Structuring Java Programs

— Data Directed Design —

COMP1100 — Introduction to Programming and Algorithms

Clem Baker-Finch

Australian National University

Semester 1, 2006

Announcements

- If you're still having trouble installing Java on your Windows machine, I have put some more detailed instructions on the course web site, linked from the *Java* page.
- We need a 1st year representative for the *Department of Computer Science Forum.*
  - Solicit views of students (e.g. on 1100.talk and 1200.talk phorums, then . . .
  - Offer feedback on DCS educational issues (teaching infrastructure, policies etc.)
  - Next meeting: 4–5pm, 31st May, N101 (One per semester)
  - Contact me or Peter Strazdins `Peter.Strazdins@cs.anu.edu.au`

The Supermarket Docket again . . . but in Java

In an attempt to bring all these Java threads together, today we will go through the development of the **supermarket docket** program we wrote in Haskell, weeks ago.

Reminder: The checkout scanner produces a sequence of bar codes. Our program is to look up each of those codes in a database to retrieve the information about the product, then generate an itemised docket including the total price.

For example, if the scanner produces a sequence of bar codes like:

```
4719 1112 1113 3814 1234
```

our program will print something like:

```
     Gosling's Sunny-Mart

Frozen Pizza...............6.49
Mars Bar...................1.60
Unknown Item...............0.00
Hokkien Noodles............2.05
Chianti, 1lt..............17.95

Total....................$28.09
```

Summary of Data Directed Design Process

Here we go again. . .

1. **Understand the Problem**
2. **Identify the Abstract Data Types**
3. **Identify the Basic Operations on the ADTs**
4. **Choose Representations of the ADTs**
5. **Implement the Basic Operations on the ADTs**
6. **Factor the Process into Manageable Parts**

2. Identify the ADTs

- Products
- Bar codes
- Prices
- Names (of products)
- Database
- Docket
- Sequence of bar codes from scanner

3. Identify the *Basic Operations* on the ADTs

Products:

- What is its bar code? Name? Price?
- Construct a new product record.

Bar codes

- Equality comparison.
- Anything else?

Prices

- Arithmetic.

Names of products

- Print or display them.

Database

- Retrieve a product, given a bar code.
- Add a new product to the database.
- Delete a product, etcetera, etcetera!!

Docket

- Print or display.

Bar code sequences

- Traverse.

4. Choose *Representations* of the ADTs

Let's do the easy ones first:

- **Dockets** and **Names** can just be *strings*.
- **Prices** can just be *integers*.
- **Bar codes?**

  Some context information: the *UPC* (Universal Product Code) is a sequence of 12 digits. It is *not* a number, so representing them as *integers* is not a good idea, especially on a computer.

  But this is just a demonstration exercise, so we will ignore reality and represent bar codes as *integers*.

Products:

Considering the operations we have identified, a simple class with fields for the product's bar code, name and price is an obvious choice.

Database:

There are lots of choices, and lots of advantages and disadvantages of different representations. But we don't know about that yet. . .

Choose something we *do* know about. We know how to search through a list, so we will represent the database as a list of products.

(In next week's lab, you are invited to use a *hash table* instead.)

So, we need to implement two Java classes:

`Product.java`

`DB.java`

Sequence of bar codes:

Since all we want to do is *traverse* the sequence, a list of bar codes suffices.

An interesting question is whether we really need to **physically represent** these sequences — that is do they need to be **stored** in the program?

From one viewpoint, the answer is no — the scanner gives us one bar code of the sequence at a time and we process them as they arrive. The sequence is purely transitory.

However, for the purpose of *testing* (during development of the program) it may be convenient to set up fixed dummy lists of bar codes, rather than have to worry about generating them dynamically.

That's what we'll do today.

5. *Implement* the Basic Operations on the ADTs

We have identified the operations and the representations for each ADT.

This part is just writing code. . .

6. Factor the *Process* into Manageable Parts

For each bar code:

1. Look it up in the database to retrieve the product information;
2. Format (and print?) a line of the docket
3. Add the product price to the total

After processing all of the bar codes, format and print the *total cost* line of the docket.

We also want to print a *header* for the docket first.

The **sequence of actions** making up the process will appear in the **driver** class. Since we are only going as far as *testing*, we'll call it `Test.java`.

Since it's a test program, we'll construct a small sample database, and a dummy list of bar codes representing the sequence of scanner inputs.

The repeated actions for each bar code will be in a loop traversing the list of bar codes.

But first...

A *Docket* Module

Notice that in the process sequence we had three activities related to **formatting** stuff for the docket:

- Format each line representing a purchase
- Format the total cost line
- Format a header for the docket

It makes sense (to me at least) to collect these operations together, but they aren't really operations on an ADT — we want a *module*, but all we have in Java is classes.

In yesterday's lecture we saw how the `Math` library class is this kind of module — a class where all methods are **static**, so they are related to the class, and we don't construct objects of such a class.

We can build a module `Docket.java` with all methods declared `static`.

It also has some *helper* methods for internal use, so they are declared `private`.

(As is often the case with formatting stuff, it gets a bit messy and tedious, so don't worry too much about the code. . .)

A Haiku . . .

Nice little program.

We thought about its structure.

I feel satisfied.