The Mysterious Case Of The Missing Reusable Class Libraries

Tim Menzies, Julian Edwards, Kekwee Ng

School of Information Systems (Edwards)
School of Computer Science & Engineering (Menzies & Ng)
University of New South Wales
P.O. Box 1, Kensington, NSW, Australia, 2033
juliane@cumulus.csd.unsw.oz.au; {timm|kekwee}@cs.unsw.oz.au

ABSTRACT: Where are the reusable class libraries as promised by the literature on Object-Oriented (OO)-methods? We argue that within corporate data processing groups, resource pressures prevent the development of reusable class libraries based on an optional, additional, generalisation of some specific OO application. We observe that the development of such general libraries need not wait for the completion of the specific application, if the application is regarded as an example of the type of processing required for the domain. The products of any particular application development could be (i) the application itself; and (ii) more importantly, a set of general tools, specifically class libraries, for building similar applications. In terms of a class library, we believe that reusable class libraries are not simply specialised/ generalised application classes. That is to say generalised reusable classes do not simply evolve from vertical refinement. More typically, they involve horizontal expansion of the application's domain via client-server/association relationships. Examples of class libraries created by horizontal extension of the initial development process are given. Seven factors are described that promote the construction of class libraries during development. Object-oriented analysis (OOA) and design (OOD) methodologies are assessed for their ability to facilitate the development of the horizontal classes. Most of the current methods do not support the development of horizontal class structures and hence a contributing factor to the lack of reusable components may be our current generation of OOA and OOD methodologies.

KEYWORDS: Analysis, design, reuse.

1. INTRODUCTION

One of the frequently cited benefits of the OO approach is the development of reusable code libraries [Meyer 88,90]. Such libraries, it is said, reduce the costs of future development since developers can rapidly take advantage of previous work. A good library generalises a recognised domain and involves generalising what were previously application-specific classes. That is, once a class library, C_0, has been created to implement some product, P_1 to meet some requirements, R, then C_0, is generalised via a secondary analysis, G to produce a generic class library C_1. Explicit recognition of G has been incorporated into a number of proposed OO software development life cycle models (e.g. the fountain model of [Henderson-Sellers 90] & [Edwards 92] and the cluster model of [Meyer 90]). C_1 then becomes the kernel of a range of P_1-like products (shown below as P_2 P_3.P_N in Figure 1).

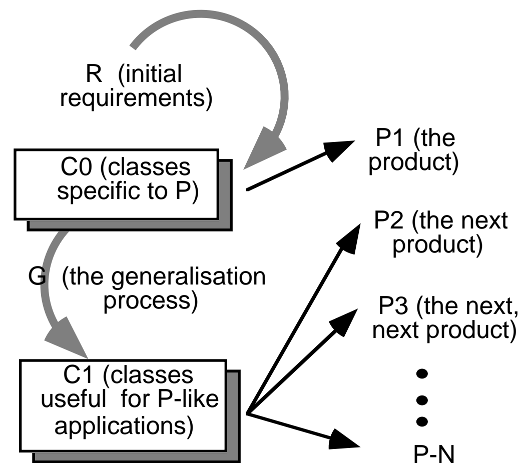

Figure 1

For example, a pull-down menu-package (C_0) developed for a word-processing application (P_1) could be used later for a spreadsheet (P_2,), database(P_3,), and communications package (P_4) but only after C_0 has been generalised via G to create C_1.

Mysteriously, such reusable libraries are largely missing from the public domain and are often restricted to either user-interface or collection class libraries e.g. *NIHCL* [Gorlen 91] and *InterViews* [Linton 89]. Even in the restricted domains of specific corporations, reusable class libraries are the rare exception rather than the rule. We suggest that one reason why this is so is that it is a mistake to make G simply an optional, additional, future project. Given the realities of the corporate environment, it would appear that companies are reluctant to allocate resources to the optimisation of an existing, working, application. Imagine, for example, the head of the DP department asking the business user for further funding in order to build a reusable class library based on their recent release. Given the current short-term economic view of corporate cost centres, generalising class libraries is not a high priority. It is unlikely that the user would allocate the necessary funds.

Instead it is important to make the development (G), of the generalised classes (C_1), part of the development cycle of the first product, (P_1). Rather than delivering P_1 using C_0, we argue for using C_0 as an analysis/design tool for discovering C_1. The first product, (P_1) can be then delivered using the generalised classes (C_1). The

corporation then has two products: an executable version of requirements, (R) as well as a library of reusable components C_1 which can be immediately used to develop P_1-like applications (see Figure 2). In this way the "reuse mindset" pervades the entire development process.

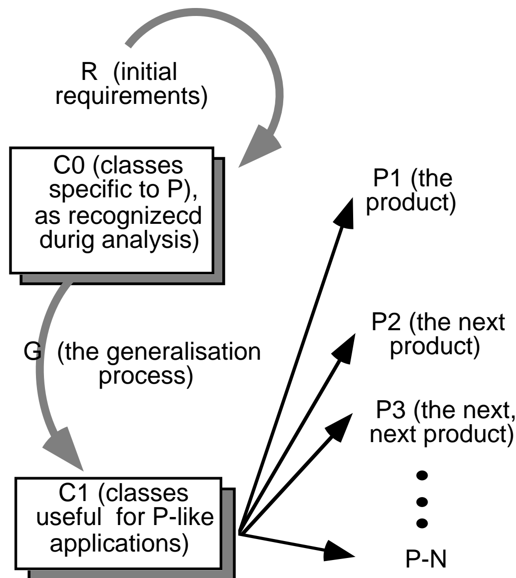

Figure 2

The rest of this paper is a discussion of the G process. Is it possible that G can be included as part of the original application development? Initially, this may seem somewhat implausible since we appear to be arguing for the generalisation of some product prior to having experience necessary for that generalisation. However, let us distinguish between the user's evolution of their understanding of the domain (as evident in updates to requirements (R)) and the software engineer's evolution of their understanding of the support utilities required to support R. It remains an open question as to whether a software engineer can preempt the user's evolution of their domain. Nevertheless, our analysis of known C_1 libraries indicates that (i) *within* any application there exist sub-systems that support the requirements (R), but which are not immediately obvious from R; and (ii) such sub-systems can be found during the initial development of P_1. We demonstrate this via examples in section two.

These examples suggest to us that software engineers can educate themselves to investigate R, develop C_0, and then generalise to C_1 prior to the release of P_1. However, we acknowledge that organisational factors can encourage or discourage G. These factors are discussed in section three. A key observation we make is that the relation of P_1 to C_1 is *not* simply the specialisation/ generalisation relationship. Section four discusses the implications of this on conventional OOA/OOD methodologies. We find that many OOA and OOD techniques lack support for the development of the horizontal links which we have found necessary for G.

2. EXAMPLES OF G-C_1

This section gives examples of output of G; i.e. the process of generalising some design for R into a reusable class library, C_1 .

2.1. SMARTASK

SMARTASK was a prototype system developed to support the mythical Mongolian Life Insurance Company's (MLI) customer service division [Menzies 90]. In this case, our R is the book of 60 tables used by the MLI's insurance agents as they roam the plains selling insurance to yak herders. For example, the following table says that a 20 year old yak herder with an old wooden house valued at 70,000 would have an annual premium of 191.25 (i.e. an figure obtained by interpolating between two known values of 120 and 130 multiplied a factor of 1.5) as shown in Figure 3.

House value	*type = wood*		*type = brick*
	age = new	*age = old*	
40,000	110	120	115
80,000	120	130	117
120,000	150	190	140

Notes:

if	client age > 65
then	use the "Premiums for the Retired" table.
if	client age > 90
then	refer to manager
if	clients age < 21
then	premium := 1.5*premium.

Figure 3

The design of the table and its associated rules gives some procedural control over the look-up process. Queries on the table can be shuffled off to other tables, abort the look-up (see the above "refer to manager" rule) or filtered via a post-processor (e.g. adding 50% to the premium of young people). Within the table, values not explicitly mentioned in the columns can be interpolated or extrapolated. The columns of the tables are headed by a set of selection criteria that control which columns are used.

A C_0 for this system could be to build one object or procedure for each table. Each cell of the table would be implemented as part of a set of nested If-Then constructs. A C_1 would build a generic*Table* class, a table look-up system that controls the queries, a specialisation of *List* that can handle conjunctions, and a *Rule* object that stores conjunctions and associated actions if the conjunction returns true. Note that these are all reasonable candidates for domain-independent objects in a reusable class library. Note also that with the C_0 design, each new table, or changes to existing table would require the support of a programmer. Contrast this with the C_1 design. If the programmer built the generic table and same tool for instantiating an instance of that table with the appropriate rules and column values, then the programmer would not be required for subsequent modifications. As with the above case, the business user could maintain their R.

The class hierarchy for *SMARTASK* (as shown in [Menzies 89]) has one child-less class called *Table* immediately under *Object*. This is just a container class storing the name of the table and pointers to instances of *Matrix*, *Rule*, and *Conjunction*. Again, the C_1 classes are associations of the application class and not specialisations or generalisations. In a C_0 system, functionality tends to be in a few objects whereas in C_1 -based applications, functionality is dispersed amongst a larger number of classes that collaborate to implement the functionality. C_1 -based classes tend not to be specialisations or generalisations of the application object, but rather a horizontal collaboration of classes.

2.2. DESCRIBE

Describe is a question-asking system originally developed for the AMP Society [Dang 90]. The original R for this system was a copy of a 40-page paper questionnaire that was to have been given to several

thousand AMP employees and the one-line statement: "We want the computer to collect the answers". C_0, as proposed by management, was for the questionnaire to be scanned and presented on screen exactly as it was shown on the paper questionnaire. The software engineers worked with that idea for four weeks, then designed a C_1 comprising of an object library for different question types (boolean, free text entry, one-of-a-list, control[1]), a general question asker, a general question/question-help display environment, and a questionnaire mark-up language that turned ascii files into instances of the class *Question*.

The mark-up language was interesting in that it was merely syntactic-sugar for *Question* instance instantiation. For example, the mark up language could say:

```
@oneOf
@prompt Where do you work?
This question wants to know in which state
office do you spend more than 90% of your
working hours.
@options
nsw
vic
qld
nt
wa
tas
qld
@end
```

A simple pre-processor turned this into the following object code (in a *Smalltalk* syntax):

```
OneOf new
prompt: 'Where do you work?'
otherText: 'This question wants to know in
which state office do you spend more than
90% of your working hours.'
option: #nsw;
option: #vic;
option: #qld;
option: #nt;
option: #wa;
option: #tas;
option: #qld;
commit
```

The mark-up language reader was implemented as two objects: *TagReader* was a generic object that handled the details of opening files of mark-up language text, and finding each "chunk" of mark-up, isolating the method calls and various arguments. A specialisation of *TagReader* was *QuestionTagReader* which handled the specific details of questionnaire definition tags such as "options".

The mark-up language files were maintained by the business users. Over the period of the development of *Describe*, the users made extensive alterations to the fine details of the questionnaire. If *Describe* had been implemented using the original *Co* design, then these changes would have necessitated the slow re-implementation of the system by a programmer.

Under the language's root object, the application is built up from four separate hierarchies: *TagReader*, *Question*, *QuestionAsker*, and *QuestionDisplay*. Yet again, C_1 comprises horizontally associated classes. The reader may wonder why we stress non-inheritance horizontal-association. Could not the same implementation be built using multiple inheritance links? For simple applications, perhaps multiple inheritance can suffice for association (e.g. the structure chart system described above could have utilised multiple inheritance). However, above a certain level of complexity, multiple inheritance becomes a kludge that complicates a design. For example, it would have been perverse in the extreme to have created a *Describe* object that inherited from *TagReader*, *Question*, *QuestionAsker*, and *QuestionDisplay*. Such complexity is best dealt with via horizontal client-server links.

[1] A control question caused the question asker to ignore irrelevant questions and directed the user to somewhere else in the questionnaire.

2.3. KNOWLEDGE BASES

If we substitute "requirements" for "knowledge bases", then the lessons and techniques of expert systems become relevant to traditional software engineering. Expert systems theory argues for the separation of control from the *knowledge base* (KB). A KB is an expression of some expert's knowledge, uncluttered by tedious processing information. So, a rule in an expert system could be:

```
if age < 4
then infant
```

A separate *inference engine* could handle the details of searching for applicable rules, gathering the information (for e.g. *age*), fully testing the if-part of a rule, then managing the then-part appropriately (here, the fact *infant* would be added to a library of assertions).

[Bustany 88] generalises this and describes the *application language* approach. When exploring a new domain, Bustany *et. al.* develop a notation that can record *R*. Next, they build an interpreter for *R*. System development then proceeds by constructing larger and larger chunks of *R*, recorded in the interpreter's syntax. This approach has the advantage that the user can evolve their understanding of their requirements via experience with an executable version of the requirements.

The development of application languages requires a tool that can (i) process code as data and which (ii) can support arbitrary abstract data types. OO-languages can do both. For example, recall the *Describe* mark-up language. The knowledge of the system was the design of the questions and the mark-up language allowed the users to express that design in a manner that allowed them to ignore tedious implementation detail.

Lest we over-state our case, we note that one feature of application languages/ expert systems is that they usually give the programmer some search algorithm which simplifies the implementation. This features allows for the rapid development of a constraint/ re-write rule that can process all instances of a particular class or some relational join across attributes in different objects. However, this brief digression into expert systems theory demonstrates that useful support tools for a requirement may not involve generalisation-specialisation of application objects. Indeed, they may require the development of elaborate architectures that may not be explicit in the original requirements document.

2.4. STRUCTURE CHARTS

Suppose a user versed in structure chart analysis gave the software engineer a specification in terms of a top-down decision tree (see Figure 4).

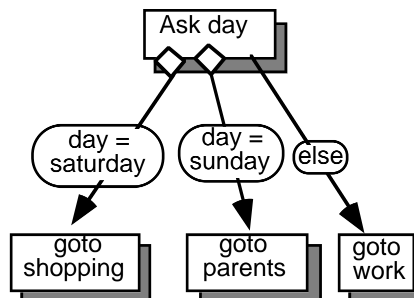

Figure 4

One design option for C_0 would be to create one class or procedure for each node and write the conditional code and calls to other objects as procedural code within each node, guarded by If-Then statements. After working with this approach on paper, a software engineer would realise that large sections of the code in different objects appear very similar. A generalisation would be to recognise that this is a network of nodes and links where nodes can be specialised into question-asking nodes (e.g. *ask day*), action nodes (e.g. *goto shopping*) and conditional nodes (e.g. *day = sunday*). The root class of our C_1 would be a *Generalised Node Object* with local variables for lists of parent and child pointers and a deferred method called *go* that handles the processing at that node. *Go* would be specialised in our other nodes as follows:

- *Question-asking nodes:* prompt the user with the question text (a local variable), wait for input, check input validity, if invalid, repeat, else place in some global store. Then call *go* recursively on each child.
- *Conditional-nodes:* if the test is true, call *go* recursively on each child.

One tacit feature of this design is that the application class is not a child-class of the generalised "node-link" class (*Generalised- NetNode*) The design process up until this point has been the traditional generalisation/ specialisation process. However, consider these node-link networks from the user's perspective. Nodes group together into networks (e.g. the *shopping* , *parents*, or *work* networks) and Node-level processing should not be confused with network-wide processing. Hence, let us create a *NetworkManager* class to manage the network-wide processing (e.g. persistent storage to a file, display in separate windows, begin processing with some special root node, authoring details, etc). Each user's application is now a named instance of *NetworkManager* which stores (amongst other things) a pointer to a special root node of every network. So, when the user creates an application, the hierarchy looks like this (see Figure 5):

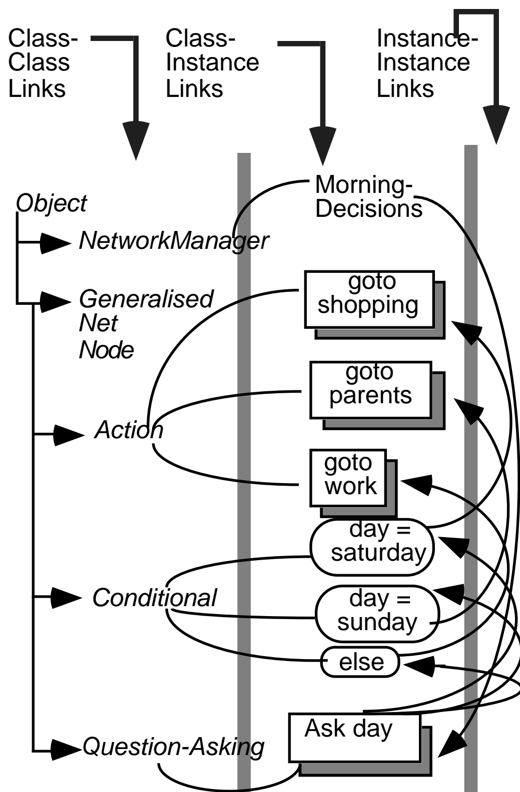

Figure 5

Our example served to introduce the concept of a node-link network that underlies many designs. [Menzies 91] argues that a standard library for any OO language should be a node-link network editor. [Bellcore 91] also argues for a node-link library arguing that lessons learnt from semantic modelling should not be forgotten, namely that relationships should be kept explicit.

2.5. MODEL-VIEW-CONTROLLER (MVC)

A famous G (although not known as such) is the model-view-controller, originally developed in the Smalltalk community [Goldberg 84]. MVC implements the concept of dialogue independence : i.e. a clear separation of a program's computational component from its interface. Such a separation permits the separate subsequent modification of a program's interface or functionality without having to re-write the entire system [Hartson 89].

OO interface toolkits can directly implement dialogue independence by assigning separate objects to a program's model and its view. Certain OO interface toolkits extend dialogue independence even further than a two-way split. Smalltalk assigns a model, view, and a controller object to each pane of each window. The controller object processes user keystrokes and mouse movements. Objective-C's interface kit adds a fourth object called a transparent controller that manages the background pane of a window (which the user sees as the top-bar of the window). See [Urlocker 89] and [Knolle 89] for more details.

MVC demonstrates again that $G\text{-}C_1$ does not solely involve specialisations or generalisations of application classes. Digitalk's Smalltalk/V® class hierarchy stores the View and Controller hierarchies directly under *Object*. Applications that use MVC exist in parallel hierarchies (e.g. *ClassHierarchyBrowser*, *ClassBrowser*, *File- Browser*). That is, G tends to generate association-links between application classes and C_1 classes rather than inheritance links[2].

We note that MVC is such a novel generalisation of an application that it would probably not be successfully generated from a single application[3]. Our other examples $G\text{-}C_1$ are at least an order-of-magnitude easier to produce that the sophistication of the MVC.

3. OPTIMISING G

What do our examples tell us about the process of $G\text{-}C_1$? We have argued that G is a process of building an executable version of R via the creation of classes horizontally associated with the application class(es). The essential change required to the mind-set of the software engineer is to regard P_1 as *an example* of the types of processing that are required for this domain. The P_1 processing should be described at a level *above* the immediate details of the requirements document. G is the process of recognising this meta-level description and C_1 is the implementation of it.

This view leads to an architecture comprising of lots of general support tools and a (smaller) section detailing how those support tools are used to realise R. When designing C_1 , we have said that R should be viewed as a

---

[2] It is possible to simulate associations using multiple inheritance networks but this is strongly discouraged as it leads to complicated networks. Further, [Menzies 91] cautions against the use of multiple inheritance networks in languages that support over-riding inherited properties (e.g. most commercial OO languages) since these over-rides leads to very confusing semantics.

[3] In fact, MVC is probably an example of C-GN, where N > 10.

evolving structure; i.e. the high-level control of the support tools may change. We should look for *R* support tools which would remain useful, despite changes to *R*. The tables of *SMARTASK* store business knowledge that could change with time. However, the generalised *Matrix* class underlying *R* would be useful regardless of how the table values change.

We expect *R* to evolve in two ways: (i) when P_n is evolved into P_{n+1} or (ii) during the development of P_n, the users understanding of their requirements evolve leading to a change in the current requirements. We have given examples where user-level tools designed to permit the business user to modify *R* have proven useful. That is, $G\text{-}C_1$ leads to a style of interactive refinement of *R* that has parallels in the field of knowledge engineering. This observation leads us to suggest that one useful strategy for *G* would be to design an application language for the business users which abstracts away the tedious control details.

There are certain organisational factors will encourage/ discourage the $G\text{-}C_1$ strategies. We have identified seven below:

1. *Cost*: *G* is a long-term strategy. In the short-term, it adds to the project cost. In the long-term, *G* pays for itself in decreased product development time. However, if the software is developed within an environment that is overly-concerned with cost-cutting, then the overheads of *G* would not be acceptable.
2. *Time: G* takes some time. Versions of C_0 have to be re-worked. If time is of the essence, then *G* is not appropriate. However, repeating the above remarks, while *G* costs more in the short-term (in terms of time and money), it can prove cheaper in the longer-term.
3. *Size of project:* The larger the project, the harder it is to get an overview of the system. Such broad overviews are very useful, if not necessary for *G*.
4. *Size of team:* For *G* to be effective, the designers need to be fully conversant with all the processing that C_1 has to manage. This is easier in small teams where the team members have regular contact.
5. *Experience:* One strategy for *G* is to generalise C_0 using experience gained in other applications. For example, an experienced software engineer might identify the similarities between current and past projects.
6. *Quality of team:* A separate factor to team experience is team quality. All the experience in the world will not convert a bad designer in to a good designer and identifying. *G* requires a certain creative flair which we would not expect in bad designers.
7. *Language*: $G\text{-}C_1$ is a evolutionary process. Languages that support incremental compilation encourage *G*.

In summary, we could say that $G\text{-}C_1$ is more likely to occur in small, skilled, experienced teams working with incremental development tools without serious time and cost constraints being a dominant factor. $G\text{-}C_1$ is a strategy that can be applied to many development projects to encourage the development of reusable classes throughout the development process. This approach can be synthesised with the later *G* phase to give a $G_1\text{-}C_1\text{-}G_2$ ... type cycle. The latter G_2 process is made considerably easier by the work on generalisation (G_1). We therefore argue that *G* should pervade the whole OO life-cycle.

4. G & EXISTING OO-METHODOLOGIES

Another serious contributing factor to the success or failure of *G* is the extent to which the OOA/OOD methodologies used by the team encourage horizontal association links as opposed to vertical inheritance links.. This section reviews various OOA/OOD approaches on their ability to use non-inheritance concepts.

One criticism that can be levelled at most currently available OOA/OOD methodologies is their lack of concern with reuse. Most of the methodologies concentrate on a "greenfield" approach to constructing object-models [Arnold 91]. That is to say they do not include techniques and heuristics for examining and integrating class libraries into the development. Those methodologies that do include a discussion of reuse tend to concentrate on development *for* reuse [Arnold 91] rather than development *with* reuse. The implication is that G is a process that occurs after the development of the object-model by generalising classes, although this process is not yet formalised or well defined for any OO methodology. Furthermore, most OOA/OOD methodologies concentrate on the notions of inheritance rather than the horizontal links of association and client-server. Most methodologies, therefore, forget the lessons learned from semantic modelling and the explicit relationship paradigm [Bellcore 91]. These horizontal linkages, it is argued, are important not only for logical modelling [Rumbaugh 91] but also for reusablility and loose coupling of classes [Kilian 91].

In this paper we have argued that generalising classes after the development of *P1* by generalisation/specialisation alone is not sufficient for the development of large-scale reusable class libraries. The implication of this argument is that current OOA/OOD methodologies are not able to lead to the development of the class libraries sought by the OO community particularly domain specific libraries.

In pursuing our arguments for software developers to consider the need for horizontal linkages, we are not stating that the usage of inheritance is not an important issue in software reuse. In fact, we favour a two-prong approach in order for any environment to develop a robust suite of reusable classes i.e. to use horizontal linkages for logical modelling and to use vertical refinement for generalising/ specialising the classes. Indeed this dual-pronged approach should be sealed together with the application of formal techniques. Further, despite existing methodologies' affinity for suggesting inheritance as a suitable mechanism for reuse, we find that here too the notion seems to be rather lacking in rigour. The laws of good software development tells us that it must be a systematic and well-managed process and it is here that the incorporation of formal methods can help. This is evident by the fact that it assists in helping the developers produce coherent object models that are capable of being validated against the requirements of the domain. This need is urgent especially in the area of developing safety critical applications. The central theme is for one to combine both horizontal and vertical approaches under a formal framework (see [Wing 90] and [Gries 91] for further arguments supporting the need for formalisms in software development). This technique has suggested that our everyday usage of existing OO methodologies can be further refined and made more rigorous.

Overall, the level of reusability is improved because of the precision and completeness our approach encourages. The pragmatic goal of our paper is to suggest a means of achieving wider reuse and this we have done by integrating the vertical and horizontal linkages allowing a more complete semantic description of the problem domain. We believe that only by combining these two types of linkages can extensive reusable class libraries be developed.

5. CONCLUSIONS

We have proposed a solution to the mysterious case of the missing reusable class libraries. The libraries are missing because we have not structured ourselves to create them. Our project plans defer object generalisation till *after* the

delivery of the first product, when resource limitations may result in the perpetual postponement of the generalisation process. Our software engineers do not "think general" when they build our systems. Building the reusable libraries, then, is always "someone else's problem". Further, our current OOA and OOD tools may not support the development of reusable components since they (mostly) do not support the discovery and specification of horizontal associations between classes and therefore, no reuse.

From this analysis, we argue that one cannot expect to find reusable class libraries unless we re-structure our endeavours. Generalisation, we have argued, should not simply be regarded as an optional add-on, but should pervade the reuse "mind-set" from day-one of any project. We should deliver version one of the executable system of the users' requirements via some generic class libraries. That is, the output of the development of any product P should be (i) an executable version of the user's requirements and (ii) a reusable class library which can support the development of P-like systems. In our view, reusable class libraries are the responsibility of every programmer/ designer/ analyst throughout the whole life-cycle of every application. We should reward our software engineers for performing G. (e.g. index pay raises to the number of C_1 classes engineered by an individual).

Also, we would argue that considerable further research into OO methodologies is required for the development of class libraries to become a reality.

Our proposal was defended by an analysis of G-C_1 systems. The generalisations used in those systems were developed concurrently with release one of the application. With the exception of the special case of MVC, we believe that competent software engineers could develop these C_1 classes during the development of release one with further refinement made easier after P_1.

6. REFERENCES

[Arnold 91] Arnold P., Bodoff S., Coleman D., Gilchrist H., & Haynes F., *An Evaluation of Five Object-Oriented Development Methods,* **HP Technical Report,** HPL-91-52 June 1991.

[Bellcore 91] *Information Modelling Concepts and Guidelines,* Version 1, SR-OPT-001826, 1991.

[Bustany 88] Bustany A. & Skingle B. *Knowledge-based Development via Application Languages* in **Proceedings of the Fourth Australian Conference on Applications of Expert Systems,** May 11-13, 1988, pp. 277-302.

[Dang 90] Dang T. *Describe: A case study in Object-Oriented Design and Programming* **Workshop Proceedings of AI '90,** Perth, Australia.

[Edwards 92] Edwards J. *Development of An Object-Oriented Methodology and Its Application to a Water Resources Problem,* **PhD Thesis**, School of Information Systems, University of New South Wales, 1992 (forthcoming).

[Goldberg 84] Goldberg A. & Robson D. **Smalltalk-80: The Language and its Implementation** Addison-Wesley, Reading MA, 1984.

[Gorlen 91] Gorlen K.E., Orlow S.M. & Plexico P.S. **Data Abstractions and Object-Oriented Programming in C++,** John Wiley & Sons, 1991.

[Gries91] Gries D. *Teaching Calculation and Discrimination: A More Effective Curriculum* in **Communications of ACM**, Vol 34, No. 3, March 1991, pp. 44-55.

[Hartson 89] Hartson H.R. & Hicks D. *Human-Computer Interface Development: Concepts and Systems for its Management*, in **ACM Computing Surveys**, Vol 21, No. 1, March 1989, pp. 5-92.

[Henderson-Sellers 90] Henderson-Sellors B. & Edwards J. *The Object-Oriented Systems Development Life-Cycle*, **Communications of the ACM** Vol. 33 No. 9, September 1990, pp142-159

[Kilian 91] Kilian M., *A Note on Type-Composition and Reusability*, **OOPS Messenger, SigPlan** Vol 2 No. 3, 1991, pp24-32.

[Knolle 89] Knolle N.T. *Why Object-Oriented User Interfaces are Better* in **Journal of Object-Oriented Programming,** Nov/Dec, 1989, pp. 63-67.

[Linton 89] Linton M.A., Vlissides J.M. & Calder P.R. *Composing User Interfaces with Interviews* in **IEEE Computer** February 1989, pp. 8-22.

[Menzies 90] Menzies T.J. *Is-a Object Part-of Knowledge Representation?*, **Proceedings of AI'90**, Perth, Australia, November, 1990.

[Menzies 91] Menzies T.J. *IS-A Object PART-OF Knowledge Representation (Part Two)* in Proceedings of **TOOLS 4,** The fourth International Tools Pacific Conference, Sydney Australia, December 1991.

[Meyer 88] Meyer B. **Object-Oriented Software Construction** Prentice Hall, 1999.

[Meyer 90] Meyer B. *Lessons from The Design of The Eiffel Libraries* in **Communications of ACM,** Vol 33, No. 9, September 1990, pp. 69-88.

[Rumbaugh 91] Rumbaugh J., Blaha M., Premerlani W., Eddy F. & Lorensen W. **Object-Oriented Modelling and Design**, Prentice-Hall, 1991.

[Urlocker 89] Urlocker Z. *Abstracting the User Interface* in **Journal of Object-Oriented Programming,** Nov/Dec 1989, pp. 68-74.

[Wing 90] Wing J.M. *A Specifiers Introduction to Formal Methods*, in **Computer**, September 1990, pp. 8-24.