

Data Directed Design

Reading: Thompson Ch.6

COMP1100 — Introduction to Programming and Algorithms

Clem Baker-Finch

Australian National University

Semester 1, 2006

Announcements

- Mid-semester exam:
 - Melville Hall 6 to 7pm Monday 3 April 2006
 - Open-book (except ANU library books)
- Drop-in labs in N116:
 - Week 5:
 - * Thursday 10 – 12am
 - Week 6:
 - * Monday 10 – 12am
 - * Thursday 10 – 12am
 - Week 7:
 - * Monday 10 – 12am

Program Design

In earlier lectures we looked at how to think about designing functions.

The key idea was **abstraction** — breaking the problem into smaller parts to reduce the complexity of components we need to manage at one time.

Now that we know a little about **modules** we should look again at *how* we might go about analysing the problem to identify the **manageable parts**.

The approach we will take is called ***Data Directed Design***.

A key step is to identify the **types of data the problem involves**.

Example: Supermarket Docket

We will work through an example based on a part of a supermarket checkout system.

The full system involves *hardware* such as the scanner and till, *software* to take the sequence of scans (barcodes) and produce the final bill, and lots of other activities, *organisational procedures* such as the operation of the checkout, management of the database of products and so on. This is part of a larger integrated system to deal with stock management, bookkeeping, and all the other matters to do with running a business.

We will look at a small part of the overall system:

- the scanner produces a sequence of barcodes
- we produce an itemised docket with a total cost

For example, the scanner produces a sequence of barcodes, like:

4719 1112 1113 3814 1234

and our program will print:

Alonzo's Mega-Mart

Frozen Pizza.....	6.49
Mars Bar.....	1.60
Unknown Item.....	0.00
Hokkien Noodles.....	2.05
Chianti, 1lt.....	17.95
Total.....	\$28.09

The strategy we will follow is **Data Directed Design**.

I strongly recommend that you follow this strategy. It consists of a sequence of well-defined steps.

1. Understand the Problem

- Thoroughly analyse the aims and requirements of the problem.
- Work some examples by hand.
- Ask questions (of yourself and others) and look for special cases.

For example, what should be done if the barcode does not appear in the database?

2. Identify the Objects

What are the “things” that the problem involves?

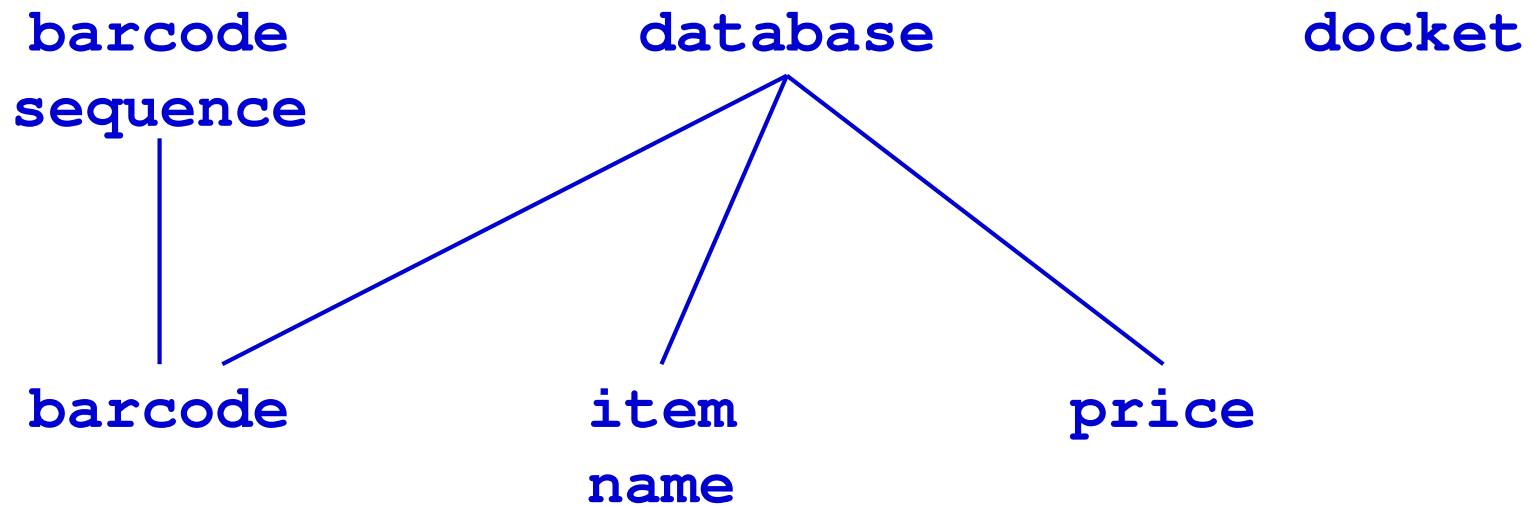
Abstract Data Types (ADTs) are “types” related to the **problem domain** rather than the programming language.

For the supermarket problem, I identified:

- barcodes, names and prices of items
- sequences of barcodes from the scanner
- the *database* linking barcodes to items and prices
- the docket itself

These ADTs are not all independent concepts, and some are much simpler than others.

The hierarchy and relations between the ADTs:



Choose some names for the ADTs

TillType : sequence of barcodes

BarCode : bar codes

Name : name of item

Price : price of item

Database : database of supermarket stock

Docket : printed bill

3. Identify Basic Operations on Objects

All data types are completely determined by the operations on that type.

TillType: Process in sequence — extract the first bar code, then the second etc.

BarCode: Compare for equality. Depending on Database representation, we may want other relational operations.

Price: Arithmetic and reading.

Database: Look up information about an item, using Barcode as an index. Realistically, we would want other operations to maintain and update the database, too.

Name and **Docket**: No operations in program, but read by system users.

There is never only one correct answer. For example, I keep talking about “items” so they should probably be included as another ADT.

4. Choose Representations of Objects

Based on the operations identified above, choose appropriate types to represent the ADTs.

`TillType`: Only needing sequential processing suggests that a simple list would be sufficient:

```
type TillType = [BarCode]
```

`BarCode`: I don't know enough about bar codes to make an informed judgement. Why an integer? Do we ever do arithmetic on a bar code? Nevertheless:

```
type BarCode = Int
```

Name: A string of characters is sufficient.

```
type Name = String
```

Price: The obvious choices are Float (number of dollars) or Int (number of cents). In general, Int is preferred.

```
type Price = Int
```

Docket: A string, a sequence of lines (strings), a “file” or such like.

```
type Docket = String
```

Database: There are various standard representations for look-up tables. The simplest is an **association list** — an unordered sequence of **records**, each with a **key** and other data fields. This representation would be far too inefficient for anything other than a tiny database. For this exercise we will take the simple approach.

The key is Barcode. The records need to contain the bar code, the name and the price. (We can infer that from the problem requirements.)

```
type Database = [ (Barcode, Name, Price) ]
```

Modules

Two observations about the [Database](#) ADT:

- The operations are not so straightforward;
- It is easy to imagine the database ADT being used for purposes other than this specific exercise.

Generally, for all but the simplest ADTs, it is appropriate to build them in their own module.

Modules

There are several advantages to packaging ADT in modules:

- While developing that module we can concentrate on *HOW* and ignore *WHY* it is needed;
- In the main application we can ignore how the ADT is implemented as long as we understand *WHAT* it does;
- Modifications to the ADT are confined to that module, rather than being scattered through a program;
- Module systems allow the programmer to control misuse of an ADT through export restrictions. That is, only certain operations are permitted.

5. Implement Basic Operations on Objects

```
TillType = [BarCode]
```

The operations (sequential processing) are simple enough that the standard functions (list patterns) correspond.

```
BarCode = Int
```

No other operations other than equality comparison (`==`).

```
Price = Int
```

Standard arithmetic operations. We also need to display the price as dollars and cents. This formatting could also be considered a Docket operation.

```
formatCents :: Price -> String
```

```
formatTotal :: Price -> String
```

Docket = String

Name = String

Standard string processing operations.

Database = [(BarCode, Name, Price)]

A *lookup* function: given a bar code, return the record with that key.

```
find :: Database -> BarCode -> (Name, Price)
```

6. Factor Process into Manageable Parts

We now have some objects (types) and basic operations with which to construct our solution.

We need to break the overall solution process down into manageable parts.

The top-level function is:

```
printBill  :: TillType -> Docket
```

How can we break that down into smaller parts?

One possible approach is to factor it into two parts:

1. look up the items
2. create the docket from the name and price information

This suggests the following operations:

```
makeBill  :: TillType -> BillType
formatBill :: BillType -> Docket
```

where the list of item information is a new type:

```
type BillType = [(Name, Price)]
```

Design is an *iterative* process. We have introduced a new data type into the design, so steps 2, 3, 4 and 5 must be repeated for that type.

The main operation (`printBill`) is just `makeBill` and `formatBill` *in sequence*.

“Sequencing” corresponds to *function composition*, so we have:

```
printBill codes = formatBill (makeBill codes)
```

Now we can concentrate on the two simpler functions.

Do they need to be factored down further?

Can they be easily implemented (in terms of the basic operations on the ADTs)?

makeBill obviously operates on TillType and the Database, so its definition will involve the look-up function and operations to retrieve bar codes from lists of bar codes.

printBill needs to construct a heading, a body and a total line. The body and the total line will depend on the names and prices of the items purchased (BillType):

```
formatLines :: BillType -> String
totalLine   :: BillType -> String
```

totalLine will be further factored into a function to *compute* the total and a function to *display* that value:

```
makeTotal   :: BillType -> Price
formatTotal :: Price   -> String
```

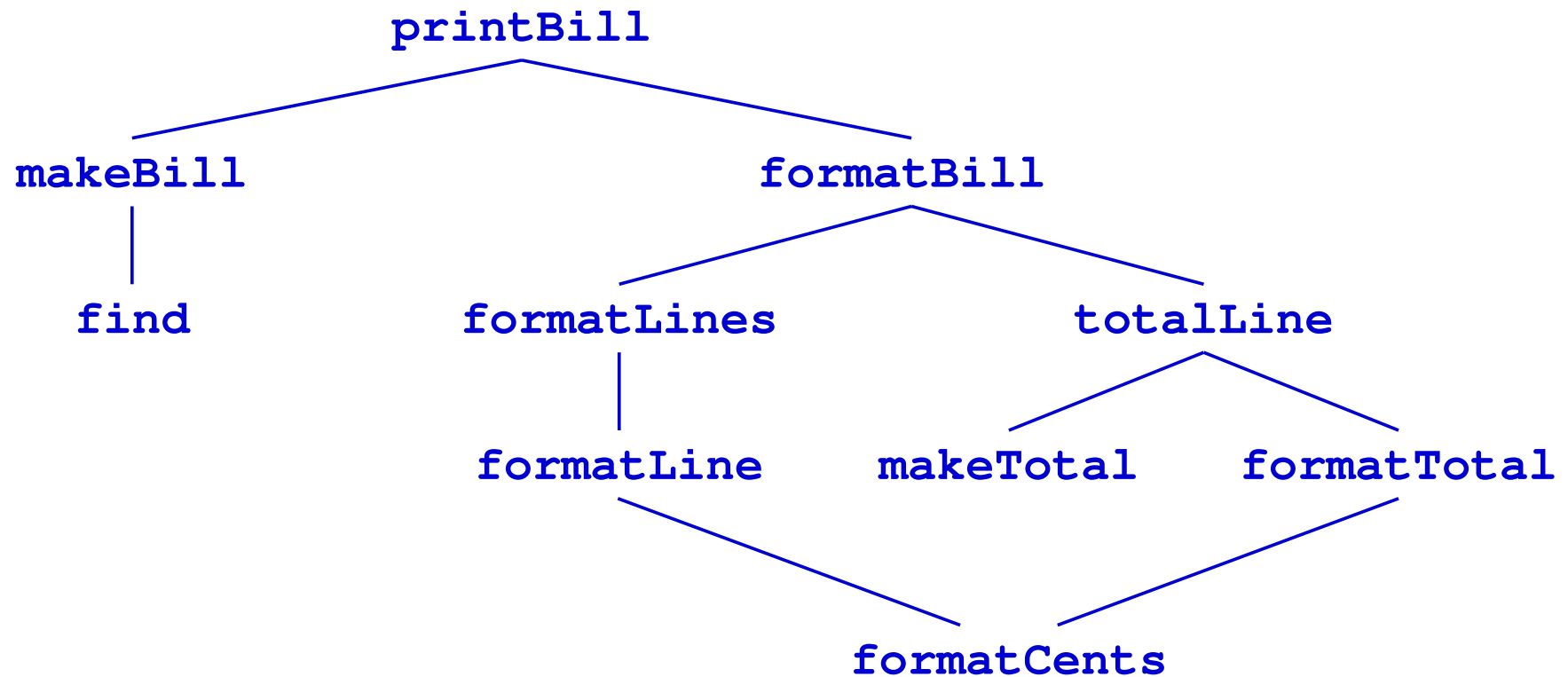
The design process is never that straightforward.

We *learn as we design*, so we make mistakes and omissions.

You should always expect to **review**, **revisit** and **redesign**.

Software development is an **iterative** process, gradually refining to a final solution.

Process (Function) Hierarchy



Summary of Data Directed Design Process

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