Well-behaved Objects

• Most of the chapter focuses on testing
• Our first project will involve a diary
  – We will study the code and then look at some test programs
  – We will see how testing can be automated
• We then look at unit testing, a feature not available in BlueJ when this first edition of the text was published
• Lab 14 will involve finding errors in the bricks project
• The next project will be a calculator; we will learn about Graphical User Interfaces (GUI) and study the code for a simple add/subtract calculator in Java
• Lab 15 will involve implementing a very simple binary calculator where the only digits are 0 and 1
The Diary Project

- We first run this project, concentrating on the classes Day and Appointment.

- We will write test methods to assure the day’s appoints are handled correctly.
/** Maintain the appointments for one day in a diary.  
* @author David J. Barnes and Michael Kolling  
* @version 2001.09.14  
*/
public class Day {

    // The first and final bookable hours in a day.
    public static final int START_OF_DAY = 9;
    public static final int FINAL_APPOINTMENT_TIME = 17;
    // The number of bookable hours in a day.
    public static final int MAX_APPOINTMENTS_PER_DAY =
        FINAL_APPOINTMENT_TIME -
        START_OF_DAY + 1;

    // A day number within a particular year. (1-366)
    private int dayNumber;
    // The current list of appointments for this day.
    private final Appointment[] appointments;

    /**
     * Constructor for objects of class Day.
     * @param dayNumber The number of this day in the year (1-366).
     */
    public Day(int dayNumber) {
        this.dayNumber = dayNumber;
        appointments = new Appointment[MAX_APPOINTMENTS_PER_DAY];
    }

    // methods go here
}
/**
 * Try to find space for an appointment.
 * @param appointment The appointment to be accommodated.
 * @return The earliest time today that can accommodate the appointment.
 *         Return -1 if there is insufficient space.
 */

public int findSpace(Appointment appointment)
{  int duration = appointment.getDuration();
   for(int slot = 0; slot < MAX_APPOINTMENTS_PER_DAY; slot++) {
      if(appointments[slot] == null) {
         final int time = START_OF_DAY + slot;
         // Potential start point.
         if(duration == 1) {
            // Only a single slot needed.
            return time;
         }
      }
      else {
         // How many more slots are needed?
         int further_slots_required = duration - 1;
         for(int nextSlot = slot + 1;
                  further_slots_required > 0 &&
                  appointments[nextSlot] == null;
                  nextSlot++) {
            further_slots_required--;
         }
         if(further_slots_required == 0) {
            // A big enough space has been found.
            return time;
         }
      }
   }
   return -1; // Not enough space available.
}
/**
 * Make an appointment.
 * @param time The hour at which the appointment starts.
 * @param appointment The appointment to be made.
 * @return true if the appointment was successful,
 *         false otherwise.
 */

public boolean makeAppointment(int time,
                                 Appointment appointment)
{
    if(time >= START_OF_DAY && time <= FINAL_APPOINTMENT_TIME) {
        int startTime = time-START_OF_DAY;
        if(appointments[startTime] == null) {
            int duration = appointment.getDuration();
            // Fill in all the slots for the full duration
            // of the appointment.
            for(int i = 0; i < duration; i++) {
                appointments[startTime + i] = appointment;
            }
            return true;
        }
        else {
            return false;
        }
    }
    else {
        return false;
    }
}
/**
 * Print a list of the day's appointments on standard output.
 */

public void showAppointments()
{
    System.out.println("=== Day " + dayNumber + " ===");
    for(int i = 0; i < MAX_APPOINTMENTS_PER_DAY; i++) {
        System.out.print((START_OF_DAY + i) + ":");
        if(appointments[i] != null) {
            System.out.println(appointments[i].getDescription());
        }
        else {
            System.out.println();
        }
    }
}

/**
 * @return The number of this day within the year (1 - 366).
 */

public int getDayNumber()
{
    return dayNumber;
}
public class Appointment {
    // The reason for the appointment.
    private String description;
    // The length (in hours) of the appointment.
    private int duration;
    /**
     * Constructor for objects of class Booking.
     * @param description The reason for the appointment.
     * @param duration The length of the appointment in hours.
     */
    public Appointment(String description, int duration) {
        this.description = description;
        this.duration = duration;
    }
    /**
     * @return The description of the appointment.
     */
    public String getDescription() {
        return description;
    }
    /**
     * @return The duration (in hours) of the appointment.
     */
    public int getDuration() {
        return duration;
    }
}
Our First Three Tests

• We concentrate on appointments that are one hour long
• We first make three appointments for different times of the day
• Then we make a double booking, that is, try to schedule an appointment for a time already taken
• Finally, we fill the day with appointments and see what happens
Make Three Appointments

/**
 * Test basic functionality by booking at either end
 * of a day, and in the middle.
 */

public void makeThreeAppointments()
{
    // Start with a fresh Day object.
    day = new Day(1);
    // Create three one-hour appointments.
    Appointment first = new Appointment("Java lecture", 1);
    Appointment second = new Appointment("Java class", 1);
    Appointment third = new Appointment("Meet John", 1);

    // Make each appointment at a different time.
    day.makeAppointment(9, first);
    day.makeAppointment(13, second);
    day.makeAppointment(17, third);

    day.showAppointments();
}
/**
 * Check that double-booking is not permitted.
 */

public void testDoubleBooking()
{
    // Set up the day with three legitimate appointments.
    makeThreeAppointments();
    Appointment badAppointment = new Appointment("Error", 1);
    day.makeAppointment(9, badAppointment);

    // Show that the badAppointment has not been made.
    day.showAppointments();
}
/**
 * Test basic functionality by filling a complete day with appointments.
 */

public void fillTheDay()
{
    // Start with a fresh Day object.
    day = new Day(1);
    for(int time = Day.START_OF_DAY;
        time <= Day.FINAL_APPOINTMENT_TIME;
        time++) {
        day.makeAppointment(time,
            new Appointment("Test " + time, 1));
    }

    day.showAppointments();
}
Automating the Testing

• The basic strategy is to have each test method return a boolean value to indicate success or failure

• Sometimes a “success” is that an operation failed
  – Double booking is an example
  – If an appointment has already been made for an hour, then an attempt to make another appointment for the same hour should not succeed
  – In this case a “success” is that the second booking failed and a “failure” would be that a second booking succeeded
public boolean makeThreeAppointments()
{
    // Start with a fresh Day object.
    day = new Day(1);
    // Create three one-hour appointments.
    Appointment first = new Appointment("Java lecture", 1);
    Appointment second = new Appointment("Java class", 1);
    Appointment third = new Appointment("Meet John", 1);

    // Assume that the test is passed unless we discover otherwise.
    boolean passed = true;
    // Make each appointment at a different time.
    passed &= day.makeAppointment(9, first);
    passed &= day.makeAppointment(13, second);
    passed &= day.makeAppointment(17, third);

    // Check that each appointment was made at the correct time.
    passed &= day.getAppointment(9) == first;
    passed &= day.getAppointment(13) == second;
    passed &= day.getAppointment(17) == third;

    return passed;
}
public boolean testDoubleBooking()
{
    // Assume that the test is passed unless we discover
    // otherwise.
    boolean passed = true;
    // Set up the day with three legitimate appointments.
    passed = makeThreeAppointments();
    Appointment badAppointment = new Appointment("Error", 1);

    // The time we will try to double book at.
    int time = 9;
    // Take a copy of the appointment at that time,
    // to ensure that it is not changed by the next test.
    Appointment first = day.getAppointment(time);
    // Note that we want this attempt to fail.
    passed &= !day.makeAppointment(time, badAppointment);
    // Check that first is still where it should be.
    passed &= day.getAppointment(time) == first;
    return passed;
}
public boolean fillTheDay() {
    // Start with a fresh Day object.
    day = new Day(1);
    // Ensure that all appointments can be made.
    boolean passed = true;
    for (int time = Day.START_OF_DAY;
         time <= Day.FINAL_APPOINTMENT_TIME;
         time++) {
        passed &= day.makeAppointment(time,
                                       new Appointment("Test " + time, 1));
    }
    // Check that all appointment times are now occupied.
    for (int time = Day.START_OF_DAY;
         time <= Day.FINAL_APPOINTMENT_TIME;
         time++) {
        passed &= day.getAppointment(time) != null;
    }
    return passed;
}
public boolean runAllTests()
{
    boolean allPassed = true;
    System.out.println("runAllTests started ...");
    if(!makeThreeAppointments()) {
        System.out.println("makeThreeAppointments() failed.");
        allPassed = false;
    }
    if(!testDoubleBooking()) {
        System.out.println("testDoubleBooking() failed.");
        allPassed = false;
    }
    if(!fillTheDay()) {
        System.out.println("fillTheDay() failed.");
        allPassed = false;
    }
    System.out.println("runAllTests finished");
    return allPassed;
}
Other Testing Methodologies

- Develop unit tests
- The Manual Walkthrough
  - High-level walkthrough the source code
  - Checking the state during a detailed walkthrough
  - Verbal walkthroughs – explain your code to someone else
- Insert print statements to print out intermediate values
- Rig the code for debugging using boolean variables to control extra print statements
- Use a debugger, if available
Lab 14

• The bricks project involves two classes
  – A Brick is an individual brick with an associate height, width, and depth (measured in cm) and weight (calculated based on the volume and a density of 2 grams per cubic centimeter)
  – A Palette consists of a number of bricks in a plane (that is, a single level) and the number of planes stacked on top of each other

• This implementation contains four serious errors
  – Your job is to discover those errors
  – Do this by writing some unit tests that reveal the errors