



Program Design

Consider the problem of developing part of a software system for use at a supermarket checkout. A scanner at the checkout produces a list of bar codes, for example:

```
[ 1234 , 3216 , 4719 , 1112 , 1113 , 3814 , 1234 ]
```

which is to be converted to a printed bill, like:

```
Alonzo's MegaMart
Chianti, 1lt.....11.95
Ayam Chili Sauce.....2.13
Frozen Pizza.....4.49
Mars Bar.....0.80
Unknown Item.....0.00
Hokkien Noodles.....2.85
Chianti, 1lt.....11.95
Total.....34.17
```



The Design Process

A summary of the design process is as follows:

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



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 bar code does not appear in the database?



2. Identify the Objects

- This is the first and most important step in the detailed design.
- 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, we can identify bar codes, sequences of bar codes, names of items, prices of items, the database and the docket.
- Clearly, these ADTs are not all independent concepts, and some are much simpler than others.



2. Identify the Objects

- Choose some names for the abstract types:

TillType : sequence of bar codes

BarCode : bar code

Name : name of item

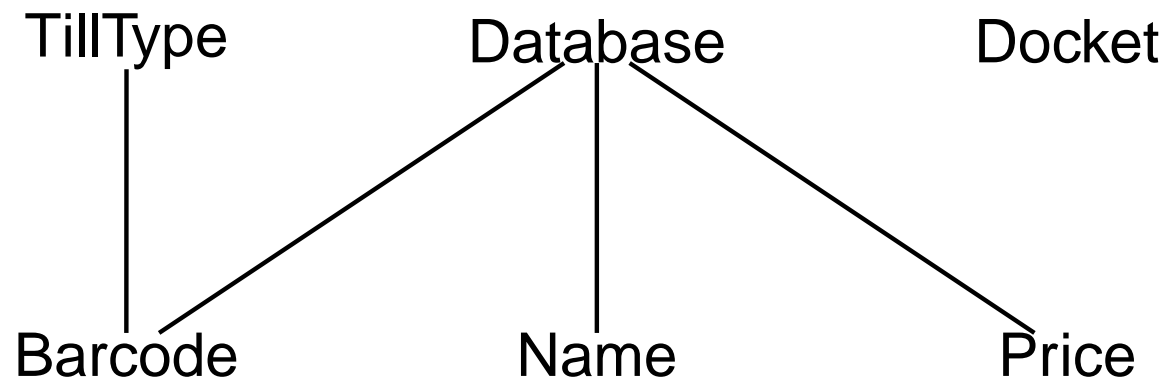
Price : price of item

Docket : printed bill

Database : database of supermarket stock

2. Identify the Objects

- The following diagram demonstrates the hierarchy and relations between the ADTs.





3. Identify the Basic Operations

- **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.
- **Name**: primitive. No operations in program, but read by system users.
- **Price**: arithmetic and reading.
- **Docket**: 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.

4: Choose Object Representations

- 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`: this can be represented as an integer:

```
type BarCode = Int
```

- `Name`: obviously, a string of characters is sufficient:

```
type Name = String
```

4: Choose Object Representations

- Price: the obvious choices are `Float` (number of euros) or `Int` (number of cents). In general, `Int` is preferred:

```
type Price = Int
```

- Docket: obviously this will be a string, or a sequence of lines (strings), or a “file” or such like. Choose:

```
type Docket = String
```

- Database: there are various standard representations for lookup tables. The simplest is an unordered sequence of *records*, each with a *key* and other data fields. The key is `Barcode`. The records need to contain the bar code, the name and the price:

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



Modules

- Generally, for all but the simplest ADTs, it is appropriate to build them in their own *module*.
- In Haskell, the first code line of a module called `Sample` (say) must be:

```
module Sample where
```
- In Hugs, the file must have the same name, i.e. `Sample.hs`
- To *use* a module in another script, include a statement like:

```
import Sample
```



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
- ADTs are often general purpose and can be reused in other applications.
- Module systems allow the programmer to control misuse of an ADT through export restrictions.

5: Implement the Basic Operations

- `TillType = [BarCode]`: the operations are simple enough that the standard functions (head or tail or list patterns) can be applied.
- `BarCode = Int`: there are no other operations other than equality comparison (`==`).
- `Name = String`: standard string processing operations.

5: Implement the Basic Operations

- `Price = Int`: standard arithmetic operations. We also need to display the price as euros and cents. This formatting could also be considered Docket operations.

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

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

- `Docket = String`: string processing operations.

- `Database = [(Barcode, Name, Price)]`: a *lookup* function is required. Given a bar code, return the record with that key.

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



6: Factor Process

- 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 toplevel 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

6: Factor Process

- 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)]
```

- The main operation (`printBill`) is just `makeBill` and `formatBill` in sequence.
- “Sequencing” corresponds to *function composition*, so we have:

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



6: Factor Process

- 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 lookup function and operations to retrieve bar codes from lists of bar codes.

6: Factor Process

- `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:

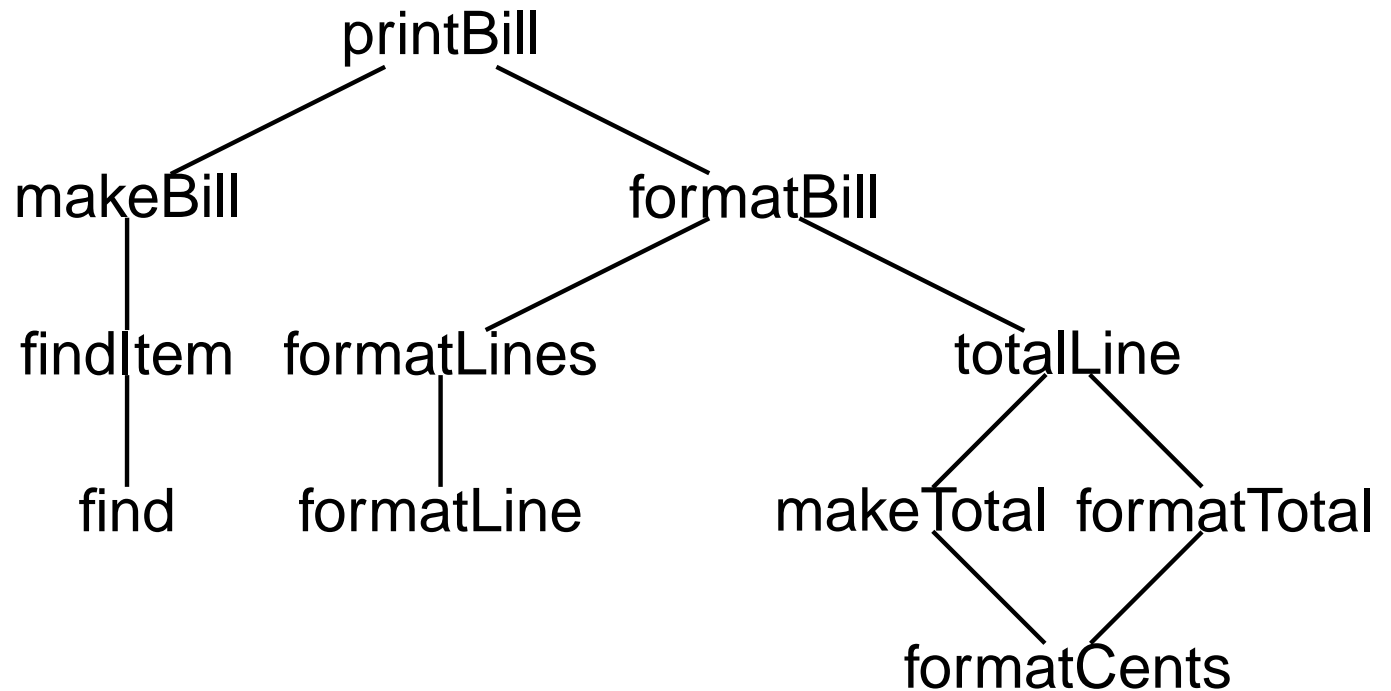
```
makeTotal :: BillType -> Price
```

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

6: Factor Process

- Alternative possibilities?
- `makeTotal` adds the prices of all the items in the list, determined by the input list of bar codes. So we could try:
`makeTotal :: TillType -> Price`
and look up the database *again* for the list of prices:
`priceList :: TillType -> [Price]`
and then add them together.
- In general, because the database will be very large, it will be less efficient to search once for the item names and then again for their prices.

Function (Process) Hierarchy





Case Study

1. Understand the Problem

- Find all the misspelt words in a text file. Report as a lexically ordered list.

2. Identify the Objects

- Clearly we need a dictionary against which to check words from the file.
- The result of the check is a list of words.
- The input will be a file path name.
- The contents of the file will be a sequence of characters (i.e. a string) to be interpreted as a sequence of words.
- Objects: dictionary, file path name, word, list of words



Case Study

3. Identify the Basic Operations on Objects

- look up word in dictionary
- compare words for equality
- sort list of words
- get file contents
- extract words from file contents



Case Study

4. Choose Representations of the Objects

- Words can be represented as strings:

```
type Word = String
```

- The file path name will also be a string, as defined in the prelude:

```
type FilePath = String
```

- The contents of the file will also be a string, to be broken into a sequence of words:

```
type Words = [Word]
```

- How should we represent a dictionary? We will choose the simplest representation:

```
type Dictionary = Words
```

Case Study

6: Factor the Process into Manageable Parts

- The input is a file path. The output is an ordered list of the words that do not appear in the dictionary.
- Four main steps seem appropriate:
 1. Get the contents of the file
 2. Get the words from (the contents of) the file
 3. Check if each word is in the dictionary
 4. Sort the misspelt words into alphabetic order

file path	⇒	readFile	⇒	contents
contents	⇒	wordsFromString	⇒	list of words
list of words	⇒	misspelt	⇒	misspelt words
misspelt words	⇒	sort	⇒	alphabetical list

Case Study

- The overall program will be the composition of functions corresponding to those steps:

```
wordsFromString :: String -> Words
```

```
misspelt :: Words -> Words
```

- Note that the dictionary is fixed, not an argument.

```
sort :: Ord a => [a] -> [a]
```

- The composition (or sequence) is:

```
check :: String -> Words
```

```
check contents =
```

```
    sort (misspelt (wordsFromString contents))
```

- or equivalently:

```
check = sort . misspelt . wordsFromString
```



Case Study

5. Implement the Basic Operations on Objects

- **Look up a word in the dictionary:**

Since the dictionary is a simple list of words, we can just use the standard functions `elem` or `notElem`.

- **Compare words for equality:**

The equality operator (`==`) applies to strings.

- **Sort list of words:**

We have developed an insertion sort function. It can be used since strings are an ordered type.

- **Get file contents:**

Use the standard function `readFile`.



Case Study

- **Extract words from file contents:**

This is more complicated.

Decompose this process into smaller parts.

Words are lists of alphabetic characters, so:

1. Skip over nonletters in the file contents string (`skipToLetter`).
2. The first word now appears at the beginning of the string (`firstWord`).
3. Repetitively apply those steps to the remainder of the string (`restOfString`) until there are no more words left.

Case Study

- Some standard functions are useful.

```
isAlpha :: Char -> Bool
```

- tests whether characters are letters.

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

`takeWhile p xs` returns the longest initial segment of `xs` whose elements all satisfy predicate `p`.

`dropWhile p xs` returns the remaining portion of the list.

- We have:

```
skipToLetter = dropWhile (not . isAlpha) s
```

```
firstWord = takeWhile isAlpha skipToLetter
```

```
restOfString = dropWhile isAlpha skipToLetter
```



Case Study

- Putting them together to return the sequence of words:

```
wordsFromString :: String -> Words
```

```
wordsFromString s
```

```
  | firstWord == ""
```

```
    = []
```

```
  | otherwise
```

```
    = firstWord:wordsFromString restOfString
```

```
where
```

```
skipToLetter = dropWhile (not . isAlpha) s
```

```
firstWord = takeWhile isAlpha skipToLetter
```

```
restOfString = dropWhile isAlpha skipToLetter
```