Adopting New Software Development Techniques to Reduce Obsolescence

C H R Lane, E S Beattie, J S Chita, S P Lincoln
BAE SYSTEMS
Crewe Toll, Ferry Road, Edinburgh EH5 2XS, UK
Tel: +44 131 343 4932 Fax: +44 131 343 4631 E-mail: charlie.lane@baesystems.com

Abstract:

This paper reports on the advanced techniques employed in the specification of software requirements and the subsequent software development for an E-Scan demonstrator Radar Data Processor. This involves the Rapid Object-oriented Process for Embedded Systems (ROPES) [1], UML syntax, object-oriented design, and automatic code generation and test.
The COTS technology reported is in terms of commercially available state of the art method and tool support for the software analysis and design. The resulting software product contains a significant proportion of COTS code resulting from the code-generation. We are also using automation in development of our MMI, a COTS GUI-builder, and COTS hardware and operating system.
In this paper we also report on the object-oriented method, using the ROPES process, together with information about how in practice we are implementing the theory. We present the structure of the software and how it relates to the application under development.

With these techniques there are significant reductions in obsolescence due to:

- customer visibility and understanding of the product under procurement, making clear the advantages and limitations of what will be produced,
- development of a coherent, consistent and maintainable system specification,
- use of use an industry-standard model notation (UML) to capture the analysis and design, enabling portability of the design to other tools and products,
- flexibility in catering for evolving requirements,
- development of testable requirements, enabling original functionality to be re-checked after addition of enhancements,
- techniques for enabling the re-use or replacement of modules with defined interfaces,
- easy and maintainable connections between specification and implementation,
- high initial quality and low rework costs.

This paper will be of benefit to those just embarking on system and software development, or considering updating processes in a legacy project. It is also applicable to those just embarking on choice of tools and methods for initiating programmes as well as for early feasibility studies.

Keywords: System Specification, Requirements Analysis, Real-time, UML, Object-oriented, Analysis, Design, Modelling, Code-generation

1 Introduction

The E-Scan radar project is aimed at producing a flying demonstrator of an electronically-scanned phased-array antenna. It will be fully capable of tracking targets and will provide some advanced features such as adaptive beamforming, but will not include the full range of functionality of a system such as the Captor Radar integrated with the Eurofighter Typhoon weapon system.

The Trials Monitor Computer (TMC) is the main processor in the radar and is responsible for the signal and data processing, as well as controlling the activities of other subsystems such as the antenna and the receiver/exciter. The TMC consists of two main areas:

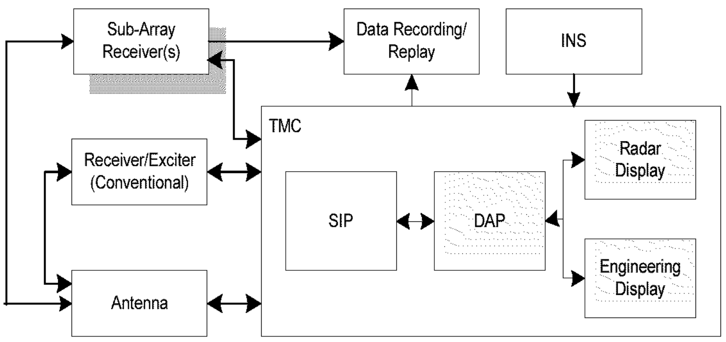


Figure 1

Paper presented at the RTO SCI Symposium on "Strategies to Mitigate Obsolescence in Defense Systems Using Commercial Components", held in Budapest, Hungary, 23-25 October 2000, and published in RTO MP-072.

the Signal Processor (SIP) is largely for handling the flow of digital data from the receiver and processing it continuously to obtain events relating to target detections; the Data Processor (DAP) is event-based, creates tracks of targets from the detections and manages the distribution of RF power radiated, and has an MMI for controlling other functions.

Both SIP and DAP use predominantly COTS hardware, with commercial operating systems and development tools.

This paper relates to the DAP.

2 Technical Details

2.1 State of the Art Software Tools

At the outset, the decision was made to invest in technology to reduce the cost and timescales of software development. This approach is key to making a successful demonstrator in a short period.

The tools have to provide analysis and design support, starting from requirements with a clear path through the design to automatic generation of code from the design (not just code frames). To validate the design, simulation is essential and the testing support must enable verification of the generated system behaviour against that defined in the requirements.

From the handful of tools that met our basic requirements, we chose the I-Logix Rhapsody tool, which provides for full UML analysis and design, code generation and automatic verification against scenarios.

Our core tool set consists of Rhapsody (analysis, design, simulation, verification), DOORS (requirements tracking) and ClearCase (configuration management). Although from different manufacturers, these tools provide useful integration and have been found to work well together. Supporting these are the usual set of C++ compilers, host support (the Wind River RTOS VxWorks) and other productivity enhancements (See Figure 2).

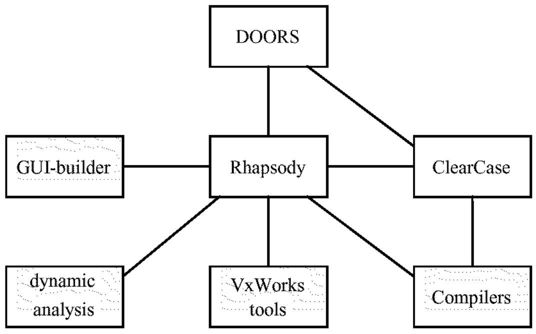


Figure 2

From an obsolescence perspective the capability of the Rhapsody tool to select a target environment is of particular importance. As target platforms become obsolete the tool has a number of platforms that can be selected and the code regenerated for that particular environment. The tool vendor is increasing the number of target platforms supported as market forces dictate.

An early decision to purchase consultancy and training on both tools and methods has proved to be very fruitful and well worth the outlay.

2.2 Using UML

The UML is a notation that has evolved from Software Development. Some of the tools, such as Use Case and Sequence Diagrams are specifically aimed at creating a realistic model of what the customer wants.

Previously, the specification of requirements have been expressed as “Victorian novel” text – often disjointedly spread across a number of documents – combined with a collection of algorithms and little consultation with the software engineers responsible for implementation.

This has often been followed by what is described as the “over the wall” approach where the requirements are passed to the software engineers and the systems engineers move onto something else. Large amounts of software development effort is then spent rewriting the contents of the requirements documents into a Software Requirements Specification (SRS). This process is illustrated in figure 3.

The traditional approach leads to a number of problems:

1. Generation of the requirements is difficult to manage.
2. Traceability to, and Verification of, the requirements is difficult to achieve.
3. Maintenance of the requirements is expensive.
4. Software is difficult to develop.
5. Changes in requirements (which are accepted as inevitable) are difficult to implement.

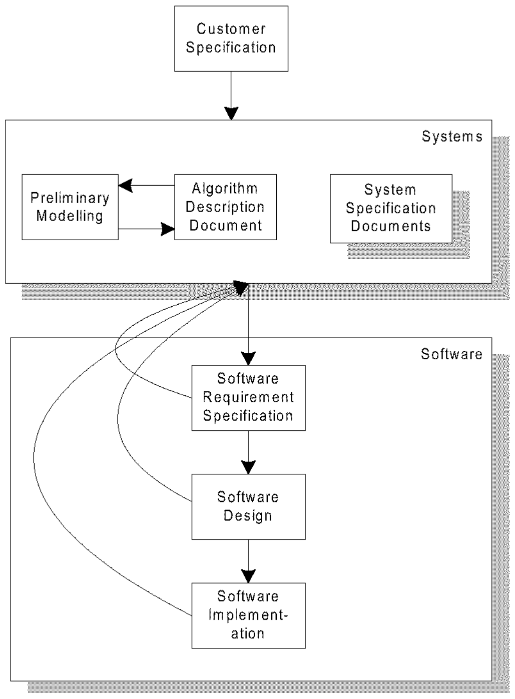


Figure 3

The approach we have adopted for the DAP is to have an integrated systems-software team (see figure 4) and a closely coupled SRS/ACD pair (see figure 5). This approach is detailed below

Requirements Analysis (from the ROPES perspective) is performed using Use Cases, Sequence Diagrams and Statecharts. The Requirements Analysis results in a functional decomposition of the DAP, the details of which are captured in the Software Requirements Specification (SRS). The Use Case descriptions give the functional details of the system in a textual manner, that will be utilised later in identifying objects, with the sequence diagrams defining the Use Case behaviour in a dynamic manner.

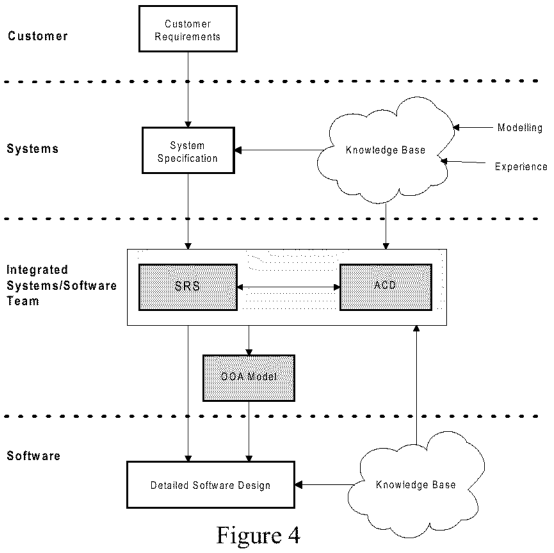


Figure 4

Algorithmic Definition is carried out based on this functional decomposition of the system, which is agreed early in the project lifecycle by the integrated systems-software team. The algorithms are described using Activity Diagrams to show the blocks and flow of algorithmic activity with references to mathematical formulae and textual descriptions as appropriate. This detail is captured in an Algorithm Control Document, which supplements the SRS.

By using the same functional decomposition for both documents, it becomes easier for the software team to understand which algorithms are required to implement a particular area of functionality (i.e. a use case).

The development of the SRS and the ACD are iterative in nature and can allow details of algorithmic implementation to be fleshed out much later in the lifecycle than would normally be the case. One of the benefits to this approach is early introduction of software engineering effort to the process which removes the lengthy delay whilst algorithms are "fully" defined by systems engineers before software development starts.

Links between the SRS and ACD enable the two documents to give a detailed and co-ordinated description of the System. Using hypertext links, an engineer or customer can navigate around the requirements with ease. Both the SRS and ACD are embedded in the DOORS Requirements Traceability tool.

With the creation of a closely coupled SRS/ACD pair a detailed definition of the system exists that can be well understood by those using it. This is the first step to ease of maintenance and the resulting reduction in overhead costs. Generally, maintenance of systems documentation (inevitable in light of changing requirements) is complicated by a poorly defined set of requirements that are scattered across a number of documents that have little or no real relationship. By ensuring that the Specification is easily understandable (using UML) and well laid out (and thus easily navigable) the impact of change can be quickly assessed and is less onerous to implement.

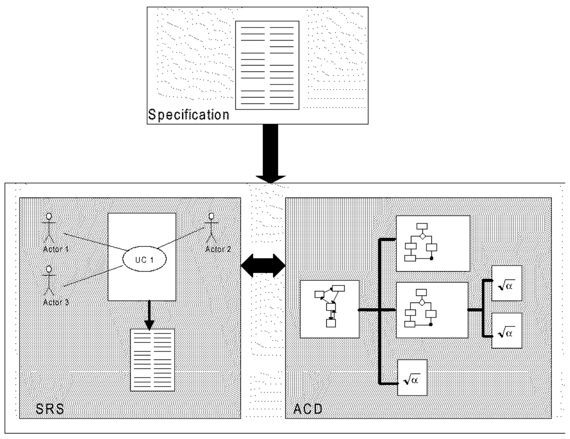


Figure 5

2.3 Systems-Software Integrated Teams

Experience has shown that unless the systems engineer understands the process by which their specification (itself another interpretation of the customer requirements) is implemented the systems-software review process is prone to failure. (Sometimes the systems-software relationship goes the same way.)

With the use of a common language of understanding that is intuitive in its usage these two problems can be alleviated. The fact that Use Case, Sequence Diagrams and Activity Diagrams are simple concepts to understand, powerful in their capability for representing complex requirements and are now widely accepted as a way of describing requirements means that the Software Engineers, who invariably pioneer these new methods, can achieve "buy-in" from the systems engineers.

For this relationship to be successful, it is essential that the systems engineering team are given the appropriate training in the development methodology and sufficient time to review the software work products – particularly the Object Analysis (both structural and behavioural).

On the DAP, systems engineers receive the same training in software methodology and tools as the software engineers. A core team is formed which allows very close inter-working to take place on a level playing field. This enables the software engineers to bring their experience into the development of the system specification whilst allowing the systems engineers a greater understanding of, and input to, the software development process in the subsequent phases of the lifecycle.

In particular, the integrated team work together to create the software object analysis model. This co-operation helps the software team to understand the requirements and means the systems team will understand how the software will implement the requirements. During the creation of this model, the integrated team can ensure, at an early stage, that the software will implement the requirements and algorithms stated in the SRS and ACD.

2.4 Flexibility with Evolving Requirements

As stated, the DAP is being developed using the ROPES process. This is an iterative/incremental means of software development using the Spiral Lifecycle.

Use Case analysis gives a functional decomposition of the system. In our application each Use Case has been identified as an "Iterative Prototype".

These prototypes are taken through the full software lifecycle to produce working software. This gives a great deal of scope for risk reduction in the early stages of a programme by allowing working code to be developed for a target platform. The choice of which prototypes should be developed first is based on risk impact assessment.

The additional benefit is that the prototype is re-useable, in so much as it is a building block to be used in the incremental development of the application.

By careful consideration, based on risk reduction and introduction of required (phased) functionality, the application is developed incrementally by the integration of the prototypes.

By adopting this development process there are two main areas of benefit in respect of flexibility.

The algorithmic development can continue during the software development process for agreed areas of functionality within the system (i.e. Use Cases that may be implemented later in the programme). The Use Cases and Scenarios give the functional structure, or framework, of the system under development at an early stage. This allows the OO development to progress to the detailed design phase before the algorithms must be completed.

In developing functional prototypes and quickly reaching the stage where executable software is running on a target (much earlier than in traditional developments), problems with requirements can be fed-back quickly and avoiding action taken. The prototype may be re-iterated and re-incorporated in the application to include the changed requirement.

2.5 Requirements Testability

Although the combination of Use Cases and Sequence Diagrams gives a powerful means of specifying the requirements there is an additional benefit to the creation of Sequence Diagrams. The use of Sequence Diagrams implicitly forces engineers to address the issue of testing the functionality being defined. This is as applicable for lower level integration test (for sub-system use cases) as it is for high-level system acceptance testing (system use cases).

On the DAP we use the Rhapsody CASE tool to carry out automated Sequence Diagram comparison. That is, the Sequence diagrams specified can be compared with those generated by the actual model created to fulfil the requirements.

2.6 Connecting Requirements through to Design Models

The analysis of requirements is carried out by first of all defining the Use Cases (i.e. the particular areas of functionality) as shown in figure 6. The Use Cases are initially just headlines, but are rapidly filled-out with scenarios: there will typically be a number of Sequence Diagrams for each use case, showing the functionality in specific situations (see figure 7).

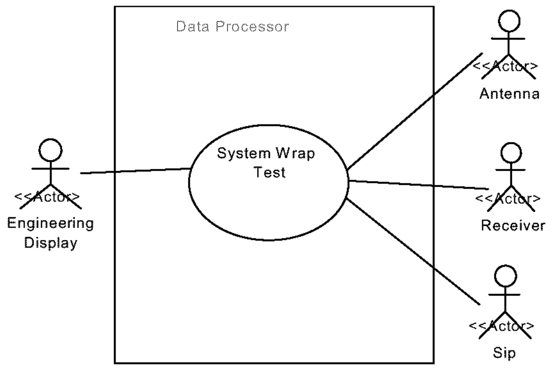


Figure 6-a

System Wrap Test

GOAL/PURPOSE:
Perform the Radar system wraparound communication BIT test.

TRIGGER EVENT: do Radar Wraparound test request from Engineering Display operator

PRECONDITIONS: The CAR radar is in a 'Standby' or Operative State.

POSTCONDITION:
Success End -'Wraparound Test Ok' message is indicated on the Engineering Display.
Failed End - Test failed or timeout. 'Wraparound Test Fail' message and the cause of failure shall be indicated on the Engineering Display.

MAIN SUCCESS SCENARIO
See Message sequence diagram 'MSC_Sys_Wrap'

EXTENSIONS
tbd

EXCEPTIONS
1.No response from any LRIs within TBD milli seconds , Initial policy is to abort operation and report a fault . In the future a recovery sequence (soft reset LRI and retry a max of 2 times) may be implemented.

PERFORMANCE (Quality of service)
Priority: Low. Any radar activity in progress should be allowed to goto completion.
Performance : Test done within 100 millisecond.
Frequency: Periodic - execute during a Resource Frame BIT slot (0.5 Hz)
Episodic - execute Wraparound Bit test on operator request

Figure 6-b

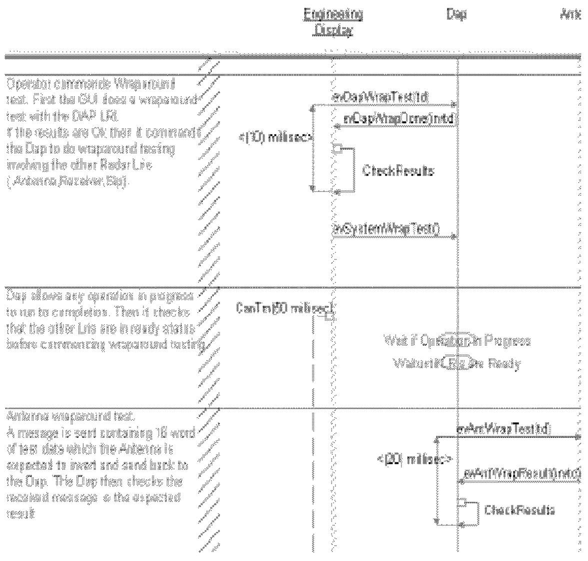

Engineering Display
Dap
Ant
Operator commands Wraparound test. First the GUI does a wraparound test with the DAP LRI. If the results are OK then it commands the Dap to do wraparound testing involving the other Radar LRIs (Antenna,Receiver,Sip).
evDapWrapTest(id)
evDapWrapDone(intd)
<[10] millisec>
CheckResults
evSystemWrapTest()
Dap allows any operation in progress to run to completion. Then it checks that the other LRIs are in ready status before commencing wraparound testing
CanTm[20 millisec]
Wait if Operation in Progress
WaitUntilLRIs are Ready
Antenna wraparound test.
A message is sent containing 16 word of test data which the Antenna is expected to invert and send back to the Dap. The Dap then checks the received message = the expected result
evAntWrapTest(id)
<[20] millisec>
evAntWrapResult(intd)
CheckResults

Figure 7

At this level, the analysis is very much in terms that a customer would understand – we are in the favourable position of being both pseudo-customer (writing requirements on behalf of the actual customer) and contractor (implementing those requirements), so we have been able to make sure that the analysis correctly echoes the requirements.

A key advantage is that, because the Use Cases and Sequence Diagrams are captured in the Rhapsody tool (subsequently used for detailed design) and linked back to source requirements in DOORS, requirements are traceable to the implementation.

2.7 Moving From Functionality to Objects: Domains and Subsystems

The methodology used on this project is Rapid Object-Oriented Process for Embedded Systems (ROPES), which brings together a number of the best practice lines of thought, specialised for real-time applications.

The first stage in this is to define preliminary "subsystems", which are in effect collections of functionality. Each use case is placed into a subsystem – the use cases are decomposed to such a level as to ensure that each use case is in only one subsystem, though the subsystems will often contain more than one use case.

In figure 8, "Radar Control", "Burst Control" and "Tracker" are the subsystems within the "DAP".

The subsystems and their identified artefacts are defined as the "Physical Model". This is captured in Rhapsody by a Physical package (shown in Fig. 11).

The often-difficult borderline between function-based specification and object-based implementation is encountered at this point: the implementation of the use cases is by domain classes instantiated in the subsystems.

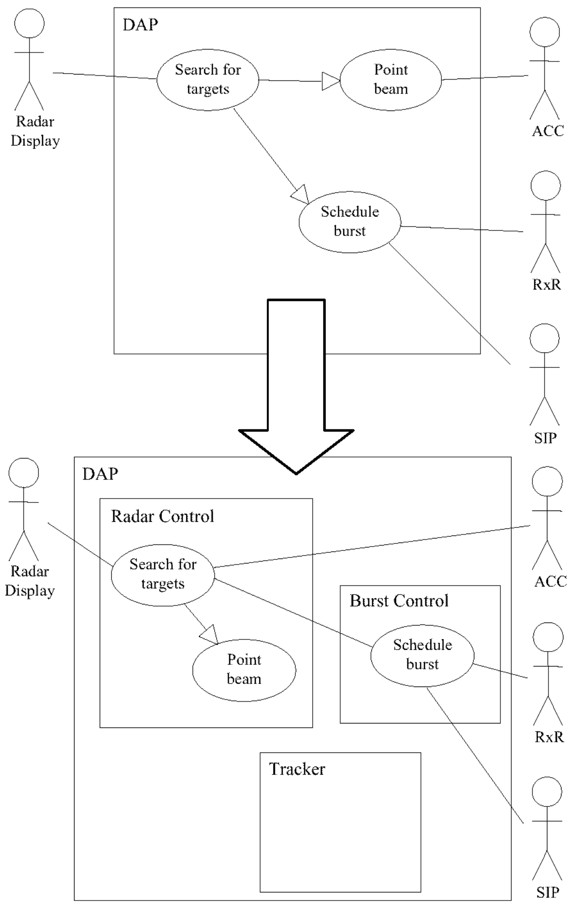


Figure 8

Subject Matter Separation, one of the useful aspects of the Shlaer-Mellor methodology has been imported into ROPES, in the form of domains. Within a domain are collected all the objects that relate to a particular subject matter (e.g. I/O, alarms, tracking). These are an orthogonal set to the subsystems: all objects are in fact defined in domains, but are "used" in the subsystems. The collection of domains identified is referred to as the "Logical Model". This is captured in Rhapsody by a Logical package (shown in Fig. 11).

A domain diagram (figure 9) shows the inter-relationships between the domains:

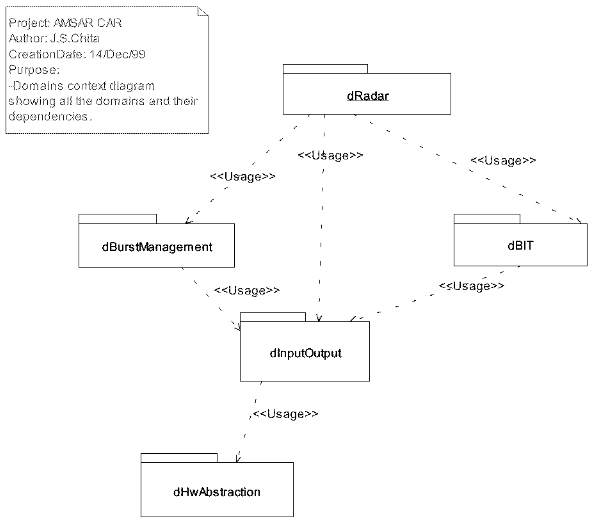


Figure 9

Although it would in principle be possible to follow the Shlaer-Mellor project organisation model and have domain specialists, we have chosen to avoid the potential for boredom in team members by dividing work by subsystem and use-case rather than domains, though we retain an element of domain-ownership to ensure consistency.

2.8 From Analysis to Design

The dilemma encountered with elaborational methods is that one may lose sight of the analysis after adding design information. The high level objects are created only in order that the use cases and their sequence charts may be defined and do not take account of whatever is found necessary for the detailed design.

One solution that has been proposed is to keep two models, the original analysis model and the design model (as elaborated). The difficulty with this is keeping the two models synchronised.

The alternative is to continue with a single model thus reducing analysis/design consistency issues. If one utilises the idea from ROPES that several "views" of the model can exist then one can show purely analysis views from which the design views are subsequently created.

2.9 Implementing Distribution

Shows how distribution is implemented when subsystems are located on separate processors linked via an Ethernet bus. Normally in a single processor system the 2 subsystem communicate with each via 2 associations links using asynchronous events:

1) MessageRouterController->iEngDisplay.
2) EngDisplayController->iDapCommand.

These associations for the distributed processor are then realised using a combination of 2 patterns (figure 10):

1) The Proxy pattern provides location transparency.
2) Forwarder – Receiver implements the interprocessor communication between the 2 subsystems.

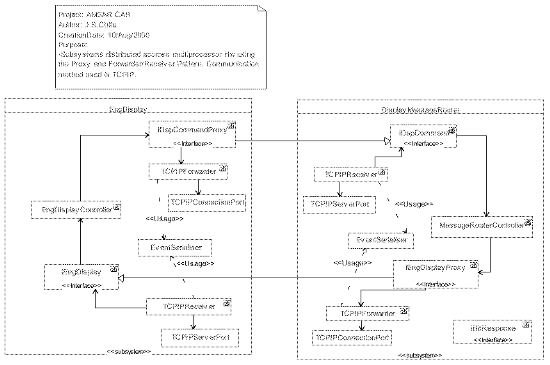


Figure 10

2.10 Units of Re-use

To achieve reduction in obsolescence, we must identify the specific units that are available to be re-used. The aim is to be able either to extract these units to be used in a new system, or to be able to replace units of the current system when changes in functionality are required.

We have identified two main areas of re-use:

a) Subsystem Re-Use, when exactly the same functionality (i.e. the same Use Cases) is required in a new system, or when the complete set of functionality is to be replaced with new functionality in the current system. The subsystem is a convenient unit of re-use as it instantiates all the classes it needs to operate (it can be seen rather as a PCB in hardware terms). When a subsystem is moved to another place, only its external interfaces need to be observed and attached into its new surroundings. This is done in practice by setting up relationships to defined *interface classes* for inputs to the subsystem and initialising the relationships from within the systems for its outputs. Both the identity of the interface class and the initialisation of output relationships are available as public operations on the subsystem.

b) Domain Re-Use, when classes in a domain, originally designed to implement a different set of Use Cases, can be re-used to create a new Subsystem. The domain classes can be seen as a "toolbox" available to implementers of Use Cases, who are encouraged by publication of the domain services to pick classes from there rather than invent new classes. The benefits of the design patterns will automatically be achieved when the classes are used in a new Subsystem to fulfil the Use Cases of that Subsystem. This can be a more difficult level of re-use to achieve, because the implementer of a Use Case may identify slightly different requirements for the classes than those in the "toolbox" – but by careful management maximum use of existing classes, with inheritance to provide for small variations, can be achieved.

Because our development method clearly identifies both subsystems and domains in the artefacts generated, we have a head-start on achieving re-use.

3 Results

We have found the Rhapsody tool and the ROPES method to fit well into our environment. The requirements analysis has provided a sound baseline for the object oriented analysis and design, which is proceeding well. One additional benefit is that we can now provide to our partners in the project not just paper documentation of the design but also animated simulation prototypes.

By using a UML-based method we have also found it easy to bring new recently-graduating members into the team, making use of the training in object-oriented techniques that now commonly forms part of software engineering courses.

We have utilised the concepts of the Physical and Logical models to develop the software application [2]. The Physical model defines the system to be implemented, the logical model captures the domains which themselves contain the essential building blocks, or classes, of the system.

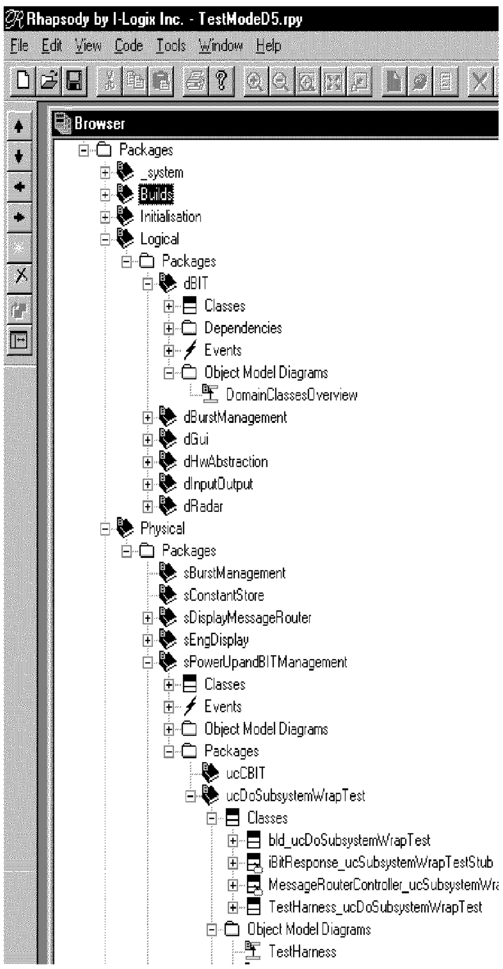
The System package contains the System Actors and the Subsystem architecture identified during Architectural design.

The Build package contains the incremental builds and is effectively the instantiation of the Physical model.

3.1 First Prototype

Our first prototype is based on a wrap-test of the system, which runs a communication check on simulations of the other subsystems. We took this prototype all the way from requirements through use cases to detailed design, implementation and test.

The following diagram shows a "browser" view of the system package structure. The domains have names starting with "d" and are captured in the Logical package, the subsystems "s" and are captured in the Physical package.


Rhapsody by I-Logix Inc. - TestModeD5.rpy
File Edit View Code Tools Window Help
Browser

```
Packages
  _system
  Builds
  Initialisation
  Logical
    Packages
      dBIT
        Classes
        Dependencies
        Events
        Object Model Diagrams
          DomainClassesOverview
      dBurstManagement
      dGui
      dHwAbstraction
      dInputOutput
      dRadar
  Physical
    Packages
      sBurstManagement
      sConstantStore
      sDisplayMessageRouter
      sEngDisplay
      sPowerUpandBITManagement
        Classes
        Events
        Object Model Diagrams
        Packages
          ucCBIT
          ucDoSubsystemWrapTest
            Classes
              bld_ucDoSubsystemWrapTest
              iBitResponse_ucSubsystemWrapTestStub
              MessageRouterController_ucSubsystemWr
              TestHarness_ucDoSubsystemWrapTest
            Object Model Diagrams
              TestHarness
```

Figure 11

3.2 Physical Model: Object Model Diagram

The objects involved in a use case can be shown on an object model diagram for a particular subsystem:

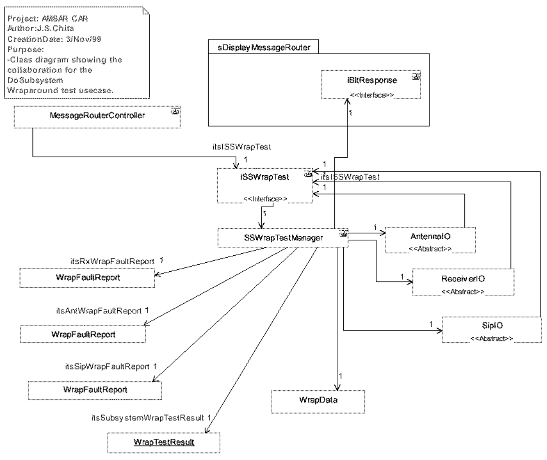


Figure 12

3.3 Logical Model: Active Class

The implementation of the behaviour of the active classes is generally shown in a state chart:

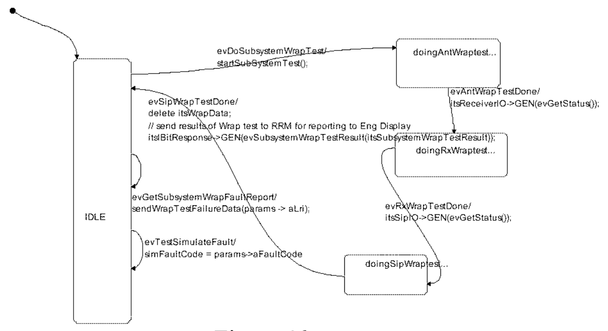

evDoSubsystemWrapTest/
startSubSystemTest();
doingAntWraptest...
evAntWrapTestDone/
itsReceiverIO->GEN(evGetStatus());
evSipWrapTestDone/
delete itsWrapData;
// send results of Wrap test to RRM for reporting to Eng Display
itsIBitResponse->GEN(evSubsystemWrapTestResult(itsSubsystemWrapTestResult));
doingRxWraptest...
evGetSubsystemWrapFaultReport/
sendWrapTestFailureData(params -> aLri);
IDLE
evRxWrapTestDone/
itsSipIO->GEN(evGetStatus());
evTestSimulateFault/
simFaultCode = params->aFaultCode
doingSipWraptest...

Figure 13

3.4 Build Model: Usecase Instantiation

Shows the classes used to build up the use case.

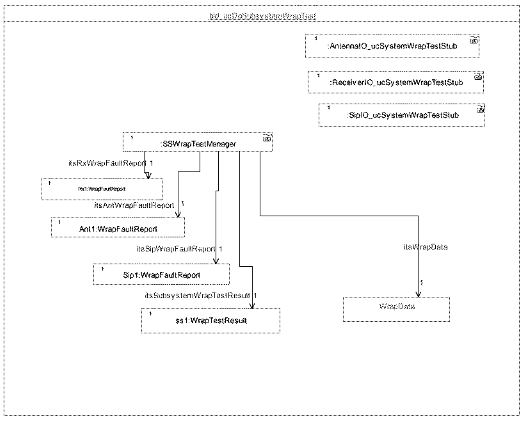


Figure 14

3.5 Build Model: Subsystem Instantiation

Shows the classes used to build up the sub-system.

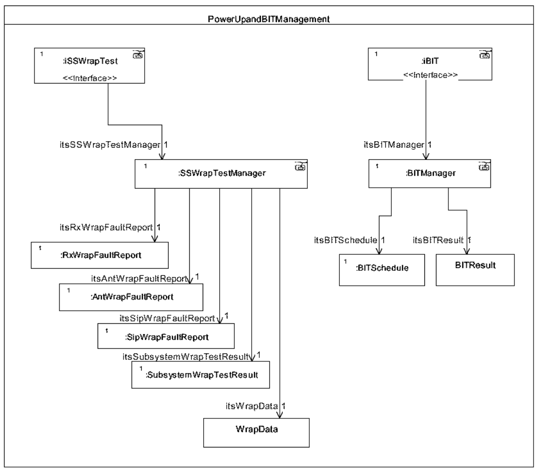


Figure 15

3.6 Build Model: System Instantiation

Shows the classes used to build up the DAP system for standalone testing on the PC development host.

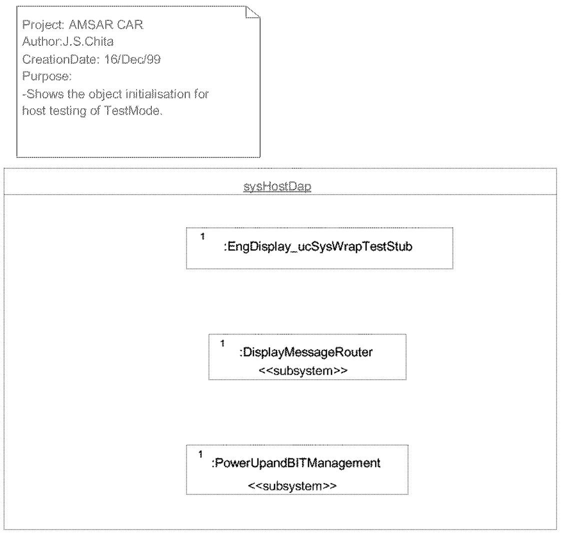

Project: AMSAR CAR
Author:J.S.Chita
CreationDate: 16/Dec/99
Purpose:
-Shows the object initialisation for host testing of TestMode.

Figure 16

The wrap-test has proved successful not only as confirmation of the tool and method choice, but also as the first real prototype that implements part of the functionality of the full system.

4 Conclusions

Software represents a large and increasing proportion of the costs of current systems. Current high software costs for almost every project indicate that this is an area where reduction of obsolescence and increase of re-use must be introduced if costs for future systems are to remain within limited budgets.

While the object-oriented approach provides a basic framework for encapsulating functionality to provide a theoretical possibility for re-use, it does not of itself provide the key advantage. Simple insertion of object-oriented analysis and design into a company's processes does not provide all the advantages that could be obtained, some companies finding little benefit.

To take full benefit from object-oriented techniques, a coherent method of capturing the requirements in a customer-visible way, analysing those requirements to provide the basis for the design, managing the key step from functionality to objects and building up functionality with prototypes must be used.

We are convinced that our use of the ROPES method with advanced tool support will enable us to build up software matching the requirements, to maintain that software as requirements evolve, and to re-use significant parts of the software in new systems having requirements in common.

5 References:

[1] Douglass, Bruce Powel, Doing Hard Time : Developing Real-time Systems with UML, Objects, Frameworks and Patterns, 1999, Addison-Wesley, ISBN 0-201-49837-5.

[2] Douglass, Bruce Powel, Effective Use Cases for Real-Time Design, Version 1.3.1

6 List of Acronyms

ACD	Algorithm Control Document
COTS	Commercial Off The Shelf
DOORS	A requirements traceability tool
GUI	Graphical User Interface
MMI	Man Machine Interface
ROPES	Rapid Object-oriented Process for Embedded Systems
SRS	Software Requirements Specification
TMC	Trials Monitor Computer
UML	Unified Modelling Language