An Iterative-Design Model for Reusable Object-Oriented Software

Sanjiv Gossain
Bruce Anderson
University Of Essex, UK. *

Abstract

We present an iterative-design approach for reusable object-oriented software that augments existing design methods by incorporating iteration into the design methodology and focuses on the set of problems within the domain, encouraging reuse of existing design information. The model has five separate stages which are described, before an example design is outlined using the model with sample code constructs in C++. Our results have shown a high degree of code reuse when using the model, directly attributable to two distinct design stages. An analysis of these results is also presented.

1 Introduction

In this paper we present an iterative-design model for reusable object-oriented software based on the experience of the authors in designing code intended for reuse directly, as components, and indirectly, through extension and modification. This approach is arrived at after using existing object-oriented design methods [3] [16] and finding that they do not satisfy our needs on two fronts. They do not recognise the iterative nature of software development, and hence fail to incorporate iteration into their approach. They also do not address the decisions faced when adding classes to an existing hierarchy or when composing solutions from both existing and new classes.

These approaches must therefore be extended in two ways, to incorporate iteration into the design approach where appropriate, and to design code that is to be part of a hierarchy or framework of classes for an application domain.

It should not be seen as a complete design alternative to already accepted approaches, but as an augmentation of those ideas that must be made to cater for continual growth of objects and methods within a hierarchy and for addition of new levels of abstraction. The design strategy shifts the emphasis from addressing one particular problem or application to focussing on the set of problems within the domain, hence concentrating on the reuse of domain concepts.

In section two we provide an overview of the design model, followed by a more detailed examination of each stage and the processes involved, aided by sample design scenarios. An outline case study from the VLSI Routing domain is presented in section eight, with an analysis of our findings using the design model, through measures of reusability and design time in section nine.

2 Model Overview

There are five main stages to the design model, each of which has substages of varying com-

*Bruce Anderson can be contacted at Department Of Electronic Systems Engineering, University Of Essex, Colchester CO4 3SQ. UK Tel: +44 206 872805/872436, email bruce@essex.ac.uk plexity. A diagrammatic representation of the model is shown in Figure 1. It is well suited to the incremental development of object oriented software systems placing emphasis on evolution and growth of software through continual review and discussion of each stage.

As we observe from Figure 1, three of the stages take place within a component environment such as that provided by classes within an existing domain-specific class hierarchy. This is not such a clear-cut case if one is designing classes without any domain-specific classes available. The component environment is then dependent only on existing component libraries such as that of Smalltalk [7], or the NIH C++ library [8]. These libraries are general and express no concepts of the application domain.

McCain [15] proposes that there should be "*at least three different points of view other than the component programmer*" represented at each review. Such an approach should be followed in the iterative model, where the different perspectives to be considered are those of the domain analyst (expert here), software component engineer and component user. These different views do not necessarily mean three different people, as will be shown later.

The five main stages are:

- *Domain Analysis:* Essential features of the domain are captured. Initial candidate classes are identified.
- *Abstraction:* The creation of abstract classes from initial candidate classes.
- *Specialization:* Abstract features broken down and used to derive the concrete classes of the domain.
- *Evaluation and Revision:* Fine tuning of the classes to be used to meet the needs of the application.
- *Implementation:* Coding and use of the classes to create solutions. Test at high and low levels and reimplement where necessary.

The outputs of each stage are shown in Figure 2. It is important to point out that each stage can highlight design drawbacks in any of the previous stages and as such can mean iterating back to the cause of the problem. This is not unusual as a good design will take many iterations. There is no such thing as an incorrect design, for all are correct if they accomplish their task, but some are more powerful than others. It is these more powerful designs that we strive for.

3 Domain Analysis

Domain Analysis (DA)[17] is an activity prior to systems analysis that results in a domain model, describing the characteristics of objects and operations common to all systems within the domain. Prieto-Diaz proposes a formal method for DA which produces a collection of reusable components specific to the problem domain i.e. candidates for the collection of classes.

Reusable components are difficult to obtain first time around as good reusable design takes many iterations [13], and so, in the iterative-design model, not all of the DA process described by Prieto-Diaz is required. We only require a modified, partial domain analysis activity. Thus, the context diagram for our modified DA differs from the original, in that it incorporates feedback paths to allow for iteration and produces different outputs.These outputs can be seen as early versions of the output from Prieto-Diaz's DA. This is because the modified DA has reduced the overall number of stages from three to two, in order to allow for the further steps needed to generate a class hierarchy.

Instead of a library of reusable components, the modified DA process produces an initial set of classes that are reflections of the main conceptual entities within the domain and as such are tentative proposals for domain classes. The software engineer gains valuable domain insight through domain analysis and this can be used when implementing further applications within the domain, or if iterating through the DA

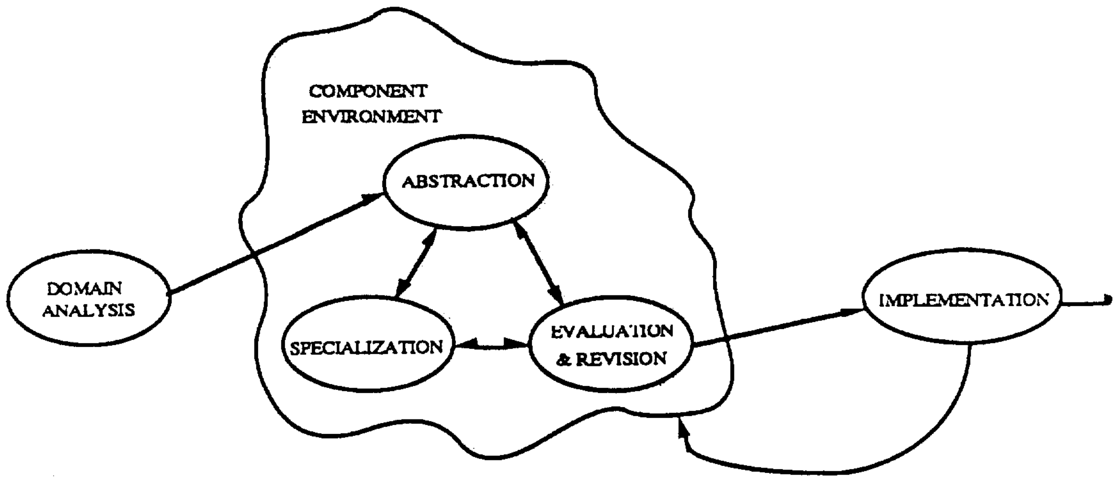
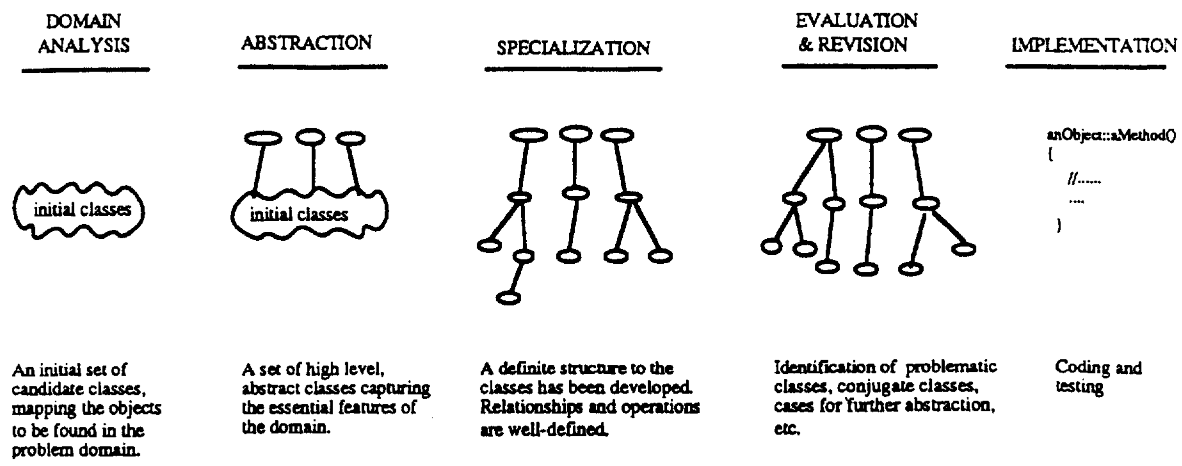


Figure 1: Overview of the Iterative-Design Model



```
anObject::aMethod()
{
   //......
   ....
}
```

An initial set of candidate classes, mapping the objects to be found in the problem domain.

A set of high level, abstract classes capturing the essential features of the domain.

A definite structure to the classes has been developed. Relationships and operations are well-defined.

Identification of problematic classes, conjugate classes, cases for further abstraction, etc.

Coding and testing

Figure 2: Design stage outputs

Figure 2: Design stage outputs

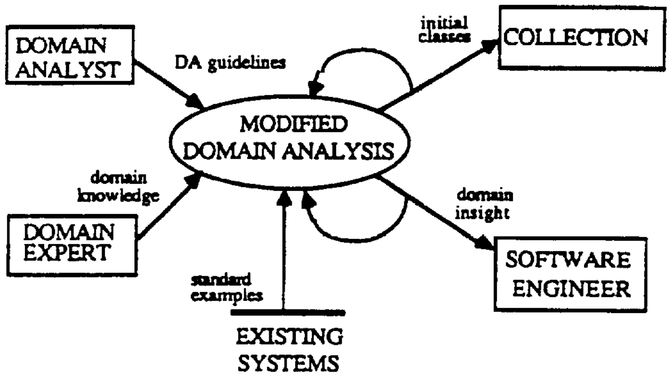


Figure 3: Context diagram for modified domain analysis

process once more.

The pre-DA activities proposed by Prieto-Diaz comprise five stages :

- Defining the DA approach: A brief statement of how the DA is to be undertaken.
- Bounding the domain: A definition of the boundaries of the domain, limiting the type of applications to be considered, and the amount of information examined. Domain Analysis for generic applications makes the importance of DA more acute as this activity determines to a great extent the range of applications that can be developed using the class hierarchy. From this we can gauge the *domain range*, a measure of the extent of our analysis, and hence the type of applications that can be easily developed.
- Defining the domain: A formal statement that clearly and unambiguously, defines the domain.
- Selecting knowledge sources: Common knowledge sources that can be used are domain experts, widely used literature on the subject and current examples of existing systems. It is important that the sources provide a 'balanced' view of the domain. The choices suggested here should, we believe, ensure such a balance and hopefully eradicate any chance of a biased source of knowledge. Such information is well documented from literature in the Artificial Intelligence field [5].
- Defining the DA requirements: A list of the topics in the domain characterizing the domain and detailing the issues relevant to the domain.

From here on the modified DA differs from the original in both its goals and its methods. Unlike the original approach which fosters the belief that a set of reusable components can be achieved through an analytic approach, our belief is that reusable design is difficult to obtain at the first attempt, and as such usually involves numerous reviews, evaluations and revisions of classes. We feel that it also requires:

- insight
- experience
- discussion and review
- an iterative design approach

Insight and domain experience can be gained only after working in that domain and after discussion with domain experts. Discussion and review play an extensive part in any successful attempt to generate reusable software and as a result require an iterative design approach. Experience in designing for reusability is also an essential prerequisite for success.

The domain analysis is a relatively straightforward approach but can have limitless iterations, depending on the discussions between the software component engineer and domain

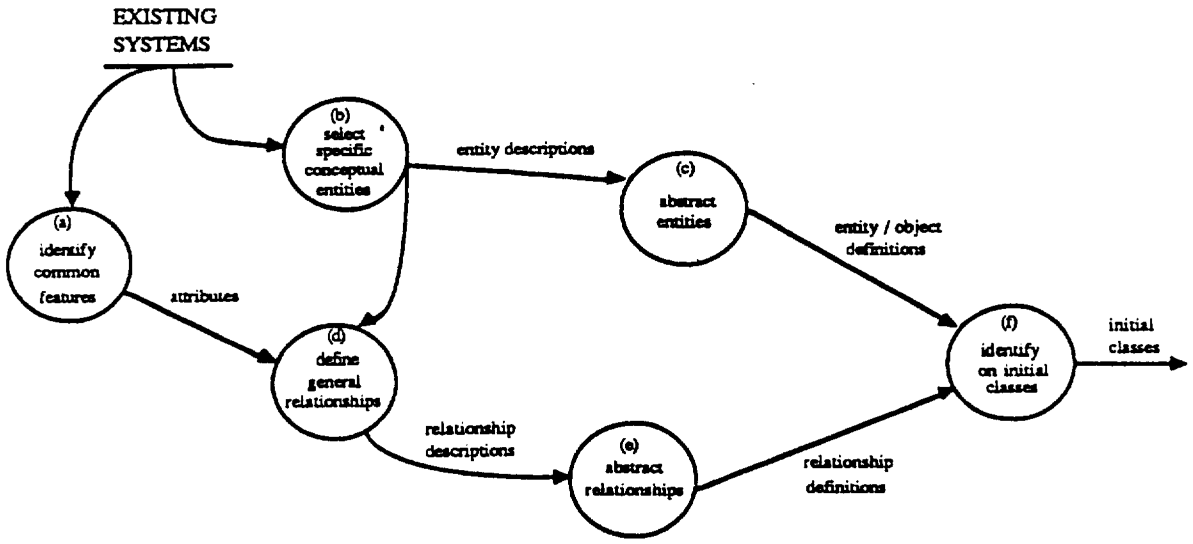

EXISTING SYSTEMS
(a) identify common features
(b) select specific conceptual entities
entity descriptions
(c) abstract entities
entity / object definitions
attributes
(d) define general relationships
(f) identify an initial classes
initial classes
relationship descriptions
(e) abstract relationships
relationship definitions

Figure 4: Domain Analysis Process

expert. A data flow diagram type representation of this process is shown in Figure 4.

For example, in the account handling domain of a bank transaction handling system, the stage outputs may be:

- *(a) common features:* opening/closing accounts, transferring of monies, depositing or withdrawing money, paying standing orders, direct debit etc.
- *(b) specific conceptual entities:* different types of customers/account holders, varying account types.
- *(c) abstract entities:* customer, current account, deposit account, higher-interest rate deposit account.
- *(d) general relationships:* customer J. Smith opens account number 10406654.
- *(e) abstract relationships:* customer has name, account has number.
- *(f) candidate classes:* current account, company account holder, personal account holder, deposit account etc.

We have worked extensively on a VLSI routing system; the routing problem requires many point pairs on a surface to be optimally connected together bypassing obstructions, using a particular algorithm. Some of the common features of this domain are areas, obstacles, cells and points. We may have initial classes such as wavefront, cell, single line, gridded area, ungridded area and multi-section line. Each of these, as in the banking example, correspond to a well-defined feature of the problem space and may or may not warrant encapsulation into components. They are merely a collection of possibilities.

The designer may find it difficult to decide whether or not a specific candidate class warrants encapsulation. This decision is dependent upon the consideration of certain criteria, which are beyond the scope of this paper, but can be found in [3, 10, 14]. Such criteria include whether a candidate represents a meaningful abstraction likely to be used effectively, or whether it is representative of a domain-world concept with real properties. Often such decisions are dependent on the views of the designer and domain expert who, using the guidelines in the literature and through discussion and review, should determine the initial solution classes.

When proceeding through such stages, the attributes and operations of an entity can also be identified providing useful discussion input which can make relationships between classes more apparent. Implementation decisions on such features should be made at a later stage, but it is useful to be aware of the existence of the particular facets of an entity.

It is important to note that the DA phase of design will be revisited during the evolution of a system, as the domain extends, new applications are built and new solution methods are created.

4 Abstraction

Abstract classes are determined from the common features, descriptions and relationships of classes within the domain. They are representative of the generic objects of the domain and form the basis from which other classes are derived. They must therefore, capture the general behaviour of the domain and form the templates for future class derivation [13]. This stage of the design can thus be construed as being the most crucial in the development of any set of classes.

Finding abstract classes is not easy; it can, and will probably require several iterations before a suitable abstract class can be determined. The abstraction process cannot really be described by a data flow diagram type of diagram as decisions are made by the object designer in consultation with the domain expert, and as such take many random feedback paths and cannot really be described as "structured". These decisions are therefore dependent on the knowledge of the expert and intuitive skill of the designer (gained through insight and experience). The existence of a component environment can aid the design process considerably, both by setting examples of design and by providing useful classes upon which to build or from which to abstract.

No methodology exists for finding them, and as a result we can only propose guidelines to aid a programmer in determining an abstract class. We must identify a cluster of classes sharing one or more of the following properties:

- common fundamental identity
- common purpose
- common behaviour
- common approaches

The fundamental identity of a class cluster is some feature that separates them from other classes. e.g. current account, deposit account, higher rate deposit account, and building society account are all types of account: account is their fundamental identity. Similarly, a volume control or tuning button on a radio and brightness knob on a screen are all types of control knob: control knob is their fundamental identity.

The DA stage will identify many abstractions which share a common purpose and fundamental identity. Their approaches may be radically different but they can still be considered as candidates for sharing a common abstract class. A truly abstract class is never instantiated, its behaviour must be general to the extreme and it should be defining only the outline of a relationship with other abstract classes, rather than in detail. Abstract classes do not exist for all class clusters and it may be that the most abstract representation of a class cluster will be be "high up" in the inheritance tree i.e. it is instantiated, but whose principal function is still in outlining the general behaviour and purpose of its subclasses. This should not be seen as constituting a poor design, and a more abstract class may become apparent at a later stage.

An example of this is when designing an abstract classes for different data representations of a routing area. Each may segment its area in very different ways and have different representations for space, yet each is still a routing area and can be identified as such. Each has an area, distinct corners, and similar generic properties. We can thus proceed to define a high-level abstraction called `RoutingSurface`. The details of such a design decision are discussed later.

The banking system discussed previously may generate an abstract class for the different types of accounts. Such a general class account would encapsulate the common behaviour of all accounts, such as depositing and withdrawing money. Its sub-classes would handle the behaviour of specialized types of account.

Communication between abstractions, and general operations for each abstract class, should be determined at this stage. This will provide for a high-level view of the objects in the domain, their attributes and how they relate to each other.

Abstract classes are not always apparent at this stage of the design, and may become visible at a later time in the cycle, so this stage is frequently revisited.

It is important that the designer understand the different mechanisms for abstraction provided by object-oriented programming. As well as straightforward subclassing there is the use of pluggability, and also the use of a hierarchy of function-like classes to provide operations [10]

5 Specialisation

Once the general abstract classes have been identified, we proceed to derive the classes of the domain, the instances of which will be used in any applications. The abstract classes serve to provide an order to the domain, in that they outline the more general concepts involved in the solution space, and can thus provide a high level understanding of important domain concepts. The specialization stage requires that each abstract class is progressively made more concrete until classes are at a level satisfactory to the domain expert and domain analyst. This means that as each successive specialization is complete, we have defined a less abstract version of a class, becoming more and more specialised as we continue.

It is in this stage that previous object oriented design methods can be utilised effectively. As each stage of the specialization progresses, the designer must evaluate design constraints, operations, attributes and communication aspects of each class, as well as its relation with its base class. Such evaluations may, and often do, require a reiteration to the previous design stage. The combination of abstraction / specialization can be seen as constituting their own sub-design cycle, as there is often a distinct interplay between the two stages.

It is here that we can now begin to formally describe our classes and their relationships. A suitable notation that we can use is one proposed by Booch [4], which at this stage would involve class diagrams of various forms expressing class relationships, cardinality of relationships, and properties of classes. The attributes and operations of a class, possibly apparent from an earlier design stage, can now be firmly decided upon by the domain expert and the software designer.

The designer must also design for reusability i.e. must cater for anticipated changes. Such changes can only be highlighted by the domain expert, confirming the importance of domain knowledge in any successful reusable object-oriented design. Although one cannot expect to design components within a domain for reuse outside that domain, the design, if carried out carefully, with the aid of domain expertise, can yield more reusable components. We discuss elsewhere our experiences in designing for reuse [1] [9] [10] [12].

LaLonde [2] advocates a name design stage which is both difficult and crucial to the construction of any abstraction. The domain experts knowledge should be used to assist in this stage. Each abstraction in each layer should have a meaningful name and behaviour relevant to the domain concept it represents and its position in the hierarchy. Each operation of a class should also undergo a name checking.

Common pitfalls are sub-classing carelessly and overspecialisation. The former leads to unbalanced, wasteful designs, and can also result in too many similar classes i.e. classes with not enough difference in functionality. Overspecialisation causes too detailed classes, for the design of the main classes within a domain requires generality to be preserved in all but the lowest level of abstraction.

Returning to the banking example, the sub-classes of an abstract class account would be decided here. Current account and deposit account are two possibilities which would be responsible for their own specialised behaviour. A deposit account may require a minimum sum invested in it. A further subclass of deposit account could be a higher-rate deposit account requiring notice be given before withdrawal, and such a feature would be implemented in this new subclass. As each layer is introduced into a solution, we are providing further possibilities of reuse, at different granularities.

At the end of this design stage, we should have a set of classes that realistically match conceptual objects in the problem space. Their attributes and operations are well-defined and the inter-class communication has been identified.

6 Evaluation and Revision

Now that the classes to be used within the domain have been identified, details must be smoothed out and revisions made to interfaces, implementations, class structures, class compositions etc. These become apparent as classes are tested by scenario, evaluated and then revisions made as required.

Classes at each level should be examined in order to identify *inter-class* and *intra-class* modifications.

- Inter-class modifications

  1. Distribution of behaviour among classes. Classes should be examined so that their behaviour can be verified with what one would commonly associate with objects of that class.
  2. Classes that may be candidates for merging. Two classes can sometimes be performing very similar tasks, and as such each may duplicate functionality found in the other. Two courses of action are open, either merge the two classes into one new class, or make the two existing classes siblings with a common parent class providing the shared functionality.
  3. Classes requiring specialisation. A class can require specialisation if behaviour is deemed too general for the concept it aims to represent. This change in functionality will require a new class to replace the general functionality of the original class. It is important that the name for this new general class is correct in its representation of the new concept.
  4. Classes requiring generalisation. Generalisation is the opposite of specialisation, and is used to generalise a class that is seen to be too specific for the concept to which it corresponds. This means the behaviour for the class becomes more general and requires a new class to fill the void left by the class that is the focus for our change.
  5. Conjunct classes requiring separation. A class may sometimes be representing the behaviour of two or more concepts, such a class should be separated into two or more distinct classes. This problem is also pointed out by [14].

- Intra-class alterations

  1. Alteration of function parameters to a class. Modification of class interfaces leads to a change in parameters.
  2. Addition of new private members of a class. It is quite common that as changes occur in the class hierarchy, and classes are redesigned, and removed, new functionality associated with a class will require new private members (and new private member functions).
  3. Removal of superfluous class members. It is often the case that there are items within a class that are unnecessary and can be removed or relocated within another class.

We will have to create objects in the solution space to encapsulate and abstract things which are not concrete in the problem domain, such as operations, transformations and interfaces; we refer to this process as *objectifying* [2]. It may be that two classes share an interface involving varying degrees of complexities at different levels of abstraction; this interface could be objectified and less complicated interfaces derived from it. Such cases should come to light in this stage as the finer details of inter-class relationships are ironed out.

When testing relationships between objects in the solution space, we can identify special cases such as lack of functionality in a subclass, an overly general class or indeed over-specialisation. The detection of such errors would necessitate a reiteration from the preceding design stage, or any other previous design stage where the error had originated.

Iterations are not a sign of bad design and should be regarded as a healthy process, by which learning takes place. It will probably be several iterations before the domain expert and the software engineer are satisfied that a set of meaningful abstractions have been arrived at whose behaviour is correct within the context of the domain. It is important to instrument the iteration process both to keep track of decisions and to allow improvement of the design process.

7 Implementation

The implementation of each class can and often will require many iterations, but these can be minimised by carrying the preceding design stages carefully and logically. A difficulty in implementation can signify erroneous judgment in a previous design stage, requiring reiteration. The Implementation and Evaluation and Revision stages are closely linked with these two stages sharing the vast majority of iterating between them. Coding highlights mainly structural and behavioural problems both of which are responsibilities of the previous stage.

8 Design Example

We now present an outline design example from the VLSI routing domain using the iterative-design model.

Domain Analysis

The problem requires various point pairs on a routing area to be optimally connected together bypassing obstructions using a particular algorithm. Some of the common features of this domain are algorithms, areas, obstructions, cells, points, lines, connections and nets.

Once these have been identified we can proceed to determine the conceptual entities of the domain. In this example some of them could be:

- gridded area - cell based
- routed connection
- point pair for routing
- gridless area
- unrouted connection
- obstruction
- wavefront
- gridded area - segment based

The general relationships between these some of these concepts might be

- obstruction occupying location to location
- path on area from location to location to location
- wavefront number 3 uses cells X, Y and Z.

Using the relationship descriptions to abstract relationships, we can say that a wavefront *has* an identity, wavefront *contains* cells, path can *contain* two or more locations, etc.

At this stage we have acquired a set of general concepts within the domain, whose main properties and relationships we are familiar

```
class RoutingSurface {
      point topLeft;
      point botRight;
public:
          // ...constructors
      virtual int size();
          // .........
};
```

Figure 5: Definition of a high-level class

with. We must now make decisions on candidates for initial classes. For the routing domain, some of these classes could be wavefront, obstruction, cell, point, gridded area, gridless area etc. We should point out here that some of these classes were not identified as conceptual identities of the problem domain; this is not unusual as new possibilities for classes may arise after examining relationships between complex domain concepts. This once again stresses the iterative nature of such object-oriented design.

Abstraction

Consider the class candidates gridded area (cell based), gridless area and gridded area (segment based). They are all typical areas upon which routing is carried out, having a common identity and common purpose. They all have distinct sizes, some may be regularly shaped and some may be irregularly shaped, they can all be described by at least two different points (top-left and bottom-right in the case of a rectangular grid). It seems plausible therefore, to assume that these points are common to all routing areas we may encounter in this domain. We can then propose a high-level class RoutingSurface which can be defined by two points, in C++ this may look like the definition in Figure 5.

Further classes derived from RoutingSurface can redefine size if they are not rectangular, add extra points to define precisely their shapes and so on. As far as the output of this stage is concerned, the general outline of a routing area has been developed. The class is not abstract but may become so at a later stage of the design cycle.

```
class Router {   // each subclass
                 // will be a
     Router()    // different
                 // type of
                 // algorithm
public:
     virtual int routeIt();
     virtual int successOrFail();
                 // ....
};
```

Figure 6: Definition of abstract class Router

Similarly for routing algorithms, we can consider each algorithm to be a Router. All algorithms have a common purpose, common identity, yet have differing approaches. This leads us to define the abstract class Router (see Figure 6).

Specialisation

For the abstract class defined in the abstraction process, we can now decide on our initial specialisations of RoutingSurface. We have two distinct types of areas, gridded and gridless, our first step would be to have them as separate subclasses of RoutingSurface. Then we note that there are two types of gridded area, cell based and segment based, each one perhaps deserving a separate subclass of gridded area. We say perhaps, as this is only the initial class sub-hierarchy for RoutingSurface. It is likely that it will change in the light of other class definitions or as other design decisions are made. We have our hierarchy as in Figure 7.

Our domain expert agrees that these names are suitable and informs the designer of what the expected functionality of each class. The designer duly attempts to define the classes accordingly, taking advantage of the features available such as overloading of functions, dynamic binding and inheritance.

The Grid knows about rows and columns but nothing about what sort they are. As a result, the first definition of Grid could be that in Figure 8. Its subclass, CellGrid, (see Figure 9) will handle the details of rows and columns of

```
class Grid : public RoutingSurface {
          int rowCount;
        int columnCount;
public:
        Grid(int, int);
        int columnSize();
        int rowSize();
        virtual int area();
      };
```

Figure 8: First definition of Grid

```
class CellGrid : public Grid {
        CellList * rows;
        CellList * columns;
public:
        CellGrid(int,int);
        //.. other functions
};
```

Figure 9: Definition of a CellGrid

cells, with `SegGrid` doing likewise for segments.

We now have a set of classes that map the concepts of the problem space to our solution. As this is only the initial attempt at defining a suitable hierarchy, it is unlikely to be in the same format by the end of the design.

Similarly for our class `Router`, we may decide upon a hierarchy of the form shown in Figure 10, where lee and line are different types of routing algorithms, and `WeightedLeeRouter` and `VariableCostLeeRouter` are successive derivations of the general Lee algorithm. Fortunately, this succession of derived algorithms falls easily into a hierarchy and our domain expert is quite satisfied with the result. Not all sub-hierarchies are realised this easily and a number of iterations might be required before proceeding on to the next stage.

Evaluation and Revision

Once the classes have been defined we can now proceed to refine and tune our classes as we proceed to test by scenario, the features of each individual class.

We may note that, when passing a particular routing problem to an instance of a sub-class router, we feel that there does not exist any firm way of encapsulating the information required for a routing problem. We decide on the object task which will be used to 'pass around' information from router to router so each can attempt a solution to the problem. This is not a conceptual entity within the problem domain but is a server class used to carry information. It arises as a result of discovering inadequacies in the solution space.

We will probably need different types of tasks for different routers, as such we will go back to the abstraction stage and decide on an abstract class for our set of tasks, continuing our iteration from there. Such a feedback will result in the creation of a class, `RoutingTask`, as in Figure 11.

```
class RoutingTask {
public:
      virtual point& identifyStart();
      virtual point& identifyEnd();
      virtual void initialise();
};
```

Figure 11: Definition of class `RoutingTask`

Once the class `RoutingTask` has been created, we can proceed to the specialization stage where its sub-classes can be determined. For the Lee routing algorithm we may declare a class `LeeTask`, as shown in Figure 12. This process is similarly repeated for subsequent sub-classes of the `RoutingTask`.

```
class LeeTask {
      point start;
      point end;
      leeParam specialLeeParameter;
public:
      point& identifyStart();
      point& identifyEnd();
      void initialise();
      leeParam getParam();
      // ....
};
```

Figure 12: Definition of class `LeeTask`

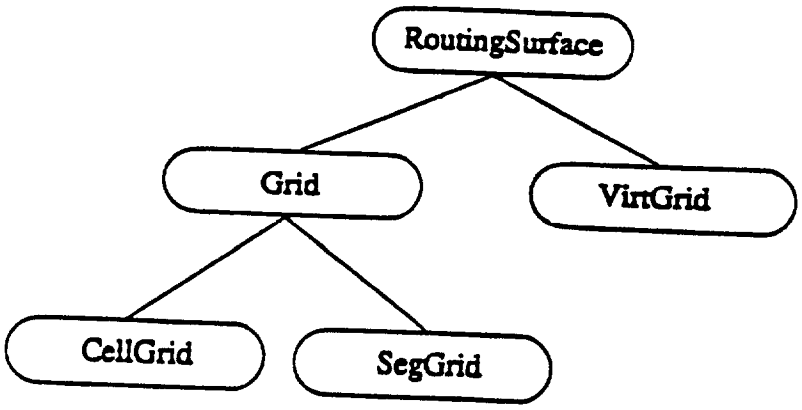


Figure 7: Initial routing area hierarchy

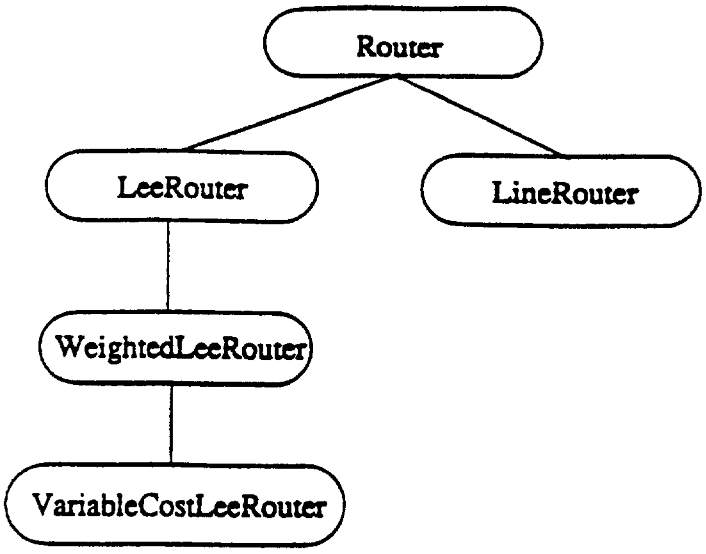


Figure 10: Initial `Router` hierarchy

Implementation

Detailed implementation and testing of each class will probably require many iterations to the preceding stage as mentioned, as deficiencies in class definitions and implementations become apparent.

9 Using The Model

The model was conceived during the development of a generic application for the VLSI routing domain [11] and has since been used to implement a number of applications, both partly and wholly within the original analysis domain. The basic functionality of two fundamental algorithms provide the skeleton upon which further related algorithms are hung by augmenting existing classes and overriding methods where necessary. The framework comprises approximately 90 classes and some 480 methods.

In all, four implementations of routing algorithms are included in our results, which were first or second order derivations of an existing algorithm. The average (physical) time spent on each stage in all four implementations is also shown in Figure 13. The figures for time spent on each stage are our best estimations. but given the nature of the design process the borders between stages are not distinct.

The initial design stages followed a rather informal approach which can now be described by our design model. Integration of additional routing algorithms meant that the domain analysis was only briefly encountered and the bulk of the design began at the abstraction stage within the component environment, as Figure 13 shows.

Figure 14 displays an approximation of the time spent on each stage of the design process for the initial design and later integrations. It show that for the initial design 68 percent of the total effort was spent on design prior to any code being written. This means that for the original design of the hierarchy, some 32 percent of the total time was expended on coding and debugging.

The decrease in coding time from 32 percent to 20 percent is due to the ability to reuse code rather than write new code.

The current system has greater capabilities than the initial design, but the total time spent on it was only 30 percent of that on the original. Reuse is difficult to measure, but if we define a *reuse factor* of an appllication A to be:

$$\frac{algorithm - specific\ code\ of\ A\ used\ by\ B}{total\ algorithm - specific\ code\ of\ A} * 100$$

where A is the original algorithm application and B is the algorithm derived from A, then the reuse factor is 76 percent [11].

We would like to be able to quantify the functionality more precisely, but even without doing that we can see that a large effort in design will produce rapid implementation of applications through a high amount of reuse. We can directly attribute the high reuse and low implementation times to the abstraction and specialization stages as most of the reuse was due to the amount of functionality encapsulated in the classes and to the mechanisms of object oriented programming.

10 Discussion

The scope of the domain considered at the outset of the design determines, to some extent, the stability of our components. Any application not taken into account when bounding the domain will probably require an identification of high-level constructs different to those already within the hierarchy and then a move to integrate those into an existing hierarchy from the specialization stage.

Since the conception of the model and the implementations described herein we have successfully implemented a hybrid routing algorithm used for commercial PCB routing [6]. The algorithm was not considered at the initial design stages which meant expanding the domain of the generic application requiring a further analysis of the domain.

This implementation required a great deal of effort in the abstraction and specialisation

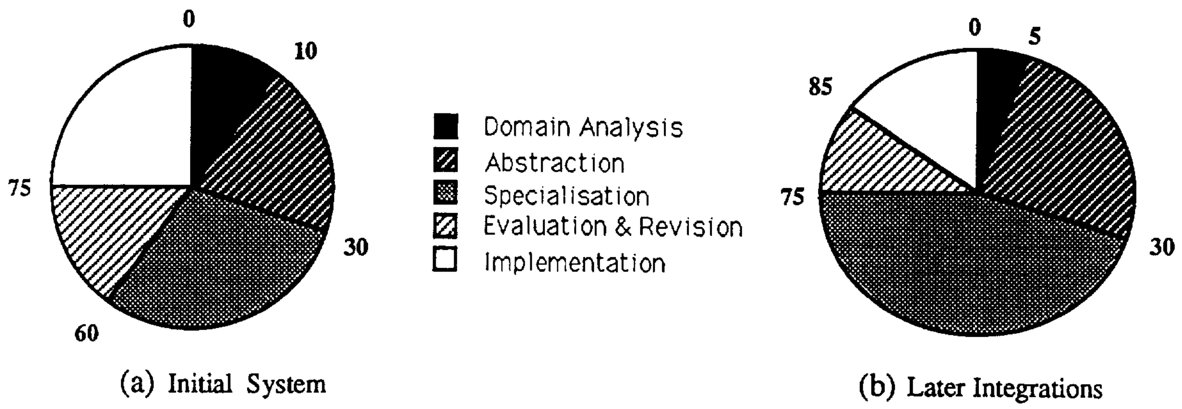


Figure 13: Pie chart depicting time spent on each stage of design

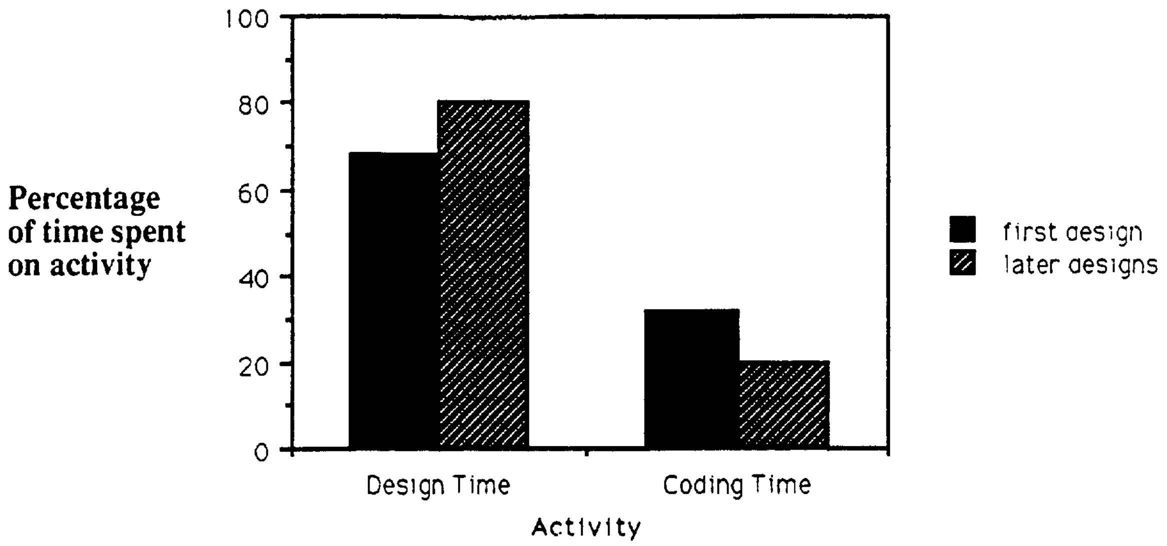


Figure 14: Percentage time spent on each stage

stages of the design where commonalities with existing classes were explored, and subsequently new classes were created — from both abstract and concrete classes. The new application introduced both new concepts, as well as new routing solutions, into the domain. As such the design effort was not a simple one and required many iterations and re-evaluations of existing structures. The iterative-design approach has, we feel, been validated to a large extent by the ease with which the new application was integrated into an existing hierarchy.

Our findings indicate a high proportion of reuse made possible by a set of high-level classes reflecting the functionality of the domain. The design of new classes has highlighted several potential drawbacks in the previous class hierarchy requiring several attempts at reorganising its structure; this has allowed the framework to evolve into a more powerful structure than before. Details of the evolution of classes and other points relating to object-oriented development and reuse will be reported at a later date.

We have only presented an overview of the iterative-design model here, laying emphasis on the iterative and evolutionary nature of software systems, by incorporating such features into its process. The model concentrates on the set of problems within the domain rather than one problem itself providing a more reusable set of classes. Our experiences have shown that a high proportion of time spent on detailed design, namely the abstraction, specialisation and evaluation and revision cycle, is rewarded by high reuse of code and low implementation times.

Acknowledgement

The authors would like to thank the anonymous reviewers of the various versions of this paper, who provided useful comment and constructive criticism. The first author was supported by a Science and Engineering Research Council postgraduate award.

References

[1] ANDERSON, D. B., AND GOSSAIN, S. Hierarchy Evolution and the Software Lifecycle. In *Technology of Object-Oriented Languages and Systems 1990 Proceedings* (Paris, France, June 1990).

[2] BECK, K. Experiences with Reusability - Panel Session. *Proceedings of the Third ACM Conference on Object-Oriented Programming Systems Languages and Applications, SIGPLAN Notices 23(11)* (November 1988), 372 -376.

[3] BOOCH, G. Object-Oriented Development. *IEEE Transactions on Software Engineering SE-12*, 2 (February 1986), 211-221.

[4] BOOCH, G. *Object-Oriented Design.* Benjamin/Cummings, Menlo Park, California, 1990.

[5] BOOSE, J. *Expertise Transfer For Expert Systems Design.* Elsevier Science Publishers, 1986.

[6] DION, J. Fast Printed Circuit Board Routing. Tech. rep., DEC - Western Research Laboratory, Palo ALto, CA, 1988. Research Report 88/1.

[7] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and It's Implementation.* Addison-Wesley, 1983.

[8] GORLEN, K. An Object Oriented Class Library For C++ Programs. *Software - Practice and Experience 17*, 12 (December 1987), 899–922.

[9] GOSSAIN, S. *Object-Oriented Development and Reuse.* PhD thesis, University of Essex, UK, June 1990.

[10] GOSSAIN, S., AND ANDERSON, D. B. Designing a Class Hierarchy for Domain Representation and Reusability. In *Technology of Object-Oriented Languages and Systems 1989 Proceedings* (Paris, France, November 1989), pp. 201–210.

[11] GOSSAIN, S., AND ANDERSON, D. B. RApp: A Generic Routing Application in C++. In *Proceedings of the EUUG Spring 90 Conference* (Munich, West Germany, April 1990), pp. 43–52.

[12] GOSSAIN, S., AND ANDERSON, D. B. Towards The Reuse of Design Artefacts In Object-Oriented Software Development. *Working Paper* (January 1990).

[13] JOHNSON, R. The Importance Of Being Abstract. Tech. rep., University Of Illinois at Urbana Champaign, Urbana, IL, May 1989.

[14] JOHNSON, R., AND FOOTE, B. Designing Reusable Classes. *Journal Of Object Oriented Programming* (June/July 1988), 22–35.

[15] MCCAIN, R. A Software Development Methodology for Reusable Components. In *Proceedings of the 1985 Hawaii International Conference on Systems Science* (January 1985).

[16] MITTERMEIR, R. Object Oriented Software Design. In *Software Engineering Environments - Proceedings of the International Workshop On Software Engineering Environments* (1986), China Academic Publishers.

[17] PRIETO-DIAZ, R. Domain Analysis for Reusability. In *Proceedings of IEEE COMPSAC 87* (1987), pp. 23–29.