```
      // remove all duplicates of x from rest of v
      while (j < v.size())
      { if (x.equals(v.get(j)))
        { v.set(j, v.lastElement());
          v.remove(v.size() - 1);
        }
        else
        { j++; }
      }
    }
  }
}
```

We don't require $v \, != \, null$ in precondition, since this case is dealt with normally in method body: it is valid to call method with null argument. Since *VectorUtil* is a utility package, it should work in widest range of possible circumstances. Indeed it should ideally tolerate null elements inside vector as well.

Part 2: Design Techniques

This part describes processes and techniques that can be used in object-oriented design.

Design Steps

Following steps in design can be distinguished, independent of notation and application:

1. *Architectural design*: define global architecture of system, as a set of major subsystems, and indicate dependencies between them.

   For example, partitioning a system into a GUI, functional core, and data repository. The GUI depends on core because it invokes operations of the core, and core may depend on GUI (if it invokes GUI operations to display result of computations, for example). The core depends on repository. There may be no other connections.

2. *Subsystem design*: decomposition of global subsystems into smaller subsystems which handle subset of its responsibilities. Decomposition continues until clearly identified *modules* emerge at bottom of subsystem hierarchy. A module typically consists of a single entity or group of closely related entities, and operations on instances of these entities.

   Eg, in restaurant system can identify subsystem of functional core which deals with waiter operations only (creating, extending, modifying and completing orders, and bill payment). This in turn can be broken down into two subsystems, one for order processing and one for bill processing. These subsystems may be considered to be modules themselves or may possibly be broken down further.

3. *Module design*: define each module, in terms of:
   (a) data it encapsulates – eg: list of attributes and their types or structures;

   (b) operations it provides (external services) – eg: their names, input and output data, and specifications. This is called *interface* of the module.

4. *Detailed design*: for each operation of module, write down list of tasks it has to carry out. From this may identify *helper* methods – private methods of module (perhaps belonging to auxiliary classes of module) that carry out part of functionality of operation. This process is continued until explicit and detailed definition of all data structures and operations within each module can be produced.

   In Java, for example, a module would usually consist of one main class whose public methods correspond to module interface. May be other subordinate classes within module, especially if data structures managed by module are complex.

Design Stages

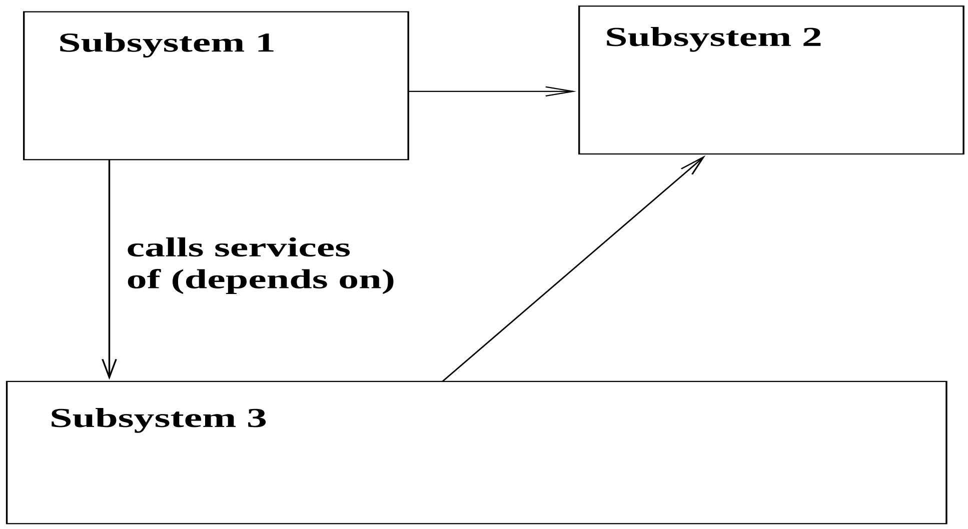
In each stage aim to minimise dependencies between components, maximise their internal coherence, and maximise flexibility and evolvability of system. At end of each stage carry out *design review* to check decisions made were sensible and meet desirable properties of design: low coupling, etc, and that design supports functional requirements of system use cases.

Architectural Design

Global architecture of system should be stable over lifetime of system.

Organise architecture to minimise effects of future changes. Eg, system may originally use programmed file-based database, then evolve to use external commercial database. By separating database management operations into separate subsystem, change can be achieved without affecting other subsystems.

Architectural Design Notation



Restaurant System Architectural Design

Architectural design description for restaurant system consists of above design diagram and following subsystem descriptions:

GUI User interface of system, providing means for users of system to: create an order; add items to an order, modify an order before it is confirmed; confirm an order as complete; create a bill for a terminated order; (etc)

It participates in all use cases of system. It is required to be as simple to use as possible, to be highly responsive and to be designed to minimise human error in high stress situations (particularly for waiter, bar and kitchen staff operations). Separate interfaces must be provided for waiters, kitchen, bar, manager and stock controller. There must be security control so that waiters, kitchen staff, bar staff and stock controllers can only access interface, data and operations specific to their role. Managers can access all interfaces, data and operations.

Functional core This performs all data processing and calculations involved in responding to updates and requests from GUI, updating and retrieving data from database, and generating data such as performance graphs for presentation in GUI. It is involved in all use cases.

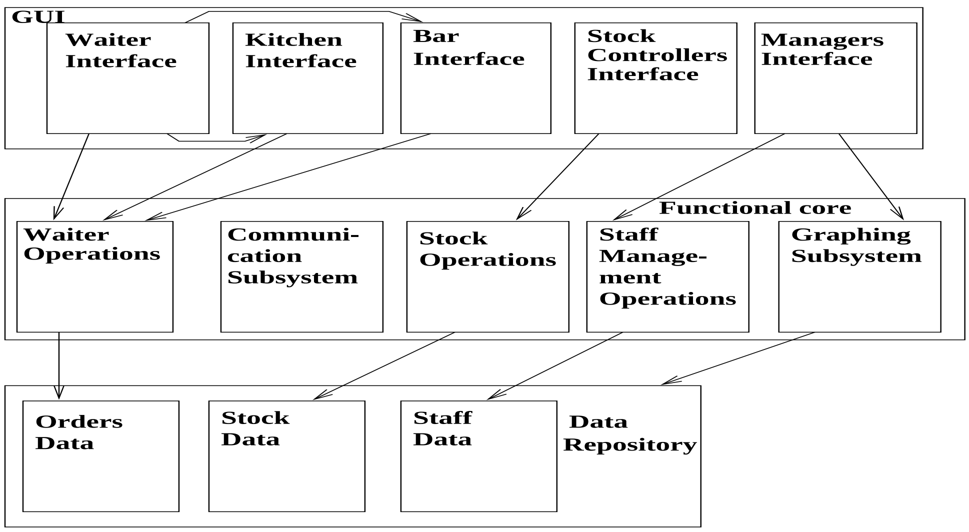
Data repository This should store in a persistent manner all data required by application, as defined in analysis class diagram, and provide operations to read and update this data. It is involved in all use cases of application. It should not allow access to stored data except via interface of restaurant management system application. It should run on a Windows 2000 platform.

Subsystem Design

Goal of subsystem design is to identify subsystems of existing subsystems of the system, dividing these in turn until a level of basic modules is reached.

For example, in restaurant system, subsystems of GUI are the interfaces for waiter, kitchen, bar, stock controller and manager. Subsystems of functional core are groups of operations for these different users, plus other utility subsystems such as a graph production subsystem, or a communications subsystem.

First Subsystem Architecture of Restaurant Management System



Documentation of this stage consists of architecture diagrams plus textual descriptions of each subsystem. For example:

Waiter interface subsystem Provides constructor to display waiters GUI on waiters console. Provides means for waiter to initiate a new order for a table, to display current and other orders, to add items to an incomplete order, to complete an incomplete order, to confirm delivery of an order item, and to generate bill for an order and record payments for that bill. It also displays changes in status of an ordered item.

It is involved in all use cases in which waiter is an agent.

Waiter operations subsystem This supports operations

- $createOrder(t : Table, name : String)$ and $createOrder(t : Table)$ to construct a new order;
- operations $add(x : OrderItem)$, $addVariation(x : OrderItem, v : Variation)$,

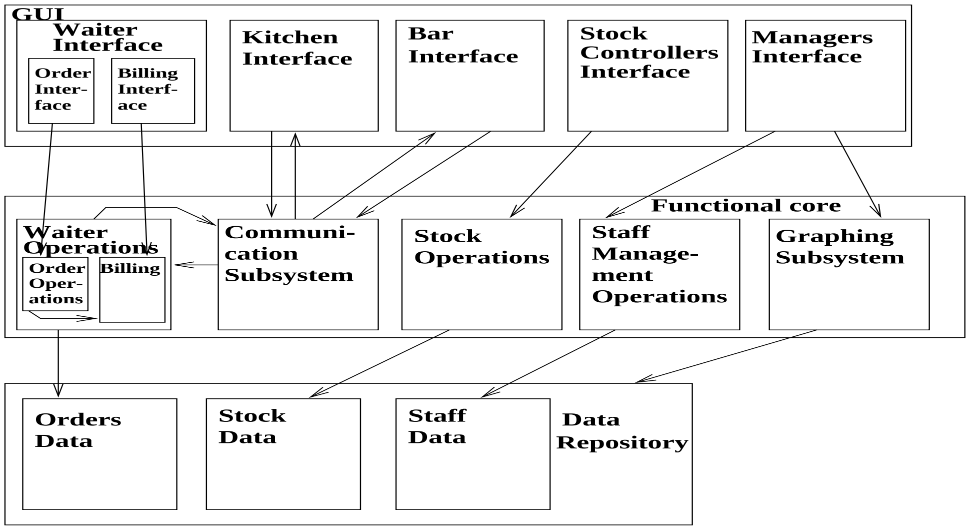
$removeVariation(x : OrderItem, v : Variation)$ to modify current order information,

- $completeOrder()$ to complete current order and initiate its sending to the kitchen and bar, and
- $confirmDelivery(x : Order, y : OrderItem)$ to confirm delivery of item y from order x.

There are similar operations to view any named order and all orders, and to initiate and carry out bill processing.

Minimising dependencies at this design stage means reducing the number of dependencies between subsystems as much as possible.

Enhanced Subsystem Architecture of Restaurant System



Module Design

Goal of module design is to identify in detail interface of individual modules, ie, set of operations/methods they offer as services to other modules, and signatures of operations. Data managed by modules is also identified, ie, set of classes from analysis model that it involves.

In restaurant system, billing module will manage *Bill*, *BillItem* and *Payment* entities from analysis model and will provide operations:

- $generateBill(x : Order) : Bill$,
- $add(p : Payment)$,
- $setDiscount(amount : int)$,
- $printBill()$,
- $completeBill()$.

Having decided on interface and outline functionality of each operation, can provide specifications:

```
module: Billing
data: Bill, BillItem, Payment
use cases: Bill order, complete payment
operations:
  generateBill(items: List): Bill
  // REQUIRES: items is non-null and consists of those
  //   order items (all non-null) to be added to bill
  // EFFECTS: returns a Bill which has a BillItem
  //   for each input order item, with cost
  //   set to item price plus the prices of any
  //   variations. The Bill total is the sum of
  //   the BillItem costs. The Bill is displayed
  //   on the billing interface and set as the
  //   current bill. Its date is set by
  //   the Bill constructor to the current date.
```



```
  add(p: Payment)
  // REQUIRES: current bill is non-null,
  //   and p non-null.
  // EFFECTS: Adds p as a payment for
  //   the current bill.
  //   The display is updated to add this to
  //   the list of payments, and to show the
  //   amount still remaining to be paid.

  setDiscount(amount: int)
  // REQUIRES: current bill is non-null.
  // EFFECTS: Removes amount from total of current
  //   bill and shows this in list of discounts on
  //   the bill display interface.

  printBill()
  // REQUIRES: current bill is non-null
```

```
// EFFECTS: displays printing menu on
//   billing interface and carries out
//   printing of all bill items, discounts
//   and total on the selected printer.

completeBill()
// REQUIRES: current bill is non-null
// EFFECTS: sets verified flag to true in the
//   current bill, adds this bill to archive of
//   bills and sets current bill to null.
```


Detailed Design

Goal of detailed design is to define in detail data structures encapsulated by each module, and classes and operations of module.

This involves listing tasks which each operation needs to carry out, and defining subordinate operations where necessary to perform parts of tasks. In turn, this may lead to separation of module data into separate classes within module, with helper operations becoming methods of appropriate classes. A class is also defined to host module operations themselves, and any data that remains in this module.

Operations should be defined in enough detail (eg, by using pseudocode) that all invoked operations are explicitly listed and therefore all module dependencies are completely identified at end of this stage.

For example, for *generateBill*(*items* : *List*) : *Bill* operation of *Billing*, could have list of tasks:

- Check that input *items* is non-null: if not, terminate with an alert to user that something has gone wrong.
- Set running total to 0.
- For each element of *items*, get price, and list of variations/extras for that element.
- Add item price and prices of all its extras together, and create a *BillItem* with name equal to item name, and cost equal to sum. Add sum to running total.
- When all items have been processed, create a *Bill* with accumulated total cost and set of *BillItem*s, and return this as result of operation. Display bill on billing interface.

This is fairly explicit and algorithmic already, this may not be case for all operations. Also we have been able to directly use analysis model entities here, without needing to define additional structures.

The corresponding pseudocode for this operation is:

```
public Bill generateBill(List items)
// REQUIRES: items is non-null and consists of
//   those order items (all non-null) to be
//   added to the bill
// EFFECTS: returns Bill which has a BillItem
//   for each input order item, with cost set
//   to item price plus the prices of any
//   variations. The Bill total is the sum of
//   the BillItem costs. The Bill is displayed
//   on the billing interface and set as the
//   current bill. Its date is set by
//   the Bill constructor to the current date.
{ int total = 0;
```

```
    List billItems = new Vector();

    if (items == null) // defensive programming
    { currentBill = new Bill(total,billItems);
      // display currentBill on billing interface
      return currentBill;
    }
    for (int i = 0; i < items.size(); i++)
    { OrderItem oi = (OrderItem) items.get(i);
      // if (oi == null) raise exception?
      String nme = oi.getName();
      int pr = oi.getPrice();
      List extras = oi.getVariations();
      int extrasTotal = 0;
      for (int j = 0; j < extras.size(); j++)
      { Variation ex = (Variation) extras.get(j);
        extrasTotal += ex.getPrice();
      }
```



```
      pr += extrasTotal;
      BillItem bi = new BillItem(nme,pr);
      billItems.add(bi);
      total += pr;
    }
    currentBill = new Bill(total,billItems);
    // display on billing interface
    return currentBill;
  }
```

This is 90% of way to implementation, to polish things off also need to handle exceptions such as null *Variation* objects, i.e., raise a Java exception, simply give a warning message to user and return some sensible result, or silently tolerate error.

Guidelines for reducing dependencies in design

- Minimise knowledge that one module has about another.
- Minimise knowledge that a subsystem has about internal details of the modules within it.
- Minimise number of methods provided by a module to others (its *public* methods).
- Use library classes (or preferably interfaces) and primitive types as parameter types for public methods of a module.
- Optimise conceptual and functional cohesion of modules: they should manage one entity type or a closely related set of entity types, a single resource, etc.
- Review and refactor designs at regular intervals during development to maintain design quality.

Generic Design Techniques

Following techniques can be used at any of design stages described above.

- *Factoring* – splitting an element into subelements, such as splitting a large method/operation into 'helper' methods, and identifying commonalities between different methods which can be factored into separate units.
- *Generalisation* – identification of how subsystems, modules, methods or classes can be generalised to be more widely useful within an application and more able to support changes or extensions in application requirements.
- *Specification/layered design* – replacing a dependence of a client on a particular supplier by a dependence on a generalised interface, together with a specification of behaviour that clients can expect from this interface.

Factoring

At architecture and subsystem design stages, this technique can be applied to identify additional subsystems which provide common services used in several other subsystems.

Eg, in restaurant management system, both staff management and order management subsystems are likely to need facilities for constructing graphs. Rather than duplicating these facilities in the separate subsystems, define a *Graphing* subsystem which provides operations to define, display and print graphs.

In some cases suitable subsystems may already be available as library packages. For graph construction and display in Java, for example, the JSci package can be used.

In module and detailed design factoring can be applied to split data of module into classes, and split operations of module into sub-operations and methods of classes in the module.

Factoring is also important in reducing size and complexity of methods – split a large method down into several subtasks and replace code of subtasks by calls to new (usually private) methods which perform them.

Finally, factoring can be used to package up a piece of code that appears in several places as a method in its own right. For example essentially same code may occur in different places just working with different variables (eg: `z = x*x + y*y` in one place, `s = p*p + q*q` in another). Replace separate occurrences of this calculation by calls to a method for it: `z = sumSquares(x,y);` and `s = sumSquares(p,q);`. Then if calculation needs to change, only the method needs to be modified, instead of all separate occurrences of calculation.

Generalisation

At architecture and subsystem design level an example of generalisation would be extending a subsystem to provide services which are natural additions to operations already required, and which are likely to be required in future enhancements of the system.

Eg, for a one-many relation such as *Order-OrderItem* association in restaurant system, may have a requirement for an operation to add a pair (x, y) to this relation but no requirement to remove a pair. In many cases removal would be a potentially useful additional operation, so could be added to subsystem that manages relation.

Data models may be generalised to deal with a wider range of possible situations than are handled by the current system, eg, by generalising a 1-many association to be many-many.

Specification/Layered Design

REQUIRES and EFFECTS specifications should be added to methods, and invariants of class data or loops expressed. Eg, for list variable, should make clear what types of element can be in list:

```
public class Company
{ private List employees = new Vector();  // of Person objects
```

Similarly, express type of object pairs in a *Map*:

```
public class UserDatabase
{ private Map users = new HashMap();  // String --> User
```

or that a variable recording someone's age can never be negative:

```
public class Person
{ private int age = 0;  // age >= 0 always
```

Keyword *final* in Java can be used to prevent redefinition of an element, particularly assignments to a variable:

```
public boolean equals(final Object obj)
{ ... }
```

indicates that method will not assign to the *obj* variable in its text. However an updater method of *obj* may still be applied to it.

Layered design means that one module should only depend on *specifications* of other modules, not on implementation details. Common case in Java is where a *Gui* class depends on some back-end class:

```
public class Gui
{ private BackEnd b = new BackEnd();
  ...
}

public class BackEnd
{ ... }
```

This makes *Gui* specific to exactly this one *BackEnd* class. Instead, decouple GUI from back end by making *Gui* depend on an interface instead:

```
public class Gui
{ private BackEndInterface b;
  ...
```

```
  public void setBackEnd(BackEndInterface be)
  { b = be; }
  ...
}

public interface BackEndInterface
{ ...
  signatures and specifications of methods
  of back end used by Gui
}

public class BackEnd implements BackEndInterface
{ ... }
```

This allows any class that implements *BackEndInterface* to be used by *Gui* class.

Part 3: Software Architectures

Architecture of piece of software is overall organisation of program code into separate subparts, responsible for different aspects of functionality.

Eg, many traditional application programs can be separated into a 'Front end' responsible for dealing with interaction with users (ie, the GUI), and a 'Back end' responsible for storing data and performing calculations in response to user requests (relayed via GUI).

Many web-based applications can similarly be separated into a client-side GUI (HTML pages and forms) and server-side processing (data access/computations and generation of web pages).

Benefit is that these separate parts can be developed, tested and modified independently, and may be deployed on separate hardware.