REUSE IN THE TELECOMMUNICATION DOMAIN USING OBJECT ORIENTED TECHNOLOGY AND ADA

Anders Sixtensson, Wenchuan Ye
Department of Communication Systems
Lund Institute of Technology
Box 118 S-221 00 Sweden
Email: anderss@tts.lth.se

May 4, 1990

Abstract

An overall object oriented method supporting all steps from a given requirement specification to an implementation in Ada is presented. The method is formulated from the experiences when extending POTS (Plain Ordinary Telephony Service) with a number of new service features on a prototyping system. Reuse is achieved on different levels. System Interactive Diagrams (SID) and Object Interactive Diagrams (OID) are used to transform the initial functional requirement into an object oriented model. An extended version of SDL called OSDL (Object oriented SDL) and Ada are chosen as specification, design and implementation tools. Ada multi sublibraries are studied for reuse.

The paper is produced at the Department of Communication Systems at Lund Institute of Technology in the research project ARISE (A Reuse Infrastructure for Software Engineering) which is part of the RACE (research in Advanced Communication Technologies in Europe) programme.

Keywords: software reuse, object orientation, Ada, telecommunication.

1 INTRODUCTION

Software reuse is widely believed to be a major key to improve the productivity and quality of software development. By using existing software components that already have been tested and proven effective, the cost and the time for developing a new product are reduced. The software quality is improved.

Object oriented Software Engineering (OOSE) is the most promising technique known today for attaining the goals of extensibility and reuse [1]. Object oriented techniques have their main power in their ability to model a specific domain, i.e. changes or modifications can be described fairly straightforward. Specifically Object Oriented Domain Analysis (OODA), has been found vital for the success of reuse. Software components that result from domain analysis are better suited for reuse because they capture the essential functionality required in that domain.

The intention of the research work has been to analyse some of the technical aspects of reuse within the telecommunication domain such as:

- How to achieve reuse in the early phases of the development life cycle.
- The use of OOSE for large and critical applications such as telecommunication system.
- How to make an consistent transformation from the initial functional requirements to an object oriented model of the problem space.

- Make an efficient use of Ada.

An overall object oriented method has evolved from the research work on the prototyping system [2] and has been used repeatedly when extending the POTS software. A pragmatic approach has been taken; obtaining knowledge and experience by analysing and reimplementing existing systems. The basic concept of the method is similar to Objectory [3] but an emphasis is made on reuse in telecommunication applications using OSDL [4] and Ada as implementation languages.

2 THE PROTOTYPING SYSTEM

The research work was based on a prototyping system. The prototyping system is a Line Interface Module (LIM) from the MD 110 PABX system by Ericsson and a SUN 3/50 workstation. LIM is a micro processor controlled unit and can connect a number of subscriber extension lines, trunk lines and operator lines. It can work as an autonomous exchange as well as part of a system comprising several LIM's connected via a switch.

The call handling software, implemented and executed on the SUN 3/50, replaces the basic switching and control functions normally made by the LIM. This is obtained by making the LIM transparent. The signals from the telephone units in the LIM are sent directly to the LIM input/output unit, which in turn sends the signals to the SUN via serial ports, see fig 1. The system makes it possible to experiment with software architectures in a very flexible manner and to examine the trade offs between different methods to produce telecommunication software. Though the research work relied on the prototyping system, the method outlined in the next section, is of a general nature.

3 OVERALL OBJECT ORIENTED METHOD

To achieve reuse in an efficient and powerful way requires a consistent method during the life time of the system. Consistency helps to reduce complexity, and increases reliability. The method described in this section has evolved from the research work on the prototyping system. For frequently modified systems, software development can be regarded as modifying from one version of a system to another. A special case is the creation of a system i.e. starting from nothing. The general phases for extending a system with a new service feature may be outlined as follows:

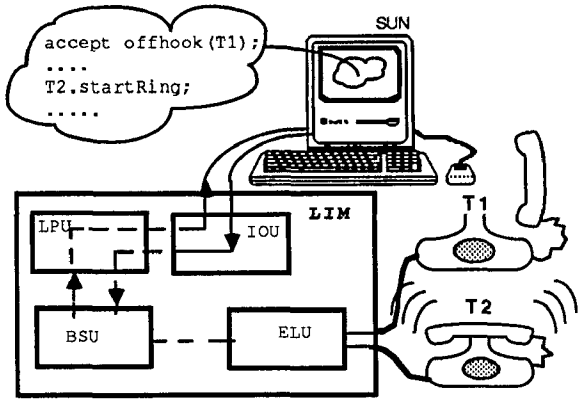

Figure 1: *The prototyping system*

1. Requirement Specification
2. Requirement Analysis
3. Domain Analysis
4. System Design
5. Detailed Design
6. Implementation

It is of course allowed and necessary to backtrack and iterate between the phases until a satisfied solution is reached. A traditional bottom up test activity is performed, i.e. unit test on code level, module test on design level, and so on, all the way up to system integration, acceptance test and validation. This paper will however not emphasize on testing methodology.

Requirement Specification

A description, often textual, is made to define what is required of the system. These initial requirements are often functional, usually appearing as certain inputs to the system and certain outputs from the system. For example, the required behaviour for ordering a call transfer could briefly be

described as follows: Call transfer is possible to order by dialing the sequence *3*(transfer telephone number)# on the telephone provided a dial tone is given. The dial tone should go off as soon as the first digit is dialed. A completed order is acknowledged by a tone, a dial tone if the order has been accepted otherwise an error tone.

Requirement Analysis

This phase will end up in System Interaction Diagrams (SID) made by analysts. A SID describes how a user and the system interacts by means of signals or commands to fulfill the requirements for normal use of the system. Exceptions are dealt with later. The SID for ordering a call transfer is shown in fig 2. The two vertical lines represent the user and the system respectively, while the directed lines between the user and the system denote the signal flow between the user and the system. The vertical arrow line and numbers denote the sequence of events. From the user's point of view, fig 2 means that when making off hook, he should hear a dial tone. Then he can dial a digit and as soon as he dials a digit, the dial tone should go off. From the system point of view, the SID means that on receiving off hook, system should give a dial tone and as soon as a digit is received, system should cut off the dial tone.

The simple diagram effectively encapsulates the user and the system interaction. It is very important that the people responsible for the requirement specification and the analysts agree on this translation. By always comparing the SID to already existing SID's, some identical parts of user interactions with the system can be identified and marked as reusable. SID's are also very suitable for making test specifications on system level.

Domain Analysis

Domain analysis is the key for OOD as well as for reuse. It is during this phase, that the objects and their operations are found. In order to reflect the problem space and to capture essential concepts, domain knowledge is necessary.

There are three steps in domain analysis. The first step is to make a conceptual model and analyse it together with the SID's to recognise objects and

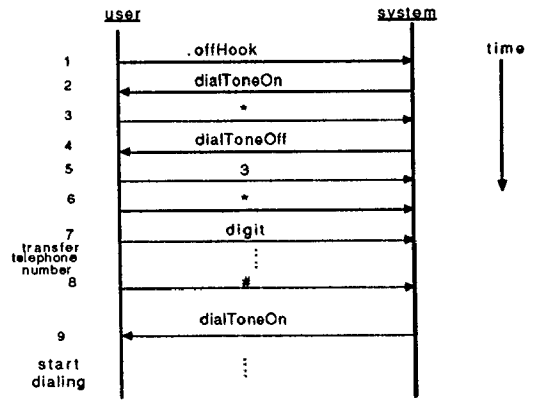

Figure 2: *Example of a SID*

their operations. Exceptions are also considered in this step but are explicitly dealt with in the detailed design. The second step is to look in the library to see whether the objects found during step one already exist, and can be reused, perhaps with some minor modifications. If the objects don't exist, then a decision is made whether the new objects should be put in the library to be classified as components. Finally an Object Interaction Diagram (OID) is made to capture the dynamic behaviour of the system; the sequence of communication is established. The operations of the objects appear as signals in the OID. see fig 3.

In fig 3, the vertical lines denote objects decomposed from the system, such as *telephoneA, toneDevice, switch, call* etc. The directed lines among objects represent signal passing between objects. The vertical numbers indicate the sequence of events. An OID is in fact an expanded SID. If a part of a SID is reused, then the corresponding OID to that part should also be reused.

Another approach of OODA is to let the library manager give a first proposal of a proper set of components before making the conceptual model. However, this approach requires very intelligent library manager tools which are not available for the moment.

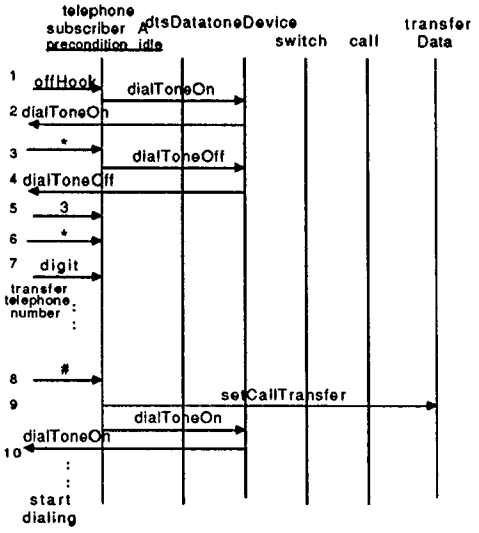

Figure 3: *Example of an OID*

System Design

In this phase the objects are classified into servers and nonservers (actors and agents) [5] using the OID's. For complex systems, it is necessary to combine a set of related objects together to form subsystems. Now subsystems and objects should be mapped into OSDL skeleton graphs. Subsystems are translated into blocks and the nonserver objects are translated into processes or procedures in OSDL. The server objects only contain one state and therefore it's no help to describe them in OSDL. Server objects may be well mapped directly to Ada package specifications during the next phase, detailed design.

Detailed Design

So far the normal use cases of the requirement specification have been covered, exceptions and error conditions are dealt with during this phase. The detailed OSDL process diagrams will now be completed and each OSDL block, process, procedure is mapped to Ada package specifications. Exceptions and server objects which are not described in OSDL, are added. This step ends with a set of completed Ada package specifications.

Implementation

In this step, each object, i.e each Ada package body, is implemented. Both bottom up and top down developments are used. OSDL process graphs are strong tools for dealing with Ada tasks that are bidirectional, which is often the case for telecommunication systems. Candidates for reusable components are implemented as compilation units in Ada. For those reusable components, that are bidirectional, recursive export mechanism has to be used. Supported by APSE, all the compilation units separately compiled are put into the Ada programme library for reuse. The TeleGen2 environment [7] is used to deal with version control, configuration management, team works, reuse of subsystems etc. The inheritance on the subsystem level is achieved with the help of multi sublibraries. Further discussions on multi sublibraries will be covered in a following section.

4 OOSE

The benefits of object oriented technology are enhanced if it is addressed throughout the software engineering process as indicated in the term OOSE. However, effective reuse requires an emphasis on the early phases of OOSE, i.e. object oriented domain analysis (OODA) and object oriented design (OOD).

OODA

How to model a specific domain and to identify the operations, objects and structures which commonly occur within the domain is a process called OODA. OODA should be used a number of times during the life time of a system. An initial OODA is an activity occurring prior to an existing and implemented version of a system inside the domain. The major reason for this is that we want to have a stable domain model not influenced by a specific version of the system. A stable domain model is a prerequisite to achieve reuse and robustness with respect to future demands of increased functionality. Within the domain of telecommunication for example, typical objects are subscribers, telephones, lines, tonedevices etc. and operations are connect two telephones, generate a busy tone, send digits etc.

The result from this initial OODA will serve as a framework for developing other versions of the system inside the domain. OODA will also be used when analysing requirements for a specific version to be developed. Specific objects and operations are introduced or expanded from the framework to meet the user requirements for that specific version. For example, in the initial OODA, we have introduced an object called *terminal* which is an abstraction of different subscriber equipments such as telephone, telex, telefax etc. When doing OODA for a specific version, a specific object *telephone* is described in the model by inheriting the common properties from the object *terminal* specialized with specific characteristics.

After several versions have been developed inside the domain, OODA should be applied again and more general objects could be found. On the other hand, this pragmatic approach would also expand the domain knowledge which is useful for the development of new systems inside the domain.

Modelling of system evolution

OODA and OOD are claimed to reflect the problem space, i.e. changes and modifications in the real world should have a natural mapping using an object oriented model. The research work has also approved of this statement. Here are some examples.

- When the system was extended with the service feature abbreviated dialing (the ability to define a shorter dialing sequence for a often used telephone number) the system needed to store and read a set of abbreviated numbers associated with every subscriber. This was modelled by inheriting all subscriber data and specializing by adding a data field and the operations associated with the added data.

- As mentioned earlier, OODA facilitates the introduction and description of different subscriber terminals like telephone, telefax etc. which all have some common properties and some special ones.

- Concurrency modelling is implicit for object oriented systems. Each object is responsible for its own protection in a concurrent environment.

- A future expansion of the system could possibly lead to a change in the internal representation of subscriber data. The abstraction concept makes the users of this subscriber data unaware of the internal change as long as the defined interface to the object remains. Furthermore, the encapsulation of data and its operations facilitate the updating of the internal implementation of the operations working on the data. Without encapsulation, a search through the whole system is necessary to find all (perhaps) functions that operates on the actual data.

OID - from functional requirements to an object oriented model

The initial requirements on telecommunication systems specified by a customer are often functional and this leads to a nontrivial transformation into an object oriented modelling of the problem space. There is no one to one correspondence between requirements and objects. One functional requirement may for example be carried out by a number of objects, and on the other hand, one single object may satisfy a number of functional requirements.

Our solution is to let the OID's in our method be the connection between the functional view and the object oriented view. An OID contains both the proper set of objects found during the analysis phase *and* how these objects interact with each other to fulfill the corresponding SID. This well defined transformation process leads to a high consistency of the system.

5 REUSE

Reuse on different levels

Software reuse is not limited to source or object code. Reuse may occur in varies phases of the life cycle. The initial domain analysis will facilitate the identification of software components in the early phases of the software life cycle. By having a well defined transformation between the different phases of the method, the reuse effort on higher levels is automatically transferred to the lower level of the system description and documentation.

For example, by always comparing a newly developed SID to already existing SID's, some identical parts of user interactions with the system can be identified. In this way, reuse on requirement analysis level is achieved. And reuse on this level will certainly influence the other phases down to implementation level. It's obvious that the earlier a reusable component can be identified, the more we save in terms of time and cost.

Reuse and Extendibility

The classification of objects into servers, and nonservers (agents and actors) [5] is significant since the characteristics are very different.

Server objects don't pass any messages to other objects, they just wait for receiving messages. This property makes the server objects very similar to abstract data types and are therefore very useful for building and representing data and structures in a bottom up fashion. A hierarchy of server objects from application independent at the bottom to more application dependent higher up is defined during the System Design phase. This hierarchy is easy to represent in Ada using the `with` construct. Extensions and modifications to server objects is done very easily, we just have to add the new operations required for a new service feature.

The nonserver objects are used to fulfill the required behaviour of the system, i.e. the desired sequence of signals. Usually the agent or actor objects need some modifications before they can be reused when extending the system. This is referred to as extendibility. Well defined interfaces for these objects are needed to achieve the extendibility. The general rule is to keep the objects both cohesive and loosely coupled, so that a change only affect one or few objects rather than trigger off a chain reaction of changes over the whole system.

Reuse beyond inheritance

Within the object orientation community, people seem to focus on the inheritance mechanism as a means for reuse too narrowly to notice any other means. Although reuse via inheritance is not to be dismissed, there are more powerful reuse mechanisms.

A special case of reuse is to have the system accept new objects that were not defined when the system was first created, rather than to modify an old object or to make a specialization of the old object. For example, the service feature call hold (make a temporary connection during an already established call) was implemented by introducing a new object *callHold* to handle the required behaviour. Object *callHold* is responsible for saving the old connection and to restore it after the temporary connection is terminated.

Another powerful reuse mechanism is based on interchangeable parts. These interchangeable parts are objects when using OOSE. By carefully examining the functionality of each part, and by having well defined interfaces among the parts, the reuse of each part is greatly enhanced. For example, POTS and all the extension software systems use object *telephone*. There are many versions of source code describing object *telephone*. They are all interchangeable. Supported by the APSE multi sublibrary management, the extensions to the system with new service features were done easily by replacing the object *telephone*. By maintaining a high degree of consistency in both the interfaces and functionality of interchangeable parts, a set of useful families of components can be constructed. Furthermore, system can be designed to readily accommodate different family members.

6 ADA

Ada and OOD

It is known that Ada is not a truly object oriented programming language. Then why Ada? Firstly, although the lack of inheritance, Ada contains many unique features, namely concurrency, exception handling, ability to address hardware, high degree of portability etc., which are essential for real time systems such as telecommunication systems. It is fairly easy to implement concurrent objects using Ada's package and task facilities.

Secondly, object orientation really applies to more than just programming. In Ada, object oriented concepts are actually applied to design. Effective use of Ada demands that it is applied in the context of modern software engineering, namely information hiding, data abstraction, encapsula-

tion above the subprogram level and concurrency. OOD is an approach that exploits Ada's facilities. In spite of the deficiencies for object oriented programming, Ada still provides a useful vehicle for applying object oriented concepts throughout the software development life cycle.

Ada and OSDL

One of the things that must be considered in software engineering is to chose proper languages. OSDL and Ada are two languages that used together cover a large part of the development phases in a traditional life cycle model, from specification to design and implementation. OSDL is an object oriented extension of SDL [6] containing concepts for how to describe a large and complex real time system at different abstraction levels. The graphical syntax is very useful for capturing the complexity and functionality of systems. OSDL has concepts like inheritance, remote procedure calls and virtual procedures which together facilitate the description of extensions and modifications of a system.

The translation from OSDL to Ada is straightforward using predefined translation rules. Roughly speaking, system and blocks in OSDL are mapped to Ada packages and OSDL processes to tasks. Once the OSDL diagrams was made, the Ada code was generated manually very easily and quickly.

Ada Multi sublibraries management

The Telegen2 [7] environment provides a powerful multi sublibrary mechanism which can be described as follows. An Ada library consists of a number of sublibraries. A library list specifies the name of the involved sublibraries. In the library list, the order of the sublibraries is significant because the compilation results are usually written into the first sublibrary called the working sublibrary. The other sublibraries are read only. When compiling or linking, the library starts with searching the working sublibrary and continues down through each successive sublibrary until either the unit is found or the list of sublibraries are exhausted. Multi sublibraries provides a powerful mechanism to generate a new software version based on old systems for software reuse purposes.

For example, the new service feature call hold was handled in this way. The sublibrary `pots.sub`, which handles POTS, remained without any changes when the new sublibrary `callHold.sub` was introduced as the working sublibrary. All the objects added or modified while implementing call hold were compiled into `callHold.sub`. The other objects which were not updated, i.e. not recompiled, were inherited from `pots.sub` during linking. After linking, a new POTS version extended with the service feature call hold was executable. If `pots.sub` was put back as the working sublibrary, a new linking would get the original POTS executable again. In the same way, a set of new service features were introduced and different sublibraries were built. By different combinations of the sublibraries, a number of versions of POTS software were acquired.

The hierarchies of server objects which are strongly recommended for reuse are also dealt with easily by the multi sublibrary mechanism. For example, objects *callHold* and *callBack* both use lists structure, the library files corresponding *callHold* and *callBack* look like:

```
library file for callHold:
   Name:  callHold.sub
   -- working sublibrary implementing
   -- callHold
   Name:  pots.sub
   -- sublibrary for implementing POTS
   Name:  utility.sub
   -- defined in [Booch87]
   Name:  basicComponent.sub
   -- defined in [Booch87]

library file for callBack:
   Name:  callBack.sub
   -- working sublibrary implementing
   -- callBack
   Name:  pots.sub
   -- sublibrary for implementing POTS
   Name:  utility.sub
   -- defined in [Booch87]
   Name:  basicComponent.sub
   -- defined in [Booch87]
```

To work in teams is necessary for large system and is much easier to organize with the help of multi sublibraries. For example, if one team is working with the object *telephone* and the other team is

in charge for the object *call*, the library file could look like:

```
Name:  callHold.sub
......
Name:  \team2\call.sub
-- to handle object call by team2
Name:  \team1\telephone.sub
-- handle object telephone by team1
Name:  utility.sub
Name:  basicComponent.sub
```

7 DOCUMENTATION

It should be noticed that the documentation in all phases must be well organised and kept consistent. From extendibility point of view, the ability to trace a requirement in the documentation of all phases is very important. No changes are allowed in a lower level without updating the documents on higher level and vice versa.

All the documents should be managed by a powerful database which can handle text files, figures as well as code libraries. Moreover, the reuse relations must also be documented, if we for example can identity common parts in different SID's, this must be reflected in the documentation.

Before delivery of an ordered system an acceptance test activity is performed together with the customer. The customer will accept or reject the system based on how closely it matches the original requirements (often functional). Now we have use of the SID's developed during the requirement analysis phase. The SID's will in fact be used as test specifications. Since the SID's have been used as a base for the system development work, the traceability of the system would be increased. The software thus produced is easily modified and extended.

The documentation should in fact be viewed in an object oriented manner, i.e. the initial requirements are inherited to documents on lower level which will add more and more detailed description of the documentation until the source code level is reached. If the requirements are inherited through all phases then we have a guarantee for traceability and quality. Unfortunately, this intelligent handling of documentation require tool support and to our knowledge there is no such tool available yet.

However, an prototype for an intelligent tool to handle all sorts of documentation and the relations between the documents is the goal for the next phase of the project. Such a tool is now in an early stage of development using ARCS [8] which is a powerful environment for the Telegen2 compiler. ARCS is originally made to support and maintain the relations between different Ada compilation units but ARCS also has the capability of organizing non-Ada document relations which are the ones we try to exploit. All experiences gained at the project have been very valuable when the tool has been discussed.

8 CONCLUSIONS

The research work has, as mentioned before, taken an pragmatic approach. The project started with a basic system (POTS) without any service features added. New service features, such as abbreviated dialing, call transfer, call hold, call back etc. were added, one by one to the system. After every new version of the system, we always tried to identify the bottlenecks when extending the system. Proposal to solutions was formulated to be verified during the next evolution of the system. In total, six version of the system were built and by always have a specific set of problems to solve for every version, great knowledge and experience were gained during the project. A number of problems and our proposal to solutions have been identified. And lessons on reuse, OOSE and Ada have been learned. A number of conclusions may be drawn from the experience.

- Reuse may occur in all phases of the software life cycle. The early recognition of reuse is of great importance.
- OOSE is the way toward reuse. Reuse is not limited to inheritance. Interchangeable parts are powerful means of reuse.
- How to define objects is the main problem in OOSE. OID provides a good translation from functional requirements to an object oriented model.
- Although lack of inheritance, Ada still provides a powerful vehicle for applying OOD.

- Ada and OSDL used together cover a large part of the software life cycle in the telecommunication domain.
- The multi-library manager supplied by APSE provides powerful tools for code level reuse, version control etc. But for a large library of reusable components including specification, documentation etc, a general database is necessary.

Identifying and collecting reusable components is only the first step in a discipline of reuse. A lot of work, such as formal specification and retrieval of components etc, has to be done to put the reuse into practical use in industry.

9 ACKNOWLEDGEMENT

The authors would like to thank Lars Reneby for valuable comments and for proofreading the manuscript.

References

[1] Bertrand Meyer *Reusability. The case for Object Oriented Design* IEEE software, March 1987.

[2] C. Yeh, L. Reneby, B Lennselius, A. Sixtensson *An educational development system employing SDL design and automatic code generation.* Presented at SDL Forum, Lissabon 1989.

[3] I. Jakobsson *Object Oriented Development in an industrial Environment*, Proceedings of OOPSLA'87, 1987.

[4] D. Belsnes, H. Dahle, B. Pedersen *Rationale and Tutorial on OSDL.* Mjolner Report Series.

[5] Grady Booch *Software Components with Ada.* The Benjamin Cummings Publishing Company 1987.

[6] CCITT recommendation, Z.100.

[7] *TeleGen2 SUN/Ada version 1.3a User guide*, TeleSoft

[8] *ARCS User guide*, TeleSoft