Paper AS08

SAS/AF and Software Prototyping – Having your PIE and eating it!

Edward Foster, Oxford Pharmaceutical Sciences (OPS), Oxford, UK

ABSTRACT

All good quality software projects must clearly map business process (how work actually gets done) to a set of well defined requirements produced by the users as part of the requirements gathering exercise. Clear requirements, devoid of ambiguous statements, will mean that the end product is more likely to conform to the original specification for the system as developers should not be making any 'leaps of faith' in assuming what the customer wants.

One way to reduce the risk of errors arising from requirements is to adopt a prototyping methodology either as part of the requirements gathering stage or as part of the development process itself. SAS/AF® can be used as a prototyping tool because its simple interface and standard objects make it quick and easy to build a prototype for demonstration, which can be used to clarify and firm up what is actually required from a system. This paper will describe how a prototyping methodology was applied to a development project within OPS and how SAS/AF was used to support this.

INTRODUCTION

Well written user requirements are the key to producing quality software. A developer working from a clear set of requirements can more easily mould them into functional and design documents. User requirements tend at first to be wordy documents, but if they can be 'atomised' or broken into constituent statements they make it easier to incorporate into a traceability matrix which can then be used to define user acceptance tests (UAT). The importance of getting it right at the start is demonstrated by the relative costs of fixing errors as time progresses through the development and maintenance cycle (F1).

F1. Relative Costs of Errors during Software Development and Maintenance Cycle

Stage	Relative Cost of Repair
Requirements	1
Design	5
Implementation	10
Unit Test	20
Acceptance Test	50
Maintenance	200

There are three questions (at least!), even with very thorough requirements gathering exercises, that should be considered as part of any risk assessment before proceeding with development.

1. Are there any missing/hidden requirements?
2. Does the customer understand the business process enough to give a full and unambiguous account?
3. Is the business process underpinning the software undergoing any significant change?

Missing or hidden requirements are always a risk but can be reduced through a thorough requirements gathering stage. Assessing whether the customer fully understands the business process is difficult to resolve, especially if the developer is external to the business, because it is not known whether a person understands the business process sufficiently. One way to reduce this risk is to involve as many appropriate people as possible to allow the problem to be approached from many angles. By triangulating the responses in this way, it should be possible to produce a tight set of requirements that can be signed off by the customer. However, even if the answer is yes to question 2, it should always be borne in mind that if the underlying business process is undergoing change (or is about to), then unless the company has a very effective change management strategy, it is likely that most of the potential users may only partially understand what changes are taking place and the impact these will have on the way they do their work. Amongst a number of other scenarios, this could lead to requirements being defined at the start of the project which may no longer be relevant at the end of development.

One approach to reduce these risks is to adopt a software prototyping methodology, whereby rough demonstrations of the system can be used to develop a better understanding of what is required from both the developer and customer point of view. It also allows the developer and customer to try various scenarios to solve a problem which may not have been clearly defined at the start.

Improvements incorporated into SAS/AF from version 8 onwards, like an improved GUI for setting and linking object attributes reduce the amount of code needed to produce a working SAS frame and hence the time taken to build a piece of software. The use of dot notation into version 8 also makes SCL easier to read and helps improve debugging at compilation time of the frame.

A prototyping methodology was used during the software development project to build a management tool for the OPS Clinical Research Organisation (CRO). The tool, named Project Information Exchange, was built to pull together various sources of project management data to produce weekly and monthly reports for project and operational managers. This type of methodology was used as the business process which the software would support was undergoing change through our iterative process development. Initial requirements for some areas of the system were therefore hard to derive

PROJECT INFORMATION EXCHANGE (PIE) 

Project Information Exchange (or PIE as it was dubbed) was a software development project undertaken at OPS to link and pull together disparate sources of client project and employee information into one database structure.

This information and data was mainly Excel based and comprised of;

- Timesheet Information – Employees accounted for time spent on projects, deliverables and tasks via an Excel spreadsheet issued every month.
- Budget Information – Maintained by project managers who on a project by project basis, noted the number of units for any given deliverable (such as a output type) and the percent complete for these deliverables.
- Pricing Information – The OPS pricing models. These had to be kept secure as this is confidential information. This would link to both the employee information and project deliverables.
- Employee Information – This included information such as the assignment of a unique employee ID number, periods of employment and job titles within these periods.

This database could then be used to produce weekly and monthly reports in SAS that were distributed via the intranet to employees.

SAS was chosen as the development tool for a number of reasons including;

- The main body of programming expertise within the company was in using SAS
    - By using SAS/AF, the SAS skills of our programmers could be extended
    - Testing, maintenance and further development of the system could be shared amongst the SAS programmers within the CRO, especially since it was decided to store key process code within SAS macros external to the application rather than embed it with the SCL.
- SAS/AF was a cost effective solution because the product was already licensed and installed on a number of users' machines.
- Quick results (demonstrating that all important Return on Investment) can be obtained using SAS/AF.

SOFTWARE PROTOTYPING

It became clear during the requirements gathering phase that the business process of collecting, storing and maintaining project information was changing. One change was that in a growing company the responsibility for the accuracy and timeliness of this information was being devolved to project managers rather than being controlled centrally.

This meant that trying to create a firm set of requirements from the start would be like building a house on shifting sands. Because of this it was decided to modularise the components of the system (data load, code list management and reporting) in order to simplify development and then implement a prototyping methodology to produce the software.

The classical Software Development Life Cycle (SDLC) is well understood and comprises the following stages;

- Gather Requirements

- Plan (design system)
- Code (develop system)
- Test (System testing and UAT)
- Release (package, sign off and release final production version)
- Maintenance (dependent upon the relationship with the customer)

The SDLC forms a fairly linear pattern of development from the start to the finish of a project. While there are feedback mechanisms built into each stage of the life-cycle these are generally implemented as part of a checking exercise or adding small incremental design changes rather than a wholesale change. This is especially true as the project progresses further through the cycle. The creation of a usable and quality product depends greatly on getting it right at the start with the requirements phase.

Software prototyping can help in those situations where the requirements phase is especially difficult. Building a working model can act as a discussion point for various aspects of the system and can be seen as a knowledge generation exercise, promoting a shared understanding of the problem between both the developer and the customer. Not only this but, in seeing a working model, it can help firm up what is actually required even on those 'shifting sands' because it can be easier to assess the impact of change against a more tangible object.

This means that prototyping can be described as a requirements risk-reduction exercise by helping in the following areas;

- Requirements Elicitation – Users can experiment with the system and see how it could support their work
- Requirements Validation – Errors or omissions in the requirements may be easier to spot.

Two types of prototyping can be employed: evolutionary or 'throwaway'. See F2 for descriptions of the methodologies.

F2. Prototyping Methodology Types

Prototyping Method	Usage	Objectives of Prototyping Exercise
Evolutionary (Open) Prototyping	Prototype built as part of requirements analysis. The prototype is then used and developed in an iterative develop-review cycle to eventually form the finished product.	To deliver a working system to the users. The development starts with the areas which are best understood.
Throwaway (Closed) Prototyping	Prototype built as a rough demonstration of a section of the system. The prototype is used as a discussion point to finalise the requirements by ironing out any remaining issues. The prototype is discarded at the end as it will not form part of the final product. The prototype is not necessarily developed using the same software packages or tools as the final product, but instead may use a dedicated prototyping package, such as a 4G language.	To validate or derive the system requirements. The areas which are most poorly understood are used first to create a prototype.

Another useful aspect to prototyping (as is demonstrated in the later examples) is that sometimes it can give the developer the freedom to try various ways to solve the same problem. Trying different methods allows the developer to work out which methodology will fit best with the system. This process can be seen as a further knowledge gathering exercise on the part of the developer. This can better inform the development project as a whole and can also feed back into the requirements process.

It was decided to approach the challenge posed by PIE using an evolutionary prototyping approach. This was deemed the better of the two prototyping approaches because it was believed that it was more cost effective to use the prototypes created as a template for final development, especially as the final product was to be built using the same software package as the prototypes themselves.

DECISION FACTORS

It should be noted that while prototyping is a useful tool, it is not appropriate in all situations, for example it may not be appropriate to build a prototype if a system does have something clearly tangible (like a GUI interface) to demonstrate. Sometimes knowing when to stop the iterative development can also be a difficult thing when the panacea of a perfect system appears to be just an iteration more away! The complexity of the system needs to be the

basis of the decision. A system that will require thousands of lines of code to be written before a working model is produced is probably not a good candidate for prototyping, especially if a throwaway methodology is used! If the complexity can be partitioned or modularised, then it may still be possible to prototype portions of the software.

PIE was potentially a complex application but it had clearly defined areas of functionality (data load, code list management and reporting) which could be modularised. While all these areas are linked, full scale development of any one area was not required for another one to work.

Another factor to bear in mind is the customer characteristics: Is the customer committed to reviewing and refining the prototype? Is the customer capable of making requirements decisions at all and also in a timely manner? It would soon become obvious if the answer to either of the above questions is no as the project is likely to come to a rapid halt. For the PIE project, the system owner was heavily involved in refinement of the prototypes that were produced. Weekly meetings were held as part of the project management and time was put aside to evaluate any new prototype functionality.

SAS/AF AND PROTOTYPING

SAS/AF may seem like a 'blast from the past' for some developers because SAS has moved into other application development areas, incorporating web development in their SAS/Intrnet and Apps Dev Studio packages. However, SAS/AF is still a viable solution for many smaller companies where it proves a cost effective development environment. This is certainly true in the pharmaceutical industry where the near ubiquity of SAS in biometrics means it makes sense to create applications that can directly create and access SAS data, with many companies building 'toolboxes' of small SAS/AF tools to do various tasks with clinical data, ranging from data cleaning environments to drug and adverse event coding.

SAS/AF uses Screen Control Language (SCL) code to provide functionality to the user. In many ways SCL can (and perhaps should) be seen as an extension of the SAS programmer's skill set as SCL can be used in a number of useful situations in datastep code as well. This means that investment in training employees to work with SCL and SAS/AF can be seen as worthwhile in a company which specialises in its use of SAS/BASE reporting and statistical elements as well as other related SAS technologies (such as SAS/GRAPH, SAS/STAT, ODS etc).

The ease with which a working SAS/AF frame can be created shows that SAS/AF can be an effective tool for prototyping. Improvements in version 8 like dot notation in SCL code, object data models and the improved GUI for setting and linking attributes make working with AF much easier compared with the 6.12 implementation. The PIE application was developed using SAS/AF with version 8.2. Conversion to use under version 9.1 is the next stage for this application to take advantage of all the new features, such as improved reporting features in ODS and the use of PROC DOCUMENT to generate multiple report formats. We are also looking at the SAS Open Metadata Architecture and the Management Console to see if we can create a company wide metadata store to link all our applications.

AF also allows you to encapsulate functionality in user defined objects and methods. Many companies already use the notion of stores for SAS code, such as generic macros. These stores can be extended to contain these methods to allow for easy retrieval for future prototyping exercises. Re-use of code is a key component of prototyping that relies on rapid development practices. SAS catalogs with appropriate descriptions are ideal for storing libraries of such code.

PROTOTYPING IN ACTION

One of the main aspects of PIE was the integration with existing Excel spreadsheets. One of the main strengths of SAS is any source of data, with a bit of work, can be imported into a SAS dataset for the user.

With MS Office products (especially Excel) there are a number of ways to import data into SAS. As a prototyping methodology was being adopted, it was an opportunity to try various methods and assess their impact on the system.

IMPORT/EXPORT FACILITY

Perhaps the easiest option is to use PROC IMPORT and EXPORT which are available as part of the SAS/BASE package (Example F3).

Whilst this method is simple, it did not allow us to customise and accurately predict the results of any import procedure as can be seen from the example (F3), Excel cell A4 has not been imported properly as a numeric format has already been selected for this column.

It was decided that because the Excel sources would be variable in their structure, a method of import that would

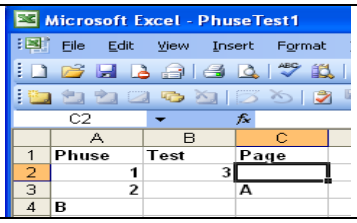
exactly define the data structure was needed.

F3. SAS IMPORT Example

This code

```
proc import
  datafile="c:\PhuseTest1.xls"
  out=Phuse1
  replace;
run;
```

On this Data



Produces

	Phuse	Test	Page
1	1	3	
2	2	.	A
3	.	.	

DYNAMIC DATA EXCHANGE (DDE)

This method is the 'old faithful' in data transfer between SAS and Excel. With a few simple statements, an Excel sheet can filled and imported in a controlled and customised manner. DDE uses Excel version 4 Macro Language (X4ML) commands that are still supported by Microsoft products, even though they now use VBA as their own macro language. DDE commands can be used to format the Excel sheet in many ways – for example, change background colours, merge cells, change the font. The import process can be controlled as easily as any other import using INFILE statements, once an active Excel session is created.

For example the following code, whilst more complex than the previous example, can import the same data in a predictable manner

F4. DDE Import Example

This code

```
/*SET OPTIONS AND CREATE CMDS FILENAME*/
options noxwait xsync;
filename cmds dde "Excel|system";

/*START EXCEL*/
data _null_;
        length fid rc start stop time 8;
        fid=fopen('cmds','s');
        if fid le 0 then do;
                rc=system('start excel.exe');
                start=datetime();
                stop=start+30;
                do while (fid le 0);
                        fid=fopen('cmds','s');
                        time=datetime();
                        if time ge stop then fid=1;
                end;
        end;
        rc=fclose(fid);
run;

data _null_;
        file cmds;
/*STOP ANY EXCEL ERROR OR WARNING MESSAGES*/
        put '[error(false)]';
/*OPEN THE EXCEL SHEET*/
        put '[open("c:\phuseTest1.xls")]';
run;


/*SET UP FILENAME FOR DATA AREA OF EXCEL SHEET*/
filename data dde
"Excel|[phuseTest1.XLS]SHEET1!R2C1:R2000C3";

/*IMPORT EXCEL DATA*/
data Phuse2;
    infile data dsd dlm='09'x notab missover;
    length Phuse $10 Page 8 test $1;
    input phuse $ page test $;
run;
```

Produces

	Phuse	Page	test
1	1	3	
2	2	.	A
3	B	.	

```
/*CLOSE EXCEL*/
data _null_;
        file cmds;
        put '[quit()]';
run;
```

This method was decided against as there must be an active session on the client PC for DDE to work. It was decided that we wanted a more seamless feel to the application and having command boxes and Excel appearing on screen was not acceptable, especially as the number of files that would need to be opened would be high.

OLE AUTOMATION

Using SAS/AF and SCL allowed us to use OLE automation as a method for importing data. OLE Automation allows us to create an Excel object and hence have access to methods and objects that Excel has in VBA. Not only this but we can create invisible Excel objects so the user does not see (and cannot click on!) any of the Excel sessions that are running because of the application. It was believed that this method would give the greatest degree of control over Excel and also use a more up to date technology (VBA objects rather than X4ML).

Using SCL, the code could be executed easily from within the application. However, there was a requirement to externalise load process code for maintenance purposes. To execute the code we used PROC DISPLAY. PROC DISPLAY allows you to execute catalog entries from SAS/BASE code. This posed another problem as you cannot pass parameters (such as the sheet you want to import) though PROC DISPLAY. To get round this we used macro variables to pass parameters. The macro variables would be defined before the PROC DISPLAY and would be captured in the SCL by using SYMGET (see F5)

F5. Executing SCL from BASE code

SAS Program	SCL Program

`%let message=Hello Everyone;`
 	
`proc display catalog=phuse.code.F5.scl;`
`run;`
	`INIT:`
	
	`message=symget('MESSAGE');`
	`put message=;`
	
	`return;`

The code in F6 is the OLE automation in action, used to the export SASHELP.CLASS dataset. It also produces an invisible Excel object. You can view the Excel Session in action by changing the visibility property to 'TRUE'.

This method worked well and gave a more seamless feel to the application and initial testing on small sets of data showed that it was an effective way to manipulate Excel. However, the code was complicated and it was decided that could not easily be maintained by an inexperienced SCL programmer. One of the requirements of the system was that all major load process code was to be held in standard SAS program files within macros, rather than buried in application SCL or in any other SAS catalog file. Storing code in this way makes it easier to maintain because programmers can more easily find the right piece of code when any issues arise. It was also noted that there were some performance issues in using this method to import large Excel files, which seemed to exponentially increase the load time.

OLE automation at first looked like a promising method for import and it is still since used within the PIE application for some processes. One of which was converting Excel files to a different version for import by creating invisible objects and saving the Excel files in the correct format for subsequent import (see next section).

F6. OLE Automation Export Example

SAS Program	SCL Program

`proc display`
`  cat=work.new.F5.scl;`
`run;`
	`INIT:`
	
	`/*CREATE EXCEL OBJECT AND OPEN AN INVISIBLE EXCEL SESSION*/`
	`declare object excel = _neo_ sashelp.fsp.hauto`
	`(0,'Excel.Application');`
	`declare object workbooks workbook activews cell;`
	
	`/*CREATE AN INVISIBLE EXCEL SESSION*/`
	`/*SET THIS TO TRUE IF YOU WISH TO SEE THE EXCEL CODE*/`
	`excel._setProperty ('Visible', 'FALSE');`
	`excel._getProperty ('Workbooks', workbooks);`
	
	`/*ADD A NEW WORKBOOK TO THE SESSION*/`
	`/*CREATE A OBJECT TO REFERENCE THE CURRENTLY ACTIVE SHEET*/`
	`workbooks._do('ADD');`
	`excel._getProperty('Activesheet',activews);`
	
	`/*WRITE THE DATA TO THE EXCEL DATA*/`
	`dsid=open('sashelp.class','i');`
	
	`call set(dsid);`
	`rc=fetch(dsid);`
	`nvar=attrn(dsid, 'NVARS');`
	`nobs=attrn(dsid, 'NOBS');`
	`do col=1 to nvar;`
	`  activews._compute('Cells',1,col,cell);`
	`  var=varname(dsid,col);`
	`  cell._setProperty('Value',var);`
	`end;`
	`do while (rc ne -1);`
	`   do row = 2 to nobs+1;`
	`      do col = 1 to nvar;`
	`         activews._compute('Cells',row,col,cell);;`
	`         if vartype(dsid,col) eq 'N' then`
	`            var=getvarn(dsid,col);`
	`         else var=getvarc(dsid,col);`
	`         cell._setProperty('Value',var);`
	`      end;`
	`      rc=fetch(dsid);`
	`   end;`
	`end;`
	`dsid=close(dsid);`
	
	`/*SAVE THE EXCEL FILE*/`
	`excel._getproperty('ActiveWorkbook',workbook);`
	`workbook._do ('SaveAs',"c:\Phuse Test.xls");`
	
	`return;`
	
	`TERM:`

```
  workbooks._term();
  activews._term();
  cell._term();
  excel._do ('Quit');
  excel._term();

put 'FINISHED!';

return;
```

SAS/ACCESS FOR PC FILE FORMATS (PROC ACCESS)

The final option to be tried was PROC ACCESS. PROC ACCESS can be used where 'SAS/ACCESS for PC file formats' is licensed. It works in two halves with the first creating an access descriptor that essentially describes the data source, and the second half using this access descriptor to create a SAS view of the Excel data that can be used in further data manipulation. PROC ACCESS had already been discounted at this point because it only imports Excel files up to Excel 97 (In version 8 of SAS) and PIE needed to work with Excel files created using Excel 2000 and Excel 2003. One possible solution was to use the generic Excel file format which allows for backward compatibility ('Microsoft 97 – Excel 2003 & 5.0/95' format in Excel 2003). This posed another problem as this file format greatly increases the size of the file in order to hold all the information required to make it compatible with Excel 97.

Initial testing of the import of the Excel data showed that by using macros it was easy to create both the access descriptor and view. This meant that would could generate the flexibility required and maintain control of the import.

The following example (F7) is a simple import of data using PROC ACCESS and the same data as the DDE example. If you try this then make sure you have SAS/ACCESS for PC file formats' licensed and the Excel file is in Excel 97 format!

F7. PROC ACCESS Example

This code

```
%let name=Phuse;

proc access dbms=xls;
      /*DEFINE ACCESS DESCRIPTOR*/
      create work.&name..access;
      path = "c:\PhuseTest1.xls";
      getnames= Y;
      worksheet = "Sheet1";
      range= "A1..C4";
      scantype = Y ;
      mixed = Y;
      type Phuse= C Test= N Page= C;

      /*DEFINE VIEW OF DATA*/
      create work.&name..view;
      select ALL;

run;
quit;
```

Produces

	Phuse	Page	test
1	1	3	
2	2	.	A
3	B	.	

The final resolution came using a mixed approach of PROC ACCESS and OLE Automation. OLE Automation would be used to loop through all the master copies of the Excel files and save a temporary copy in the generic file format. In this format PROC ACESS could then import the file into SAS and processing of the data could continue. These temporary copies would be deleted after import using WIN32 API functions so that space on the network servers can be preserved. Using WIN32 API functions (through the SAS MODULEN function) kept the seamless feel to the application as there was no need to execute a SAS 'X' or SYSTASK command which would cause popup of the command prompt.

This mixed approach worked very well because PROC ACCESS could be used with Excel sheets created in any format and still get the seamless feel to the application. Another advantage of using this method was that the complicated OLE Automation code could be stripped to a minimum and the rest of the code could easily be externalised in SAS macros.

CONCLUSION

The use of a prototyping methodology worked well for OPS because a system could be developed for use by project and operational managers quickly and iteratively. It is important to note that even though there are distinct advantages to working in this way, like an improved and shared understanding of the issues involved in development, the rapid appearance of ROI and reduced requirements risks a proper assessment of whether or not this is an appropriate methodology that should be carried out. Not only this, but it also needs to be noted that with rapid iterations of a prototype it can be difficult to make sure that documentation is complete for all the decisions taken at each iteration. Great care should be taken to ensure that documentation is kept up to date, especially in a highly regulated environment.

SAS/AF, while not a dedicated prototyping tool, works well as such because its relatively simple interface allows applications to be created with minimum amounts of code. Code can also be encapsulated within custom made objects and methods which can be stored in code libraries for easy retrieval for subsequent prototyping exercises.

REFERENCES

Pressman, R (2000) Software Engineering - a practitioners approach (5th Ed) McGraw-*Hill*

ACKNOWLEDGMENTS

Thanks to Katherine Hutchinson and Trevor Clulow for reviewing this paper and my wife Amanda for her help in writing this paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Edward Foster
Oxford Pharmaceutical Sciences Limited,
The Stables,
114 Preston Crowmarsh,
Oxford, UK OX10 6SL
Work Phone: 01491 833338
Email: edward.foster@ops-web.com
Web: http://ops-web.com