Structuring Interfaces

Alexander Ran and Jianli Xu
alexander.ran | jianli.xu@research.nokia.com
Nokia Research Center, Software Technology Laboratory
P.O.Box 45, 00211 Helsinki, Finland

Abstract

Software components are often described by interfaces organised as sets of externally visible elements, such as signatures of callable functions and procedures, or types of messages and signals accepted by the component. While it is a common practice to construct larger components by composing smaller ones and abstracting their function, interfaces of composite, higher-level components are still described as sets of function signatures or message types. This situation indicates a granularity mismatch between the level of abstraction of components and their interfaces. We believe that just as components are structured as compositions of lower level components, interfaces must be structured as compositions of lower level interfaces. In this paper we present an approach to structuring and describing interfaces of large software components. We model interfaces as collections of interface objects that have state and may exhibit non-trivial behaviour. We also compose basic interface elements such as functions signatures and event types into composite elements that we call interactions. Interactions correspond to meaningful, from the component user point of view, services provided by interface objects.

1. Introduction

Software components are often described by interfaces organised as sets of externally visible elements, such as signatures of callable functions and procedures, or types of messages and signals accepted by the component. While it is a common practice to construct larger components by composing smaller ones and abstracting their function, interface elements are usually not structured or abstracted but are just propagated as such through component aggregation hierarchies. Interfaces of composite, higher-level components are still described as sets of function signatures or message types. Thus higher level components of even a small system may include several hundreds of interface elements. Situation may get significantly worse with large software systems. For example in a telephone switch hierarchical component groupings may span four levels. Components on the lowest level of the hierarchy typically include dozen elements in their interfaces. Since only components are grouped and abstracted, interfaces between composite components still contain lists of messages and events. You can estimate that if only five to ten components are composed on each level, on the higher levels of this composition hierarchy components may have thousands of elements in their "interface". This situation indicates a granularity mismatch between the level of abstraction of components and their interfaces. We believe that just as components are structured as compositions of lower level components, interfaces must be structured as compositions of lower level interfaces.

Most interface description languages view interface elements as stateless entities with no behaviour. These languages can describe only static, structural properties of interface elements, and treat interface elements as unrelated, independent entities. We model interfaces as collections of interface objects that have state and may exhibit non-trivial behaviour. Thus our interface structuring approach also allows to describe dynamic properties of interfaces, valid interaction patterns, as well as context information relevant for understanding and controlling the interaction. We also compose basic interface elements such as functions signatures and event types into composite elements that we call interactions. Interactions correspond to meaningful, from the component user point of view, services provided by interface objects.

Our main emphasis was on effective techniques for structuring interfaces similar to the techniques used for software components. Since our experience is based on structuring interfaces of existing software the steps documented in this paper have a bias towards re-engineering. However we believe that the same principles are valid in forward engineering and lead to well-structured interface designs.

We felt it is important to demonstrate our approach with a real rather than a synthetic example. We used the Windows Telephony Application Programming Interface (TAPI) [1]. It is complex enough, but not excessively large for a paper. However since our approach targets significantly larger systems than TAPI, to demonstrate certain ideas we had to stretch a bit both the example and the approach.

The paper has two main parts. The first part describes main concepts and ideas we use to structure interfaces of large software components. The second part describes briefly the case study and includes some useful hints on how to apply the ideas.

2. A model for structuring interfaces

Our model for structuring interfaces is based on threee main concepts : domains, interface objects, and interactions.

2.1 Domains

Large software components often have interfaces that correspond to different aspects of system design. For example a component may provide interfaces for configuration, management, operation, co-ordination, consistency maintenance, persistency, reliability, monitoring etc. One may say that a component provides different categories of services to its clients. These different service categories are used in different domains of interaction.

Effective approaches to software design are based on separation of concerns. Interactions of a component in different domains should be and is often designed separately. However this separation is often lost at the later stages of design and is rarely used for structuring interfaces. When software designers use a component, they usually need to consider at the same time only one category of services provided by the component. Filtering important information from unstructured interface descriptions is often hard. Therefore we partition component interface

SIGSOFT 96 Workshop, San Francisco CA USA
© 1996 ACM 0-89791-867-3/96/10 ..$3.50

descriptions by domains of interaction in which the component may participate.

Partitioning interface descriptions by domain hides possible interactions between domains from the interfaces. One should analyse these "hidden" interactions carefully in order to ensure that interface descriptions provide sufficient information for safe use of the component. Interaction between domains may indicate that either separation is inappropriate or component implementation introduced unnecessary cross-domain coupling.

Partitioning interfaces by interaction domain or service category has several benefits:

Separation of concerns. Organising interfaces by domain effectively filters information required by a software designer when using a component. It is also very effective for the design of interfaces provided by components.

Reuse. A system may include a number of components that provide a particular category of services. For example check-pointing may be required for co-ordinated roll-back, hot restart on failure, etc. It must be supported by all components whose state must satisfy some consistency constraints with other components. Separating check-pointing interface from other activities performed by a component makes standardisation of this interface across different components easier. For example design of check-pointing interface may be reused; a component that requires check-point interface for co-ordinating roll-back would work with all components that support standard check-pointing and roll-back interfaces.

Controlled propagation of change. The rate of change in different domains is different. While user interface tends to change continuously, component interfaces to support reliability may be rather stable. We can bound changes within the component by partitioning its interfaces by domain.

2.2 Interface Objects

The advantages of separating interface from implementation are well understood. Implementation of a component can be changed independently of other components as long as it provides the same interface. Multiple implementations of an interface may exist and coexist to provide variability required in the system. However the common interpretation of component's interface as a set of names, understood by the component and visible to other components, has some weaknesses. Are there any relations between the interface elements of a component? When several components are combined to form a composite software component, what is the relation between interfaces of the composite and the components? What does it mean for two implementations to support the same interface? It is obvious that being able to respond to the same set of names is not sufficient. We know there are groups of related operations provided by components, relation of order between different operations, state-dependent availability of operations, etc. Set elements, as an abstraction, are not well suited to describe such complexity.

One way to address some of these problems is to augment usual interface descriptions with a formally specified protocol for using interface elements. PSL is one interesting example [2]. However many software designers find using formal description techniques too demanding. Also since availability of component services depends on the state of the component, a protocol that does not refer to the state of the component may get rather complex. UniCon [3] promotes an architectural approach that re-allocates the complexity of interaction from components to connector objects. Connector objects may then be used to deal with complexity of interaction. This re-allocation however requires an unusual partition of components and shift in design and implementation practices that may be hard to accomplish in reality. It is also not clear how composite connectors can be supported and used.

We tried an approach using a slightly different partition of the component that does not necessarily require a change in the design and implementation of software components and supports composition. Rather than viewing interfaces as flat sets of names we interpret them as objects in their own right — **interface objects.** Interface objects can serve to

- group related services as services provided by one interface object
- describe conditional availability of interfaces based on the state of the interface object
- compose higher level interface objects using object composition techniques

In essence, interface objects open a part of the component implementation that is relevant for understanding the interactions of the component and thus serves as precise description of its interface.

A very well known example of usefulness of interface objects is provided by *iterators*. Lists, sets, arrays, sequences, trees, and other composite structures must provide iteration service to the clients to obtain contained elements one at a time. It used to be a common practice to mix iteration interface with other services provided by the composite structures such as adding or removing an element. This was problematic in a number of ways. Iteration service may include several operations such as initialisation of iteration, iterating, testing iteration state, etc. How can these operations be separated from other operations supported by a composite structure? Iteration has state, like for example the current element of the composite structure. How to separate this state and its representation from the state and representation of the composite?

The answer to these questions was found in the form of iterators. Rather than providing iteration services, a composite structure can provide iterators. Iterators are interface objects that group together and encapsulate iteration operations and state. They can be composed with other interface objects using all available object composition techniques to structure the interface.

Consider an application in which a container (a list of names for example) needs to notify iterating clients when a new element is inserted during an iteration process. On the implementation level the component that implements the container will have to add notification and transaction management services. In the interface objects paradigm the composite interface would be composed of iterator, notifier, and transaction manager, rather than a flat list of all the services provided by the component.

2.3 Interactions

Interfaces are commonly described as sets of elements such as signatures of callable procedures, or types of messages accepted by the component. This is probably due to the fact that these are the elements with which interactions are implemented in the programming languages. These are indeed the "points of contact" between different components on the implementation level. However from the modelling point of view a procedure call or a message often does not constitute a meaningful unit by itself, but only as a part of a longer interaction, a scenario, perhaps involving several components.

Therefore we specify interface of a component as set of interactions or scenarios that accomplish a meaningful task from the modelling point of view. Functions, procedures, messages and events are the simplest kinds of interactions. We call them atomic. Composite interactions are composed of atomic and composite interactions. We do not have a well-defined formalism for composing interactions, but use instead common techniques for describing data and control flow abstractly with scenarios, object interaction graphs, message sequence charts, statecharts, pseudocode, etc.

Since interactions are allocated to interface objects that have state, composition of interactions may use this state to express conditions for sequencing component interactions. Interactions may be described by various means. Message sequence charts are probably the simplest widely used notation that can describe interactions. MSC however has well known limitations, like for example in describing partial or unspecified order of events. For more complex cases, where SDL was used as an implementation language, we used SDL process diagrams to specify interactions in component interfaces. Two components may be connected if their interfaces include compatible interactions.

3. Structuring TAPI Interfaces

In this section we describe a case study in structuring interfaces. We also include where appropriate descriptions of several additional techniques we found useful in our work.

3.1 What is TAPI

The Windows Telephony Application Programming Interface (TAPI) is an application programming interface standard promoted by Microsoft and Intel for telephony and call control under Windows. Its purpose is to insulate programmers and users of telephony application software from the complexities of the underlying telephone network. TAPI is a component of Microsoft's Windows Open Services Architecture (WOSA), and as such it consists of an API used by application and an SPI (Service Provider Interface) implemented by service providers (makers of hardware). There is a software component that sits between API and SPI, called TAPI.DLL. It is shown in Figure 1. TAPI.DLL acts as a "broker" that routes requests and replies between applications and the appropriate service providers. TAPI.DLL also implements a number of TAPI functions internally, like Multi-application features, address translation, etc.

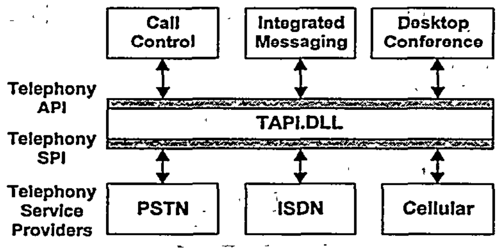

Figure 1. TAPI within the WOSA model

TAPI allows applications to control telephony functions. This includes such basic functions as establishing, answering , and terminating a call. It also includes supplementary functions, such as hold, transfer, conference, and call park found in PBXs (Private Branch Exchanges) and other phone systems. The API also provides access to features that are specific to certain service providers, with build-in extensibility to accommodate future telephony features and networks as they become available.

For the application programmer, TAPI abstracts telephony into just two kinds of devices, line devices and phone devices. By generalising telephony functions in this way, TAPI operates independently of the underlying telephone network and equipment. It isolates the application from the network complexity. TAPI is designed to be a superset of telephone network capabilities, allowing all networks to be modelled, like POTS, ISDN, PBX, cellular, wireless, etc.

3.2 Separating Interaction Domains

TAPI includes 115 functions and 21 messages. We can significantly simplify design of TAPI applications by imposing structure on TAPI and its description. First we identify the domains in which TAPI operates. TAPI is not a large component and in a sense is domain specific. It could be a part of a larger component that would also address, for example, content processing required by telephony applications. However even TAPI may be decomposed into two domains : *configuration* and *operational* (See Fig. 2).

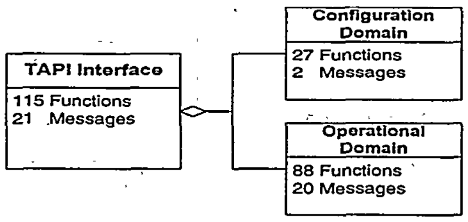

Fig. 2. Partition of TAPI by domains of interaction

Configuration domain provides the services for creation and shutdown of connections between TAPI and applications, negotiation of API versions, filtering of status messages to be received, querying device and address capabilities, etc. *Operational domain* contains the interfaces used to implement telephonic applications. Operational interfaces provide services for opening and closing devices, call processing, call and device monitoring, and media control functions, etc.

3.3 Finding Interface Objects

How to identify interface objects of a software component? Most software components play multiple roles even in the same interaction domain. Multiple operations, states, and some abstract behaviour are naturally associated with different roles. We use component role identification as the main guiding principle for identifying interface objects.

Roles are a very useful concept for object design. Reenskaug views objects as synthesis of multiple roles [4]. We structure interfaces of existing software components by identifying their roles and representing them as interface objects. Lea provides another detailed presentation of concepts related to roles and objects in [2]. Since we rely on roles as the main heuristic for identifying interface objects we often use both terms to mean the same.

After we have separated TAPI into two domains, we look for roles played by TAPI in each domain. In the configuration domain TAPI plays the roles of line device manager and phone device manager. They are responsible for the configuration and management of devices of two different classes. In the operational domain TAPI plays the roles of line device, phone device, and call. TAPI also can be used as a server for *assisted telephony service*. Fig. 3 illustrates the result of this decomposition.

Interface objects should not be confused with object-oriented (re)design of application. When an application is developed using object-oriented approach interface objects may be directly represented by implementation objects. However, in general this is not necessary.

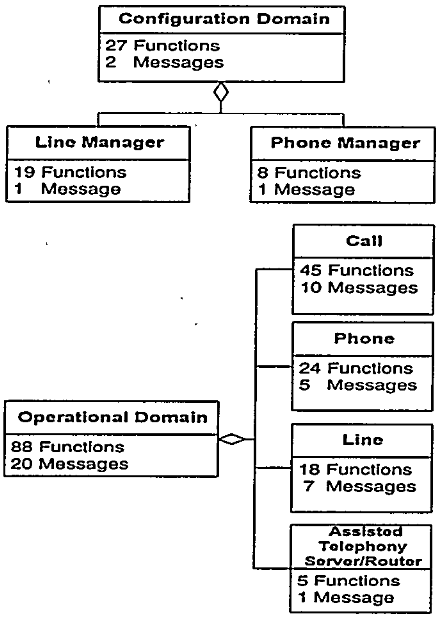

Fig. 3 Interface Objects of TAPI in Configuration and Operational Domains

3.4 Classify Interface Objects

Roles played by different components in a particular domain may exhibit a significant degree of similarity. In order to localise information, similarity and variability of roles can be represented using role classification. TAPI does not provide a natural illustration for this step. This is because design of TAPI does not provide a mechanism to represent specialisations of different device types, but treats them through an escape interface.

The idea of role classification is simple and is very familiar from classification hierarchies used in object-oriented design and programming to factor similarities. Earlier we explained that iterators are interface objects that represent roles played by composite structures or container objects. In STL, for example, iterators form a rich specialisation hierarchy.

3.5 Specifying interactions

Operations provided by roles are not independent of each other. Often an operation represents only one step necessary to acquire a meaningful service. This information may be represented by grouping such operation into services provided by an interface object. We use meaningful services as the main source for identification of interactions included in the interface of an interface object. For example the *call* interface object provides service *call handoff* that contains operations *lineHandoff(),* *lineSetCallPrivilege(),* and message *LINE_CALLSTATE, LINE_CALLINFO.* The structure of *call handoff* interaction of TAPI *call* interface object, may be described by a message sequence chart that specifies sequencing of component operations and events. Conditions of access to component operations can refer to the state of the interface object that provides the service. Since we rely on meaningful services to identify interactions we often use both terms to mean the same.

Fig. 4 Interface of *call handoff* service as interaction of several components specified with MSC.

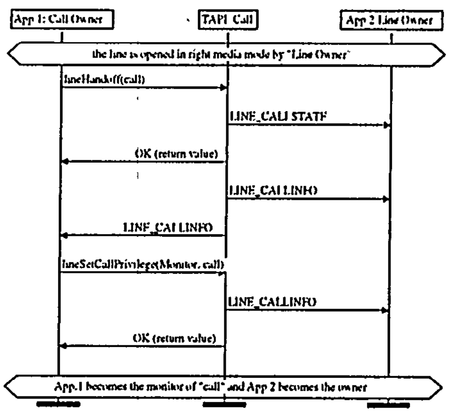
App 1: Call Owner
TAPI Call
App 2 Line Owner
the line is opened in right media mode by "Line Owner"
lineHandoff(call)
LINE_CALLSTATE
OK (return value)
LINE_CALLINFO
LINE_CALLINFO
lineSetCallPrivilege(Monitor, call)
LINE_CALLINFO
OK (return value)
App.1 becomes the monitor of "call" and App 2 becomes the owner

Fig. 4. MSC for service *call handoff*

3.6 Classify collaborators

Roles may provide different interfaces to different users or collaborators. Even in the programming languages some basic mechanisms are used to control visibility of operations. In the interface definition it is especially important to indicate the intended user of an interface. Specifying intended users by identity may be rather ineffective since we try to achieve independence between different components. Establishing a fixed classification of possible users for a component is restrictive, since there is rarely one classification that is useful in all situations. Therefore we include in our interface structuring a step to classify possible collaborators. Such a classification is extensible and does not need to be unique.

Each role of TAPI also provides different interfaces to different kind of applications. The services available under each state are not visible to all kind of users, some parts of them may be only visible to specific kinds of users. First we have to classify different types of TAPI users by so called collaborator's roles. The roles of TAPI users and the relations among them are shown in Fig. 5. TAPI applications can be divided into call users, line users and phone users. They can be further classified respectively into call monitor and call owner, phone monitor and phone owner, line owner and line monitor.

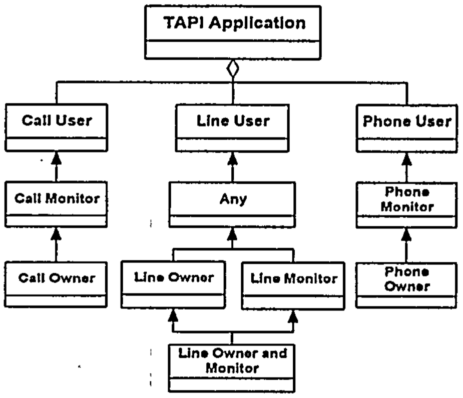

Fig. 5. Types of TAPI users

3.7 Visibility of services to collaborator classes

In our example, a call or phone owner is a specialisation of call or phone monitor. It can access the services that require owner privilege, and the services accessible by a call or phone monitor. The situation is different for the line users. A line owner and a line monitor have their own special services available from

TAPI. The services for a line monitor are not all visible to a line owner. There is a class of line users with no special priviledges called "Any", and a class of line users that combines line monitor and line owner priviledges.

3.8 Service availability in different states

Roles (and thus interface objects) may have different states that affect their capability to engage in different interactions. As interactions correspond to meaningful services provided by roles, one may say that service availability depends on the state of the role. For example the *call* role has 14 states which affect the availability of some of its services. For example you can only send *user-user* information through a *connected* call, and you can answer an inbound call only when it is in *offering* state. So through analysing the states of each role, we can further group the interfaces and describe their availability. Example state classification trees of role *call* and role *line* are given in Figure 6 and 7. The services listed in each state in Figure 5 and 6 are only available when the role is in the corespondent state. The services listed in the "Any State" are always available.

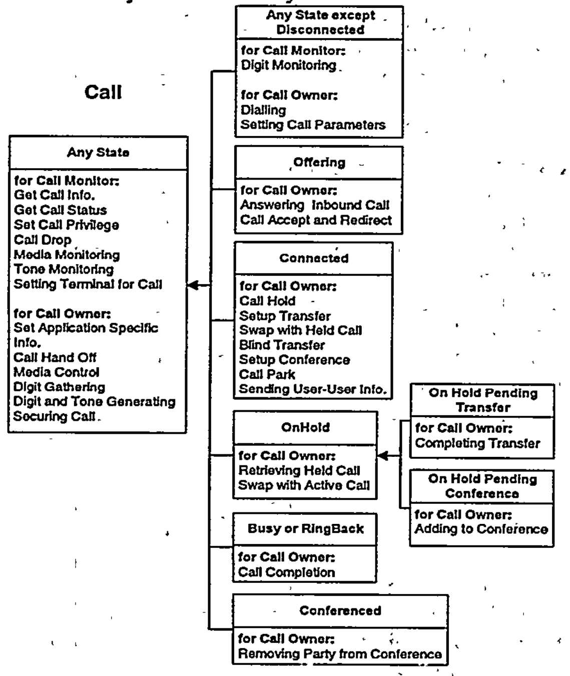

Figure 6. Grouping the interfaces of role *call* by states and users

4. Conclusion

The ideas presented in this paper are based on successful experiences of re-structuring and re-describing interfaces of two existing telecommunication products of very different size and complexity. We see the contribution of this paper in provoking thought and discussion on an important area of architectural design, that is often ignored. We see our own ideas as an example of a possible approach, rather than an in-depth analysis of the field.

We continue refinement of the discussed techniques for structuring and describing interfaces by applying them in new projects. We are also interested to study the potential of the design paradigm, based on interface objects and interactions, to improve reuse and evolution of large software systems.

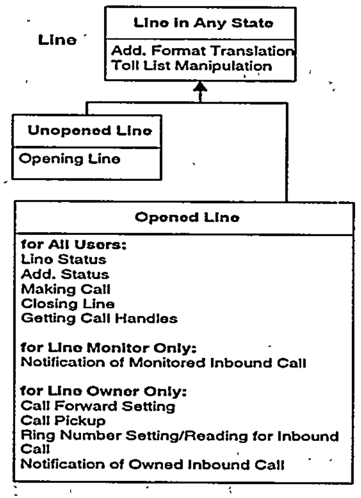

Figure 6. Grouping the interfaces of role *line* by states and users

5. Acknowledgement

We wish to thank the members of Nokia ARES team Anssi Karhinen, Juha Kuusela, Tapio Tallgren, and Juha Tuominenfor their help in preparing this paper. This work was in part sponsored by the European Commission.

6. References

[1] Windows Telephony API -- Programmer's Guide, Microsoft, 1996.

[2] Doug Lea and Jos Marlowe, Interface-Based Protocol Specification of Open Systems using PSL, SUNY at Oswego / NY CASE Center, Sun Microsystems Laboratories, Technical Report, 1995.

[3] M. Shaw et al., Abstractions for Software Architecture and Tools to Support Them, IEEE Transactions on Software Engineering, 21(4), April 1995, pp. 314-335.

[4] T. Reenskaug, P. Wold and O.A. Lehne, Working with Objects - The Ooram Software Engineering Method, MANNING Greenwich, 1996.