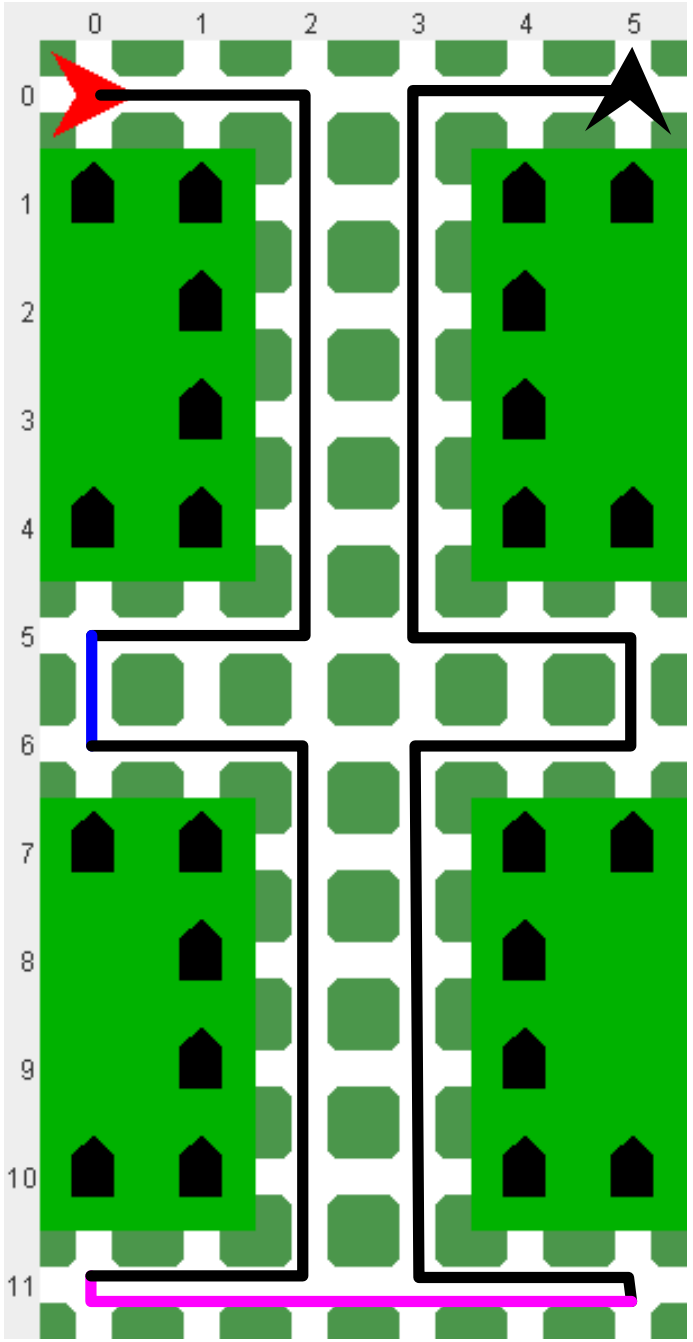
Case Study: Go Around Corner; Deliver Last

```
import becker.robots.*;
public class DeliveryBot extends RobotSE
{   public DeliveryBot…
    public void deliverFlyers()…
    public void deliverOneAvenue()…
    public void deliverOneSide()…
    public void deliverBlock()
    {   this.deliverHouse();       // x2
        this.goAroundCorner();
        this.deliverHouse();       // x3
        this.deliverLastHouse();
    }
    public void deliverHouse()…

    public void goAroundCorner()
    {  this.turnRight();
       this.move();
       this.move();
    }
    public void deliverLastHouse()
    {  this.goAroundCorner();
       this.turnRight();
       this.move();
       this.putThing();
       this.turnAround();
       this.move();
    }
    public void crossStreet()
```



Case Study: Finishing Up

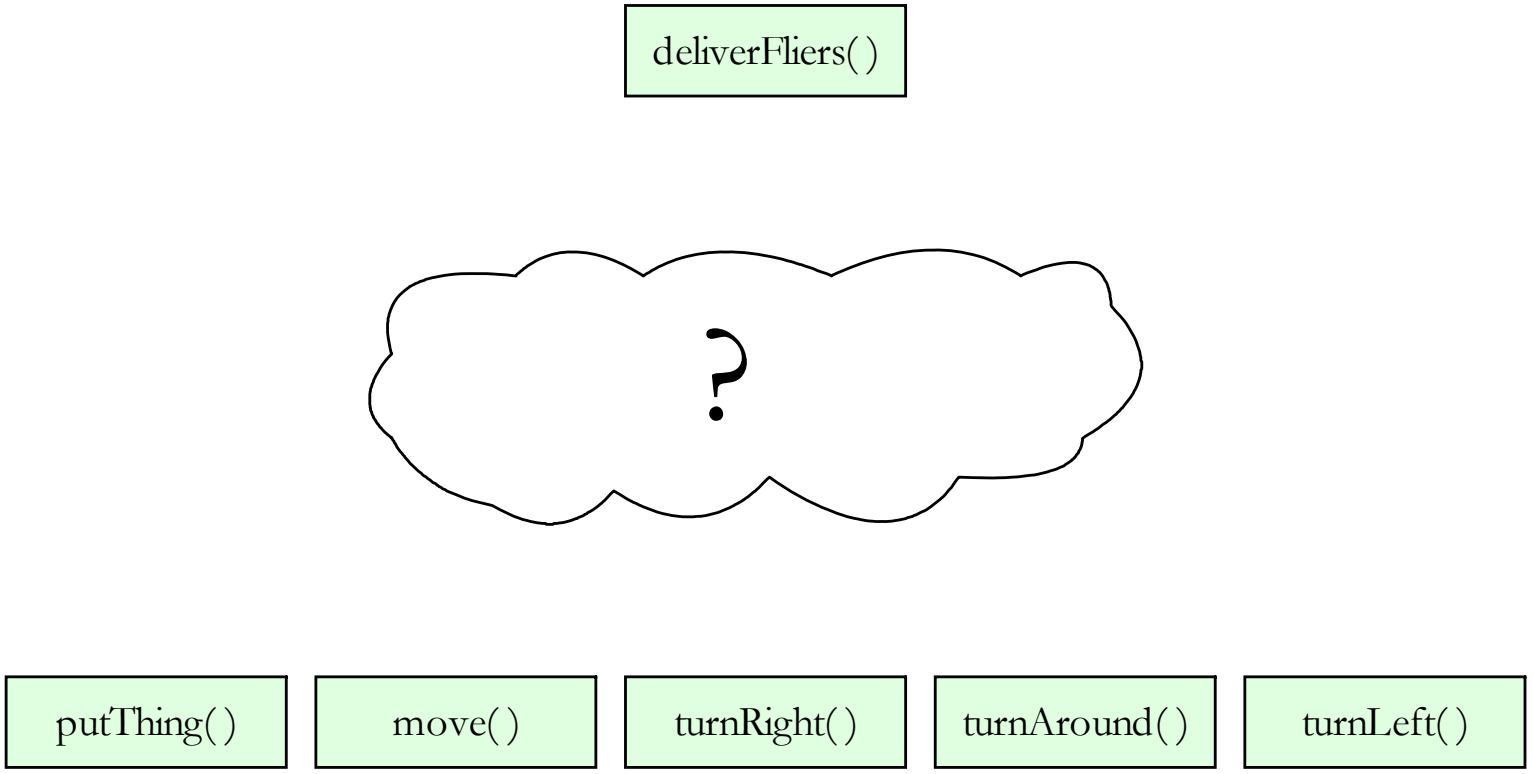
```
import becker.robots.*;
public class DeliveryBot extends RobotSE
{   public DeliveryBot…
    public void deliverFlyers()…
    public void deliverOneAvenue()…
    public void deliverOneSide()…
    public void deliverBlock()…
    public void deliverHouse()…
    public void goAroundCorner()…
    public void deliverLastHouse()…
    /** Cross street and position to deliver next block. */
    public void crossStreet()
    {  this.move();
       this.turnLeft();
    }
    /** Go to the other side of the Avenue.  We're on a side
     * street and need to go to the opposite side street. */
    public void goToOtherSide()
    {  this.turnLeft();
       this.move();
       this.move();
       this.move();
       this.move();
       this.move();
       this.turnAround();
    }
}
```



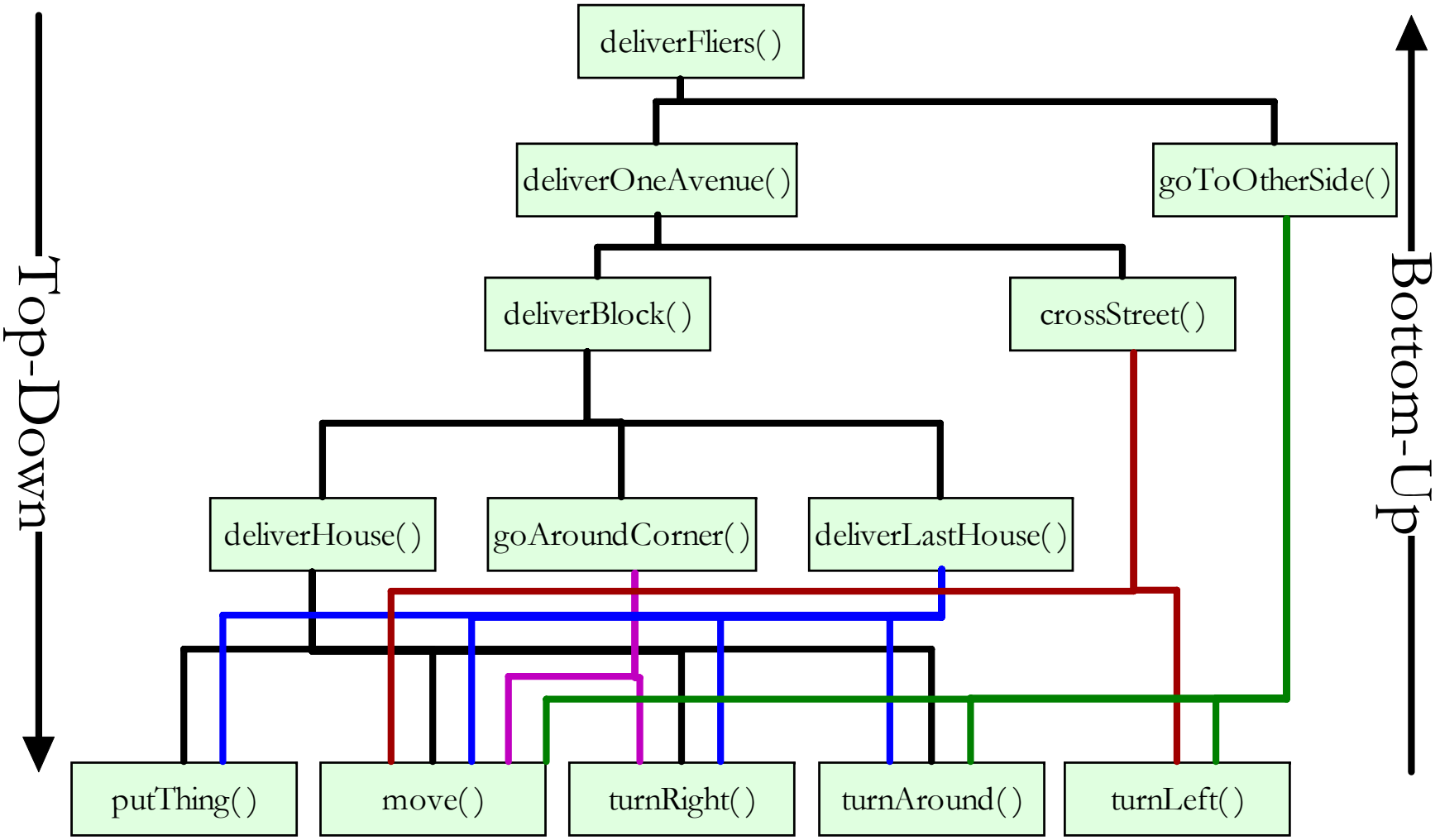
3.2.8: Summary of Stepwise Refinement (1/2)

Stepwise refinement decomposes a complex algorithm (implemented as a method such as **deliverFliers**) into simpler sub-algorithms (implemented as helper methods such as **deliverOneAvenue**).

One view: stepwise refinement is an approach to bridging the gap between the method we need (**deliverFliers**) and the methods we already have (**move**, **turnLeft**, **putThing**, etc.).



3.2.8: Summary of Stepwise Refinement (2/2)



Design: Start at the top and work down

"top-down design" aka "stepwise refinement"

Implementation: Similar (top-down implementation)

Sometimes work bottom-up

3.3: Advantages of Stepwise Refinement

Programs developed using stepwise refinement are more likely to be:

- Easy to understand
- Free of programming errors
- Easy to test and debug
- Easy to modify

Why?

- People can remember only a limited amount of detail
- Stepwise refinement imposes a structure on the problem, keeping related parts together in a method
- Identifying these methods with a descriptive name helps us think at a higher level of abstraction

3.4: Pseudocode

Focus on the algorithm instead of the program implementing it by using *pseudocode*

- Combines naturalness of natural language (such as English) with the structure of a programming language
- Becomes more important when programs make decisions (next lesson!)

Example:

deliver fliers to each house up to the corner

turn the corner

deliver fliers to each house up to the corner

turn the corner

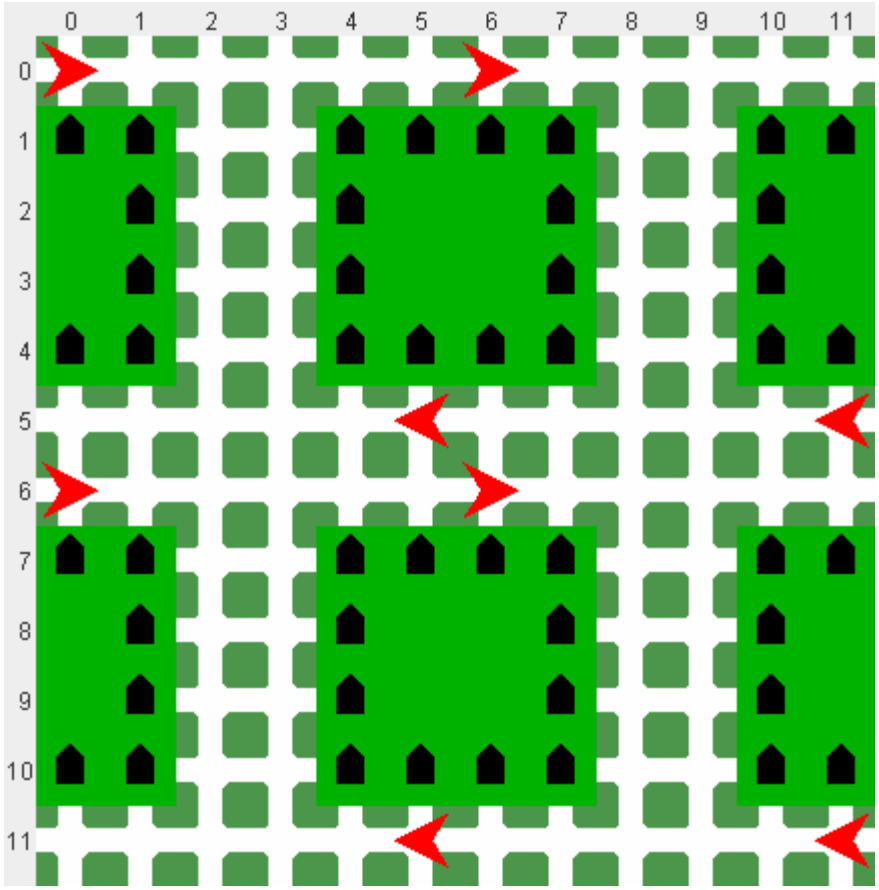
deliver to the last house

3.4: Advantages of Pseudocode

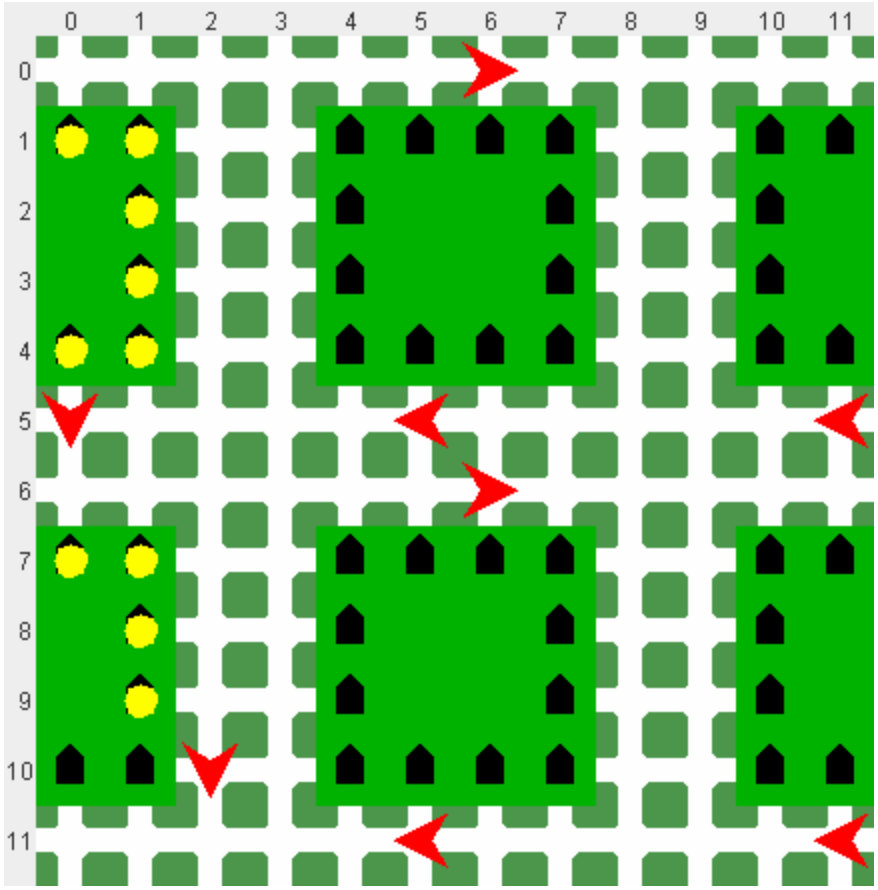
Advantages include:

- Pseudocode helps us think more abstractly, allowing us to ignore many irrelevant details.
- Pseudocode allows us to trace our programs very early in development.
- Pseudocode can provide a common language on a development team, even with non-technical users.
- Algorithms expressed in pseudocode can be implemented in a variety of programming languages.

3.5.1: Using Multiple Robots (1/2)



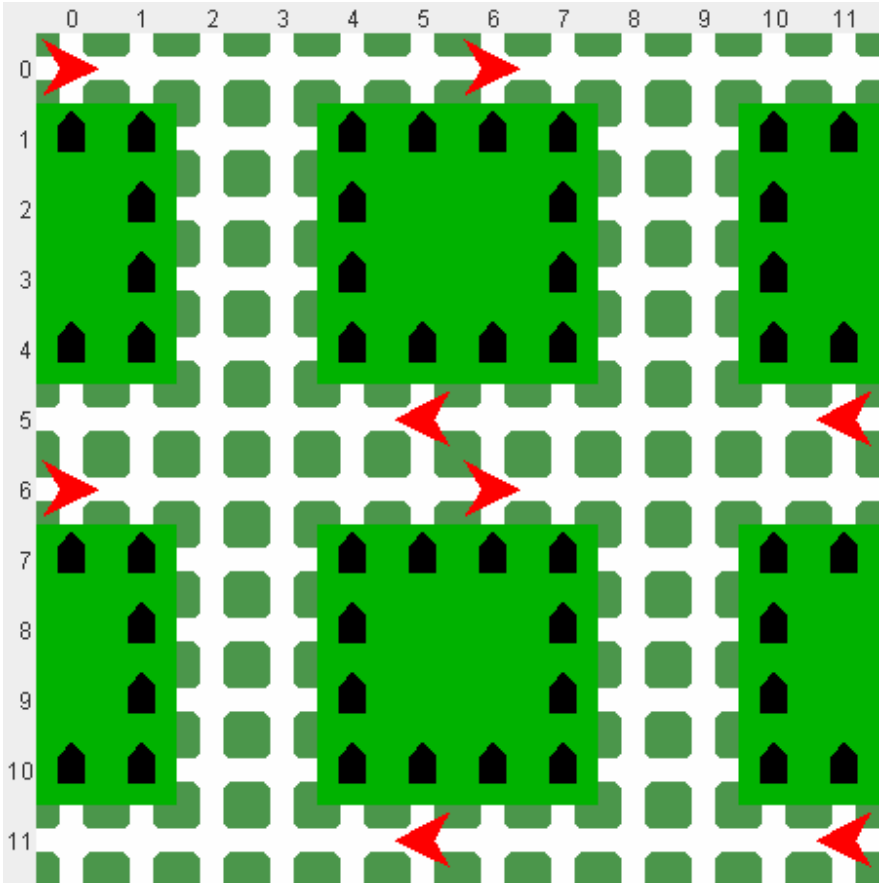
Initial Situation



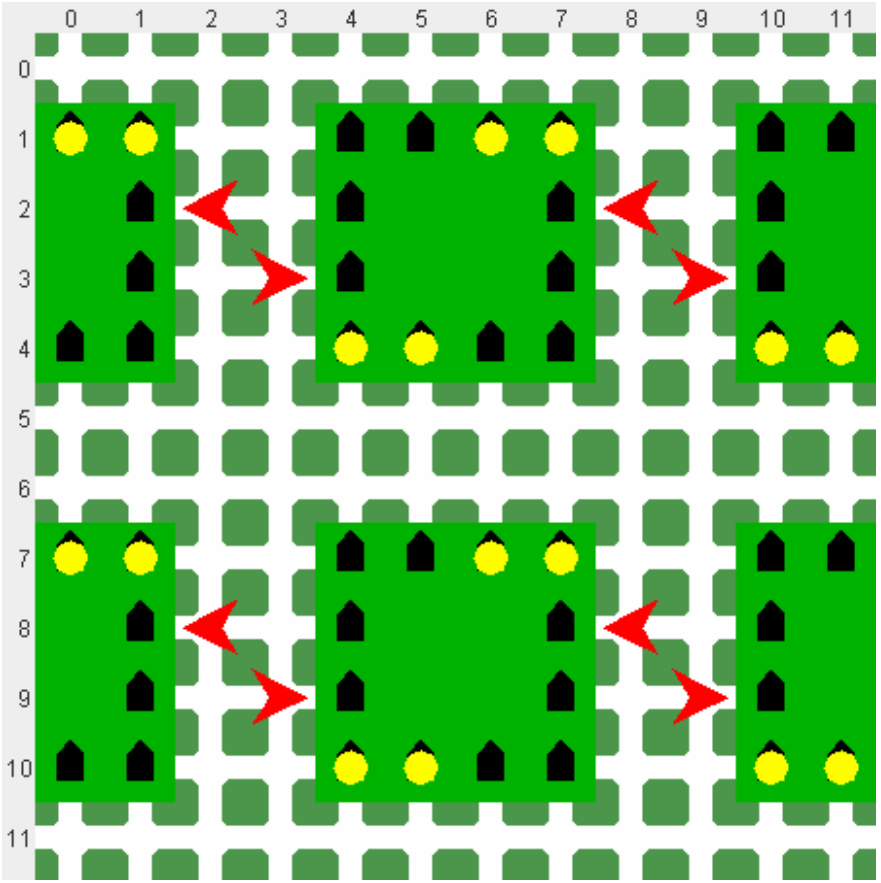
During Execution

3.5.1: Using Multiple Robots (2/2)

```
import becker.robots.*;

public class DeliverFlyers
{  public static void main(String[ ] args)
   {
      Route route = new Route();
      DeliveryBot db1 = new DeliveryBot(route, 0, 0, Direction.EAST, 6);
      DeliveryBot db2 = new DeliveryBot(route, 6, 0, Direction.EAST, 6);
      DeliveryBot db3 = new DeliveryBot(route, 5, 5, Direction.WEST, 6);
      DeliveryBot db4 = new DeliveryBot(route, 11, 5, Direction.WEST, 6);
      DeliveryBot db5 = new DeliveryBot(route, 0, 6, Direction.EAST, 6);
      DeliveryBot db6 = new DeliveryBot(route, 6, 6, Direction.EAST, 6);
      DeliveryBot db7 = new DeliveryBot(route, 5, 11, Direction.WEST, 6);
      DeliveryBot db8 = new DeliveryBot(route, 11, 11, Direction.WEST, 6);

      db1.deliverBlock();
      db2.deliverBlock();
      db3.deliverBlock();
      db4.deliverBlock();
      db5.deliverBlock();
      db6.deliverBlock();
      db7.deliverBlock();
      db8.deliverBlock();
   }
```



Initial Situation



During Execution

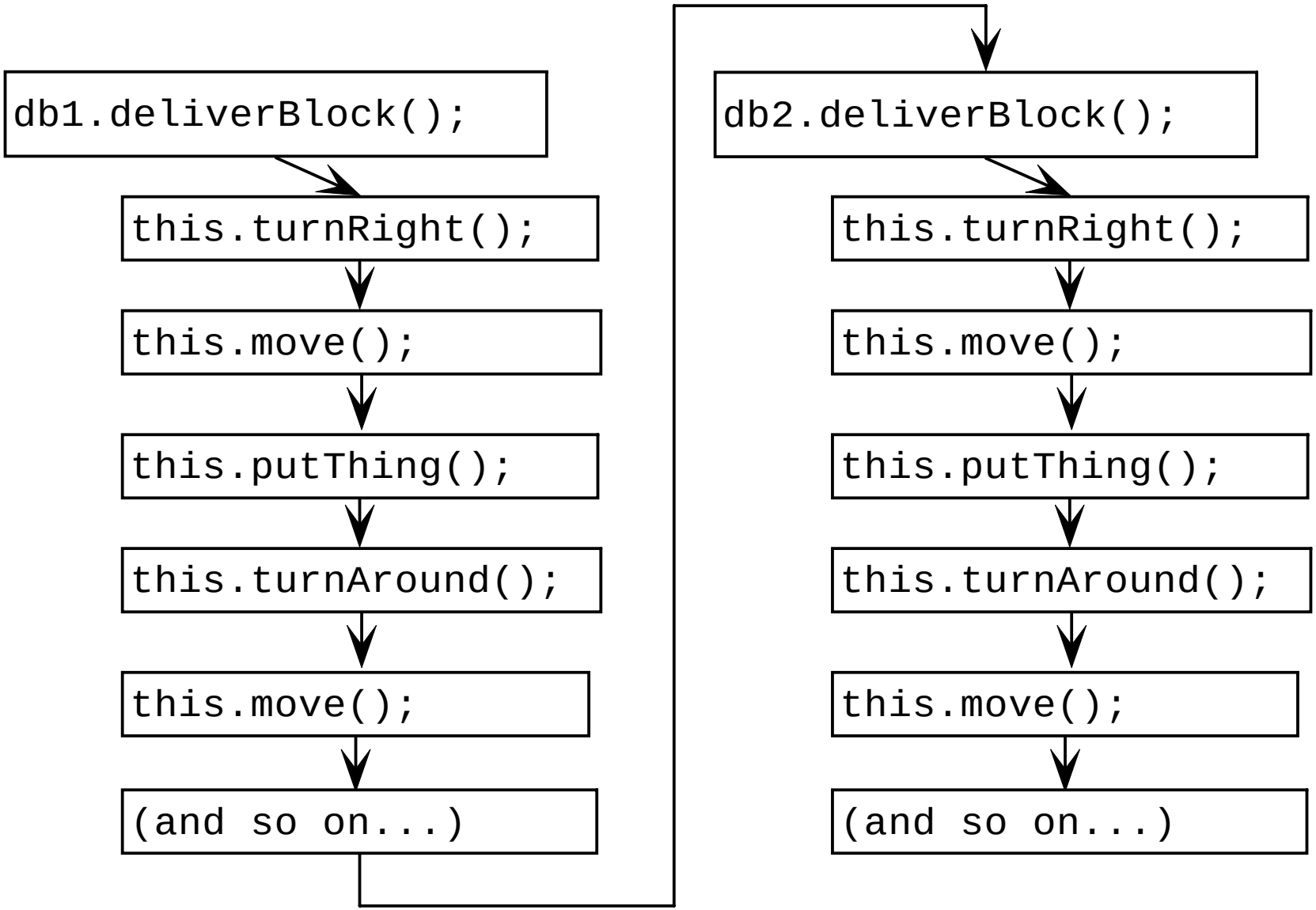
3.5.2: Changes to DeliveryBot

```
import becker.robots.*;

/** A robot to deliver flyers on a prescribed route.
* @author Byron Weber Becker */
public class DeliveryBot extends RobotSE implements Runnable
{
   /** Construct a robot to deliver flyers. */
   public DeliveryBot(City aCity, int aStr, int anAve, Direction aDir, int numFlyers)
   {  super(aCity, aStr, anAve, aDir, numFlyers);
   }

   // The run method contains the code to be executed within the thread.
   public void run()
   {  this.deliverBlock();
   }

   /** Deliver flyers to one block of houses, including the side streets. */
   public void deliverBlock()
   {  this.deliverHouse();
      this.deliverHouse();
      ...
   }
}
```

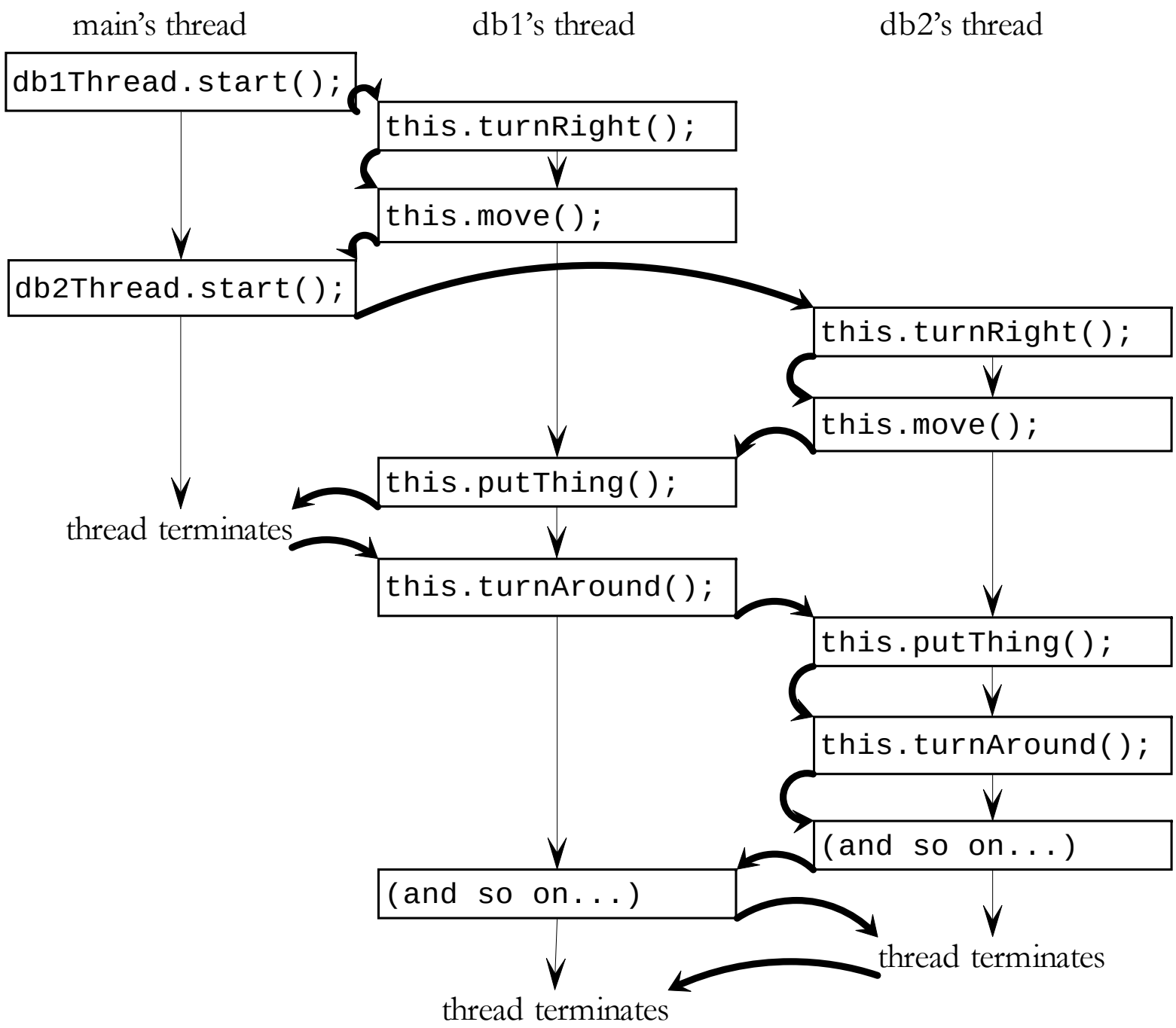
3.5.2: Changes to main

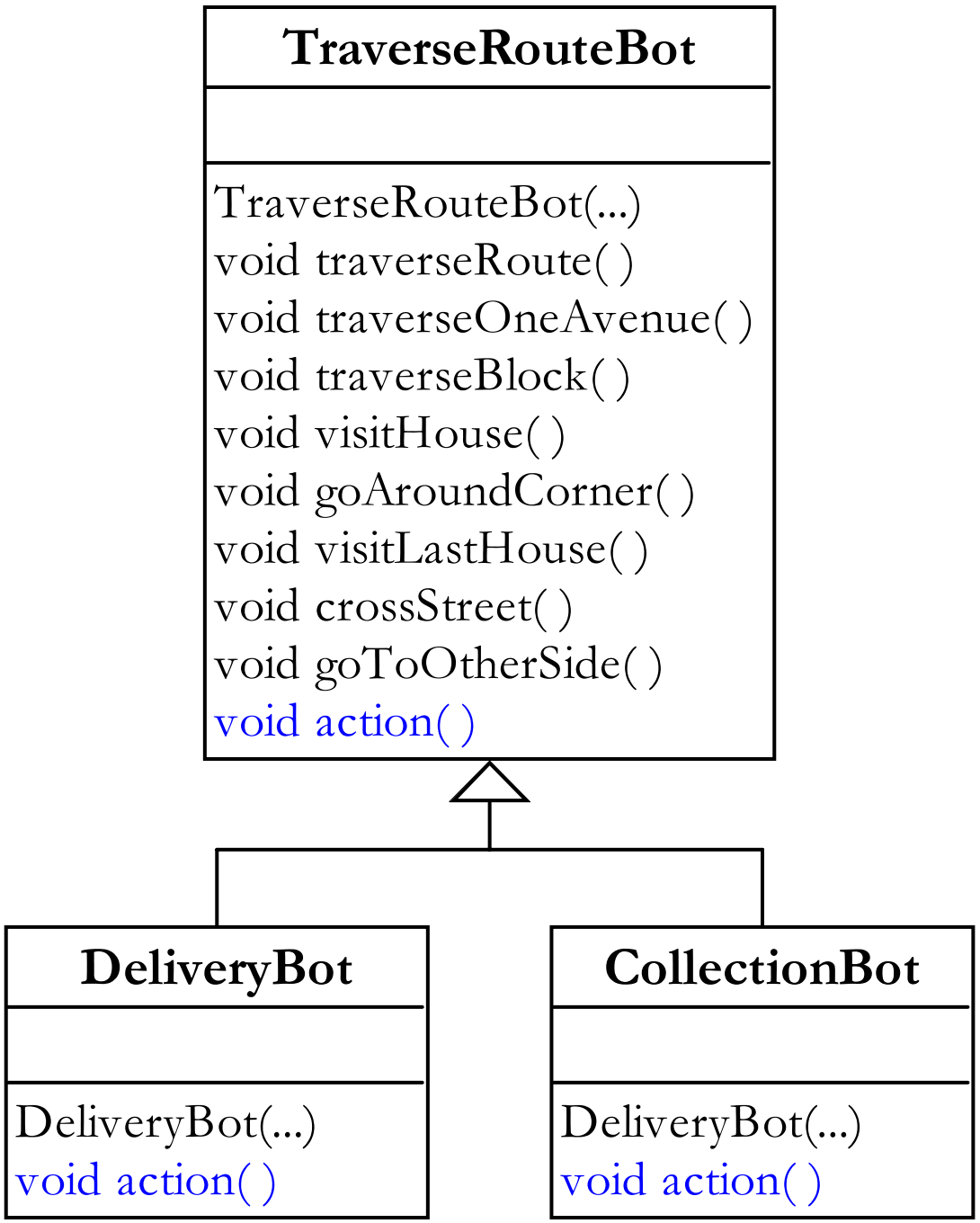
```
import becker.robots.*;

public class DeliverFlyers
{
   public static void main(String[ ] args)
   {  // Same as before
      Route route = new Route();
      DeliveryBot db1 = new DeliveryBot(route, 0, 0, Direction.EAST, 6);
      DeliveryBot db2 = new DeliveryBot(route, 6, 0, Direction.EAST, 6);
      …
      db1.deliverBlock();
      db2.deliverBlock();
      …

      // Set up to run db1 and db2 in parrallel
      Thread db1Thread = new Thread(db1);
      Thread db2Thread = new Thread(db2);
      …

      // Start executing the code in run()
      db1Thread.start();
      db2Thread.start();
      …
   }
}
```

3.5.2: How? (1/2)


db1.deliverBlock();
this.turnRight();
this.move();
this.putThing();
this.turnAround();
this.move();
(and so on...)
db2.deliverBlock();
this.turnRight();
this.move();
this.putThing();
this.turnAround();
this.move();
(and so on...)

3.5.2: How? (2/2)


main's thread
db1's thread
db2's thread
db1Thread.start();
this.turnRight();
this.move();
db2Thread.start();
this.turnRight();
this.move();
this.putThing();
thread terminates
this.turnAround();
this.putThing();
this.turnAround();
(and so on...)
(and so on...)
thread terminates
thread terminates


TraverseRouteBot
TraverseRouteBot(...)
void traverseRoute( )
void traverseOneAvenue( )
void traverseBlock( )
void visitHouse( )
void goAroundCorner( )
void visitLastHouse( )
void crossStreet( )
void goToOtherSide( )
void action( )
DeliveryBot
DeliveryBot(...)
void action( )
CollectionBot
DeliveryBot(...)
void action( )

deliverLastHouse is one of the helper methods in **DeliveryBot**:

```
/** The last house is special because we don't need to move
 * on to the next house. */
public void deliverLastHouse()
{  this.goAroundCorner();
   this.turnRight();
   this.move();
   this.putThing();
   this.turnAround();
   this.move();
}
```

Should clients be able to call it? For example:

```
public static void main(String[ ] args)
{  City route = new City();
   …
   DeliveryBot karel = new DeliveryBot(…);

   karel.deliverLastHouse();
}
```

public methods:

- **public void deliverFliers()**
- May be called from any method (eg: **main**), including other methods within the class and subclasses.
- Should be used for methods explicitly designed as one of the classes' services.

protected methods:

- **protected void deliverOneSide()**
- May be called from any method in the same class or a subclass.
- Often used for helper methods that might be overridden in a subclass.

private methods:

- **private void deliverLastHouse()**
- May only be called from methods within the same class.
- The usual case, unless there is a reason for **public** or **protected**.

3.8.1: The Helper Method Pattern

Name: Helper Method

Context: You have a long or complex method and want your code to be easy to develop, test, and modify.

Solution:

Look for a logical decomposition, putting each part into a helper method. Use a pattern such as *Parameterless Command* to implement the helper method.

For example:

```
public void deliverFliers()
{  this.deliverOneAvenue();      // call a helper method
   this.turnRight();
   this.move();
   this.deliverOneAvenue();      // call a helper method
}
```

Consequences: Methods are easier to develop, understand, modify.

Related Patterns: Almost identical to *Parameterless Command* and patterns to appear in later chapters. The difference is in the context and motivation.

3.8.2: The Multiple Threads Pattern

Name: Multiple Threads

Context: Multiple objects need to carry out tasks "simultaneously."

Solution: Start each task in its own thread of control.

```
public class «className» extends «superclassName»
                                    implements Runnable
{ …
  public void run()
  { «statements to execute inside a separate thread»
  }
}

…
«className» «runnableObject» = «className»new (…);
Thread «threadName» = new Thread(«runnableObject»);
«threadName».start();
```

Consequences: Execution of two or more threads can be interleaved. If the threads can interfere with each other, many problems result.

Related Patterns: Java Program, Extended Class, Object Instantiation, Method Invocation, etc.

3.8.3: The Template Method Pattern

Name: Template Method

Context: Several tasks are very similar, resulting in duplicate code.

Solution: Factor out the duplicate code into a common superclass. Provide methods to override to encode the differences between the tasks.

Consequences: Writing common code once helps reduce the effort required to write, debug, and maintain the code.

Spreading the code over two or more classes makes it more difficult to understand.

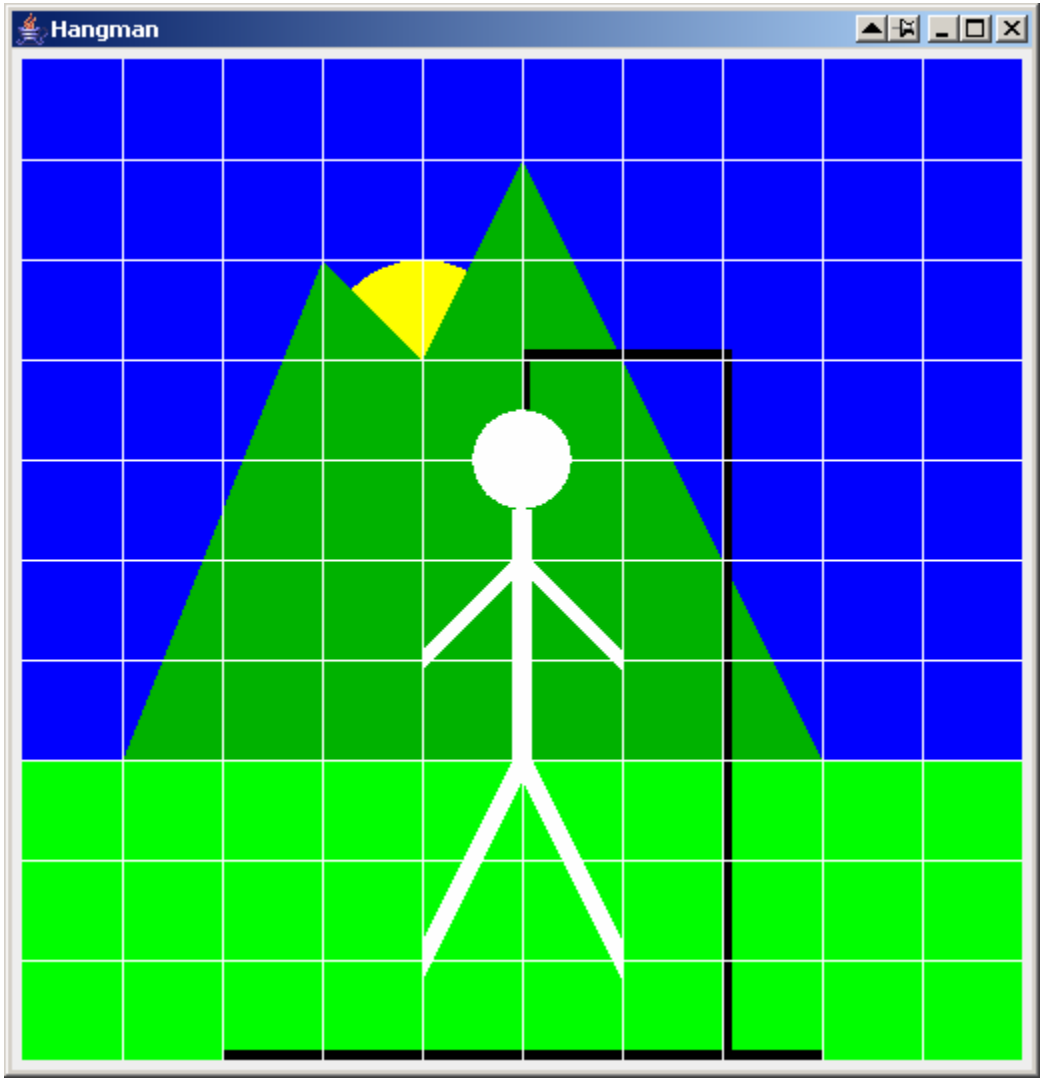
Related Patterns: This pattern is a specialization of the Extended Class pattern.

Application: Drawing for Hangman

The game of Hangman uses a drawing with a person hanging from a gallows as a way to keep track of a person's progress in guessing a word or phrase.

Extend **JComponent** to create a new kind of component that draws this scene. Override **paintComponent** to do the actual drawing. Use stepwise refinement to make your code easier to understand, write, and debug.

A 10x10 grid is shown here to aid the drawing process. It should not appear in the final product. Make the entire drawing 500x500 pixels.



Application: The main method

```
import javax.swing.*;

/** Display an image of a person hanging from a gallows, as for the game of Hangman.
*
* @author Byron Weber Becker */
public class Hangman
{
   public static void main(String[ ] args)
   {  JFrame f = new JFrame();
      JPanel contents = new JPanel();
      GallowsView view = new GallowsView();

      contents.add(view);

      f.setContentPane(contents);
      f.setTitle("Hangman");
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      f.pack();
      f.setVisible(true);
   }
}
```

Application: Extending JComponent

```
import javax.swing.*;
import java.awt.*;

/** Draw the gallows for Hangman..
    @author Byron Weber Becker */
public class GallowsView extends JComponent
{
   /** Construct the specialized component. */
   public GallowsView()
   {  super();
      this.setPreferredSize(new Dimension(500,500));
   }

   /** Paint the component.  This is called automatically by the system.
   *   @param g The graphics context for painting. */
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);
   }
}
```

Application: Decomposing paintComponent

```
import javax.swing.*;
import java.awt.*;

public class GallowsView extends JComponent
{
   public GallowsView()…

   public void paintComponent(Graphics g)
   {  super.paintComponent(g);
      this.drawBackground(g);
      this.drawGallows(g);
      this.drawPerson(g);
   }

   private void drawBackground(Graphics g)
   {
   }
   private void drawGallows(Graphics g)
   {
   }
   private void drawPerson(Graphics g)
   {
   }
}
```

Application: Implementing drawBackground

```
import javax.swing.*;
import java.awt.*;
public class GallowsView extends JComponent
{
   public GallowsView()…                              // done
   public void paintComponent(Graphics g)…            // done

   /** Draw the background with sky, mountains, sun, etc.
   @param g The graphics context. */
   private void drawBackground(Graphics g)
   {  g.setColor(Color.BLUE);              // sky
      g.fillRect(0, 0, 500, 350);

      g.setColor(Color.YELLOW);            // sun
      g.fillOval(150, 100, 100, 100);

      g.setColor(Color.GREEN);             // foreground grass
      g.fillRect(0, 350, 500, 250);

      this.drawMountain(g);
   }
   private void drawMountain(Graphics g)
   {
   }
   private void drawGallows()…
}
```

Application: Implementing drawMountain

```
import javax.swing.*;
import java.awt.*;
public class GallowsView extends JComponent
{
   public GallowsView()…                                  // done
   public void paintComponent(Graphics g)…        // done
   private void drawBackground(Graphics g)…     // done

   private void drawMountain(Graphics g)
   {  g.setColor(Color.GREEN.darker());
      Polygon m = new Polygon();
      m.addPoint(50, 350);
      m.addPoint(150, 100);
      m.addPoint(200, 150);
      m.addPoint(250, 50);
      m.addPoint(400, 350);
      m.addPoint(50, 350);
      g.fillPolygon(m);
   }

   private void drawGallows(Graphics g)…
   private void drawPerson(Graphics g)…
}
```

Application: Implementing drawGallows

```
import javax.swing.*;
import java.awt.*;
public class GallowsView extends JComponent
{
   public GallowsView()…                              // done
   public void paintComponent(Graphics g)…            // done
   private void drawBackground(Graphics g)…           // done
   private void drawMountain(Graphics g)…             // done

   private void drawGallows(Graphics g)
   {  g.setColor(Color.BLACK);

      g.fillRect(100, 495, 300, 5);       // base
      g.fillRect(350, 150, 5, 350);       // upright
      g.fillRect(250, 145, 105, 5);       // top
      g.fillRect(250, 150, 4, 25);        // rope
   }

   private void drawPerson(Graphics g)
   {
   }
}
```

Application: Decomposing drawPerson

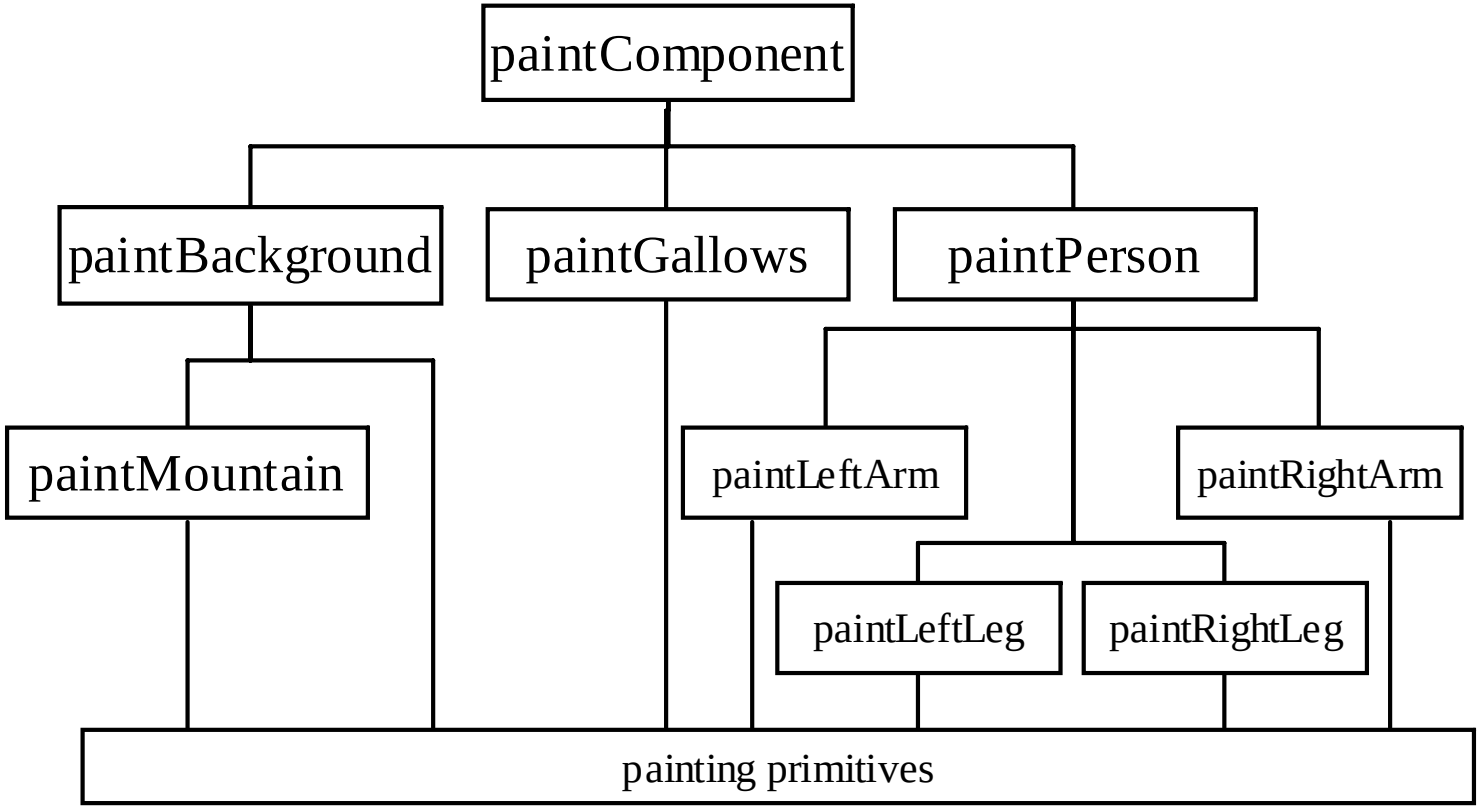
```
import javax.swing.*;
import java.awt.*;
public class GallowsView extends JComponent
{
   public GallowsView()…                          // done
   public void paintComponent(Graphics g)…        // done
   private void drawBackground(Graphics g)…       // done
   private void drawMountain(Graphics g)…         // done
   private void drawGallows(Graphics g)…          // done

   private void drawPerson(Graphics g)
   {  g.setColor(Color.WHITE);

      g.fillOval(225, 175, 50, 50);      // draw head
      g.fillRect(245, 225, 10, 125);     // draw body

      this.drawRightArm(g);
      this.drawLeftArm(g);
      this.drawRightLeg(g);
      this.drawLeftLeg(g);
   }
   private void drawRightArm(Graphics g)…
   private void drawLeftArm(Graphics g)…
   private void drawRightLeg(Graphics g)…
   private void drawLeftLeg(Graphics g)…
}
```

Application: Implementing drawRightArm

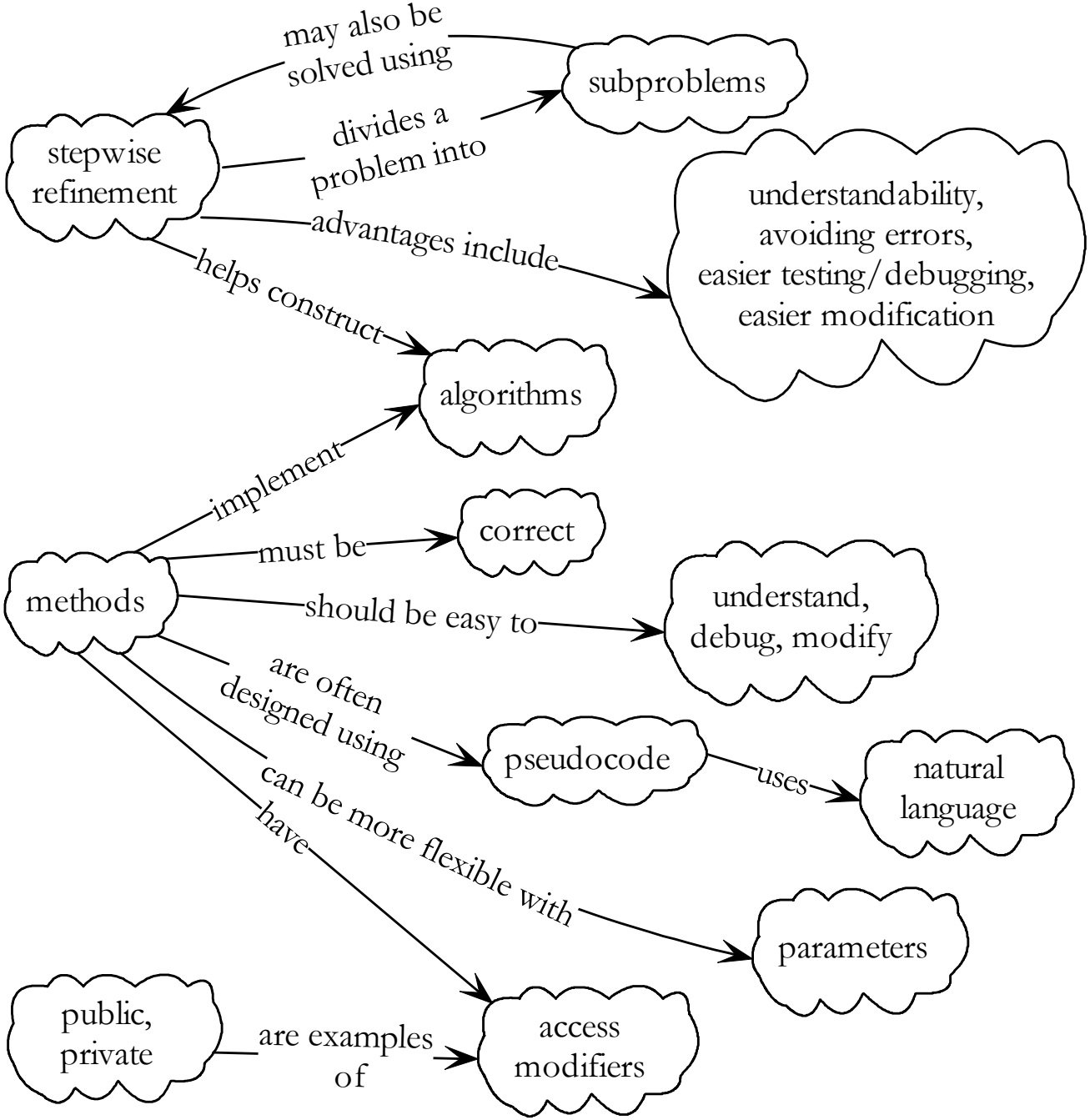
```
import javax.swing.*;
import java.awt.*;
public class GallowsView extends JComponent
{
   public GallowsView()…                          // done
   public void paintComponent(Graphics g)…        // done
   private void drawBackground(Graphics g)…       // done
   private void drawMountain(Graphics g)…         // done
   private void drawGallows(Graphics g)…          // done
   private void drawPerson(Graphics g)…           // done

   private void drawRightArm(Graphics g)
   {  Polygon arm = new Polygon();
      arm.addPoint(250, 245);
      arm.addPoint(300, 295);
      arm.addPoint(300, 305);
      arm.addPoint(250, 255);
      arm.addPoint(250, 245);
      g.fillPolygon(arm);
   }

   private void drawLeftArm(Graphics g)…
   private void drawRightLeg(Graphics g)…
   private void drawLeftLeg(Graphics g)…
}
```

Application: Decomposition

We decomposed a difficult problem (painting a hangman scene) into a series of simpler problems. When the solutions of the small problems are combined appropriately, we solve the difficult problem.



3.9: Concept Map



Summary

We have learned:

- how to decompose a complex problem into simpler problems using *stepwise refinement*.
- that the solution to each simpler problem should be encoded in a helper method.
- that stepwise refinement leads to programs that are more likely to be easy to understand, free from errors, easy to test and debug, and easy to modify.
- that pseudocode is a mixture of a programming language and natural language and allows us to think about our solutions at a higher level of abstraction, and find and fix bugs earlier.
- that there are often several solutions to a problem, perhaps involving different resources (e.g. additional robots), doing parts of the task simultaneously using threads, and factoring common parts of solutions into a superclass.