The Objective evidence
A real-life comparison of Procedural and Object-Oriented Programming
IISL Innovative Solutions Project

Aug 1991

Mark Humphrys
humphrys@dubvm1.vnet.ibm.com

IBM Ireland Information Services Ltd (IISL)
Knockmaun Ho
Lr Mount St
Dublin 2
Ireland.
tel +353 1 603744
fax +353 1 614246

Preface

The Object-Oriented paradigm is now poised between its academic origins and what many believe is a widespread commercial future. Promoted as the successor to the now well-established Procedural (or Functional) software methodology, it attempts to address the latter's most serious shortcomings.

In theory, Object-Oriented (OO) programming is well-thought out, powerful and elegant. In practice however, there is a lack of practical experience with OO, and a lack of hard evidence to show that it really does produce results. Until that evidence is forthcoming, management will understandably be reluctant to commit to the new technology.

I hope that this work will help to provide some of that evidence.

Contents

Preface . . . iii

Chapter 1. Disclaimer . . . 1

Chapter 2. About this document . . . 3
Audience . . . 3
Conventions . . . 3
Acknowledgements . . . 3

Chapter 3. Management Summary and Conclusions . . . 5
Object-Oriented Programming - some background . . . 5
Introduction to the IISL Innovative Solutions Project . . . 7
Findings . . . 9
Terms of Reference . . . 11
Conclusions . . . 12

Chapter 4. An Introduction to Object-Oriented Programming . . . 13
The Procedural paradigm . . . 13
The Object-Oriented Paradigm . . . 15
How does Object-Oriented programming improve on the Procedural approach? 18
Philosophy . . . 18
Myths - user interfaces . . . 18
Myths - my product is 'object-oriented' . . . 19
Abstract classes . . . 19
Summary . . . 20
Some Questions . . . 21

Chapter 5. The IISL Innovative Solutions project . . . 23
Analysis of ADAM-Procedural . . . 23
The Form class . . . 24
The FormField class . . . 28
The FormDlg class . . . 30
The File class . . . 35
Implementation Decisions . . . 36
More Questions . . . 40
ADAM-OO . . . 41

Chapter 6. Cost Comparison Statistics - OO versus Procedural . . . 43
Caution . . . 43
Raw Statistics . . . 43
Program Complexity . . . 46
Reuse . . . 48
Extendibility - specific scenarios . . . 51
Conclusion of Statistics . . . 52

Chapter 7. Deliverables . . . 55
The Report . . . 55
The Forms class library . . . 55
Further work . . . 56
Migrating to OO . . . 60

The OO Development Site 64
6-Sigma 65

Chapter 8. Summary of Technical Work 67

Appendix A. References 69

Appendix B. C and C++ Code Fragments 71
Procedural v OO 71
ADAM-Procedural v the Forms class library and ADAM-OO 72
New World Demo v the Forms class library 80

Appendix C. Layout of C++ files 83

Glossary 85

Chapter 1. Disclaimer

As flagged by disclaimers below, this document contains in *some* places opinions, and commentary on the results of this project.

Such opinions belong to the author alone, and in no way are to be taken as representing the views of IBM Corporation or of any of its subsidiaries.

All my employers did was allow me the latitude to say these things here. They do not necessarily agree with them.

Chapter 2. About this document

Audience

This document is intended for a wide audience, including management, C programmers, non-C programmers and OO programmers.

The introduction to the project, summary of main findings, and conclusions are contained in the non-technical ***Chapter 3, "Management Summary and Conclusions" on page 5*** . This should be read first by all readers.

The main body of the document is, however, somewhat technical, and is aimed only at the reader with some knowledge of contemporary programming practices. Having said that, I have tried to keep it as general and readable as possible. The uniqueness of this report comes not from my explanations of OO, but from the figures in ***Chapter 6, Cost Comparison Statistics - OO versus Procedural*** .

Conventions

- Non-C people should find most of the references to C terminology explained in context and/or flagged with warnings.
- Non-OO people should note that the convention

  ```
  thing.DoSomething()
  ```

  means 'send the message *DoSomething* to the object *thing*' (or in C++ terms, 'call the object *thing*'s internal *DoSomething* function').

  Also, the convention

  ```
  // this is a comment
  ```

  denotes a comment.
- The main document contains only simplified pseudo-code fragments. Real C and C++ code fragments are confined to the Appendices.
- A list of possibly unfamiliar words is to be found in the Glossary.
- The words 'he,his,him' in this document can be taken to mean 'he/she, his/her, him/her' etc. No offence intended. These conventions are purely for readability.
- The technical body of the document should be read from *front to back*. Cross-references are only provided for a 'second pass'.

Acknowledgements

Thanks to John Chambers, artist and cartoonist, New York City, for the cover of the bound report; and also for the animated wing bitmaps in the Direct Manipulation demo.

Thanks to Glockenspiel Ltd, Dublin, for permission to distribute their run-time library with sample code internally within IBM.

Chapter 3. Management Summary and Conclusions

Object-Oriented Programming - some background

Object-Oriented Programming must be the most widely discussed topic in the software industry today, and yet it is much misunderstood and misrepresented. In these few paragraphs, I wish not to explain what it is, but rather to simply place it in some *context* with other contemporary methodologies.

* * * * * * * *

Computer Science can be approached from two very different directions - from a human-oriented viewpoint, or from a machine-oriented one.

On the one hand, the existence of machines which can store and manipulate complex data, and which can be programmed to execute intricate algorithms, has prompted many to impose human-like qualities on them. Artificial 'Intelligence', natural language systems and Expert Systems all seek to impose properties of the human brain on these machines, and ask them to solve problems, make judgements, read and write. So far these disciplines have produced futuristic applications, but no signs of actual machine intelligence (such, indeed, may not be in their brief). (Pen90)

Far away, across a seemingly unbridgeable gap, are the machine-oriented disciplines. These recognise the nature of what we are dealing with - a dumb mechanical device, with a set of maybe fifty instructions, no intelligence, no learning ability, no self-awareness or emotion, and no comprehension whatsoever.

This is the view of the machine that is ultimately useful to contemporary industry, and this is the view that drives hardware design, formal (mathematical) methods, and commercial programming.

New ideas come, and sometimes go, but this is the bedrock on which all computer science must build. We all want more abstraction. We all want our machines to have intuition and learning ability. But these things must come slowly, because they are not yet understood. Even in humans.

Out there in the slow, evolutionary proving ground of industry, the 'species' that is thriving is Procedural programming. It is intuitively tied into the nature of the machines, yet provides high-level data abstraction, and sophisticated division of algorithm into subroutines. It works - the ultimate criterion.

Object-Oriented programming is *not* something brand new. It is a natural progression out of what went before.

It is *not* some unreasonable set of ideas imposed on the machines from above. Rather, it has grown out of the 'dirty' world of contemporary programming. It is a way to give proper support to the good design methods being implemented *now* by

industry in the Procedural paradigm. It is one more step on the road to modelling complete environments in which programs 'live', rather than writing once-off, 'blind' applications.

It is commercial. It is practical. It works. It is a measured evolutionary step forward. It is the 'son of' Procedural programming.

Introduction to the IISL Innovative Solutions Project

This project is part of the worldwide IBM Innovative Solutions program, launched to encourage, non-research IBM sites to try out new ideas and processes without business risk.

IBM Ireland Information Services (IISL) has long had an interest in Object-Oriented programming - in 1990 we developed the SAA Delivery Manager, an IBM Program Product written in a proprietary Object-Oriented environment.

Despite the growing interest surrounding OO, there are few if any 'test cases' - comparisons of OO techniques with what Procedural techniques could have accomplished in exactly the same situation. Many new applications have established OO as workable, but the question still remains - how would this have been done using the old ways?

The objective of this project was to help fill this gap, by making a head-to-head comparison of a Procedural application and an OO version of exactly the same program.

Objective

To perform a practical and business comparison of Object-Oriented versus Procedural methodologies.

Approach

Redesign and recode an existing Procedural application. Redesign it with OO design and recode it in an OO language, to perform the same function. Make out a full cost comparison of the two applications.

The results of this comparison form the unique feature of this document, and are contained in ***Chapter 6, Cost Comparison Statistics - OO versus Procedural*** .

As the developer involved here, I was *not* a member of the OO Delivery Manager group. Rather, I began this project from a totally neutral viewpoint, having no real opinions about the usefulness or otherwise of Object-Oriented programming. I was steeped in the Procedural tradition, and had not been impressed with any alternatives offered to this old and trusted methodology.

During the course of the project, I entered fully into the OO paradigm, coding in one of the new languages (C++), building generic 'classes' and designing 'class hierarchies'. I was given free reign to explore a form of programming quite different to anything I had seen before; and came away impressed, and converted.

Project

The project selected for re-implementation was typical of the applications developed in the MCU PWS group here in IISL. A project called ADAM, it is a PWS-based, user friendly front end to the WTAAS host system. It consists of order entry on the PWS, which sends the data down to the host screens in the background. It is written in C for OS/2 PM, using a standard Procedural design.

The data capture section was rewritten in C++, also for OS/2 PM, but using some radically different Object-Oriented internals.

The host communications were not rewritten *(not because they were unsuitable for OO, but for reasons of time. There is no reason to believe that similar results would not have been obtained for host communications)*.

As part of the Object-Oriented redesign of the project, I found myself developing a generic library for this sort of data capture (which I call 'form handling').

This may surprise those coming from a Procedural background, to whom writing anything generic is usually much harder than writing something application-specific. But one must understand that this does not hold in the Object-Oriented world, where it is often easier to build non-working classes (or 'types') expressing generic behaviour and then inherit them into application-specific, working classes. (for full explanation, see “Abstract classes” on page 19)

Also, note that this by no means conflicts with the aims of the comparison. Building generic objects and inheriting their behaviour is what one *must* do in the OO paradigm - otherwise one might as well not be doing OO at all. To take the C code and somehow 'translate' it into C++ (whatever that means) would have been a pointless exercise.

I had to fully participate in the OO paradigm in order to make this comparison meaningful. That meant building my function into objects - generic function into generic objects, application-specific function into derived objects.

Of course, allowances have been made for this generic work when making out the final comparison figures.

Findings

An existing IS Product, written to a traditional Procedural design, was re-designed and re-coded using Object-Oriented techniques.

It was found that most of the code in the old program expresses generic-type algorithm, similar to work found in other applications. It was hoped that much of this behaviour could be somehow expressed in a common, shareable library, which different applications could inherit from and adapt to their own purposes.

It was found that the old, Procedural methodology could *not* be used to extract this work into a shareable library; but the new OO methodology *was* able to do this. OO techniques enabled the encapsulation of generic behaviour into reusable, generic components, in a way that was previously near-impossible.

As a result, the new application can be stripped down to ***less than 20% of its original size***, inheriting work from a library unrelated to its specification. Such a library can be shared by other applications, thus reducing the size of each one, and the costs involved in developing and maintaining them.

The claims of OO regarding dramatic new forms of reuse and flexibility are totally substantiated by these findings. It is difficult to see how these innovations could have been implemented without Object-Oriented support.

On a technical level, it was found that OO works, and is practical, in every sense as much as current, Procedural programming. The language used, C++, is appraised as a serious development language, losing none of the power and freedom of its Procedural antecedents. It is concluded that use of an Object-Oriented *language* was an essential feature in the success of the innovative design.

	Procedural Program	Generic Library	OO program
Time taken	35 days	51 days	17 days
Lines of code	5827 LOC	3919 LOC	1 59 LOC
Size of executables	346 K	221 K	132 K
Hard-coding of fields	48		2
Global variables	245	6	5
Average function size	39 LOC	7 LOC	6 LOC
Generic/application split		91 % /	9 %

Figure 1. Summary of Statistics

Object-Oriented programming enables new methods of sharing and reuse of code and algorithm, that were difficult, if not impossible, using Procedural methods.

The expected effect of this will be the widespread development of generic software 'components' - robust objects whose behaviour can be used in different applications. For *someone's* investment in designing good generic objects (not necessarily *yours*), the cost of dozens, or even hundreds of applications using these objects will be substantially reduced.

This reuse of code is a process that began with Procedural function libraries, and is taken to a higher level with OO. Procedural function libraries have only proved useful in certain limited situations. Object-Oriented *class libraries* should provide one more degree of freedom, allowing developers to use *inheritance* to subtly customise other people's objects for their own use. (for full discussion, see Section “Migrating to OO” on page 60, Migrating to OO)

Straightforward extrapolation from these figures indicates that if a site is to spend time designing its own generic objects, then it will have a sizeable initial expenditure, ***but it will be cutting the cost of further applications by 50%***.

After 3 applications using these objects, the initial investment has been recovered, and from then on there will be a straight 50% saving per application.

It is believed that such an extrapolation is not unrealistic.

On the other hand, if a site merely *purchases* useful objects, its savings may be even greater. Because of the Object-Oriented property known as inheritance, third-party objects may be customised for use without losing their function and integrity.

Terms of Reference

The findings are impressive within the terms of reference of this particular project, but it is important to state exactly what these are.

On the OO side, a full Object-Oriented language was used, and I would argue (see “Myths - my product is 'object-oriented'” on page 19) that these findings are *not applicable* to anything other than a full OO language development. Doing OO work in a home-grown or Procedural environment may or may not be worth your while - this document cannot provide any comparison figures.

In terms of the application chosen, this was a relatively small, single-programmer project. The applicability of OO to large, multi-programmer projects awaits further study. Code management tools are likely to be the deciding factor.

This was also a PWS-based project. The applicability of OO to host programming cannot be commented upon here. At the time of writing, there is a lack of software tools to implement OO ideas on mainframes, but this does not mean that it should not be done.

Also, the Procedural application chosen, ADAM, does not represent the *highest level* of abstraction and flexibility possible in the Procedural paradigm. What it *does* represent is the kind of application that is being delivered to real customers in the early 1990s.

If you are already achieving high-level data abstraction, *controlled* dynamic binding and *binary* reuse, then ADAM-Procedural may appear primitive to you. However, if you are already doing these things, then you will need little convincing that Object-Oriented programming is the next logical step for your business. You are half-way there already.

No comparison is perfect. ADAM-Procedural was written on a production line, to a tight deadline, for real customers; whereas ADAM-OO's schedule was more relaxed, and I was free to implement creative and experimental ideas.

But at the end of the day, the important fact is that the two programs are exactly the same externally - and the dramatic analysis of their internals *must* tell us something about the performance of the OO methodology.

For the evidence suggests that it was the use of the Object-Oriented methodology that made the *real* difference in these comparison statistics, rather than any other factor. And if it made that difference here, then maybe it could make that difference in other situations.

This was one case study. These were the limitations of my brief. And this is what happened. Extrapolate what you will.

Conclusions

In an ordinary business application, not *apparently* well suited to the application of Object-Oriented techniques, OO delivered an 80% reduction in code size, vastly reduced complexity, vastly increased flexibility, and a large projected drop in maintenance costs. The price to be paid for all this is an initial investment in generic object design, which is more than justified in the long term.

This was not achieved by magic, or by programming 'wizardry'.

This was achieved because it was written with a new and powerful technology, which builds on the experience of the first four decades of programming, and which directly addresses the business needs of reuse, productivity and maintainability.

I cannot, of course, guarantee similar results for all projects. But these are statistics that simply cannot be ignored. This was a first experience with Object-Oriented programming, and the old application was stripped down to a mere template.

If this work is really representative, then 80% of the code being written today is unnecessary code, which with more advanced languages could be inherited from other work.

If this is true, and if OO can really enable this, then the Procedural paradigm has finally found its successor.

Chapter 4. An Introduction to Object-Oriented Programming

It is not my intention here to provide a full explanation of OO (for a proper reference, see (Cox87), (Meyer88)). Instead, I wish to develop my statement that OO is an evolutionary development from state-of-the-art Procedural programming, addressing the weaknesses of its predecessor with a new set of design concepts.

I wish to briefly outline current, Procedural programming, discuss its deficiencies; deficiencies which everyone lives with and takes for granted - and discuss the Object model which overcomes so many of Procedural's barriers.

Even if this analysis does not convert the reader to OO, I hope that it will at least sow seeds of doubt in his/her mind - that there may be directions in which programming will someday have to change.

The Procedural paradigm

- Data stands alone, and is separate from the diverse functions that manipulate it.
- The function is all-important - a program is a network of functions. Good design tends to separate these into modules, reflecting separate function.
- Actions in the program consist of calling specific functions to operate on specific data.
- The methodology works, and can be used to build highly complex applications.

However..

- ***Reuse is not explicitly supported***.

  Despite the Procedural revolution, despite the prettiness and elegance of abstract data types, subroutines, procedures, functions, modules and libraries, remarkably little has changed in the software industry. The rule, rather than the exception, is still to write any serious code almost entirely from scratch.

  Programmers have consistently found it too difficult and costly to use other people's work, and prefer to write things themselves, rather than expend the effort to adapt someone else's work to their own purposes.

  And so we are still writing the same old things that have been written a million times before. To quote (Cox90):

  > *'.. because programmers invent and build them all from first principles, everything in the software domain is unique and therefore unfamiliar, composed of modules and routines that have never been seen before and will never be seen again.'*

  Why? Because of the inherent nature of Procedural programming. Actions consist of calling particular functions to operate on particular data. Function and data are bound together *only by the spec. of each particular program* - and so algorithm (not code) gets duplicated across similar applications.

  In the Procedural world, 'reuse' is often a euphemism for 'cut-and-paste' of standard code blocks. 'Black box' library functions are only helpful in very limited situations.

- ***When the data changes ...***

  ... all the functions which act on that data have to change. The functions express the algorithm, but they are tightly bound to the particular type of data they use. There is no way for a new data type to link in to the old algorithm and make use of it.

- ***When the problem changes ...***

  Procedural design typically involves functional decomposition. The problem is expressed as a single main function, which calls sub-functions to solve sub-problems. The whole application is designed to implement the solution to this particular problem, and the whole application has to change when the problem changes. To quote (OOPSDeNat89a):

  > *'.. all of the advances in software engineering over the past ten years like top-down design and stepwise refinement conspire against reuse. Just look at the methodology. Take a specific problem, divide it into sub-problems, iterate until the sub-problems are trivial and then design and code modules. What do you end up with? A bunch of modules designed to fit a particular context! This is exactly the way we built real things (i.e. not software) before the industrial revolution.'*

- ***Data hiding and function hiding***

  Most applications contain data structures and functions that are only meaningful in a limited, local context - and which the programmer would normally like to 'hide' from the rest of the application. e.g. a 'Stack' should provide 'Push' and 'Pop' routines to the outside world, but should be able to protect its private 'stackpointer' data (in fact, it should be able to hide its entire implementation). The outside world should only access it in a controlled way through the services it provides.

  Likewise, all (or at least most of) the functions which manipulate a particular data structure should be gathered together in one place, to minimise the effects of change when the data structure definition has to change. This also allows thorough debugging of all uses of this 'object'.

  Good modular design tries to achieve this, but it takes a lot of effort. The next step after Procedural should be to take these good design techniques and build them into the language. (Strou88)

- ***Initialisation/cleanup***

  Any complex data structure typically requires an initialisation routine - to allocate memory, open files, etc., and a cleanup routine - to free memory, close files, etc. The Procedural programmer has to *explicitly* call these when the structure goes into/out of existence.

- ***Inheritance***

  Say you have a Box data structure, with a module of Procedural functions to process it: InitialiseBox (Box),DrawBox (Box),etc. For the next release, you want a NewBox, which is exactly the same except with a different Draw function. What do you do? Define NewBox as Box and make sure only to call NewDrawBox (NewBox)? But then how do you make sure that DrawBox (NewBox) can never be called? More importantly, what happens to your nice module?

  Yes, there are workarounds. There are ways to implement inheritance of properties in Procedural code. But they are not *supported* by the languages.

The history of programming languages is a history of new design techniques overburdening the old languages, and ultimately justifying new syntax for their proper expression. (Strou88)

The design ideas being implemented in Procedural programming have finally outgrown their parent, and call for a new syntax, and a new way of looking at what a program should be.

The Object-Oriented Paradigm

- An 'object' is data plus methods (functions) ***(Encapsulation)***.
- An object's methods (should) express all valid actions on the object.
- Actions are carried out by sending messages to objects.
- An object's methods typically express how it responds to a message, so objects can customise their own response to the same message ***(Polymorphism)***.
- The algorithm is expressed in the messages passed around, not in specific function calls or specific data types.
- A message may produce different results, depending on the object that receives it.
- Old objects may be replaced by new ones, without changing any of the system that manipulates them - just respond to the old messages.
- The *class* is the definition of an object and its behaviour (e.g. the class Integer). Objects themselves (program variables) represent instances of a class.
- Classes have 'parents', whose behaviour they inherit, and 'children', who inherit their behaviour ***(Inheritance)***.
- Generic behaviour is encapsulated in generic library classes, e.g., NaturalNumber.
- This behaviour can then be 'inherited' by application-specific classes, e.g., CustomerNumber, with parent NaturalNumber. In this case, a CustomerNumber 'is a' NaturalNumber, and so will receive all the old NaturalNumber messages. It may decide to give some customised response to them, or just allow the default response from NaturalNumber to be called.
- Objects typically have special methods for construction and destruction, which are called automatically by the language.

Note: that when old messages get delivered to new derived objects, the old Object-Oriented code is effectively calling functions that did not exist when it was compiled; and it is also manipulating a data type that did not exist when it was compiled.

This new power, controlled and properly regulated by the OO language compiler/interpreter, is what drives OO programs internally, and enables their rapid extension and modification.

Pseudo-Code Fragments - An OO 'Desktop' v A Procedural 'Desktop'

Below is a comparison of the kind of code one would write to define a desktop and the objects that sit on it.

Procedural - data definitions:

```
type Triangle
{
  angle a;
  angle b;
}

type Square
{
  integer x;
}
```

Procedural - function definitions:

```
function DrawTriangle ( Triangle ) { ..(code).. }

function DrawSquare ( Square ) { ..(code).. }
```

Procedural 'desktop':

```
function DrawObject ( Type )
{
  case ( Type ):
   Triangle : DrawTriangle ( triangle );
   Square   : DrawSquare   ( square   );
   Circle   : DrawCircle   ( circle   );
   ....
}       // draw the correct object, depending on the type
```

OO - object encapsulates data and function:

```
class Triangle
{
  angle a;
  angle b;
  Draw () { ..(code).. }
}

class Square
{
  integer x;
  Draw () { ..(code)..}
}
```

OO 'desktop':

```
DrawObject ( Object )
{
  Object.Draw()
}
      // tell the object to draw itself, not caring what it is
      // the programming system will work out which real function to call
```

If you add a new Hexagon object, all the Procedural desktop handling has to change to deal with the new function DrawHexagon. The OO handling does not. You still simply:

```
Object.Draw()
```

Oblivious as to what type of object this is. You simply do not care. (Cox87)

How does Object-Oriented programming improve on the Procedural approach?

- It introduces a 'communications' or 'messaging' layer, to express actions that should be performed, regardless of the object that receives them.
- This allows new objects to be added to and removed from the system, without changing any of the code that sends messages to them.
- This gives explicit support for extendibility.
- It provides real inheritance, rather than the cut-and-paste 'reuse' resorted to by Procedural programmers.
- Constructors/destructors handle initialisation/cleanup. Each object has *integrity* - when it comes into existence it initialises itself automatically, when it is being destroyed it cleans up automatically. An object *cannot* be used un-initialised, and cannot be left in a 'messy' state.

Philosophy

The 'philosophy' of OO is hard to pin down. OO can probably best be summarised as the natural growth of many good design ideas from the Procedural paradigm. OO beliefs include the following:

- Modern software is much too complex to specify or to build. It has to be grown. (And90)
- Software should build on stable, working objects, which can be re-used *without* modification.
- Software should be able to inherit the properties and behaviour of objects in previous work, and need only express how their behaviour is different. Nothing should be coded that has been coded before.
- Components and sub-systems should be able to change and be replaced without changing the rest of the system.
- Objects in a system should manage their own integrity, in all their interactions with the outside world. The system should not have to worry about the object's internal integrity. It should be notified by the object itself when there is a problem. The system should just *use* the object, confident that it is secure and robust.

Myths - user interfaces

OO is *not* a user interface. It is nothing to do with user interfaces *per se*. It is a new programming methodology, irrespective of application or environment.

A recent quote in 'Computerworld' shows the confusion:

> *'With traditional languages, the sequence of events is driven by the program. The user reacts to the program. With object-oriented programs, it is controlled by the user.'* (Tash90)

This is not true. This is not Object-Oriented programming. This is the contemporary user interface model of Concurrency and Amodality.

Yes, user-driven programs are easier to implement with OO programming. Yes, windowed user interfaces are easier to implement with OO. But these are merely specific *implementations* of what is a new and all-encompassing methodology for all parts of the application.

As if to prove the point, the application of OO in this project has nothing whatsoever to do with the user interface. In fact, I use the same user interface as the old program, but attach it to some new Object-Oriented internals.

Myths - my product is 'object-oriented'

Now that OO has come of age, everyone is claiming to be 'Object-Oriented'. It is in danger of becoming a meaningless badge attached to all new products by their salesmen.

'Object-Oriented' is not something that you are or you aren't. It is not as simple as that. Object-Oriented Programming is based on a number of ideas, the essential ones being ***encapsulation***, ***polymorphism*** and ***inheritance***. These are all good design ideas, which have been and still are implemented (with difficulty) in Procedural languages.

These techniques all support each other. Individually they are all good things by themselves, but do not expect major cost savings if you only have one or two of them. Yes, polymorphism is good - you may even feel free to call your product 'object-oriented' (whatever that means). But if you have no inheritance, then you cannot expect anything like the savings discussed in this report.

This particularly applies to doing Object-Oriented work in Procedural languages. This is so far removed from doing OO work in OO languages that I do not believe the findings of this report can be used to justify it. Doing OO in Procedural languages may not result in any cost savings at all.

The IISL Innovative Solutions project was implemented in a full OO language, C++, which supports encapsulation, polymorphism and single inheritance. Systems supporting anything less than *all 3* of these will not fully realise the potential of the Object-Oriented methodology, and the conclusions in this report *cannot* be considered applicable to them.

Abstract classes

With the proper support of an Object-Oriented language, polymorphism really comes into its own when you start designing objects that do not actually work - 'abstract classes'.

The abstract class is not meant to be used directly, but rather inherited from, into some class that *will* actually work. The abstract class expresses 'algorithm' that will work when derived objects provide the specific routines needed at different points in the algorithm.

It is difficult to code this type of generic behaviour in Procedural languages, because Procedural languages want to know about real function calls and real data types. In OO languages you can more easily express generic behaviour because

you can call functions that you don't know exist, and manipulate data structures that you don't know exist.

For example, one could implement a Sort routine which could sort *anything*, so long as it responded to the 'less than' message. The Sort routine does not know what data type it is sorting, but it can manipulate it all the same. See also “The FormField class” on page 28 for an example of a class that does not actually 'work'.

Hence the class hierarchy is *not necessarily* meant to be a hierarchy of useful objects. It can be used also as a repository for useful algorithms, which have no meaning until certain function is provided by your derived classes.

Summary

Procedural programming solves a *particular* problem in a particular environment. Asks the question 'How do I write this program?' (Eck89)

OO programming models the environment itself. Asks questions like 'How do I build this out of what I already have?', 'Why do I have to write this at all - didn't I write something like this before?' and 'How do I write this so I can change it completely in the next version? (Because experience tells me that's probably what's going to happen.)'

OO looks further than this particular problem. It tries to write code:

- for the next release - just change responses to messages.
- for a similar business problem - inherit the same objects.
- for a similar type of application - write generic 'applications'. An application is merely a specialised instance of this.

OO design and languages provide the first *practical* way of modelling environments for long term use, rather than merely trying to 'solve a problem' (OOPSBoaz90). The next step after *that* will be to try and enforce 'natural laws' throughout all actions in the environment (RISKSLeich90). But that, of course, is beyond OO.

To summarise: the first coding of an application should not be regarded as the final solution to a well-defined problem, but rather as the first step on a long road that will pass through many variations on the same theme. To quote (OOPSDeNat89b):

> *'I really think that what we've learned during the first 40 years of programming is that the software never stops evolving, that maintenance is the overriding cost, and that we need to focus on how to make software reusable not just for source code but for object code, and not just for the moment but over time.'*

Some Questions

But surely all this generic behaviour is a massive overhead?

You might think so.

So instead of a MyList which inherits from a thoroughly debugged IndexedList which inherits from a List which inherits from a Bag, you write your own quick MyList from scratch.

It has only half the function, adds a few hundred lines of code to the project, costs more time to debug, has to be completely scrapped in the next release, and all maybe for a few per cent performance improvement *(or maybe not, if the author of IndexedList is a better programmer than you..)*.

But of course, the Procedural programmer has no choice. He can't use anyone's IndexedList, because it won't be a list of MyObjects. So he *has* to write it himself.

At least the OO developer has the choice. IndexedList is a list of IndexedObjects. He just adds IndexedObject to the list of parents of MyObject, and off he goes. If he needs more performance, he can write his own, and start sacrificing all these nice things like reusability, flexibility, etc.

Yes, dynamic function binding *must* take longer than hard-coded function calls. If repeated thousands of times in a loop, this cost will be noticeable. There is a trade-off, the same old trade-off as between high-level and assembly languages.

As hardware evolves, higher and higher-level abstraction becomes practically possible. High-level Procedural languages are used everywhere these days, where once anything except assembly would have been unthinkable.

The issue is that modern OO languages can finally be considered for serious applications. In an I/O bound, user-driven program such as this one, there is no apparent performance impact. (See “The Executable Image” on page 45.)

Can Object-Oriented design be implemented in Procedural languages?

Yes, of course, just as Procedural design can be implemented in assembly.

Any methodology can be implemented in any language, but why code in a language that doesn't *support* your design methods? It doesn't make any sense. You will expend unnecessary effort trying to implement all the features that should come for free. And features like messaging and inheritance are indeed non-trivial.

There are two approaches to building OO systems on top of Procedural languages:

- Retain Procedural syntax - everything is still a function call with data as parameters. Pre-processors and libraries handle the OO extensions, whereas the code remains syntactically correct to the Procedural compiler.
- Introduce new syntax - the approach taken by AT&T, who felt the need for a new syntax to express these genuinely new coding methods. As such they had to build what could only be described as a compiler, which compiles the new language C++ into the still-high-level language C.

Years of work have gone into OO languages, and this is not something that should be lightly thrown away. And for any serious OO work in a non-OO environment, you *will* end up inventing your own messaging system, your own inheritance system, etc.

One wishes to write an application, not a compiler and an application. If there is any alternative at all, one should try to avoid re-inventing the wheel.

At the technical level, a Procedural language will always be worse than an OO language when it comes to implementing OO ideas.

In terms of logistics, there are a few arguments advanced for doing OO in something other than an OO language:

- To save/call/use old code - yes, clearly, this would be a problem (as indeed, would be *any* attempt to upgrade to a better language than the one you have now. Nothing unique to OO here - the 'installed base' of code can prevent a site from even doing structured programming).

  One exception would be if your old code is in C, in which case you can throw it into a C++ compiler and it will compile, same as ever. Stand-alone C functions can then be brought inside objects, modules can be rephrased as classes, and function can be sorted into a class hierarchy. There can be a gradual transition to full OO code.

  To a certain extent also, C++ will be able to call functions written in other languages, since so much work has been done on getting its subset, C, to do just that.
- If there is no choice - if there is no OO compiler available for your platform.
- To save costs of retraining - this doesn't make sense. How are you going to teach your programmers OO design without teaching them an OO language?

(See “Why not a Procedural language?” on page 36 and “Does it have to use an OO language?” on page 60.)

Chapter 5. The IISL Innovative Solutions project

Analysis of ADAM-Procedural

The Procedural version of ADAM is a front end to a host order entry system. The user types his order data into panels on the PWS. The data is error-checked, possibly stored temporarily in files on the PWS, and at some stage ADAM sends the data down to the host screens, reporting back any errors. As far as the data capture section goes, it is a very standard type of application.

As the developer, a number of ideas came to me when I considered how to apply Object-Oriented techniques to this application. My line of thought was as follows:

- this is an order entry system.
- it provides an entry screen to represent the order form, with various fields that must be filled in.
- but the actual field layout of (date, customerID, partID, quantity..) is hard-coded throughout the application (48 times in the data capture routines alone) - and this makes most of the code unusable in any other application. Is there any way we can just be editing a 'list of fields' instead of this *particular* uninteresting list of fields?
- the user will need to assemble information fast!
- it is natural to assemble information from components - CustomerInfo, PartsInfo, etc, but we don't want to hard-code component layouts either. A 'copy customer information from order histories' function would hard-code all the fields.
- as a rule, if you type in information once, anywhere, into any screen or data structure, you should never have to type it in again.

  So I would like to find a way to easily copy bits and pieces of old orders, standard configurations, etc. Note that this is true to the OO idea that everything should be an object, able to talk to any other object. If it exists, then talk to it until you recover the data you want. Unfortunately, ADAM-Procedural stores its data in fixed-size C structures and we must know the layout of these structures in order to extract the data.
- make it as quick and easy as possible for the user to throw together the information he needs, e.g., by scavenging old orders and dragging information across with the mouse (NWD90).
- but don't hard-code the field layouts.

How could I make this generic?

- There is much common algorithm in all the field handling:
  - Display text associated with field.
  - Input into field must be verified.
  - Save text in field's format.
- So why not handle some sort of generic 'Field' object, sending it 'Display', 'Verify' and 'Save' messages?
- A record should change from being a hard-coded list of fields to a flexible 'Form' - a list of any number of these 'Field' objects.
- But can a 'Form' be given the intelligence necessary to copy relevant information from a subrecord or superrecord (or intersecting record) without having to hard-code actual record layouts?

I adopted this golden rule:

> *The specific form layout information should not be hard-coded into the program.*

The actual format of the order form being edited is not important, and is liable to change with the next release. The number of fields, the type of fields - none of these things should matter. I should just be editing 'a form'.

This proved an extremely hard rule to stick to, but by doing so I managed to build a tough and powerful generic object - the 'Form'.

The Form class

To encapsulate all this behaviour in a generic object, I invented the 'Form' class.

A Form is a dynamic 'record', or in C, a dynamic 'struct'. Its data is a dynamic list of 'FormField' objects, and its methods are the methods for handling the list.

In short, a Form is a list of FormFields.

I do not regard the Form as a complete replacement of the C 'struct' - else I would be discussing the 'Struct' class library here. Rather, it is a C structure-substitute, to

be used where the speed and efficiency of a C structure is outweighed by its inflexibility.

In a situation like this, where a user manually edits a small number of Forms before dispatching them, the speed and efficiency of a fixed-size C structure is irrelevant. Of far more importance is whether or not that particular pattern of fields can be changed at will, or is hard-coded in every nook and cranny of the program.

Selected methods of the class Form:

Form.Add (Type, Name)

add a new FormField to the Form, of type 'Type' and variable name 'Name'

Form.Remove (Type, Name)

remove a FormField from the Form

(a Form can grow or shrink dynamically)

Form.Eat (Form)

copy all the data that fits from one Form into another. Each Form may have *any* number and type of fields. For any fields that match, the data is copied across.

If we use a 'structure' or 'record' (a fixed list of fields) how do w copy data from one to the other? A Procedural programmer cannot define a function to:

```
Copy ( AnyDataStructure, AnyOtherDataStructure )
```

He must define specific functions to pick out the corresponding field one-by-one and copy them using =, stringcopy etc. But this hard-codes the structure layouts into the 'Copy' function, with code such as:

```
destination.customerID = source.customerID
destination.partID     = source.partID
stringcopy ( destination.name, source.name )
 ...
```

He must also then define a separate function for each pair of structures. The trouble this causes, and the hard-coding of field layouts it encourages, is illustrated in “New World Demo v the Forms class library” on page 80.

Here in the Form class, Object-Oriented messaging allows implemention of a generic copying algorithm that does not know what data structures it is copying.

When a Form receives the message Eat (Form), it sets up in conversation with the other Form. They identify matching fields by sending the messages 'Type' and 'Name' to their fields. Matching fields are set up in conversation. The source field will receive the message 'StartAddress' to get the start address of its stream of bytes, and the destination field will receive the message 'Eat' to copy these bytes into its own format. In this way, new data types might be added as Form fields, but so long as they respond to the messages 'StartAddress' and 'Eat', they can still be manipulated by the old 'Copy' or 'Eat' command.

If some fields don't match, nothing happens. The default is to simply copy all data that can be copied. This means that large Forms can 'Eat' small component Forms, and small Forms can 'Eat' large source Forms, filtering out the information they need. For example:

```
OrderForm.Eat ( CustomerDataForm )
     // fill in the customer subcomponent of the order form
     // with some existing customer data

CustomerDataForm.Eat ( OrderForm )
     // extract the customer subcomponent from the (larger) order form
```

I do not have to write any of these functions. All the work is already done for me.

The 'Eat' message can be applied to any child or descendant of the class Form. The code for 'Eat' has no idea how many or what type of fields it is dealing with until it is actually called (at run-time). More importantly, it does not *want* to know.

One extremely nice application of this 'Eat'ing would be in the area of host front ends (such as ADAM).

One would define a Form for each host screen, expressing the information needed to be sent to that screen, and the code to copy that information into the relevant host entry fields. Such a 'filter' would simply extract the information it needed from whatever Form was thrown at it.

The host screen objects would sit in a run-time library, shareable by all applications, and one could throw *any* Form from *any* application at them - they would Eat what they need.

Inheritance from commercially available classes

The C++ compiler used came from Glockenspiel Ltd, who also supply two libraries of useful classes - 'CommonView' and 'Container'. Rather than build things myself from scratch, I attempted to inherit their work as much as possible.

The Form class is built on top of Container's Ring class.

The inheritance of behaviour can be seen in the following class hierarchy tree (see the Appendices for the actual C++ files layout).

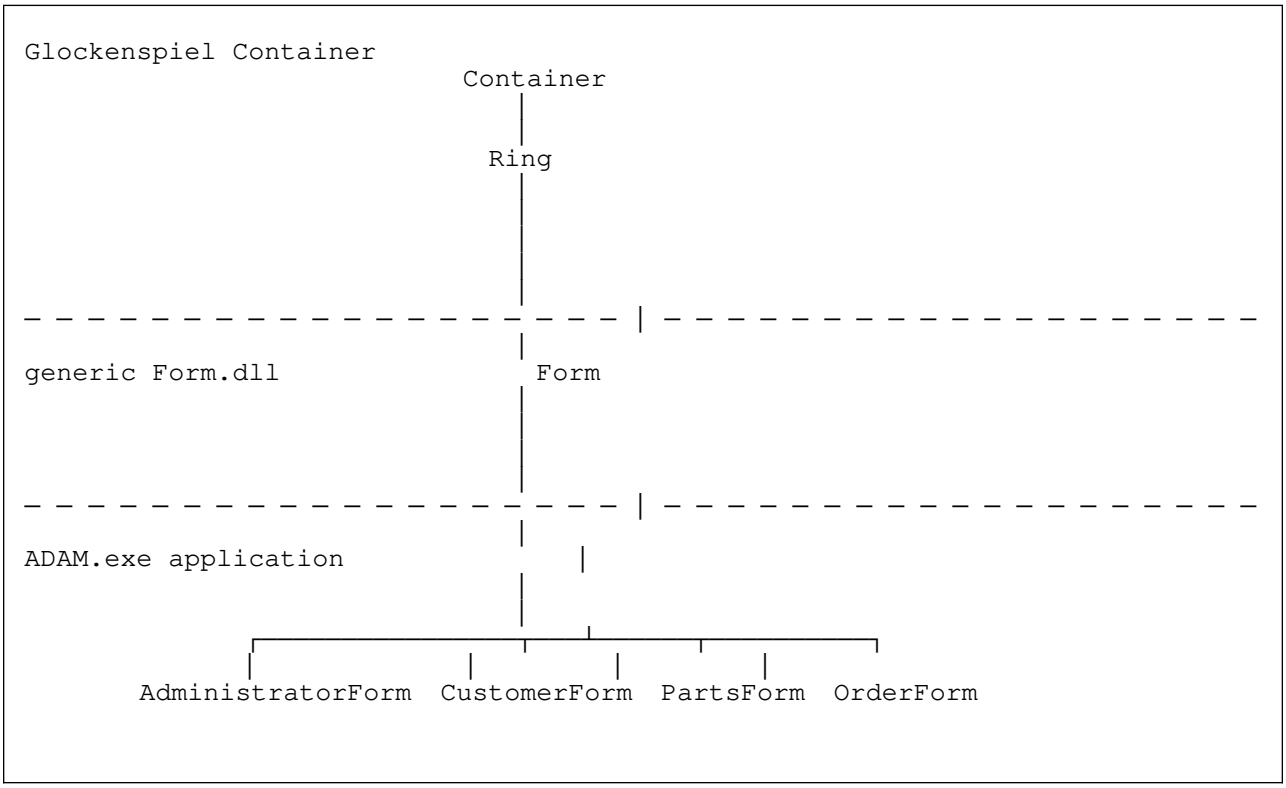


Figure 2. Form class hierarchy

The FormField class

The generic 'FormField' contains all the methods for handling 1 item of data - 1 'stream of bytes'.

What it does *not* contain is the actual data itself - whether integer, character string etc. That is only implemented in derived objects, which inherit all the generic methods of FormField.

'Formfield' itself is called an 'abstract class', in that instances of it are not meant to be used directly, but only through inheritance.

What is common to all fields?

(and hence can be implemented in the generic FormField class:)

- each field is a single 'stream of bytes'. Whatever particular structure it has, each field can be fully represented by a start address and a length (number of bytes to read from that address). In C++, this is a 'void pointer' and an unsigned integer.
- this means that we can copy information without knowing its format, each derived field responding to the message Eat (address) to copy information from that address into its own format.

Because of OO messaging, we can implement the entire algorithm to handle a 'stream of bytes'. All the application has to do is tell us what exactly the stream of bytes is.

OO allows inheritance of *algorithm* like this, in a way that would be near impossible in a Procedural language. (See “error-checking of the 'AMC' field” on page 78.)

selected 'virtual' methods of the class FormField:

FormField.Clear

set myself to a blank null state.

FormField.Default

set myself to the default value for my class, if there is one.

FormField.GetString

return a string representing my contents, suitable for displaying to the user.

FormField.isValidChar (character)

accept or reject this character (quick way to filter out certain characters).

FormField.MakeMe (string)

take this whole string and try to convert it into my own internal data format (whatever that is).

Note: the abstract class FormField only expresses the *desired system of messages*. It does not actually implement any of these. These can only be implemented in derived classes such as Text,NaturalNumber etc. These are 'virtual' functions, which only get bound to real function calls at run-time.

By burying all the error-checking and validation inside the Fields themselves, and by handling a list of fields that all respond to the same messages, the entry screen routine is dramatically simplified (See “The FormDlg class” on page 30.)

In my Forms class library, standard fields are provided - NaturalNumber, Text,NumericText,Date etc. All inherit from FormField, and all in turn may be inherited from into application-specific fields.

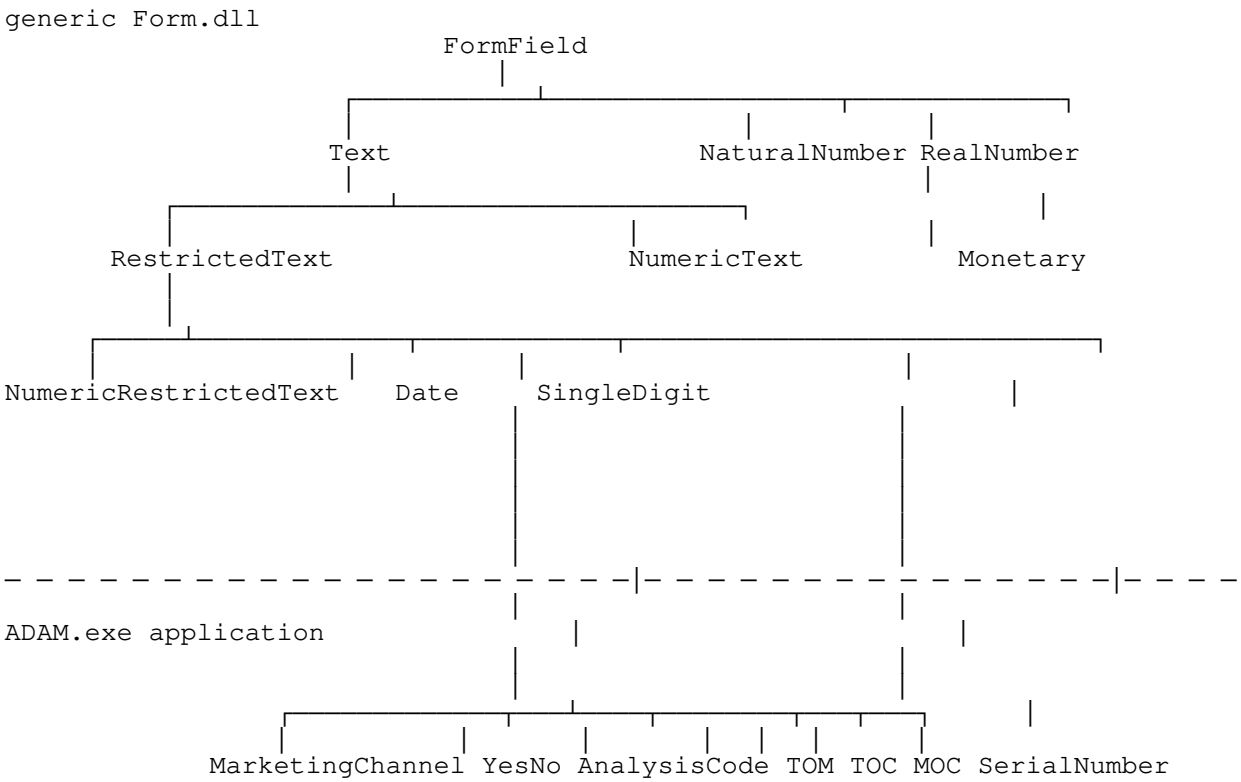


Figure 3. FormField class hierarchy

Note: the bottom-level fields are just various items needed for the AAS screens in ADAM. They are of no real interest.

The FormDlg class

The FormDlg class provides generic handling for linking any Form to any entry screen (or 'dialog', hence Form 'Dialog').

The Form and FormDlg have only limited knowledge about each other. The Form responds to requests for information, and the FormDlg sends input down to the Form for evaluation.

They are set up in communication by the FormDlg.Associate method, which associates entry fields in the dialog with real fields in the Form. Once this association is defined, everything else is taken care of by automatically generated messages.

- If the Form changes, it tells the FormDlg.
- If the FormDlg changes, it tells the Form.

A Form can exist internally, independent of any FormDlg, but a FormDlg only exists in conjunction with some Form it is editing.

On initialisation, each entry field asks its corresponding Form field for a string representing itself (the 'GetString' message). When the user inputs data, the entry field sends it down to the (sealed) FormField via the 'MakeMe' message - and gets a return - accepted or rejected.

To make this properly object-oriented, every single keystroke goes immediately down to the FormField object for verification. *(There is no time lapse noticed by the user typing, since any machine fast enough to run OS/2 will run any normal verification routine before the user has hit his next keystroke. This is an example where fast hardware enables a cleaner and more natural design)*

The rule I followed here is never to let bad input 'sit' in the dialo. You are typing into an 'intelligent' dialog box which is checking everything you are putting into it; not into a dumb box which lets you enter nonsense, and tells you hours later. As shown in “error-checking of the 'AMC' field” on page 78, coordinating calling all the verification routines with every keystroke would be near-impossible without OO messaging.

Also, from a design point of view, the data in the FormDlg should have no validity by itself, but should merely display what is contained in the Form it is editing.

The entry field should not know what type of field it is dealing with nor should the field know who is sending it messages. This type of information should not be hard-coded into each object - else the integrity of one object's code is lost when the object it communicates with changes. This is classic OO design. You do not have to know what exactly you are talking to. More important, you do not *want* to know.

The FormDlg class is (ultimately) built on top of CommonView's DialogWindow:

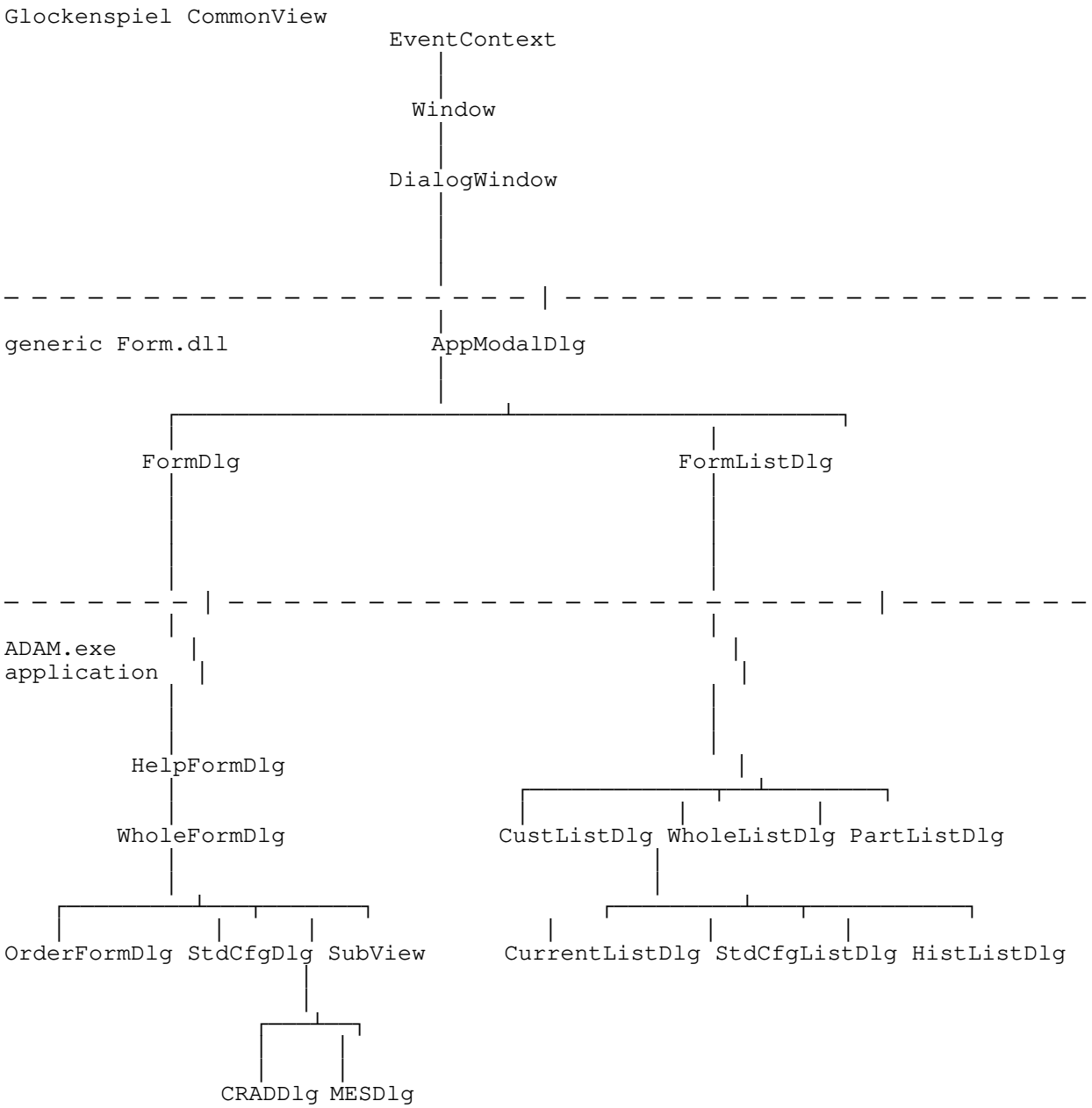


Figure 4. FormDlg and FormListDlg class hierarchy

Pseudo-Code Fragments - Form Initialisation and FormDlg Association

Initialising the form (defining what its fields are):

```
OrderForm.Constructor()
{
  Add ( NumericRestrictedText, 6, "customer number"   )
  Add ( Date              ,   "date of order"    )
  Add ( AnalysisCode       ,    "analysis code"    )
  Add ( NaturalNumber      ,    "quantity"        )
  Add ( Text              ,   "comments for PCMS" )
}
      (type names)            (variable names)
```

Initialising the entry screen
(associating its entry fields with fields in the form):

```
OrderFormDlg.Constructor()
{
  Associate ( EF_CUSTNO   , NumericRestrictedText, "customer number"   )
  Associate ( EF_DORDER   , Date              , "date of order"    )
  Associate ( EF_ANALYSIS , AnalysisCode       , "analysis code"    )
  Associate ( EF_QTY     , NaturalNumber      , "quantity"        )
  Associate ( EF_PCMSTXT  , Text              , "comments for PCMS" )
}
           (entry field ids)
```

In PM, each field would need an integer id to correspond to the PM entryfield. But an integer could also serve as the unique variable 'name', as in:

Initialising the form (defining what its fields are):

```
OrderForm.Constructor()
{
  Add ( NumericRestrictedText, 6,  EF_CUSTNO   )
  Add ( Date              ,    EF_DORDER   )
  Add ( AnalysisCode       ,    EF_ANALYSIS )
  Add ( NaturalNumber      ,    EF_QTY     )
  Add ( Text              ,    EF_PCMSTXT  )
}
      (type names)            (variable names)
                           (and also the ids that should
                            be used for the entry fields
                            in any dialog that wants to
                            edit this)
```

Any dialog editing this Form simply makes sure to use the correct ids. Add is done here, and Associate is done *implicitly* in the creation of the dialog.

Hence, with a still basically environment-free Form, the actual contents and layout of the Form can be referred to *once* only in the entire application.

Pseudo-Code Fragments - the entry screen procedure

Procedural entry screen handling:

```
// when the dialog is set up, we have a huge list of ..
  SetTextLimit ( EF_CUSTNO, 6 )
  SetTextLimit ( EF_DORDER, 8 )
    ....

         // set maximum lengths for each field on the entry screen


// .. and another huge list of ..
  SetText ( EF_CUSTNO, record.customer )
  SetText ( EF_DORDER, record.dorder   )
    ....

         // fill in the text into each field on the entry screen



// .. and all input is treated separately for each field ..
  QueryText ( EF_CUSTNO, string )
  if ( validCustNo ( string ) )
    then record.customer = string
    else error

  QueryText ( EF_DORDER, string )
  if ( validDate ( string ) )
    then record.dorder = string
    else error
    ....
```

OO entry screen handling:

```
// when a dialog is shown, it sends 'YouAreNowVisible' messages to
// all the fields, and they respond by sending it their own
// 'SetTextLimit' message
// the length information is buried in the object, not the screen handling

  CustomerObject.YouAreNowVisible()
  {
   EntryScreen.SetTextLimit ( 6 )
  }
  DateObject.YouAreNowVisible()
  {
   EntryScreen.SetTextLimit ( 8 )
  }
    ....



  EntryScreen.Constructor()
  {
   for ( all fields )
     field.YouAreNowVisible()
  }


// to get display text, we just send 'GetString' to all the fields
// we do not need to know how many or what kind
// of fields we are dealing with

  EntryScreen.Constructor()
  {
   for ( all fields )
     SetText ( field.id, field.GetString() )
  }

// and we receive input by sending 'MakeMe' to the field
// we do not need to know how many or what kind
// of fields we are dealing with

  EntryScreen.Destructor()
  {
   for ( all fields )
   {
    if ( not field.MakeMe ( QueryText ( field.id ) ) )
      then error
   }
  }
```

In the OO system, the entry screen does *no* validation. It doesn't even call validation routines. Validation is buried in the derived field object, which error-checks all attempts to put data into it.

You just use it. It validates itself.

The File class

Completing the generic Form handling library is the File class. This is the interface to a data file which is not fixed in format, but can store any variety of heterogeneous objects in the same physical file.

There is no longer a problem with data files when upgrading to new versions. We can mix old and new data formats at will. There is no longer any such thing as a 'file of (particular data structure)'. Files can contain anything.

The File class can read from the old data files into any new order form layout. Fields no longer used will be simply ignored, and new fields will simply stay blank. You can say things like this:

```
rel63History.Read ( rel7OrderForm )

   // read from the obsolete ( release 6.3  ) format order forms
   // into the new ( release 7.   ) order forms.
   // all matching fields will be copied across
```

There is total flexibility in this class - you can Read from any File into any Form, just like you can Eat anything.

Implementation Decisions

Warning: because it outlines my real implementation decisions, this section contains technical discussion which will only be comprehensible to (and only of interest to) the technical reader. I make no *attempt to explain the language in this section.*

The non-technical reader is advised to skip to *"More Questions" on page 40.*

Note: *this section contains opinions I formed during the course of this work, relevant to the various debates going on within the OO world at present.*

As such, I believe this section contributes to the general background of the project. If you are not interested in my opinions then please skip to "More Questions" on page 40. As I have said before, my main contribution to the OO debate is the set of comparison statistics, not *these general discussions about OO.*

The opinions in this section are the author's alone, and do not necessarily reflect the opinions or strategy of IBM Ireland Information Services Limited.

Why not a Procedural language?

The possibility of using a Procedural language was an issue at the start of this project. However, such a decision would, I feel, have been a disaster, and would have killed off most of my innovative ideas before implementing them.

First, I had to teach myself OO design methods. This was difficult, and required a major change in thinking. I would never have accomplished it without the aid of a language that supported me, namely C++. If I had to force OO design ideas into some Procedural language that did not support them, my learning curve would have been much longer.

Second, I had to code, and after a few months teaching myself OO, I never wanted to touch a Procedural language again in my life.

From considering C to be the state of the art in real applications development a few short months ago, I now feel that I could no more code in C than I could in one of the earlier languages like FORTRAN or COBOL.

I am not saying that this project could not be done in one of the old Procedural languages. But *I* would not like to try it.

(See also "Can Object-Oriented design be implemented in Procedural languages?" on page 22 and "Does it have to use an OO language?" on page 60.)

Why C++?

- because I wanted to retain direct access to the environment's API.

When choosing my OO language, I preferred to use a superset of C because the environment in which I was working, OS/2 Presentation Manager, only supports a C API. I knew that I would need access to this sometimes, no matter how sophisticated a programming environment I bought. I felt API access would be much simpler within a C-superset than within a separate language, notably Smalltalk/V PM.

(See “Migrating to OO” on page 60.)

(From recent experience with Smalltalk/V PM, API access is rather indirect, but as good as could be expected in the circumstances. It does seem that *any* API call can be made, but some will require defining new PM data structures, which could be quite a headache. Also, new releases of OS/2 come with a new C API, which can immediately be used by C++ applications, whereas Smalltalk applications have to wait until someone defines the Smalltalk equivalent for the new C function calls. (In fact, I *did* move between releases of OS/2 during the course of this project).

I used C++ 1.2, with single inheritance, and it was a real problem in trying to customise the behaviour of the supplied objects. I could not 'insert' items into the supplied class hierarchy (because I had no access to the source), and I ended up duplicating code in different branches of the derived tree.

Class reuse will, I believe, make *no* significant progress unless all developers have the ability to multiply inherit behaviour (in C++, this is available in rel 2.0).

To quote (Hardin):

'Faced with an extensive library of classes, ... if I can inherit from only one of them I will have to choose which one gives me the most leverage and then re-implement (or probably copy code) from the others I need. I suspect that this "do we need multiple inheritance" discussion is due to the early stage of OOP in most commercial enterprises. Ten years from now we will all wonder how we could have been so naive.'

Why Container?

- As the basis of my Form, I needed a list that could hold any number of any type of heterogeneous objects - basically a list that could hold anything.
- Container's Ring object met these requirements.
- It is rather clumsy, but most of its idiosyncrasies can be hidden in the generic library.
- Glockenspiel have totally re-written Container with C++ 2.0. I have not seen this yet. File support is apparently included.

Why CommonView?

OS/2 Presentation Manager is at heart an Object-Oriented system, and yet its manufacturers provide no OO programming interface to it. It is an environment of window classes, 'object' classes and messages, and yet we are expected to program it using old fashioned functions. Only when one has sampled GUI programming in a real OO language does one realise what an incredibly bad fit this is.

GLockenspiel's 'CommonView' - a collection of C++ classes - provide *some* idea of what an API for a window and messaging system should look like.

In CommonView, the PM windows, dialogs, controls, etc. are all C++ object sending C++ messages to each other. The C++ class DialogWindow inherits from Window, which inherits from EventContext, and so on. WM_MOUSEMOVE's get resolved automatically into a call of the virtual 'MouseMove (MouseEvent)' function inside the class 'Window' (MouseEvent itself being a class with methods such as WhichButton and ScreenPosition). Classes and messages are actually real *in the language*, not merely the environment.

In sad contrast, PM's C API has to 'fake' OO, with awkward workaround such as WinSubclassWindow and HWND_OBJECT. C was simply not designed for this type of job (see C++VanC90). We have 273 functions called WinDoSomething (HWND hwnd, (args)), when they should all be member functions of class Window called DoSomething (args).

In C, you WinRegisterClass to get a measly MyClassWindowProc, from which you distribute the messages *yourself* to MyClassWM_CREATE, MyClassWM_SIZE, etc. All messages are sent to the same function, from which the 'C' programmer must laboriously sort and re-route them to his various message handling functions. This is typical of the difficulty Procedural languages have in calling 'functions' unless they really exist, hard-coded, at compile-time. (See C++Lew90).

In C++, the *system* itself distributes the messages to each object's WindowInit method, each object's ReSize method, etc. Why is this possible? Because C++ can call the same 'virtual' function across a set of objects, mapping to a different real function in each one. In practice, C has a lot of difficulty doing this sort of thing. That is why the C API does not distribute the messages for us.

Of course, CommonView does not provide a *complete* API for PM (only Microsoft/IBM could do that). Instead, it gives you some sort of an OO framework around which to build your applications. The programmer will frequently dip into the C API to augment the classes provided. I have dozens of API calls buried in the generic library; but only 2 API calls were needed in the application itself.

An OO API fits onto a modern windowed GUI like a glove. The code needed to write user interfaces in CommonView or Smalltalk V is an order of magnitude smaller, simpler and more flexible than that needed to do the same work in C. And the reason why, is that the OO API is able to take away much more of the work from the programmer, because of its closer match to the environment.

(An issue which illustrates this is CUA, IBM's standard for user interfaces. Once the decision is made to write CUA code, who wants to spend time laboriously checking that an application conforms to it? One wants to simply inherit from system-supplied CUA Objects, all of whose behaviour is CUA conformant.

Someday, such objects will come with the operating system, standardising something that was unnecessarily free-floating, and liberating the programmer to work on more important things. These classes will be *CUA - the definition of CUA will be written in binary and not in text).*

Presentation Manager, like any modern GUI, is *made* to be programmed with a class library, not with a function library. Until such appears, CommonView helps in some way to fill that gap.

More Questions

But surely I need to know how your generic 'Forms' work if I want to reuse them?

Not at all. I myself have no idea how Rings work, and I reused *them* to make the Forms!

All I know is that they give me a list of objects of any type. I don't even know what the data structure of the Ring is, or whether it is a linked list or whatever. I just send it messages like 'Reset' and 'Next', and it does what I ask it to.

In OO, objects typically only modify the data contained within themselves. And they maintain their own consistency. When you use objects, you only have to know what the messages are 'meant to do', not how it is actually done.

ADAM-OO

```
ADAM-Procedural    OS/2 PM   C    Procedural        data capture, host

ADAM-OO           OS/2 PM   C++  Object-Oriented    data capture
```

Essentially, the entire Object-Oriented implementation of ADAM is a simple derivation from all the generic objects and behaviour designed in the Forms library.

The Forms are given some real lists of fields to work with, the FormDlgs are linked to some real OS/2 .DLG files, and off they go, talking to each other with all their generic knowledge. It could be 'ADAM', it could be something else - it doesn't matter. What I have got in the Forms library is a 'generic application' for this type of data capture.

The ADAM-OO application itself merely defines forms to capture the data, and sets up dialogs to edit them in, with menus driving the whole application. The data consists of WTAAS fields, required for different screens on the host, and requiring individual error-checking. What exactly these are is of no interest here.

Chapter 6. Cost Comparison Statistics - OO versus Procedural

Caution

No *single* set of figures can stand on its own as a comparison between the two applications. Lines of Code may give some idea of the cost in time to produce the application from the start, but gives no information as to how difficult it will be to upgrade/modify it, or give it to a new programmer to work with.

These statistics must all be taken *together* in order to form an accurate comparison of the two applications.

* * * * * * * *

Out of all the figures in this section, perhaps the most exciting are the ones that show how, in an ordinary, common or garden application, over 90% of the actual data entry handling was really generic-type action that didn't belong in the application at all.

But until OO, there was no way of lifting it out.

Raw Statistics

Design and Code Time

Because of the innovative, re-iterative nature of this work, design time and code time cannot be properly separated.

The basic design of Forms co-ordinating Fields of bytes remained more or less unchanged, but there was a lot of movement up and down the class hierarchy. This being my first experience with OO, I underestimated the amount of algorithm that could be made generic, and I was coding fairly general behaviour into the ADAM-OO objects themselves.

I soon realised that the ADAM-OO objects should contain nothing except purely ADAM-specific behaviour, and I started lifting algorithm out of ADAM-OO and up into generic Form objects.

Having had some experience now with class hierarchies, I believe that I would achieve a clearer separation of application-specific code and generic code next time. However, I still think that *some* re-iteration or feedback from the application is necessary for designing good working objects. (see “How should it actually design class hierarchies / applications?” on page 62)

These figures are for design and code *combined* of the data capture section (so that we can properly compare the two applications). Note that 'EVE' stands for a

similar data capture application with a different specification (different fields to be edited and filed, different screens to capture the data, different edit-checking).

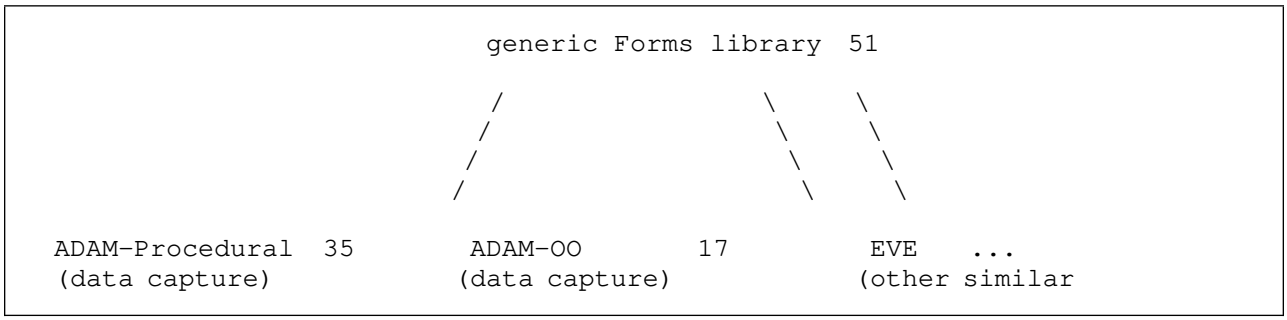


Figure 5. Man days taken for design and code

For 'EVE', there would probably be a lot less re-iteration. It still remains to be seen whether the 'design, then code' path is applicable to OO. Possibly a more accurate path would be 'design, specifying required objects, search class libraries, derive necessary objects, (re-iterate, extracting generic behaviour into new library objects)'. (Cav90)

These figures would indicate something like a 50% cost saving per application, once there is a well-defined set of generic objects to 'inherit' from. We will see if the Lines of Code count backs this up.

Lines Of Code

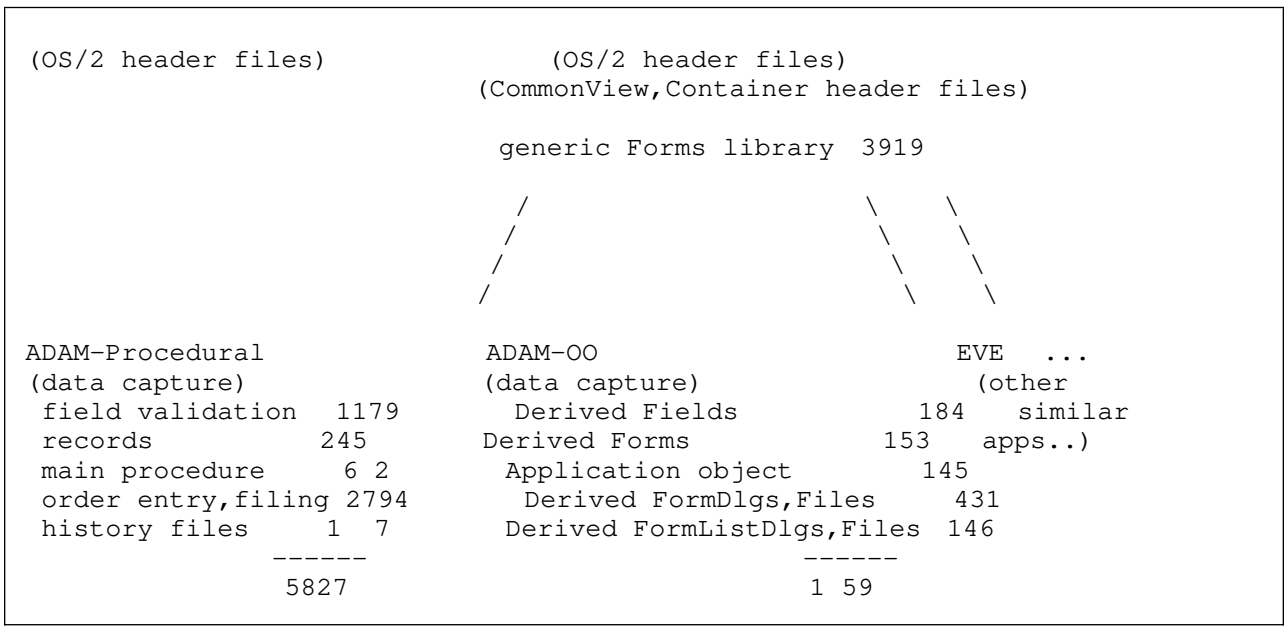


Figure 6. Executable lines of code

ADAM-Procedural does not 'reuse' code in the sense that it does not use standard library code or any generic form handling.

It did, however, involve cut-and-paste (with considerable modifications) from the previous, DOS version of ADAM. This cannot be accurately separated from the

code, but I allow for this in my analysis (See “Analysis of ADAM-Procedural” on page 52).

The Executable Image

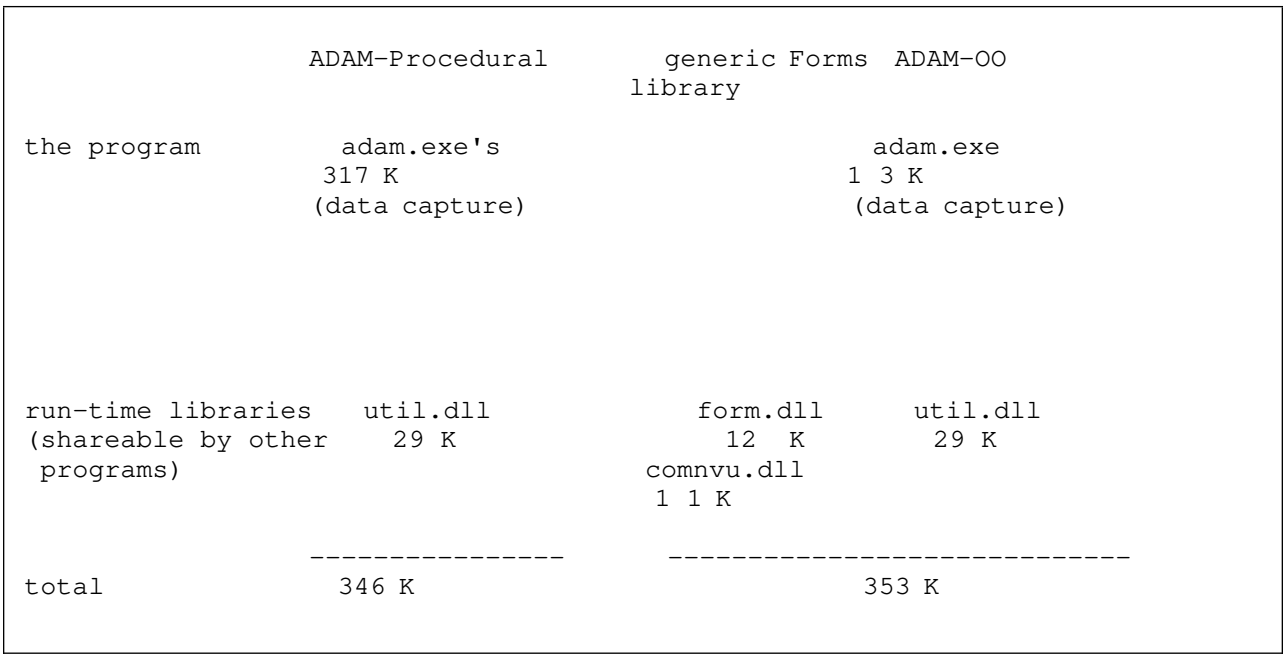


Figure 7. Executable file size in Kilobytes

ADAM-Procedural had a few .exe files, of which the data capture elements totalled 317 K. Of course, this figure would be smaller if they were all combined into one adam.exe.

When ADAM-Procedural's host section was finished, all .exe's *were* combined, into one 257 K adam.exe file.

Note: that I have continued the separation of generic, shareable objects into the executable files. ADAM.exe requires only 103 K, and uses the services of the Forms run-time class library Form.dll, which can be shared by any other application which uses Forms *(note here that the application is actually smaller than the library..)*.

In contrast, ADAM-Procedural's code is ADAM-specific, and so a .dll would be of no use to any other program.

```
program performance      no timing measurements were taken.
                      ( from a user's point of view, however,
                        there is no visible difference )
```

Program Complexity

This is something that is very difficult to evaluate. Before reading the following section, make sure you have understood Chapter 5, “The IISL Innovative Solutions project” on page 23.

For a deeper understanding of *why* the following statistics were possible, the C programmer should look at the C++ code fragments in the Appendix.

No one statistic can measure program complexity, but I believe that taken together, the statistics in this section do illustrate that the OO version of ADAM is an order of magnitude more malleable and flexible than its predecessor.

Fixed-size, hard-coded field layout is eliminated

The fixed-size data structures used to hold the order entry data have been replaced by flexible Form objects.

In ADAM-Procedural, the precise field layout is referred to everywhere, and any changes would get echoed all throughout the application. In ADAM-OO, everything 'talks' to the Forms, and the Forms may change their field layout anytime they like.

	ADAM-Procedural (data capture)	generic Forms library	ADAM-OO (data capture)
# data structures	3		
# Form classes		1	4

Figure 8. traditional data structures and OO Form classes in the applications

But the really telling statistic is the number of hard-coded references to the actual form layout (the number and type of fields).

Throughout the application, ADAM-Procedural refers to editing the customer field, editing the AMC field etc, whereas ADAM-OO is built on a system where messages get delivered to whatever field is there. New fields may be plugged in and out without disturbing the application.

Number of hard-coded references to the actual layout of the fields:

	ADAM-Procedural (data capture)	generic Forms library	ADAM-OO (data capture)
field validation module	1		
record definitions	4		1
main procedure			
order entry, filing module	26		1
history file module	8		

Figure 9. number of hard-codings of the ADAM field layout

Global variables are near eliminated

```
                     ADAM-Procedural      generic Forms  ADAM-OO
                     (data capture)        library      (data capture)

# application globals      134                3            3
# module globals          111                3            2


global variables used to hold order entry data
(accessed in each of the data entry functions)

                         49
```

Figure 10. global variables

ADAM-Procedural relies heavily on global variables to pass data, thus binding the network of functions even tighter together, and increasing the ripple effects of any changes. Any global variable tends to bind the whole program together, and destroys the concept of reusable sub-components.

Admittedly, this is not good practice, even in Procedural design (see below).

The OO code uses object internal data and heavy message passing to maximise the independence of separate components.

Function is successfully buried inside Objects

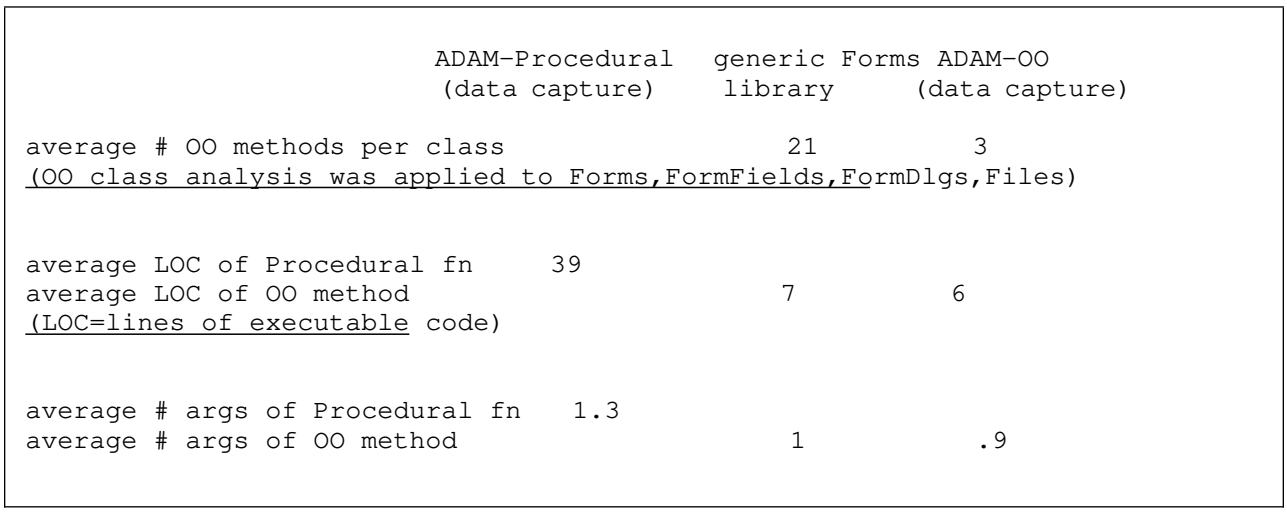
This chart will compare the internals of a Procedural program, which is made up entirely of a network of discrete functions, not bound to any data; with an Object-Oriented program, which consists mainly of objects with internal methods (or 'member functions') which do all the real work, when they respond to messages sent to the object.

```
                     ADAM-Procedural     generic Forms  ADAM-OO
                     (data capture)       library      (data capture)

# Procedural functions     1 7               2           2

# OO methods                                284          63
```

Figure 11. Procedural functions and OO methods

This shows how natural class hierarchies are. Program function really does fit naturally inside objects, inheriting function from more general objects. Also note that ADAM-OO is in fact more successful at 'functional decomposition' than ADAM-Procedural - at least in terms of breaking down algorithm into small and simple subroutines.

Note also the continued existence of the odd old-fashioned, stand-alone function - allowed by C++, not allowed by Smalltalk.



	ADAM-Procedural (data capture)	generic Forms library	ADAM-OO (data capture)
average # OO methods per class		21	3
(OO class analysis was applied to Forms,FormFields,FormDlgs,Files)			
average LOC of Procedural fn	39		
average LOC of OO method		7	6
(LOC=lines of executable code)			
average # args of Procedural fn	1.3		
average # args of OO method		1	.9

Figure 12. Analysis of Procedural functions and OO methods

The general OO trend is towards a large number of methods per class, but most of them being quite short.

The low number of arguments passed in ADAM-Procedural is due to the bad practice of using global variables instead. The low number of arguments in ADAM-OO comes from the possession by the objects of their own internal data.

Had the Procedural programmer made an effort to eliminate global variables, we would find that his average number of arguments per function would have risen considerably.

Reuse

Here I attempt to measure just how 'generic' my Forms classes really are.

The Forms library contains the classes Form, FormField, FormDlg and File, with some standard derivations of FormField such as NaturalNumberField, TextField etc.

The ADAM application contains derived (or inherited) classes. ADAMForm's derive from Form, ADAMField's from FormField or one of its children, and ADAMFormDlg's inherit the behaviour of FormDlg.

Hence, for any given class in the ADAM application, most of its work is really being done by its generic 'parent'. All of this represents work that was previously hard-coded into the ADAM application.

Here I compare the code involved in the various classes, under the headings:

- number of internal data items in the class definition
- number of internal functions (methods) in the class definition
- total executable lines of code of the class's methods
- average lines of code per method
- average number of arguments per method

Algorithm is shifted upward into a generic library:

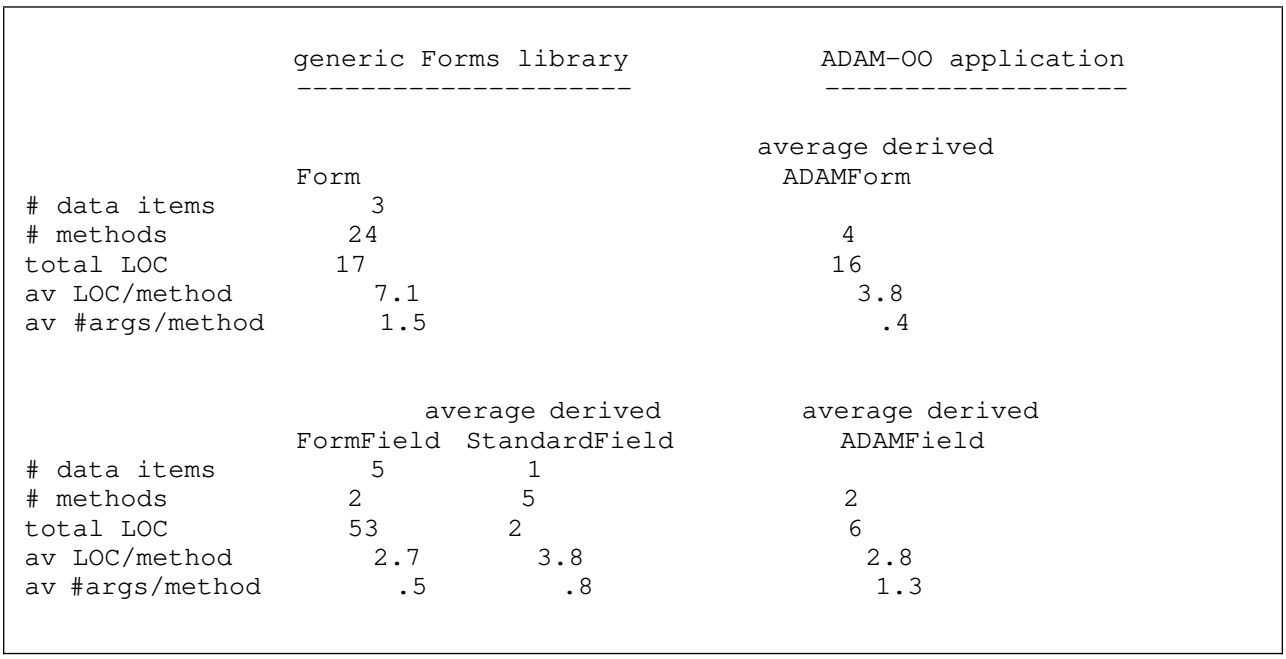


Figure 13. breakdown of code in class hierarchies (LOC=lines of executable code)

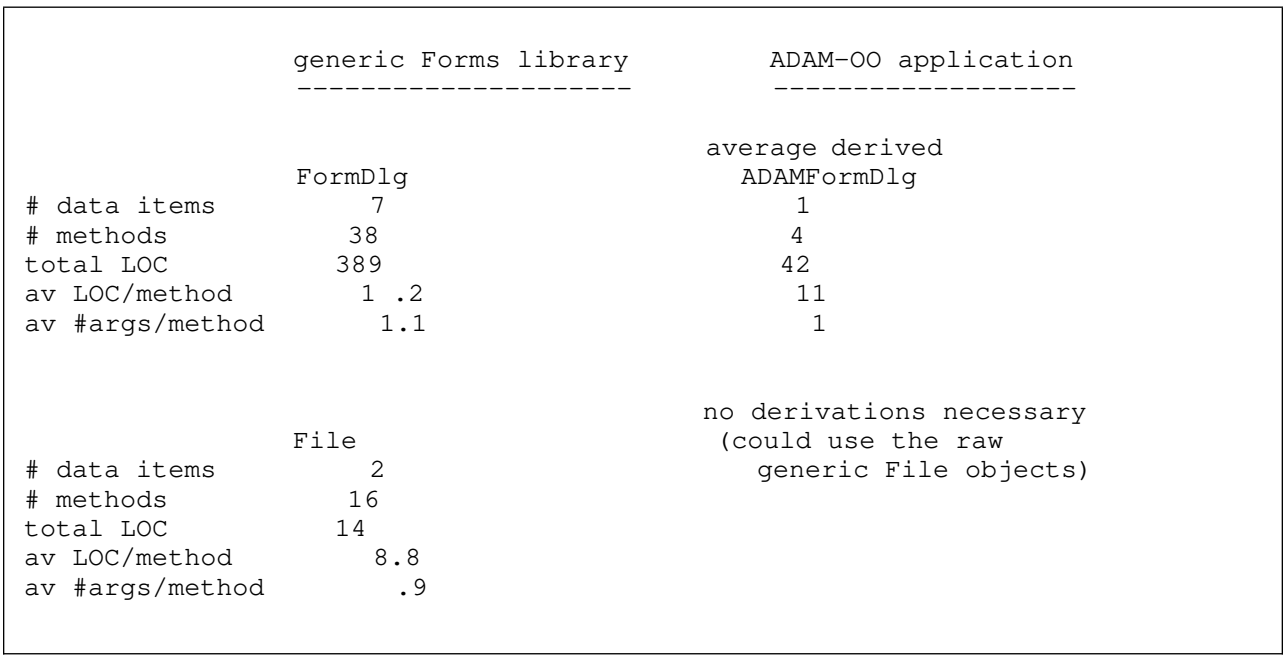


Figure 14. breakdown of code in class hierarchies (LOC=lines of executable code)

These figures are quite startling:

- The typical ADAMObject is able to extract over 90% of its algorithm into generic FormObjects!
- Most of this algorithm is explicitly coded in ADAM-Procedural, but is generic in nature, and so does not belong there at all.
- Could it be that *all* applications needlessly repeat so much work that has been done before? And could it be that with a little more advanced languages we could *really* find a way to cut it out?

- In this instance, the ADAM application is stripped down almost to a mere template - purely expressing what makes ADAM 'ADAM'. This is the Brave New World of component and inheritance programming - this is the way it should be!

Extendibility - specific scenarios

Scenario - spec changes, need 10 more (standard) fields in order form

- ADAM-Procedural

  redesign .DLG file, redefine form data structures, rewrite all dialog handling (new SetTextLimits, new SetTexts, new QueryTexts, new validation code), rewrite functions to copy configs,history etc into orderforms, rewrite functions to copy orderform info to host screens, write functions to read old data files and read into new data structures and write to new data files

- ADAM-OO

  redesign .DLG, add 10 *lines* to the single method OrderForm.Constructor().

Scenario - want to redesign all the dialogs used to edit the forms

- ADAM-Procedural

  redesign .DLG files, rewrite all SetTextLimit,SetText,QueryText, and validation code (essentially, rewrite the entire user interface)

- ADAM-OO

  redesign .DLGs, set IDs so that different objects get associated to different entry fields.

Scenario - want to add a Clear function, to clear all the form fields

- ADAM-Procedural

  write ClearOrd(OrderForm) to go thru fields, setting one to 0, another to the null string etc (will have to refer to each field explicitly by name), write ClearStd(StandardConfig) to go thru *its* set of fields, write ClearPart(PartForm), .. etc

- ADAM-OO

  send a 'Clear' message to all the fields. Don't need to know how many fields there are, or what they are. They each respond to the 'Clear' message in their own individual way.

Scenario - a new company directive requires credit-checking of all customers

- Procedural site

  Each individual application must be re-edited, analysed, checks for credit-worthiness inserted at appropriate points, and re-compiled. Any 'touching' of an old application like this is time-consuming and likely to introduce new bugs.

- OO site

  The 'Customer' object, shared by all applications, is modified to check its own credit-worthiness. The applications are not even aware that this is happening.

Conclusion of Statistics

One conclusion of these statistics is clear, I hope, to the reader - the OO application is much simpler, less cluttered, and an order of magnitude more flexible than the Procedural version. (if unconvinced, see Appendix B, “C and C++ Code Fragments” on page 71)

I will turn now to the real development cost involved in using these techniques.

Analysis of ADAM-Procedural

The Procedural version *did* 'reuse' code. In the classic Procedural tradition of 'reusing' function written for different data structures, much of the old, full screen ADAM for DOS could be cut and pasted into the editor for the OS/2 version, and re-edited to work. Similarly, standard blocks of Presentation Manager dialog handling could be edited to fit.

Out of 5800 total executable LOC, the original author estimated that there was 2000 LOC written from scratch, and the rest cannibalised from other sources (involving only about 500 LOC of actual new code).

Therefore, on a once-off basis, ADAM-Procedural represented 2500 LOC of original effort.

Similarly, he estimated the cost of building a Procedural 'EVE' application - representing the next version of ADAM, or at least a similar data capture application in this environment.

He estimated that about 1/2 of ADAM-Procedural could be reused in an EVE-Procedural, giving a real cost of about 2500 LOC new code.

Analysis of ADAM-OO

The 1050 LOC in ADAM really does *belong* there. It is ADAM specific, and is the kind of thing that must be written from scratch with a new specification.

(But that is the way it should be. If it was generic code, it would not belong in ADAM at all.)

Hence we will estimate the real cost of each similar OO application to be 1050 LOC.

Qualifier - is the Forms library generic?

I assume that the generic library is a good, once-off encapsulation of the site's generic objects, *requiring minimal maintenance*.

I say this because I believe that my Forms library is truly reusable, and hence represents the effort required to build a library which can be reused without maintenance.

How can I know that after one application? Because of the complete lack of attention I paid to the specification of ADAM while I was developing most of the Forms library. I established that it was some sort of data capture application for sending data to host screens, and I put it aside and went off to work out my Forms ideas. Only when Forms, FormFields and Files were built and working did I move

on to actually have a proper look at ADAM's specification. And I found I *could* use them. They did turn out to be generic.

The Form has no idea how many fields it has, the File has no idea what it is filing, the FormField has no idea what its actual piece of data is, etc. They represent general 'algorithm' for form handling - not a solution to some specific problem.

Whatever you think of the 'reuse' of Forms here, I *did* demonstrate completely independent OO reuse in this project - by reusing Glockenspiel's C++ class libraries. Most of my objects are built on top of working objects from the Container and CommonView class hierarchies, which I scavenged and subclassed rather than build anything myself. Because of the pains I kept taking to separate generic algorithm from application algorithm, I do believe that my Forms class library is reusable in a similar fashion.

Hence I believe these figures do provide an accurate measure of the savings gained by real OO reuse in normal application development.

Projected Procedural v OO costs over time

This is an extrapolation, on the basis of one single project, but I believe that it is illustrating a real point.

I assume (see “Qualifier - is the Forms library generic?” on page 52) that the Forms library is a true generic library for form handling, reusable by other similar data capture applications (of which there are many here in IISL).

On this basis, I estimate the cost of developing a few such applications using the old Procedural way of writing it all from scratch, versus the OO way of inheriting generic work. These statistics show that the cost of building a generic library is more than offset by the savings in the long run.

The figures themselves represent new hundreds of lines of code that must be written from scratch (see “Analysis of ADAM-Procedural” on page 52).

once-off, initial investment in generic library:

```
                                        generic library  39
 Procedural application   25                 OO application   11
                     ---                                 ---
                     25                                  5
```

after another, similar-type application:

```
                                        generic library  39
 Procedural applications  25 25              OO applications  11 11
                     ------                                ------
                     5                                   61
```

and for 3 applications plus, OO gives 5 % saving per application, as cost of original investment becomes irrelevant:

```
                                        generic library  39
 Procedural applications  25 25 25           OO applications  11 11 11
                     ---------                                ---------
                     75                                  72


                                        generic library  39
 Procedural applications  25 25 25 25        OO applications  11 11 11 11
                     ------------                             ------------
                     1                                   83

           etc                                 etc
```

Obviously, there is a certain 'critical mass' needed to make OO worthwhile. If you are only writing once-off programs, with nothing in common, then you should hard-code it, quick and dirty.

If you are ever going to be asked for a release 1.1, or a similar-type application, then you should consider looking into the new forms of reuse empowered by Object-Orientation. The cost of designing some generic, shareable objects will more than pay for itself in the long run.

Chapter 7. Deliverables

The Report

To obtain a softcopy of this report from within IBM, issue the command:

Request Report from Humphrys at DubVM1

This package contains the full report (i.e. this file), a summary report, and also summary foils.

It contains binary versions (for immediate printing), and source versions (in case the binaries are unusable, and also to give you plain text to cut-and-paste from if you want to quote this electronically).

Feel free to distribute this report (within or outside of IBM - it is unclassified). I hope that it can be quoted to site management reluctant to take the step into the OO world.

To obtain a softcopy of this report from *outside* IBM, send a note to:

humphrys@dubvm1.vnet.ibm.com

I would appreciate some (brief) description of who you are to accompany the request.

The Forms class library

I have tried to write the Forms library as a standard module, reusable across applications. Only attempts by others to use it in their applications will actually prove its worth.

To obtain a copy of the Forms class library from within IBM, issue the command:

Request FormDemo from Humphrys at DubVM1

This package contains the Forms class library, and some demonstrations of its use. Pre-requisite is OS/2 1.2 or above. If you want to try some coding with these classes, there will be the additional pre-requisite of Glockenspiel CommonView 1.1 (with their C++ 1.2 compiler).

I am making this requestable purely to generate discussion and feedback. I can guarantee no support for this library, not even bug fixes. It is most definitely 'use at your own risk'.

Note that if you find bugs, there is no need to dump the whole library. Remember this is OO - and OO provides new ways of 'salvaging' useful but flawed code. Fix the bugs entirely in subclasses, overriding the flawed method and sending it into that twilight zone of methods that cannot be called.

Further work

Add multiple inheritance to Forms class library

Multiple inheritance capability has now been added to C++ - the ability for a class to inherit behaviour from more than one parent.

This is something I would like to incorporate into my Forms library. It will allow me implement a cleaner low-level design, separating environment-free 'Forms' from 'windowForms' (for any windowing system), 'PMForms', 'MSWindowsForms', 'MacForms' etc. I can then inherit from separate branches, rigorously separating data from environment. The portability of applications using this system would be greatly enhanced.

Multiple inheritance is a fundamental step, enabling a powerful new way of writing applications. It adds one more degree of freedom to the flexibility of an OO program. It is an optional feature, which can be ignored or used as desired.

Direct Manipulation:

Multiple inheritance will allow me add all my enhancements to the CommonView objects without mutilating the original class hierarchy. In particular, elegant Direct Manipulation will at last be possible.

There are innumerable schemes for Direct Manipulation, and no standard has emerged. ***Note: this section was written before OS/2 finally standardised drag and drop, but the discussion should still be of interest.***

In this project, I found I had (independently) implemented Direct Manipulation in a similar fashion to the New World Demo (NWD90), but single inheritance makes both of our implementations clumsy, and fails to separate the DM function from the rest of the application.

The basic problem is that I *want* my window to be monitoring mouse messages, but I don't want to *know* about it. It should all be hidden from my application. (DRAGDROP on OS2TOOLS shows the limits of what can be done in PM C, using WinSubclassWindow to substitute DrpSourceProc for the Window procedure.)

To me, there is one ideal way to do Direct Manipulation, and it requires OO, and above all, multiple inheritance:

Objects who want to be manipulated should simply inherit from DMObject. DMObjects monitor the mouse, and send all mouse drag messages to the global object DMMonitor. DMMonitor monitors all drags around the desktop, and when a drag is successful sends a 'this DMObject has dropped on you' message to the target DMObject.

All DM code is encapsulated in the library classes DMObject and DMMonitor. Nowhere else. The derived Object knows nothing at all about DM. All it does is something in response to the 'DroppedOnYou (thing)' message (i.e. Eats it, kicks off a program with it, whatever).

(Note:. that without multiple inheritance you cannot say 'simply inherit from DMObject' since your single-inheritance object probably already has a parent. For example, it might already be embedded in some fixed hierarchy like CommonView.)

Long term - commercial possibilities

- ***Locally develop Forms class library to maturity***.
- ***Distribute/sell generic Forms class library company-wide/externally***.
  - include internal, environment-free Form objects, and PMForms, MSWindowsForms, FullScreenForms, DOSForms etc for different environments
  - ideal for data-gathering applications (such as gathering PC data for sending to host).
  - an all-PC application would probably want to process the data itself, so a full OO database might be more applicable.
- ***Look into the setting up of worldwide IBM class library repositories***.

  OO is here. Sites are writing OO, defining classes, reusing other work. Even on a local level OO has many benefits, but on a corporate level the potential of all this reusable, modifyable algorithm is immense. If this potential is to be exploited, we must have a well defined network of class repositories from which sites can select ready made objects to fit their needs.

  This should be begun on an informal basis, since it will take a long time to sort and identify genuinely useful and reusable hierarchies.
- ***Look into the definition of a worldwide IBM Corporate Library***.

  In the long term, if most sites are really going to go to OO as their standard software methodology, then why not get more ambitious? Most IBM sites do similar business with similar objects, and write similar code and routines to process those objects. All could usefully inherit a company wide 6 digit CustomerNumber object, for example. Why should they all define their own spec for this? Why not just inherit from an IBM-wide object?

  Would it *really* be possible to define objects common to entire areas of IBM's business? After probably a thousand different implementations worldwide of the IBMPartNumber, maybe it is worth a try.

  A possible scheme for this would be:

  - given the diversity of languages and platforms, one should begin by defining protocols, expressing the desirable behaviour of the IBM wide objects.
  - then, the protocols should be implemented in each environment, producing a working IBMObject hierarchy for each. Sites would use and inherit from this, saving them a large amount of effort.
  - much of the protocol would be very high level behaviour, which would make it easier to implement across different platforms.
  - sites would receive an IBMObject hierarchy, which they would leave untouched.
  - all sites would customise, but they would agree to build all their customisation into derived classes. Thus they would all have the same base IBMObject hierarchy.

  My experience with reusing Glockenspiel's product leads me to believe that such a scheme would have no chance at all unless all sites had access to

multiple inheritance. One needs to be able to completely isolate hierarchy trees from each other, yet still inherit from each.

Migrating to OO

Note: this section contains opinions I formed during the course of this work, relevant to the various debates going on within the OO world at present.

As such, I believe this section to contribute to the general background of the project. If you are not interested in my opinions then please skip to "The OO Development Site" on page 64.

The opinions in this section are the author's alone, and do not necessarily reflect the opinions or strategy of IBM Ireland Information Services Limited.

The question arises - ***How should a site begin using OO?***

Does it have to use an OO language?

Yes. No question about it.

The programmers are going to have to learn a new and powerful methodology. They are going to have to discard all the bad habits that conspire against abstraction and reuse. They will never achieve this unless they work in a language that supports the methodology.

Doing OO in a Procedural language will *not* save costs of retraining. It will *increase* the costs of retraining, since it will take far longer to understand and appreciate OO. It will also be far more difficult to implement.

I cannot emphasise enough that OO is different. Learning OO is far more difficult than simply learning some new language in a familiar paradigm. If your programmers cannot learn a new language, there is *no way* that they will be able to learn a completely new programming methodology, so your efforts to move to OO are doomed from the start.

All technical arguments aside, a short term problem for IBM sites may be the current lack of any IBM OO compiler. There would not be much risk involved in using non-IBM C++ or Smalltalk, since C++ has so many people selling compilers for it (including Microsoft soon), and since Digitalk have made arrangements with IBM to guarantee Smalltalk support. However, using non-IBM *class hierarchies* might be considered too risky unless source code is purchasable.

If at all possible (given the environment and application), one should endeavour to use an OO language. (See "Can Object-Oriented design be implemented in Procedural languages?" on page 22 and "Why not a Procedural language?" on page 36.)

What is the learning curve?

Considerable. OO's ideas are subtle and easily misunderstood. Even when they are finally understood, it is often difficult to see why they are a good thing.

An open mind is essential. So is an ability to deal with a higher level of abstraction. Not every programmer will be able to deal with such things as virtual function calls and objects that 'do not work'.

This is a long term investment, and will demand a proper investment in education.

What language should it use?

Problematic. Smalltalk and C++ are the oldest/best-supported, and the differences between them illustrate the two fundamental differences of approach to OO languages.

Smalltalk is the purest, allows rapid development, testing of sub-components on the fly, and typically comes with a powerful debugger and probably the best class hierarchy in the OO world.

The one potential problem with Smalltalk is that it *forces* you to do high-level Object-Oriented programming, all the time. This is not necessarily a good thing.

When you are in Smalltalk, you can forget about memory spaces and function addresses - you completely lose touch with the underlying environment. It is a language where the developer can no longer conceptualise what kind of machine instructions his high-level code is actually mapping to. One sometimes feels that one has handed over too much control to the Smalltalk system itself.

OO is good, but it should not be a religion. It is nice to be able to step outside it once in a while, and write some good old-fashioned dirty code.

Also, Smalltalk only supports single inheritance, which is probably an unreal approach to the development work of the future. Developers of the future will need to inherit from classes embedded in many different hierarchies (see (Hardin)'s quote above). C++ (like C) is probably better equipped to deal with these nasty, 'real-world' issues.

C++'s big strengths are that you can *still* get dirty and low-level if you want to, its run-times are (typically) much quicker, and it has multiple inheritance.

As a short-term advantage, you also have full access to the large body of C code existing worldwide. This would include such things as the Windows/PM APIs, your site's own store of useful C code, commercial libraries, and old friends such as stdio.h (enamoured as I am with C++, I *still* refuse to abandon sprintf ..)

But more important than being a bridge from the Procedural world, C++ has a fundamental philosophy difference with Smalltalk. C++ retains the philosophy of C - that the programmer should be able to do anything he wants - irrespective of good design or style.

This flexibility is why C became the pre-eminent Procedural language for developers. And it is retained also in C++, which allows the odd heretical stand-alone function, and inter-leaving of OO messages with calls to old-fashioned function libraries - rather than having to force *everything* into the object-message paradigm. This is a language that treats the programmer as an adult, rather than telling him what to do.

'Pascal and Smalltalk are straitjackets that enforce their programming paradigms; perfect for students.' (OOPSZimm90)

C and C++ *support* their programming paradigms, but still let you write assembly language. Perfect for developers.

It is still anyone's guess which language will become the industry standard, but I have a feeling that the ability to 'get dirty', à la C++, will be something that is needed for many years to come.

Should it build a class library?

Yes, but it should build it in iterations. It should write applications with objects, gradually extracting site-generic behaviour into site-generic class library objects. This is the only way they can be shared across applications.

It should buy such generic objects as Lists, Streams and Windows from a thoroughly debugged commercial library; and then inherit their properties into objects that can express its commercial activities.

How hard are class libraries to build?

It depends. I attempted quite an ambitious one here - trying to replace the C structure as I had seen it used in innumerable applications. My objects are very generic, having nothing in particular to do with work here at IISL, let alone the program ADAM. I found it took a long time to get them working properly, involving many iterations.

A more modest class library would simply try to organise the site's common functions into some sort of structured hierarchy of objects.

How should it actually design class hierarchies / applications?

Guidelines for OO design are still emerging, but a few points are now generally agreed on:

- Top-down is completely the wrong approach. It relies on a stable specification, breaking the problem down into functional sub-problems. This is completely contrary to the OO ethic. It models the aspect of the system most likely to change, and therefore goes against reusability.
- Good OO design should be first of all bottom-up, ignoring program specifications and concentrating on modelling the objects in the problem domain. These will be (relatively) stable across applications. The specifications describing their interaction will constantly change.
- More than this, however, OO design is not pure data-driven. The objects are described by the function they offer to the outside world.
- Hence the OO methodology to design *class hierarchies* works out as 'consider problems in this domain - use them to design objects and their methods - re-iterate by trying to solve problems with these objects'. All the time the problem is irrelevant. The end product is the system of objects.
- The goal of reuse through inheritance also complicates the design process, since many iterations may be necessary to move function up and down the class hierarchy until it finds its proper place - generic function buried in the generic objects, application function only in the application objects. (Cav90) This form of re-iteration is unlikely to be disruptive, since the objects end up inheriting the same code anyway.
- *Applications themselves* will consist of objects interacting with each other. All will be part of class hierarchies, inheriting behaviour from their parents. The application may choose to inherit from a *fixed* class hierarchy, or it may choose

to provide feedback to the class hierarchy, and suggestions for its improvement.

- As described in the statistics in this project, planners must distinguish between work done in building a class hierarchy, which will be used for years to come, and work done in writing an individual application. Normally these will go hand in hand, since the best test cases for the class hierarchy are the site's own real-life applications.

See (Meyer88), and the discussions (OOPSReuse), (OOPSEmperors) See also “Design and Code Time” on page 43.

The OO Development Site

A site developing OO applications will need a certain reorganisation of resources. Development should be managed along *class* lines, the owner of a particular class being responsible for its integrity and function, and supplying its services to the other developers. (Cav90)

One can foresee a situation even in small sites where some developers might never even deal with specific applications, but rather concentrate on defining the generic objects used across the range of the site's applications (e.g. objects from a particular business domain). This would lead to two job categories:

Class library Maintainer, who would:

- build and then maintain libraries of generic objects, business related objects, etc.
- handle requests from application developers for new objects
- handle distribution of new objects, education on their behaviour
- *insulate* application developers from all sorts of things - filing systems, message handling, maybe even the environment itself

Application Developer, who would:

- handle business cases, build applications inheriting from class libraries
- request new objects from class library maintainer
- only deal with application-specific code
- might even be insulated from the environment, e.g., may only need C++ expertise, need never learn PM to any depth, simply inherit PM objects from the PM class library. Only the site's PM guru ever touches an API call. Everyone else simply derives from the guru's supplied objects.

Current attempts to reuse code involve libraries of functions (disjoint, must be explicitly called by the application, and tied to particular data structures), and cut-and-paste style standard code blocks (e.g. where the data structures change - like dialog handling etc).

With the power of OO messaging, all of this work will not be thrown away, but will be encapsulated into well behaved objects.

It is very important to note this, since many sites already have libraries of useful functions, built up over the years. Building a class library means defining objects with these functions as their internal methods, and defining a useful inheritance tree.

Encapsulating data wasn't enough to build truly reusable libraries. With the encapsulation of function, maybe that will now change.

6-Sigma

There really is only one way of re-using code - leaving the original code unchanged, untouched, most of it not even explicitly called. Software is such a delicate process that to even re-use a network of functions introduces new possibilities for error - calling them in the wrong order, in an inappropriate place, with the wrong arguments, forgetting to close files, release memory etc when finished. And certainly cut-and-paste 'reuse' introduces huge possibilities for new bugs.

OO class libraries allow you use self-contained, consistent, well-behaved objects, which initialise themselves (constructors), monitor their own interactions (messaging) and clean up after themselves (destructors). You don't worry about them. You just use them.

And so, when a library of objects is thoroughly debugged, once, using it in applications is an order of magnitude safer, and less likely to lead to new bugs, than any of the Procedural methods of reuse.

```
Procedural                    OO
  Debug similar objects,          Debug once,
  similar algorithm              then never touch it again.
  as debugged before.             Only use it through inheritance.
```

Hence the quality of the derived application is building on the quality of the thoroughly debugged classes it inherits.

A quality drive such as IBM's 6-Sigma programme is unlikely to achieve any results without innovations of this nature. 6-Sigma seems to demand a vast improvement in software quality, without specifying any real-life tools for achieving this.

Only a paradigm shift beyond Procedural programming could possibly make the required difference.

Chapter 8. Summary of Technical Work

The old program:

- fill in forms on PWS, error-check, store on disk **(6 KLOC)**.
- send to hosts for updating.

The new program:

- fill in forms on PWS, error check, store on disk **(1 KLOC, and an order of magnitude more flexible)**.
- for reasons of time, the host section was not re-implemented.

To develop this, I built a generic form handling library **(4 KLOC)**, reusable in any application, containing the classes Form, FormField, FormDlg, FormListDlg and File as described above.

However these findings are interpreted (see Chapter 3, “Management Summary and Conclusions” on page 5), there can be no doubt that the application of Object-Oriented techniques had a dramatic impact on this project.

The claims of OO regarding powerful new forms of reuse and abstraction have been totally substantiated in this instance. This has *not* been an academic exercise. This code was really written, and it works. Behind these figures is the application of a technology that is a genuine and definite step forward.

Procedural has not been abandoned. It has been built upon.

Appendix A. References

(And90) 'A Growth Industry'
Jim Anderson
Smalltalk/V Scoop newsletter Vol.III No.1 February 1990
Digitalk Inc

(Cav90) 'Practically O-O'
Cathy Cavendish, Stacey Ramos, R.J.(Bob) Torres
IBM Corp, ASD Software Development Lab, Roanoke, Texas
request-able on VNet via:
'Request OOPASKEL from CathyC at DalHQIC1'

(Cox87) 'Object Oriented Programming - An Evolutionary Approach'
Brad J. Cox
Addison-Wesley Publishing Company, 1987

(Cox90) 'There *is* a Silver Bullet'
Brad J.Cox
'Byte' magazine, Oct 1990

quotes from the internal C-C++ FORUM:
(C++Lew90) Rhys Lewis, 17-18 May 1990, archived in C-C++ FORUM902
(C++VanC90) David Van Camp, 17 May 1990, FORUM902

(Eck89) 'Using C++'
Bruce Eckel
Osborne McGraw-Hill
ISBN 0-07-881522-3

(Gates89) Bill Gates
quoted in fall 1989 issue of IBM Personal Systems Developer
and in Smalltalk/V PM Tutorial and Programming Handbook
Digitalk Inc, 1989

(Gates91) Bill Gates
quoted in 'Computerworld' magazine, February 4, 1991

(Goguen87) 'Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics'
Joseph A.Goguen, José Meseguer
Computer Systems series, MIT Press, 1987

(Hardin) contribution to comp.lang.c++ on InterNet
sometime in late 1989 or in 1990
by John Hardin
hardin@hpindgh.hp.com
this discussion is available internally as LANGC++ DIGEST

(Meyer88) 'Object Oriented Software Construction'
Bertrand Meyer
Prentice Hall
ISBN 0-13-629049-3

(NWD90) The New World Demonstration
by UK Technical Support and UK IIS
NWDEMO on the UKSAA and IBMSAA tools disks

The internal OOPS FORUM proved a source of some very eloquent quotes:

(OOPSBoaz90) Wade R.Boaz, 21 Mar 1990, archived in OOPS FORUM902
(OOPSDeNat89a) Rick DeNatale, 6 Nov 1989, FORUM901
(OOPSDeNat89b) Rick DeNatale, 13 Dec 1989, FORUM901
(OOPSEmperors) 'Emperors Clothes' discussion, Dec 1989, FORUM901
(OOPSReuse) 'Reuse.... How do YOU do it??' discussion, Nov 1989, FORUM901
(OOPSZimm90) Douglas Zimmerman, 14 Feb 1990, FORUM902

(Pen90) 'The Emperor's New Mind'
Roger Penrose
Oxford University Press, 1990
ISBN 0-09-977170-5

(RISKSLeich90) contribution to RISKS forum on InterNet
Jerry Leichter
leichter@lrw.com
9 Sep 1990

(Strou86) 'The C++ Programming Language'
Bjarne Stroustrup
Addison-Wesley, 1986

(Strou87) 'Multiple Inheritance for C++'
Bjarne Stroustrup
EUUG Conference, Helsinki, 1987

(Strou88) 'What is Object-Oriented Programming?'
Bjarne Stroustrup
IEEE Software, May 1988

(Tash90) Jeff Tash
quoted in 'Computerworld' magazine, November 5, 1990

Appendix B. C and C++ Code Fragments

Warning: this appendix is highly technical. No attempt is made to explain the C++ syntax, but it should be somewhat *comprehensible to a C programmer.*

Procedural v OO

Initialisation / Cleanup

OO grants data structures a degree of integrity they never had before C++ provides 'constructors', which are *always* called when an object is used, and 'destructors' which are always called when the object goes out of scope.

For example, the constructor of the File object does the DosOpen, the destructor does the DosClose. The programmer just uses the object, confident that it is taking care of itself. An entire category of programming errors is thus eliminated.

C code:

```
static File file;


{
 FileOpen ( file, "sales.customers.list" );
     // opens the physical file, does initialisation, etc

 FileWrite ( file, newcustomer );

 FileClose ( file );
     // closes the physical file, etc
}
```

C++ code:

```
static File file ( "sales.customers.list" );


{
 file.Write ( newcustomer );
}
```

ADAM-Procedural v the Forms class library and ADAM-OO

The code in both has been tidied up a little to cut out unnecessary lines, but *no* improvements have been made. This is what the original of both looks like:

definition of order form

C code:

```
typedef struct adamstruct
{
                            /*   Offsets from zero      */
                            /*  Decimal  Hexadecimal    */
  char sysdate[6];             /*     -  5       -  5      */
  char docso[3];               /*    6-  8      6-  8      */
  char aaso[3];                /*    9- 11      9-  b      */
  char userid[7];              /*   12- 18      c- 12      */
  char custref[8];             /*   19- 26     13- 1a      */
  char ordd[6];                /*   27- 32     1b- 2       */
  char toa[1];                 /*   33        21          */
  char pcmstxt1[34];           /*   34- 67     22- 43      */
  char pcmstxt2[34];           /*   78-1 1     44- 65      */
  char amc[1];                 /*  1 2        66          */
  char mrc[1];                 /*  1 3        67          */
  char saleman2[6];            /*  1 4-1 9     68- 6d      */
  char sbo2[7];                /*  11 -116     6e- 74      */
  char mes[1];                 /*  117        75          */
  char sw[1];                  /*  118        76          */
  char qty[2];                 /*  119-12      77- 78      */
  char acode[1];               /*  121        79          */
  char crad[6];                /*  122-127     7a- 7f      */
  char custlink[6];            /*  128-133     8 - 85      */
  char linkno[5];              /*  134-138     86- 8a      */
  char supnumb[6];             /*  139-144     8b- 9       */
  char toc[1];                 /*  145        91          */
  char orn[6];                 /*  146-151     92- 97      */
  char newmod[3];              /*  152-154     98- 9a      */
  char nmtype[4];              /*  155-158     9b- 9e      */
  char moc[1];                 /*  159        9f          */
  char oneshot[2];             /*  16 -161     a - a1      */
  char serial[7];              /*  162-168     a2- a8      */
  char docn[5];                /*  169-173     a9- ad      */
  char pcmsref[5];             /*  174-178     ae- b2      */
  char custnam[3 ];            /*  179-2 8     b3- d       */
  char remark1[21];            /*  2 9-229     d1- e5      */
  char remark2[21];            /*  23 -25      e6- fa      */
```

```
  char status[1];              /*  251        fb         */
  char cradmgt[1];             /*  252        fc         */
  char linkmgt[1];             /*  253        fd         */
  char entity[4];              /*  254-257    fe-1 1      */
  char fetqty[12][2];           /*  258-281   1 2-119      */
  char tla[2];                /*  281-283   11a-11b      */
  char thrpart[6];             /*  284-289   11c-121      */
  char maint[1];              /*  29        122         */
  char custo[6];              /*  291-296   123-128      */
  char pcmscat[1];             /*  297       129         */
  char mtype[4];              /*  298-3 1   12a-12d      */
  char model[3];              /*  3 2-3 4   12e-13       */
  char feat[12][6];            /*  3 5-376   131-178      */
  char inputfile[8];           /*  377-384   179-18       */
  char sys[5];                /*  385-389   181-185      */
  char cpu[1];                /*  39        186         */
} adamrec;
```

```
// all are made into character strings for interaction with dialog.
//
// changing this struct requires re-compilation of the whole system.
//
// this is '#include'd into every 'C' source file that uses it
```

C++ code:

```
AdministratorForm :: AdministratorForm()
{
  Add ( 8, RestrictedText, "input file"          );
  Add (   Date        , "system date"         );
  Add ( 3, RestrictedText, "document source"      );
  Add ( 3, RestrictedText, "DSC WTAAS source code" );
}

CustomerForm :: CustomerForm()
{
  Add ( 6, NumericRestrictedText, "customer number"                 );
  Add ( 6, NumericRestrictedText, "3rd party customer number, for PCMS" );
  Add (   Date              , "date of order"                   );
  Add (   MarketingChannel    , "alternative marketing channel"      );
  Add ( 6, NumericRestrictedText, "salesman's agent number"          );
  Add (   Text              , "sales branch office"             );
  Add ( 1, RestrictedText      , "marketing response code"           );
  Add ( 2, NumericRestrictedText, "months for Term Lease Agreement"    );
  Add (   Text              , "comments for PCMS"               );
}

PartsForm :: PartsForm()
{
  Add (   YesNo             , "CPU indicator"                    );
  Add ( 4, RestrictedText      , "entity scheduling"                 );
  Add (   YesNo             , "Miscellaneous Equipment Specification" );
  Add (   AnalysisCode       , "analysis code"                    );
  Add ( 4, NumericRestrictedText, "machine type"                     );
  Add ( 3, RestrictedText      , "machine model"                    );
  Add (   NaturalNumber       , "quantity"                         );
  Add (   RealNumber         , "percentage discount"               );
  Add (   TOM               , "type of maintenance"               );
  Add ( 6, RestrictedText      , "supplement number"                 );
  Add (   TOC               , "type of contract"                 );
}
```

```
OrderForm :: OrderForm()
{
 // construct some (unnamed) component forms and eat them
  Eat ( Administratorform(), Extend );
  Eat ( Customerform()    , Extend );
  Eat ( Partsform()       , Extend );

 // and a few more fields
  Add (   Text              , "comments for WTAAS (1)" );
  Add (   Text              , "comments for WTAAS (2)" );
  Add ( 4, NumericRestrictedText, "new machine type"      );
  Add ( 3, RestrictedText      , "new machine model"     );
  Add (   MOC               , "MES Order Class"       );
  Add (   SerialNumber        , "machine serial number"  );
}
```

```
// all are real objects - numbers or strings or structs.
//
// all these Forms and components can be chopped and changed
// and Eaten at will, even dynamically,
// without affecting any of the objects that interact with them.
//
// these defns are not '#include'd anywhere, but rather
// 'link'ed with the object files that use them.
```

initialisation of dialog

C code:

```
dispstd ( HWND hwnd )
{
  WinSetDlgItemText ( hwnd, EF_INPFILE , inputfil );
  WinSetDlgItemText ( hwnd, EF_DSCAA   , aaso    );
  WinSetDlgItemText ( hwnd, EF_DATE    , sysdate  );
  WinSetDlgItemText ( hwnd, EF_DOCSO   , docso    );
  WinSetDlgItemText ( hwnd, EF_CUSTNO  , custo    );
  WinSetDlgItemText ( hwnd, EF_3PARTY  , thrpart  );
  WinSetDlgItemText ( hwnd, EF_DORDER  , ordd     );
  WinSetDlgItemText ( hwnd, EF_AMC     , &amc     );
  WinSetDlgItemText ( hwnd, EF_ISR2    , saleman2 );
  WinSetDlgItemText ( hwnd, EF_SBO2    , sb 2     );
  WinSetDlgItemText ( hwnd, EF_MRC     , &mrc     );
  WinSetDlgItemText ( hwnd, EF_TLAMTS  , tla      );
  WinSetDlgItemText ( hwnd, EF_PCMSTXT , pcmstxt1 );
  WinSetDlgItemText ( hwnd, EF_CRAD    , crad     );
  WinSetDlgItemText ( hwnd, EF_LINKO   , custlink );
  WinSetDlgItemText ( hwnd, EF_SYSNO   , sysno    );
  WinSetDlgItemText ( hwnd, EF_CUSTREF , custref  );
  WinSetDlgItemText ( hwnd, EF_CPU     , &cpu     );
  WinSetDlgItemText ( hwnd, EF_ENTITY  , entity   );
  WinSetDlgItemText ( hwnd, EF_MES     , &mes     );
  WinSetDlgItemText ( hwnd, EF_ANALSYS , &acode   );
  WinSetDlgItemText ( hwnd, EF_TYPE    , mtype    );
  WinSetDlgItemText ( hwnd, EF_MODEL   , model    );
  itoa ( qty, qtystr, 1  );
  WinSetDlgItemText ( hwnd, EF_QTY     , qtystr   );
  WinSetDlgItemText ( hwnd, EF_DISCNT  , ocd      );
  WinSetDlgItemText ( hwnd, EF_TOM     , &maint   );
  WinSetDlgItemText ( hwnd, EF_SUPPNO  , supnumb  );
  WinSetDlgItemText ( hwnd, EF_TOC     , &toc     );
}
```

// just one of many such lists
// cluttering up the ADAM-Procedural application
// and unusable in anything other than ADAM

C++ code:

this is from the generic Forms library:

```
void FormDlg :: _FAR Show ( ShowState s )   // when I become visible ..
{

  UpdateFields();             // .. update all my entryfields from the Form

  AppModalDlg :: Show ( s );    // ( and then call my parent's Show method,
}                            //   whatever that does )



FormDlg :: UpdateFields()
{
  for ( pform->Reset(); pform->Next(); )     // run thru the Form ..
  {
    FormFieldLock l ( * pform );

    WinSetDlgItemText ( Handle(), l.FF()->ID(), l.FF()->GetString() );
  }
}                                 // .. l.FF() is 'Get Next FormField'
```

The ADAM-OO application itself has *no* work to do in initialising the dialog.

error-checking of the 'AMC' field

(the AMC field is a single digit field, which may be blank,5,6,7,G or B)

C code:

```
VOID readscr1(HWND hwnd)                  // read the variable
{
   ...
  WinQueryDlgItemText(hwnd,EF_AMC,MAXCHARS,tempstr);
  amc = tempstr[ ];
   ...
}

int verfix()                             // verify it
{
   ...
  if ((amc == SPACE) || (amc == NULL))
    acmp = TRUE;

  if (toupper(amc) == AMCB)
    bchk = TRUE;

  if (toupper(amc) == AMCG)
    gchk = TRUE;

  if (acmp || bchk || gchk )
    ochk = TRUE;

  if (!ochk)
  {
   if (amc < '5' || amc > '7')
   {
    rc = AMCERR;
    amcerr = TRUE;
   }
  }
   ...
}

USHORT ordmsg(HWND hwnd, int vrc)         // and return to the dialog
{                                      // with any errors
   ...
  case AMCERR:
   faterr(hwnd, "Invalid AMC");
   return(EF_AMC);
   break;
   ...
}
```

```
// we would like to trap bad input to this field,
// but how can we do this in 'C' ?
// we can't trap all input in the dialog, and call
// 'the relevant field's validcharacter routine'
// (unless we invent our own OO messaging system,
//  or introduce some appalling case statement ).

// and it is too much trouble to define a window class for
// all our fields, so we don't bother.
// we validate it when the order is being sent.
// bad input sits in the dialog and nothing is flagged until
// the user tries to exit the whole screen

// also, AMC data gets hard-coded all over the application
```

C++ code:

```
class MarketingChannel : public SingleDigit
{
public:
  MarketingChannel ( Type type, Name name ) : ( type, name, " 567gGbB" ) {}
};
```

```
// somewhere in FormDlg, every keystroke gets sent to the FormField's
// isValidChar method, and SingleDigit is designed to store
// a string of 'valid' characters, so I can just inherit from SingleDigit
// and thereby inherit its behaviour

// I now just use MarketingChannels. Every time they appear
// in a Form, they will filter out the bad characters.
// all keystrokes are checked immediately,
// and information about MarketingChannel is all localised in the tiny
// piece of code above

// note that this system requires FormDlg to be able to 'call functions'
// in objects whose existence and properties it is not really aware of.
// FormDlg just sends out the 'isValidChar' message. The system maps it
// to some real method inside the field
```

New World Demo v the Forms class library

The New World Demonstration is a partial demonstration of order entry and cooperative processing in a CUA PC environment (NWD90). I use it as an example of Direct Manipulation as implemented in C.

'Eat'ing the customer information into the order form

C code:

// in the window procedure for the order form dialog window:

```
MRESULT wporderform ( HWND hWnd, USHORT Message, MPARAM mp1, MPARAM mp2 )
{
   ....
  case WM_USER+6:      /* Accept customer data from customer list */

    pPrivate=QueryMem(hWnd,ALLOC_PRIVATE);
    strcpy(pPrivate->customer,PVOIDFROMMP(mp1));

    if (pPrivate->customer[ ]!=NULL)
    {
      strncpy(pPrivate->account,pPrivate->customer,6);
      pPrivate->account[6]=NULL;

      strncpy(pPrivate->name,pPrivate->customer+7,32);
      pPrivate->name[32]=NULL;

      strncpy(pPrivate->addr,pPrivate->customer+4 ,6 );
      pPrivate->addr[6 ]=NULL;

      strncpy(pPrivate->phone,pPrivate->customer+1  ,12);
      pPrivate->phone[12]=NULL;

      pPrivate->creditlimit=atol(pPrivate->customer+112);

      WinSetWindowText(pPrivate->hwndent4,pPrivate->account);
      WinSetWindowText(pPrivate->hwndent3,pPrivate->name);
      WinSetWindowText(pPrivate->hwndent2,pPrivate->addr);
      WinSetWindowText(pPrivate->hwndent1,pPrivate->phone);

      WinPostMsg(pPrivate->hwndparent,WM_USER+97,(MPARAM)1,(MPARAM)TRUE);
      WinPostMsg(pPrivate->hwndparent,WM_USER+98,NULL,NULL);
    }
    break;
  ....
}
```

```
        // and a similar scheme to handle part data being dropped onto
        // the order form, etc ...
```

C++ code:

```
  // OrderFormDlg gets the message 'this thing has dropped on you'
  // from DMMonitor.
  // this is how it handles it:


OrderFormDlg :: _FAR DroppedOnYou ( FormDlg & formdlg )
{
  Eat ( formdlg );
}


      // Eats it, whatever it is, and however many fields it has.
      // to be precise, my form Eats formdlg's form,
      // and all changes are automatically updated to my entryfields
      // ( WinSetWindowText's are generated whenever
      //   the internal Form changes )
      //
      // this can handle Eating part data, other order forms,
      // standard configurations, anything - it doesn't matter.
      // just call the generic library to Eat it, whatever it is.
```

Appendix C. Layout of C++ files

As a supplement to the class hierarchy trees above, this is how the Forms library and the application live together.

.h are .hxx header files, containing class declarations
.c are .cxx files, containing the definitions of class methods

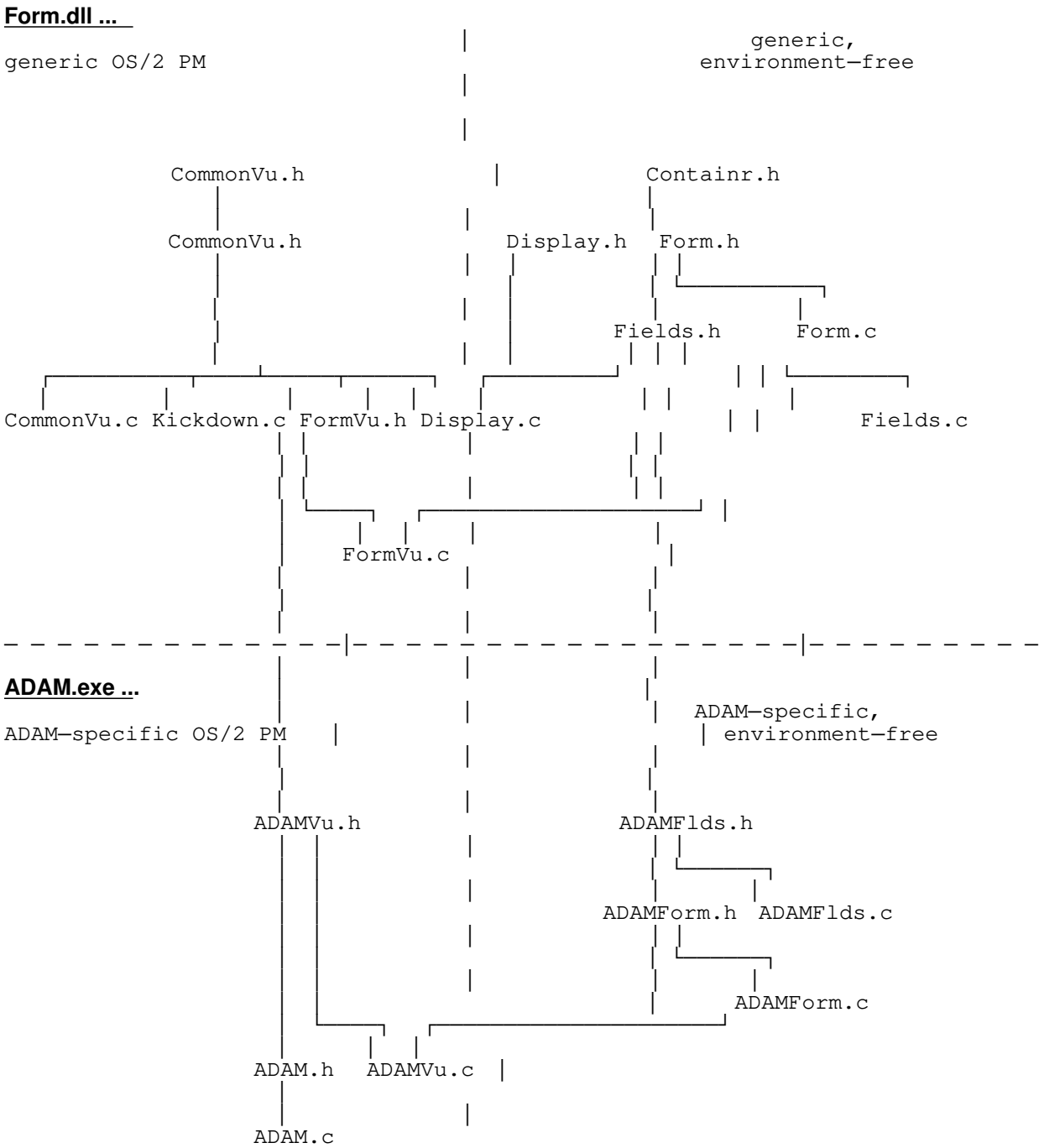


Figure 15. C++ source and header files relationships

Glossary

6-Sigma. IBM's company-wide quality drive

AAS. see WTAAS

abstract class. a class that is not designed to be used, but only inherited from. It is a class that does not actually 'work'. Its methods are typically virtual functions.

ADAM. Administrative DSC Application Model - the local IISL application chosen for re-implementation using OO techniques

Amodality. the removal of the need to complete one step of a transaction or dialog before doing another piece of work

API. Application Programming Interface - operating system functions that can be mapped to high-level language function calls. e.g. in OS/2 all system functions can be called from the C language, using C data types.

AT&T. American Telephone and Telegraph Company, parent company of Bell Labs

Bell Labs. AT&T Bell Laboratories - the inventors of UNIX, C, C++ and Plan 9

C++. The object-oriented language used to implement this project. Invented at Bell Labs, C++ is an object-oriented superset of C.

cfront. the public-domain algorithm for translating C++ code into C code. Developed at AT&T Bell Labs, this is the basic algorithm used by Glockenspiel in their C++ compiler.

class. the definition of an object-oriented data type. objects themselves (program variables) represent instances of a class.

class hierarchy. orders the different classes into parents and children. illustrates the inheritance of data and behaviour down through the generations

class library. a collection of one or more class hierarchies containing useful classes which can be used in applications, either directly or through inheritance

CommonView. a class library of objects for programming in windowed environments. Provided by Glockenspiel Ltd.

Concurrency. the ability to view and work with several parts of an application at the same time

Container. a class library of objects for object storage. Provided by Glockenspiel Ltd.

CUA. Common User Access - IBM's definition of a consistent user interface within and across applications - for everything from PWS's running GUI's to dumb terminals running host applications.

DLG. Dialog - a (windowed) data entry screen in PM

DLL. Dynamically Linked Library - denotes run-time library file in OS/2

encapsulation. data is bundled with the functions (or methods) that manipulate it. The data cannot be manipulated except via these 'access functions' (see “The Object-Oriented Paradigm” on page 15).

EVE. The hypothetical 'next' application after ADAM. Represents either ADAM version 2, or some roughly similar data capture program in the same environment.

EXE. denotes executable program file in OS/2

form. a list of fields to be filled in by the user

Glockenspiel. Glockenspiel Ltd, Dublin, Republic of Ireland, suppliers of the C++ compiler used in this project

GUI. Graphical User Interface - the graphics-screen, window and icon based, 'desktop'-type user interface which has become the PWS industry standard, e.g. Mac, PM, Windows, NeXT, etc

IISL. IBM Ireland Information Services Ltd, Dublin, Republic of Ireland (my employers)

inheritance. a class may inherit the behaviour of another class (which is then referred to as a parent class). This inheritance of data and function is (should be) handled in such a way that the properly debugged, good behaviour of the parent is replicated in the child (see “The Object-Oriented Paradigm” on page 15).

IS. Information Systems - IBM Division responsible for internal software

KLOC. Thousand LOC

LOC. Lines of Code. Normally refers to executable code only

MCU PWS group. Multi Country Unit PWS group - the group within IISL that developed the original, Procedural ADAM application (I was not involved with this).

multiple inheritance. a class may inherit behaviour from multiple parents

object. an instance of a class ('type').

Object-Oriented Programming. a program models the environment in which it executes, in such a way as to support algorithms to solve many different problems in the same environment. The data contains its own intelligence, and many of the subroutines in the system are not explicitly called, but are secretly called by the data itself when it is being manipulated. Many techniques work together to support this model - the essential ones being encapsulation, polymorphism and inheritance.

OO. see Object-Oriented Programming

PM. Presentation Manager, the windowed user interface for the PC operating system OS/2

polymorphism. the same message can be sent to different objects, each of which may have its own customised response to the message (i.e. the same 'virtual' function can be called across a range of objects, causing a different real function to be run in each one (see “The Object-Oriented Paradigm” on page 15))

Procedural programming. a program is the implementation of an algorithm to solve a particular problem. The actions in the program consist of passing data into subroutines, executing the subroutines, and recovering data afterwards.

PWS. Programmable Work Station - a single-user machine which can run its own software on its own chip. Includes PCs and workstations, but not dumb terminals.

Ring. a class in the Container library. A Ring is a general list of any type of objects

single inheritance. each class may have only one parent

struct. in C, a record, or data structure, or list of fields, or composition of primitive data types

structured programming. techniques to impose structure on Procedural programs, in particular by regulating the flow of control. Typically, structured design will specify that a program should be broken down into subroutines, each with a well-defined entry and exit point, and well-defined pre and post-conditions.

subroutine. or function, or procedure - a block of code separated from the rest of the program to perform a specific function

virtual function, or virtual method. a method that is declared as one of the methods of a class, but that has no actual code to be executed. The method can be called for the class, but the actual code to be executed is only provided in derived classes. It is a function that 'does not work'. The real function call may not be resolved until run-time.

Window Procedure. in PM, the function that receives all messages for that window. In the C API, the Window Procedure must sort the messages and redistribute them to specific subroutines

WTAAS. World Trade Advanced Administration System - the host application that ADAM communicates with