Author's Foreword

The ever-expanding field of Software Engineering is in need of targeted textbooks. Existing books are often too thick and full of more words than necessary to express a concept, in addition to lacking in engineering substance. This fact is especially a drawback when the audience is considered: usually the third or fourth year undergraduate students confront the topic in the classrooms. These students come from a line of deterministic small model manipulations and all of a sudden they get introduced to social content - engineering approaches that have a considerable "art" component, yet more: managerial considerations. Not having appreciated the real difficulties in the industry, exposition to problems that are neither tangible nor have unique or optimally correct solutions further repels the novice reader. For the experienced engineers, the need for a concise presentation in the text is preferred even more.

Another more important aspect of the evolving software engineering practice is a need to shift the existing paradigm towards modern foundations. Young minds should be freed of old restrictions inherited from primitive technologies. Now there is sufficient hardware and conceptual maturity to devise methods merely targeting the ideal. That is why the later chapters are dedicated to domain and component orientation. We could not afford to selectively contain the contemporary approaches in this book. The classical heritage should also be conveyed even for the modern engineer at least to be able to relate to the existing situation. Also the intention to utilize the content as a textbook (in addition to introducing the innovations to the industry) necessitates the containment of the common body of knowledge. The classical part displays some basic concepts that are universal and will support the new orientations.

Is the extensive wording not justified? It is serving a purpose for sure. The introductory software engineering book accommodates concepts more than anything. This includes abstraction that in turn is a key instrument in sorting out the terminology and their meanings. Describing such content is not easy in limited space. Our intention is to present the introductory concepts concisely, in the first chapter. As far as techniques are concerned, no matter how much detail is introduced it will never be sufficient within the realm of a book. The technologies find their extended definitions only in the industrial environments.

To sum up, the book is organized in three parts:

1- a summary of the traditional approaches,

2- modern approaches, and

3- case studies.

The first part can serve as a common textbook extending to the following part to prepare the novice software engineer to the expected near future challenges. For the experienced, the initial chapters will serve as a reference and the second part offers an alternative paradigm with orders of magnitude improvement intended in development performance. The new orientation is the natural, simple, as yet powerful methodology other engineering disciplines have attained and are using with success.

TABLE OF CONTENTS

Chapter 1: Software Engineering Concepts

Introduction

Few fields in the 21st Century are enjoying the popularity of Software Engineering. The demand for software is increasing exponentially. Wider populations are aware that every artifact will include software components especially as its vital part, in the near future. Yet this attractive field of engineering suffers growing pains due to its infancy and also is required to grow a lot faster than others have grown in the history. Although lacking infrastructure, techniques, and philosophies, improvement methods and tools for software engineering are to be offered in a rush. Half a century is experienced already after the introduction of the modern computer and yet no "silver bullet", or panacea, has emerged; the effort is continued for finding at least acceptable *steel* bullets, or standard approaches for solving complex problems. Engineering techniques can help in the more efficient development of bigger software without requiring more resources - especially personnel. This chapter will introduce the software engineering concepts that require the appreciation of the software crisis phenomena that is partially reflected in Figure 1.1. This pessimistic sounding introduction is not for discouragement; on the contrary it is expressed to expose the importance of this new discipline. Engineers enjoy challenge, after all.

Figure 1 symbolizes the growing gap between the offer and demand in the software market. Offer is assumed proportional to the number of computer scientists and related engineers together with their capacity to develop software.

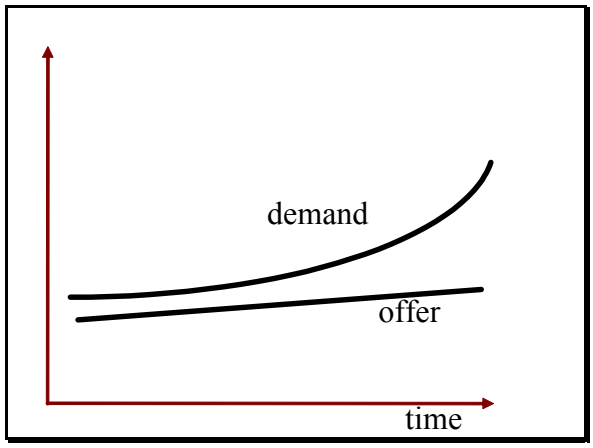


Figure 1.1. Software demand versus supply

It is also known that software projects are usually behind schedule and over budget. A majority of the projects result in failure. In the beginning, this could have been due to the lack of engineering approach for the construction of big software systems. Later, inadequacy, insufficiency and improper application of engineering practices kept the failure to sustain itself as an accustomed property of the field. The crisis has two sources:

1. Growing gap between demand and supply offering
2. High failure rates in supply offering.

Naturally searching for a solution, the software engineering field got established, offered different concepts and methods, and even came up with improved versions for the very definition of software. The struggle is still in progress. It should also be noted that the improvement in the engineering practices is shadowed by the increase in the complexity of the growing demand. This is the reason for not being able to come up with acceptable levels of success despite considerable improvement taking place in our engineering practices.

On the other hand, this complex problem brought with it a very different way of thinking that is the exciting part of this new field. More control over abstractions and devised mechanisms to handle complexity supported the software professionals to attack the modern and serious problems incorporating high-technology solutions. This kind of a merited expertise in turn propagated to other engineering disciplines. Software professionals are usually very enthusiastic and dedicated technicians in their jobs and they believe they are helping the society in important ways. Rapid advances in Information Technology are making improvements and impacting social life. A revolution is happening, faster than those happened in history. The comparable phenomena are the invention of writing, and then, printing press. The computational element (such as the processor) took the initial credit as

the major fueling log of this revolution. Soon, it was software that made the hardware useful and was acknowledged as the concept that holds the keys to this new transformation.

Software

In the early days a program was called software. Today the definition is more involved, having expanded over the years: documentation and data, as well as the advanced understanding of the 'code:' system of programs. Small programs that can be generated by one person in a week or a month are not within the scope of software engineering. They still can be called programs. Problems begin with the enormous complexity (size) of the software. Such a product comes in a very complex package:

- documentation,
- training packages,
- maintenance options, and even
- personnel dedicated to help in its smooth operation!

Software needs to be engineered as in the case of hardware we are more exposed to; like a car, or a television set. Before programmers start coding there is so much to do: the conception of the future product has to be investigated for feasibility, then requirements must be defined, and a sophisticated design is required to guide its implementation. After delivery, maintenance is a major issue. All of this "lifecycle" process requires supervision: a project management dimension that needs to be considered simultaneously with development. Actually engineering efforts that define the success of the project are mostly the ones that take place *before* implementation.

The software field brings a radically different concept after a line of different hardware fields have matured in engineering history. For the first time, the artifacts are not tangible; they are abstract systems of models. The cycle time for a small idea from formation in the mind to realization, is comparable to seconds, that is virtually zero when engineering practices are considered. Whereas in Electrical Engineering this time can be measured by minutes, while in Mechanical Engineering it can be measured by hours. Hardware products wear out whereas software does not. Also, mass production is not of concern for software; once it is developed, multiple copies can be created with ease, whereas hardware requires a production engineering, as well as design. Such differences necessitate special development care and technology for this new type of product.

Types of Software

There are different types of software and their development has to take into account the characteristics of their class. Although this categorization could be done with respect to different parameters, a list that contains the most frequently mentioned classes, without any categorization, is presented below:

- Business software,
- System software,
- Real-time software,
- Embedded software, and
- Scientific/engineering software.

A big share belongs to the business category. A banking application or a student course registration system would be in this group. System category includes the software that is written for some general-purpose computer hardware and enables its fundamental operation. Operating systems, some network or database management system applications are considered system software. Real-time software is generally fast. The requirement for an application to be real-time is that it has to respond within a pre-determined time limit. If this response requirement is real tight, then the term "hard-real-time" is used. Embedded software is written for a specific hardware and is integrated into the system, making the software an integral part of the machine. The scientific or engineering kind of applications are usually mathematically complex and the efficiency and correctness requirements account for the most of the complexity. Of course it is easy to extend this list for the contributors to complexity. One other important property could be security. Security recently became a very popular parameter, applying to many kinds of software, especially the ones accessed over networks where unauthorized access needs to be prevented. A safety-critical software assumes control functions that could be fatal if not operating properly.

Software Engineering

Software Engineering (SE) is one of the newest engineering disciplines today, branching out of computer science which began to take shape in 1950. There are two contradicting trends in the development of engineering fields. While a narrowing in any specialization field is observed, the need for interdisciplinary engineers is also growing. There is a constant definition of new fields of engineering with quite a few being related to software. However, most of those newly defined fields lack official and academic recognition. Now many Bachelor of Science degree programs in SE exist, not to mention the graduate programs. States grant professional engineer certificates in SE. Society of Design and Process Science (SDPS) has

established the first SE society. The founding date can be accepted as 1970, about the time when the first engineering approach was published and the first professional journal started as IEEE Transactions on Software Engineering by professors C.V. Ramamoorthy and R.T. Yeh. This acknowledgement process started a little earlier, and formally discussed in the 1968 NATO conference on Software Engineering.

The field is involved with the approaches and techniques used in the development of complex software projects. Software Engineers create and use such tools. Development involves a variety of tasks such as the elicitation of customer requirements, design of the solution, coding, testing, and maintenance.

Software engineering problems that are especially complex are referred to as "wicked problems." The most shocking property of such problems is that they cannot be completely specified until they are completely solved. There appears to be a bottleneck in the developer's understanding of the requirements for the project.

The deceivingly short conception-to-execution time causes an impression as if software were flexible; a less-disciplined developer would rush to implementations thinking that it would be easy to modify the system later. That is why sometimes, although a solution for a given problem may be present elsewhere, we want to start from scratch and with some over-confidence try to build everything ourselves. On the contrary, it is generally very difficult and increasingly more expensive to make changes as the project advances. Also newly created code is full of risks; especially having the potential for housing hidden errors as well as for extensive and difficult future modification. There is a need for achieving what other engineering fields have already, in terms of utilization of previously solved sub-problems. It is known that locating and integrating a previously written piece of code is more advantageous than to do it all over again. The principle is described with the *reuse* concept which is finding its best practice through the recently emerged "component" technologies. It can be concluded that SE is about to change its attitude about reinventing the wheel as common practice.

Difficult projects employ thousands of personnel, they may last nearly a decade to complete, and they are of huge sizes measured sometimes by tens of millions of lines of code. The costs can go up to hundreds of millions of dollars. Such a complexity cannot be handled by a linear production approach for coding. Reuse in different levels with higher-level tools to help with our intellectual control over the system is required. If the complexity is not dealt with, it will not be realistic to expect a successful termination of such projects in terms of correctness, timeliness, and quality. There has been a constant search for finding better tools, as the systematic approach has advanced considerably. Meanwhile, the complexity and the demand also increased, rendering a non-satisfactory improvement rate for success.

In the early days, computing sciences and related engineering were bound with the limitations the primitive machines imposed. Later the complexity suggested approaches that would enable better understanding of the problems – technology was not expensive anymore and it was more important to establish correct methods. Hardware technologies improved and software solutions were behind. Now is the time for formulating the best approach to efficient development procedures, rather than limiting ourselves to early hardware architectures.

Further sub-fields

As a major engineering field, SE also is composed of engineering fields defined for more specific tasks. An important one to mention is "requirements engineering." It is not uncommon to run into position titles such as "test engineer" and "design engineer" in the job advertisements. Despite its immature status, a strong foundation is present for a valid engineering discipline that corresponds to the biggest body of current problems.

It would be appropriate to classify the topics covered in SE into two major areas. Development and management are the two separate kinds of activities that have to be closely interwoven during a software projects lifecycle.

Software management is not very far from management science. This new field, however, is more human-oriented. Any information system needs to consider the human and organization dimensions on top of the technology dimension. Especially software development is very dependent on the individuals and the composition of the development team. Common management procedures are involved. A quick system definition can be made following feasibility analysis, and the project needs to be separated into tasks according to the system definition with completion deadlines and resources allocated to them. Budgeting, staffing, and tracking need to be performed. A critical procedure is the estimation of the cost, size, and development effort for the prospective system. All those mentioned procedures have software-specific attributes that have to be mastered for successful management.

Software development fundamentally being an engineering effort, displays some subtle differences due to the peculiarities mentioned in the previous sections. Understanding of the user definition is a major problem; that alone can be detailed as a separate engineering field. First there is the gathering process for requirements. Collected requirements should then be understood, analyzed and modeled for saving and further articulation. Then, a definition of the solution in terms of a design model follows. Coding, accounting for about 15% of the total project effort, is accompanied by testing, debugging, and integration. There is still maintenance to be carried out even after delivery. These activities could be classified according to different

parameters. For the novice software engineer, probably it is interesting to base every activity on coding. Figure 1.2 depicts the relative efforts coarsely corresponding to the activities: before coding, after coding, and coding itself.

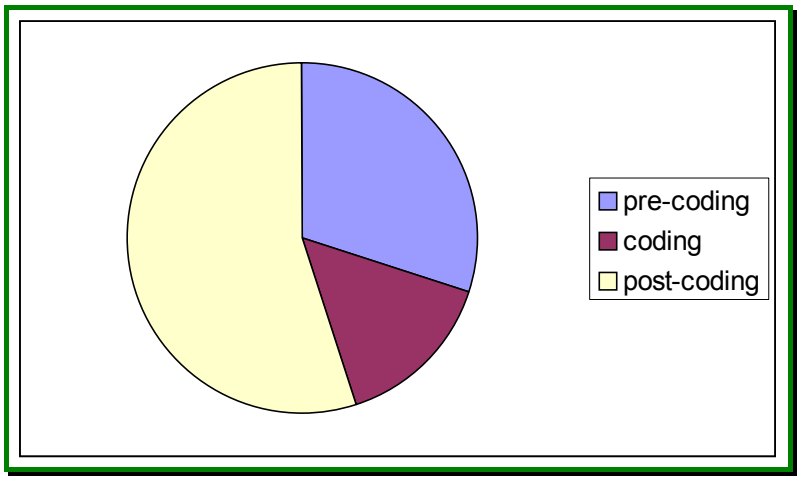


Figure 1.2. Effort distribution among development activities

Of course no activity alone should be recognized as the most important one, nor should be ignored. Without coding, there cannot be any products. Recent ads would list programming jobs as "software engineering" jobs. Most of the time, the employers mean programmers, not software engineers. Probably they could benefit from hiring real software engineers, who would conduct some programming as well as other SE tasks. Nevertheless, programming can be handled automatically to a considerable extent, today. Although it is important, pre-coding activities are more important. For example, design is where almost every parameter is fixed. The quality of the product is determined at this stage. It is of course still possible to abuse the programming duty by producing low-quality code from a good design. Requirements stage is strategic. Big errors made here if uncovered, are very likely candidates for jeopardizing the project.

This book's emphasis slightly tilts towards the development activities with further emphasis on requirements and design engineering. Some fundamental concepts developed since the inception of the field need to be digested. These are mostly related to the *process* that produces software, involving both management and development aspects. That is why an introduction to such concepts will be provided in the following sections, before the user is referred to the *development* and *management* related chapters.

Relation with other fields

Complex systems require the collaboration of different engineering domains. Most of the artifacts have a heavy software component. Besides having to define the interaction among the software and hardware sub-systems, a holistic view is necessary to yield a consistent product. Experience gathered along abstraction exercises earned some privilege to software engineers. This very fact that SE deals more with "abstractions" rather than tangible items now becomes a plus for the first time. The plight for solving a problem at higher-levels of abstraction had achieved some fruits in this field faster than it would in the other fields. The ideas were then duplicated in some other fields. Examples to this phenomenon are the Computer Aided Design (CAD) tools used in Electrical and Mechanical Engineering fields. Such tools have moved from primitive geometries to "features." A related outcome is that *programming* is replacing other forms of traditional design activities in such fields: now hardware engineers conduct their design through coding. Also to be mentioned in the next section, the process for development is manipulated as an important issue: Software Engineers have offered tools for constructing processes. Now process modeling is in any manufacturing or service industry.

Design is probably the basic engineering task. The psychological aspects of design have been investigated to aid the developers in this complex task. Herb Simon's [1969] work has been seminal for designers in any field. Later another question arose about the common denominator for all the design engineers – could there be fundamental design procedures independent from the application field? The abstract design notion [Tanik and Chan 1991] suggests that the earlier stages in the design of complex systems can provide a positive answer to the question. If the solution can be modeled disregarding the final implementation technologies, these early stages of the design will be conducting abstract design. Further articulating on this idea and its extension "system interface engineering" [Tanik and Ertas 1997], we can benefit from a late assignment of a "development domain" to any component. In other words, an established hardware/software co-development will start a design without first determining what part is mechanical, electrical or software. The more optimal solutions will develop in the later design stages, hopefully through the help of intelligent tools in the future. The abstract design idea is actually a disciplined explanation of the widely known "do not over-specify" principle.

The process

A process is an ordered set of tasks to accomplish a goal. In our case, the goal is the development and maintenance of software. The topic gained importance in other engineering domains also. "Business Process Reengineering" is a phase serious enterprises go through, before enacting any

automation. Now there are process consultants. Usually a systems-engineering effort is conducted that considers hardware, software, and other operational aspects of the future product. Then the scope of software engineering is demarcated, having its boundaries set by systems engineering. The hardware component could be negligible so it is not uncommon to see software engineers starting with this Systems Engineering activity. Personnel involved in these early phases were called "systems analysts." Even in strictly software fields, requirements engineering is frequently referred to as "system engineering."

Process engineering asserted itself by the turn of the century following the discovery that automating a system without process reengineering, may result in speeding up an inefficient approach. Also, it was found that quality products come out of quality processes and rather than establishing product quality standards the quality organizations started to certify the quality of *processes* within organizations. The leading process quality systems are the Capability Maturity Model (CMM) from the Software Engineering Institute (SEI) of Carnegie Mellon University, and the ISO 9000 series of standards from the International Standards Organization. CMM is in more demand currently, and other standards are also developing.

It all started in 1970s by Winn Royce through his "Waterfall Lifecycle." The lifecycle term is slowly being replaced by "process model." Waterfall was the first to point out that software projects are engineering projects and their "lifecycle" starts with the idea formation and ends only when they are ready to retire after long years of maintenance. Figure 1.3 shows the main activities proposed by Royce in a process model that suggests their order. Actually any box in this figure can be further detailed and some texts do present such different versions of the lifecycle. The requirements tasks can be separated into elicitation, analysis, and specification activities. Design is usually separated into logical and detailed (sometimes physical) design activities. Implementation starts with coding, sometimes considered simultaneously with debugging. Coding of the units may also be packed with unit testing. Testing is an important activity; after being conducted on the individual modules, it is applied also for the integration of the units. There is also system testing, which is carried at the developers' site, or even at the customers' site. Maintenance is also another engineering area but details of it usually do not take place in waterfall figures.

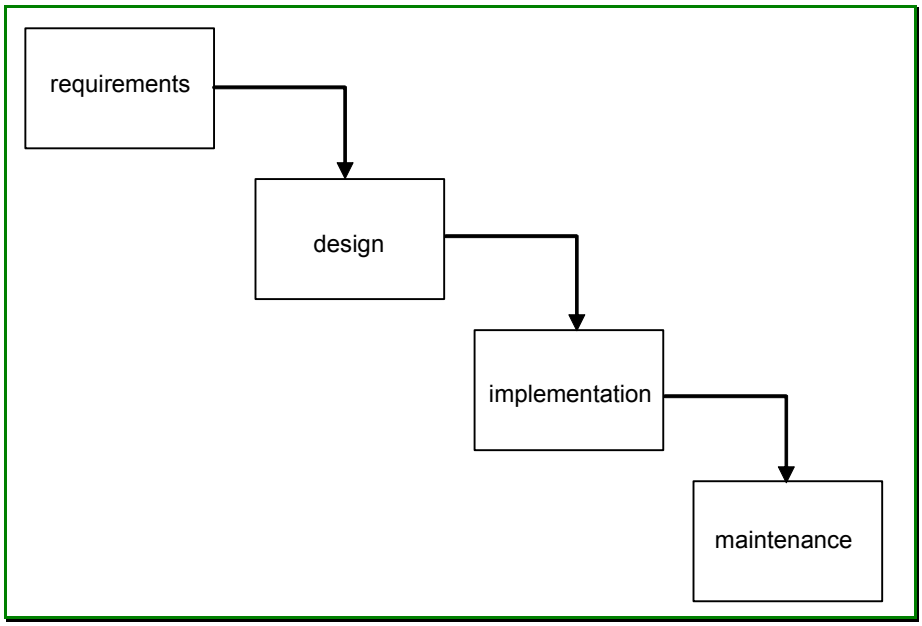


Figure 1.3. The Waterfall lifecycle

Since its introduction, this model has been the source of a considerable amount of discussion in the field. Although Royce may not have meant to strictly block a reverse flow of information across the tasks, Waterfall was taken so by the industry. Since software was defined to be engineering, the traditions settled in the *hard* engineering fields penetrated the new field. One activity had to be completely studied and correctly finished before the next could start. This was suggesting a "phased" approach with a linear flow of activities. Soon it was understood that previous activities had to be revisited frequently, and the software project had to be a dynamically defined one. The wicked nature of the problems was to be discovered. Then, modifications to the lifecycle soon followed, allowing arrows pointing in reverse directions even across far phases. It is also possible to find sources citing Waterfall with feedback arrows, and calling the one-way version, the "baroque" lifecycle. Nevertheless, either approach can find very appropriate problem classes and render the modern process models less efficient. Mostly, new projects follow other approaches that are "evolutionary" rather than "phased." Now we accept that the definition of the problem will change during development and we have to keep our doors open for change requests.

The evolutionary approaches were appreciated through mainly two process models, namely Spiral (Boehm 1975) and Rapid Prototyping (Tanik and Yeh, 1980). There are a few more models that are widely acknowledged and so many projects select a process model among those to proceed. However, there is confusion around the terminologies related to the process concept. It

will be beneficial to present a perspective so that the reader can find better meaning to such frequently used terminology that exists in different sources.

Terminology

In this chapter so far, a *process model* and two *approaches* were mentioned. These terminologies will be discussed in this section along with other ones that are process related. There are *methodologies* that need to be selected before any development can start. Also the term *paradigm* is another important one that is used with different meanings. To start the discussion, the closely related two can be compared. *Process model* is the abstract version of a *methodology*; a *methodology* is a detailed step-by-step definition of how to conduct the development with respect to the coarse definition that is the *process model*. The *process* was defined early in the previous section. Modeling a *process* (almost always) graphically produces the *process model*.

We are not limited to a standard set of process models. Organizations often adopt one and modify it for their specific needs. There are Process Modeling tools to aid in the formation of such models. These tools are used in Business Process Reengineering in any discipline, as well as software. One such tool is Funsoft (Funsoft 2000) that facilitates the instantiation of similar objects classified into infrastructure, artifacts, and communication as shown in Figure 1.4. A network with flows can be constructed and some simulation can be run. Actually the model can be detailed towards methodological levels. Some organizations do have their own methodologies besides the established ones that are known by their proper names and have been documented in volumes of books.

Process Modeling is a closely related concept to *workflow* systems. Similar modeling media is employed for both concepts and the two phrases have been used interchangeably. Recently the workflow systems gained a meaning that relates more to implemented process models that enact the operation in an enterprise. Equipped with workflow engines and supporting editors, computer based automation can be adapted to similar organizations with minimal code modification or development. The operation modeled through the editor can be modified and the engine will interpret the modified model as a new running software system tailored to an enterprise.

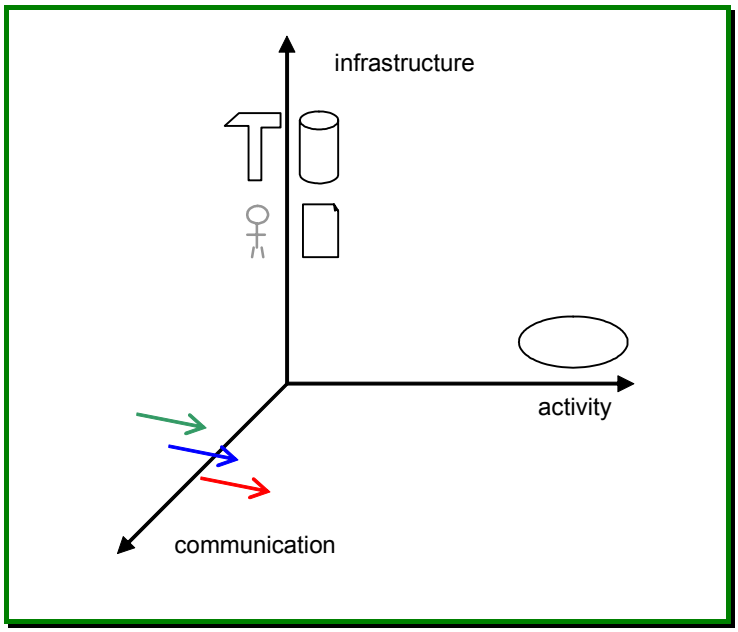


Figure 1.4. The 3 dimensions classifying the process modeling elements

A methodology has two dimensions: process and representation. The models are usually graphical. When classifying a methodology, its procedural dimension is referenced to decide if it is phased or evolutionary. The representation formalism suggests if it is "Object Oriented (OO)" or "structured." A newer term for the structured approach is "traditional." Today the developing contemporary approach is the "component oriented" development. "Component based" approaches have advanced to a level where commercial applications are feasible [Brown and Wallnau, 1998].

Before getting involved with the process related mechanisms, one should first have a world-view about software development. That is a *paradigm:* the main understanding behind the way the engineering will be carried out. Previously mentioned evolutionary and phased approaches are actually paradigms. Many Process Models can obey a single paradigm. Likewise, more than one methodology can be supporting a process model. Figure 1.5 associates the three terms with their abstraction levels. Abstraction is a fundamental concept and its levels are instrumental in explaining the real meaning of the terminology. If there is doubt about the intended meaning for some terminology mentioned in a context, trying to understand what abstraction level it applies to will help in clarification.

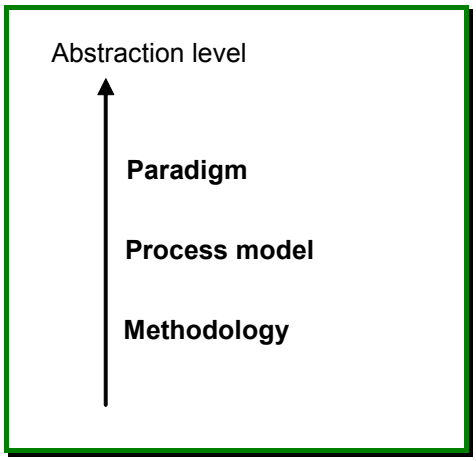


Figure 1.5. Terminologies in an abstraction level scale

Finally, a distinction can be made between the words *methodology* and *method*. Quite often they are used interchangeably. Two famous methodologies had named themselves as methods: Structured Systems Analysis and Design Method (SSADM) and the Fusion Method. A clear distinction and definition can be found in [Aktas 1987]; methodologies have bigger granularities and they are less formal – they are combinations of methods. Methods are smaller and more concrete, sometimes mathematical algorithms.

Other Process Models

There are variations of the Waterfall model; some split the individual boxes into more than one activity while others combine two activity boxes into one. Since a process model does not prescribe detailed procedures this is acceptable. They provide a general understanding about the ordering of the top-level activities. Probably the best alternative in the industry to Waterfall is the Spiral model, where each cycle of similar activities produces prototypes at its end. Mostly, the number of iterations is planned and there is an expectation of what should be included in each prototype. Figure 1.6 displays a spiral process model. This also has its variations. More detailed versions followed the initial model that introduced the "risk" procedures as an important task. Other tasks are planning, development, and evaluation that somehow correspond to activities in waterfall. A common interpretation of a cycle in the Spiral model corresponds to the phases of the Waterfall. At the end of an early cycle, a prototype for the requirements model is produced.

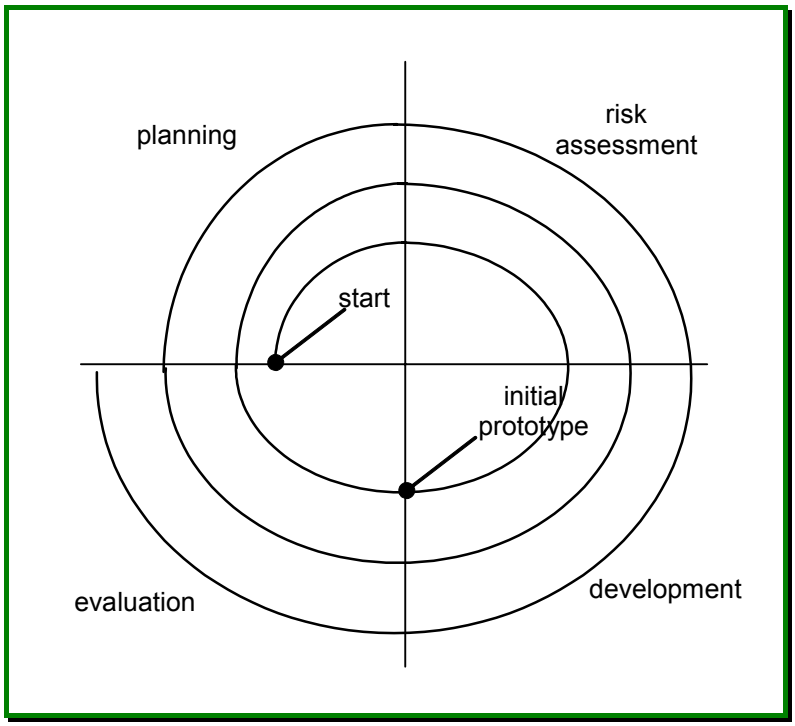


Figure 1.6. A Spiral process model

Rapid prototyping is another process model to follow. Owing to the evolutionary nature of software, build/evaluate cycles continue until the customer requirements are satisfied. This one does not escape variations also. A Rapid Prototyping process is shown in Figure 1.7.

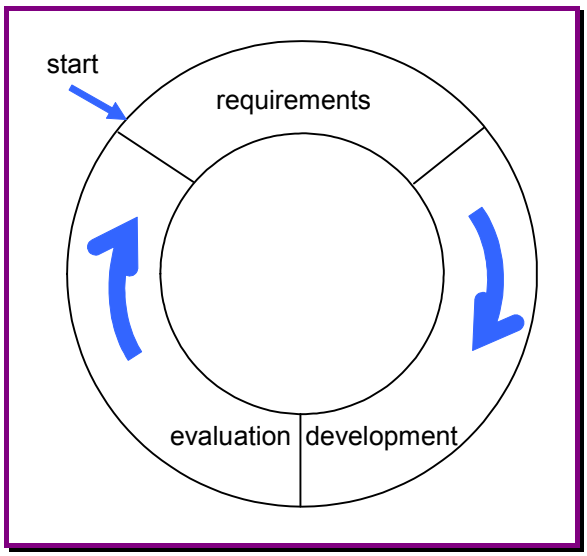


Figure 1.7. A Rapid Prototyping process model

Although there are more models, the mentioned set is found to be quite representative of the major industry usage. Incremental Delivery is a practical and relatively new process style where the whole system is separated into "builds" before development. Builds are developed and delivered to the customer. While the earlier build(s) is (are) in operation, the new builds continue development and, once finished, they are integrated into the system. This is another divide-and-conquer avenue with the additional benefit that the customer can get some functionality earlier than having everything at once. Also capitalizing on the modern tools the *Rapid Application Development (RAD)* is another option.

One OO methodology, namely the Unified Modeling Language (UML) [Booch et al. 1999] has been accepted as a standard in a very short period of time. This is despite the fact that its process dimension is not as popular as its modeling formalism and it lacks methodological level guidelines. The earlier UML texts encouraged their "iterative waterfall" process model. UML has been the second major attempt to combine the best practices of different methodologies – the earlier being the Fusion Method. Furthermore, UML is supported by the Rational Unified Process (RUP) for its missing methodological dimension.

Modeling Formalisms

There are different representations involved in SE, starting with the programming languages at the bottom of the abstraction levels and going up to requirements specification languages that can be textual or graphical.

There has always been an effort for providing better languages and models. Programming languages have advanced from the lowest-level machine formats to higher-level languages. Object orientation is an example. It is both a type of programming language and a foundation that guides software engineers in conceptualizing their models. The trend appears to be inventing new technologies followed by an engineering approach developed to utilize those new technologies. After assembly languages, FORTRAN and COBOL arrived as high-level languages. The advancement of the languages through gaining abstraction levels was thought to be a way to fight growing problem complexity. Initial SE approaches have been called "structured" because they targeted code creation based on the formerly available structured languages. PL1 and Pascal were pioneering the consciously structured programming for the general public. Three mechanisms have to be represented in the language for it to be structured. Those three mechanisms relate to execution that is:

- sequential
- conditional
- repetitive

Actually, the earlier SE approaches did not have much to do with structured programming concepts.

In any case, those initial SE approaches utilized different tools to model a system. As a result of conceptions inherited from the early languages and due to being conditioned to less capable machinery, different aspects were isolated in different model views to define a system. *Data, function*, and *structure* appear to be the 3 dimensions a software model is based on. These same concepts constitute the main design activities in the traditional approaches. However, object orientation demonstrates that at least data and function could be considered as a unity. We probably owe the separation of function from data, to the history when programmable calculators had to accommodate two different memory organizations. Data words had to be long and instructions were extremely short – an instruction is the selection of one out of a handful of keys on a keypad. Memory was expensive and small instructions were not desired to waste huge word lengths dedicated for data operands. Figure 1.8 depicts such a memory organization.

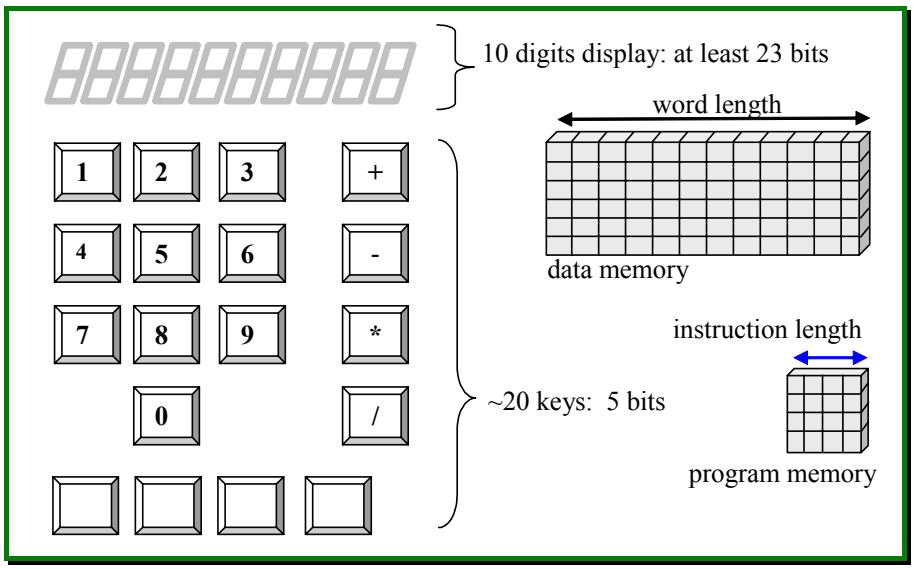


Figure 1.8. Different memory requirements in calculators

Object orientation is a step towards better models for human cognition. Still, the fundamental divide and conquer procedure should not be based on "data" or "function" both of which are more abstract entities when compared to "structure." The developing contemporary approach (i.e. component oriented) should be based on structure [Dogru 1999]. Among those three concepts, structure is not theoretically required; it is necessary for easier understanding. However, any modeling should account for data, function, and "control" to be *Turing Machine* compatible; to be able to represent any kind of an executable process such as a computer program. Sometimes, the control dimension is implied by the structure modeling facilities such as the "structure chart."

Modeling emphasis in different approaches

Based on modeling formalisms, so far traditional, OO, and component-oriented approaches have been mentioned. Although any approach has to represent all of the necessary dimensions, the approaches all have a primary view of concern. This emphasis is sometimes deliberate and sometimes by coincidence.

In the traditional era, the "function" concept had top priority. Although the modeling activity could start with data and structure, and end up with functions, the result was a network of functions calling each other. That was the view we had for software: functions calling functions. Later object orientation shifted this emphasis to data. Data structures are a connected set of data elements plus a set of access functions. Objects are almost exactly asserting this view. Although the methods inside objects are not limited to only "access" functions, any access should be allowed only through those

methods for correct OO practices. In traditional models, data abstractions were modeled by "entities" in the Entity Relationship Diagrams (ERD). Also shown were relations among entities. In class diagrams (the fundamental OO model) classes replace those entities and the relations are retained from the ERDs. Later component technologies introduced the structural pieces that pack data and function for integration into any system [Altintas 2001]. To utilize the component technologies in an engineering approach, the whole lifecycle must consider the part/whole relations where parts can be as small as components. Earlier approaches consider lines of code for the smallest "part." Figure 1.9 displays the emphasis dimensions for the three approaches.

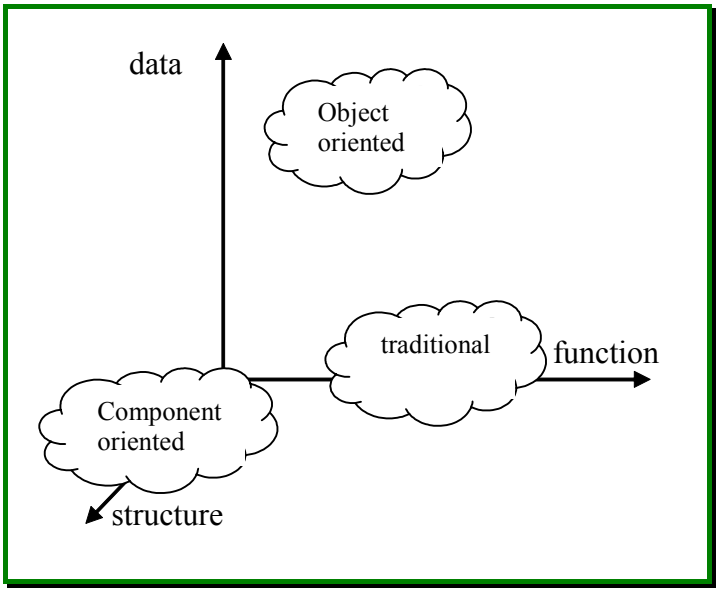


Figure 1.9. Modeling emphasis for software development approaches

Structure is a natural dimension for decomposing a complex system. Such decomposition is easy to understand and the composition of readily available components is fast to conduct.

Selecting appropriate methodology

So many approaches and their historical evolution were mentioned. Also some detailed methods will be introduced in the later chapters. One might ask at this point how to determine what approach to use. As there are many different solutions by different programmers to the same problem, there can be different selection of approaches once a project is given. Experience is always important and SE is very much human oriented. Nevertheless, there are established methodologies and to a lesser extent, guidelines are provided for a given a project.

There are benefits of following the current trends - especially standards. Some approaches are in fashion for some time-frames. During this period, if the widely accepted tools and procedures are applied, the project will enjoy support availability, relative ease in replacing personnel, and appreciation by a wider audience. Today, evolutionary approaches with OO modeling especially in UML are in favor. Of course, the project specifics and organizational aspects will affect the selection of the methodology. Sometimes the team has to follow a more practical decision rather than a theoretical one: availability of tools (and related support) may dictate the methodology selection. Engineering was defined heavily as a design task. Now it is time to also say that engineering is actually optimization – other tasks are the application of known techniques. In this case the optimization is a high-level one: better modeling versus better tool support.

The early doubts about OO approach's adequacy for large-scale projects seem to fade away. Any given product might have a structured approach history and an engineer may be assigned a project that is the improvement over this product. For minor modifications, the initial approach and its documentation can be followed. If it is estimated that the change will demand considerable complexity, it is an option to re-engineer the existing project and build it over using a modern approach that is most probably OO. As a result, an easier to modify version will be achieved.

There is also a trend to utilize "components" in an effort to minimize new code writing. Depending on the availability of the components, a complete "component oriented" approach can be selected. No matter how sound the idea seems, it is a new one yet to be approved by the industry for actual large-scale, industrial projects. "Component Based" approaches [D'Souza 1998] are, however, more realistic with respect to maturity of the established tools. The established OO tools now allow the patching of components. This way, existing codes can be reused with the component protocols that bring a structure to the integration procedure. Also the overall model is a proven one (OO): component based environments are usually OO – the component orientation is a radical change in the approach. If this discussion triggers the idea to develop any new code as components, the associated cost increase must be considered. This increase amounts to about five times more for such "reusable" version of any module. However, the advantage is that once the components are present, integrating them for different systems costs less than developing specific code for a target system.

Summary

Software is presented in this chapter as an engineering artifact, having its specific properties. It is highly in demand but very difficult to develop. It is difficult to completely define the problem for a software engineering project. Starting already with this difficulty the failure rate is further increased with

the immaturity of tools this new engineering field is equipped. There are engineering approaches that bring some determinacy to the delivery time and product quality. These are mainly methodologies guided by paradigms and process models. All such mechanisms have continuously improved along with modeling formalisms since the computing related disciplines were founded.

There has been a constant trend towards addressing higher-level abstractions in the programming languages and modeling media. Also the process concept that emerged as a key quality factor has improved earlier linear approaches to evolutionary processes that try to accommodate the dynamism in requirements. The ideas about abstraction and processes have been applied to other engineering disciplines as well. These help in the modern engineering practice that has to both deal with narrowed specialization fields and an interdisciplinary coordination. The interface concept emerges to be an important aspect defining how components will connect, especially for interdisciplinary designs.

The process should be tuned or even freshly invented to meet the challenges of a specific project, the realities of the organization, and the utilized infrastructure. Before any development can start, tools and such approaches have to be decided for selection. Software development is a heavily human-oriented task. The careful and skillful allocation of tools and procedures should be accompanied with a modern management and staffing understanding for increased chances of success in this very difficult but also a very rewarding field that is defining our near future.

Questions

1. Define the areas that are common and different between “hard” engineering disciplines and software.
2. Define example projects that are more suitable for each of the mentioned process models and modeling formalisms. Explain how they are suitable.
3. Explain why structure is important for the component oriented approaches and how data and function dimensions can be modeled in this approach.
4. Explain how traditional/object-oriented/component-oriented approaches differ with respect to problem understanding and problem representation.
5. Name two other kinds of software categories besides those already listed in the “software” section.
6. Software statements can easily be changed and the resulting operation can be observed very fast. Discuss the advantages and

disadvantages of this “flexibility.”

7. What are the potential risks in designing a new methodology for the new project your company has undertaken?
8. Since we are narrowing the specialization of the engineers in various disciplines, how can we successfully build systems that require the cooperation of components gathered from different engineering fields?
9. Software Engineering is heavily human-oriented. What measures can be taken to protect the project against a case where a considerable number of key personnel leave the project?
10. Is there any technique other engineering fields have benefited from, that you could propose to help also in the software field?

References

Aktas, 1987 — Ziya Aktas, 1976, *Structured Analysis and Design of Information Systems,* Prentice Hall.

Altintas, 2001 — Ilkay Altintas, 2001, A Comparative Study for Component Oriented Design Modeling, M.S. Thesis, Computer Engineering Department, Middle East Technical University, May, Ankara, Turkey.

Booch et al. 1999 — Grady Booch, James Rumbaugh, and Ivar Jacobson, 1999, *The Unified Modeling Language User Guide*, Addison-Wesley.

Brown and Wallnau, 1998 — Alan W. Brown and Kurt C. Wallnau, 1998, “The Current State of CBSE,” *IEEE Software*, September-October.

D’Souza, 1998 — Desmond Francis D'Souza, Alan Cameron Wills, 1998, *Objects, Components, and Frameworks With UML: The Catalysis Approach*, Addison-Wesley.

Dogru 1999 — Ali Dogru, 1999, “Component Oriented Software Engineering Modeling Language: COSEML,” *Technical Report TR-99-3,* Computer Engineering Department, Middle East Technical University, Ankara, Turkey.

Simon 1969 — Herb A. Simon, 1969, *Sciences of the Artificial*, MIT Press, Cambridge, Massachusetts.

Tanik and Chan 1991 — Murat M. Tanik and Erik S. Chan, 1991, *Fundamentals of Computing for Software Engineers*, Van Nostrand Reinhold, New York.

Tanik and Ertas 1997 — M.M. Tanik and A. Ertas, 1997, "Interdisciplinary Design and Process Science: A Discourse on Scientific Method for the Integration Age," *Journal of integrated Design and Process Science,* September, Vol. 1 No. 1: pp. 76-94.

Chapter 2: Software Project Management

Introduction

There are different approaches to software development. Management of software projects should also follow those different approaches closely. Luckily, most of the techniques and concepts used in software project management are common and are adapted in different ways for different development processes. This chapter presents main techniques and approaches for software project management.

Project planning is the umbrella activity that guides the other activities. Like most of the tasks, planning needs to be dynamic and adaptable. Nevertheless, with the information gathered after a preliminary analysis, a plan needs to be shaped to cover all the lifecycle of a project. This plan is documented along with other plans such as a quality assurance plan and a testing plan. All such documents will be revised and should be kept up-to-date during the development. Sometimes it is not unrealistic to anticipate software activities to produce more documentation than code. If this becomes an unnecessary burden, more flexible approaches become reasonable such as Agile methodologies and Extreme Programming (XP). The field has so far suffered though, not because of the over-documented projects but the under-documented ones.

Managing a project with so many parameters and unknowns would be a prohibitively difficult mission. Luckily some of the knowledge is available as templates in case the enterprise has established some process maturity. Project specifics are mapped to the components of the process, and tracking and process improvement are left as the major attention demanding activities.

Project resources

Planning involves project resources, estimation of cost and duration, staffing for the project activities, producing documentation, and tracking. The resources are:

- Human
- Software
- Hardware

Such resources need to be defined and their times for utilization and acquisition should be planned.

Human resources

Software projects are often multi-disciplinary. Most of the time, automation is requested for a process that is being conducted in conventional ways. It is possible that the software developer company is not familiar with the domain. Experts in the domain may be acquired to accompany the team. The roles tasks require may demand full-time or part-time contribution. Project plan should indicate exactly what role is allocated when and for what time duration. The durations will be estimated by techniques to be explained later. The roles include:

- Project manager,
- Systems analyst,
- Designer,
- Lead programmer,
- Programmer, and
- Test engineer.

However, it is often quite possible to extend the list with the roles, for example:

- Quality manager,
- Quality expert,
- Project secretary,
- System manager,
- Database Manager,
- Web designer,
- Hardware Leader,
- Network expert,
- Trainer.

Being costly, the human resources can be allocated through the formulas that suggest varying effort demands with respect to the phase of the project. Another difficult scheduling problem surfaces in case the organization has more than one project under development: the pool of personnel will be considered for varying effort demand, based on the scheduled tasks of multiple projects.

Software and hardware resources

Software tools are helpful in so many of the development tasks. Computer Aided Software Engineering (CASE) and Computer Aided Design (CAD) tools are the important ones. Compilers, word processors, and project specific tools may also be used. There are also tools that are used for management tasks. Besides tools, the reusable software components are also very important resources. Component technologies are being developed and they can be utilized in different ways:

- Common off the shelf components (COTS) are usually large-grained: they are substitute solutions for considerably big parts of the problem. Acquired per need, they usually are not part of the organization's library.
- Medium/small grained components can be bought individually, or as a library (sometimes called a framework). They may be part of some CASE or framework environment. They obey a component protocol or architecture such as DCOM or JavaBeans.
- Components developed inside the organization, may be complying with a strict protocol or could be loosely defined components – they may alternatively be called a library of functions.

Reuse is a concept whose benefits are not questioned; utilizing readily available software is simply advantageous in practically all cases. The defined architecture for reuse today materializes itself as component frameworks and consequently the component set constitutes an important part of software resources.

Process modeling tools help in the understanding of the workflow throughout an organization. Data-flows, task deadlines and the bottlenecks can be visualized through their usage. Scheduling tools help in assigning times and durations to the work breakdown structure. Estimation tools help projecting the overall time for the project, required effort, and manpower.

The bulk of the CASE tool usage is related to analysis and design modeling. Sometimes they help in evaluating the quality of the system under development. They also undertake the production of the skeleton code, and sometimes they generate code automatically for some of the routine functionalities. Compilers are inevitable. Some of them carry capabilities that qualify them as CASE tools. Debuggers, test case generators and text formatters are among the software resources. Prototyping and simulation tools help develop intermediate applications that give an idea about the future systems behavior.

For maintenance, there are also software tools that help with reverse engineering of existing systems. It is possible to analyze the code and retrieve design information out of it, even through presenting diagrams. Also

system software such as operating systems, network software and e-mail or media management, electronic conferencing tools are part of the inventory.

Hardware resources are various computers and the network equipment. There used to be mainframe computers which have been replaced by servers, such as unix workstations or, more currently, powerful Personal Computers (PC). The PC architecture has grown in different directions anywhere from home PCs to servers that can contain multiple CPUs. The inventory should include any other computers utilized as clients or stand-alone devices. Network equipment is the cabling, the switches and routers that make the computers connect to the network, and the dedicated servers for network management. Besides these fundamental technologies, buildings, rooms, and office equipment could also be assigned to a project.

Process Maturity

Not listed as a standard resource, any established process mechanism is a high-level and sometimes a vital asset of an organization. The documented know-how of how to plan, conduct, and monitor the project brings determinism to the quality and time-response of a development. The topic is discussed along with the process quality and standards. The Capability Maturity Model (CMM) developed at the Software Engineering Institute (SEI) is currently the most widely accepted process evaluation system [Paulk et al. 1994]. Five maturity levels are proposed for processes and they are substantiated with activities and further measures. These levels and their characteristic attributes are shown in Figure 2.1.

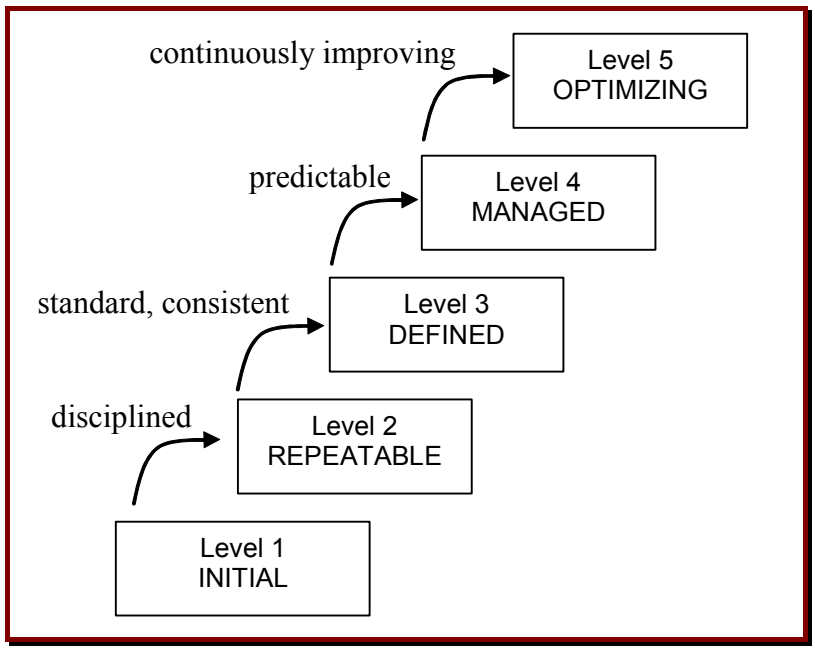


Figure 2.1. Five levels of the Capability Maturity Model

A Level 1 process is considered ad-hoc, where success depends on the unorganized actions of individuals. Organizations with Level 2 processes are more independent able to repeat the previous achievements of successful programmers. Level 3 is where a process standard is established throughout the company. Measurements of the achievements are expected as a major contribution for Level 4. At this point, tracking the process is facilitated by the measurements an organization would devise over its process. Finally, such measurements are exploited to modify the process for continuous achievement, in Level 5. Key process areas and activities are defined that correspond to different levels. There are other process quality models and the topic is gaining importance, as more software customers demand such compliance. The key process areas for four levels of the CMM are listed below:

- Level 2 (Repeatable):
    - Requirements management
    - Project planning
    - Project tracking
    - Subcontract management
    - Quality assurance
    - Configuration management

- Level 3 (Defined):
    - Organization process focus
    - Organization process definition
    - Training program
    - Integrated software management
    - Product engineering
    - Inter-group coordination
    - Peer reviews
- Level 4 (Managed):
    - Quantitative process management
    - Software quality management
- Level 5 (Optimizing):
    - Defect prevention
    - Technology change management
    - Process change management

Estimation and Metrics

Measurement is a key activity for project management [Pressman 1997]. There needs to be some indicators of how the process is advancing. In the beginning, we need to know how much work is needed to complete the project. For both initial estimation and tracking, measurement parameters are important. Other factors relating to quality should also be measured otherwise there cannot be a communication among the team members for objective assessment of any parameter and for definition of the goals.

The metric to be measured is the question. Especially for software, there are very few directly measurable parameters. Coupled with the inherent difficulties of the field, management can easily become frustrating because of the subjectivities in assessments. Luckily, some practical and empirical methods emerged that were verified in industry. There is a common language now for measurements and estimation in software discipline.

Base for metrics

Metrics can be defined as the degree to which a process or product achieves an attribute. There are fundamentally two different bases for measurement: (1) size oriented and (2) function oriented. Lines-of-code is the most frequently used unit. Actually thousand (Kilo) Lines Of Code (KLOC) is

used for most of the size oriented metrics. The alternative is Function Points (FP) that gives a numeric assessment of the total complexity of a software artifact. These measures for complexity are trying to give an idea about the overall "weight" of the software. Number of lines could be a misleading indicator. Same functionality can be obtained by varying code sizes depending on who writes it. The Function Points method is addressing this problem and presenting numeric results that assess the complexity ("weight") independent from the programming language statements. Also different languages will yield different number of lines for the same algorithm hence presenting another problem with the number of lines approach. On the other hand, lines-of-code is a directly measurable metrics that can be produced automatically. Before implementation is finished, the KLOC measure can only be estimated.

There are advantages and disadvantages of size and function oriented metrics. While size is easy to obtain and more objective, function oriented estimations can be calculated before any code is written. Some problem properties are required for calculating function points. These properties are defined before design anyway. Both approaches support estimations before development and measurements after development. The FP method, however, prescribes how to assess the complexity with the logical definitions. If no information about past development measurements exist, estimation could be performed using FP. Then it is also possible to convert FP to KLOC and continue with size-oriented parameters.

The following sections present size oriented and function oriented approaches to metrics. However, quite frequently those two approaches are used together. There are actually new methods supported by tools that combine the approaches and even add newer techniques. Object orientation addresses this matter and object oriented models provide specific input other than numbers of lines and other values. Also some dynamic modeling to account for changing effort with respect to time is included in such tools.

Size Oriented Metrics

Mostly utilizing the KLOC unit, process and product attributes are usually based on size. A product (or sub/intermediate-product) can be estimated or measured to be of certain KLOC size. Indirect metrics can be produced, such as productivity, that is, the Lines of Code per unit effort. Effort is measured by person-months. Productivity is reported in terms of KLOC/person-month. Likewise, cost can be calculated as Dollars/KLOC. Other frequently used metrics are errors/KLOC, defects/KLOC, Documentation-pages/KLOC, errors/person-month, LOC/person-month, Dollars/documentation-page.

The rest of this section covers lines of code based metrics owing to the established nature of such practices. However, other fundamental size

measures such as number of screens (or forms) are also being used especially after the popular demand for windows based graphical software.

There is some dispute in the utilization of size oriented metrics. For those who really need objectivity, more has to be accounted for. Are all the lines the same? For big projects, such inequality does not constitute an issue. Big projects inevitably contain code originating from different personnel and different environments that yield a natural averaging. As a result the meaning of one line makes sense from the complexity point of view. But, if the size includes comments then the size has a different meaning. Some size oriented metrics prefer to differentiate the executable instructions from non-executable lines (comments).

Measurement is less of a problem when compared to estimation, although still more difficult for software than for hardware. Most of the compilers are equipped with tools that can report the total lines of code. Estimating the size of a code to be developed is more difficult. In any estimation, past experience and any recorded information helps. With or without historical data, some techniques are employed to ease this difficult task of estimation. The "Lines of Code" (LOC) method is for arriving at an overall estimate of the product once the system is decomposed to a set of subsystems and estimations can be done over these parts. Of course overall size is a linear combination of the sizes of the subsystems. There is a fundamental problem with this approach: Complexity of the total is not equal to the sum of the complexities of the parts. For size only, this does not seem to be a big problem but for effort and derived estimations more care should be taken. The relation is not linear – some exponentiation is required! Equation 2.1 formulates this complexity law where C is for complexity, A and B are the parts of a software system.

$$C(A+B) > C(A) + C(B) \qquad 2.1$$

If this approach is taken however, some extra effort for the integration can be reserved and that could represent the excessive complexity. Table 2.1 presents an example for Lines of Code method where a project is suggested to have components such as User Interface and 3-Dimensional Geometry, for instance.

Table 2.1. Lines of Code Method

Function	Estimated LOC
User Interface	2300
Database management	3350
3D Geometry	6800
Peripheral Control	2100
Estimated Total LOC:	14550

Estimating size is the initial activity. After that, effort can be estimated especially if the efficiency of the organization is known. Other columns can be added to Table 2.1 for listing the efforts and cost, among other parameters,. per function. If the known efficiency for the organization reveals the value of 100 lines per person month, then the User Interface function would require (2300/100) 23 person-months.

Empirical estimation

The nonlinear relation between size and complexity is also observed in the measurements averaged across the industry. The general relation is in the form of:

$$\text{Effort} = A + B\,(\text{size})^{C} \quad \text{2.2}$$

There have been numerous studies for fixing the constants (A, B, and C) in Equation 2.2. The commonly used formula belongs to the Constructive Costing Model (COCOMO) [Boehm 1981]. This model actually is made up of three sub-models based on model sophistication. To be able to use this approach, after selecting the sub-model, the problem class has to be determined. Problems are rated with respect to the degree they are "embedded." This attribute of software has been listed in the introductory sections about software. If embedded, the project will be more complex than a project of the same size that is not embedded. Also real time software is more complex than regular business software. The project types are:

- Organic
- Semi-detached
- Embedded

The Sub-models for COCOMO can be listed as:

- Basic: provides effort and time, as a function of size.

- Intermediate: Same as Basic Model but includes a set of cost drivers for subjective assessments of product, hardware, personnel and project attributes.

- Advanced: Like Intermediate, but is also dynamic. Each phase such as analysis and design, are considered. The results are not constant values but they change as a function of project duration time.

Finally the model is presented as equations and the coefficients used in these functions are given in Table 2.2. The advanced sub-model is not included. Size should be entered in thousands of lines of code, time is calculated in months and effort is in person-months.

- Basic

$$\text{Effort} = a_b \text{ KLOC}^{b_b} \quad 2.3$$

$$\text{Time} = c_b \text{ Effort}^{d_b} \quad 2.4$$

- Intermediate

$$\text{Effort} = a_i \text{ KLOC}^{b_i} \text{ x } \text{EAF} \quad 2.5$$

$$\text{Time} = c_b \text{ Effort}^{d_b} \quad 2.6$$

Table 2.2. Coefficients for the COCOMO

Project type	a_b	b_b	c_b	d_b	b_i	b_i
organic	2.4	1.05	2.5	0.38	3.2	1.05
semi-detached	3.0	1.12	2.5	0.35	3.0	1.12
embedded	3.6	1.20	2.5	0.32	2.8	1.20

For a dynamic model, the following formula can be used. This is Putnam's equation. Now, the project duration must have been assessed before and it is entered in terms of years. Effort is still expressed in person-months.

$$E = [\text{LOC x B } 0.333 / P]3 \text{ x } (1/t4) \quad 2.7$$

There are other values this equation requires. B is the special skills factor and takes the values (0.16 … 0.39). P is the productivity factor. Some values for P are listed below:

- 2000: for real-time embedded software,
- 10000: for telecomm and systems software, and

- 28000: for business systems applications.

Using Putnam's equation, a project with a long duration can be planned for staffing. Typically effort curves start low and they climb up soon. Later there are few peeks corresponding to the different development activities such as coding.

Function Oriented Metrics

Independent from the coding style and languages, the FP method utilizes the "domain" parameters related to a project and arrives at an estimate [Albrecht 1979]. The equation below is used in the calculation of Function Points. The count total in this formula is calculated using Table 2.3. Also the complexity adjustment factors (F_i) are taken from Table 2.4.

$$FP = countTotal \times [0.65 + 0.01 \times \sum F_i] \quad \cdots\cdots\cdots\cdots\cdots\cdots \quad 2.8$$

Table 2.3. Count Total calculation for Function Points

parameter	count	simple	Average	complex	
# user inputs	m	3	4	6	m x 4 (average)
# user outputs	n	4	5	7	n x 7 (complex)
# user inquiries	p	3	4	6	p x 3 (simple)
# files	…	7	10	15	…
# external interfaces		5	7	10	…
				Count total:	

A few words are necessary for the usage of Table 2.3. First of all, if there are different numbers of items for a single parameter of varying complexities, they should be individually weighted and added to the right-most column. For example, if there are 15 inputs of simple complexity, 16 average inputs and 17 complex inputs, the number to enter to the right is 15*3 + 16*4 + 17*6, that is 211.

The next issue is what should be counted as one item? Is an address information one input or should we count the street name and street number

as separate inputs? The answer is in the way the information will be used in the system: other than the input procedure, if an item is going to be accessed anywhere as a data item alone for processing/printing, for instance, then it will count once. In other words, if the complete address will always be treated as a single entity throughout the program, then the whole address will be counted as one input. If for example, the street name or the city will be used independently such as in the example of listing all the cities in the system on one page, then city will be counted as one input and consequently a complete address will count for more than one.

Table 2.4. Complexity Adjustment Factors

Factor	Grade (0..5)
1 Reliable backup and recovery	
•2 Data communications	
•3 Distributed processing	
•4 Critical performance	
•5 Heavily utilized operational environment	
•6 On-line data entry	
•7 Input transactions over multiple screens (on-line)	
•8 Master file updates on-line	
•9 Complex input/output/file/inquiries	
•10 Complex internal processing	
•11 Reusable code design	
•12 Conversion and installation included in design	
•13 Multiple installations for different organizations	
•14 Design for facilitating change and ease of use	
Total :	

Extensions to Function Points

FP has been a useful metric for estimation teams. Soon after its introduction, some extensions have been proposed and used. Mostly Feature Points and 3D Function Points can be mentioned. In Feature Points, FP is virtually used as is, except for including the number of algorithms. Any function calculating a value is an algorithm. This parameter is added to the existing five in Table 2.3.

3D Function Points utilizes the Data, function, and Control dimensions. Parameters are organized in those dimensions before including them in the formulas. These dimensions are accounted for as below:

- Data: same as in Function Points.
- Function: number of transformations is also counted.
- Control: number of states contributes to the complexity as one count each.

Translating between the approaches

Once more, average numbers from industrial projects come into play. One function point is translated to lines of code for different programming languages. This translation also indirectly suggests the conversion ratios among different programming languages. The numbers taken from Table 2.5 can be used, to convert the FP based estimation to KLOC values: FP analysis can be conducted before development but KLOC values are required in the empirical models such as COCOMO. Now having the KLOC values, empirical estimation techniques can be utilized in the calculation of effort and time. Using the two approaches sequentially, one can arrive at time and effort estimations starting with the information that is available at definition time. The values in Table 2.5 have evolved in the past decade especially towards a direction that would increase the size/function ratio.

Table 2.5. Function Point equivalencies for Programming Language Statements (adopted from [Albrecht 1983])

Programming Language	FP/LOC
Assembly	300
C	125
COBOL/FORTRAN	105
Pascal	90
Ada	70
Object Oriented	30
4th Generation Languages	20

Also Graphical User Interface (GUI) based size metrics can be converted to Function Points. These numbers are more recent than those of Table 2.5. Nevertheless, they can be useful for a preliminary complexity analysis. Table 2.6 presents the FP equivalents of basic graphical user interface modules.

Table 2.6. Function Points for Windows structures

GUI Feature	Function Points
Message Box	4
Simple Dialog	3
Average Dialog	4
Complex Dialog	6
Simple frame	4
Average frame	7
Complex frame	13
Average file	7
Complex file	15

Scheduling

An estimation for the overall project may have been obtained. There is no guarantee that the project will meet the deadline imposed by the estimation or by the contract. Just like in the estimation by LOC method, the divide and conquer approach is a key factor in trying to meet the deadline. This time, the total effort is divided into tasks and each task should have its own deadline. Now, dependencies among the tasks, if delayed, may cause a problem and the tasks lying on the "critical path" will result in deadline problems. All the separated tasks may need different resources, or they may share some. Output of one task may be input to the other; such constraints suggest serialization and ordering of the tasks. This topic is intimately related to process modeling and starts with identifying a number of manageable tasks.

After the tasks are known, their dependencies are analyzed and their effort requirements, starting and ending times are assigned. Tasks should deserve the effort assigned. Also responsible personnel and input/outputs, as well as other resources should be indicated. It is also helpful to mark milestones for important stages when there is usually a reporting and/or evaluation. There are two types of graphical representation for scheduling. The first one is the Timeline Chart or the Gantt Chart. The time axis is horizontal and each task is displayed as a rectangle on a separate row, with its starting and ending times clearly indicated. Milestones are marked as diamonds at specific time points. These charts sort the tasks in time but they do not show the

dependencies as good as the second type of charts which are in the form of task networks. Figure 2.2 displays a Timeline Chart example.

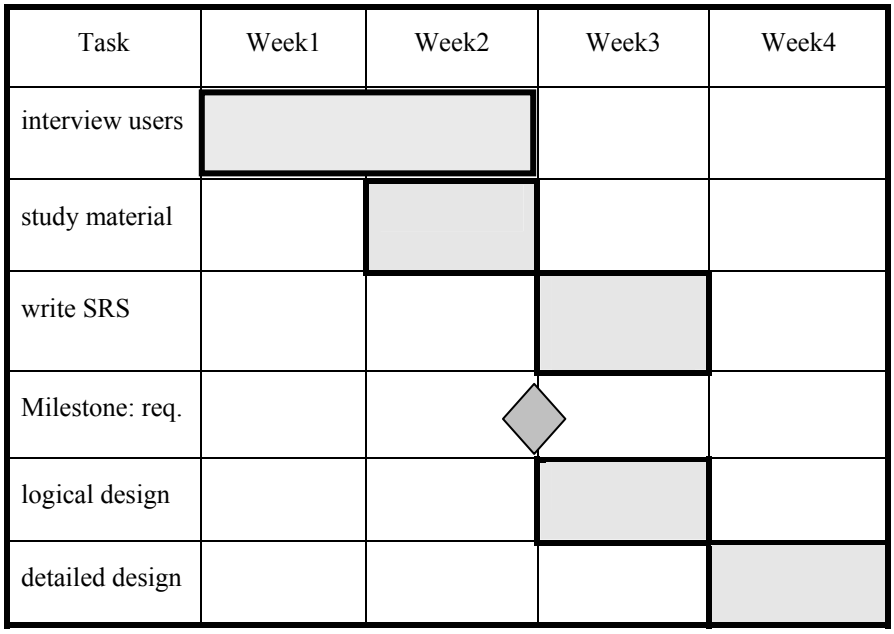


Figure 2.2. An example Timeline Chart

These charts can be drawn in different detail levels for the same project. A summary chart may include aggregated tasks and may have its time intervals defined in months, quarters, even in years. A detailed chart has to be drawn laying out all the defined tasks. Time intervals could be as small as weeks and even days. Tools help organize the project scheduling by associating the beginning and ending times with the tasks and other explanations.

Task networks help visualize the dependencies, and are instrumental in finding the Critical Path. The bottlenecks in the process should be identified so that the management knows where to interfere to ensure the meeting of the deadline. Figure 2.3 depicts an example task network.

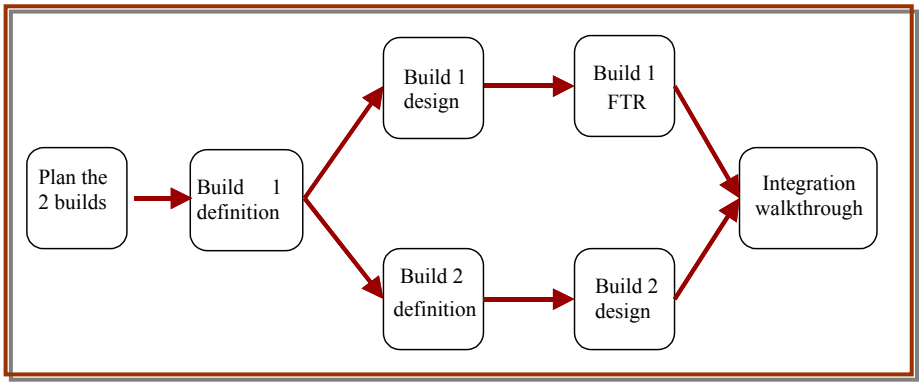


Figure 2.3. An example task network

The Process Modeling tools on the other hand, can display a more detailed view of a task network. Input/output products, resources, and roles assigned to the tasks are visible. Also the communication connections among different tasks can be modeled. Once built, a model can be run to produce simulations and more analytical data can be obtained for the evaluation of the bottlenecks. Further information is contained that could help in analyzing the needs of individual tasks and in justifying the effort assigned for them. Figure 2.4 presents an example model page, corresponding to the FunSoft [FunSoft 2001] tool.

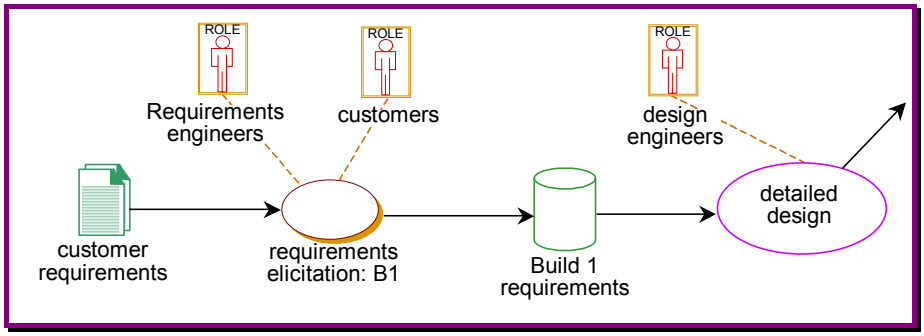


Figure 2.4. An example Process Model

Staffing

Early ideas about software personnel can be summarized by the term "programmer teams." This is an idea that is still valid to a lesser degree. It is generally known that strict hierarchy in the personnel management does not fit the software field. Programmers usually need a more relaxed atmosphere to conduct their creative work. On the other hand, such an expensive process requires methodological approaches and the product is not a result of a single

person. Programmers have a tendency to act without discipline wherever software engineering notions are not mature. An organization that optimizes between flexibility and process compliance is what is needed.

The programmer teams [Baker 1972] are small groups of two to seven people working on the same block of code. There is a chief programmer or the leader who provides the coordination among the members, but this is not like a rigid manager's task. There is a librarian who would provide required research and maintain the versions of whatever produced – mostly code. The version management task today is greatly facilitated by Configuration Management tools. There may be critical skills in the team; for example, an expert in the domain of the current project. It is a good idea to have an alternative person for any critical role in the team. Usually meetings provide the communication inside a team and across teams. Communication should be encouraged, especially through informal formats. Since it is another fact that technical personnel in the software field change jobs very frequently, keeping the rest of the group informed about everybody's accomplishment helps preparing personnel "backup."

About one quarter of a person's time is productive and the rest is spent for communications and other non-productive efforts such as training. Between any pair of team members there is a communication connection. Any communication connection takes about 5 to 20 percent of a person's total effort. Although it is expensive, the information sharing among the team members is worth the cost. Big teams start getting inefficient; after some size the effort gets spent on mostly communications. Figure 2.5 displays a small team with communication connections.

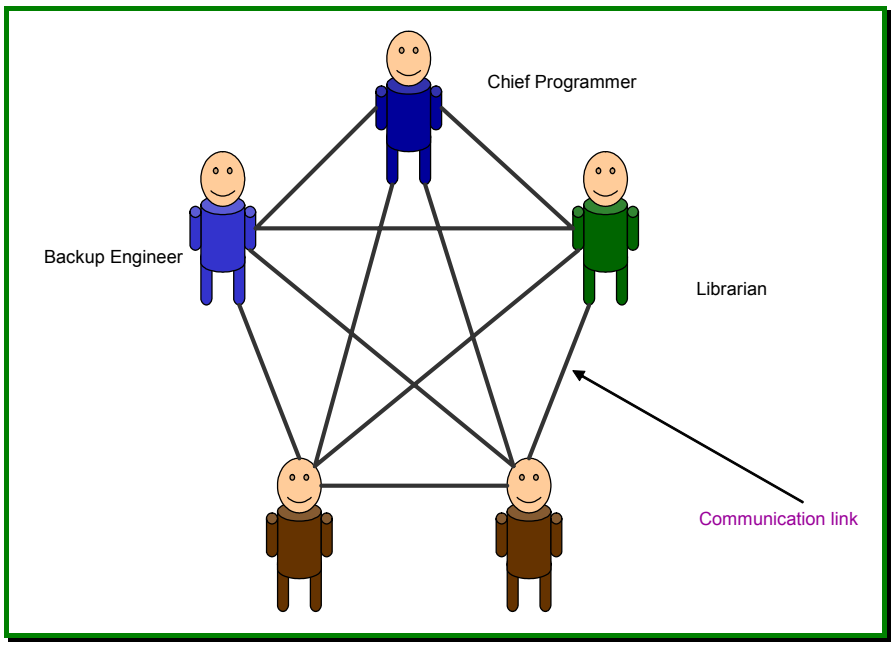


Figure 2.5. Communications in a programmer team

It is a common observation that projects fall behind schedule. The first corrective action that comes to mind is to hire more personnel in the middle of a project. This is usually not a remedy, due to the increased communication effort on top of the new training needs. The newcomers need to catch up with the project that has already advanced.

For a big project, a dynamic estimation tool can be used and personnel with different capabilities with respect to the phases, such as requirements or coding, can be employed. On the other hand, for small projects this may not be practical. Also eXtreme Programming (XP) techniques suggest that everybody should be familiar with the entire code and coding should be done in teams of two, sharing a single computer.

Risk Management

The difficulty in software development needs to be managed. One handle the managers have is identifying the areas that could be responsible for what could go wrong. Risk management is basically identifying and assessment of potential threats to project success, and preparing for corrective measures. The parameters that would hinder the development define constraints. If the probability is not 100 % for any factor's being a threat, then it is termed a risk rather than a constraint. Risks can be assessed for their chance of attack and for the estimation of the severity of their consequences.

Risks could be classified with respect to areas concerning business, technical, or procedural error fields. The classification helps in the consideration of a

variety of risks during the planning phase, trying to completely cover the possibilities. Business related risks could be the discontinuation of the financial resources, loss of interest in the management, market loss, among other things. There are also risks related to the operational and development environments accommodating new technologies. The lack of interest on the user side is another important one. Also, high personnel turnover rates define one of the peculiar risks of software development. Managing a risk includes mitigation; measures should be taken before the project (or the related stage) starts to avoid a risk. If this is not successful or an unprecedented threat happens, then it is time to repair the process. Perhaps the project plan has to be overhauled. Major plan changes are not very uncommon during a projects development stage.

A risk management plan can also be prepared and documented. Assessment helps in prioritizing the risks so that with limited resources an effective strategy could be drawn for project safety. A risk could affect a project in two ways: delaying the completion time, and increasing the cost. The two components of project hindrance need to be evaluated simultaneously to determine the severity of the threat. Sometimes this may lead to the early termination of the project. Management makes the decision to terminate a project and absorb any losses to prevent further losses in the future. Also, groups of risks could be evaluated for the probability to occur together and assess the composite consequence: none of them might be severe enough if attacking alone, but together the threat might prove to be fatal for the project.

Quality

Engineering always includes an optimization effort between cost and quality. The new competitive markets rule-out products of less quality. Also restrained with short cycle times the development staff is really challenged. One remedy is quality standards which themselves bring extra cost, effort, and time to the game. They provide some determinacy to the engineering output in terms of the quality of the product. The quality of the process that produces the product has surfaced lately as the main issue. Now the quality standards aim the process more than the product, which indirectly is affected anyway.

A closely related concept is measurement, to assess quality. Metrics and Estimation section introduced some basic concepts for measurement. Monitoring of a development effort cannot be achieved without measurements. The development should not be taken as merely meeting deadlines; quality factors need be traced as well. The difficulty is the definition and measuring quality related parameters. It is often difficult to directly measure quality. Some quality factors must be defined and their measurement techniques should be described. There has been a variety of classification and definition for quality factors.

In the beginning, there was less control over the process to affect quality. Rather, end products were tested and defects were detected. Quality control was applied to minimize the variation among the product instances. Later, the quality assuring measures were brought into the process to yield lesser defects. Total Quality Management (TQM) is partially a result of this idea yielding systematic approaches to eliminate the root causes of the defects. Software Quality Assurance (SQA) is the umbrella activity concerning the whole lifecycle. The activities involved in software development include quality management, formal technical reviews, documentation, compliance with standards, and measuring mechanisms. Testing, measurements and their reporting are crucial for the monitoring of quality.

A common measure is "number of errors." Tests are conducted to discover errors. Finding and fixing errors are expensive. The cost to repair an error increases as the development progresses. Figure 2.6 illustrates this increase in the repair cost.

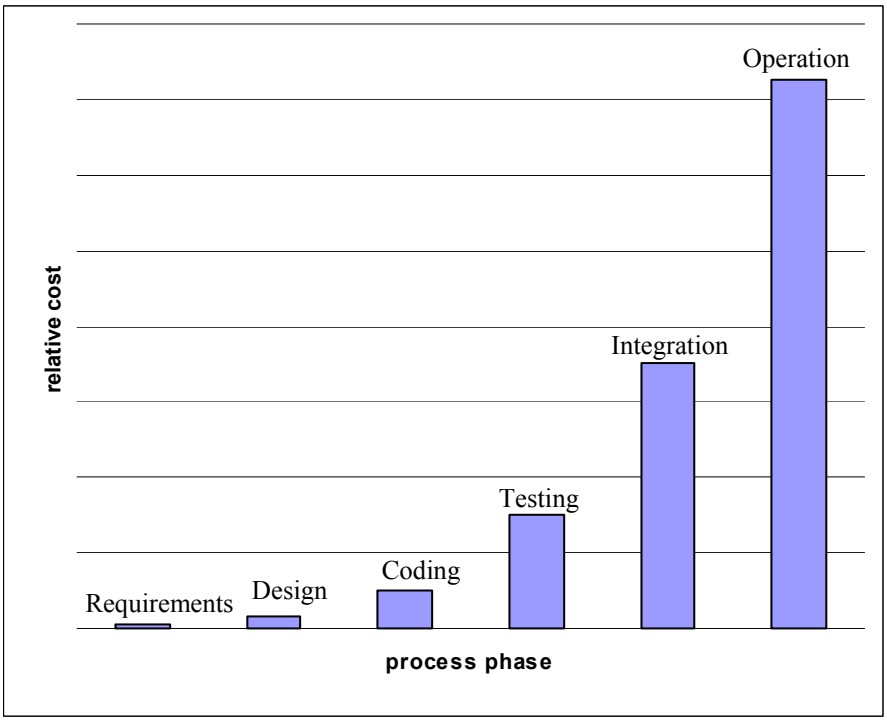


Figure 2.6. Increase in the error repair cost

Validation and Verification (V&V) are other measures for quality. Both tasks are usually carried out through tests. Validation is the questioning of requirements correctness. Verification is the questioning of the solutions meeting the (validated) requirements. Requirements are the main reference for judging the quality of software. Also compliance with standards and "implicit requirements" are important components of software quality.

Implicit requirements are not mentioned by the customer but conscious engineering practices assume them as default additions.

Quality Factors

Among a variety of classification attempts to quality factors, McCall [1977] can be selected as an early and valuable work. Due to the difficulty in directly measuring those factors, a set of metrics are also defined. A weighted addition of related metrics per quality factor is calculated to arrive at a quantitative interpretation. The factors are:

- Operational characteristics:
  - Correctness
  - Reliability
  - Usability
  - Integrity
  - Efficiency
- Changeability:
  - Maintainability
  - Flexibility
  - Testability
- Environment adaptability:
  - Portability
  - Reusability
  - Interoperability

The metrics used in the evaluation of quality factors are:

- Audibility
- Accuracy
- Communication commonality
- Completeness
- Conciseness
- Consistency
- Data commonality
- Error tolerance

- Execution efficiency
- Expandability
- Generality
- Hardware independence
- Instrumentation
- Modularity
- Operability
- Security
- Self documentation
- Simplicity
- Software system independence
- Traceablity
- Training

There have been more categorizations of quality factors and some are being used today.

Statistical Quality Assurance

The expensive nature of quality encourages a more quantitative control over the problem. The statistical approach helps in prioritizing the areas to allocate the quality effort. Errors are classified and they are traced to the underlying causes. The *Pareto principle* states that 80 percent of all defects are caused by 20 percent of all causes. These causes are the ones to attack first. A list of causes determined for statistical quality assurance is given in Table 2.7.

Table 2.7. Software Error Causes

Cause	Abbereviation
Incomplete or erroneous specification	IES
Misinterpretation of customer communication	MCC
Intentional deviation from specification	IDS
Violation of programming standards	VPS
Error in data representation	EDR
Inconsistent module interface	IMI
Error in design logic	EDL
Incomplete or erroneous testing	IET
Inaccurate or incomplete documentation	IID
Error in programming language translation of design	PLT
Ambiguous or inconsistent human-computer interface	HCI
Miscellaneous	MIS

An organization should report error counts and make a table organized with respect to the causes shown in Table 2.7. After deciding which cause relates to each error, a seriousness value for it is also indicated. The serious, moderate, and minor errors are weighted and added to a total count that is called the "phase index" to indicate an overall error based quality value for a development phase.

Software Acquisition

When a project is granted to a company (or a consortium) for development, there should not be an immediate expectancy for developing all the code especially from scratch. There may be readily available sub-solutions to the problem, and other companies may be more competent for developing some of the sub-problems or the complete problem. An analysis can be performed accounting for a variety of options for the acquisition, their costs and other benefits. The options are development, reuse, sub-contract, or a mixture for the complete project or for parts of it.

To attain a level of quality, sub-contract management is another task if there is such cooperation. It is also common now for projects to be outsourced overseas for sub-contracting. There is the risk of relaying requirements at a distance. Problems are already experienced with requirements even when the customer is near the developer. Still cost differences make it feasible for some projects to use distant sub-contracting.

It is worth mentioning that the general trend in any industry is to outsource the parts of the project that are not within the main expertise of the enterprise. The "core competency" of the organization, however should be preserved as the strategic asset and development related to this field should be carried out internally. Any other component of the product could be outsourced from companies that have better expertise/competency for the related development. There have been efforts to present the core-competencies as process models, ready to be integrated with others to provide the architecture of bigger enterprises in the virtual environments [Manzer, 2002].

Configuration Management

Often, a version of a source file is changed and later a request to revert to the earlier version arises. Probably the earlier versions are lost and to recover, some of the effort has to be repeated. If an effective version management is employed, nothing is lost; all the previous versions are kept and retrieved per need. This is valid for documentation and data units as well as code. Then there is the question of who determines when to accept the status as a valid version to be stored. Also there is the management problem for access rights on any unit stored in the configuration management tool. Change is constant in software development. A threshold where the accumulated changes correspond to some maturity in the item is accepted as a baseline and the item is stored with a version number.

If there is a need to change any aspect of an existing module, someone must initiate a request for change. Often, companies have change request forms that are subject to approval when filled. After the request is authorized, it is dispatched to the related personnel to conduct the change. Routine testing and documentation updates are enforced and the changed status can be evaluated for constituting a new version. Actually some of the versions will be called "releases" if the version is released to the customer. A release is corresponding to a system, whereas modules and other items can go through different changes and versions. Versions are numbered following a hierarchical syntax where decimal points are separating numeric sections that correspond to smaller and smaller changes.

Configuration management is aided by tools that are as important as the CASE tools. Powerful yet free tools are available. It is a good idea to get the team acquainted with a tool and start using it before the construction of any document or code.

Maintenance

Maintenance is the engineering effort dedicated after delivery. There could be any number of reasons to change a finished system. There may be

necessities or speculations about better use of the product. Maintenance can be classified as corrective, adaptive, perfective, or preventive. There are so many systems waiting to be ported to a different operating system or Web platform, thus requiring adaptive maintenance. If a system in operation reports frequent errors (defects) and maintenance is difficult and expensive, it is time to consider a re-engineering process also. For a product, the effort to repair one source line is many times more than the effort to create it initially. Before deciding to re-engineer a system, expected lifetime and expected maintenance costs for the target system must be estimated and compared to the estimated maintenance costs if it is not re-engineered. Ever since the industry started with ad-hoc approaches, there used to be plenty of software products that needed to return to a more documented and maintainable status thus creating a lot of re-engineering projects. Soon after, there were reverse engineering tools serving the developers at different levels.

The lowest-level reverse engineering tool is a disassembler. A disassembler takes executable code and produces an assembly language corresponding to it. Actually, converting any lower-level representation to a higher level is a reverse engineering task. What is more desirable is to create design models out of existing code. Finally, requirements will be reached. Once requirements are known, modern approaches can be followed to forward-engineer and re-create the system. Of course the newer version is an improved version that is a lot easier to maintain and change. Other quality parameters should be improved also. Re-engineering is reverse-engineering followed by forward- engineering.

It is not only software products that can be re-engineered. This procedure was also first discovered by the traditional "hard engineering" fields. Business processes can also be re-engineered. If applied correctly, the effort can be extremely useful for any service or manufacturing industry. Actually, it is a rule to engineer the business process before any software-based automation is requested. There have been very successful examples, especially among huge enterprises for business Process Reengineering (BPR) applications. Such a project starts with the analysis of the current process. The process analyst must be an expert to conduct such a difficult procedure. Often involved are the interviews with the personnel at different levels. There are so many human and organizational factors that affect the acquisition of correct process knowledge. Finally a model for the current process is created. Process modeling tools (or workflow management tools) are used for creating the current process, followed by the target process. A process model example created by such a tool is given in Figure 2.4. It is then time for a transition from the current to the desired process, and this transitioning process needs to be designed also. A key success factor is concentrating on the business goals rather than existing company structure or procedures.

Summary

Software Project Management and its component activities were introduced. Resources have to be allocated to activities that are scheduled in Project Planning so that deadlines can be met with minimum cost and maximum quality. Risk planning involves identification, assessment, propagation, prevention and management measures. Quality is achieved by planned actions to assure better software throughout the process. Staffing should consider the communication links between any two members of a team as a sink for a fixed amount of effort. Measurement is important to aid in the estimation and monitoring of the project. There are size and function-oriented metrics. Various process and product parameters can be measured based on such metrics. Software process maturity provides a degree of determinism to product development. Parts of a project can be outsourced through a sub-contract to an organization that is competent in developing that kind of software. It is best to develop only what an organization defines as its "core competency" and outsource others. Maintenance is conducted on finished code for any reasons to change the product. Configuration management is applied through specific tools that organize access to any past versions of code, data, and documentation units.

Questions

1. Assume that you are the manager of a 5-year project, and, by the beginning of the 4th year, according to the metrics collected, you can estimate that the deadline will be missed at least for 6 months. What kind of corrective measure can you take?
2. It is known that personnel turn-over rates are high. You are about to start the development of a new project. You are also aware that there are competing software companies that offer higher salaries than your organization. What measures can you suggest to manage this risk of losing personnel in the middle of the project?
3. For an example project you will define, estimate the complexity applying the Function Points method. Write your assumptions about the project such as number of inputs and other complexity parameters.
4. For the same project you have defined in question 3, estimate the effort and time using the Lines of Code method.
5. For the project you have defined for question 3, estimate the effort and time using COCOMO. State your assumptions about the problem type and your selection of the model (basic intermediate or advanced).
6. There can be some indirect metrics for evaluating different quality

factors such as number of errors per KLOC. Can you propose metrics for maintainability (ease of maintaining the system, through changes in the code?).

7. Your organization is responsible for the maintenance of some software systems produced also by your organization. Currently there are no new projects in development and you have to keep some number of personnel regardless of missing new projects. How would you decide which system to re-engineer?
8. A re-engineering project is in progress. Starting from code, the reverse engineering process has finally obtained a requirements model. What are the factors that will make the forward engineering easier than a similar projects first-time development?
9. In a traditional programmer team, how would you replace a librarian with a configuration management tool? What would be the duties of the remaining roles in the team for utilizing the tool?
10. After investigating several alternatives to acquire a software product, including sub-contracting, reuse and outsourcing kind of options, you have found the least costly alternative. Would this alternative be your decision, or are there other parameters other than cost that you would consider?

References

Albrecht 1979 — A.J. Albrecht, "Measuring Application Development Productivity," *IBM Application Development Symposium,* Monterey, California, October 1979.

Albrecht 1983 — A.J. Albrecht and J.E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," *IEEE Transactions on Software Engineering*, November 1983, pp. 639-648.

Baker 1972 — F.T. Baker, "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, Vol 11, No. 1, 1972.

Boehm 1981 — Barry Boehm, *Software Engineering Economics,* Prentice Hall, 1981.

FunSoft 2001 — Funsoft users manual, Funsoft, 2001, Austin, Texas.

Manzer 2002 — Ayesha Manzer, Formalization of Core-Competency Processes for Integration of Value-add Chains, PhD. Dissertation, Middle East Technical University, July 2002.

McCall 1977 J. McCall, P. Richards, G. Walters, Factors in Software Quality, NTIS AD-A049-014, 015, and 016, November 1977.

Paulk et al. 1994 M.C. Paulk, C.V. Weber, B. Curtis, M.B. Chrissis, *The Capability Maturity Model: Guidelines for Improving the Software Process,* Carnegie Mellon University Software Engineering Institute, Addison-Wesley, 1994, Reading, Massachussetts.

Pressman 1997 R.S. Pressman, *Software Engineering: A Practitioner's Approach*, 4th Edition, Mc-Graw Hill, 1997.

Chapter 3: Traditional Software Development

The introduction of the waterfall lifecycle enabled engineering methodology to be carried over to the software field. Approaches that were pioneering the field targeted structured programming languages as their coding media. Development processes were mostly phased. Graphical modeling formalisms were employed as practical aids to further comprehension. So much had been done in the traditional age, and its techniques are still being used. Some of the products of this era have been reengineered to their object- oriented versions.

System analysts conducted tasks similar to those of industrial engineers. Dataflow diagrams were very important for software engineering as well as for some other disciplines. Requirements and design models contained diagrams reflecting different cross sections, which were actually different facets of the system.

Requirements engineering has been very important as it is today. Feasibility study followed requirements elicitation, then analysis, modeling and specification. Some prototyping and validation investigation were also carried out even before starting the design. Dataflow diagrams were first used in the requirements, and then in design with different categories of details being entered in each phase. Design could be conducted in two different abstraction levels: first logical then detailed levels. Requirements dictionary and entity-relation diagrams were the other two widely used instruments for requirements specification.

Design activities started with revising the requirements model, refining the dataflow diagrams, then conducting a data design utilizing the entity-relation diagrams of the requirements study. After data, the next design "cross-section" was structure. The architecture of the system was defined in blocks within a control hierarchy. Finally internal details of the modules were specified as functional details.

A very important phase in development is the integration phase. Often carried together with testing, units were connected to the growing system after completing their unit tests, one at a time, and the system integration test should be conducted after every unit's integration. Integration is a difficult stage, where hard-to-find errors surface that have passed the unit tests before. Finally system tests were conducted.

Looking back

The traditional approaches had carried the industry from its infancy days to the object-oriented era. Although there were so many ad-hoc attempts

accounting for some of the failures in the industry, structured approaches were not that lucky also. The bottleneck in the process appeared to be the requirements problems. Also the general complaint about the waterfall habit that prevented the flexibility required in adjusting to changing requirements is characteristic. Sometimes the process has been too heavy and forcing excessive documentation to be produced.

So many huge projects were also completed with acceptable levels of success. Some current approaches are basically an adaptation of the traditional methodologies. It is not uncommon to meet process models that are somewhere between spiral and waterfall. In the earlier years of the object-oriented trend, the new modeling option was not trusted by everybody, especially for big projects. Now this worry is fading away. Object-orientation is also being challenged by newer alternatives. Evolutionary processes have replaced waterfall.

Requirements

Also known as the analysis phase, requirements engineering has always been critical to design. In this chapter, general characteristics of the traditional approaches are being presented. Some peculiar methodologies may be following different routes but we believe that a great majority of the approaches are being represented in this chapter. So, the most important model used in requirements that is also used in design is the dataflow diagram. Used with Entity-relation diagrams and a requirements dictionary, dataflow diagrams allow a function-oriented look into the systems where data view is also preserved.

A feasibility study precedes the requirements gathering. What is covered more in-depth in this chapter is the modeling of requirements analysis. The modeling can only be conducted after eliciting the requirements through activities such as interviews and meetings.

Dataflow diagrams

The dataflow modeling is only concerned with the kind of information flowing among the system modules. Other information answering when and how this flow occurs are deliberately hidden. One common mistake students do is trying to include the control information such as “if” conditions to guard the data-flows. Also the order of execution is another example of mistakenly included information in the wrong data-flow models. Because the diagrams are so simple, a developer with less training is attracted to flooding the diagrams with other kinds of information. Actually these diagrams only give one view; all by themselves they cannot explain all aspects of an executable process – especially the dynamic aspects.

The data-flow analysis starts with defining the interface between the system and the "external entities." The system is seen as a single unit and the external entities are identified that interact with the system. Then the nature of this interaction is defined in terms of data flows, i.e. what kind of data flows, in what direction, between one external entity and the system. Any connections among the external entities are not shown because they are not included in the system to be developed. The developer side is only responsible for the system so dataflow diagrams should not include anything that will not be developed. External interfaces however, need to be displayed for the context analysis of the system. That is why this initial diagram is called the "context diagram" also referred to as the level 0 diagram. As it can be guessed after introducing a level, such diagrams are actually a system of diagrams hierarchically organized in levels. Figure 3.1 displays a level 0 diagram.

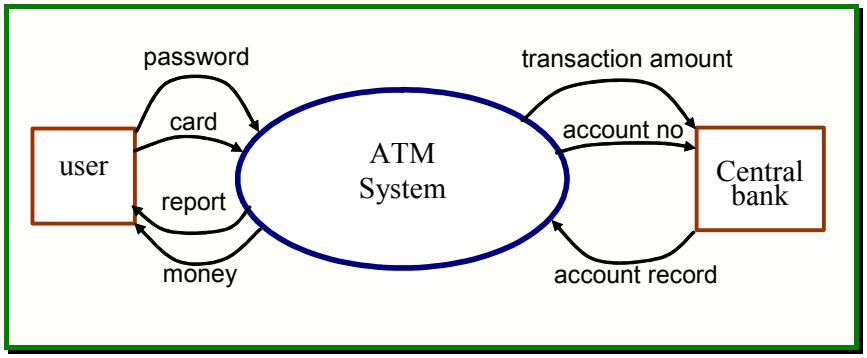


Figure 3.1. A context diagram example

For any oval that represents a "process," a lower-level Data Flow Diagram (DFD) can be drawn. The next level after the context is referred to as Level 1; the "overview" diagram. Like the context diagram, this one also corresponds to the whole system. Since the Level 0 diagrams contain one process, the lower level can only have one diagram corresponding to the system. Three to five levels are typical included in a project but there is no limitation to how deep in hierarchy levels a system of DFDs can go. The developers should decide how much detail to represent. In constructing a model, the goal is to understand the system therefore unnecessary details should be hidden while the desired type of details should be included. This principle guides the amount of refinement to take place.

Before continuing with the example, some properties of the DFDs can be introduced. Figure 3.2 depicts the four elements used in DFDs.

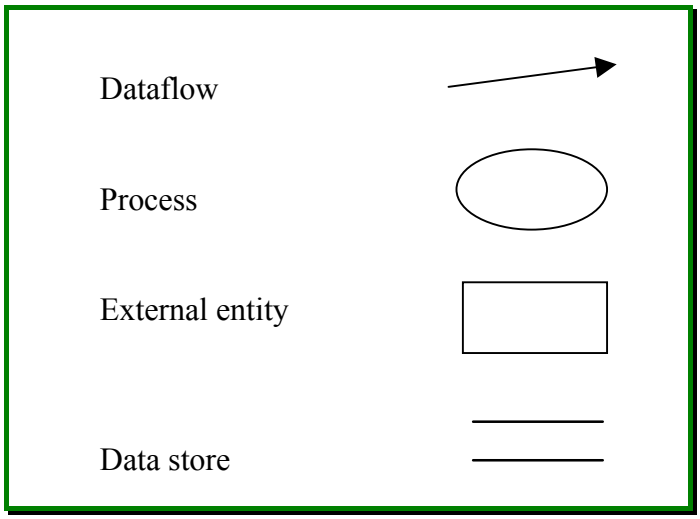


Figure 3.2. Dataflow Diagram elements

There is some variation in the representation of DFDs. Some sources show a data store as a rectangle with only three sides drawn instead of the two shown in Figure 3.2. Inclusion of the external entities is sufficient in Level 0. The lower levels do not need to redraw them. Some CASE tools accept the repetitive representation of the external entities in any level, especially in the overview diagrams. It may be a syntax error if an external entity is shown in the lower levels in some other environments. If they appear only in Level 0, this will help with consistency: any component should be shown only once.

A dataflow cannot be repeated more than once in a diagram. Where a process in a super diagram is detailed in a sub-diagram, the data flows coming in and going out of the process should be repeated at the boundaries of the lower-level diagram. This rule is an important rule referred to as "balancing the diagrams" and is valid even going from Level 0 to Level 1. To ensure diagram-balancing, before starting the lower level diagram, a box can be drawn to indicate the boundaries of the new diagram and the dataflow arrows can be copied form the higher-level diagram and drawn so that they cross the box boundaries. Their connections will be completed as the internal processes for the new diagram are defined and drawn inside the box. External connections for the arrows can be left disconnected; in the higher diagram those connections have been introduced anyway.

Other rules will be mentioned along with the developing example. In Figure 3.3, the overview diagram (Level 1) for the ATM system example is shown. Note that the connections on the process in Figure 3.1 are exactly duplicated with respect to their directions and naming. Also should be noted the absence of control logic; the particular operation the user selected cannot be traced and the conditions that trigger a withdraw process or a balance inquiry

process are unclear. If there is a chance that some data can flow between two nodes, it is drawn in the model.

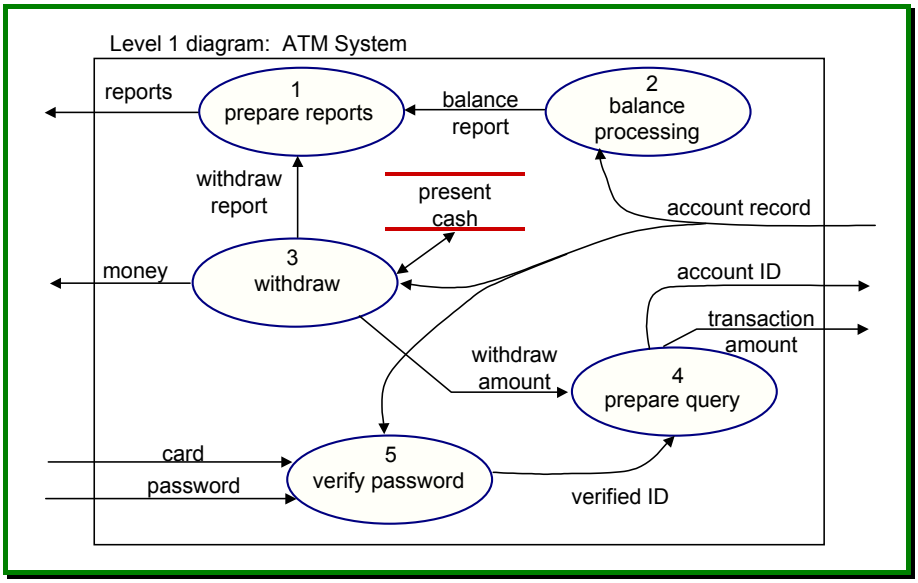


Figure 3.3. Overview diagram for the ATM system

In Figure 3.3, the rectangle drawn around the DFD is titled on the left top to indicate what process and what level this diagram corresponds to. It is also considered good practice to number the processes to reflect their levels. For example, the process "withdraw" numbered 3, will be detailed in a Level 2 diagram where the sub-processes will be numbered as 3.1, 3.2, etc.

Another rule is not to duplicate the same name on any two data-flows. This is like declaring a variable twice in a program. If a data item enters a process, it will not leave with the same name; the process must change the data somehow therefore the transformed data leaves the process with a new name. Also all data-flows should not be left unnamed, except in the case of connections on a data store, where the name on the data store implies the same name on any connection to it. Data-flows can split and reach different processes (or data stores or external entities) but two different flows cannot join unless they arrive at a process and the process produces a combined output.

Data stores can contain a single data item such as the current cash amount in an ATM, or a more complex data structure. Usually they correspond to database tables and records are retrieved in practice, rather than a field in a record. There are different usages of data stores as to allow individual field accesses to a data store that contains records of information. One other

hesitation is about the repetition of the data stores in different levels of DFD diagrams. The consistency rule mentioned before applies here also; the developer should first determine what process owns this data store and then should let that process contain the data store and should display it only inside that process.

The example will be extended to the next level for only one process in the overview diagram. Actually if desired, there could be five different "Level 2 diagrams" produced, corresponding to the processes 1 to 5 in Figure 3.3. A DFD will be shown in Figure 3.4 for Process 3 in Figure 3.3 modeling the "withdraw" operation.

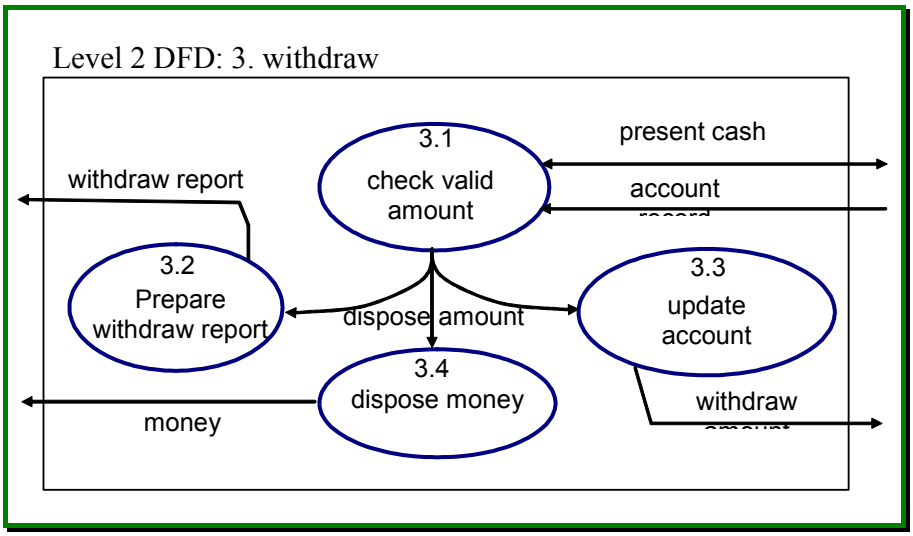


Figure 3.4. Level 2 diagram for the withdraw process

The example can be extended for other processes in Level 1 and to further levels to refine the processes in Level 2 diagrams. For demonstration purposes the diagrams included so far will suffice.

Other kinds of information require different models. A common extension to dataflow diagrams is the control-flow representation described in the next section. Rarely used, control-flow diagrams accompany DFDs if temporal management of events is important.

Control Flow Diagrams

Control is one of the essential modeling dimensions among the others; data and function. Not frequently used unless synchronization and related timing is important to model, the control-flow diagrams are only used with DFDs. A finite-state machine can be assumed per DFD, receiving input signals from sources such as buttons or data conditions produced by the DFD processes

and its outputs as signals and flags. Control-flow Diagrams (CFD) are superimposed on DFDs. There are two approaches, allowing for control bubbles to mix with data bubbles, or allowing control-flow interfaces between the DFD and the underlying state machine. To understand the control logic, the fundamental state machine concepts must be digested first.

Finite-State Machine

There are cases when a software system has to know the history of the actions to interpret the current input. In other words, with the same input data, the system may have to respond differently based on what happened before. For example, in a Windows application, when the mouse is released the system has to take an action based on where the mouse traveled before and where the button was pressed. To enable such an operation, the control software should keep track of the state: the state should change each time the cursor enters or exits control areas on the screen. If the state is known, the mouse release will trigger an operation based on that state – the active area the cursor has entered. Figure 3.5 displays a state machine that models the control logic of a vending machine disposing a drink if sufficient coins are inserted. This machine is in the reset state meaning that zero cents have been received so far. Subsequent insertions of coins carry the machine to "next states" corresponding to the total money received. The state of "35 cents" can be reached by receiving 25 cents and then 10 cents, or first 10 and then 25 cents. Alternatively, 7 repeated insertions of 5 cents can bring the state to 35 cents. The machine will dispense a drink if it can reach the state "35".

There is always a unique initial state (reset) but there can be more than one final state where the machine halts and usually produces a related output. Some machines would return to the reset state once a result is achieved thus eliminating the need for a specific final state. In this case, the final state is equal to the initial state. There have been two major modeling approaches with state machines. Either the state transition generates an output that lasts for a very short time, or a stable state is generating an output signal as long as the machine remains at that state: output is active during a transition or during a particular state.

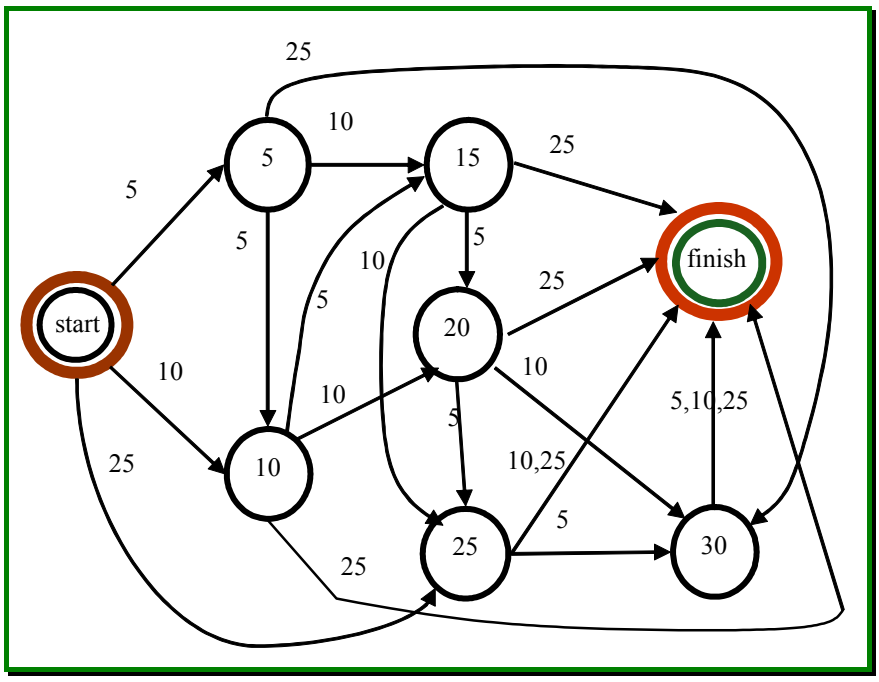


Figure 3.5. State machine for a soda vending machine

Ward and Mellor control flows

This is one approach to add the control information to the DFD models [Ward and Mellor 1985]. A continuous control processing is assumed together with the data processing. Bubbles representing data processing nodes are taken as a template for the control units. The control processing nodes are again bubbles drawn with dashed lines. Also the control item arrows are made of dashed lines. There are control stores drawn like data stores but with dashed lines. This treatment of control information requires some continuously updated data items to be represented also. Such items are primarily data items but they help in modeling the control logic. Therefore the "quasi-continuous" data flows are introduced by Ward and Mellor, as data flow lines with double arrow heads drawn in-line. Control-flows can trigger the start of data processes. Figure 3.6 displays the Ward and Mellor extensions to DFDs for modeling control-flow.

Hatley and Pirbhai control flows

This approach treats the data and control-flows a little more separate [Hatley and Pirbhai 1987]. The interface between the two flow models takes place on the DFD diagram, shown as thick bars in Figure 3.7. The dashed control-flows ending at the vertical bars mean control-flows sent to the underlying state machine. The dashed arrows coming out of the thick bars indicate control signals coming out of the state machine as output signals. The

control-flows can be signals generated by external events such as push buttons or sensor values, or generated by the state machine as signal outputs. Also data processes can produce control signals based on data conditions.

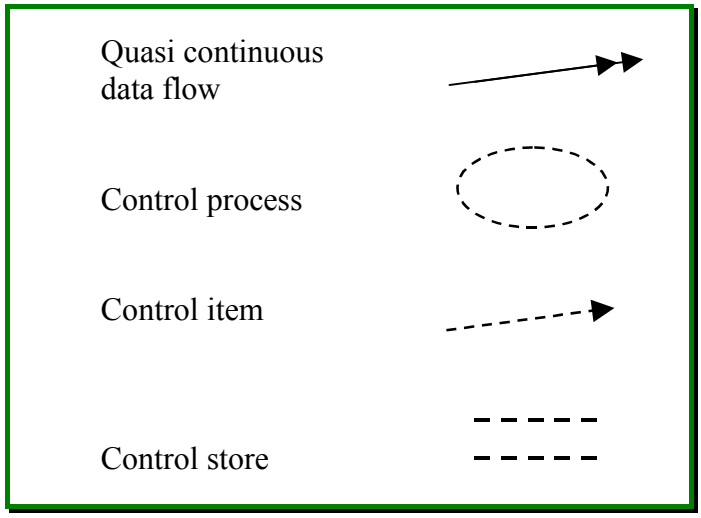


Figure 3.6. Ward and Mellor notations for control flow

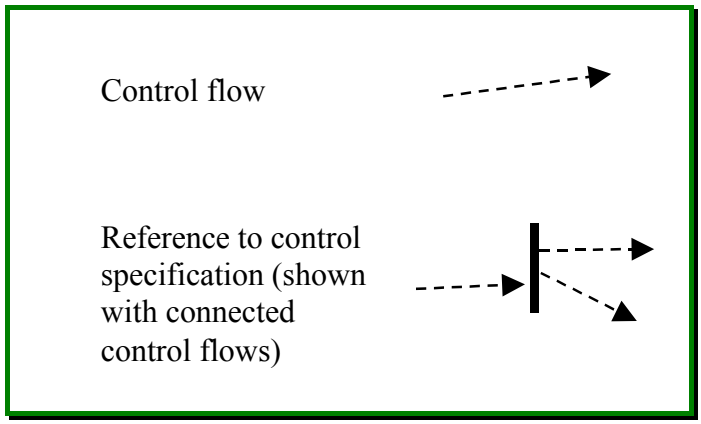


Figure 3.7. Hatley and Pirbhai notations for control flow

The overview diagram shown in Figure 3.4 can be enhanced with control-flows to demonstrate the Hatley and Pirbhai notations. The "verify password" process is considered for control-flows such as an input for "card inserted" event, an output from "verify password" process to the control mechanism for "invalid password" data condition, and an output from the control mechanism to be used as the "card reject" signal. Also a process-triggering control-flow (start verification) is included suggesting that the "card inserted" signal will initiate the process "password verification." The mentioned control-flow enhancement is displayed in Figure 3.8. Of course,

more flows could be generated but only the small demonstrative set is included. It should be remembered that the vertical bar represents only an interface to the control mechanism that could be provided as a finite-state machine, as well.

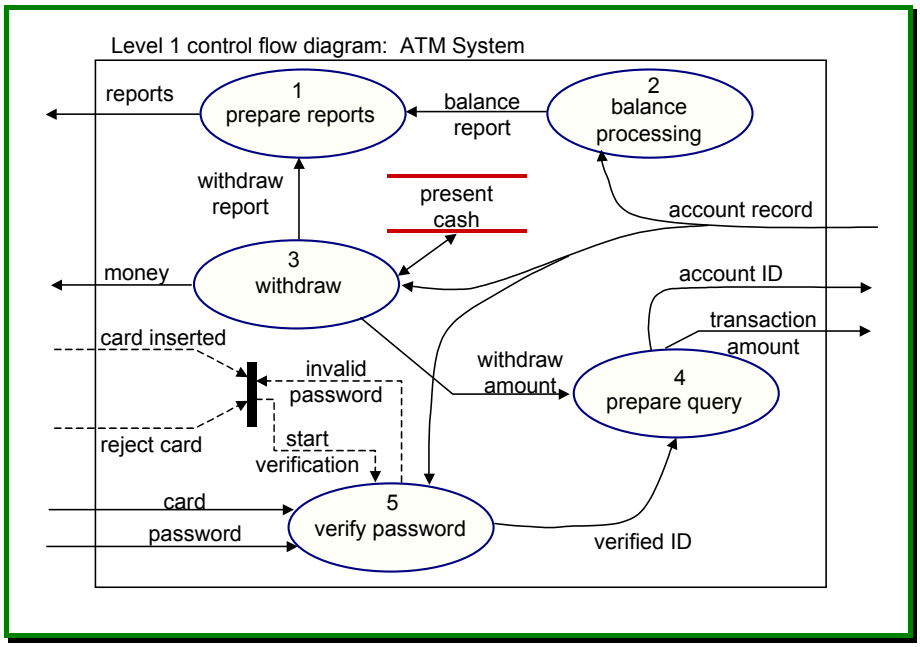


Figure 3.8. Hatley Pirbhai enhancements to the overview diagram in Figure 3.3.

Entity relationship diagrams

Some data items have been introduced in the DFD diagrams. They may correspond to important data elements in the requirements model. Actually some kind of data modeling facility is required. With or without the DFDs, the entities in the system requirements are modeled with their inter-relationship in Entity Relationship Diagrams (ERD) [Chen 1977]. Actually a starting point for the requirements modeling is the parsing of the requirements descriptions. The nouns are candidates for data items or external entities in DFD and the verbs correspond to processes. Some approaches may start with data modeling rather than DFDs.

Actually ERDs are a common way to "data modeling." The important data objects processed by the system are identified, their internal attributes are defined, and the relations among those data objects are described. The relation between two entities is further described in terms of cardinality and modality. Cardinality of a relation specifies how many of one entity are related to how many of the other entity kind. These quantities are represented as one or many. ERDs do not specify exact numbers; what is

important is the number being one - or more than one - in which case it is said to be "many." The cardinalities are one-to-one, one-to-many, and many-to-many. Later in design, these specifications will have very specific implications. Modality enlarges the cardinality with the inclusion of "zero or…" meaning to the one and many specifications. Optional modality means that an entity can be in a relation but it is also possible that the entity will not take part in the relation. Figure 3.9 displays a relation with cardinalities and modalities, between entities.

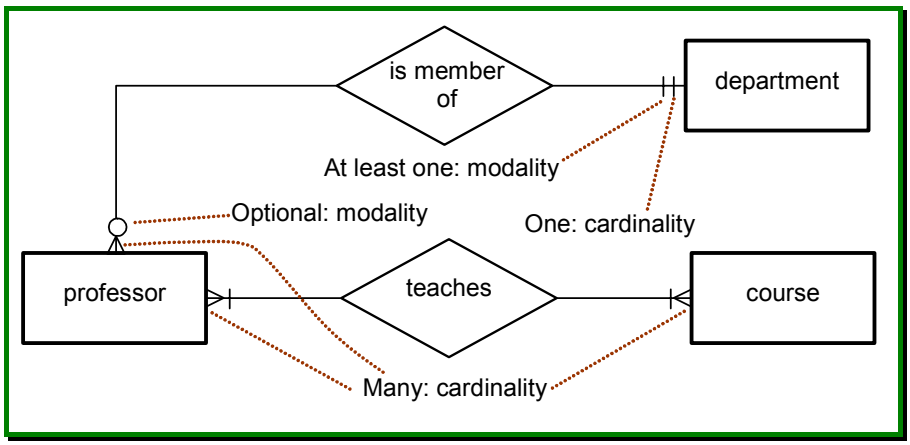


Figure 3.9. An example relation between entities

To figure out the cardinalities, one entity should be considered to determine the cardinality of the other. For example in the professor to department relationship, the cardinality on the department end can be found by considering "one" professor's relationship with the plurality of the department. Of course the semantics of the diagram can be debated. The idea is to allow the modalities and cardinalities as the system specification suggests. For example, the optional relation between the professor and the department could be the fact for the project, whereas in general one would assume that a professor should always be a member of one department and a department should have at least one professor.

Entities can have attributes. Each attribute can be drawn as an oval outside the entity rectangle. The ovals are connected to entities by lines. CASE tools of the traditional era provided facilities to draw DFDs and ERDs. Also some compilers that include a database facility provide a graphical modeling of the tables and their relations. Although data modeling is an abstract task, in most cases their outcome is translated into database systems due to the wide availability of the technology.

The early ERD definitions included attributes for the entities that can be drawn as ovals. Figure 3.10 depicts this representation for the "professor"

entity selected from Figure 3.9, as an example. The relations grew in the direction of Object Oriented (OO) concepts such as inheritance. The classification mechanisms in the class diagrams of OO models are later included in the ERDs.

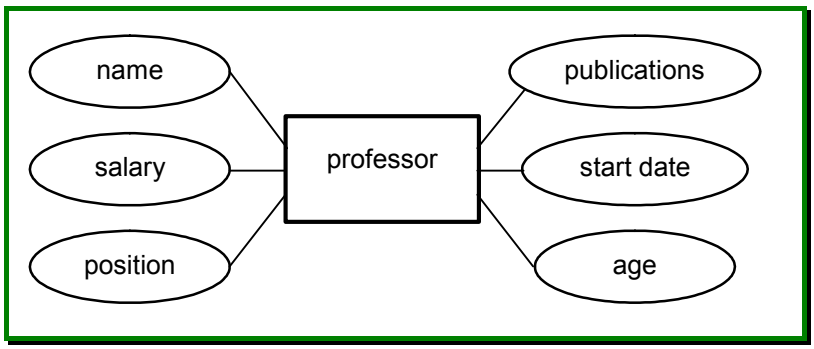


Figure 3.10. Attributes of an entity

Requirements dictionary

Also called the data dictionary [Yourdon 1989], this facility helps in the organization of the concepts used in the DFD or ERD models. Especially the data-oriented items such as data-flows and entities are explained in the dictionary. There is an entry per item with its name and any other names used throughout the requirements. An entry is shown with its attributes, and whether it is input or output to the defined processes. There is no common format for a requirements dictionary. This facility could be created using a word processor, but CASE tools include it and allow links back and forward from this dictionary to other graphical models. During the analysis task, clicking on a data-flow in a DFD a tool should be able to direct the developer to the dictionary where more explanations can be found. Similarly, clicking on a process should take the developer to another text-based medium; the procedural specification for the process.

Procedural specifications

So far the processes were only defined but not described. The ovals in a DFD correspond to processes and they need more detail for the requirements specification. There can be a variety of methods to describe how a process will execute. Care should be taken not to describe the solution but the problem definition. Step-by-step description of the actions can be provided in plain English. A more formal medium is “Structured English.” Also known as procedural specification language, this notation takes the control keywords from a programming language such as C and fills in the statements in plain English. The entities and data items introduced in the previous models can be used as variables in this specification. For functions requiring

mathematical functions or tables, such representations can also take place in procedural specifications.

The procedural specifications in structured English are presented below for the example shown in Figure 3.4, Process 3.1: Check Valid Amount. Keywords taken from the Pascal language are printed in boldface in this listing.

```
Get account record
If desired_amount > account.balance then
Begin
        Give message ("balance not enough");
        Exit;
End
Get present cash
If present cash < desired_amount then
Begin
        Give message ("not enough cash in the ATM");
        Exit;
End;
If dispense(desired_amount) AND
  Update balance(desired_amount) then
        Prepare withdraw report (desired_amount);
```

Design

Once the requirements definition is satisfactory, design activity can begin. This is valid for individual modules or for the whole system. The common pattern is to conduct data design first then structural, followed by interface design, and finally procedural design. Before modules can be dispatched to teams for development, common data structures should be defined; data design identifies the "global" entities for the access of different modules.

Data design includes the data structures and data tables for the database management systems. If abstract data structures such as stack or queue elements were defined in requirements, it is time to guide their constructions in terms of linked lists or tables, for instance. Now, rather than concentrating only on the pure definition of correct logic, efficiency should also considered here. For this reason previous definitions could be altered or new data structures can be added. The information contained in the ERDs guides the design of the tables and their links in terms of keys and indexes. Normalization of the tables is an issue here; some normalization is necessary but again due to efficiency reasons, sometimes normalization is deliberately violated. Normalization is a compacting operation performed on a relational data model where no information is repeated and one field may contain at least a single data item.

DFDs are revisited and refined in the design phase to include the implementation level details. The goal is to arrive at a structural specification of the system with modules corresponding to functions at various levels of abstraction. The recent view about a program: a hierarchy of functions controlling the execution flow through function calls is gradually obtained. After the data design, it is now the time for structure (or architectural) design. Dataflow diagrams [Pressman 1997] are the starting point for this effort. The transform mapping method introduced in the following sections suggests the construction of "structure charts" based on the information contained in DFDs.

Structural design

There are different ways to represent the architecture of software systems. Structure charts are a simple way of achieving this. The main view a structure chart conveys is the control hierarchy among the modules.

When mentioning modules, the strategically important issue of "modularity" [Dennis 1973] is invoked. Design is a fundamental activity - if not the most important – a good engineer must master. When design is mentioned, probably the heart of the problem is the modularity. This issue is also present in data design. Modularity needs to be analyzed based on vital parameters to produce good designs. Those parameters are:

- Hierarchy,
- Cohesion, and
- Coupling.

Modularization is the decomposition of a system into its components. This is also the crucial step towards the quality of the system. An obvious advantage of good modular design is the ease in locating errors.

How should the modules be defined and the boundaries demarcated? An old story about wrong conceptions for modularization can be mentioned here: When the team leader asked the programmers to decompose a long piece of code into modules, a programmer unfamiliar with the concept asked about the typical length for a module. The answer was 30 to 40 lines. The programmer counted 30 lines and measured their height on the printed source code listings. He then drew a line separating the code into chunks, for every so many inches. Then, he organized those chunks as subroutines so that the main program could call those "modules", hence presenting a modular structure!

The principle of hierarchy suggests the correct spacing of a module with respect to *abstraction* levels, as opposed to simple distance. When the system is decomposed, the top-level modules should be defined first, containing high-level decisions. It should also be noted that the control

propagates from higher-level modules to lower-level modules. Vertical connections should be organized to reflect the correct hierarchy. The early work about design by herb Simon [1967] explains the importance of hierarchical decomposition.

Cohesion is the degree to which a module does only one task. A good module should not assume different functionalities and should not contain items that are used for different purposes. Coupling is the degree of dependence to other modules. Neither aspects can be directly and objectively measured but there are fuzzy metrics for their evaluation. It is desired to have high cohesion values and low coupling values for any module. There is some trade-off involved for the designer due to the racing conditions. If there is zero coupling, then there cannot be a system – modules are not connected. On the other hand, if there is perfect cohesion for every module, then to conduct the systems goal collectively, some coupling is needed. The other important duty of a designer is the optimization. Two modules can be combined to reduce coupling but cohesion is sacrificed. It is a common understanding to organize cohesive modules and compromise a little on coupling. Figure 3.11 relates the three design parameters to abstraction-level and horizontal-connectivity.

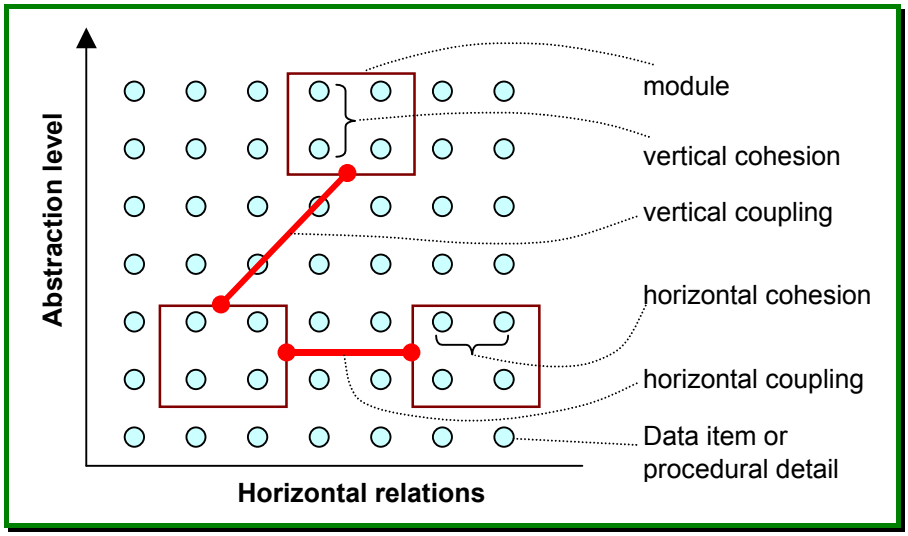


Figure 3.11. Design parameters and relations in two dimensions

For example, a module that calculates both sine and cosine is not cohesive. Also, to call this module, if a flag is being set to select between the two functions, this constitutes a "control coupling" which is not a desirable case. The caller would have to send for example a "1" to initiate a sine function or a "0" to initiate a cosine function. A better design suggests separate sine and cosine modules, where both are cohesive, and to refer them only as the minimum information that is the inevitable input value needing to be sent. Consequently, coupling is also reduced. The above example was about

horizontal cohesion. In the decomposition demonstrated in this chapter through structure charts, the couplings are mostly vertical.

In another example a lack of cohesion in the vertical dimension is presented: Assuming old fashion text-based user interfaces, we used to present a list of items and prompted the user to type a number corresponding to the desired item. . The following code segment combines the top-level menu logic and formatting of the input character:

```
                input := “0”;
        While (input > “0”) do begin
                                writeln (“enter 1 to add a new person”);
                                writeln (“enter 2 to delete a person”);
                                writeln (“enter 0 to exit”);
                                write   (“     ? :  “);
                                readln (input);
                                {*** character manipulation begins ***}
                If input <> “0” then begin
                        if input in Control_Character_set then
                                execute-control (input);
                        else if  (not ( (ord (input) – ord (0) ) in [1, 2] ) ) then
                                give-error-message(input);
                end {if};
                {******** end of character manipulation ********}
                Case input
                        “1” :           add-person;
                        “2” :           delete-person;
                End {case};
        End {while};
```

What this segment tries to accomplish is to present a list of action items, obtain the user’s selection, check for errors in the user input, and finally dispatch the desired function. Except for the character manipulation segment that is actually simplified in this example, the module is cohesive. It is intended to serve the top-level menu functionality. In order to conduct this duty, the codes attempt to carry out very low-level operations for error checking. This character manipulation for error checking reduces the readability and introduces low-level processing to the highest-level module of a system. Although error checking is an integral part of the menu

processing, we do not accept it as a cohesive part of this particular function. Lack of vertical cohesion is evident. Rather, a single statement could call another function that would do the error checking. The name of the function though, will fit in the module, as error checking (not its details) can be accepted as part of the menu processing function. So, a revised version of the above code is given below that improves vertical cohesion:

```
input := "0";
        While (input > "0") do begin
                                        writeln ("enter 1 to add a new person");
                                        writeln ("enter 2 to delete a person");
                                        writeln ("enter 0 to exit");
                                        write   ("     ? :  ");
                                        readln (input);
                if check-error (input) then
                   Case input
                       "1" :            add-person;
                       "2" :            delete-person;
                   End {case};
        End {while};
```

After the completion of design, the natural task to follow is coding. The traceability of requirements or design concerns depends on the programmers complying with the design and coding conventions accepted by the organization.

Transform Mapping

With this technique, a DFD can be mapped to a structure chart [Pressman 1997]. When done, the processes in a DFD are converted to modules in a structure chart. There may be extra modules generated in the structure chart, or more than one process could be represented in a module. The first step is to define the flow boundaries in a DFD. There are two kinds of DFDs so before drawing the flow boundaries, the type must be recognized as a "transform flow" or a "transaction flow." If the DFD is of transform flow, the boundaries are drawn for the following processes:

1. input flow
2. transform center
3. output flow

This means, a set of processes in the DFD are considered as part of the input flow and other two sets are considered for the transform center and the output flow. A structure chart for the system is started with the top-most module that controls the three DFD regions with one control module representing each region. Under those control modules, all the DFD processes are drawn as rectangles, in some order.

To demonstrate the technique, the overview diagram in Figure 3.4 is redrawn in Figure 3.12 with the flow regions separated. Of course, as there can be different modularization decisions for the same problem there could be different boundary determinations. For example, Process 2 "balance processing" could be decided to be in the input flow by a different designer. Likewise, "withdraw" (3) could belong to the output flow region. Process names and insight will help in such decisions.

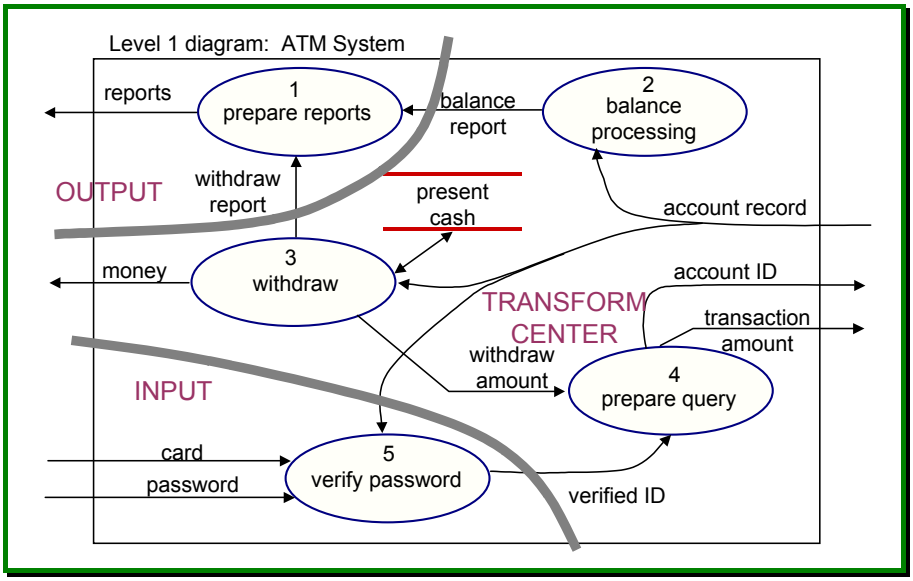


Figure 3.12. Flow boundaries in the overview diagram

A first-cut structure chart is immediately constructed. Since this example is determined to be of a "transform kind" flow, the main module and the control modules for the three sections are created. Figure 3.13 depicts the top-level control modules and other modules mapped from the processes in Figure 3.12. The three modules under the "operations" are treated as peers. However, a designer considering for example, "withdraw" as a control module for the "balance processing," should place "balance processing" under "withdraw" rather than next to it.

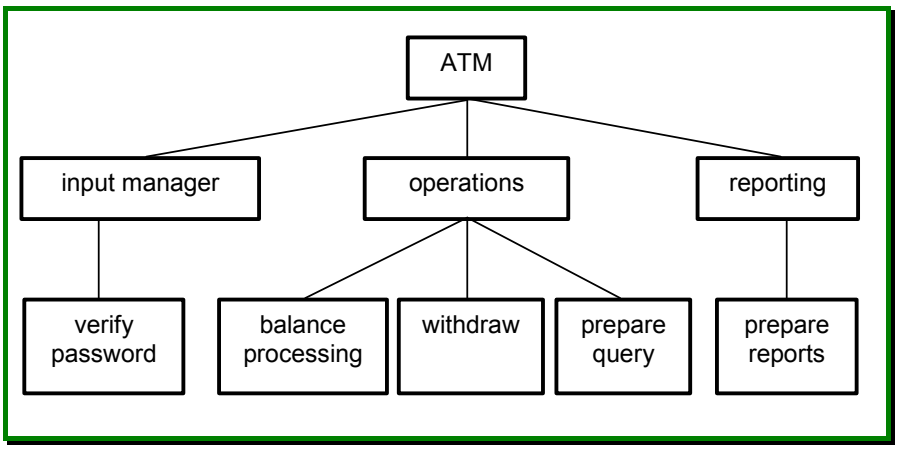


Figure 3.13. Structure chart corresponding to the overview diagram in Figure 3.12

Continuing with the mapping process for the example, the information contained in Figure 3.4 is considered for replacing or enhancing the "withdraw" module in Figure 3.13. Typically, the DFD in Figure 3.4 is analyzed for flow boundaries and the three control modules can be inserted under "withdraw." At this level, the control modules can also be omitted and the few processes can directly be connected under "withdraw." Unfortunately, the simple example in Figure 3.4 makes it difficult to determine the flow boundaries. Instead, the processes are directly mapped to modules controlled by "withdraw" in Figure 3.14.

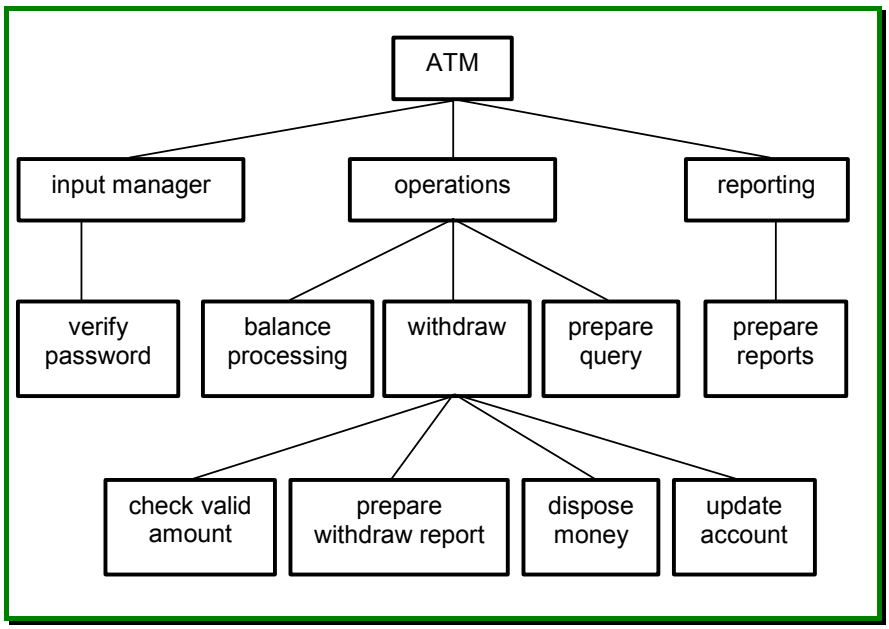


Figure 3.14. Enhanced structure chart for the ATM example

The transaction flow kind of DFDs corresponds to systems where there is a decision process that selects one among the possible action paths. For those, there is again an input flow region, but the transaction center now controls the individual action paths rather than an output flow region. So there is an input region, a transaction module and action paths take place under the transaction module. Any region in transform or transaction flow DFDs, are subject to a recursive evaluation: they can in turn be other transform or transaction flows. While the structure chart is growing downwards starting with the system as the root of the chart, the flow type decision will be repeated and one of the decomposition templates will be repeated under many modules.

To demonstrate the transaction flow based mapping, the ATM problem and the overview diagram for it will be slightly changed. Figure 3.15 displays the alternative DFD to Figure 3.3 and the flow boundaries drawn with respect to transaction flow regions.

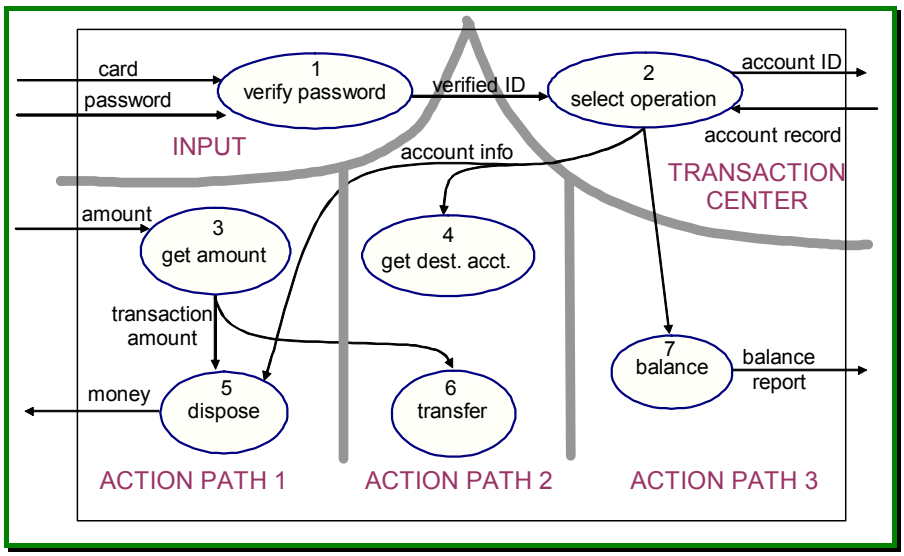


Figure 3.15. Overview DFD for an ATM example with transaction flow characteristics

The input flow and the transaction center regions are mapped to corresponding control modules arranged for transaction flow. Figure 3.16 depicts these control modules and the action paths controlled by them. It should be noted that a lower-level module can be called by more than one module that take place in any kind of flow organizations or at any levels. The "get amount" module is an example. It is controlled from two higher-level modules that belong to different action paths (withdraw and transfer).

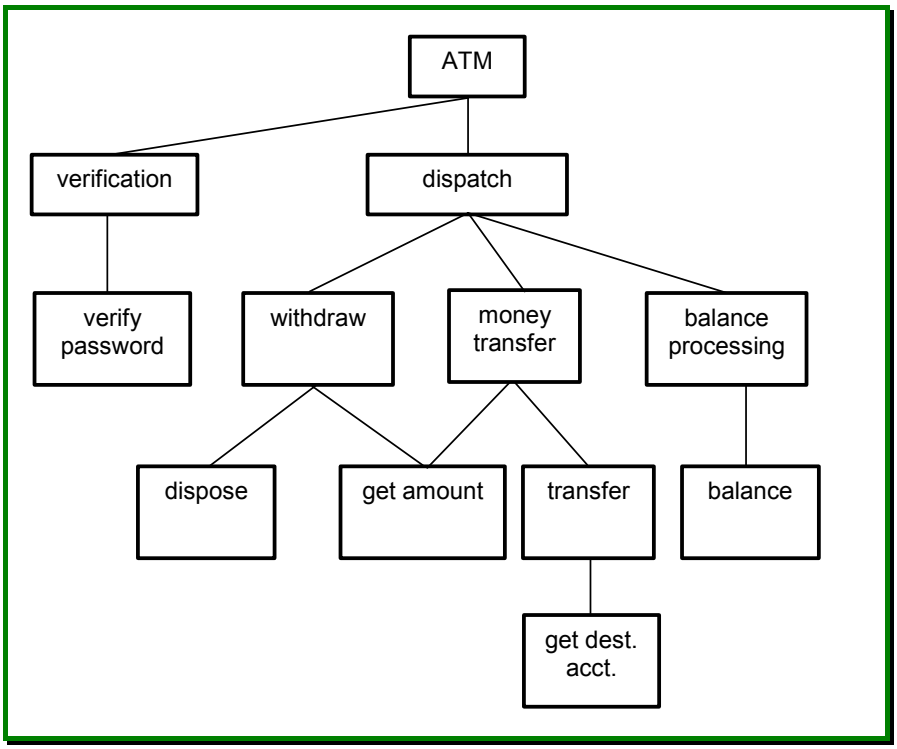


Figure 3.16. Structure chart for a transaction flow type ATM example

Vertical connections represent the control relation. Higher modules control the lower ones. Under the module that controls an input region, for example, the corresponding processes in the DFD should take place in an order that would suggest a horizontal positioning if there is no control relation, or top-down if there is. For the case where there is a control relation, the processes that are closer to the center (transform or transaction center regions) will be placed closer under the control module. This is usually in reverse direction with the dataflow arrows for the input flows. For output flows or action path regions, usually this control direction (top-to-down) is the same as the dataflow directions in the DFD.

The main concern in a structure chart is in determining what module controls what other. Although the structure concept hints at the composition relations (what module owns what other), a designer should think more in terms of who calls who. The higher-level modules call the lower-level ones, and optionally parameters for either direction can be placed next to the connections in a structure chart. The parameters are presented as little arrows with the data or control-flow name on them.

A final consideration on the transform mapping can be mentioned about the combined representation of the structure charts in contrast to the separate dataflow diagrams. The above-defined process will actually work if all the

DFDs were joined in a huge diagram. Such a diagram may be too difficult to construct. Since the recursive application of the "flow kind and boundary determination" define a hierarchical order, the system of DFDs can be used without having to combine them. Top-level flow boundary determination can correspond to the overview diagram. Small structure-charts will be produced for the lower-level DFDs. These small structure-carts will replace the individual modules in the initial structure chart corresponding to the overview diagram – thus enacting a refinement step. Some region in the structure chart can be constructed based on a joined set of DFDs, rather than considering all the DFDs in separation or in integration.

Coding and Debugging

A disciplined structure to coding is very important for the lifecycle of the software, no matter what kind of platform was used for the development of the code. There will be need to review the code, for modifications or additions. A maintenance programmer studies the documentation and the existing code before any modification. Any written code should be easy to read and understand. A programmer should organize his code quality in a way where others could easily maintain it.

There is no universal standard style for code writing therefore the team should agree upon the rules and their own protocol before starting the development. The components for effective coding can be listed as:

- comment lines
- code writing structure
- meaningful naming
- structured programming elements

Comment lines and code formatting

Comments are vital for the understandability of codes. There should be module headers, descriptions for the code lines, and spaces around cohesive segments. Special characters usually surround module headers as a box containing descriptive lines. Typically, the header includes the name, functionality, input/output, and dates for the modification history for the module. Control structures such as loop and if statements should be explained through comments. Indentation is used to delineate the blocks of statements that should stay together.

Structured programming

For the readability and maintainability of the code, "structured programming" structures must be used. These structures omit the "go to"

statements and they have a single entry and a single exit points. The three fundamental structures are as follows:

- sequential
- conditional
- repetitive

The conditional structures are implemented by the "if" and "case" statements. If statements are used to test only a single condition. They can handle two conditions when accompanied with the "else" clause. Multiple conditions are possible through a nested if-then-else or a case statement.

Repetitive blocks are loops, of which there are three kinds:

- Loops that iterate for a certain number of times (for loops)
- loops with top-tests that could repeat the iterations for zero or more times, depending on the condition (while loops)
- loops with a bottom test that repeat for at least once (repeat-until or do-while loops)

The development will be more efficient if only such blocks are used without "go to" statements. Appropriate commenting should accompany such structures also.

Debugging

After the testing phase finds errors, debugging takes place to find the cause of the errors. Debugging is very difficult and often frustrating. It is usually the programmer of the code segment who debugs it. Spending time and being intimate with the segment, the programmer often overlooks some causes that may be more obvious for an outsider. It is not uncommon for a different person to quickly determine the cause that took an inordinate amount of effort for the initial developer.

Debugging facilities are an important and recognized part of development environments. Debuggers allow for a controlled execution of a code segment where the statements can run one at a time and allow for the examination of selected variables as they change during the execution. Breakpoints can be installed so that a group of statements can run and the execution stops for examination. Some experienced programmers apply conventional techniques such as implanting output statements to monitor the values on selected variables, for debugging purposes. Some believe that if advanced debugging techniques are used, something is already very wrong, i.e. an organized code should require minimal debugging. However, at some point, there will be a time when all the educated guesses for diagnosis get exhausted and a

debugger must be used in an attempt to systematically find the source of the problem through "brute force" as a last resort.

A couple of popular approaches to debugging are backtracking and cause-elimination. In backtracking, the problematic locality in the code is taken and the code is manually traced back until the cause is found. This technique could be useful for small programs, but can become unmanageable for big programs. In the case of cause-elimination, possible causes are stated and the program is tested with different data to prove or disprove the cause candidate. These approaches can also utilize a debugger tool.

Once the cause of an error is found, the bug should be eliminated. Care must be taken not to introduce new bugs while repairing the code. Also, proactive thinking will help in locating other causes possibly produced by the erroneous logic that is responsible for the bug just fixed.

Testing and integration

Testing is conducted for finding errors. For reasonably complex software units, it is practically impossible to prove that no more errors exist. The more tests are conducted, the more errors will be found. The rate for finding new errors will decrease as the process continues. While errors are found, a repair operation must go hand-in-hand. It gets more difficult to test to find more errors when the discovery rate decreases. Testing is a costly procedure. Figure 3.17 displays the idealized and practical error discovery rates.

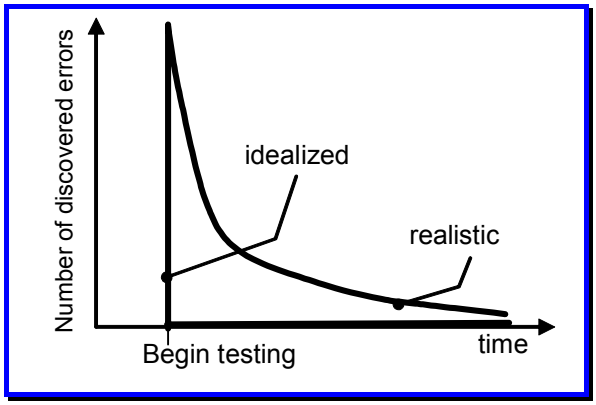


Figure 3.17. Error discovery rates

Testing is the vital tool for quality assurance, validation, and verification procedures. There are different approaches for different levels of activities. Testing also participates in the planning effort: before development, a test plan is prepared. Often the developers are requested to design the test cases along with the design, and sometimes with the coding tasks. Extra code is written to conduct the tests and data modules are generated. Even we might

want to consider the whole lifecycle interwoven with testing; the “V Process” model displays such a view as shown in Figure 3.18.

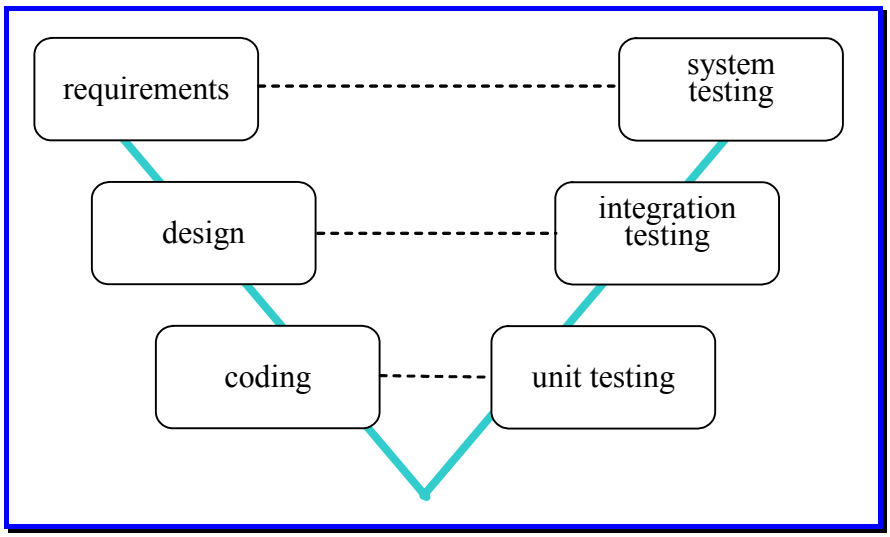


Figure 3.18. The V Model

There are testing personnel in a software organization. The team may have a dynamic structure; composing per project, out of different technical personnel. Some organizations have test engineers and a constant testing group. It is usually preferred to have a unit tested by people other than the developers of that unit. This is due to human psychology; the producer of the code will test it to show the superior quality in terms of less errors whereas a tester will conduct the testing to find more errors. There are also tools for testing and test case generation. Some modern CASE tools have the testing ability as a built-in feature.

Testing approaches

Depending on the nature of the problem, the amount of effort dedicated, and what kind of errors we are after, different testing techniques can be employed. In general, based on the evaluation of modules and how they are treated in the testing strategy the white box or black box techniques can be mentioned. The black box technique does not consider the internals of a module. The external definition in terms of inputs and related outputs are the information utilized by the tests. This is very much like the component-based approaches where only the interfaces of the components are known and internals are hidden. White box technique considers the internal details of a module and tries to test every unit inside.

In black box testing, the system is an interconnection of modules and the information about the modules consists of the functional definition that binds

the output to the input of that module. Such tests rather find integration errors, and also module errors but without being able to pinpoint the error's location inside a module. The cases when black box testing is preferred are if and when:

- the module internals are not known such as in the acceptance testing of a component
- the functional definition of a module in terms of its inputs and outputs is available explicitly
- a hierarchy of test stages is planned where first the faulty modules are to be identified
- the test is more integration oriented rather than unit oriented
- prohibitive costs preventing the tests to proceed towards lower-level details

White box testing ideally would try any possible execution sequence for a module. This means every branch in the code is taken and every combination of input values is tried. This desire is not realistic because of the prohibitive number of test cases required even for trivial pieces of short code. Therefore white box testing techniques usually compromise by proposing a reasonable amount of testing while trying to cover a meaningful representation of the complete picture. One such strategy is to test every statement in the code segment, at least once.

Basis path testing

Given a code segment, one should determine the possible execution sequences so that test cases can be prepared to drive the program to visit every statement at least once. Graph based techniques are utilized in finding the set of execution paths that cover the whole program. This brings another question: how many independent paths exist in a programs flowgraph? If the answer is known, and the indicated number of paths is determined then "basis path" testing can be applied. A test case will be generated to force the program to traverse the set of independent paths.

To answer this question and also find a set of independent paths, the program segment is converted to a flowgraph. This is achieved by representing statements as nodes and execution order by edges. It is easier to convert the flowchart of a program to a flowgraph. Table 3.1 introduces an example program for finding the basis path options.

Table 3.1. Code Segment for program menu processing

```
Selection := 0;
While (selection < 5)
   DO BEGIN
      Read (Selection);
      If not (selection in numbers) then exit
      Case selection
               1:  Add;
               2: delete;
               3: modify;
               {else do nothing – by default}
      end {case}
   END; {while}
```

Figure 3.19 presents a flowchart drawn for the code in Table 3.1. The earlier flowchart notation did not include a symbol for "case" statements. The diamond shape that is defined for the "if" statement is used, with multiple (4 in this case) out flows, to represent the case statement. Actually this complex statement can be implemented through a series of "if" statements. The result (number of paths to test) would not change.

The set of independent paths also define the complexity of a program. There are different complexity measures and this one is called the "cyclomatic complexity [McCabe 1976]." Cyclomatic complexity is equal to the number of independent paths. It can also be found by adding one to the number of decision points. Another way of calculating this number is by counting the nodes (N) and edges (E) in the flowgraph (the flowgraph corresponding to this problem is presented in Figure 3.20) and using the formula:

$$\text{Cyclomatic complexity} = E - N + 2 \qquad 3.1$$

This is equal to the independent loops in the graph. To arrive at the cyclomatic complexity, the internal loops can be counted and added to one. This final one can be thought of as the count for the external loop. It can be seen that a case statement produces loops for each case selector. Therefore for the purpose of cyclomatic complexity or the basis path determination, a case statement is equivalent to a set of nested if statements.

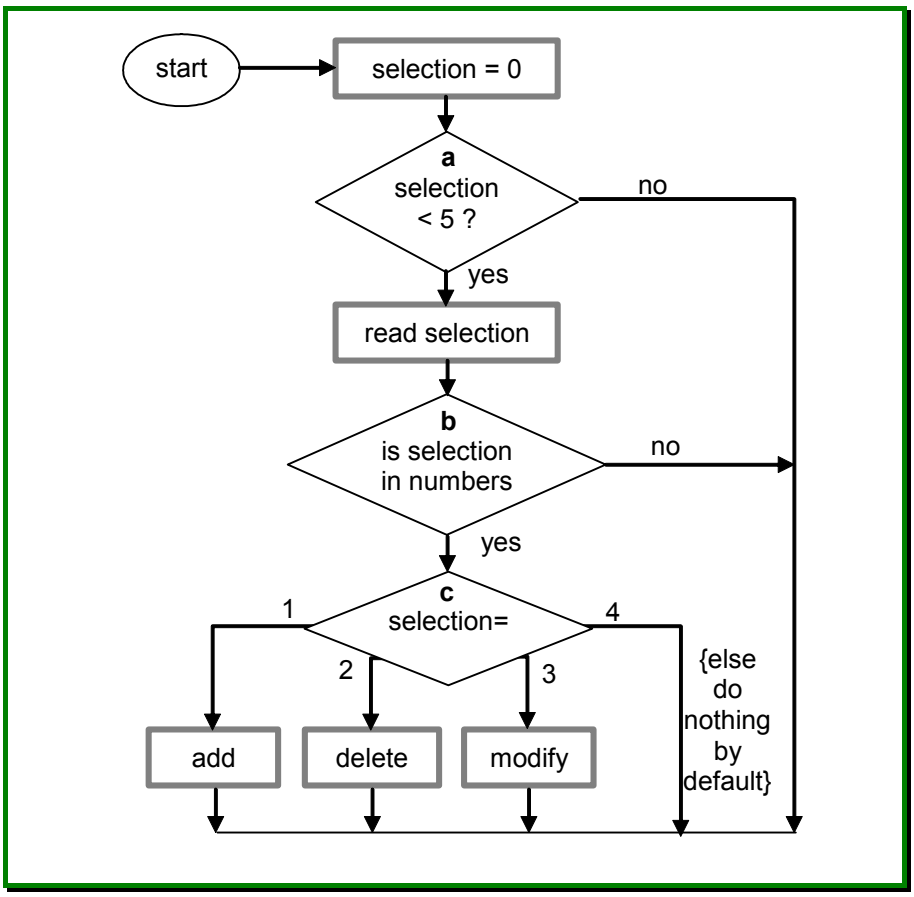


Figure 3.19. Flowchart for the program in Table 3.1

The decision points are important. Represented as diamonds, they have been marked with letters (a, b, and c). These elements play the key role in drawing the flowgraph for this program as shown in Figure 3.20. The rules to convert a flowchart to a flowgraph are:

1. represent every box or diamond with a node
2. connect the nodes with edges corresponding to the arrows in the flowchart
3. simplify the graph by deleting the nodes (and their connections) corresponding to boxes in the flowgraph. In the basis path analysis, nodes other than decision points are not important (every deleted node also cancels an edge so the final evaluation value of Equation 3.1 does not change)

4. for each decision point that forks the operation flow, introduce a "join" node (otherwise, the arrows joining with each other in the flowchart cannot join in a flowgraph)

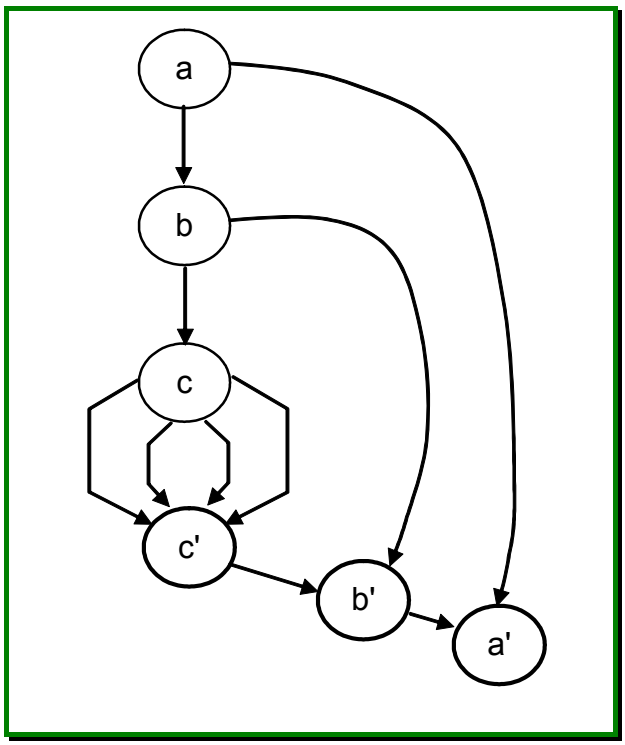


Figure 3.20. Flowgraph for the example

Actually decision points in the program affect the independent path count through contributing loops. Having two outgoing flows, an "if" statement introduces only one loop - with or without an "else" clause. So the contribution of a decision point to the complexity is its fan-out minus 1 (out degree -1). Also should be noted that a decision node is terminated with a "join" node that has the equal fan-in with the corresponding decision node's fan-out. The if statements count as single decision points each, but the case statement in the example (node c) counts as 3 decision points. Finally, loop control statements need to be addressed as conditional structures. They count as single decision points.

In the example above, the cyclomatic complexity is 6, applying any of the described three techniques:

1. Decision points +1: the nodes a, b, and c in Figure 3.20 correspond to the while, if, and case statements, respectively. The case statement has 4 out-flows thus contributing 3 counts to the result. Decision points = 5. Cyclomatic complexity = 6.
2. E - N + 2: E = 10, N = 6, 10-6+2 = 6.

3. Number of loops: 5 inner loops + 1 = 6.

Other test types

Black box and white box approaches were discussed, regarding modules. Unit testing corresponds to modules before they unite with the others for integration. Integration testing is a difficult development stage, where errors surface that were never thought about. Actually testing and integration usually go hand-in-hand. It is not recommended to finish the development and test at once as the "big bang" approach defines. Due to the non-linear characteristics of the complexity-size relation we prefer to locate errors in smaller units of code. An incremental test and integrate iteration is carried out. Any new module is first subject to unit test, and then it is integrated to initiate an integration test. Regression testing is also employed, to explore the unseen side effects of a new addition to the remote units tested before. This kind of tests defines the repetition of the previously conducted tests due to a new modification.

Validation tests serve the questioning of the requirements itself (are the requirements valid/correct?). Verification is done through testing the developed codes compliance with the validated requirements. Those two kinds of testing often get pronounced together and sometimes as an acronym: Verification and Validation (V&V). System tests are conducted after integration, over the entirety of the product. Verification tests can also be system tests, if the coverage is defined by the complete requirements.

Once the product is ready to pack, it is tested at the developer's site. Users can contribute by observing and commenting, even conducting the tests. This kind of a system test is called the Alpha Test. A similar test conducted at the customer's site by both of the stakeholders is the Beta Test. The beta version of a software product means that it is distributed with the sole purpose of gathering error reports from the users.

Integration

The complexity of software development is proportional with the size, but in a fashion that exceeds a linear relation. Integration can account for the extra effort on top of a linear addition of the complexities corresponding to the modules. Most of the projects face a problematic time after a long honeymoon with unit development, once they arrive at the integration stage. Some preventive measures will help ease this difficult task:

- Specify the interfaces even during the earliest decomposition activity
- Obey the less coupling / more cohesion laws
- Avoid side-effect causes such as global variables, uncontrolled access to data structures, while creating interfaces whenever different

layers are present. For example, a file input/output facility should be organized as an interface layer between the file access calls and the rest of the software.

Basically there are two strategies to integration:

1. top-down and
2. bottom-up

Also a mixed approach can be added as a third option:

3. the sandwich method

The top-down strategy suggests an approach of starting with the unit coding and testing of the top-most level module. Any module needs connections. Except for the lowest-level modules, all the modules control lower-level modules and depend on them. To be able to test a higher-level module, we need the lower-level modules. But with the top-down progress, they may not be available at the time of the testing. To fill this gap, "stubs" are written that imitate the lower-level modules with almost no functionality inside. The stubs merely stand for the connections and they return dummy values to the superior module just to make it execute. A breadth-first order is followed for the test-and-integrate activity.

The bottom-up strategy is the reverse of what top-down prescribes: lowest-level modules are first tested and integrated with the immediate superior module and the hierarchy of the structure is composed towards the top. This time the higher-level module may be missing and it would be impossible to drive the modules under test. A "driver" is written that could act as an executable program and control the modules under test. Figure 3.21 shows the driver and stubs for two integration approaches.

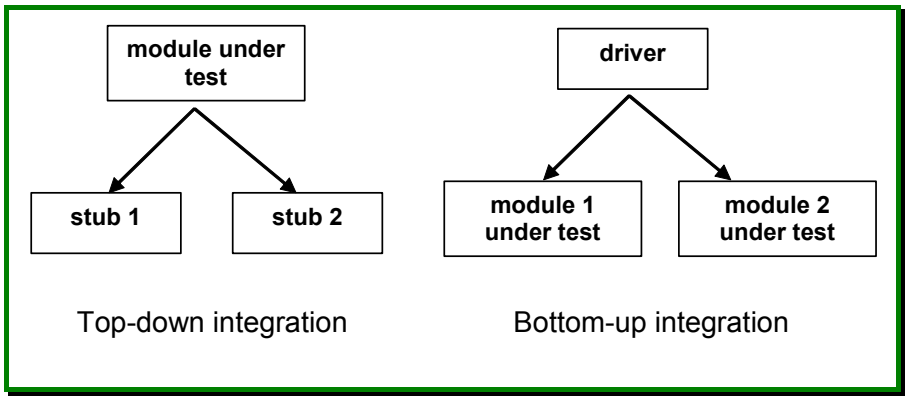


Figure 3.21. Driver and stub usage in integration schemes

In general, any phase of a development is preferred to conduct a top-down pattern. This is due to the holistic view that presents the system as a whole, in the beginning and later introduces lower-level details in a hierarchical order. If for some reason, the fine-grained worker modules are already present, one may select the bottom-up strategy. In a different case, in which the intellectual control is maintained through a top-down approach and some technologies are required to be exercised at the lower-levels of the architecture, the sandwich approach is appropriate. Applied widely, this last option requires good judgment on how the integration proceeding from two directions will meet. In other words, the decomposition assumed to be present before the integration starts should remain stable throughout for an efficient integration process.

Maintenance

This is an interesting phase in the lifecycle of a software product. Although software does not wear by usage there is still so much room for maintenance. A simple definition for maintenance can be any modifications conducted after product delivery. There are different reasons for getting involved with this expensive kind of effort. First of all, there will be defects discovered as long as the software is in use. There may be newly discovered opportunities if the current version of the product is adapted to other functional requirements or other platforms, for instance. Some basic reasons for undertaking maintenance task are:

- corrective
- adaptive
- preventive

Of course this list can be extended and its items can be further classified. Corrective maintenance is the repairing of known defects. Any software product has an accumulated list of errors throughout its usage history.

Adaptive maintenance involves the modification to port the software to another operating system or platform. Technologies change so fast and users expect to solve the same problem using new technologies. A program developed for early operating systems would be required to run on windows, then in a client-server architecture, and finally to run as an Internet application. There may be regulatory/legal changes the business environment has to cope with and the software automating the business has to adapt to such changes. Preventive maintenance is to modify the software so that it has a better structure to avoid defects to surface later. In other words, before the errors make themselves known, they are eliminated or the less structured parts of the software are improved so that it is less likely to contain defects, making it easier to fix.

A software complexity property should be remembered here that changing a line of code is more expensive than creating it for the first time. Especially for a finished development, one might try to avoid such trouble. On the other hand, especially if the initial development was not disciplined and the product is heavily being used, the error reports and the tedious repair efforts may force the developer organization to an overhaul, as soon as possible.

On the other hand, the picture may not be that pessimistic, after all. Especially if the personnel who were involved in the initial development are still available, maintenance is easier. Insight into the existing code is very valuable. Poorly documented code is always difficult to maintain. Existence of documentation is another asset that helps with maintainability of software.

Before starting the expensive maintenance project a feasibility analysis should be conducted. Expected life of the product can be estimated. How much maintenance will be required for every year is another estimation that needs to be conducted. This forecast should take into account the increasing maintenance costs for each year due to more problems expected to surface, especially in software that has not been well engineered. Adding the yearly maintenance costs for the expected remaining life of the software will yield the total expected maintenance cost. This total-maintenance cost can be compared to the estimated re-development cost, making it possible for a "reengineering" avenue to be considered also.

Reengineering

Reengineering is the process of developing an existing software product again. There may be several reasons to conduct this activity. Most of the time, it is done to re-structure the codes and the design model behind them because it is very difficult to understand or maintain the existing system. Reengineering consists of two steps:

1. reverse engineering
2. forward engineering.

Moving in the opposite direction of the Waterfall tasks defines reverse engineering. So, starting with machine code, we can move up to produce the assembly language. This is the lowest-level reverse engineering. The target may be a higher-level language source code, or a design model, or even the requirements.

Reverse engineering is also a difficult task. CASE tools now offer aid in a variety of levels of reverse engineering. Having a working program at hand is an asset that can be taken as an operational requirements definition.

Once the requirements model is reached, forward engineering can start. After a careful investigation of the requirements model for possible improvement, development can proceed with the goal to provide documentation and models that are complying with modern techniques and standards. Reverse engineering studies the current implementation and forward engineering defines the "to be" version.

Summary

Initial approaches to structuring the software development process are introduced. Starting with phased approaches, the industry adopted some standard techniques for requirements analysis and design. Dataflow diagrams were widely employed by systems analysts. Entity Relationship diagrams accompanied the dataflow model. For design, requirements models were enhanced and development-related information was added. Also structure of the software system had to be designed. Earlier work on modularity was guiding the architectural specifications. Later, interface was introduced as an important design component, both for user interactions and the inter-module interactions. Data, structure, interface, and procedural design were conducted, supported with graphical techniques and tools. Integration surfaced as a problematic phase where uncovered errors would be discovered, directing the developers to revisit the previous activities towards discovering the sources of the errors. Modules were tested before integration and the partial system was tested after the integration of each module. The testing and integration process either followed a top-down or a bottom-up procedure. Various types of testing emerged as another engineering field, for validating, verification and other quality-related quests. All the phases of the waterfall lifecycle were supported with tools to produce fresh code line-by-line. Huge systems were developed owing to the pioneering engineering approaches for software development despite the frequent failure reports. Finally, maintenance is presented as an expensive task that cannot be avoided. There are different reasons for an existing product to undergo modifications.

Questions

1. Draw Dataflow diagrams for a flight reservation system. There are different flights depending on the date, time, origination and destination locations. Aircraft are assigned to flights. Lists of aircraft, locations, personnel, and customers will be maintained. Booking and sales will be carried out by the personnel.
2. Draw Entity-Relationship Diagrams for the problem in Question 1.
3. Construct a Requirements Directory for the problem in Question 1.
4. Starting with the Dataflow diagrams, try to construct a structure chart for the problem in Question 1. You may enhance the DFDs as your design might require.
5. Design the interface among the major modules of your design: specify what kind of parameters should flow across the modules, their formats and synchronization requirements.
6. Draw the screens for the user interface for the application described in Question 1.
7. Draw a state diagram for the reservation operation: Your reservation advances slowly: first flights are browsed and one flight is selected. Then the seating plan is viewed and a seat is reserved. Finally this reservation is bought. Meanwhile, after your selection of a flight, another person will attempt to quickly reserve the same seat on the same flight and finish buying before you finish reservation. Identify different states and give meaningful names to them, also identify inputs to the state machine, and outputs. Designate a reset state and at least one final state (could be the same as the reset state).
8. Explain why a complete software system is not tested at once after integration, and rather modules get tested and stages of integration are tested also.
9. Define the three elements of structured programming and explain how one could violate those structures.
10. Describe the differences and similarities among testing, debugging, and maintenance.

References

Chen 1977 — P. Chen, *The Entity-Relationship Approach to Logical Database Design,* QED Information Systems, 1977.

Dennis 1973	J.B. Dennis, "Modularity" in *Advanced Course On Software Engineering,* F.L. Bauer (ed.), Springer-Verlag, New York, 1973.
Hatley and Pirbhai 1987	D.J. Hatley and I.A. Pirbhai, *Strategies for Real-Time System Specification,* Dorset House, 1987
McCabe 1976	T. McCabe, "A Software Complexity measure," *IEEE Transactions on Software Engineering*, Vol. 2, December 1976.
Pressman 1997	R.S. Pressman, *Software Engineering: A Practitioner's Approach*, 4th Edition, Mc-Graw Hill, 1997
Simon 1969	H.A. Simon, *Sciences of the Artificial*, MIT Press, Cambridge, Massachusetts, 1969.
Ward and Mellor1985	P.T. Ward and S.J. Mellor, *Structured Development for Real-Time Systems,* Yourdon Press, 1985.
Yourdon 1989	E.N. Yourdon, *Modern Structured Analysis,* Prentice-Hall, 1989.

Chapter 4: Object Oriented Software Engineering

After a relatively long struggle to equip software engineers with better tools, first software-conscious understanding shaped up as the industry converged on the techniques that supported Object Oriented (OO) development. Change that usually suffers resistance was now offering more than only a discipline to the process. New concepts had to be mastered, even by the programmers. The OO languages maturing by the late 80s were more difficult to learn than procedural languages. Reluctant to new methodologies, the industry debated the maturity of the OO technologies for serious projects. Today, any new project is developed using OO unless the company is not yet caught up with the transformation from the traditional inertia.

The idea gained popularity and it did not take long before the field began to regard Object Orientation as a panacea to all problems. A more natural modeling made it easier to keep the system under intellectual control. However, difficulties with the new orientation were soon uncovered: Developers were getting lost in "inheritance" depths. Nevertheless, the industry considers that there is no better approach as yet, so it is advancing with what is available. Contemporary technologies find a way of accommodation in OO representations. Good practices from the traditional era have already been incorporated.

No matter how good a technique is, it should not be expected to be the silver bullet for wicked software problems. The problem is much more complex than that, and better efficiencies in its solution will not do away with the complexity. Anyway, so many of the ideas maturing in the infancy of software were formalized in Object Orientation. In the course of learning this new paradigm, a more formal understanding of some classical principles is also gained: A programmer, after being exposed to an OO language will produce better organized program codes even using a procedural language.

Object Orientation frees the mind from the constraints imposed by the days of early hardware. A lucky programmer who starts with OO languages will be initiated to higher-level abstractions at the outset of learning. The entities in the programming model represent their real-world counterparts closely. An object includes data and function primitives packed together which points to the fundamental OO principle: encapsulation. This very idea of grouping related items together is a strong notion that helps in the organization of the code. Not only object's internals, any item is soon organized with its peers. A developer now sorts the elements of the system with respect to their semantic proximity. That is also related with the fundamental design rule that is *cohesion*.

The encapsulation concept is closely related with the "controlled access" principle. An object provides access functions for external entities if its data needs manipulation by them. A good OO developer never uses global variables, never allows direct access to the properties of an object, classifies properties and operations in a hierarchy, has an understanding about protection levels of properties (variables), and also about an object's self awareness, in that high-level operations can be dispatched without regarding the type of the object (through polymorphism). Some of these concerns can be achieved in other environments and OO background expertise provides consciousness to do so.

Objects, being structural units as well as logical elements are also a good mechanism to manage coupling and cohesion. A set of objects can be defined to represent a system and the set can inter-relate the objects. Some of the relations are "structural", in that they involve aggregation of objects into larger groupings and inheritance of object definitions (classes) from a more general class in the logical hierarchy of classes. An object should be a meaningful unit, therefore properties and methods to include should naturally be cohesive. How much coupling that is foreseen among a set of objects can change the way object boundaries and compositions are determined.

Object Orientation

There are three fundamental properties for a modeling tool or a language to be Object Oriented (OO). These are:

- **encapsulation,**
- **inheritance,** and
- **polymorphism.**

Encapsulation suggests that an object represents its data primitives along with its functional primitives. This means variables and functions are declared together inside an object. In more general OO terminology these are properties (also called attributes) and methods. In the real world, the actions that can be done by or with an object are as important as its characteristics - also referred to as attributes or properties - such as color and weight. For example, a coffee cup has a diameter, color, weight, shape, and material as its properties but, more importantly, it can be filled, emptied, sipped and broken. A cup is for drinking rather than for observing how it looks. Such functions packed together with its static parameters define the cup. Figure 4.1 represents this encapsulation on the coffee cup example. Also encapsulation will be utilized to aid in information hiding: Only the important aspects for observation from outside will be represented as properties and methods.

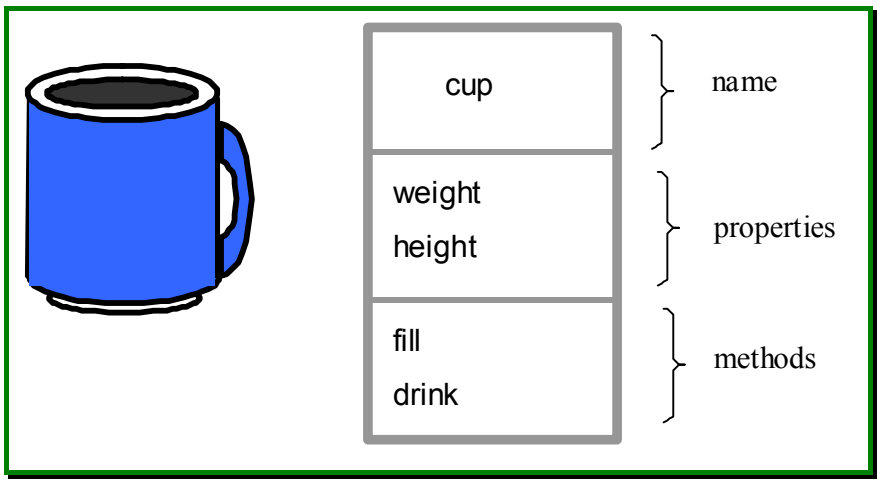


Figure 4.1. Encapsulation for a coffee cup.

Actually such declarations for the properties and methods take place in a class. A class is like the type definition and an object is like a variable of that type. An object should have a unique identifier. A class can be instantiated to generate several objects of that type (class).

The members in a class - properties or methods - also come with their protection definitions. The three protection levels (public, protected, private) are widely considered standard and most environments support their representation and implementation. Public members are open for access to outside: in a programming environment, calls from anywhere in a program can be made to the public methods of a class instance. Private and Protected members are accessible by the objects of only this class, and that of the classes inheriting from this one, respectively. Inheritance concept will be explained later. A developer should make it a habit to protect properties and declare the methods public as the default configuration. Exceptions to this prescription can be incorporated later, as the need arises.

Observing a class definition, it is possible to understand what an object stands for and what it can do. The question of how the methods work is not visible at this point. That is how it is desired any way; information hiding principle suggests that only the required amount of detail should be presented at any point. The set of public members can be referred to as the “interface” for the class. That is the part accessible from outside and that is what is necessary to use this class. The interface concept is another important tool for the organization of complex software. Especially with the emergence of the component technologies, the concept is explicitly utilized to separate the definition of a unit from its implementation.

Object Based Environment

The popularity imposed stress on toolmakers to come up with OO products, even if they are OO versions of their existing products. Soon there were so many tools claiming to be OO. In fact, some of them were only Object-Based. Early versions of some BASIC and Ada compilers can be regarded as Object-Based although for some time they were presented as Object Oriented. The difference is that such tools did not incorporate more advanced capabilities other than "encapsulation." If they had inheritance and polymorphism, they would be qualified as OO. Polymorphism requires that inheritance is available. Both concepts are more difficult to provide when compared to encapsulation.

Interaction

Objects interact as a requirement for their interconnection to serve as the building blocks of a system. A system is a connected set of elements to solve a common goal. Such units could be traditional software development modules, or modern components in which case they are built for multipurpose usage – not for a specific system. Reusable components are for easy composition into different systems. In any case there are blocks that communicate. The communication is through "messages." The messages can be as simple as traditional function calls (OO functions are methods declared inside classes) in a single processor environment, or they could be strings of data transferred over communications media among remote nodes of processors. An object can represent a reusable component or a traditional module.

An OO model is a distributed one. There are stand-alone objects that exchange messages. The most popular implementation of the message mechanism is the method call facility. For any message, the calling and the called objects can be observed as a client and a server. The initiator of the message is requesting a service from the receiver. The "server" object that also provides methods for the service should maintain required data as its property. The message by convention should have the same name as the method. For an understandable model, meaningful names should be given to the methods that also comply with the direction of the call.

A distributed model is more complex than a lumped one. That is the main theoretical difference between the traditional software models and the OO models. The traditional or the structured approaches possess a single state space for the whole system. Whereas, an OO model is a composition of individual state machines that communicate. As a principle, less powerful models cannot represent all the problems that a more powerful model accommodates. For this reason, it is not advisable to start a modeling task with OO approaches and later transport the model to a traditional environment. The distributed nature will not be easily represented in a

traditional model. Even if the problem is physically not distributed, its OO model may be utilizing the distributed mechanisms with its more expressive power.

Classification

Inheritance is meaningful in a classification network. Objects can be analyzed to find generalizations among them and the common factors for a set of objects define the class. Classes further can be generalized to super classes. Likewise, any class can have specializations and may lead to the derivation of many sub classes. The factors utilized in the generalization/specialization considerations are the properties and the methods. Its sub classes also own all such members of a class. Sometimes the super class is called the base class and the sub class is called the derived class. Figure 4.2 displays a simple classification scheme where the specific vehicles are a specialization of the general "vehicle" class. The figures in this text follow UML [Booch et al. 1999] graphical syntax where classes are rectangles with three compartments for class name, properties, and methods.

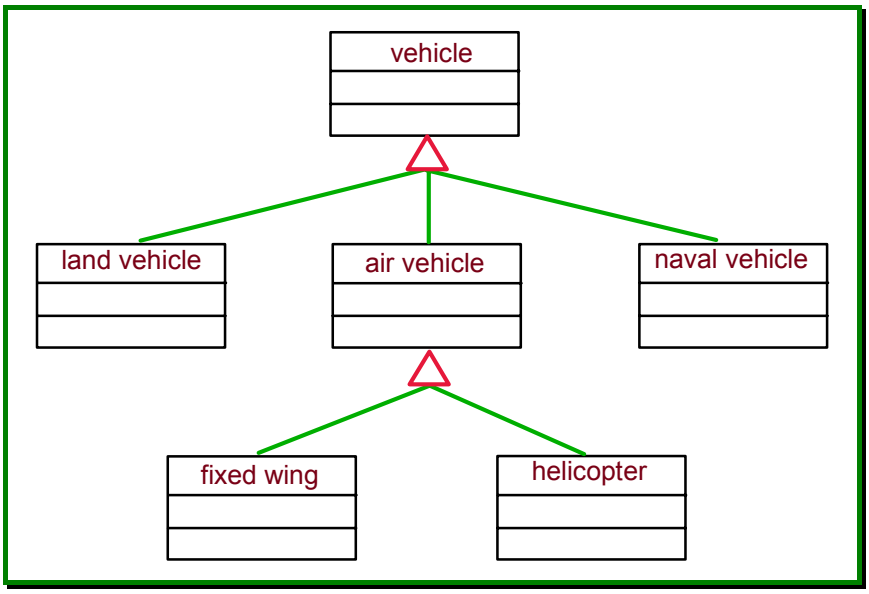


Figure 4.2. A Classification example

Classification is by nature a complex task. For years, the topic had involved librarians in an effort to represent the material in an easier to locate indexing mechanism. A single item could take place in different categories based on the classification parameters. If all the concepts could be perfectly classified in a single-root tree structure, where any item is a special kind of only a single general item then the problem would have been much easier. In that

case, there would only be one parameter to determine the classification at any node.

Classification goes hand-in-hand with the inheritance concept. A sub class by definition inherits all the members of a super class.

Inheritance

When a class is defined in terms of specialization of a base class, the derived class automatically owns any properties and methods of the base class. To use these members it is not necessary to explicitly state that a member is obtained through inheritance. If for example there is the "cup" class and the "coffee cup" class is derived from it, the weight property and the drink method declared in the cup is automatically present in any coffee cup. The coffee cup may additionally have a handle that needs to be declared in its class definition. Figure 4.3 displays the inheritance of x and y coordinates for a graphical application where the base class is a "graphical object" declaring those properties.

There are many complications related to inheritance. The same member can be re-defined in a sub class just like it was declared in a super class. In this case the newly declared member "overrides" the inherited counterpart. This is more meaningful if the members are methods. For example the draw method declared in the graphical object class can be overridden by another method with the same name, declared in the circle class. In implementation, the two different draw methods can have different codes defining different actions when a draw method is called. If a circle object invokes the draw method, it is the latter one that is selected to run. The general rule is to search bottom-up for the same method (function name plus the parameters list define a method). Once found, the search is terminated and the first method found is the one activated.

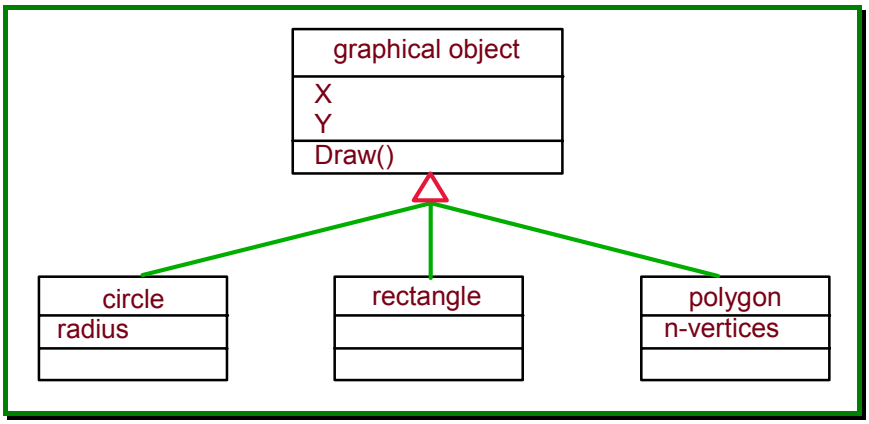


Figure 4.3. Inheritance

There are related concepts for the selection of the function to call once its name is used. The more primitive one is "overloading", which means the name of the function is overloaded with different meanings. The parameters list, however, helps in the identification of the exact function, although the name could be the same for more than one function. Both overloading and overriding are resolved at compile time. In other words, before run-time, the compiler knows which function is meant and determines the link between caller and the specific function. The more interesting concept is polymorphic function calls that will be explained further later.

Actually, if a member overrides a previously defined property or method, the derived object contains all of the defined members. If no differentiation is given during the access, the bottom-up search determines the member as the first found one. As an example, the fixed wing, air vehicle, and vehicle classes could all declare a "wheel" as an attribute. Now it is optional to pick any one of those wheels. For a fixed wing object, the default is the wheel declared in "fixed wing" class. If the wheel declared in "vehicle" class is desired, it can be specified as "vehicle.wheel" in most of the programming and modeling languages.

Multiple inheritance

Not all OO environments are equipped with this capability. Some languages include "multiple inheritance", such as C++ and Smalltalk, but some others do not, such as Java and some popular windows based Pascal versions. It is therefore important to know in advance, if there is going to be a language restriction for implementation. Unlike the traditional approaches, in OO development, initial modeling activities starting with requirements specification are closely bound to the implementation capabilities. Since similar models are refined at differing layers of detail – from requirements to coding - OO development suggests such a restriction. If the required

language does not support multiple inheritance, the modeling should omit multiple inheritance at any stage.

A class could belong to one super class as well as another one. In this case, the properties and methods of both the super classes are inherited. Figure 4.4 presents a multiple inheritance example. If possible, this mechanism should be avoided. It may not be a good idea for an object to have more than one class simultaneously, anyway. But the question is, is there a theoretical need for such a classification technique? Are there cases where multiple inheritance is the only correct solution? There does not seem to be a binary answer to this question. There are some cases where multiple inheritance is a better modeling option. Then comes the feasibility of incorporating this mechanism in the development of a current project. Some issues with additional complexity are introduced. The repetitive declaration of members in super classes is a problem repeated here with a different flavor: if two base classes declare the same member, which one is picked when an object of the derived class tries to access? The answer is undetermined. Explicit specification of the class-member pair is the safe access to such repeated members.

Interfaces

This term corresponds to a powerful concept especially utilized after the component technologies. In general, an interface is the publicly visible specification of the services and accessible parts of a module. Their structures are strictly defined in the component protocols. Other tools have used the concept also. The “h” files created with c programs actually are the interface definition of the corresponding “c” file. Likewise, some Pascal compilers require the “interface” and the “implementation” segments to be specified for any source code. The same policy had been an actual part of the Ada language.

Actually it is a good principle to develop interfaces whenever a system seems to be composed of layers or different subsystems. Such pieces should not come together with free connections but rather all the interactions should be localized to “interface” modules. This technique breaks the dependency of a module to others. If a module is changed the dependent modules do not have to be changed except for a local unit that is the “interface” section.

Following this convention, the public members of a class can be referred to as its interface. For a group of classes that are organized as a package to represent a part of the system, a class can be defined specifically to serve as the interface of this package. This special class can contain methods for external requests and their primary duty may be to dispatch the request to the actual service provider in one of the classes inside the package.

For this reason, Java includes a special kind of a "class" that is referred to as "interface." The Java approach limits this new kind of class and its inheritance to methods only. In other words, no properties (variables) can be inherited from an "interface." Although component technologies allow the declaration of properties in their interfaces, the principle of controlled access does not promote public variables. Java's excluding variables from the interface thus makes sense. This language allows inheriting from only one class but does not limit "implementing" many interfaces.

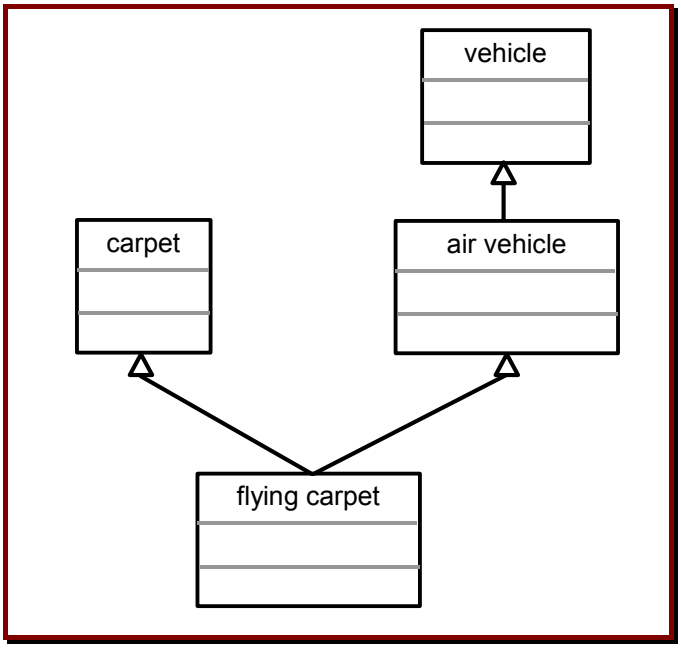


Figure 4.4. Multiple inheritance for the flying carpet class

Figure 4.5 represents the problem with repetitive member declaration in the base classes taking part in multiple inheritance. Actually the different members are both available and one has to be specifically identified at access time. The amphibious class in Figure 4.5 has two A properties, one inherited from the land vehicle, and the other inherited form the naval vehicle classes.

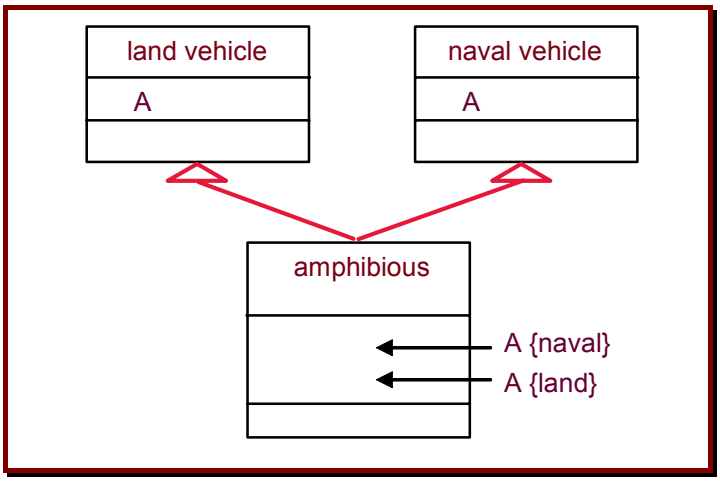


Figure 4.5. Repetitive naming problem in multiple inheritance

There are cases where the different levels of classification are required and it is not easy to determine any priority among the classification parameters. In other words, the parameter to base a classification can be employed before (at a higher-level) or after another specific parameter. Figure 4.6 depicts this situation with a classification of animals employing the parameters of "feeding" and "leg count." In Figure 4.6a, first feeding parameter is applied followed by the leg count. In Figure 4.6b, the parameters are applied with different order.

Depending on the domain the classification parameters may have priorities; abstraction levels also apply here. Then it is easy to employ the classification sequence. With our limited knowledge in zoology, the classifications in Figure 4.6 imply that the two parameters are of equal priority. This can be concluded since the two alternative classification orders in Figures 6a and 6b do not present any difference. In such cases incorporating multiple inheritance may be justifiable. Figure 4.7 presents the multiple inheritance adaptation of the example in Figure 4.6. In this case, a derived class picks among the base classes, one class at a time for each classification parameter. The example illustrates the picking of herbivorous eating pattern, four-legged leg count parameter and fur class for the skin type.

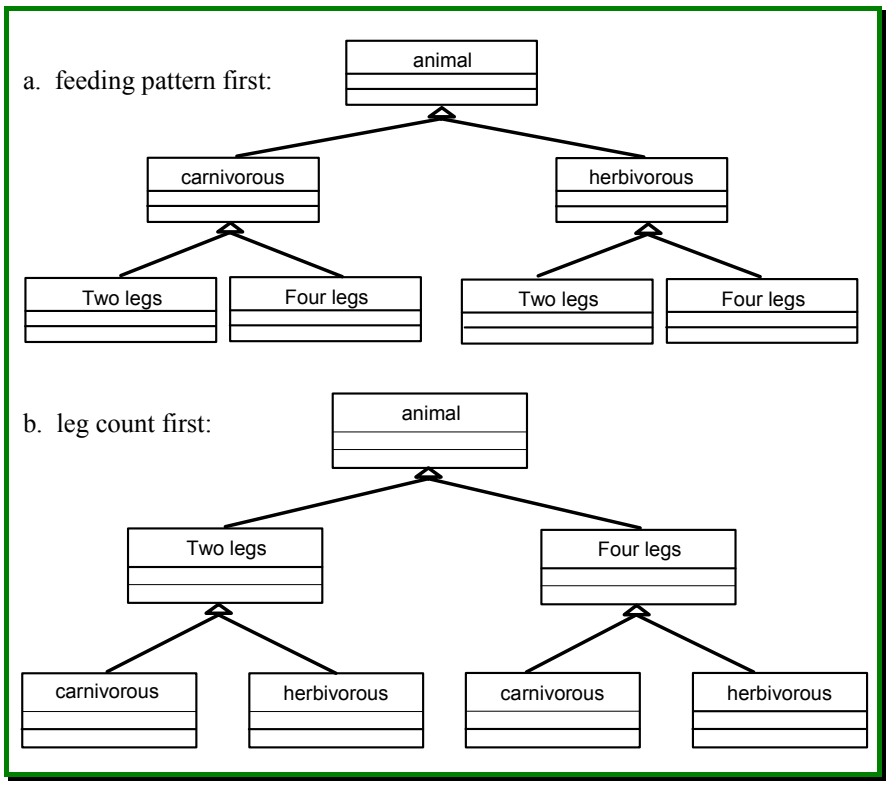


Figure 4.6. Different orders for classification parameters

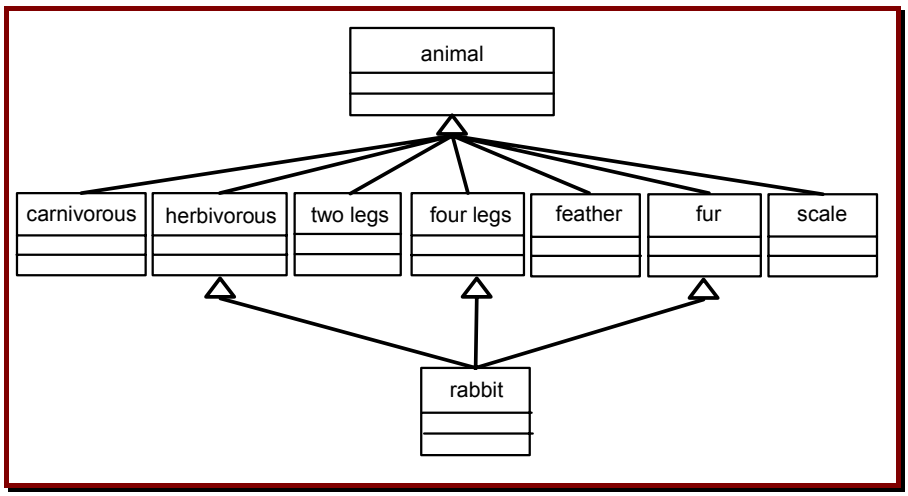


Figure 4.7. Applying different classifications at the same level

Polymorphism

This property is the automatic selection of the correct method at run time. If the same name is used for methods defined in different classes, without checking the type of an object its method can be activated. This capability is valid for a related set of classes – that inherit from the same base class. The virtual method declared at the base class will be replaced by a real method once activated for a specific object. This is best explained through programming examples. A collection of classes should be inheriting from a base class where the method is declared. Also, the same method needs to be defined differently in each of the derived classes. An object of the base-class type will be assigned one object at a time, among a set of objects corresponding to the derived classes (an object of the base class is capable of being assigned to any of its derived classes). After each assignment the object of the base class will activate the method. Each activation through the same method name will locate the correct method depending on the object type that was last assigned.

The trick works in the run-time environment by changing the type of the main object to the last assigned objects type, once assigned. Then, the object knows who he is and hence will call his own method. So the object of the base class has the ability to contain any descendent classes and will behave (change to) as any such object assigned to it. Figure 4.8 depicts a polymorphic "run" method example where a list of items is instructed to run, without checking what kind of items they really are. This provides insight on the organization of an OO program: To build up the definitions such as inheritances and polymorphic methods one needs extra effort in the beginning. Once that structure is in place, powerful operations can be achieved with concise code that is easy to understand.

There are some related concepts that need to be explained along with polymorphism. Virtual methods are those defined in such base classes only with the purpose of replacement at run-time. These methods do not need a definition (body). Some languages also allow a definition and activation of such replaceable methods. A class that will never be instantiated as an object but is defined only for other classes to inherit is called a virtual class.

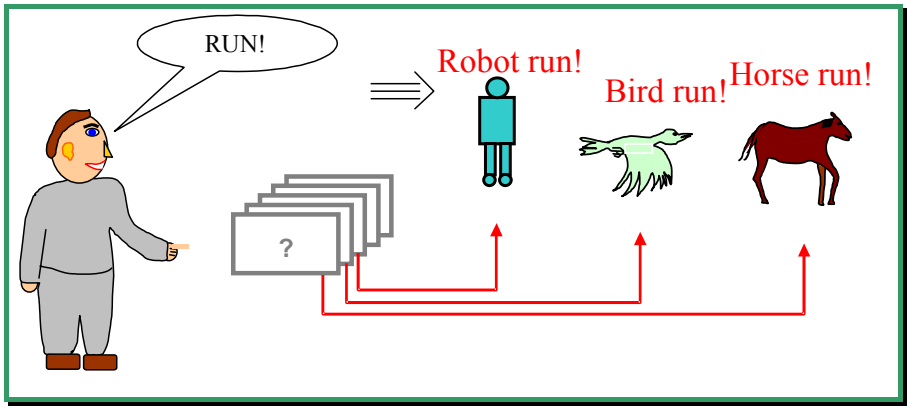


Figure 4.8. Polymorphic calls

Composition

Inheritance is a very powerful mechanism OO systems offer. Besides polymorphism, the reuse in the structural definition is also achieved through this capability. The other mechanism to build more detailed modules out of existing ones is composition. This is physically including classes inside a class. Rather than simple properties, other classes participate in the container class as being internal parts of it. If a windows based graphical programming model is analyzed, so many levels of inheritance will be witnessed. However, huge information systems defining the bulk of the software industry do not utilize inheritance much.

While inheritance assumes the ownership of a duplicated set of all contents of the base class, composition implies a part-whole relation among classes. The container class contains other classes as if they were its properties. Since the two mechanisms have different semantics, they need to be used correctly in models. Figure 4.9 presents a class diagram that includes composition as well as inheritance relations. The hollow triangle as an arrowhead stands for inheritance and a diamond represents composition. The example in Figure 4.9 represents inheritance from the vehicle class by the air, land, and the naval classes whereas the land class composes (has) the wheels and the body classes. Actually, any kind of connection in a class diagram defines a “relation” between classes. The relations should have self-

descriptive names. Inheritance and composition are part of OO modeling formalisms, as pre-defined relations.

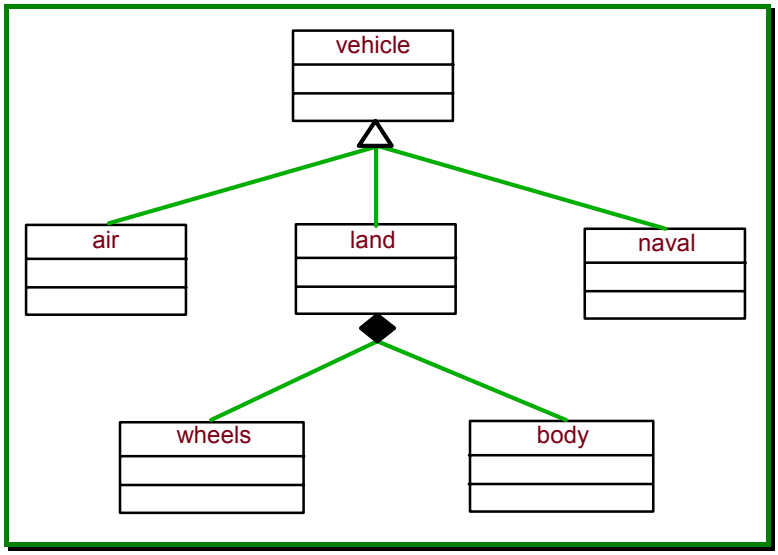


Figure 4.9. Composition and inheritance relations in a class diagram

Inheritance stands for the "is-a" relation whereas composition is for the "has-a" relation. In cases where a developer is not sure which relation to use, the meanings of the is-a and has-a can be questioned for the application and the appropriate one can be selected.

Besides the low-level composition among classes, the architecture of a complex system can be defined as compositions of bigger-granularity modules. Best modeled by sub-systems in UML, such big chunks can be incorporated in the model by drawing one inside the other to represent the part-whole relations. Actually the first step in OO design is usually a system/subsystem architecture definition. If this high-level composition relation is going to remain at the architecture level, such graphical representation is sufficient. However, a component-oriented approach where the entire system is represented in terms of levels of composition among the components of varying granularities and abstractions (more logical/more physical/ actual components…) requires a better representation of the composition hierarchy. Such needs will be addressed in component orientation related chapters. UML's high-level composition mechanisms are presented in Figure 4.10 as a "component diagram" where subsystems model large-grained chunks.

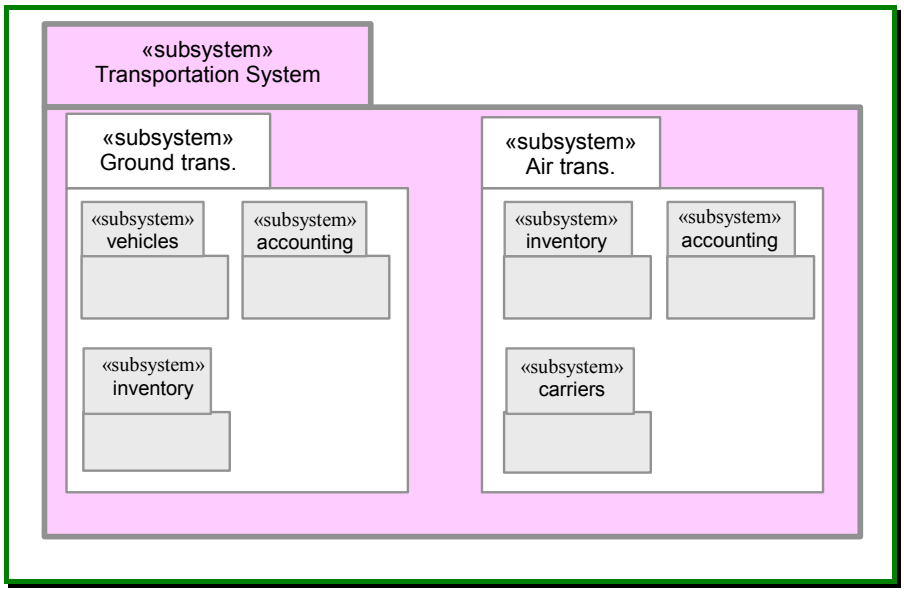


Figure 4.10. Subsystems in UML

Finally, a loose version of composition, namely aggregation, must be mentioned. Not every OO modeling method includes two kinds of composition relation, whereas UML does. The graphical representation is a hollow diamond for aggregation, and a filled diamond for composition. If the contained objects will not survive in case the container object is deleted then a composition relation must be used. If the contained objects can still survive then aggregation is the better choice. For example if professors in a department will still exist as part of a university system after the department is closed, there is an aggregation relation between the department and the professors. If the rooms of a building have to be demolished with the building, composition is the relation between the building and its rooms.

Composition versus inheritance

Modern OO oriented software engineering is intimately related with component based development. Components are by nature composition oriented rather than inheritance: They are built for composition. The development of a component, however, can exploit any technique OO tools offer, including inheritance. From an OO point of view, inheritance is more a logical description of similarities to guide the development of a unit. Composition on the other hand does not regard the development of smaller unit internals; it is used in the integration of readily available executable pieces of code for the creation of complex systems.

From a programmer's perspective, if the meaning of a model is ignored, using inheritance in any case may be desirable. This way, shorter codes can be written. Of course, both the models and the code should comply with

correct relations among modules. Two programming examples are presented below to display the usage differences for the two mechanisms. Inheritance allows access to the obtained attributes as if they were local. Composition requires the mentioning of the class names before their internals are accessed.

```
// inheritance example:
//-----  class declarations typically in a "h" file for C++ programs -----
class wheel {
                    screw s1,s2,s3,s4;
};
class car :: wheel {                // wrong inheritance! Car is a wheel ???
  ……
}
//---- a segment in the implementation part (in ".c" file) -------
car car1;
car1.s1 = …….                       // s1 is part of car1
```

```
// composition example:
class wheel {
        screw s1,s2,s3,s4;
};
class car {                         // no inheritance
        wheel W1;
};
//---- a segment in the implementation part (in ".c" file) -------
car car1;
car1.w1.s1 = …….                    // wheel "w1" has to be mentioned.
```

Object Oriented Methodologies

Many OO methodologies have been developed. Some widely known methodologies can be accessed through these references: [Booch 1994, Booch et al. 1999, Coad and Yourdon 1991, Coleman et al. 1994, Jacobson 1992, Rambaugh et al. 1991]. The consistency in representation from requirements through coding provided a similarity among the methodologies. So many levels of details need to be addressed and it is impossible for any two approaches to be identical in all of the activities. Nevertheless, it is easier than traditional methodologies to arrive at general commonalities. As a consequence, there have been two successful attempts to unite different approaches into a standard methodology. Fusion Method [Coleman et al. 1994] selected better practices and combined them with additional notations and gained popularity quickly. Later Unified Modeling Language (UML) was introduced as a result of a similar effort by three names that were already known for their widely accepted methodologies.

An important difference can be mentioned among the methodologies based on the classification/specialization choice in the determination of the object and class hierarchies. While the bottom-up approach is favored more through starting with the objects and continuing with their classification, there are some that suggest to start with general abstract classes and arrive at specific objects after a chain of specialization refinement. If the development starts with a written description of what the system is expected to do, it may be feasible to select the bottom-up approach because specific candidate objects will be located in the text.

General approach

Either Class Responsibility Collaboration (CRC) modes or use-case diagrams are used to support requirements analysis for the majority of the approaches. Every class assumes responsibilities relating to some of the system functions in CRC, collaborating with others to accomplish those responsibilities. Figure 4.11 depicts a CRC example. In use- case diagrams, system functions are presented and their interactions, including those with actors, are defined. Then at the time for drawing the class diagrams, objects and classes are determined.

Potential objects from a problem specification are [Pressman 1997]:

- External Entities,
- Things,
- Events,
- Roles,
- Organizational units,
- Places, and
- Structures.

Coad and Yourdon [1991] suggests the six criteria listed below, for the inclusion of an object in the model:

- An object must retain information,
- Must include services that could change the values of its attributes,
- Should have multiple attributes during analysis,
- All occurrences of an object should have the same set of attributes,
- All occurrences of an object should have the same set of methods, and

- External entities essential to the operation of the system should be objects.

Classes are refined with the definition of their properties and methods. Relations among the classes are introduced and class hierarchies are formed. Next comes the modeling of the interaction among objects. The models set forth so far will be refined through verification by scenarios.

Design activities follow with further details on the class diagrams and interaction models. Another aspect that was not addressed in requirements is the structure of the system. Starting as the initial design activity, system decomposition into subsystems is a structure related task. Also packaging related classes/objects into components or subsystems is part of structure design.

Since OO approaches are the *de facto* standard, any supporting technique is imported into OO methodologies. Some examples to those techniques that are not necessarily OO are the state charts, sequence diagrams, and use-case diagrams. Some of these techniques were in use before Object Orientation was invented. Figure 4.12 displays the primitives used in the use-case diagrams as actors, use-cases, and dependency relations. Also an example use-case diagram is presented in Figure 4.13.

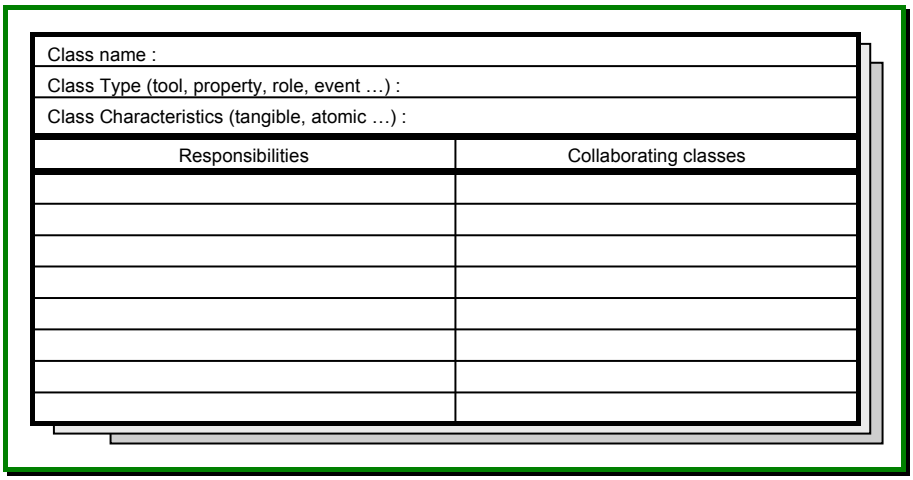


Figure 4.11. A Class Responsibility Collaboration model

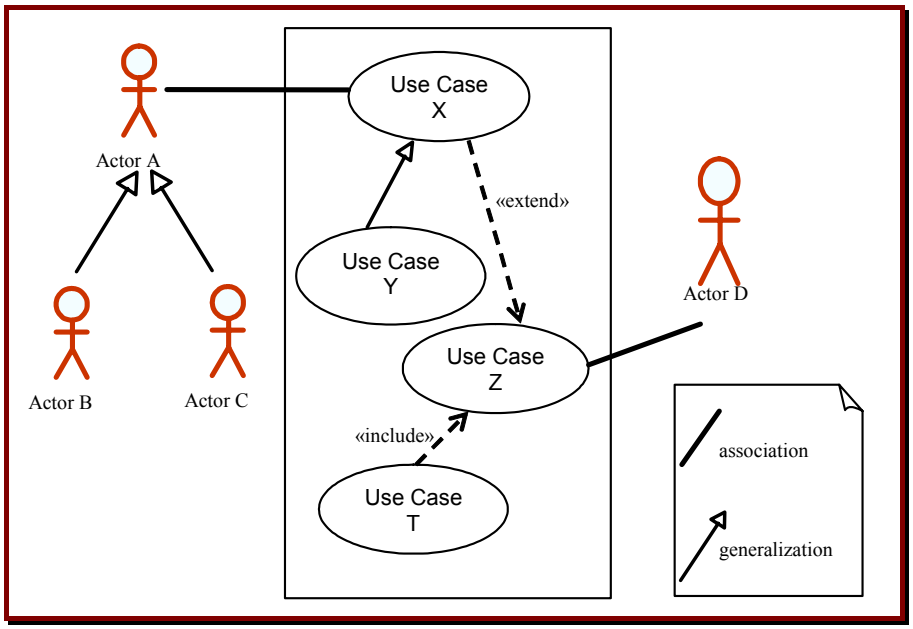


Figure 4.12. Use Case diagram primitives

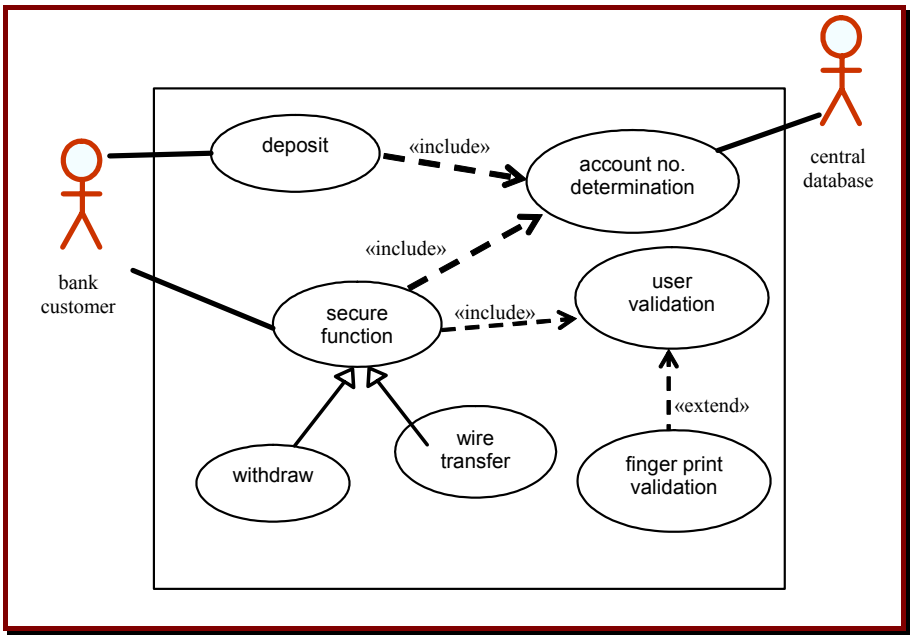


Figure 4.13. An example Use Case diagram

Requirements analysis and specification

Two models have already been introduced that are CRC and use-case diagrams with the latter being part of UML. No specific methodology is followed in this chapter. UML is selected for graphical representations. Techniques will be presented in an order that could stand for a coarse methodology employed by experienced engineers.

Use case analysis

Use case diagrams are suggested to be simple enough to present the system definition to the customer. Like other diagrams, these have the capability to be used in advancing details to the procedural levels but that is not desired. Unlike the dataflow diagrams of the traditional era, only one level of diagrams rather than a hierarchical system of diagrams is desired. If an analogy to dataflow diagrams helps, a level 0 dataflow diagram is perhaps the best counterpart. Use case diagrams incorporate actors that interact with the system components. Here, the components are highest-level functionalities that will be referred to as "system capabilities" and, associated immediately under those capabilities, the sub components are referred to as "system functions." Dataflow diagrams only displayed the interaction among the external entities and the system. Use case diagrams replace the external entities with actors and also allow for the interactions to be drawn for different system functions rather than only the complete system. Some code needs to be developed, corresponding to the actors in this model. Whereas, the external entities in the dataflow models are strictly external; they are not part of the system to be built. The actors will be represented as classes in class diagrams. Such classes can be thought of as the interface related to that particular actor.

A separate use case diagram is suggested to be drawn per system capability. The use cases in the diagram (ovals) correspond to system functions under the capability. Details related to the use case need to be explained. Denoting the expected process in terms of ordered events accomplishes this task. A scenario is thus described textually and then formally in terms of "interaction diagrams."

There are two kinds of interaction diagrams in UML: collaboration and sequence diagrams. There are objects and messages in these diagrams, corresponding to the run- time interaction among objects. Dynamic modeling is thus achieved. Messages are ordered in time by numbering them in collaboration diagrams or by utilizing the vertical axis in sequence diagrams for the time dimension.

An OO model consists of different diagrams. Care must be taken to preserve consistency. The analysis task is basically supported by three diagrams: use case, interaction, and class diagrams. The messages used in the interaction

diagrams must be declared in the class diagrams as methods in classes. A method is declared in the class that plays the receiver role for the message.

For problems that require complex temporal management, the state machine concept can also be incorporated. For this purpose, a class can own a state chart that is another one of the UML's diagrams. Activity diagrams also contain similar information, organized with respect to "activities" that take place while the object is in one state.

Class diagrams

Classes and relations take place in class diagrams. The main structural unit of an OO model, a class, presents its properties and methods. Also, a variety of relations take place in a class diagram. The structural relations are inheritance and composition. Also other relations like those of Entity Relationship Diagrams are included and referred to as associations in UML.

Class diagrams are like the global variables for a program. All the classes do not have to take place in a class diagram. There may be more than one class diagrams for a system. Different relations could be used in different diagrams and classes could be repeated across diagrams. Figure 4.14 depicts a class diagram that only includes associations just like those in the Entity Relationship Diagrams (ERD). Note that pluralities are more specific than those in the ERD: any range can be specified. Sometimes a class diagram is called an inheritance diagram or a relation diagram based on what associations it includes. Usually many kinds of relations take place at once. Other class diagram examples can be found in Figures 2 through 7. Figure 4.9 is another class diagram that also includes a composition relation.

There may be complex relations that require further specification than merely the name, direction, roles on both ends of the relation, and pluralities. In such cases, the relation is modeled by an "association class." The principal classes taking part in the relation are connected by an association link that is further connected by a dotted line to the special association class.

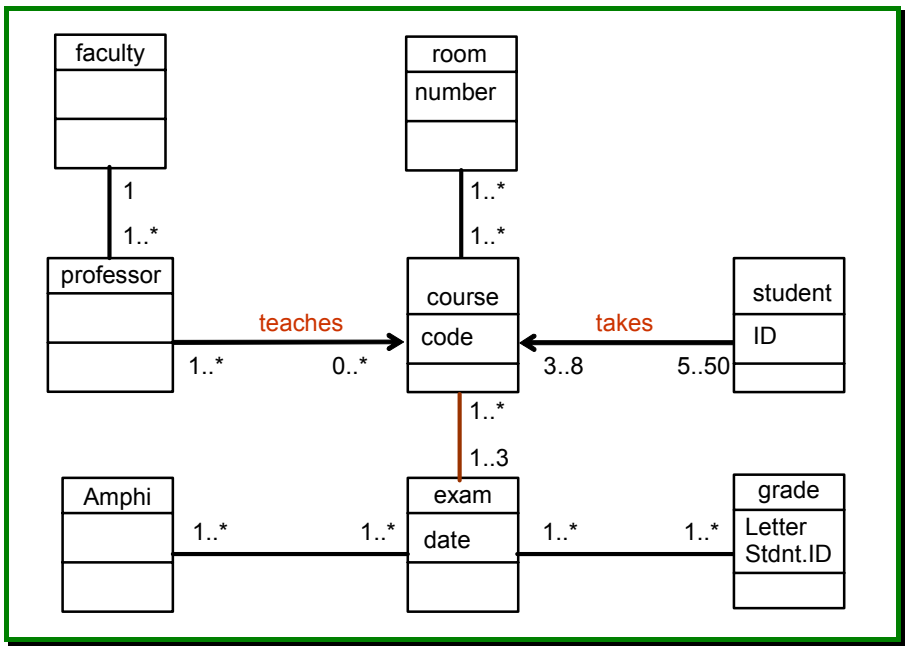


Figure 4.14. Class diagram with associations

Figure 4.15 displays a class diagram with inheritance and association relations. Also a reflexive relation is shown where the same class is associated with itself through the "calls" relation. Also this reflexive relation indicates roles on both ends of the arrow. The relations can have direction, name, pluralities and roles - all optional. Of course the sole reason for constructing a model is to understand the system so it is very important to include all possible information along with the relations. The examples in Figure 4.15 also present two alternatives in determining the classes for similar objects; a telephone user class could assume the different roles of "caller" and "callee" or, alternatively, those roles could be assigned to separate classes.

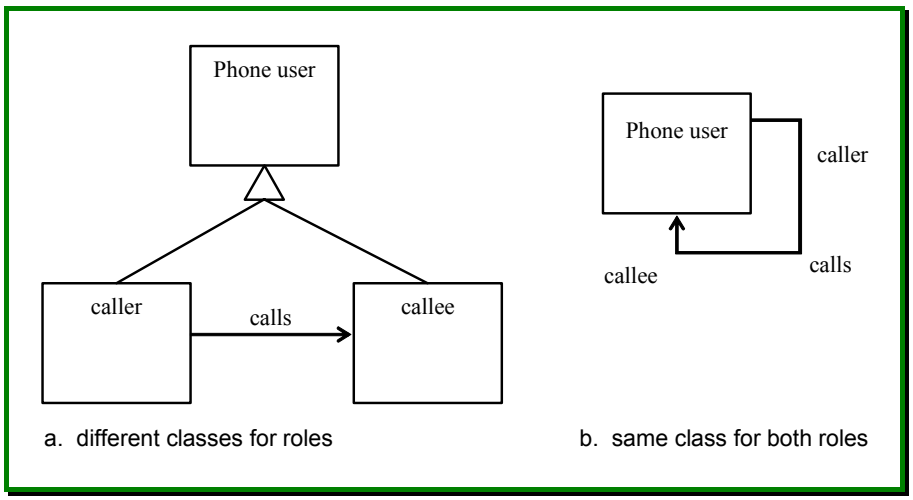


Figure 4.15. Further relations in class diagrams

Interaction diagrams

Interaction diagrams contain objects modeling dynamic behavior. Classes are logical definitions but objects exist in run-time and dynamic modeling corresponds to the run-time. Briefly introduced in the use case analysis section, the collaboration and sequence diagrams contain similar information. Some environments can automatically translate a collaboration diagram to a sequence diagram with only a single press of a button.

Figure 4.16 depicts a collaboration diagram that corresponds to the problem with class diagrams presented in Figure 4.15. This collaboration diagram assumes different classes for different roles. It should be noted that the first message (Pick-up) is initiated by the caller, although the name-caller suggests that it is a human user and should do the "picking-up" task. However, it is the phone device that should react to the pick-up and any action taken afterwards is the responsibility of the phone. Therefore, the services related with the pick up operation should reside inside the phone and that is the class where the "pick-up" method belongs to. The naming and direction for the other methods also follow the same rule. The last message "7: hello" is actually not a message the software system is responsible for. It is merely placed in the diagram to complete the session concerning connection for telephone dialing for understandability.

A sequence diagram version of the interactions in Figure 4.16 is provided in Figure 4.17. Objects are listed at the top of the diagram and dotted lines are drawn down from the objects. Message links connect the dotted lines corresponding to objects and message numbers are omitted.

It is possible to group related messages that account for a sub-scenario and number them using a decimal point such as (2.1, 2.2, 2.3 …). Also, control

information can be demarcated by adding letters to the numbering to mean that those messages belong to the same block. Thickening the dotted lines into hollow rectangles in sequence diagrams can represent blocks. For example, a message can start an "if" block in the caller objects' dotted line that ends after the responses are received for both the "if" and its "else" sequences. Conditional message activations can be modeled by preceding a message with a "guard" statement enclosed in angled brackets. Once again, the idea is understandability. Low-level programming complexities are not desired with so many control structures in the interaction diagrams.

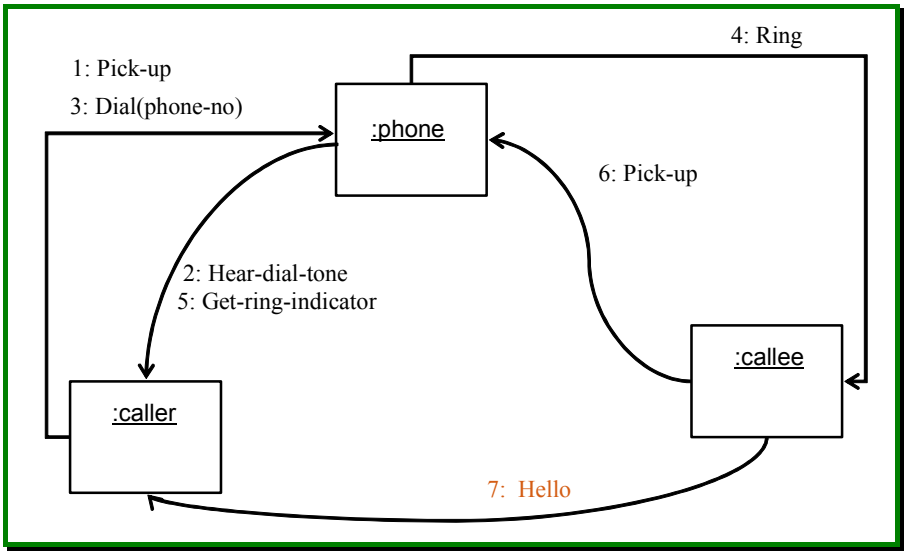


Figure 4.16. Collaboration diagram for a telephone connection scenario

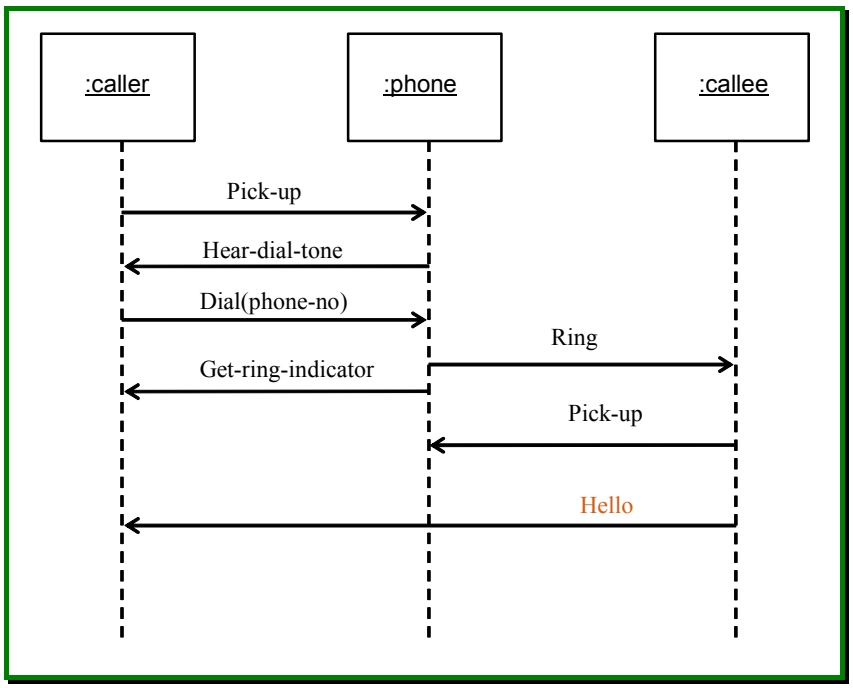


Figure 4.17. Sequence diagram for the telephone connection

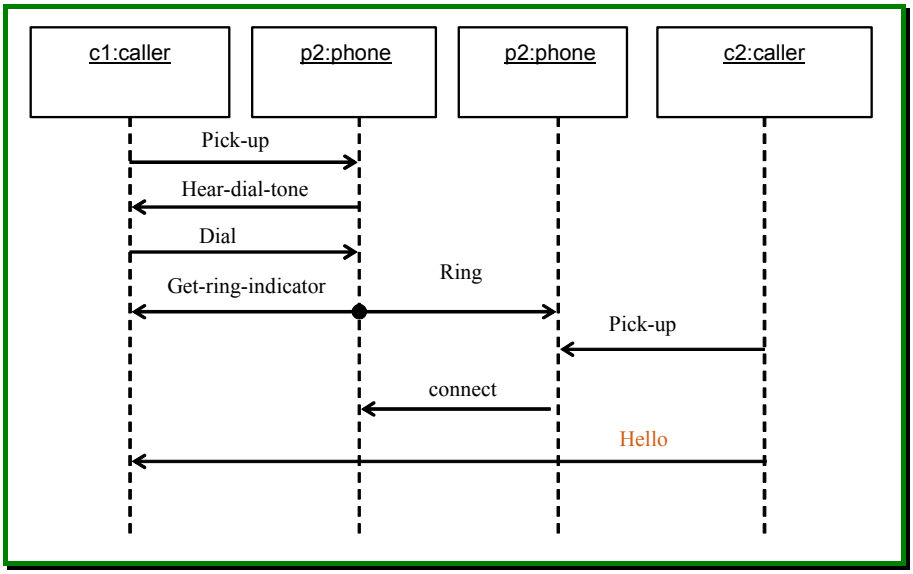


Figure 4.18. Using same class for different roles.

Design

Following up on the model developed during the requirements specification, design-level details can be introduced. In some cases the requirements level information may be revised. New classes and more members to existing classes will be introduced. Also new messages in the interaction models will be added. The detailed specification of the methods needs to be defined. This is the same as the procedural specifications done in the traditional approaches. Messages also may require synchronization specification. UML's message representation allows different kind of synchronization to be shown with different arrow shapes.

Besides procedural specifications, new techniques to be used in design are the structural tools and distribution of the components to nodes. The structure-related specification activities involve packaging different elements of the design into modules that are called packages or subsystems in UML. Initially UML only included packages that accounted for a logical grouping of other packages, classes, objects, and relations. Later subsystems introduced as derivations of packages are for physically partitioning the design components. Actually "components," packages, and subsystems reside in the "component diagrams" of UML. Such modules have dependency relations among themselves implying a "compile dependency" for the future implementation.

The modules should be defined with interfaces. Component units have their interface elements by default. If the environment does not provide specific interface facilities, the functionality can be provided by classes that publish public methods for external access, at any module they are meant for.

General modularity principles of coupling and cohesion should be considered in determining the scope of any structural unit such as a class, a subsystem, or a component.

Design stages

Depending on the methodology, various activities can be performed with different orders. The general approach suggested in this text starts with a top-level decomposition of the requirements model into candidate subsystems of the design model. Then comes the design of classes and objects. Finally message designs follow. Once a satisfactory level of a design model is achieved, scenarios can be enacted to verify the design model.

It is highly probable that some of the implementation will be assumed by readily available components. Most of the component technologies are OO development compliant and they can be represented in the design. If it is possible to minimize new code development, the design should start with this consideration. One option is to follow a bottom-up integration path starting

with the existing components as the leaves of a tree structure. Super components (or containers) can be formed by composing elementary components. This synthesis operation will continue until the whole system is reached by the final composition. There could be a one-level integration of all components into a system, or a hierarchical composition is possible. The intermediate levels could be a logical-design activity: some code writing is necessary but the intermediate components do not have to be implemented as physical components. If, at any level, component technology is to be followed, then the intermediate level modules also should be coded as components, complying with the underlying architecture that would suggest interface and connectivity conventions.

For a top-down decomposition, subsystems will be refined until they need to be implemented in terms of objects. The objects or a collection of them in terms of a subsystem could be substituted by component technologies. To maximize reuse, the top-down decomposition may have to be adjusted for accommodating the existing components so that minimal new code ends up being developed. In any case, OO media does not provide much support to guide the hierarchical structures.

Coding

The design model is converted to program codes to implement the system. Although developers desire to finalize all the parameters at design stage, the programmers have the final say. If the process does not employ hard measures for design compliance verification, care must be taken to have consistency between the design and the code. Anyway, OO makes it easier to link design to implementation. This section will illustrate the C++ examples [Muller 1997] for some of the basic models. CASE tools can create most of the “skeleton code”, assuming common usage of classes or so, for example providing constructors and destructors and assignment operators. Detailed implementation is left to the programmers.

A class is presented with its UML representation and C++ code in Figure 4.19. Default method declarations also take place. Figure 4.20 displays a similar case with one property defined. The C++ codes reflect the relevant segment; assignment methods are not included for brevity.

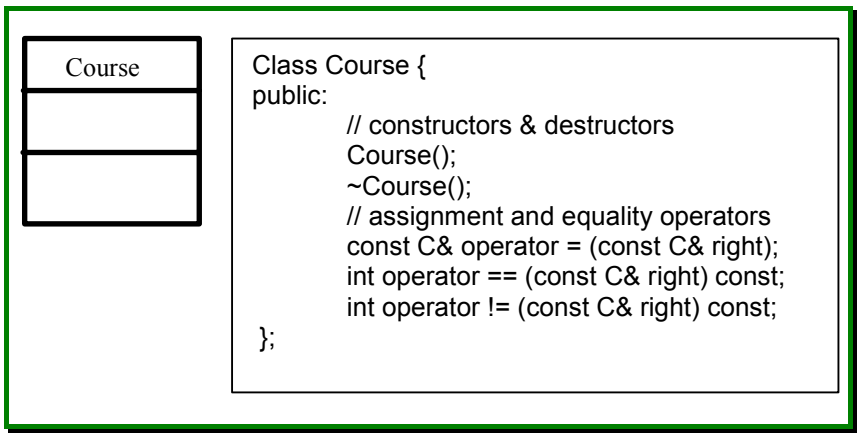


Figure 4.19. The simplest class in UML and corresponding C++ codes

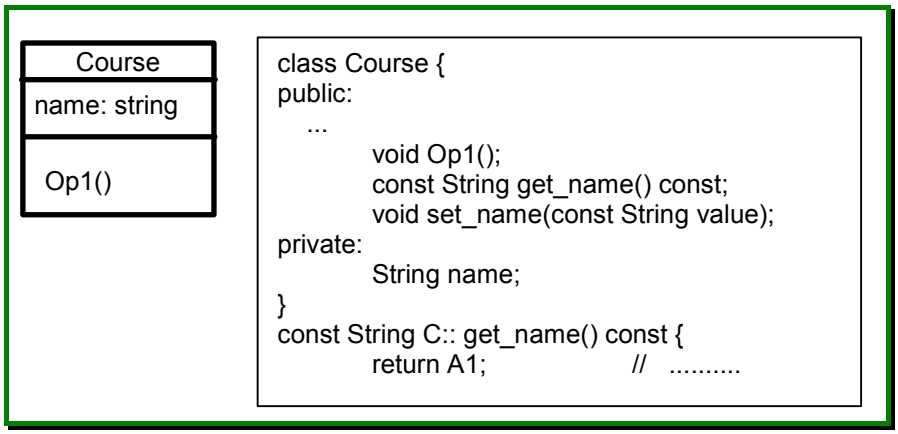


Figure 4.20. The UML class with a property

The trivial class definitions might seem intuitive. For more advanced features it is also not very difficult to devise coding mechanisms. However, it is better to follow the experienced practices and conventions that tools adhere to. The next feature is the associations in UML diagrams. Figure 4.21 depicts two classes with a relation. This relation is reflected in the code as mutual pointers in the participating classes.

Sometimes the association is more complex and a dedicated association class will be used to describe it. Figure 4.22 represents such an example. In this case, the original classes contain pointers to the association class that, in return, holds pointers back to the original classes. Again, default declarations are not shown; only the new items corresponding to the association class notion are included.

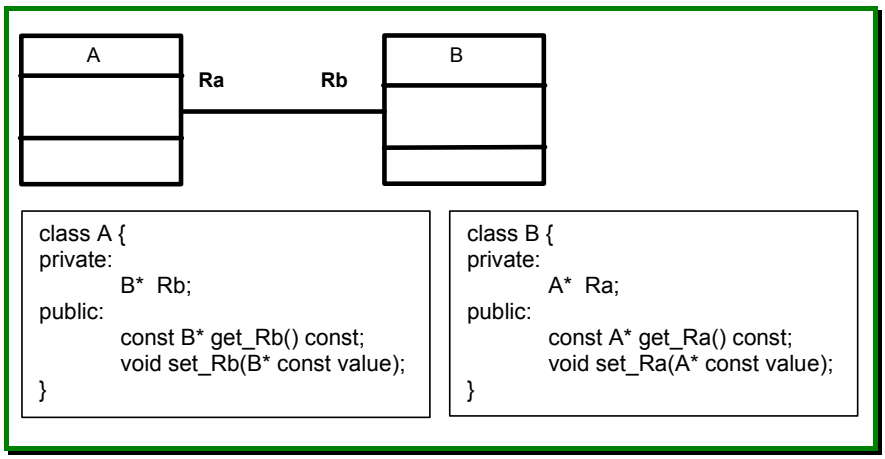


Figure 4.21. Association between two classes

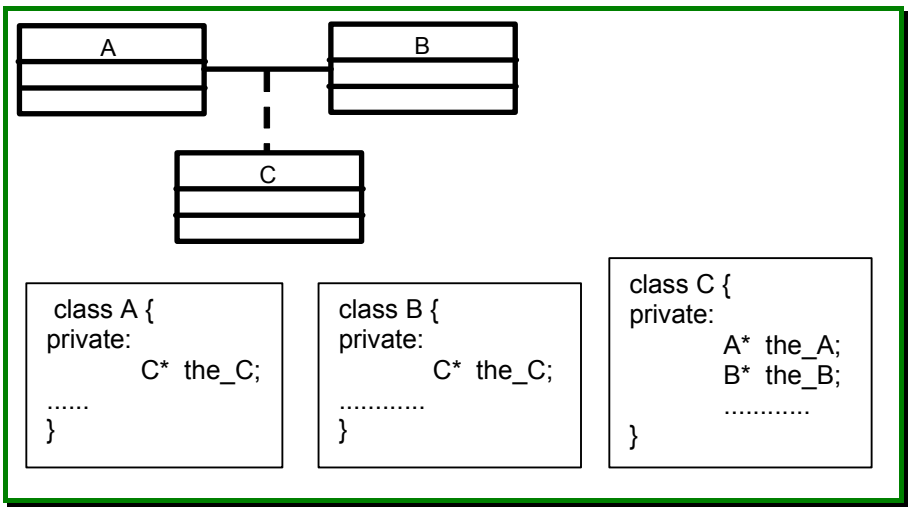


Figure 4.22. Association class

A composition relation can be named in addition to the default meaning on the diamond-end link. The other end of the line can have a role name that will be used in naming the pointer from the container class to the contained class. A reverse pointer is also provided in the contained class for access to the owner class. The name of this reverse pointer is simply utilizing the name of the container class. The looser version of composition that is the aggregation relation is also translated into a pair of pointers in both classes. Figure 4.23 depicts the translation of the composition relation to C++.

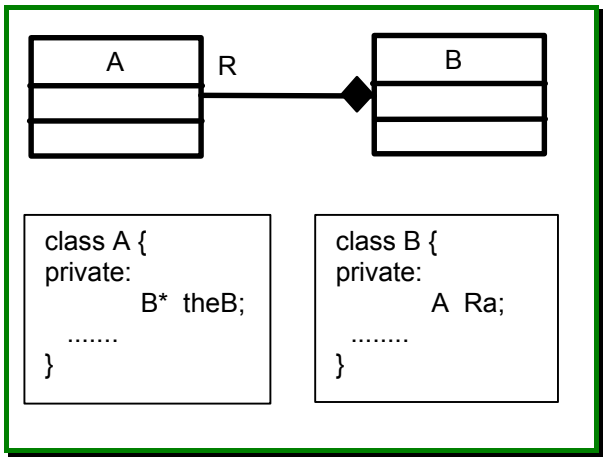


Figure 4.23. Composition in C++

Finally, an inheritance example is presented in Figure 4.24. The multiple inheritance from two classes is also capable of incorporating additional access security for the members. The class declaration comes with the inheritance definition:

```
class B : public A2, public A1
```

where B is inheriting from A2 and A1. The "public" keywords mean that the protection levels should be preserved for the members that are inherited. A different keyword such as "private" or "protected" would only bind extra access limitations: any member with a looser access limitation has to be raised to the security level of the keyword. This means that the members in the base class can never be opened for easier access but they can be limited for stricter access control, after inheritance by the derived class.

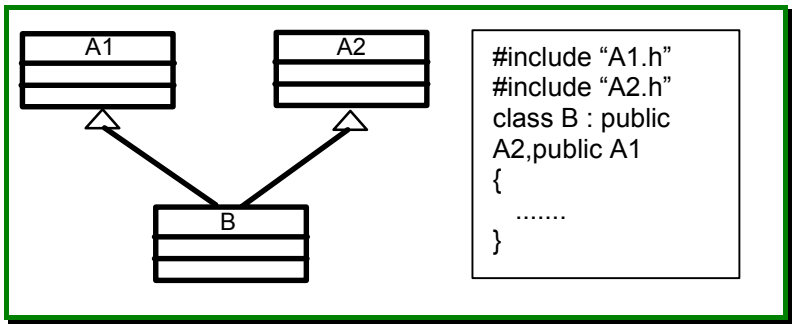


Figure 4.24. Inheritance in C++

It should also be noted that usually one class is dedicated a file (.h and .c files). Sometimes closely connected classes take place in one file. In Java codes, since there are no pointers, variables are used instead of pointers. The

coding examples can easily be interpreted for Java language with this principle in mind.

Summary

Object orientation has been introduced as an appropriate move in the software engineering discipline. The models correspond to real world entities more than the traditional era models offered. The idea is moving towards component-based development also.

UML has been observed as the prominent modeling language. A suggested approach is followed for ordering the tasks in a lifecycle through OO techniques. Use case diagrams are for capturing the customer's requirements. They need to be supported by the interaction diagrams for the detailed modeling of the execution. Class diagrams also are required to provide the players in the interaction game. Use case diagrams represent system capabilities where each use case for a system function is explained by an interaction diagram. If temporal management is necessary, state machine based modeling can also be used. Each class is capable of containing a state machine that is modeled through a State Chart Diagram in UML.

Design for the objects start with refining the requirements model. Implementation oriented details are further supplied to the class and interaction diagrams. Structural organization of the OO elements is achieved by grouping in chunks such as packages. Decomposition of the system is important to utilize the readily available components. Message specifications follow, including synchronization modes. Finally, objects and their relations are converted to programming language statements to illustrate how the concepts are applied. Most of the relations translate to mutual pointers in both of the participating classes.

Questions

1. Object Orientation is presented as a positive move in the software history. Can you point out some weaknesses or drawbacks?
2. How is the Structure dimension supported in OO models (assuming the elementary design dimensions as Data, Function, and Structure)?
3. Discuss the Object Orientedness of use case diagrams, sequence diagrams, and state charts.
4. Compare the top-down and bottom-up approaches in determining class hierarchies. State the advantages and disadvantages of both.
5. Describe and compare the notions: overloading, overriding, and polymorphism.

6. Give a classification example where multiple inheritance is appropriate.
7. Draw a class diagram that includes an "association class".
8. Assume a relational database will be used. How would you save the objects – ignoring their methods – in the database? In other words, what features in OO models can relate to database tables, relations, and fields?
9. If you were to develop code without an object oriented compiler, how would you approximate the OO logic using a procedural language?
10. Define a problem of your own and develop the OO requirements model utilizing use case, class, and interaction diagrams.

References

Booch 1994	G. Booch, *Object-Oriented Analysis and Design*, 2nd Edition, Benjamin Cummins, 1994.
Booch et al. 1999	Grady Booch, James Rambaugh, Ivar Jacobson, 1999, *The Unified Modeling Language User Guide*, Addison-Wesley.
Coad and Yourdon 1991	P. Coad and E. Yourdon, *Object Oriented Analysis*, 2nd edition, Prentice-Hall, 1991.
Coleman et al. 1994	D. Coleman et al., *Object-Oriented Development: The Fusion Method*, Prentice-Hall, 1994.
Jacobson, 1992	I. Jacobson, *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
Muller 1997	P-A. Muller, *Instant UML*, Wrox Press, Birmingham, Canada, 1977.
Pressman 1997	R.S. Pressman, *Software Engineering: A Practitioner's Approach*, 4th Edition, McGraw Hill, 1997.
Rambaugh et al. 1991	J. Rambaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

Chapter 5: Introduction to Domain Oriented System Development

Introduction

Since the beginning of the computational era, there has been a variety of approaches in an effort to provide better engineering solutions to software development. Maturing out of the early hardware-dependent and limited conceptual foundations, software engineers have been productive in developing methods during the definition of their new discipline. Discovered relatively late, the most striking problem surfaced to be the huge complexity of the average system demand in the market. Linear improvements in production tools –conceptual or practical– would not do the trick: the discipline was requiring improvement in orders of magnitude to satisfy its ever-growing quest for development efficiency.

Such ambitious demand required intellectual control over the problem, which is only possible after the digestion of higher-level abstractions by the developers. The semantic gap was diminishing between the cognition of a problem in the brain and its formulation for the machine, i.e. the computer, as a result of the new ideas being applied to the field. Some examples to such novelties adding to our leverage are Functional Programming, Object Orientation, and abstraction mechanisms in design models. The struggle to solve the main problem at higher-levels of abstraction [Tanik and Chan 1991] was proper and well accepted. It was not sufficient though. Gigantic sizes of the systems had to be matched by production speeds in an ever-hungry market.

One persuasion is clearing itself out of the fog of uncertainties – it is not feasible to create tens of millions of lines of code, one line at a time. Luckily a wide spectrum of functionalities had been coded somewhere by someone. A new system should be built by locating and integration of such islands of code. It is not a coincidence that an array of new technologies and ideas all relate to the integration concept. Domain Analysis is one such idea, and so are component-based technologies, design patterns [Gamma et al. 1995], Common Off the Shelf Components (COTS), frameworks [Fayad 2000], and software architectures [Gamma et al. 1995].

A domain can be defined as a set of current and future applications that share common attributes. Domain analysis is actually a set of activities for defining domains. Existing applications and their development processes are studied to identify and represent relevant information. Domain experts,

underlying theory, and developing technologies are also good resources for accumulating such knowledge.

Domain orientation primarily lends itself to the understanding that the experience and reusable structures in developing similar systems can systematically be managed. With similar meaning in the same domain, experts accumulate knowledge over a family of similar applications. To engineer this experience, a methodology sets forth the mechanisms to capture and represent knowledge plus a mechanism to introduce building blocks for the system builders, in a particular domain. Further, utilization of the captured experience and the building blocks (components [Szyperski 1998]) in an effort to instantiate them to form products is the continuation of the methodology. As shown in Figure 5.1, the two separate cycles, defining a domain and utilizing the domain, constitute the foundation of Domain Oriented software development. Early research that prepared the Domain Orientation can be found in a variety of work such as in [Arrango 1994, Diaz 1987, Holibaugh 1993, Itoh et al. 1998, Kang et al. 1990, Neighbors 1989, Simos 1996, SPC 1990].

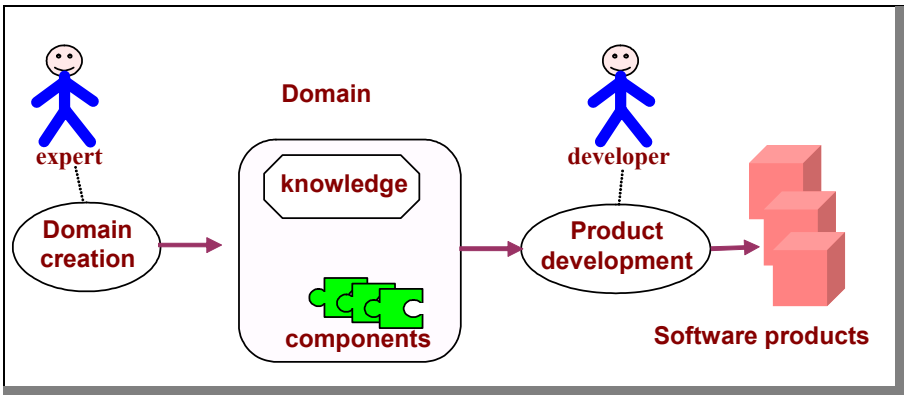


Figure 5.1. Domain Oriented Software Development

Domain Analysis in the developing Perspective

Besides the "build by integration" paradigm, the Domain Analysis (DA) activity serves the cause to cope with complexity, utilizing another understanding: separation of concerns by offering "separate domains." Both concepts agree with the most natural approach for complex tasks – divide and conquer. Introduction of software components as the building blocks for integration should be followed by a categorization of the components in related fields. Better stated, component categories (Domains) should first be defined, and then populated with components. This has been the actual trend.

Early attempts to provide solutions across domains in a single development environment have proven inefficient. An example is the pioneer structured language, PL1. Soon after its introduction, tools to address different concerns were being appended. Database query statements were now built into PL1. Arriving later, Pascal, as a pure procedural language, enjoyed common acceptance while PL1 suffered lack of interest. An early research work addressing the need to manage different domains in an orchestrated separation is Design Abstract Requirements Manipulation Shell (DARMS). Developed by Christiansen [1989] during his PhD study, DARMS facilitated two categories of developers as Domain definers and Domain users. The first category of developers can also be referred to as Domain Experts or component developers. They provide the environment for any specific domain by supplying a set of software units. As a matter of fact, the domain-supplying developer category can contain more than one set of experts, actually one set per domain. The latter category comprises the build by integration practitioners. They locate and integrate components into systems and make the system work. This task will benefit from employing domain-experienced developers, but not to the extent the domain definition task would.

Figure 5.2 displays the relation of the technologies with respect to the two dimensions that are instrumental in analyzing the divide and conquer benefits. The dimensions are the separation of domains and the granularity of the code. The technologies are the monolithic systems that are built in discretion and components that are developed with the intention of integration in different systems.

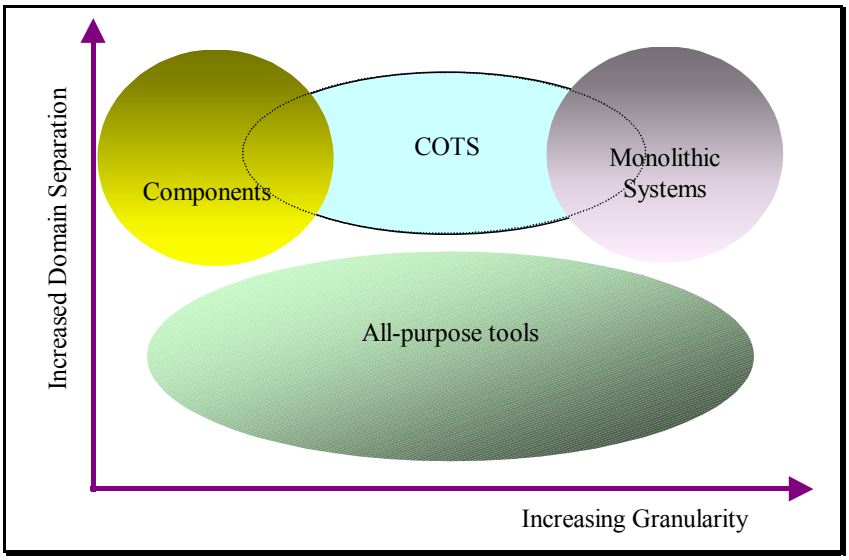


Figure 5.2. Domain and size based separation

The COTS concept was introduced earlier than the component technologies, generally targeting large-grained modules. Grain size in component technologies is an important attribute, and components do not have to be small-grained. Figure 5.2 assumes the common understanding of the industry as of today.

In isolation, the Domain Analysis activity can be utilized for either component based or monolithic development. As a natural player for the Domain Oriented Software Development, Domain Analysis provides the basis for creating components. As the mentioned DARMS research suggested, domain specific development can better be achieved utilizing the component technologies. Recent research indicates that the trend in the domain specific approaches is towards supporting "component concepts", even if not yet supporting the "component technologies" as they are practiced today.

Justification for Domain Specific Development

Reuse is no doubt a mechanism for efficiency. New technologies have also been investigated within the reuse paradigm as a critical criterion. Object-Oriented modeling promotes reuse mostly through the inheritance ability. Before any infrastructure establishments such as object orientation or component architectures, reuse was attained through constructing software libraries. This is only to mention code reuse. Today, component technologies are regarded as the utmost reusable modules. Components are produced for reuse, more than any other concern.

Domain is the context for which a component gains its meaning. It is not realistic to expect that a limited set of components should be suitable for

solving all kinds of problems. The other extreme is providing a limitless number of components to address the whole software problem world, without domain separation. There is no need to discuss the inappropriateness of this approach. A given problem is almost always that of a domain. If a customer needs a system to do everything (like PL1 attempted to code in every domain), different systems for discrete domains can artificially be integrated – again suggesting that one domain-specific system should be developed at a time, before integration. Sounds like recursion? Yes, and no problem about it also. One person's system is another person's component. There is no reservation about further organizing a domain in component categories of different abstraction levels. So a domain can further be divided into sub domains – vertically or horizontally. Figure 5.3 depicts the partition of a domain. Nevertheless, a domain is needed for production, so lowest level (implementation) constructs have to be accommodated.

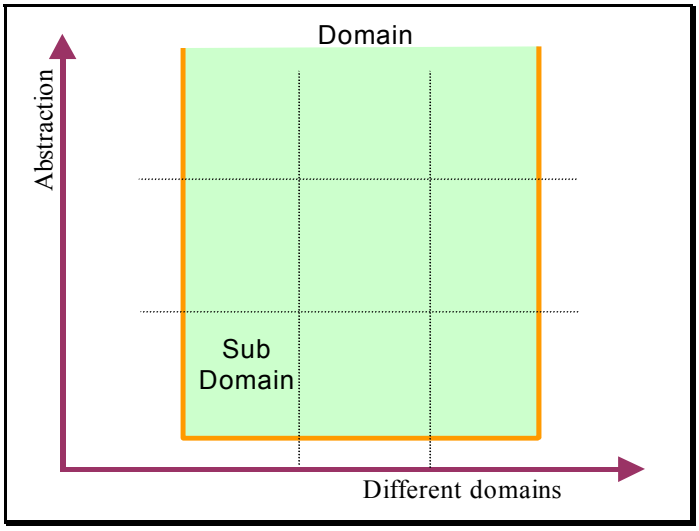


Figure 5.3. Partitioning a domain

Reuse can be enabled by constraining development within a domain. Efficiency to locate a component depends on how well the experts know the component library. Familiarity with the domain library is easier with a smaller library. Natural limitations of the human mind that make it difficult to concentrate on too many objects at once have resulted in the "divide and conquer" approach. In mature domains, developers know the components by name, which immediately signifies the meaning, the capabilities, and other specifications of a component. A good example to matured domains is the digital circuits field. An experienced designer knows how many "and gates" are in a specific chip, when told its name, and further how the pins are laid out. Established standards are an indication of the maturity of a component domain. Figure 5.4 depicts the layout of a component in a digital circuit

domain that is known as the Transistor-Transistor Logic (TTL). Here, the components are chips that contain a variety of gates and higher-order units that are laid out within some standard patterns. This example is of a collection of four "NAND" gates. The numeric code that identifies this chip is 7400 – this is the first component in the so called 74 series. This nomenclature implies the electrical characteristic of the family of components, corresponding to a standard. Also the connection pattern is another standard that applies to many of the chips in the 74 series of TTL components: Most of the "quadruple 4-input" gates have their input/output pins connected in the same pattern as the example depicts in Figure 5.4.

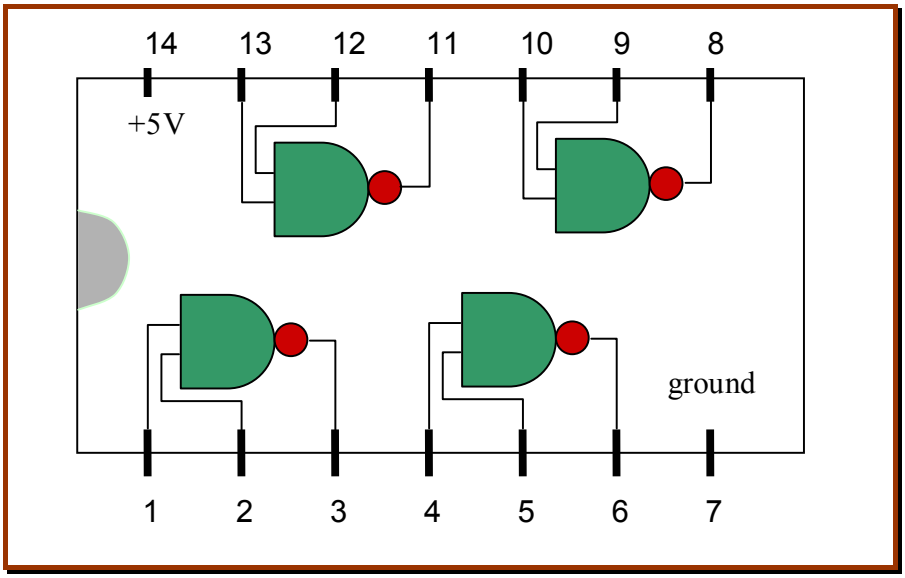


Figure 5.4. The quadruple 4-input NAND gate

Soon after its introduction, variations on some "quality factors" were available in the form of faster (74S series) or low-power (74L series) versions. Some of the chips allow a limited modification capability by allowing some mode setting pins to be connected to ground for one mode, or to V_{cc} (5 Volts) for another.

Using the TTL families of digital circuit components, a designer is freed from lower-level development for example to construct a gate. Also, the electrical and speed characteristics are guaranteed within specified limits. As a result, designing bigger systems is faster and more reliable. Some efficiency is lost, however: If only one AND gate is needed, a chip that contains four has to be bought and connected, with three gates wasted (unused). On the other hand this "componentization" saved so much that, in general, such inefficiencies are not even felt. Also, the technology became so inexpensive that the waste of a handful of gates ca be disregarded.

The DARMS environment actually provided means for creating different domain infrastructures on a single tool. By allowing different experts to create different sets of objects and providing wiring-level automation, the tool would provide integration for objects in any domain. Figure 5.5 depicts the multiple domain ability of the DARMS tool. The tool's capability to accommodate multiple domains should not be taken as its deficiency in supporting specifics of different domains. Such special capabilities are defined by the experts, while defining the domain. Once different domains are defined, DARMS actually offers different environments per domain rather than behaving like a single tool aiding all concerns.

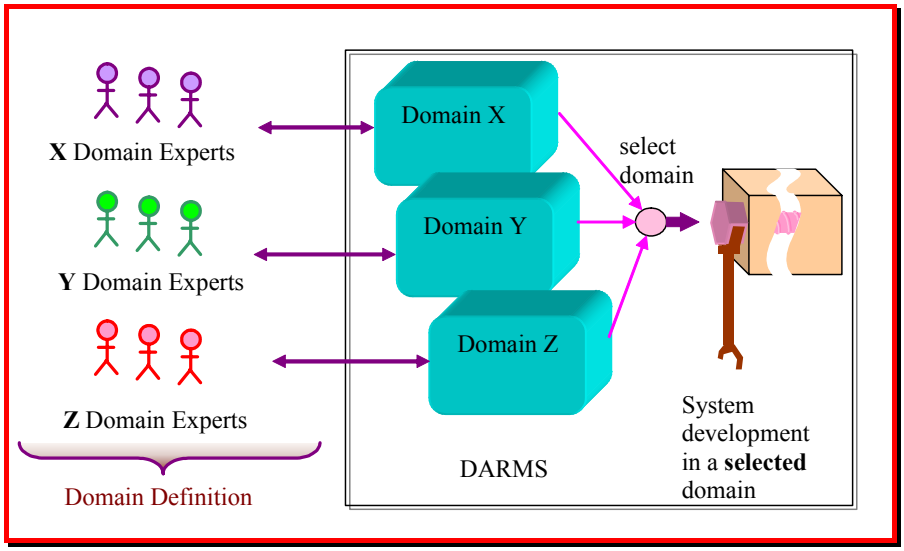


Figure 5.5. DARMS as a multi-domain tool

Partitioning the problem space into domains results in limiting the number of objects a developer has to study. It is easier for a developer to know a limited library than to know every component on earth. Also the experience and know-how about the usage of the components are very valuable, and time consuming assets are more effective in a limited domain.

The pattern is not only applicable to humans; artificially intelligent machines also are more efficient when the breadth of the required knowledge is limited in scope. Automatic code generators have not replaced programmers and designers: they cannot do everything. For more specific usage (in narrower domains), however, they have been more successfully utilized. The visual code generating environments (Visual Basic, Visual Café, Visual C++, JBuilder, Delphi, JDeveloper, BDK, Power Builder, etc.) are indispensable tools for today's programmers. They are good in generating a Graphical User Interface (GUI) and almost nothing else, automatically. A more domain-specific example is a tool that would accept specifications

graphically and produce code for a complete application. The catch is, the application had to be of a medium-size business automation type. MicroStep is the name of such a tool was introduced two decades ago.

One explanation of the success in limited domains is the accumulation of default information that is automatically installed without a need for specification. A domain can be characterized by a set of similar applications. Once similarity is inherent, some default detailing can be assumed. A knowledge base is produced as a domain matures. Within this perspective, maturity relates to know-how for "filling in the blanks" besides relating to familiarity with the component set.

There is no free Reuse

Domains promote reuse and ease for software development if locating and integration of pre-developed software components is convenient. To be able to solve all kinds of problems optimally in the domain, the number of required components is theoretically infinitive. This requirement contradicts our condition for the usability of a domain. It is not easy to locate the desired one among a large number of components. Also it is not possible to be familiar with all the components if there are so many of them. The solution is to introduce modification capability for the limited set of components.

With an analogy to language theory [Hopcroft and Ullman 1979], a domain is a grammar and the products are language statements. A grammar is capable of producing infinite number of statements just as there could be infinite number of systems a domain environment could produce. A grammar uses a finite set for input as an alphabet and a domain accommodates a finite set of components. Rules to generate expressions in a grammar correspond to rules to integrate components. Also, some rules should guide their modification. These production rules can be interpreted as a methodology for a domain-oriented component integration. Rules can also be treated by a tool as specification in automatically guiding the development activity. To complete the analogy with a regular grammar in the expression of a software development domain, automata theory can be utilized. A finite state automaton can accept expressions generated by a regular grammar. Actually, system under development is the acceptor of the production. Equation 5.1 displays a five-tuple consisting of a set of states, an input alphabet, a transition function, an initial state, and a set of final states to represent Domain Oriented development. Table 5.1 provides the explanations for the symbols in Equation 5.1, in terms of automata theory and its interpretation for the software domain modeling.

$$D = (S,\Sigma,\delta,S_0,F) \qquad 5.1$$

Table 5.1. Software development domain interpretation of Finite State Automata

Symbol	Meaning	Interpretation
S	Set of states	Set of development states
Σ	Input alphabet	Components
δ	Transition function	Specification action for any refinement
S_0	Initial state	Request for a product
F	Set of final states	Set of final specification states: where an observable status of the product is achieved.

$$S = \{S_0, .. S_i, .. S_n\} \quad 5.2$$

$$S_j = \delta(S_i , a_j) \quad 5.3$$

where $S_i,\ S_j \in S$ *and* $a_j \in \Sigma$; $\Sigma = \{ a_i, a_j .. a_n \}$

The general constraints on reuse need to be traded off. For a component, the smaller the size, the greater is the chance of reuse. But the amount of reuse is low. For a large component, the amount of reuse is high, once the component is utilized. But the chance to use a large component is low. A large component will be conducting a lot of functionality, all of which exactly may not be easily contained in a demand, unless some modification is done.

Developing the same functionality is about five times more expensive if it was designed for reuse. Most of the additional cost of reuse is in the domain definition. Components are developed for reuse and they are more complex than simple functions. Some visual environments started with the non-component versions of the GUI objects, which were later converted to components by adding a 'wrapper' code around them. However, in an established domain-oriented development medium, it is expected that the expensive preparation of the domain will pay-off by repetitive productions. Generating systems will be less expensive, in addition to other benefits when compared to non-domain-orientation. Figure 5.6 depicts the contradicting dimensions of reuse potentials.

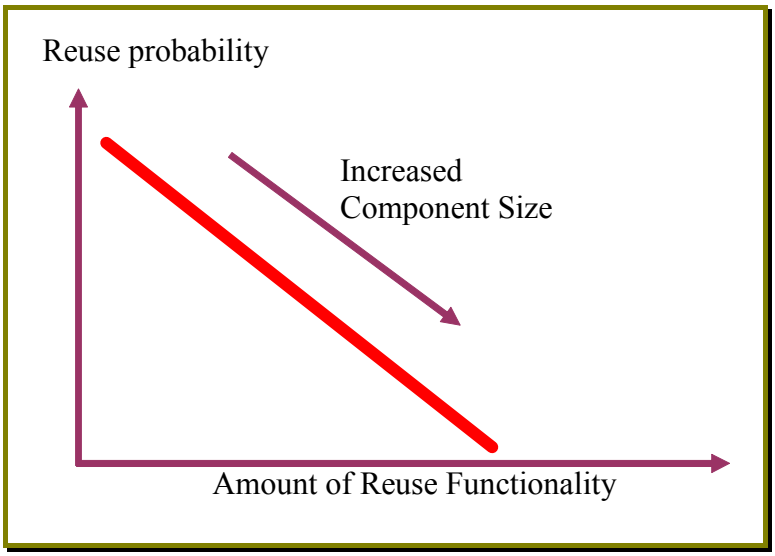


Figure 5.6. The probability and amount of reuse

As a final word, although it is possible to define a domain only at abstraction levels, or even at lower levels but without component technologies, modern domain technologies are equipped with the component concept. This is besides the fact that the produced objects are not always required to comply with an established component protocol.

The Domain Oriented Process

There are few approaches to Domain Oriented Software Development. A commonality that is observed across those approaches is the two kinds of basic activities: Definition of the domain and development of domain specific software. Both activities are further detailed to methodological levels. The domain environment is shown in Figure 5.7 with the separate kinds of users.

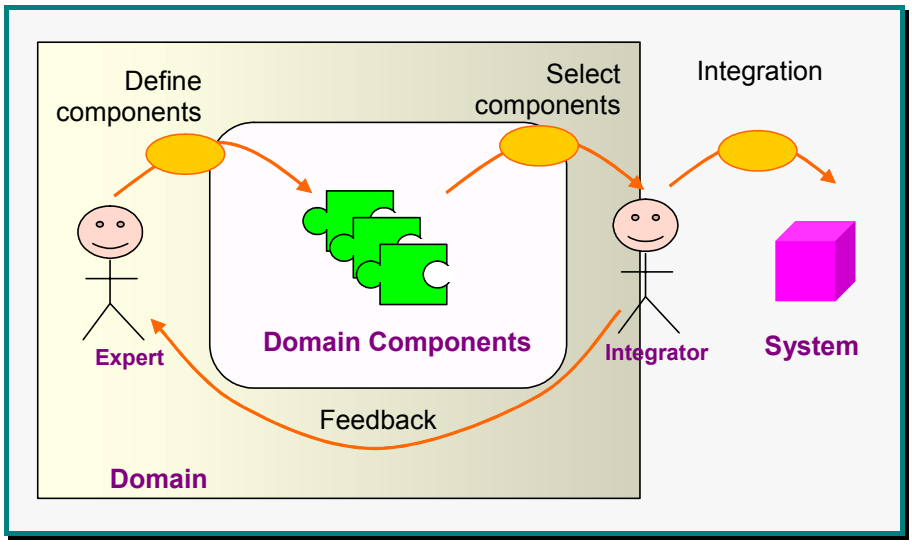


Figure 5.7. Domain usage

Definition of a Domain

The Definition of a domain is a complex activity. Similar to the development of software, there are methodologies addressing various stages of activities producing a domain model. This is like defining a family of software products, rather than developing only one. Actually, an analysis is conducted for the commonalities that can be found in the possible variety of applications for the domain. This analysis results in sets of primitives to be used in constructing the components and in the behavioral description of the domain aspects.

Domain analysis uses a set of existing or to-be-built applications and human experts as sources of information. Commonalities are derived from the applications and abstracted to form a basis for the domain structural elements. Also other important knowledge elements such as a terminology dictionary, a list of features to parameterize the domain knowledge, and higher-level rules as policies and patterns of usages are gathered from the human experts. At this phase, the abstracted capabilities and system-level functions common to the applications are also catalogued as entities. Relation patterns among those entities are recorded.

A generalization can be made for the expectations from the Domain Analysis activity. The goals of Domain Analysis can be stated as follows:

1 Produce reusable software objects

2 Produce reusable requirements

3 Define reusable software architecture

4 Design reusable code

5 Define reusable structures

Once all the precious information is extracted, next comes the problem of representing them. Knowledge representation is a problem of its own with peculiar difficulties. Here, the emphasis is in the ease for access and applicability to new problems. Features and abstract structures will be manipulated in the construction of new systems. Graphical models have proven to be useful in representation. Actually, this step is for the modeling of the domain after its analysis.

There is also some room for implementation. Besides the required implemented pieces in the form of components, some architectural structures may be handy if they are present. In a domain, there would be typical applications of recurring patterns. The connectivity for a pattern composed of components, will be provided by a coded facility to be replicated per system.

The knowledge gathered during domain analysis is used also for the integration of components. Such knowledge is classified into various structures by different methodologies. Some integration information is inherent in behavioral descriptions such as 'collaboration.' Manipulated as interaction diagrams in UML [Booch et al. 1999], collaboration among a party of components can be explained in a scenario. A more precise modeling is an ordered set of messages across the participating components. Since the messages form the connections among the components, the collaboration concept is a blueprint for the integration among the components participating in a scenario.

Some of the approaches to Domain Definition are:

- Feature Oriented Domain Analysis (FODA),
- Domain Analysis, Modeling and Engineering (D-AME), and
- Joint Task Force Object Oriented Domain Analysis (JODA).

FODA process for Domain Analysis consists of three steps:

1 Context Analysis,

2 Domain Modeling, and

3 Architecture Modeling.

D-AME suggests processes that contribute to its abbreviated name:

1. Domain Analysis,
2. Domain Modeling,
3. Domain Engineering, and
4. Utilizing the domain for the phases of the software lifecycle.

SPC defines different activities:

1 Domain Description (conceptual taxonomy),

2 Domain Qualification (feasibility analysis),

3 Knowledge base (domain knowledge), and

4 Canonical requirements (Reusable requirements).

These approaches are mentioned to present an idea about the existence of domain based engineering methodologies. Included activities can be explained in further detail. Interested readers can be directed to the references for lengthy descriptions of prescriptive development processes.

The original Domain Analysis techniques did not include provisions for extending to Web services. The idea was to provide a self-contained framework for "instantiating" applications. However, the philosophy is so close to the contemporary development towards achieving similar goals using the Internet. With an analogy to Feature Models, "ontologies" are used as a semantic description of the modules that populate a domain. Equipped with such definition tools, directory services such as UDDI help the designers to locate components. Once found, such components – implemented as Web services – can be accessed through messages that must comply with related standards. It is possible to extend the philosophy of FODA to the Internet for locating and integrating modules for generating applications. Figure 5.8 depicts the Internet enabled version of locating and integrating components.

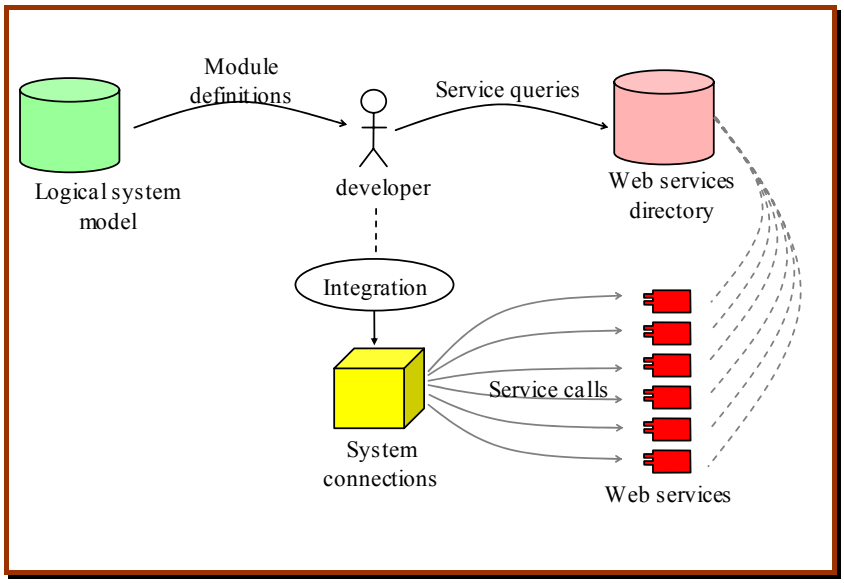


Figure 5.8. Locating web services and connecting them to the application

Although the Web services related work seems disconnected with the domain analysis approaches, there are commonalities. Both approaches try to aid development of systems through pre-existing artifacts. There is some research towards joining the advantages of both fields.

Exploiting the Domain

The sole reason in the engineering of a domain is to use it in the generation of software products. Besides the expensive nature of reusability, an engineer should also expect to approach it with care. Perhaps the domain definition is regarded as a more important step. Still there are still difficulties that need to be overcome, before fulfilling the primary objective of synthesizing the product.

Component based synthesis is a natural task for domain orientation. Yet no established component-oriented methodology is available. Even if we were able to map the infrastructure in the domain to a component-based design, such a design is lacking guidance. To make matters worse, we also lack the mapping from the domain to any system synthesis approach. Actually a component-oriented synthesis methodology is required to enable the domain-oriented construction. The following sections discuss component-oriented software development and the process of utilizing the domain environment for such development.

FODA

Some time passed after the introduction of the Domain Engineering approaches. With the increasing interest toward composing systems out of

existing Web services and components, supporting approaches surfaced as worthy to be mentioned. FODA [Kang et al. 1990] is an approach that developed in the mentioned direction. The feature model that is the central element in domain definition is now utilized for capturing the requirements and for indirectly defining a solution. A feature is a user-visible aspect of a software system. Feature Oriented Reuse Model (FORM) [Kang et al. 1998] was later introduced and is based on FODA.

The feature model is basically an AND/OR tree. Higher-level capabilities of systems in a domain take place in the higher levels of this tree. The branches representing the AND connections imply the mandatory selection of all the children of the parent feature. OR branches imply optional selection of the children features. A feature model also includes constraints that require the selection or omitting of a feature at (otherwise unrelated) location on the tree once a specific feature is selected. Feature models have attracted attention of researchers who proposed enhancements. For example, a constraint can be imposed such as a range of selections. In this case a minimum number and a maximum number (both specified separately) of children features are required [Riebisch 2003]. Figure 5.9 presents a feature model for the personal computers domain.

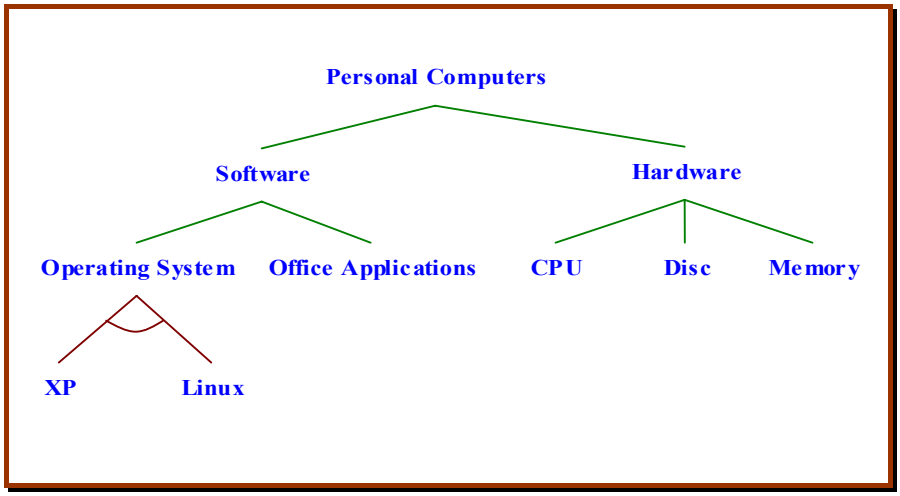


Figure 5.9. Feature Model for personal computer (PC) development

Forming relations among the actors, activities, and the infrastructure will be instrumental for explaining domain engineering. Figure 5.10 depicts the fundamental roles that take part in the definition and the utilization of a FODA environment.

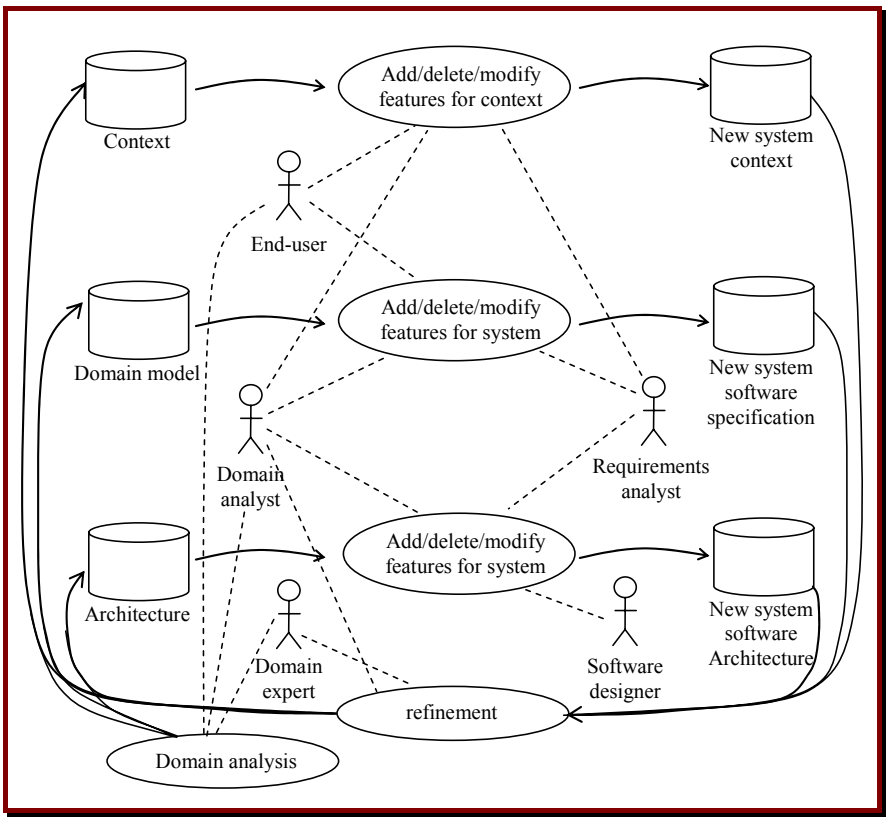


Figure 5.10. Defining and using the FODA framework

To have a better idea about the contents of various repositories, the modeling formalisms used in the different phases of the analysis process are listed in Table 5.2. Most of the techniques used in representing the models have existed before FODA.

Table 5.2. Models used in the FODA Framework

Context Model	Domain Model	Architecture Model
Structure Diagram	Entity relationship model	Process interaction model
Context Diagram	Feature Model	Structure chart
	Functional Model	
	Domain terminology dictionary	

FORM

The most powerful aspect of FORM is its intention to link the features almost directly to components. The domain should be so well defined that selecting

features will aid in selecting the architectures and eventually the components. FORM suggests a four-layer structure for organizing features. On the artifact side the constituents are also organized as layers.

The feature model is basically an AND/OR tree. Higher-level capabilities of systems in a domain take place in the higher levels of this tree. The branches representing the AND connections imply the mandatory selection of all the children of the parent feature. OR branches imply optional selection of the children features. The feature model is equipped with more capabilities. There are also constraints, suggesting the selection or omitting of a feature anywhere in the tree once a feature is selected and it takes part in this constraint definition. The feature model has attracted some attention of other researchers, and enhancements were proposed. For example, a constraint can be imposed such as a range of branches for selection. This means that there is a minimum number and a maximum number (both specified separately) of children features that are needed to be selected out of a given set.

Figure 5.11 depicts the Domain Engineering process for FORM. Figure 5.12 displays the application engineering corresponding to the defined domain.

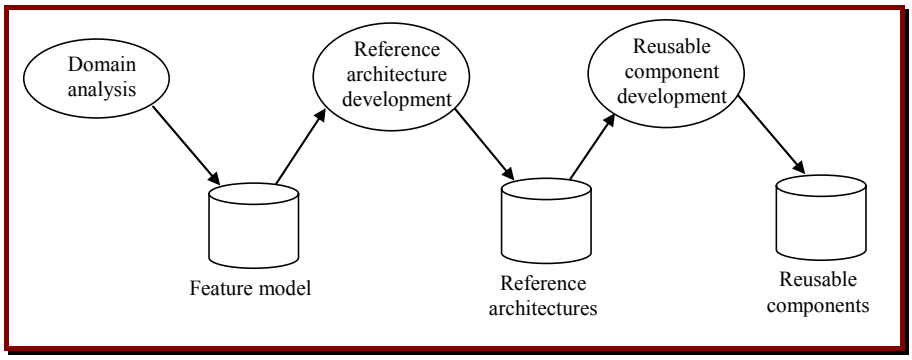


Figure 5.11. FORM Domain Engineering

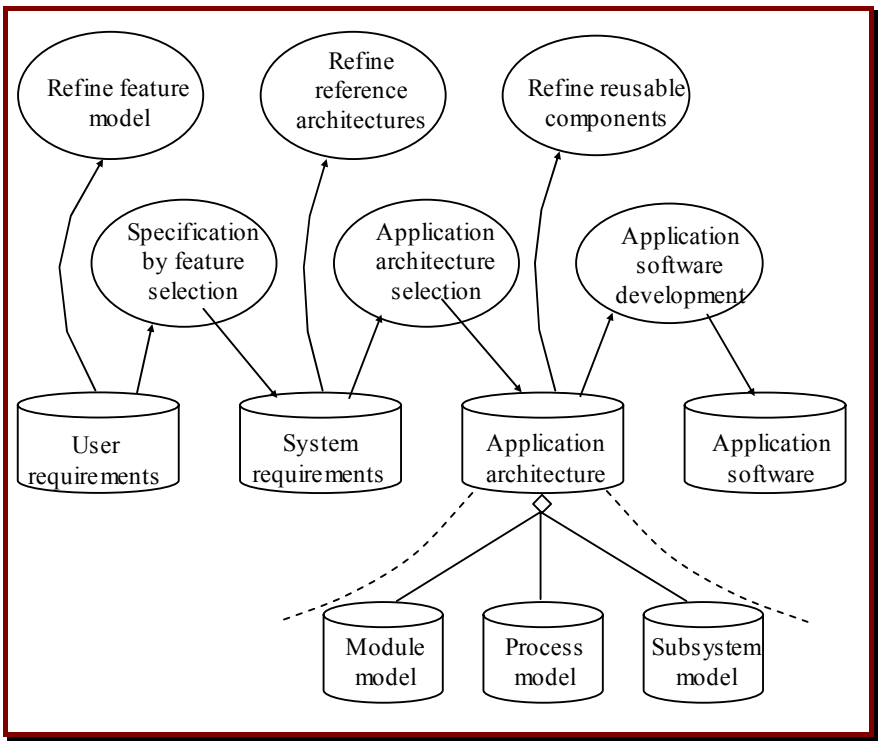


Figure 5.12. FORM Application Engineering

So far the location and adaptation of components were not discussed much because in an established domain framework, most of the components are expected to be well defined. However, as in the case of Web services or in general, in component search and integration, more intelligent operations will be needed. Locating a component, and then integrating it, will be more difficult. Linguistic and intelligent support may be necessary for especially automatically locating a component. After finding the module, the wiring-level mechanism has to be matched so that correct functions are called in the correct manner. This operation may require automated "adaptor" generations.

Component Oriented Design

Object Orientation blended itself into component-based models. This process took place due to two reasons: Object Oriented approaches were the current fashion in software development and they had to be the host for the newly developing component technologies. In other words, other alternatives were either too old or not invented yet. Also, the close resemblance in the static look of an object and a component definition made it easy for the Object Oriented models to accommodate components with minor modifications.

One important notion was overlooked because of this natural trend: component technologies needed to be utilized further for a paradigm shift they were silently enabling. Build-by-integration notion could be realized, if a component-oriented rather than component-based path were to be devised. It is relatively easy to represent components in an object-oriented medium, but this does not gain us much. The fundamental concept to structurally decompose a system in abstraction for logical modeling, (and corresponding physical decomposition of code for detailed design) to exploit the executable product nature of components had to be formulated.

The industry has produced components and we are given problems to be solved as different integrations of those components. A methodology is needed as guidance for such project tasks. Rather than concentrating in the internals of the components, which are already constructed, the approach should concentrate on representing a complex system as a network of these components. The network is rather a hierarchical one, to keep the complexity under control with respect to the human design psychology as Herb Simon suggested [1969]. Figure 5.13 depicts the transformation from a system's definition to its realization. The hierarchically organized abstract decomposition of the system is finally matched with existing components. Connections shown among the nodes of the system correspond to structural relations. Operationally important connection among the components is based on message paths rather than part/whole relations defining structural relations. The message connections are not shown in Figure 5.13.

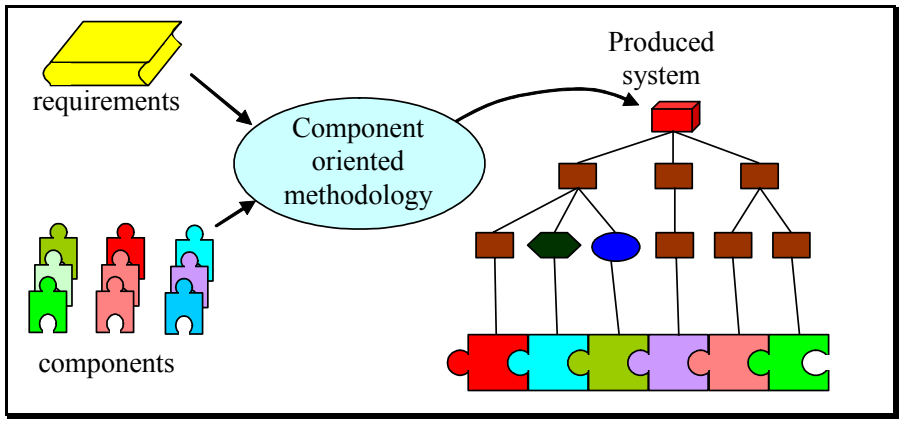


Figure 5.13. Transforming requirements to a system

Expectations from a Component Oriented approach

Component orientation is enabled by the component technologies. Long after the establishment of component-based design ideas in hard engineering disciplines, similar leverage was being speculated in the software world. Later, the technology is formally founded by the introduction of component

protocols [Szyperski 1999]. Otherwise, the soft nature of our discipline encourages the diversification among the methods used in different component implementations – thus disabling the essence of component domains. The last sought capability will be the compliance with existing component protocols so that the environments support final implementation activities, which may be done automatically.

An initial requirement, however, is the modeling capabilities that provide abstraction mechanisms. A system is desired to be composed of existing pieces. It should not be expected that a complex system definition be compiled into a set of interconnected components in one step! The hierarchy-based divide and conquer algorithm to suit the human design psychology, should be employed. A system is defined in layers, starting with the most abstract whole, growing towards the least abstract modules. In traditional software development, probably the most troublesome activity has been the system integration. To facilitate integration, it is the best time to provide interface [Tanik and Ertas 1992] specifications, while decomposition is taking place. When concepts are partitioned in abstraction, the knowledge is very fresh about the whole, its parts, and what should be communicated among the parts. Also, important Interface specifications should be carried out simultaneously with decomposition.

As a summary, the desired capabilities can be listed as:

- abstraction mechanisms,
- top-down decomposition,
- logical model verification,
- structure-oriented design,
- correlation between Specification and Implementation, and
- compliance with a component protocol.

A full component-oriented approach is hard to find. Recent research has demonstrated the appreciation for such a direction. Work towards component-oriented methodologies is hinted in [Wallnau et al. 2002, Heineman and Councill 2001, Herzum and Sims 2000] as a selected list.

Peculiarities

The most peculiar aspect of Component Oriented development is its coupling between the abstractions and implementation. On the other hand as a principle, requirements have to be isolated from implementation details. To avoid over-specification and to have a control over the structure, high-level abstractions should be exercised in isolation from implementation. The

discriminatory questions, what to build and how to build refer to requirements analysis and design activities, respectively. Concerns should be kept separate for requirements and design, since having to work with implemented components makes developers "design for component correspondence." That means in the early stages, the concern for suitable decomposition so that existing components can match the lower-level modules should be maintained.

Why not start with components in a bottom-up synthesis to form the system? The answer is simple: it is the system, not the components, we are trying to build. Components are an efficient intermediary in the effort to reach the main goal – the system. Consequently, we start from the point we are sure about: the system definition. Any route to reach the implemented components is acceptable within engineering parameters – there is always more than one solution to any engineering problem. There will be cases where the decomposed abstractions will not exactly meet existing components. Then, new components may have to be created, existing components may be modified, or the abstraction hierarchy will be revised. This is the point that triggers cycles of abstraction-specification/component-composition activities until a perfect match is attained. Figure 5.14 displays a flowchart to model the sequencing of alternate activities.

Although system definition is more associated with solution patterns in a domain- and component-oriented approach, the conceptual separation of what/how related concerns is still valid. The boundary of this separation has been pushed higher in the abstraction-levels. Now it is the domain analysis that is more concerned with the question 'how' rather than system analysis. System definition itself is blending the two concerns in a fuzzy region along the abstraction scale. Figure 5.15 depicts the abstraction levels for activity types in Domain Oriented development. This coupling between the definition and solution is a result of limiting the domain. Parallel to an automatic code generator, due to the domain orientation a high-level concept can directly be linked to a solution pattern. Repetitive application of mapping definitions to solutions allows for pattern extraction from the domain experience. Flexibility is lost as a trade-off. As long as the application technology remains the same, similar concept-to-implementation patterns may keep being used.

Separation of definition and implementation concerns is mostly for the understandability of the model. For the theoretical solvability of the problem, separation is not necessary, and automatic code generation will work the same for the cases of both hierarchically and flat organized instructions.

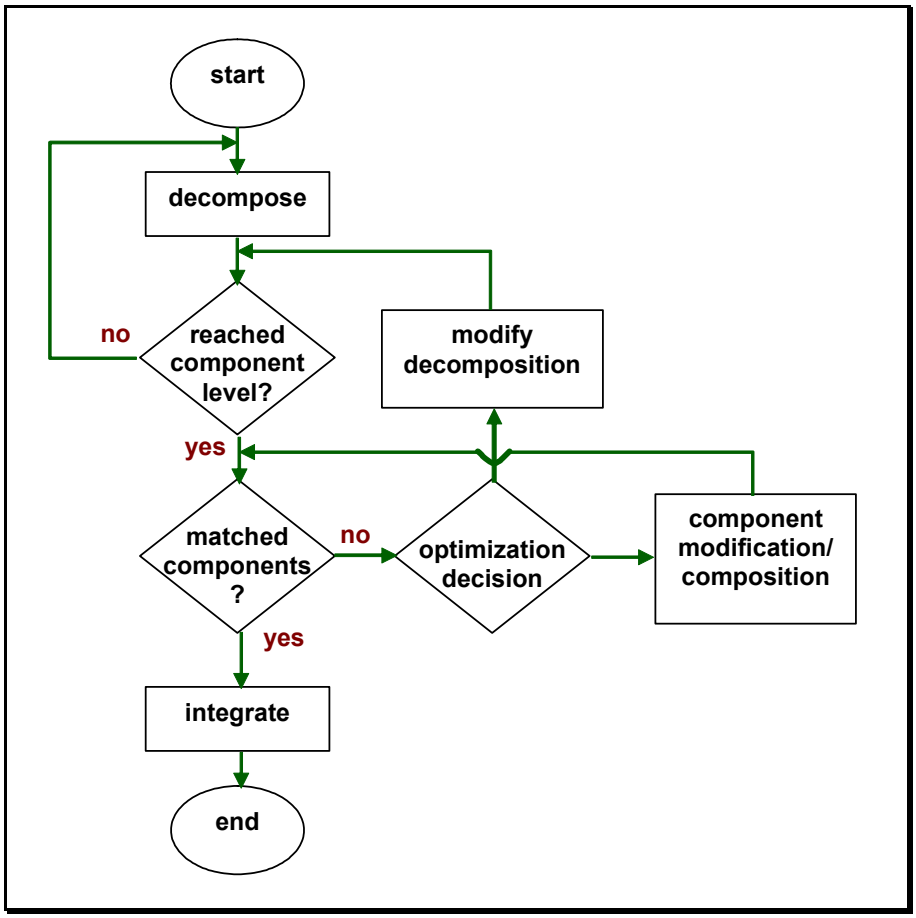


Figure 5.14. Decomposition and composition

Conceptually, experienced developer will keep the what/how discrimination in mind during any levels of modeling. The tools for modeling requirements and design have always shared similar constructs and even similar models for both activities. This assertion has been valid for both traditional (structured) and Object Oriented methodologies. For the latter, even the implementation language should better be known before starting the highest-level analysis modeling: is multiple-inheritance allowed in the model? The answer is yes, if the language supports!

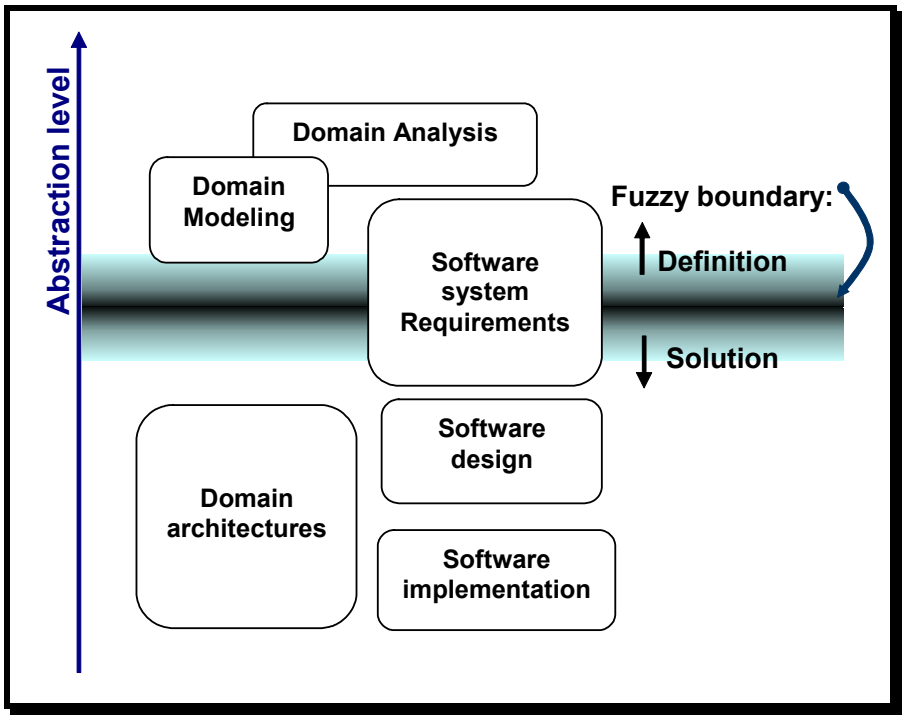


Figure 5.15. Shifted concerns after Domain Orientation

Specification

A fundamental advantage in Build by Integration paradigm is the shift of emphasis to more abstract levels of specification. It is also another general fact that errors found in later phases require more resources to fix. They should be resolved at the earliest possible stages. This means that maximum effort should be spent during the early activities that relate to high-abstraction levels. The requirements and its modeling should be *exercised*, or refined, in order to stabilize and clear them from errors as much as possible.

When emphasizing top-down design, specification becomes more important than ever. Ideally, specification should be the only human-dependent activity in software development. If specified correctly and in a manner a computer can understand, a system definition could be automatically converted to code. Usually high-level descriptions of systems are naturally more abstract and they lack details. For an automatic code generation to take place, the details should also be inserted in the specifications. If it is a domain that has matured over a period of time, then the specifications can be interpreted with their common implementation patterns for automatic filling of the details. This requires a populated knowledge base.

Specification of a system is carried out essentially by the process of decomposition. An analyst knows the system by name, so the initial module is a black box representing the system with its name. The analyst should further know the immediate capabilities or top-level system functions do be defined as top-level abstract components of the system. As in the case of the traditional Lines of Code method for software size and effort estimation, decomposition can be achieved before the completion of the analysis model. Here, the decomposition is conducted not for estimation or other purposes; it is done only for specification. The system requirements are modeled through partitioning the whole into its parts, through recursion. Connectivity is also specified along with decomposition. This is like providing the composition information after every action of decomposition. Two different kinds of connections can be regarded as "vertical" and "horizontal." The vertical connections refer to part (lower end) / whole (upper end) relations that are structural and that correspond to decomposition. The horizontal connections refer to the relations among the identified modules and will have reflections to the dynamic modeling. Horizontal connectors are comprised of an early specification to compositions that will take place later.

Expectations from a good specification environment include:

- informal entry
- automatic internal formal representation
- feedback capability, with incomplete specification
- representation of the system, plus its environment
- test for conflicts
- support abstractions and reusability at all levels
- prototyping capability for exercising the specifications

Most of the modeling activity is expected to correspond to the specification process, rather than the design. At early stages, the tools should provide feedback in terms of various dependencies, sequencing among events, and time delays between actions besides different graphical views. These abilities correspond to prototyping of the specification. Also for a more involved futuristic addition, visual prototyping aids can be devised in a run-time simulation tool [Dursun and Dogru 1995]. That way, before implementation, a representation of the product's look alike can be dynamically exercised.

A Specifically Component Oriented Approach

Structured programming provides enough power to represent any problem solvable by a Turing Machine [Prather 1997]. But our structured methodologies were not satisfying the modern engineers. Also, object orientation is a powerful mechanism, but component utilization can be more effective through a more compliant approach, rather than accommodating components in an object-oriented model. Build by integration and the structure emphasis are the key notions missing in the previous modeling methodologies.

To serve the needs for the contemporary software developer, Component Oriented Software Engineering (COSE) [Dogru and Tanik 2003] approach has been under research. Yet in its infancy, the approach anticipates help in the development of a supportive methodology. However, a modeling language (COSEML) [Dogru and Altintas 2000] is already available. Implemented as concept CASE tools, the prototypes have been used by students in class projects [Altintas 2001]. The language provides primitives for decomposition among abstractions and also mechanisms for their correspondence with component representations.

Packages in COSEML operate as container abstractions that can hold packages, data, function and control abstractions. Any of those abstract primitives can be implemented by a component. In a model, a component represents the implementation of an abstraction and at the same time an abstraction represents the definition of a component. This relation between the corresponding abstraction and component pairs is depicted by the "represents" link. First, the hierarchy is refined to a satisfactory level by utilizing abstractions and then the components are linked to them. Components can own interfaces and message connections among the interfaces which are shown by the message-links. A group of messages among interfaces of the components can be abstracted as a "connector" primitive and graphically modeled with the abstractions. Detailed description for the COSE, its language and usage suggestions can be found in [Dogru and Tanik 2003].

As a final light on the topic in this section, the multiplicity that represents the connections can be mentioned. In-line with the structural emphasis and the cohesion principle, an abstraction can be implemented by only one component. If it requires different kinds of services, then probably it is possible to decompose the abstraction, define the connector among the participants of the new decomposition, and to link the participants to different components. One-to-many connections from a component to a set of abstractions, however, are allowed theoretically. If a component supports the functionalities specified by more than one abstraction, based on the concern for utilization of the existing components, this is the way to go.

Abstract Design Paradigm

An earlier study set forth the structure as a fundamental view to a model, and decomposition as the means for specification [Dogru et al. 1992]. COSEML borrowed its abstraction primitives from this paradigm. Also emphasizing the higher-level of activities, this approach suggests prototyping of the specification. The observations paving the way for the school of thought can be summarized as:

1. Hierarchy: The way to organize the modules of a complex system
2. Feedback: Evolutionary prototyping for the definition of the problem
3. Automation: Designer should receive aid in labor-intensive design tasks both in specification and in feedback. The inclusion of the "computational aid" had a broader range of meanings for Tanik and Ertas [1997] that shaped into a claim in later study that is related to the very fundamentals of the *scientific method*.

The paradigm proposes four stages of activities with iterations across any stages allowed. They are specification, domain assignment, component acquisition, and system integration. The specification activity corresponds to the observation on hierarchy and progresses by the means of a structural decomposition. Domain assignment is a preliminary suggestion for the development environments for abstract modules. Once a module is assigned a domain, then an existing computer aided environment would take on from that point in the development of the module, constituting the third stage. There could be a variety of Domain Environments corresponding to the third stage. The last stage was actually thought as an inter-domain integration of large-grained modules.

This study was supported by three prototyped tools; namely DODAN [Yin 1988], DARMS [Christiansen 1989] and SYDEN [Dogru et al 1992]. As a summary, the high-level constructs were ready waiting for an enabling technology for applicability. The mentioned attempts as academic research were far from offering component protocols that industry would adopt. Also component library domains were required to mature in the real world. Such work could only be a result of intensive industrial experience and development. COSE actually played the role of a matchmaker between the Abstract Design Paradigm and the component technologies.

Towards a Methodology

COSE is supported with a graphical language and a coarse process model as depicted in Figure 5.14. A detailed recipe in the form of step-by-step instructions for developing systems is missing. Some suggestions are emerging in this direction. This section briefly introduces a preliminary methodology which is under development.

The modeling activity is fundamentally different from that of the object-oriented approaches. Nevertheless, besides the structural decomposition view, other aspects can be shared. The established procedures of Use-Case Diagrams and Collaboration Diagrams utilized in UML can contribute to a component-oriented methodology.

Class Diagrams as an essential tool in object-orientation are regarded as the semantic modeling for the Domain in our approach. Rather than incorporating them in the definition of a system, COSE starts after the utilization of class diagrams in domain definition and initiates the structural decomposition. Class diagrams represent a variety of relations among classes. Inheritance and Composition (or aggregation), are frequently referred to as structural relations. With an inherent hint on hierarchy, such relations are invaluable in the classification and organization of domain-wide features and concepts. Also, the associations corresponding to the relations in the Entity-Relation Diagrams are another very powerful means for defining the domain ingredients. It is this conceptual coupling among items that integrate a domain's knowledge, out of a set of otherwise independent set of primitives.

A compact description of the process is as follows:

1. Once the idea formation about the boundaries of a system is achieved, introduce it as the top-level abstract module using a COSEML "package." This is the top of the hierarchy.
2. Decompose the system into its immediate abstract components, referred to as sub-systems. Try to correspond to structural units in abstraction, covering all the system capabilities.
3. After the introduction of each new abstraction, review its connectivity with the previously declared ones. Declare and specify connectors as needed. This applies for decomposition at any level. For each abstraction, try to declare a separate interface per connection. Specify service-requesting calls, as well as service methods, in the interfaces.
4. Conduct a Use-Case Analysis: The abstractions taking place in the first levels of the COSEML hierarchy can be used as "use-cases" in a separately drawn use-case diagram, or alternatively, the hierarchy can be preserved and "actors" can be superimposed with their relation links. Actors and use-cases are the two elements for drawing use-case diagrams. The use-case purposes can be achieved on the parts of the hierarchical decomposition.
5. Continue further decompositions to satisfy the use-case analysis. Actually, a use-case diagram should be drawn per system capability – corresponding to partially a subsystem, or a collaboration of

subsystems. A use-case diagram will require use-cases corresponding to system functions. These additional use-cases may need new items in the hierarchy to correspond. New introduced abstractions are the decompositions of the sub-systems. Also, they always represent a structural part of some "whole."

6. For every abstraction that corresponds to a use-case, draw a collaboration diagram. A separate diagram can be used, or the abstractions participating in the collaboration can be selected out of the hierarchy, optionally "turning off" the others into invisibility. Connectors will model messages among the nodes of the new diagram. The collaboration diagram may require and guide in the declaration of further decomposition products – more nodes on the hierarchy.
7. Continue decomposition as a refinement to the specification based on your experience, keeping in mind possible chunks of components.
8. Iterate until all abstractions are met by a set of components, as shown in Figure 5.14. Prefer to utilize existing components before having to create new ones, also before composing super-components out of existing ones.
9. Try to represent the components with packages only (rather than data, function, and control abstractions).
10. For each component, try to declare a separate interface per connection.
11. Now all the components should be located and non-existing ones must be specified.
12. Implement new components if required.
13. Integrate the system along the lines of connector specifications.

More tools are required. Some are currently under research and development, such as testing and metrics tools.

Futuristic Step: Automatic Location and Integration

The domain entities are refined to the component level. This refinement produces a specification for the components. If the specification were sophisticated enough, a matching component could be located automatically, even over the Internet. This would reduce the complex issue of software development into specifications only. Such sophistication requires semantics to evolve and yield the similar cognition on the system specification side and the components side as well. Component specifications on the system should include semantic information to conduct the search. The self-specifications desired to accompany components should also include semantic descriptions.

Being a difficult topic, semantics will also benefit from limiting the domain coverage, but will always remain to be a difficult problem.

Domain Model to Development Medium

In general, developing a system starts with a mapping procedure. After the initial steps in the definition of the system, the domain model will be searched for related concepts, processes, features, structures, and finally components. The elements will be mapped to a component-oriented development tool. At the highest levels, corresponding features and concepts corresponding to biggest chunks of structures will be copied from the domain to the structure hierarchy. Modifications will be made, and then the compliance measures will be enacted with the new representation media for the developing system.

Other tools have been developed besides component protocols, which are directly helping in domain-oriented development. Architectures and frameworks correspond to implementation related to bigger chunks. Design patterns relate to medium-sized partitions in the solution. Design patterns are abstract, but they are smaller than frameworks. This results in a need for more instantiation and specification towards the lower-level refinement. Nevertheless, they provide the reusability of design so their leverage should be incorporated. These ideas, corresponding to a group of components, are powerful mechanisms in the mapping of higher-level descriptions and more amounts of reusability, from the domain model to the development environment.

Summary

The benefits of Domain Oriented software development are:

- Reuse of development experience: A team, assigned to similar tasks will have better chance to improve their experience and apply it to forthcoming projects.
- Limitation of the breadth of required learning for the development personnel: less time in training, more effective learning.
- Reuse of requirements and design: recurring patterns make it easy to apply high-level development activities, a targeted order-of-magnitude gain in reusability.
- Leverage for the organization in competitiveness.
- Almost no new code modules to be tested in matured domain environments.
- A tool aid for focusing on the core competency of a development organization.

- More accurate expectations for the deadlines.

Disadvantages

- Expensive and non-predictable set-up activity for a revolutionary re-organization scheme.
- Eventual boredom for developers for developing similar systems.
- Overhead for adjusting to new technologies, as a smaller-scale repetition of the set-up activity.
- Supporting methodologies are not yet established very well.
- There are very few domains that have matured in terms of component recognition.

Domain orientation utilizes previous techniques in modeling and knowledge/information acquisition. The concept can be vitalized by the integration of component-oriented development ideas. Some essential methods have been devised and domain-oriented development is ready for the industry. More tools can be incorporated to enhance the coverage of the entire lifecycle tasks.

Questions

1 It was explained in the text that environments trying to target many domains have not been successful. PL1 programming language was given as an example. Today, most of the very successful visual environments are offering **database** support with querying languages and database-aware objects, besides their primary domain: GUI. How can you explain their success in spite of the wider domain coverage?

2 Give an example development step for a specific project where automatic filling in of details is not desired. List some conditions for which automatic detailing is desired and some conditions for which it is not desired.

3 What are the different abilities that should be sought for the Domain defining expert and the integrator in a domain specific software development environment?

4 How dependent are the concepts of Domain and Components:

What are the factors for enabling domain-oriented development without components?

What are the factors for enabling component-oriented development without domains?

5 How can the **experience** of the experts in a domain be captured and represented for reuse?

6 Could there be generic tools that can be used across any domain? How about multi-domain components? Explain.

7 How can one combine "implementation-independent specification" with "design for component correspondence?"

8 If a domain is supported with both abstraction-level reusable specifications and implementation-level components, explain if there is still need for an approach such as COSE: COSE also starts with abstractions and arrives at components.

9 Compare the three terms: Frameworks, Design Patterns, and Components, with respect to 1) abstraction levels, 2) granularity, and 3) their applicability for Domain Oriented Software Engineering.

10 Propose a testing scheme for the proposed component-oriented methodology.

References

Altintas 2001 — Ilkay Altintas, A Comparative Study for Component-Oriented Design Modeling, M.S. Thesis, Computer Engineering Department, Middle East Technical University, May, Ankara, Turkey, 2001.

Arrango 1994 — G. Arrango, "Domain Analysis Methods," in *Software Reusability*, W. Shcaefer, R. Prieto-Diaz, M. Matsumoto (editors), Ellis Horwood, 1994.

Booch et al. 1999 — Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

Christiansen 1989 — M. Christiansen, Integrating Domain Knowledge into Software Components, Ph.D. Dissertation, Southern Methodist University, Dallas, Texas, 1989.

Diaz 1987 — Ruben, Prieto-Diaz, "Domain Analysis for Reusability," *COMPSAC 87: The Eleventh Annual Computer Software and*

Applications Conference, October pp: 23-29, 1987.

Dogru and Altintas 2000 — Ali. H. Dogru, Ilkay. Altintas, "Modeling Language for Component-oriented Software Engineering: COSEML," *The Fifth World Conference on Integrated Design and Process Technology*, June 4-8, Dallas, Texas, 2000.

Dogru and Tanik 2003 — Dogru A., Tanik M.M., 2003, "A Process Model for Component Oriented Software Engineering," *IEEE Software*, Vol 20, No. 2, March/April, pp. 34-41.

Dogru et al. 1992 — A. H. Dogru, S. N. Delcambre, C. Bayrak, Y. T. Chen, E. S. Chan, W. Yin, M. G. Christiansen, and M. M. Tanik, "An Integrated System Design Environment: Concepts and a Status Report," *Journal of Systems Integration*, October, 2(4), pp. 317-347, 1992.

Dursun and Dogru 1995 — Huseyin Dursun, Ali H. Dogru, "Prototyping Specifications through Visualization," *The First World Conference on Integrated Design and Process Technology*, December 8-9, Austin, Texas, Vol. 1 pp:362-368, 1995.

Fayad 2000 — Mohamed E. Fayad, "Introduction to the Computing Surveys' Electronic Symposium on Object-Oriented Application Frameworks," *ACM Computing Surveys,* Vol. 32, No. 1, March: pp. 1-11, 2000.

Gamma et al. 1995 — Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, Massachusetts, 1995.

Heineman and Councill 2001 — George T. Heineman and William T. Councill, *Component-Based Software Engineering,* Addison Wesley, 2001.

Herzum and Sims, 2000 — Peter Herzum and Oliver Sims, *Business Component Factory*, Wiley, 2000.

Holibaugh 1993	Robert Holibaugh, Joint Integrated Avionics Working Group (JIAWG) Object-Oriented Domain Analysis Method (JODA), CMU/SEI-92-SR-3, November, Pittsburgh, Philadelphia: Software Engineering Institute, Carnegie Mellon University, 1993.
Hopcroft and Ullman, 1979	John E. Hopcroft, Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 1979.
Itoh et al. 1998	Kiyoshi Itoh, Toyohiko Hirota, Satoshi Kumagai, Hiroyuki Yoshida (editors), *Domain Oriented Systems Development: Principles and Approaches*, Information Processing Society of Japan, Gordon and Breach Science Publisher, Japan, 1998.
Kang et al. 1990	Kyo C. Kang, Sholom C. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, CMU/SEI-90-TR-21, ADA 235785, Pittsburgh, Philadelphia: Software Engineering Institute, Carnegie Mellon University, 1990.
Kang et al. 1998	K. Kang, Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., “FORM : A Feature Oriented Reuse Method with Domain-Specific Reference Architectures”, *Annals of Software Engineering*, Volume 5, J. C. Baltzer AG Science Publishers, Red Bank, NJ, USA, pp. 143-168,1998.
Neighbors 1989	J.M. Neighbors, “DRACO: A Method for Engineering Reusable Software Systems,” *Software Reusability,* Vol. 1, pp: 295-320, ACM, 1989.
Prather 1997	R. Prather, "Regular Expressions for Program Computations," *The American Mathematical Monthly*, Vol. 104., No. 2, pp. 120-130, 1997.

Riebisch 2003	Matthias Riebisch: "Towards a More Precise Definition of Feature Models." Position Paper. In: M. Riebisch, J. O. Coplien, D, Streitferdt (Eds.): *Modelling Variability for Object-Oriented Product Lines*. BookOnDemand Publ. Co., Norderstedt, 2003. pp. 64-76.
Simon 1969	Herb A. Simon, *Sciences of the Artificial*, MIT Press, Cambridge, Massachusetts, 1969.
Simos 1996	M. Simos, "Organization Domain Modeling (ODM): Extending Systematic Domain Analysis and Modeling beyond Software Domain, *IDPT,* 1996.
SPC 1990	Software Productivity Consortium, *A Domain Analysis Process* Domain_Analysis-90001-N, January, Herndon, Virginia, 1990.
Szypersky 1998	Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming,* Addison Wesley, New York, 1998.
Tanik and Chan 1991	Murat M. Tanik and Erik S. Chan, *Fundamentals of Computing for Software Engineers*, Van Nostrand Reinhold, New York, 1991.
Tanik and Ertas 1992	M.M. Tanik and A. Ertas, "Design as a Basis for Unification: System Interface Engineering," *ASME PD-Vol. 43*, pp: 113-114, 1992.
Tanik and Ertas 1997	M.M. Tanik and A. Ertas, "Interdisciplinary Design and Process Science: A Discourse on Scientific Method for the Integration Age," *Journal of integrated Design and Process Science,* September, Vol. 1 No. 1: pp. 76-94, 1997.
Wallnau et al. 2002	Kurt C. Wallnau, Scott A. Hissam, and Robert C. Seacord, *Building Systems from Commercial Components*, Addison Wesley, 2002.

Yin 1988 Weiping Yin, An Integrated Software Design Paradigm, Ph.D. Dissertation, Southern Methodist University, Dallas, Texas, 1988.

Chapter 6: Component Oriented Software Engineering

Introduction

Software is now recognized as a critical field in the rapidly advancing human history as we prepare for a new millennium. Intelligence related solutions have recently moved from mechanical or electrical technologies to software. More artifacts are containing a computational element as an integral part. Yet the engineering for the software technologies are far from the maturity levels other disciplines have reached. Within this context, one of the most important characteristics of this field is that software is still being custom-built rather than being assembled from existing components [Pressman 1997]. The predecessor fields have discovered this "reusability" long before software that should have exploited the notion through more concrete approaches.

It was only after the emergence of component technologies [Krieger and Adler 1995] we started to anticipate the benefits of an engineering methodology for system development that is formally oriented toward reuse. Today, extensive research and technological utilization work for the component technologies are being reported constantly. Yet the potential contribution to increase the development efficiency by an order of magnitude is not exclusively addressed. Defining wiring-level issues and representing components in Object Oriented (OO) platforms have constituted the bulk of this work. Now, we must capitalize on the feature of components that will enable a paradigm shift towards "integration" [Tanik and Ertas 1997] rather than "code development."

Observing the demanding market for software with complexities in excess of ten million lines, the proposing of new methods that only improve the development efficiency through linear gains will not be satisfactory. Requiring development periods comparable to a decade, such projects are deemed to be extremely troublesome. The definition of the project will drastically change during this period. Initial request will evolve in the context of the rapidly changing world, new technologies, and business or market conditions. Rather than undertaking such huge risks, readily existing code should be utilized that has been developed and tested already. Component technologies offer the ease of the utilization of existing modules. Meanwhile new engineering approaches are needed that facilitate the formulation of problems in a decomposable manner. If its modules can relate to existing component technologies the decomposition model will aid the integration of the system.

Recent Trends

Computer science itself is recent. In the course of the last half century the third radically different paradigm is in the making. The ad-hoc approaches were replaced with structured methodologies after the structured programming languages were appreciated. Another decade passed before OO approaches could follow their enabling technologies – the OO languages. Currently, Component technologies are maturing and attempts to engineer their utilization are defining the Component Oriented Software Engineering (COSE). The infancy of the approach asserts itself with modest definitions. Most of the attempts can be referred to as Component Based [Heineman and Councill 2001] rather than Component Oriented. The earlier work concentrated on the connection of a pair of components. Later, higher-level concepts were exploited such as design modeling to accommodate components in OO representations.

The alternative offered through OO platforms to accommodate components carried the intention to develop the code. To obtain the desired leverage gain on the order of a magnitude, Build by Integration paradigm is offered. One way to utilize this concept is by maturing the Component Oriented Software Engineering [Dogru and Tanik, 2003]. The orientation regards composition of components from the requirements specification stage onwards. Even logical modeling takes into consideration the existing set of components.

Components are often regarded as an extension to OO technique because of the resemblance of interface definitions to class definitions. If anything is to be developed, the contemporary approach is OO. Therefore it is natural to develop components using OO techniques. Components obey a protocol that includes structured interfaces and the commitment to provide the services declared in the interfaces. As long as it sticks with these commitments, a component can be developed through any non-object-oriented approach, also.

Development, wiring, and protocols for components are important; they constitute the enabling technologies for the Component Orientation. Although improvements will still be welcomed, the fundamental problems defining the “software crisis” are not related to technology level issues. There is enough know-how built for connecting and running components, despite the fact that there is still plenty of room for further improvement. Also there are so many algorithms coded before, and creating new components or converting existing code to comply with component standards does not seem to be extremely complex. New focus should be on searching and integrating existing components to satisfy a given set of requirements.

Constituents of the new approach

So much has been investigated towards putting component orientation into practice but yet the industry is still in its infancy. Component Oriented Software Engineering (COSE) is developing with its concepts and tools. The idea is based on the principles of Herbert Simon [1969] and the success routes of the previous engineering disciplines. The "Abstract Design Paradigm", [Tanik and Chan 1991] bridged that early work to a comprehensive engineering methodology idea. Later the approach gained its fundamental definitions [Dogru and Tanik 2003], a graphical modeling language (COSEML [Dogru and Altintas 2000]), and some research defining parts of a methodological approach [Bayar 2001, Salman 2002]. Possible complementary utilizations of some new ideas such as design patterns and frameworks are being studied for the new paradigm [Avkarogullari 2003]. Basically process models, supported with languages, tools, and methodologies are required in order to capitalize on the paradigm in the field of engineering practice. The research and the development of supporting tools are in progress. The following sections introduce so-far-established constituents of the approach.

Possibilities

Expectations from the new orientation are growing and also new avenues in computer science are being developed. If successfully deployed, COSE may be the default and natural way to the software industry and the innovations will be appended to this kind of approaches. Information Theoretic and quantum computing related research is not far from connecting with the "systems view," where a set of communicating well-defined components need to be modeled. Aspect-oriented software development is another idea that is heavily being discussed. Here, the idea is to have a handle on user-related aspects that cut across all the units of the software. It is not feasible, for example, to localize the security aspect of a software system to an object or a component. Components are a huge step towards separation of concerns. Also their connection for any "aspect" can be more structured. Component orientation promises the capability to support the novel ideas. On the process dimension, agile methods are the innovation and an efficient build-by-integration approach may find a wonderful application foundation within an agile methodology.

A system integrator only needs to understand the interface definitions of the components. How the component works internally is not very important. Building on this idea, one can extend the notion to the hardware/software co-design practices. Actually, the system specification starts in logical levels and the developers do not need to rush into decisions about the development media for the future components. In other words, they should not know if a specific function will be implemented in hardware or software. Even,

different kinds of hardware such as mechanical or electrical are considered in this kind of a hybrid co-design. More and more systems involve multi-disciplinary engineering. Component orientation also supports the abstract design notions where the discouraging of over-specification is rewarded with optimal solutions to multi-disciplinary engineering designs. Figure 6.1 depicts a process model that has been studied for a multi-disciplinary platform supporting Abstract Design Paradigm (ADP).

The ADP suggested decomposition of the logical specification so similar to the desired approach for COSE. The next stage refines the defined units of the logical decomposition, possibly assigning a development domain such as some hardware or software field. Refined to an almost complete specification the decomposition units are well-suited candidates for the modern components to replace. However, they may still be at a level that is not detailed enough. Such design details may be completed in the next stage that is responsible for completing the individual units – those are modules or components. Finally, the polished components will be integrated using the connector/interface specifications that were defined starting with the decomposition stage.

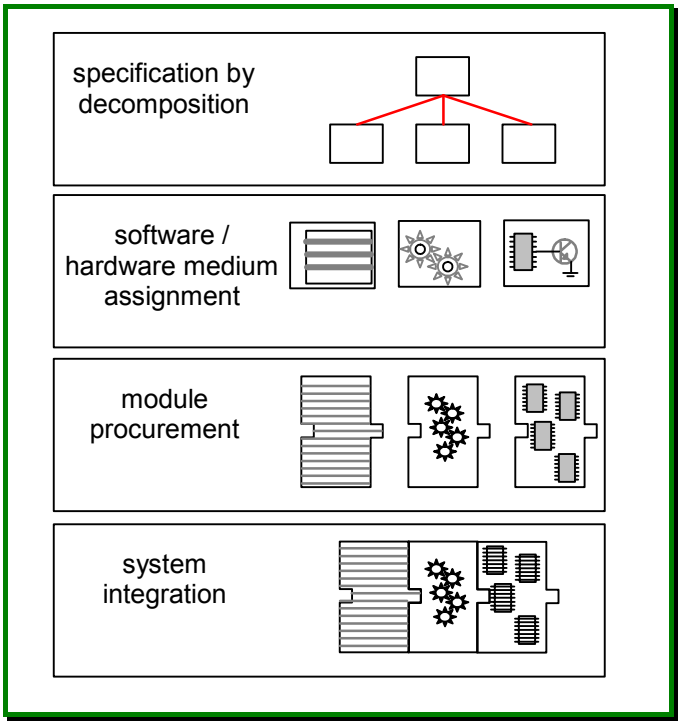


Figure 6.1. Abstract Design process for multi-disciplinary development

The emphasized interface notion gains some different flavor when hardware components are also accounted for. Interface engineering needs to be defined and supported with domain-specific lower-level models in addition

to abstract level specifications. In high abstraction levels, interface specifications may be of a similar nature. Whereas, in the lower-levels of specification corresponding to implemented components, the interface ingredients will represent particular quantities (units) such as volts, meters, etc. Moreover, interfaces linking cross-domains will have to address units from different domains. In the lowest-levels, this kind of an interface corresponds to the classical definition of a "transducer." A microphone is a transducer that converts the values in the acoustics domain to the values in the electrical domain. Recalling the fact that the "interface" has been the location of many failures, the due deserved emphasize should be returned to the interface concept, thereby addressing a key problem area. For instance, tragic consequences occurred when space shuttle Challenger exploded as a direct result of an interface problem: the faulty gasket between the fuel tank and the body.

Incorporating architectural concepts

Object Orientation provided the organization of some general principles in a formal presentation. More concepts developed building on the new practices such as component technologies and domain orientation. In this section, the issues in possible incorporation of the architectural frameworks and design patterns are being investigated.

Design patterns are introduced in [Gamma et al., 95] as generic solutions to recurring sub-problems. Design patterns are abstract mechanisms. They can be represented through collaboration models [D'souza 98]. In COSEML, a small number of components can represent a design pattern, although at an abstract level. To utilize the concept, a logical as well as an implementation level representation of components is proposed in [Avkarogullari, 2003]. Since COSEML can contain both logical and implementation level representations for components, a design pattern can be placed as a part of the decomposition in either abstraction level. The instantiated details of a pattern are also maintained as a unit. Whereas frameworks are already implemented structures but they correspond to bigger chunks of a solution than design patterns. Figure 6.2 depicts this relation among the architectural elements with respect to size and abstraction levels.

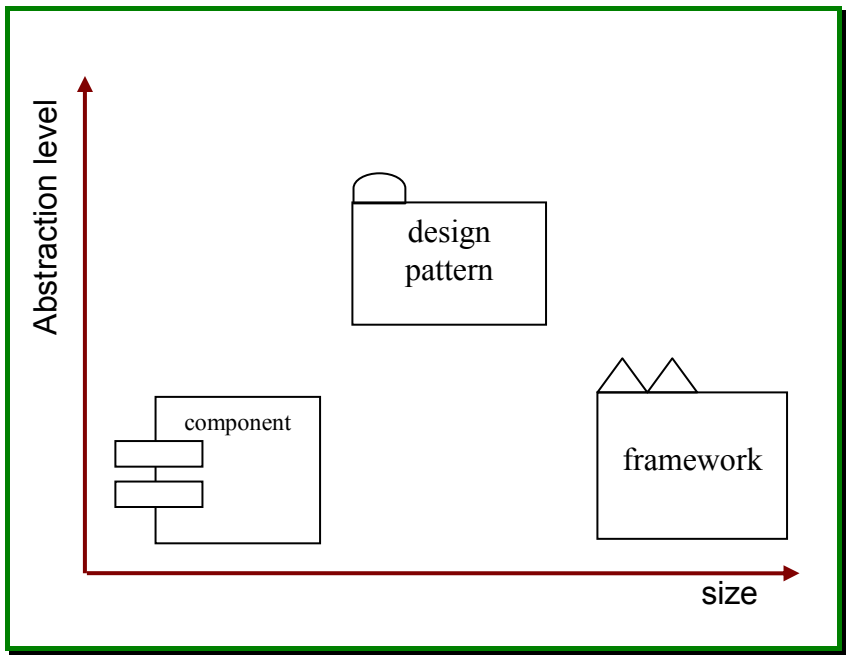


Figure 6.2. The abstraction levels of various architectural elements

Any element in a COSEML model corresponds to a structural piece, big or small. For those that include more than one component, there has to be an abstract representation.

Component Oriented Process

The fundamental concepts and a process model were introduced in [Dogru and Tanik 2003]. According to this process that orients the developers towards the possible availability of a component set, the first critical step is to decompose the definition of the system into an abstract set of communicating components. The communication among the units implies a connectivity that is modeled through abstract connectors. The decomposition activity hence results in a defined set of abstract components and a set of connectors as shown in Figure 6.3. An abstract component is actually a specification of a real component to be searched. Using the abstractions as definitions of what to expect from the (hopefully existing) implemented components, the component acquisition stage can be activated. There may always be some implementation or modification need, but the ambitious expectation is that most of the needed components should be available in the market. Integration of the gathered components follows, guided by the specifications contained in the connectors.

An efficient development will be supported better in matured domains. A domain is a set of applications that solve problems in a field of industry. Domain orientation, therefore, is a closely related concept to component

orientation. One way to improve the effectiveness of the proposed approach is to conduct domain analysis [Itoh et al. 1998]. Domain orientation will supply the developers with tools and components at various levels of abstraction, besides know-how and any practice that can be utilized in the convention of patterns.

Although a revolutionary way of thinking is sought for software development, there is so much that can be exploited out of the practices defined so far. As long as the critical issue of complying with the new paradigm that suggests Build by Integration, borrowers from existing methodologies are welcome. In this respect, the early symbolic definitions in the ADP research are replaced by their UML [Booch et al. 99] counterparts, where possible. The fundamental view is the structure of the model that should support a developer's decomposition, and later composition decisions. COSEML was designed with respect to these considerations. The widely accepted use-case and interaction modeling defined in UML can be adopted by COSEML. The abstract-level elements in a COSEML model can be used as ovals in a use-case diagram. To employ the interaction modeling such as that in UML, represented in collaboration or sequence diagrams, the messages in COSEML provide the basis. Messages, representing the lowest-level connector information can be ordered by means of numbers to yield the collaboration models in UML.

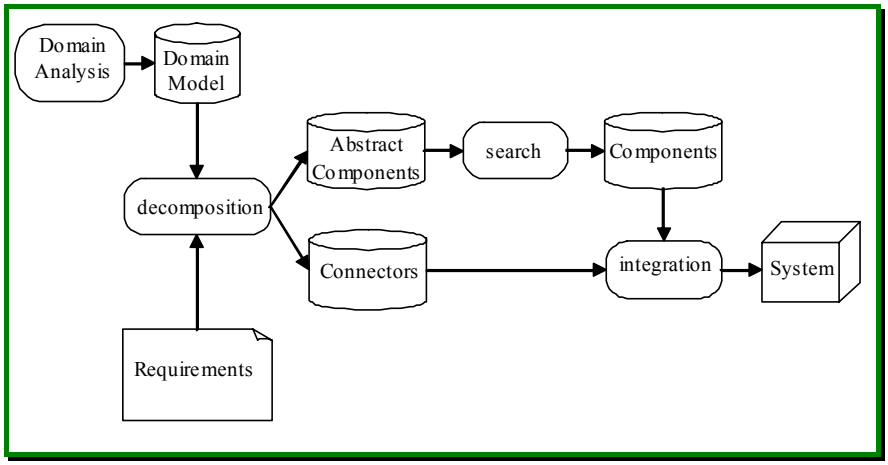


Figure 6.3. Component Oriented development process

Solution is available in parts

Our goal is to partition the system into modules in such a way that those modules can be implemented by existing components. Then the system development reduces to:

- decompose

- find
- integrate

rather than the existing strategies that implies

- define
- develop from scratch.

However, there is a paradox that has to be addressed in this simple and radically new approach:

- During decomposition, existing components should be considered for efficiency;
- On the other hand, early specification should not consider implementation level issues.

This observation is supported by the concern about the violation of the conventional practices: There seems to be an ambition to defy the following two principles:

1. Specification of problem vs. solution should be kept separate
2. Over-specification should be avoided.

Conventionally, requirements related activities are concerned with the definition of the problem. Developers should be careful about not defining the solution during requirements specification, in order to comply with the first principle. Respecting the second principle even when refining the implementation-related specification, namely during design, initially the models should only incorporate more abstract definitions. This early design activity is commonly referred to as the "logical design phase." COSE, implying an efficient specification activity through an existing set of (implementation level) components hence seems to propose an unconventional understanding.

An explanation to this problem is offering some justification to the desire to employ COSE with domain engineering support. The problem definition is assumed to have been accomplished for a family of problems; namely, the domain. Most of the requirements engineering should be conducted while defining the domain. Actually the boundary between the two concerns (problem vs. solution) is pushed towards the earlier activities.

As in the TTL example, component orientation is targeting the fast composition of the solution in a dependable manner, at the expense of losses in efficiency which should be absorbable. When conducted in the context of a domain, the COSE practice requires intimacy with the problem classes and an ability to quickly link the problems to solutions in order to yield the desired development efficiency.

Looking closer, one can see that actually it is not the solution being specified, but the definition of the solution. The very nature of components isolates the

definition from implementation. It is the specification of a component molding in the form of interface descriptions that the developer regards while conducting the decomposition. As a result, defining the solution is never a concern in COSE. The slight conceptual paradox for a skeptical colleague is caused by regarding the *logical* definition of *implemented* units in early stages. Nevertheless, this change in the conceptual activity classifications is worthy of a debate. Figure 6.4 displays the concepts related to activity stages suggesting that COSE is basically involved with an intermediate level of activity that can be defined as "detailed requirements and logical design." Domain Engineering may be involved with the development of components also. This would shift the burden of all kinds of activities to domain engineering. Here we are interested in "Domain Analysis" only where implementation issues are not of any concern. The "Component Development" kind of engineering shown in Figure 6.4 may very well be conducted as part of a Domain Engineering effort.

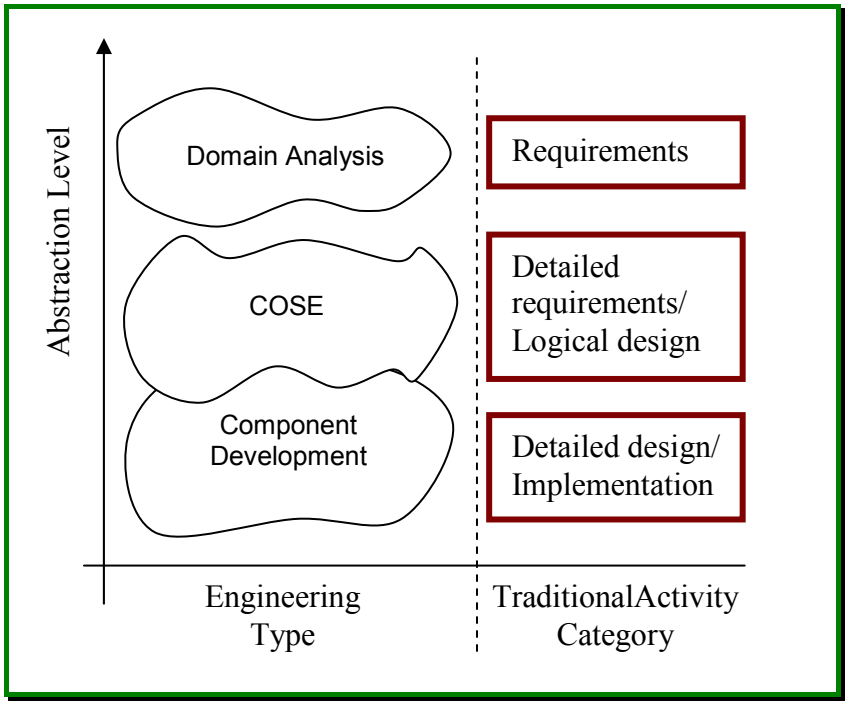


Figure 6.4. Conceptual activity classification related with COSE

Component Oriented Modeling Language

The modeling language had to be graphical and well suited for visualizing a hierarchical decomposition. A typical development starts with the abstract definition of system parts starting with subsystems and advancing towards lower-level components (in abstraction). Later physical components need to be introduced that are responsible for implementing the responsibilities

encapsulated in the previously defined abstract modules. A special kind of a link connects an abstraction to a component that is actually its implementation. This link is referred to as the "represents" link. This link points to a component that "represents" an abstraction in the physical world, and at the same time an abstraction represents the component at a logical level. Relations among the abstractions are modeled through connectors. Connectors represent communications among abstract or among physical components. The default links that connect the elements of the top-down structural refinement hierarchy are the composition links.

Abstract components in COSEML correspond to packages and data, function, and control abstractions. Specific icons depicted in Figure 6.5 represent these logical units. At the physical level, components are the main units. Any component comes with at least one connector. Connectors are also represented by a specific symbol. For the components that have only one interface, the symbols for the component and an interface are almost joined forming another special kind of a symbol.

Thus, accommodating both logical and physical components and structural and operational connections, one COSEML model is capable of representing a complete model. Representing different aspects or parts of a system is also possible.

Package is a symbol that is used mainly for encapsulating the more primitive abstractions (data, function, control). A package defines a scope for the contained abstractions and forms a main node in the hierarchical decomposition. There can be further packages and other abstractions inside a package. Actually, COSEML does not constrain the types of abstractions that can be declared inside another. Such limitations as stating that "a data abstraction can only have data abstractions, as its components" can be defined in a methodology.

Former research has proven that specification of the aspects organized in three dimensions, namely data, function, and control is sufficient to represent any process such as a computer program. Actually, programming languages and modeling languages directly or indirectly address those three dimensions. That is the reason for COSEML to include abstractions besides the "package" that represent the specifications for the three fundamental dimensions. A similar concern has been articulated in the definition of the structured languages. The Turing Machine equivalent modeling power can be attained if the three control structures are addressed; these are sequence, condition, and repetition. Offering such abstractions, COSEML ensures the ability to represent any functional requirement of a system.

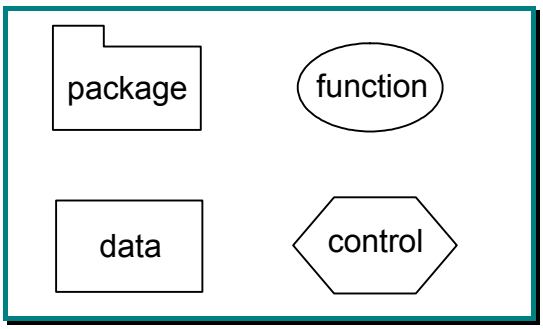


Figure 6.5. Abstractions in COSEML

Of course, so far, the non-functional requirements have not been considered. This language merely offers a structure-oriented decomposition alternative to the existing modeling languages that are object-oriented or traditional.

Especially the abstract level units should be used for high-level abstractions corresponding to the system under development. Although concepts such as data and function may evoke programming-level considerations, here they are abstractions and they need to relate to logical definitions in the system specification. Data abstractions can be used to model data structures that correspond to high-level entities in the requirements model. A data abstraction can declare internal operations just like those of a class in OO modeling. Similarly, function abstractions may have their internal local data structures and they are meant for representing high-level system functions. Control abstractions represent basically state machines, accepting messages that cause state changes. During a state transition, outgoing messages can be generated. State changes also can trigger the execution of the function abstractions, or operations inside data abstractions.

Leveling with components!

The abstract elements are used to define the "decomposition." However, this decomposition may yield a set of abstract components that may or may not correspond to existing components. An efficient development will yield a set that does match existing components. This requires some experience and a good knowledge of the defined components in a development domain. Regardless of how the abstract decomposition was used, it was for defining the real component set that would realize a solution to the specifications. A set of real components has to be acquired and connected to compose the system. The model should be capable of representing the run-time behavior also. Therefore, components, besides being connected to their abstract definitions, are connected among themselves through messages that define the set of operations to yield the desired system functionality.

The components and their connections may recall the "collaboration" technique employed in OO development, especially through UML. A similar

approach is also possible using COSEML. Once a structural break-down is facilitated, the collaboration of the defined units can be studied also in COSEML models having accomplished the main problem of identifying the sub-solutions to the problem.

A connector between two components represents a set of bi-directional messages. Following the convention in component technologies, events are considered as well as methods. Although a method and a message that eventually calls that method are capable of representing the semantics of an "event call," a component-related representation should address the events. An event is actually a special kind of a message that is generated by an external or a sporadic request. In any case, a collaboration can be modeled as an ordered set of methods or event calls.

Interfaces and messages

Universally, an interface defines the incoming calls. These are the methods or event notifications that result in a component performing a service. Some component protocols suggest the representation of the outgoing calls in the interface, as well. This means, besides publishing what this component can do for the external world, what this component needs as services during its effort to accomplish its functionality should also be defined.

This problem is analogous to the representation of external references in a multi-file development platform. If not compiled at the same time, different units that make a program have to know where to locate the external functions that they need, but do not have, and do know that other units have. Compiler technologies have two different approaches to resolve external dependencies:

1. Each unit publishes the services it can offer to outside.
2. Each unit declares what external services needed by it.

Example technologies are C and FORTRAN compilers. C language development suggests the publishing of the offered services in a kind of an interface structure commonly referred to as ".h" files. Whereas FORTRAN source code files declare function names as "external" if such functions are required to be offered by other files. A linker is a system program responsible for resolving the cross-file call references while it unites different modules as an executable program (linking).

When it is components rather than source files, we may prefer to indicate both kinds of external references, request and response, so that we know what it takes for the component to integrate. What services a component offers in this case may even be less important. We have incorporated this component because of what it should do for us. On the other hand we should

also know what other services this component relies on, so that we make sure to include additional components that include such services.

Considering a CO representation that does not support outgoing methods, a method connection has to originate from a component rather than from a specific "outgoing method slot" in an interface. COSEML allows messages originating from non-specific locations on a component. Also, there is no restriction to interpret methods as only "response methods." This means, some of the methods declared in the methods chamber of an interface can be used as requests as well as some others as responses. It would be more preferable, however, to separate the two kinds of methods. The interface representation in Figure 6.6 contains a possible adaptation with a specific chamber dedicated for "request" kind of methods by splitting the methods chamber using a dashed line.

It should be noted that the interfaces act as a specification of a component. This additional representation in an interface that may be handy as an interface is a partial specification to a component. Actually, a component-based development considers only the interfaces of a component (ignoring the internal implementation). In other words, using implementation-level terminology, a program is written to the interface, rather than to the internals of a component.

The method or event links take place at the lower levels of a COSEML design together with the components and their interfaces. A graphical representation of the physical-level elements is shown in Figure 6.6.

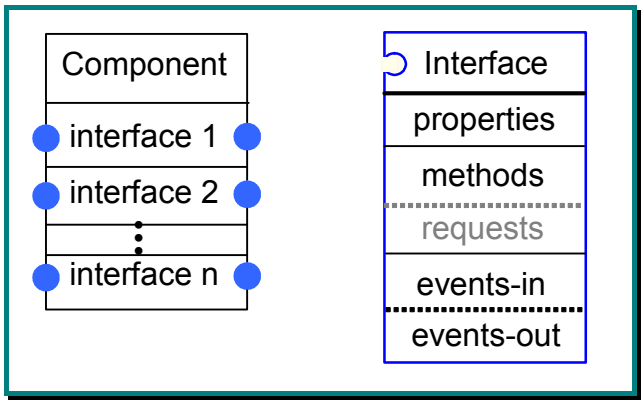


Figure 6.6. Physical level elements in COSEML

To accommodate different interface definition trends, the language should be capable of representing method calls that originate freely from any location out of a component, rather than specific outgoing message definitions (or requests, as declared in Figure 6.6). As a result, a method call can originate

from a request defined in an interface, from a non-specific location of an interface, or from a non-specific location of a component.

Connectors

The representation of a specific connector can be repeated in a model at various levels. Connectors can be drawn between:

- two abstractions
- interface ports of two components
- headers of two interfaces
- A method or event slots of two interfaces in the form of a set of method or event links.

Figure 6.7 displays the different kinds of connectors.

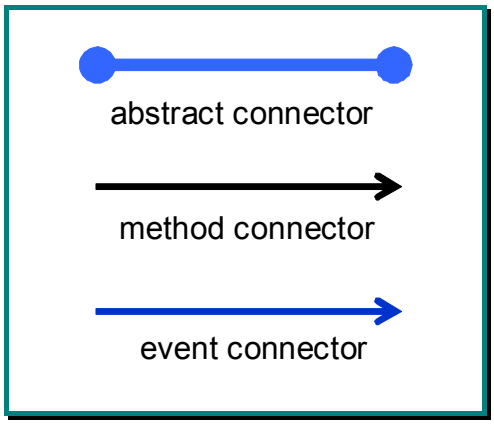


Figure 6.7. Connectors in COSEML

Methods and events conduct similar duties. They both trigger the execution of a process, they carry some information, and they may have some implied synchronization semantics. However, the popularity of "event-based programming" especially when used to define the graphical user-interface behavior has been a factor in the events having to be explicitly addressed.

It is common for the component technologies to differentiate between the input and the output events. Components can subscribe for "input events" with the environment, in order to be notified whenever the event happens. Also, a component can publish "output" events, so that the environment will prepare itself to be notified in case the component generates such events. However, the differentiation between the input and output methods is not as universal. Some literature suggests the listing of the input methods only.

Development

Three fundamental types of activities are concerned here for the demonstration of the CO specific features:

- Requirements Specification
- Design
- Implementation.

Some key concepts should be repeated here: Implementation is mostly integration and some modification, and the traditional boundary between the requirements and design deserves a different angle of sight. Actually, the "implementation" as a major activity in conventional approaches is basically not existent here, therefore such naming of this phase loses its meaning.

Requirements specification is basically a decomposition activity. System definition is conducted by separation into abstract components and by specification of relations among those parts. Whereas the traditional design activity corresponds to mapping of the abstractions to the real components and further detailing the connections, especially among the components.

Requirements Specification

Independent from the approach's being traditional, OO or CO, requirements is a key activity and should be conducted with care and methodology. Promoting this activity to the level of an engineering discipline would not be an over-estimation. Besides the acquisition of the knowledge through different means, and negotiating with the customer, the information must be documented in "specification" form. This form is very dependent on the approach. In COSE, the result of the requirements engineering is a set of connected abstract components defined in a hierarchy. For the elicitation task, existing techniques can be used and supported with visualization and communication tools such as the use-case analysis. OO approaches also have incorporated other techniques that are not necessarily OO. Some examples to such techniques are the use case and state chart diagrams that are successfully blended into modern methodologies. There is no reason a CO approach should not utilize these also.

An OO approach can be revisited to adopt use-case analysis in COSE. We suggest starting with identifying system capabilities that can also be interpreted as highest-level system functions. A use-case diagram can be drawn per system capability. Every use- case (oval) in these diagrams corresponds to system functions and should be further specified through scenarios. The scenarios will be modeled by a sequence of messages to enact them. This brings us to another technique borrowed from the OO graphical tools that is collaboration. UML is the most widely used OO formalism and

the interaction diagrams (collaboration and sequence diagrams) demonstrate a set of messages among objects corresponding to run-time behavior. In COSEML, messages do not appear in the abstract levels of a model. Their definitions can take place in the specification of the connectors. It is only *after* representing the abstractions by components is the instance when the actual messages are drawn.

The use-case diagrams are limited to the abstract elements of a COSEML model. Ovals in a use-case diagram can correspond to packages or function, data, or control abstractions. After presenting the use-case diagram to the user in UML syntax, the diagram can be replaced by another version where the ovals are replaced by package, data, function, or control elements. Figure 6.8 represents a simple demonstrative case where system functions correspond to different types of abstract COSEML elements. It should be kept in mind that any element in a use-case diagram actually is a node in the hierarchy diagram that is the main COSEML view. Some abstract elements are copied to a use-case diagram. Actually, those elements are declared in the hierarchy diagram, and used in the use-case diagrams.

Use-cases represent system functions. A use-case diagram conforming to a hierarchy diagram implies the top-level decomposition is to be conducted with respect to system-functions rather than structural chunks. If this implication will not work out for the current system being modeled, a formal adaptation of use-case diagrams does not prove viable. In that case, use-case diagrams will be treated as separate tools to study requirements. Then necessary mappings of partial or combined use-cases (system functions) to COSEML elements can be conducted. Actually, the function abstractions in COSEML are its corresponding elements to the use-cases.

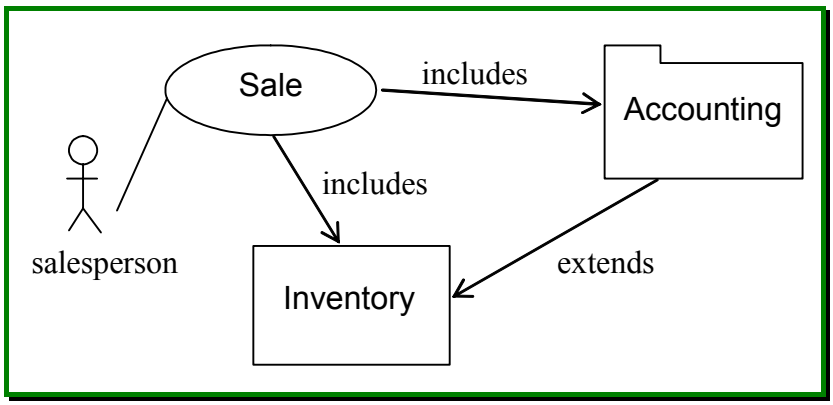


Figure 6.8. COSEML elements in a use-case diagram

If use-case analysis is selected to be the initial activity, highest levels of the COSEML specification will assume the information modeled in the use-case diagrams. Continuing specification decomposes the defined elements to lower-level abstractions, hence developing the hierarchy top-down. User requirements will be accommodated more in the higher-level elements. The development team will introduce more and more of their own inputs to requirements as the decomposition proceeds downward.

During the decomposition it is important to pause after the inclusion of every new element, investigate the need for new connectors, and create an update on the existing connectors. The best time to explain how parts will connect is the time when they are being separated from each other.

Domain Model Utilization

Software Engineering had discovered the virtues of reuse relatively long time ago. By now, it is widely accepted that reuse of higher abstraction entities besides code is very important. A domain can offer partial solutions in a wide spectrum of abstraction levels and other concerns. Recalling the main view of COSEML which is a hierarchy diagram representing the whole system, an intermediate level solution is a sub-tree in this diagram. Such a partial solution can be retrieved from the domain model and pasted into the model under construction. This imported sub-solution may need some modification. This is an effort welcomed, as opposed to wasting time reinventing the wheel. It should be recalled that after any inclusion to the diagram, the connection scheme should be revised.

The domain model will provide the developers with

- know how
- a dictionary of domain terminology
- reusable requirements models
- reusable design models
- Reusable implemented software components.

The developer's task should start with adopting chunks of requirements information from the domain, for the current development. Any requirements reuse as such also earns the team considerable amounts of design and implementation as well. When a problem is defined, its solution is also defined for a matured domain. Once again, this solution may not be exactly what is desired, but still minor modifications are better than creating a whole solution from scratch. This adopted solution piece can be as small as a single abstract COSEML element corresponding to one component. Or it could be as big as a system capability. In this latter case, the component is said to be of large granularity. The Common off the Shelf Components

(COTS) introduced earlier than the contemporary component technologies are generally of this large granularity class. As an example, the inventory capability of small business automation software could be handled by a large-grained component.

In the context of utilizing sub-solutions of various sizes, design patterns can be regarded as medium granularity components. Their abstract nature, however, is meant for representing program structures in a domain independent manner. Two issues should be remembered in this section about design patterns:

1. COSEML treats instantiated versions of design patterns as a sub-tree in the main decomposition diagram
2. Dynamic aspects of design patterns can be modeled using collaboration diagrams after their composition are defined in terms of architectural components.

The first item implies that different instantiations of a design pattern can have specific usages per domain. After such an adaptation, a new name can be given to the instantiation that is meaningful for the domain. The second item is an adoption from OO techniques, which can be useful in articulating on the relations among higher-levels of a COSEML representation. Next section investigates a suggested collaboration analysis that supports the component-oriented approach.

Interaction Analysis

UML, as the leading OO modeling approach, has been used by a wide majority of software developers while depending on the interaction diagrams heavily. Interaction diagrams are the collaboration and the sequence diagrams, both containing the same information but represented in different views. Collaboration diagrams are considered in this chapter but it is not difficult to extend the idea to the sequence diagrams.

Dynamic aspects of a model are represented in collaboration diagrams that correspond to a scenario to explain a system-level function. A sequence of messages ordered in time is the basis of the representation mechanism. The system function described verbally in the form of a scenario script will be modeled through ordered events. The events correspond to messages that actually mean conducting methods or input event processors. Eventually a scenario is enacted by the ordered execution of component functions specified in the collaboration model.

To visualize collaboration models, communicating parties besides messages have to be drawn. These parties are objects in UML, and components in COSEML. However, COSEML's nature of representing abstract as well as physical components in the same decomposition suggests here that

collaboration logic can also be applied to two different abstraction levels. The interaction modeling can be superimposed on the main decomposition view. For a better interaction model representation, the parties taking part in a collaboration can be copied from the main diagram to a specific collaboration diagram. This way the information hiding principle can be obeyed towards the understandability of the collaboration logic.

Special care is needed to construct a logical-level collaboration model. The fundamental elements of an interaction are messages that cannot be drawn in the logical (abstract) levels of a COSEML representation. At these levels there are abstract elements such as packages and abstractions of primitive specifications that can be related only through connectors. Connectors are not only abstractions, but they also represent a set of messages the physical components will use for communication purposes. For simple cases, a logical level interaction can be shown using connectors while not further specifying the messages within them. This will not yield a complete collaboration model since the sequence of the messages cannot be represented. To exploit the interaction model, messages are allowed in collaboration diagrams among abstract elements. In UML, collaboration diagrams can be incorporated in requirements modeling, implying that every kind of information is logical. Objects in UML may be considered as corresponding to our components as physical level entities. However, UML incorporates objects in both requirements and design models.

The messages to be introduced in the logical (or abstract) collaboration diagrams are also meant to be treated as logical representations of messages. Physical messages are actually requests made to the methods of existing components. The logical messages introduced in an abstract collaboration diagram may not find exact matches in the later stages where components are located: Existing components may not honor the services an abstract model suggests. This is like the basic decomposition risk of not being able to match the lowest-level abstraction elements to existing components.

An example, “sale” scenario is used in the logical and physical (run-time) collaboration models represented in Figures 6.9 and 6.10, respectively. In this example, a sale request triggers a check for the availability of the item in the inventory. If the sale operation proceeds, two other main tasks have to be carried out: recording the sale information in the accounting sub-system, and updating the inventory. After a sale, the quantity of the sold item is reduced and this final quantity value has to be recorded in the inventory to replace the former value.

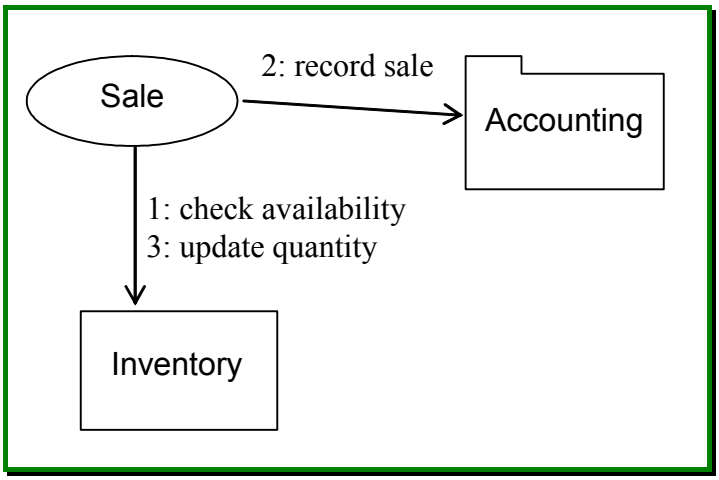


Figure 6.9. Logical level collaboration example for sale operation

Collaboration analysis is another tool to be exploited in the validation of the defined set of abstract components. Tracing the message sequences, one can confirm that system functions can be achieved. However, this validation only applies to the logical model. The natural next step after defining a decomposition of the system in the form of a set of connected abstract elements is the matching of those elements to physical components. All the requirements of the system should be represented by the set of physical components. There is no guarantee that the physical components, when located, will conform to the specifications set forth in the abstract model. This non-conformance is a problem for both locating a physical component all together, and for the available services offered by the component the instant it is located.

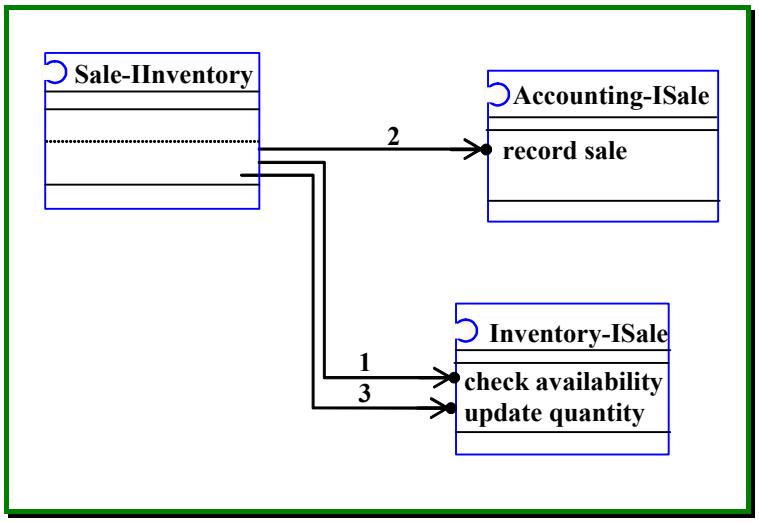


Figure 6.10. Run-time collaboration diagram

Detail Design and Implementation

Specification in the abstractions corresponds to requirements and some logical design. Now, it is time to develop the solution on a COSEML model that corresponds to executable code. For that, specifications contained in abstract elements are used to locate candidate components. In general, more lower-level abstractions will be implemented by components. A simplistic look into the implementation problem of the existing abstract tree would suggest representing only leaf-level abstractions by components. For completeness, no leaf-level abstraction should be left out also. The issue of the pluralities between the abstractions and the components such as one-to-one, one-to-many, many-to-one, or many-to-many is discussed in this section. Also the critical matching problem of the existing components to the abstractions in the specification has to be dealt with.

Plurality of mapping

The abstract half of a COSEML diagram is desired to be a tree; no lower-level element should be composed to more than one "container (parent)," at higher levels. This rule changes, however, at the physical level where a component may represent more than one abstraction. The opposite is easy to avoid, hence it should be avoided – no single abstraction should be represented by more than one component. At this level, an abstraction can be split into two abstractions that match the components through one-to-one "represents" relations. Figure 6.11 shows the desired and undesired multiplicity alternatives for the represents relation.

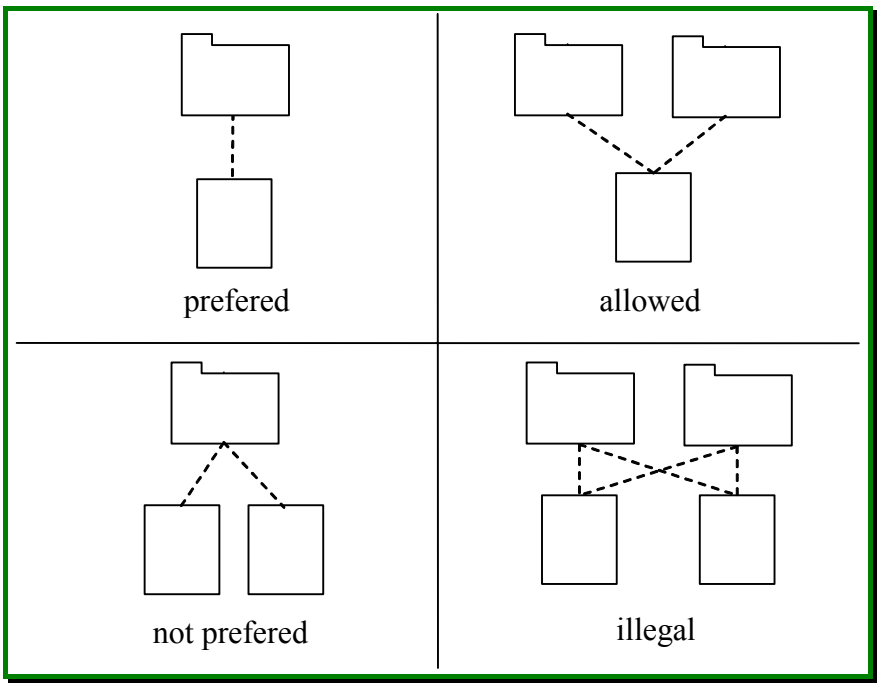


Figure 6.11. Different numbers of abstractions and components per represents relation

Partial representation of abstractions

There may be cases where an abstraction that is higher than leaf-level may be represented by a component. The meaning of this kind of a mapping is that the component is implementing all the requirements preserved in the represented abstraction that is the total requirements in the contained abstractions. However, some of the contained abstractions may be additionally represented by other components. This situation is depicted in Figure 6.12. In this case, the component representing the container abstraction is responsible for implementing the total requirements for those of the contained abstractions that are not individually represented by other components.

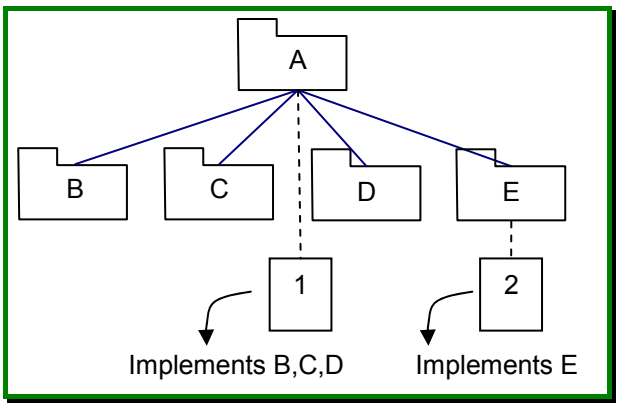


Figure 6.12. Partial representations of container abstractions

Matching abstractions to components

Implementation corresponds to the acquisition and integration of components. It is not always possible to find a set of components that are defined along the requirements. Any imperfect patch implies the revisiting of design decisions. This revision may or may not require code-level development. It is the objective to avoid code development as much as possible, according to the "Build by integration" persuasion.

The specifications in the abstract components will be used in locating candidate components. If there is a perfect match between any abstraction and any component, the component is placed in the graphical model and a "represents" link, is formed between the abstraction and the component. In case some of the abstractions cannot be realized by existing components, the design-level activity should assume one of the alternatives listed below:

1. Modifying the decomposition in the abstract levels, with the objective to arrive at the existing components.
2. Modify the component through built-in facilities (without code development) so that it can be represented by an abstraction.
3. Code-level modification of existing components.
4. Building components from scratch.

This list of alternatives should be tried in the given order. One possible option to replace the first alternative is to combine existing component into a super-component that covers the requirements modeled in the abstraction. Even though this method seems to be naïve so far, this option also brings in code-level involvement. Actually components at their physical-level are also capable of composing into super-components which themselves are components and they should obey the protocol such as exerting interfaces. The simplest of such an interface would duplicate the services etc. on the contained components and, in run-time, would act like a relay. However,

any such interface does not get created automatically with today's technology so combining components immediately requires coding. However, it is another fact that in such simplistic cases automatic interface generation should not be very difficult to realize today. The best suggestion is to try to modify the abstract levels rather than components. An abstraction split into two and the two being inter-connected through a new connector is generally easier than composing two components into a super-component.

Component Acquisition

Once industries adopt COSE compatible development, domains are expected to mature and for a majority of projects, all the components will be available off-the-shelf. Frameworks will further aid the development with graphical drag-and-drop facilities for importing a component out of a palette of alternatives, into the design under construction. There will always be cases where a specific component definition cannot be matched within the currently owned set. Then the search for a component will be executed which defines the next problem to be solved in component orientation. There are already component marketplaces available on the Internet where aided searches can be conducted. Such facilities will be improved and widened. The fundamental complexity in component searches especially from outside the developer's organization is the semantics of the query. The specification contained in the abstractions need to be represented in such a formalism that components in the market can be understood. Further, the components should assert themselves within a similar language so that some authority can decide a match between what is desired and what is offered, i.e. what is specified in the abstraction and what is defined for a component. Meanwhile hybrid development will continue for mixing legacy code with existing components and even relatively recently developed object-oriented modules.

Some semantic information added to the syntactic descriptions in the Interface definitions will support this automated search for components. Some kind of denotation semantics is already considered in research work for this purpose. Ontology is commonly offered as a basis to study a domain in an effort to represent a semantic description.

After the matching of functional specifications to component descriptions, determination of the efficiency and other quality factors will continue to be problematic for some time. One way to classify components with respect to quality parameters has been solved to a considerable extent, in the digital electronics field by the TTL technology. TTL has been very successful, owing to its low-granularity basis and the advantages of it existing in the tangible hardware domain. Different families of components are offered for different speed and power consumption requirements for TTL. Once configuring the parameters to classify quality factors and consequently producing different families of components based on varying price/quality

values, the dependability and testability of components with respect to such quality factors will continue to be an important issue to resolve.

Internet services

We define an Internet service as a component which is a part of a solution, but resides at a remote host. The messages present in OO or CO platforms will need to be physical data packages traveling across the Internet and also require a response message. Once such virtual applications can be effectively created, the electronic enterprises for software development will be on their feet. Every organization can offer what they are best at and an application can be integrated across the Internet without having to connect the components on a single machine. Locating a required service is another way of looking into component acquisition, although not every component can be replaced by a remote service. There may be security and efficiency constraints. There is also another different aspect of services; a single Internet service is not a part of a system. The same service may offer its operations to different systems simultaneously. It can work as a shared resource (component). Also, an Internet service may virtually act as a part of a system only "part-time." The important concept of a component being a structural part of a system is not really observed by the services. They do not really become an exclusive part of a system, but they only act as an interface during which a request is processed. Nevertheless, the structural decomposition of a system definition is not fundamentally affected by a unit being located as an exclusive part, or as a virtual part residing at a remote end.

Some Guidelines

The observations [Bayar 2001] suggest some methodological rules in the utilization of COSEML. These rules are mostly intuitive and they are a result of a limited exposure to academic case studies. Such rules are expected to be specified further and better after the exploitation of the approach by the industry. Like the previous development paradigms, COSE also will make more sense after practical methodologies settle, being both labor-intensive and taking considerable time. Suggestions and notes listed below outline the fundamentals of a methodological approach:

- Abstraction Level
  - Do not represent connectors among the components inside a package.
  - Only packages should be decomposed to lower-level components.
  - Only one control abstraction should be allowed per package and nowhere else.

- One Abstraction should not be represented by more than one component.

- Implementation Level
  - Components should define an interface per connection.
  - One component is allowed to represent more than one abstraction.
  - Abstract connectors may require some code writing.
  - Complexity in a connector may lead to the introduction of new components.
  - Declare outgoing methods in interfaces where it is feasible.
  - Try to only represent packages by components (avoid data, function, and control abstractions).

Some creation/modification may be inevitable especially during the early days of a component domain. Nevertheless, the process should try to refine the model with respect to the existing components, rather than creating them wherever necessary. The following additional suggestions are crucial for the efficiency of a COSE practice:

- Employ Domain Analysis
  - Developers should be familiar with the domain model.
  - Experience helps with efficiently mapping abstractions to components
- Model Refinement
  - Be familiar with previous practice in decomposition.
  - For unmatched abstractions, try a bottom-up approach:
    - Find components that represent the closest solution for those abstractions
    - Declare immediate abstractions above the selected components
    - Revise the model to accommodate the modification.
- Information hiding through selective displaying of the model
  - Include children abstractions only when more detail needs to be modeled.
  - Hide any sub-components if they contain more detail than the level desired to be observed.

Finally it should be noted that the advantages targeted in selecting this new approach will be achieved as long as the component orientation understanding is not ignored. Abusing any kind of orientation is possible. A LISP program can be constructed as a procedural one rather than the side-effect free functional style, or a C++ program can be purely procedural avoiding Object Orientation. It is also easy to bypass the philosophy of this approach, especially if no mature set of components is to be considered. The suggestions provided in this section can be adopted and expanded or modified. The development by integration paradigm should be kept in mind for the total conversion.

Testing and Integration

The classical testing strategy problems are basically retained in component orientation. However, effective testing of a component has its peculiarities. Information about how well a specific component has been tested before, especially under what circumstances, may reduce the testing costs and increase the effectiveness. Built-in test utilities have been considered also, but the drawback with this approach is the swelling of a component for an extra load that would probably not be used once the system is successfully composed.

Conclusions

The approach introduced in this chapter supports the Build by Integration paradigm. Previous and existing approaches have oriented themselves towards "code writing." Some techniques to fill the gap between the problem statement and executable code are presented within this understanding that does not involve code-level development. Component Orientation can be easily regarded to be in its infancy state, which brings together some difficulties and shortcomings. Nevertheless, demonstrating, at least theoretically, the feasibility of switching to the proposed approach is the goal of this chapter as of today. There is enough attainment to conduct experimental development. The methods have been applied in class projects and also in two commercial systems. Those systems have been developed in OO methodologies already. Therefore, the OO design models were available before these case studies. Alternative design modeling in COSEML has been conducted for these existing commercial systems.

There is however a lot to exploit among the proposed techniques, even within today's methodologies, in effect. The top-down decomposition facilities supported through tools provided in Component Oriented Software Engineering is a desired property for many developers. Some ideas are also applicable to Object Oriented or traditional models, provided that their usage within the existing approaches is explained. Once, the decomposition of the model constructs in abstraction is obtained, it is possible to substitute leaf-

level abstractions with “objects” instead of “components,” where the interfaces also should be added. The interface concept is an old one and has been interpreted through various techniques in traditional and especially OO approaches.

If the approach is utilized in an OO platform, for example, code-writing for the implementation of the specified objects will still be necessary. However, this inevitable implementation may pay off better when conducted within the guidance of Component Orientation. The modularity for the objects will be determined better and a further effective reusability level will be reached. Actually a conscious reusability will require considerable increase in the development effort, buying long-range benefits in return. The ultimate reuse consideration is facilitated today through complying with a component protocol. Developed within a protocol, the future reuse of the components will further increase and also its marketability will be captured.

Best practices develop in different fields in due course of time. Component orientation will benefit from some engineering expertise to be gained. There are peculiar problems to this new approach that will be interesting to watch as they yield rules for the engineers. One new problem to cope with is the apparent deviation from the principle of separation of definition from implementation. An efficient specification is conducted inevitably considering the implemented components during decomposition. The separation of definition from solution may be moved to an earlier activity that is domain analysis.

This problem is also closely related to the accommodation of top-down specification approach with the bottom-up nature of composition which is the destiny of the components. A developer could specify abstractions according to a logical view of the system but the outcome might not be compatible with available components. Some iteration with respect to different optimization parameters will result in modifying the logical view or the component set. This iteration will converge when specifications meet the executable building blocks. Design rules to mature in the field will increase the effectiveness of the process of development.

Eventually, it will be possible to automatically search and integrate components, once a specification is made. Before that can happen, we can still benefit from component orientations, even without utilizing components.

Questions

1. How can a requirements engineer know that sufficient detail level has been reached and therefore know it is time to stop the decomposition?
2. What is the criteria in preferring some code-level development, rather than strictly utilizing available components “as-is”?

3. Discuss if it is possible to conduct Component Oriented software development without Domain Orientation.

4. Describe how an OO environment can be used to conduct COSE development, in terms of the utilization of provided diagrams, such as the 9 diagrams provided in UML. Especially how can the visualization of a hierarchical decomposition be supported?

5. Assume there are no components available and the code needs to be developed through defining classes and objects. How can you benefit in such an OO environment from the ideas set forth in Component Orientation? Suggest the usage of an OO environment such as UML.

6. Criticize the dependability of built-in testing facilities in components.

7. A component is capable of creating its mirror image in run-time, hence creating copies of itself. Can the created copies be considered also as components within the development approach emphasis, given in this chapter?

8. What kind of measures should be taken to work with components that do not include "events" in their interfaces due to their protocol?

9. Give an example definition of a problem and its abstract decomposition where a higher-level abstraction is implemented mostly but not completely by a component and the remaining requirements in that abstraction are separately implemented by other components (those other components will implement some of the children abstractions of this partially implemented abstraction).

10. For the choice of problems presented below, first conduct a traditional model, then an OO model, and finally a COSEML model. The traditional model should include dataflow, entity-relationship, and structure chart diagrams. The OO model should include the use-case, class, and the interaction (collaboration or sequence) diagrams. Only the main decomposition diagram is sufficient for your COSEML model. The problem choices:

    a. An "ATM machine," where a user can deposit, withdraw, or transfer money.

    b. A "student registration system," where students can register, add/drop courses, and get approval from their advisor-professors.

    You can define extra specifications to the problems, just enough to be able to build meaningful models. Avoid expanding the problem.

References

Avkaroğulları, 2004 — Okan Avkaroğulları, 2004, Representing Design Patterns in Component Oriented Design, M.S. Thesis, Middle East Technical University.

Bayar, 2001 — Bayar V., 2001, A Component Oriented Process Model, M.S. Thesis, Middle East Technical University.

Booch et al. 1999 — Booch G., Rumbaugh J., Jacobson I., 1999, *The Unified Modeling Language User Guide*, Addison-Wesley.

Dogru and Altintas, 2000 — Dogru A., Altintas I., 2000, "Modeling Language for Component Oriented Software Engineering: COSEML," *The 5th World Conference on Integrated Design and Process Technologies*, Addison, Texas.

Dogru and Tanik 2003 — Dogru A., Tanik M.M., 2003, "A Process Model for Component Oriented Software Engineering," *IEEE Software*, Vol 20, No. 2, March/April, pp. 34-41.

D'Souza, D.F. and Wills 1998 — D'Souza, D.F. and Wills, A.C. *Objects, Components, and Frameworks With UML: The Catalysis Approach*. Reading, Massachusetts: Addison-Wesley, 1998.

Gamma et al. 1995 — Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, Massachusetts, 1995.

Heineman and Councill, 2001 — Heineman G.T., Councill W.T., 2001, *Component-Based Software Engineering,* Addison Wesley.

Krieger and Adler 1998 — Krieger D., Adler R.M., "The Emergence of Distributed Component Platforms", *IEEE Computer,* March, 1998.

Pressman 1997	R.S. Pressman, *Software Engineering: A Practitioner's Approach*, 4th Edition, Mc-Graw Hill, 1997.
Simon 1969	Herb A. Simon, 1969, *Sciences of the Artificial*, MIT Press, Cambridge, Massachusetts.
Salman 2002	Salman, N. "Extending object oriented metrics to components." *The 6th World Conference on Integrated Design and Process Technology*. Pasadena, California, June 23-28, 2002.
Tanik and Chan 1991	Tanik M.M., Chan E.S., 1991, *Fundamentals of Computing for Software Engineers*, Van Nostrand Reinhold, New York.
Tanik and Ertas 1997	Tanik M.M., Ertas A., 1997, "Interdisciplinary Design and Process Science: A Discourse on Scientific Method for the Integration Age," *Journal of integrated Design and Process Science,* September, Vol. 1 No. 1: pp. 76-94.

Chapter 7: Traditional Development of a Travel Reservation System

Introduction

This chapter presents a case study for developing a bus reservation system. First, some project management activity will be presented, followed by the requirements and the design modeling of the project. These models will contain a considerable amount of information, but not for the complete system due to space limitations in this book.

The example system is quite like the airline reservation system, but simpler. There are different busses with different seat layouts. Trips are identified by origination and destination locations, and date and time for the take-off. Seats can be viewed with an indication for their status (i.e. free, reserved, and bought). A reservation can involve a set of seats for a series of trips for a group of people. Tickets can also be returned.

The chapter will heavily base its model on the requirements definition. Design model will be introduced but with less content involved. The transition from the requirements to design is intended to be represented here. Also example detail models are included for the design. The information that strictly corresponds to design rather than requirements is however represented more in the design sections.

Estimation

Before the project can be contracted the developer should be able to guess the cost and the duration. This activity is among the first and very important tasks. Some problem definition is also required to arrive at a useful estimation. Since there is nothing developed yet, only by investigating the problem definitions we need to arrive at estimations. That is why the "function Point" method is used first. That requires some domain parameters to be known. These include the number of inputs, outputs, queries, files, and external interfaces. To arrive at those numbers, different system functions or entities are considered and their required numbers are added. Table 7.1 presents the calculation of the domain parameters. Some of the calculations for the entries in Table 7.1 are explained below the table.

Table 7.1. Domain parameters for Function Point calculation

Function:	reservations	trips	personnel	clients	busses	total
Inputs	0	4	14	11	2	31
Outputs	3	1	3	8	3	16
Queries	8	11	7	9	4	39
Files	1	2	2	1	2	8
interfaces						3

Reservations

The reservation function includes ticket sale and return operations as well. A trip is selected and seat locations are determined. Actually, the trip information is not entered during the reservation operation. Such information should have been entered before and the reservation operation only selects among existing trips. This is more a user query than an input. Similarly, we assume all the information entered for specifying a reservation record are merely selections among previously entered information. So there are no inputs. The number of user queries is 6:

1. add a new reservation
2. delete an existing reservation
3. list existing reservations
4. search reservations by name
5. search reservations by destination
6. search reservations by take-off time
7. sale
8. return

Actually, more queries will be conducted to complete a reservation request, such as requesting a seat-layout display. Such a query belongs to the "trips" component so we will count it there. A ticket is comprised of one output, while reports for a returned ticket and an itinerary request each will count as the other two. So, we have totally three outputs. A database table will be counted as a file. Although, it is too early to mention a database since it is an implementation issue, the logical entity corresponding to a table is a practical concept to aid in the estimation effort. There will probably be a list of reservations saved as a table or a file. The interfaces however, are more global to the system rather than belonging to any of the system functions

listed in the columns of Table 7.1. Reservation operations do not have external interfaces. Actually, the reservation results will be saved in a remote location that is central to all offices. But this interface is the Internet and it is used for other queries also. There will be a repeated usage of the same interface specification. Every office will have one of them. But the repetition does not build on the development complexity since an office (client) application will be developed once and will be deployed at many places. The server application, however, will also have the other end of such an interface. Another interface could be developed for getting the date information from a calendar application already provided in the run-time environment.

Trips

The trip information needs to be entered probably by the system administrator or someone who is responsible for setting up the information before the front office can utilize it. During this entry, the required input items are the origination and destination locations and times. We can assume there are 4 inputs related with trip list creation, since each of them can be displayed or accessed individually upon request. Otherwise, a single trip record could be counted as one input. Requesting a list of current trips is one query. Such a request may require search criteria such as the origination/destination locations, origination times, etc. All different listing options will mean a different query. Let us assume there will be four different trip-listing queries. Selecting one trip and requesting seat layout for that trip are also queries. Deleting a trip definition is another query. A list of trips will constitute a file. Also a list of locations needs to be maintained. This is another list that requires a listing, a search, delete, and insert operations each of which can be handled by queries.

Calculating the Function Points

This calculation begins with the weighting of the counts. However, some of the inputs may be simple, some average, and some complex. Based on that, different inputs should be weighted differently. The total counts shown in Table 7.1 do not provide a breakdown with respect to the complexities. Average complexities will be assumed for most of the counts. Some other inputs have complex weights: interfaces are complex. The bus related component has outputs corresponding to the detail of a single bus, and a print of the bus list, are with average complexity while the seat-layout is a very complex output. Also the input for defining the layout is complex. The file that stores the layout information is a complex file. With the information provided in this paragraph, a break-down in the count complexities can be configured. Hence the count-total value will be reached using the tabular representation of the calculations in Table 7.2.

Table 7.2. Count Total for the FP calculation

	simple		average		complex		
	count	weight	count	weight	count	weight	totals
inputs	0	3	30	4	1	6	126
outputs	0	4	15	5	1	7	82
inquiries	0	3	38	4	1	6	158
files	0	7	7	10	1	15	85
ext.interfaces	0	5	0	7	3	10	30
					count	total:	481

To continue with the FP calculation, the complexity adjustment factors should be determined and added together. Table 7.3 shows the 14 factors and the grades assigned to each of them for the problem. The total will be inserted into the "Function Points" formula:

$$FP = countTotal \times [0.65 + 0.01 \times \sum F_i]$$

$$FP = 481 \times [0.65 + 0.01 \times 51]$$

$$FP = 558$$

The project is estimated to be of 558 function Points. This value can be converted to Lines of Code. Assuming a C or Pascal language implementation, the size can be estimated as 56 KLOC. This is rather a low value for the project. We can suspect the simplistic look to the problem definition that was reflected as small numbers for domain parameters in the FP estimation.

Table 7.3. Complexity Adjustment Factors

Factor	Grade (0..5)
1 Reliable backup and recovery	5
•2 Data communications	4
•3 Distributed processing	5
•4 Critical performance	3
•5 Heavily utilized operational environment	2
•6 On-line data entry	5
•7 Input transactions over multiple screens (on-line)	3
•8 Master file updates on-line	5
•9 Complex input/output/file/inquiries	2
•10 Complex internal processing	1
•11 Reusable code design	4
•12 Conversion and installation included in design	3
•13 Multiple installations for different organizations	4
•14 Design for facilitating change and ease of use	5
Total :	51

Empirical estimations

We will continue with the size estimated above, to explore the time and cost related estimations. Since the size is known in terms of lines of code (that is 56 KLOC), COCOMO can be utilized. The basic model is used and the problem is determined as organic. Effort and project duration can be estimated as:

$$\text{Effort} = 2.4\,(56)^{1.05}$$

$$= 164 \text{ person months.}$$

$$\text{Duration} = 2.5\,(164)^{0.38}$$

$$= 18 \text{ months}$$

$$\text{Number of persons} = 164 / 18$$

$$= 9.11 \text{ persons.}$$

To be on the safe side, the rounding of the results was actually used as rounding to the nearest higher integer (ceiling). Also, the resulting staff size can be modified as 10 persons.

To estimate the cost of the project, recent project measurements can be referred. We need to know the average salaries of the personnel and multiply it with the total effort. Also, the average cost per line of code may be extrapolated to the 56 KLOC value. A round figure such as $10,000 as the monthly cost of a person to the project will yield the development-related cost of $1,640,000.

Other indications of the above numbers can also be drawn. 56 KLOC to be developed using 164 person months implies a development speed of about 15 lines per day per person. There is a little uncertainty about the total cost to the company since the parameters in consideration were only the technical dimensions of the effort. Anyway, the estimations should be adjusted with respect to factors specific to the organization.

An early prototype for investigating requirements

A throwaway prototype is a good idea in terms of screen designs to show the customer and get their feedback at early stages of the requirements work. A short sequence of screens will be presented that represent user interactions for a reservation operation. Figure 7.1 presents the main menu in this prototype with the reservation item selected.

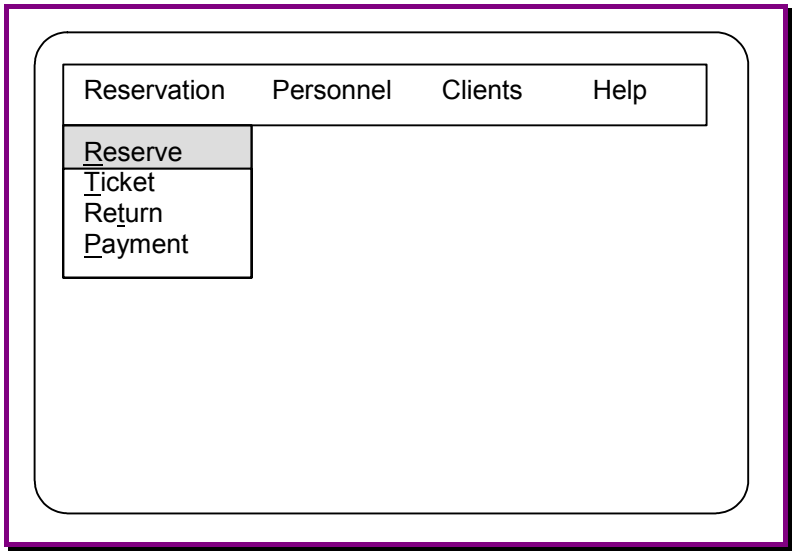


Figure 7.1. Reservation menu item

Upon selecting the reservation option, a dialog box pops up, that handles various data to be placed on a reservation record. This includes the trip with its origination and destination locations, take-off time, seat number, and information required for the organization such as the sales persons

identification. Figure 7.2 depicts the initial look of the reservations dialog box.

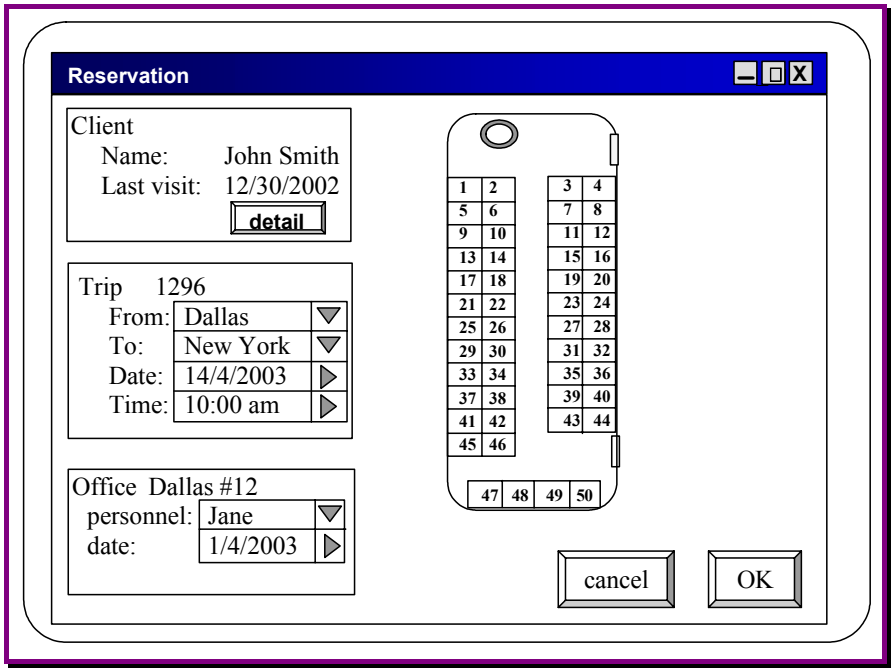


Figure 7.2. Reservation interaction screen

The idea is to show the customer how the screens will change in response to the user actions. Other screens called as a result of, for example, selecting a seat on the layout in Figure 7.2 through a mouse click, should also be included and demonstrated to the customer.

Requirements Analysis

After the investigation of the customer-supplied problem definition, interviews, and other means for requirements elicitation, some knowledge has already been accumulated within the developer team. Now, it is time to represent this knowledge in terms of a requirements model and a specification document. The first model to build is, the “Dataflow model”. The initial diagram to draw is the context diagram (level 0). Here, the external entities and the data flows between them and the system are represented. We are assuming a customer and an administrator as external entities interacting with the system. Figure 7.3 depicts the context diagram.

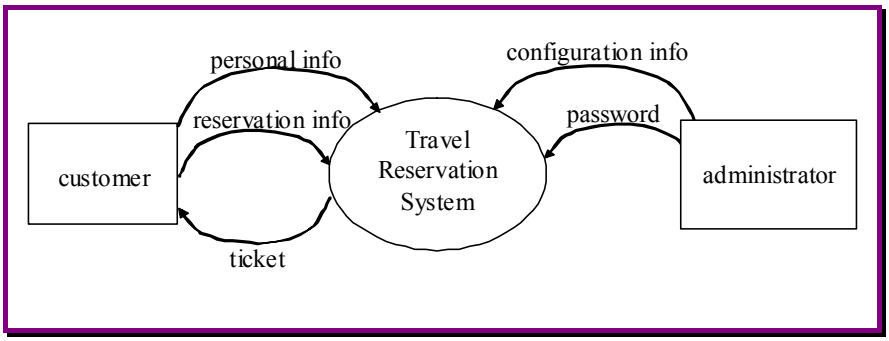


Figure 7.3. Level 0 DFD for the travel reservation system

The context diagram will be instrumental in analyzing the external interfaces of the system. Immediately after the context diagram, an overview diagram will be drawn, showing the top-level processes inside the system. Figure 7.4 presents the level 1 DFD for the reservation system. At level 1, the authentication process compares the administrator's input with the records to check if the correct password is entered. Based on the outcome, the system should allow the admin to continue with using the system. Actually, this outcome cannot be observed in the diagram since a dataflow diagram does not include control information. Since the go or no-go command out of this process will *enable* other functions, it is considered to be a control flow. The trip management process is responsible for maintaining the lists of busses, locations, and trips and providing such information to the reservation process. The personnel management process maintains the personnel list and provides the ID of the person who does the sale or reservation. The clients-processing oval in the diagram represents the operations related with the list of customers, including the sending of the current client to the reservation process. Finally, the reservation process conducts the travel reservations, receiving information from various processes and producing a ticket. Some detail about the initial description may not be observed in the model presented thus far. For example, the displaying of the seat layout is not visible yet. It is an option to model an output device such as a video display and present the layout information as a dataflow sent to this display. Another way would be to send the layout information directly to the user. The developer for the case study prefers to model the seat layout processing as an internal task of the processes so far presented in the initial stages of the DFD diagrams.

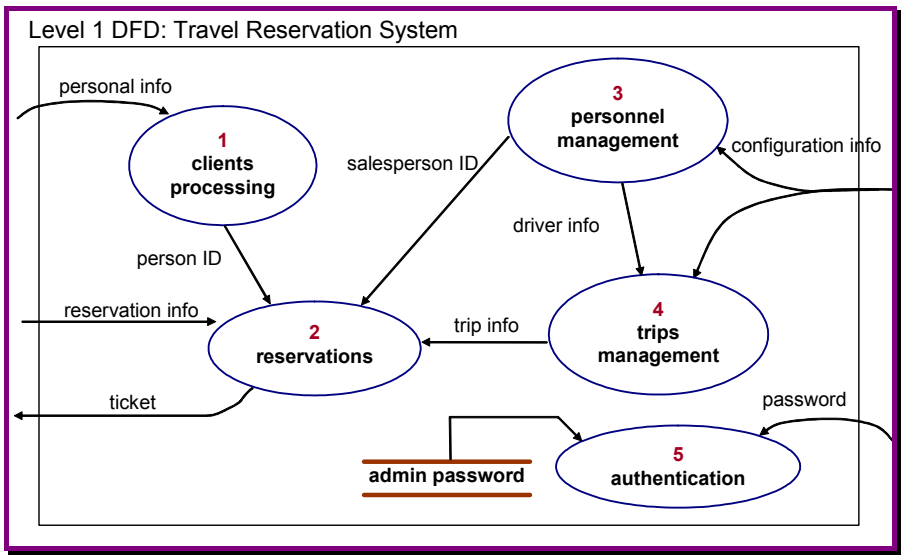


Figure 7.4. Level 1 DFD for the travel reservation system

The DFD analysis continues with different DFDs drawn for the processes defined in level 1. The reservation and trip management processes seem to be the most complex of all the processes in Figure 7.4 so they will be explored in upcoming DFDs. Figure 7.5 depicts the Level 2 diagram showing the internal details of the reservation process. Take a look at the numbering of the processes; the title for the DFD in Figure 7.5 will include the process number, as well as its name.

Figure 7.6 presents the level 2 DFD for the trip management process. This process is responsible for storing the trip records, maintaining the lists for busses and locations. The drivers are also needed in this process, but are handled in the personnel process. Therefore, the driver information needs to be retrieved from the personnel process for associating a driver with a trip.

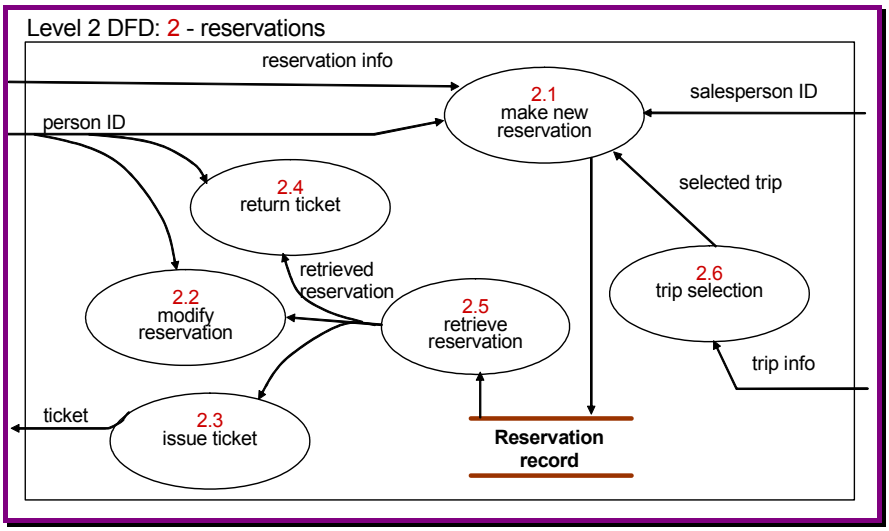


Figure 7.5. Level 2 DFD for the reservations process

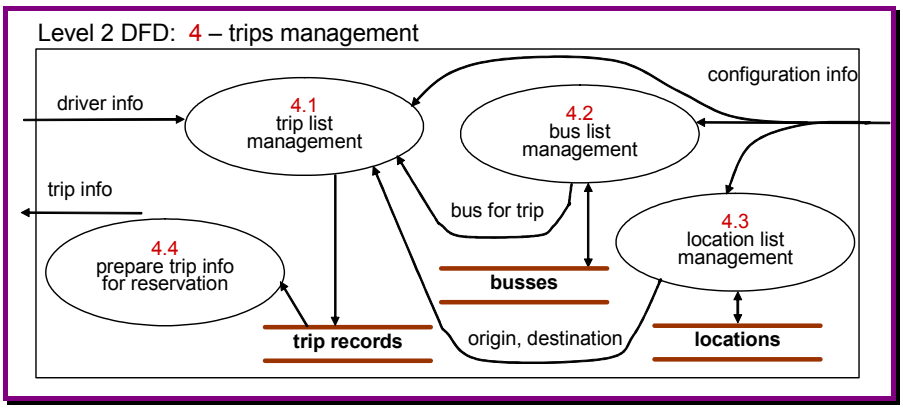


Figure 7.6. Level 2 DFD for the trips management process

Finally a level 3 DFD will be included in this example solution. The most complex process to pick seems to be the "make new reservation (2.1)" process. Figure 7.7 depicts its DFD.

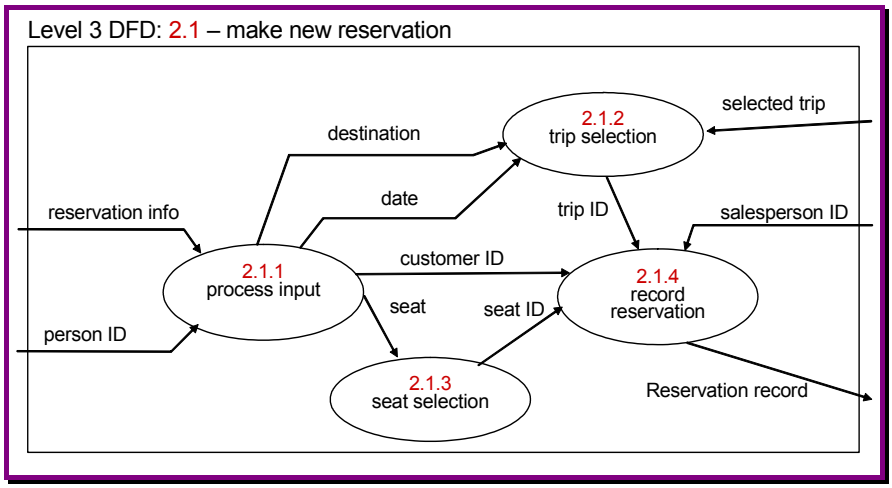


Figure 7.7. Level 3 DFD for the make new reservation process

Entity Relationship diagrams

This stage will produce a data model that is logical. The main entities in the system will be identified with their important attributes and relations among different entities will be specified. So far, reservations, customers, trips, busses, and location are very good candidates to be modeled as entities. Figure 7.8 displays the Entity Relationship Diagram, (ERD) for the travel reservation system. This single diagram accounts for all major entities and is the global model for the whole system.

Concluding the requirements model

A data dictionary would accompany the DFD and the ERD models, which is not a very complicated item for the case study. Also, a control specification including control flow diagrams for state machines could be included which is not necessary in this example.

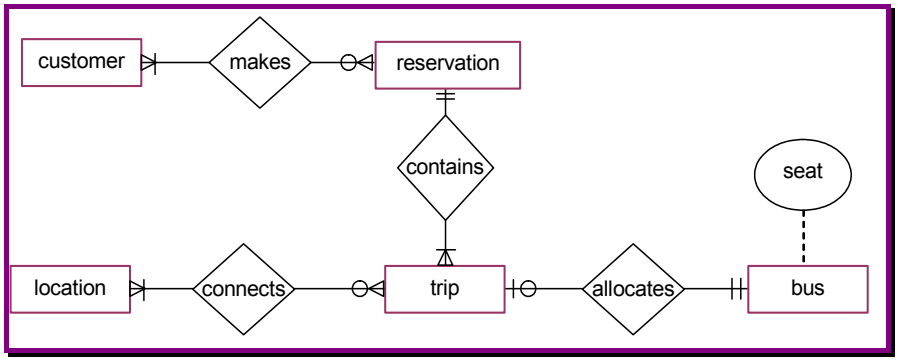


Figure 7.8. ERD for the system

Design

The requirements model will be used to start. DFDs will be revised and refined. The refinement may include solution related information. The first categorically design activity will be the specification of the data structures. Heavily, it is the relational database. Anyway, this chapter will introduce the first-cut design approach. It should be noted that a complete design is not contained, nor an optimally correct one. After the presented models, the designers should revise and refine the design until they are satisfied.

Data Design

Tables that correspond to the entities in the ERD will be used along with some extensions. There may be other local data structures required such as lists, tables, and queues. Figure 7.9 displays the table design that considers some normalization. Also, some intermediary tables to represent a "many-to-many" relation are included.

Some information is repeated in different tables. This is not desired in terms of ideal data modeling. However, for avoiding join operations in favor of efficiency, some violation of normalization rules can be employed. One example for such violation is the repetition of the amount filed in reservation and in payment tables. The same applies for the "seat number" field repeated in both the seat and the trip seat tables. Whenever a reservation is made, the customer may want to know the price before buying the ticket. During the customer's booking process, the reservation records will be revisited many times and for each visit, the amount would be accessed through the ticket and the payment tables. If the frequently accessed amount is provided within the reservation table, the other two tables are not required for checking the price. Also, at the time of the purchase, there may be some discount which can be taken into consideration. In this case, the amounts in the reservation and in the payment tables are different variables.

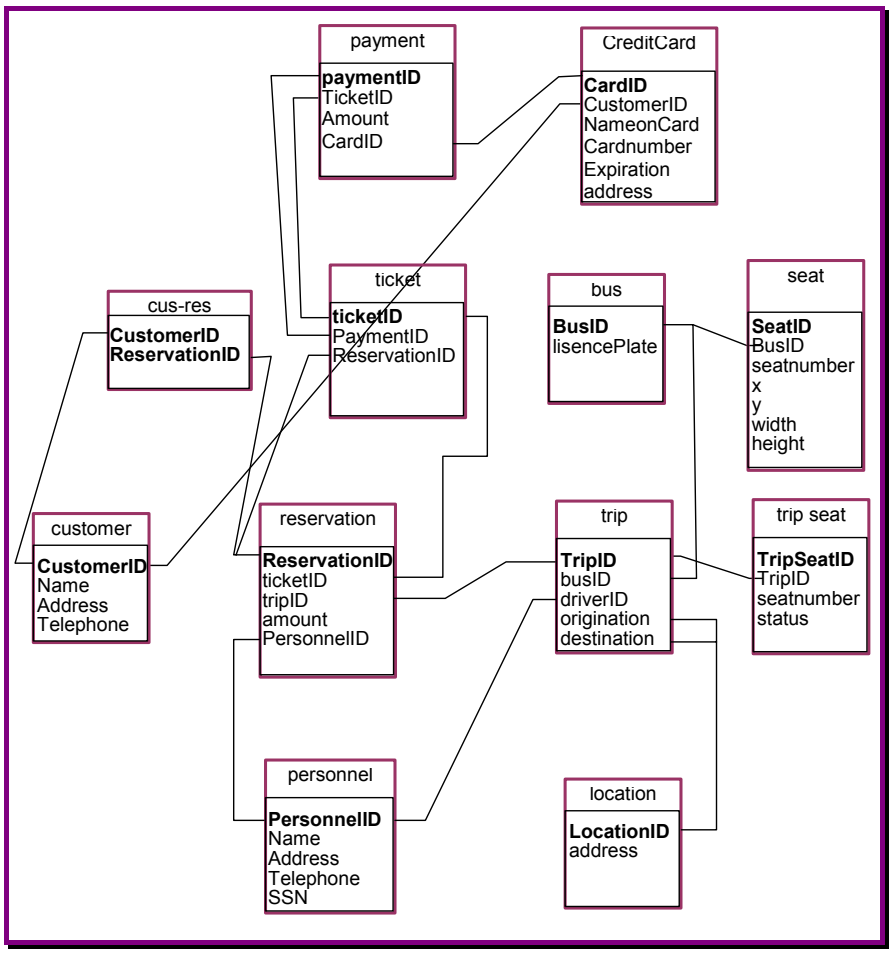


Figure 7.9. Tables and relations

Let us assume that a customer will be given alternatives once the origin and destination is specified and there could be connecting busses over different intermediate locations. This requirement can be implemented by allocating different reservations per alternative. However, to avoid occupying seats for a temporary decision process, the alternatives may be saved in the local memory. A two-dimensional linked-list could be a good structure to achieve this objective. A list will dedicate an entry per alternative reservation. All the entries in this list are actually themselves linked-lists of travel segments with durations. Figure 7.10 depicts a graphical representation of the aforementioned data structure.

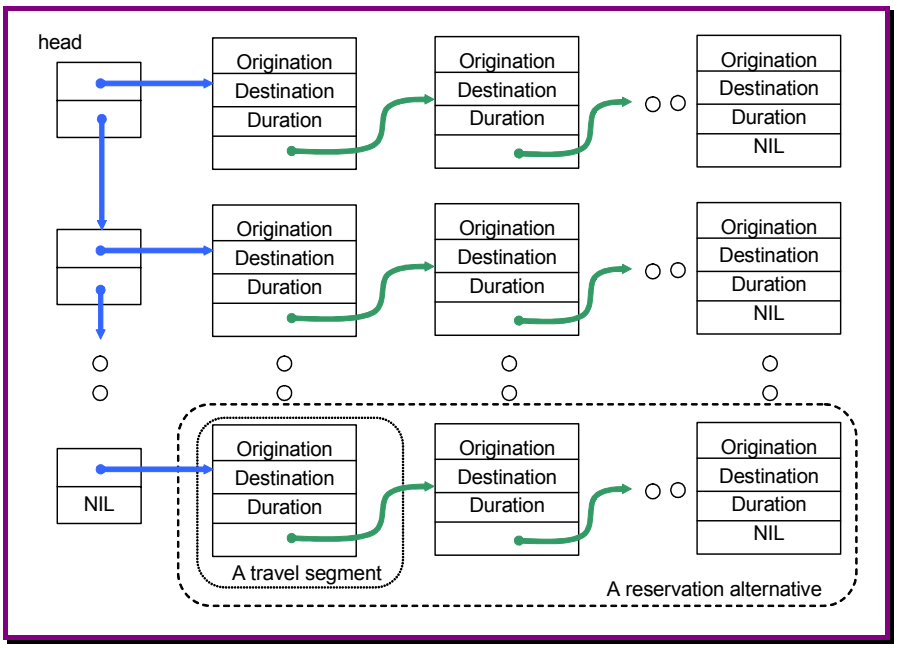


Figure 7.10. Linked lists for the reservation alternatives.

Refining the dataflow diagrams

Existing DFDs can be modified and enhanced with the implementation-related information, or lower-level DFDs can be drawn that describe the designer's solution to the sub-problem. Now that the data structures are more tangible, the data-flows corresponding to a process can be presented in further detail. Figure 7.11 contains the design-level information for the refinement of the "seat selection" process (Process 2.1.3) presented in Figure 7.7. Now we discover that Figure 7.7 needs to be modified: To select a seat, the process requires trip information. The trip information also contains the bus information that is required for displaying the seats.

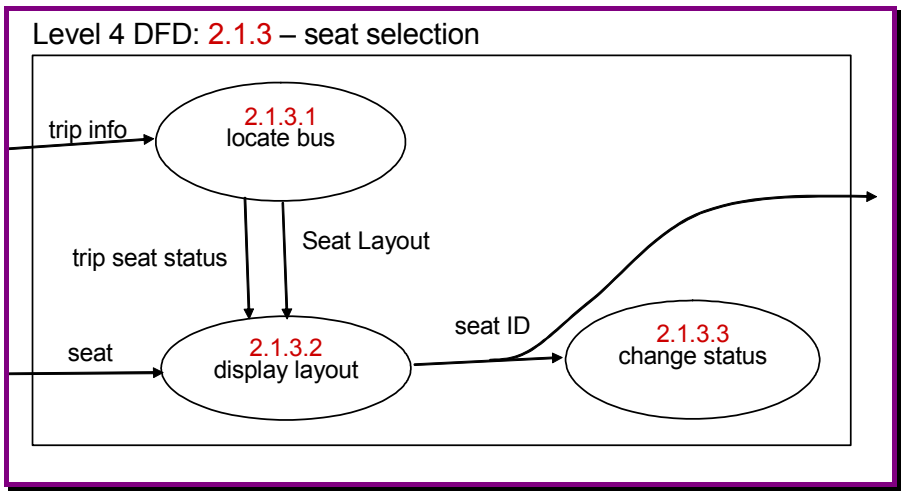


Figure 7.11. Design-level refinement for the seat selection process

More refinement could be conducted for the presented set of DFDs. After the completion of the refinements over the DFDs, the design can proceed towards structure charts.

Structural design

Transform analysis will be conducted to draw structure charts, using the information in the DFDs. The hierarchical organization of the DFDs will be observed and the structure chart will be attempted to be drawn in a corresponding top-down order. Looking at the overview diagram in Figure 7.4, it is possible to conclude that we need to join the level 2 diagrams in order to have a meaningful set of processes for separating the input, transform, and output flow boundaries. Actually, observing the overview diagram more carefully, one can deduct that there is a hidden "transaction flow" rather than a transform flow in Figure 7.4. A process standing for the transaction center, however, is missing. The structure chart can make up for the missing process and can suggest that apart from the input flow, the boundaries correspond to different action paths corresponding to reservations, personnel management, and the trip management. Figure 7.12 depicts the first step in drawing the structure chart that utilizes the model in the overview diagram only.

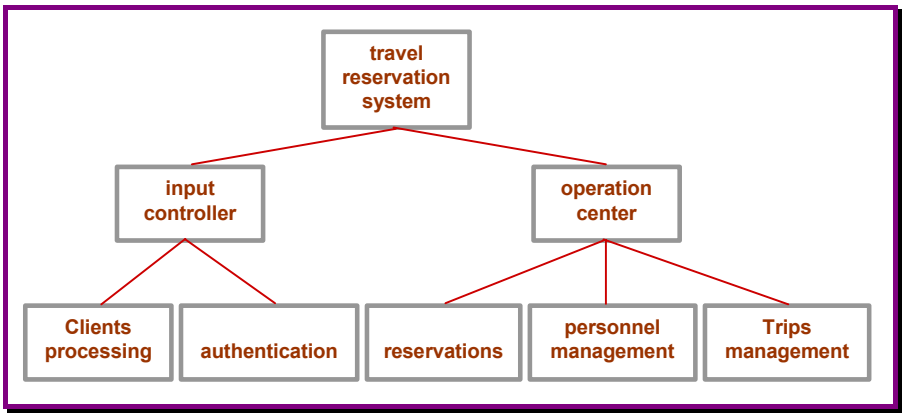


Figure 7.12. Design-level refinement for the seat selection process

Figure 7.5, that explodes the reservation process, can be interpreted as a transform flow. The input region is decided to contain two processes: trip selection and retrieve reservation. The only process in the output flow region is "issue ticket." The Level 2 diagram for the reservation process is redrawn in Figure 7.13 to illustrate the separated flow boundaries. In light of the flow regions, the processes can be organized under the control units that decompose the reservation responsibilities. To show the continuity of the design process, Figure 7.12 will be refined in Figure 7.14, with the mentioned reservation related transform mapping. In order to achieve the following refinement, newly introduced structure chart modules will be presented in separate diagrams. The whole structure chart will be presented in the upcoming section, with small modules that cannot accommodate names due to the limitation in the dimensions of this page.

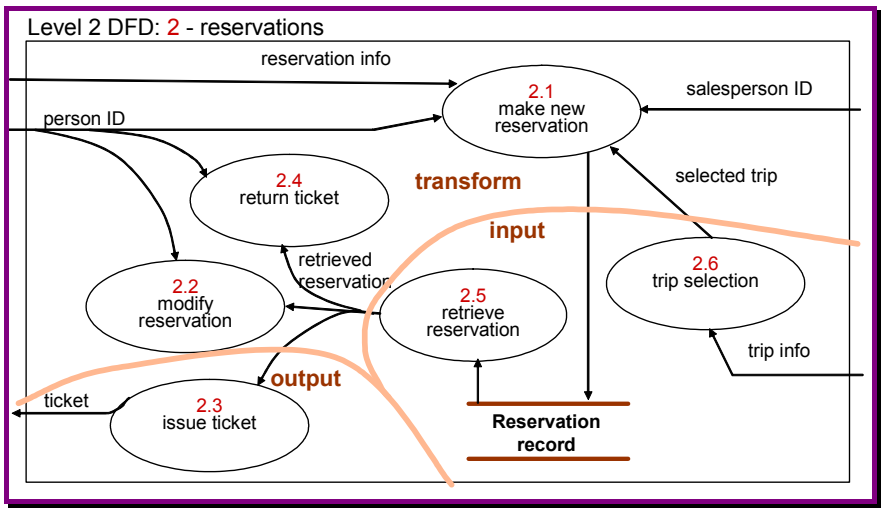


Figure 7.13. Flow boundaries in the Level 2 DFD for the reservations process

After applying the rules to construct a structure chart, a designer should consider modifications. For example, in Figure 7.14, the "reservation output control" module controls only one module which is "issue ticket." The control module can be eliminated and the "issue ticket" module can rise to its level.

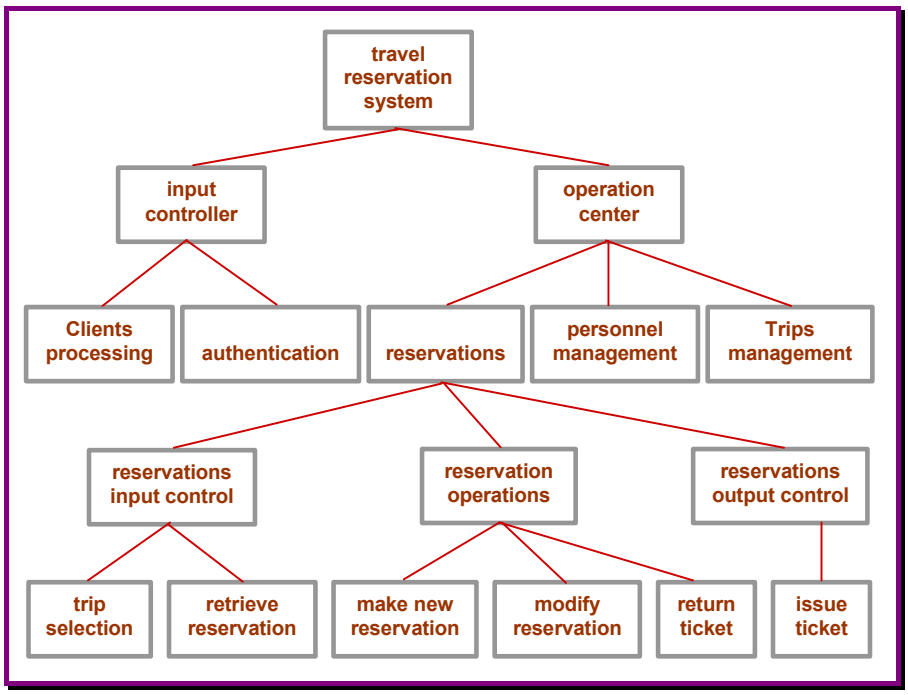


Figure 7.14. Structure chart refined for the reservation items.

The other level-2 diagram was presented for "trip management." The next refinement on the structure chart will govern the associated processes under the trip management module. Required processes are contained in Figure 7.6. Trying to separate the flow regions, it can be observed that there is a missing "input region" process. Hence, it is not possible to separate the input flow. The "configuration information" entering this DFD, is split towards three different processes without any processing. Perhaps, it is a good idea to revisit the level-2 diagram for trip management. Figure 7.15 introduces a new process to the modified version of the mentioned DFD, which is "admin interaction" that processes the input to the trip management DFD. Now, we are ready to draw the structure chart section corresponding to the "trip management" process.

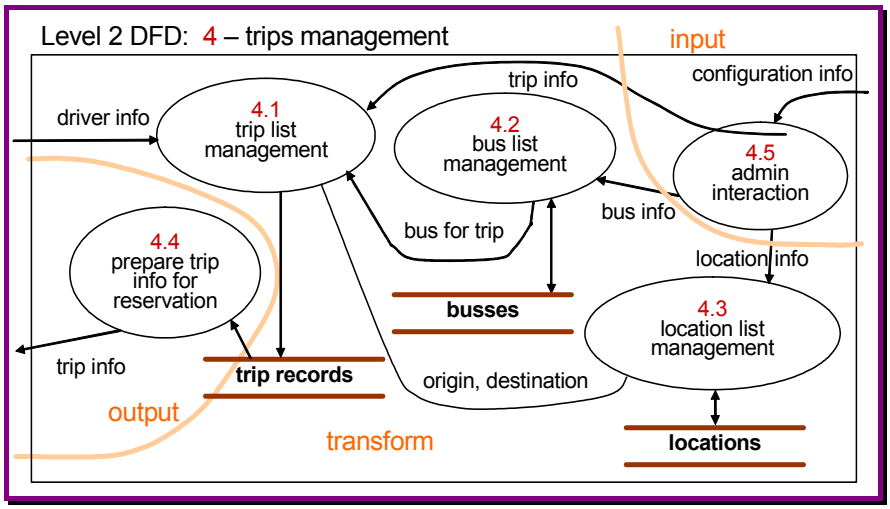


Figure 7.15. Modified Level 2 DFD for the trips management process with flow boundaries

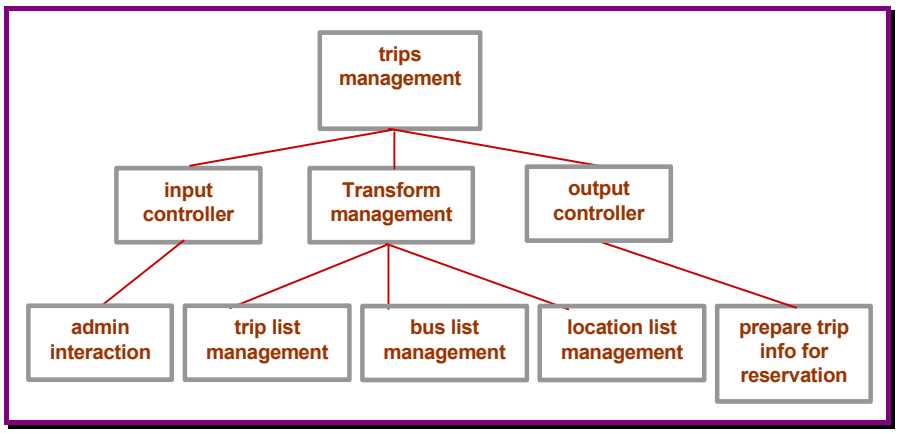


Figure 7.16. Structure chart section corresponding to trip management

Once again, there are controller-units that only manage one module, which may make them unnecessary. When all the sections will be split into units as the system structure chart, such modifications can be considered. The case study will continue with the definition of the structural sections corresponding to the dataflow diagrams that have been depicted. We need to create a new reservation process, and the seat selection processes are the only ones left, which have not been converted to structure charts. Figure 7.17 depicts the structure chart corresponding to Figure 7.7 for "make new reservation."

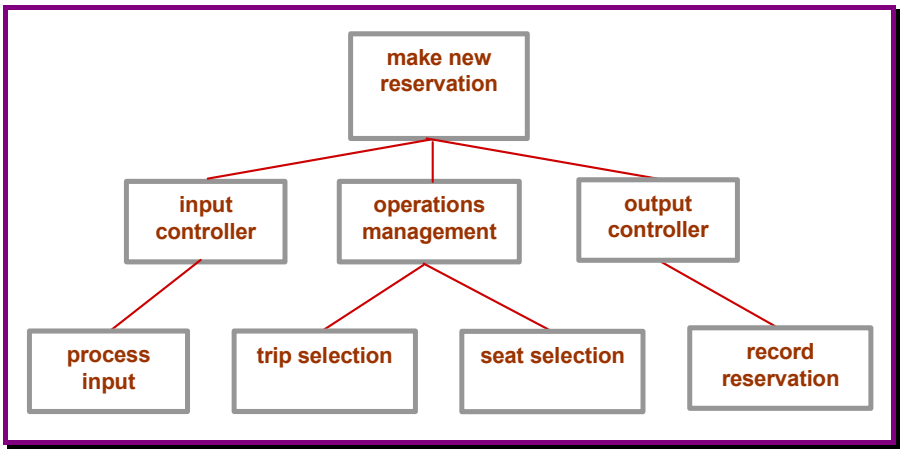


Figure 7.17. Structure chart section corresponding to make new reservation

Finally, Figure 7.18 will display the structure chart for the seat selection process, where no control blocks are introduced. With the given dataflow model this is as far as we can go. Figure 18 is a combined structure chart for the current design refinement level.

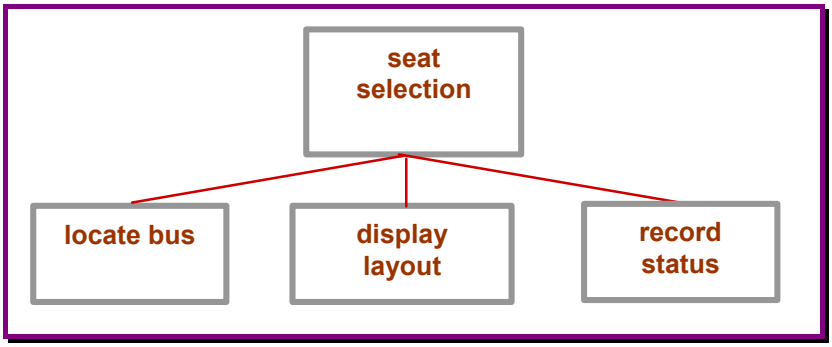


Figure 7.18. Structure chart section corresponding to seat selection

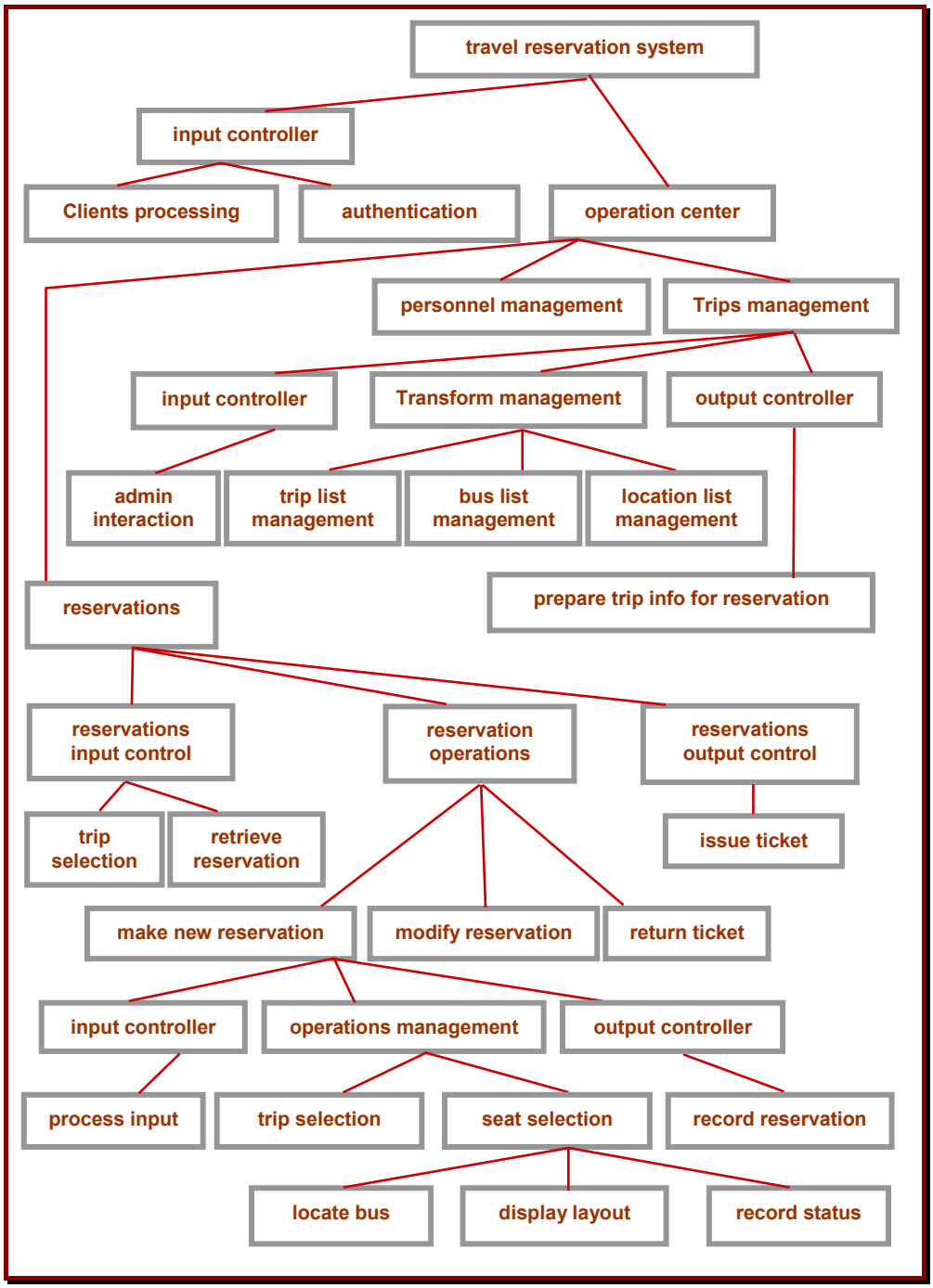


Figure 7.19. Structure chart refined for the reservation item

Chapter 8: Object Oriented Development of a Travel Reservation System

Introduction

This chapter presents a case study for developing a "Bus reservation system". The example system is quite like the "Airline reservation system", but simpler. There are different busses with different seat layouts. Trips are identified by origination and destination locations, and date and time for the take-off. Seats can be viewed with an indication for their status (i.e. free, reserved and bought). A reservation can involve a set of seats for a series of trips for a group of people. Tickets can also be returned.

The chapter will heavily base its model on the requirements definition. Design model will be introduced but with less emphasis. The transition from the requirements to design is intended to be represented here. Also, detailed example models are included for the process of design. The information that strictly corresponds to design rather than requirements is, however, represented more in the design sections.

More specifications

The system should be capable of maintaining a customer database. Personal information will be accompanied with other details to aid the marketing efforts. Birth-date cards can be mailed to the clients and their travel patterns can be analyzed to offer deals that are likely to be of interest to some of the customers. Also, the frequency of their purchases can give clues about the right time to contact them to get their opinion about any dissatisfying issues. Of course, the example can easily be grown into detailed levels surpassing those existing in the commercial applications. Credit card information for example, can extend their personal information. This should suffice for the customers, for now.

A similar list should be maintained for the personnel. Besides the personal information, the employment related information is required for any employee. The list has office personnel and drivers. Their employment dates and salaries should be part of the record. Depending on sales an office person does or the amount of trip activities for a driver, monthly salaries can be adjusted.

The trips are important for the application. A trip with its trip number has a lot of meaning for the travel domain. A trip comes with its number, origination and destination locations, take-off and arrival dates with times. This statement suggests that there needs to be lists of locations and trips that should be conveniently accessed and sometimes modified by the office

personnel. Not required by the customer, the developers can reserve facilities in their assumptions for future enhancements such as automatically detecting and recording the arrival or departure of a bus related with a trip so that customers can query such information through phone or the Internet.

Also, the system should maintain a list of busses. There will be different makes and types of busses with different seat layouts, all displayed clearly on interactive screens. If a new bus model arrives, the office personnel should be able to easily define the information, including the seat layout about the new bus.

Another important concept is reservation. The reservation information is the main information utilized in the purchase or in the event of the return of a ticket. Although the reservation information can be implicitly accessed through viewing the seats on the busses assigned for the trips, the users may want to treat reservations as important records that can be searched with respect to different parameters such as the customer name or trip date, etc., which can be listed, selected, modified, and printed. A reservation links the customers with the seats in trips.

There will be many offices distributed around the world. Every office should be able to access reservation information online. A central location is proposed to keep the main records. For the initial application, the office personnel will carry out the reservations on their desk computers. A walk-in customer or someone on the phone will be aided by the personnel. Therefore, we can assume that the customer is conducting a reservation.

Although, the problem may seem to be easy and well defined at this point. For a realistic application, so far a very little part of the definition has been expressed. This much however, is enough to start the limited case study activities. It should be remembered that possible effort should be spent on defining the problem before any further development begins. Depending on the problem, part of the requirements could be enhanced enough to start design while other parts of the problem are yet being defined through the requirements elicitation activity.

Starting with the requirements modeling

We use UML for modeling language. Use case diagrams are suggested to begin with. If the application is going to be a complex one with different capabilities, a different use-case diagram can be drawn per capability. The heart of the application is the reservation capability. Figure 8.1 displays the use-case diagram for the reservation capability. The only actor in this diagram is the customer. The customer interacts with the reservation function, purchase, and return functions. The purchase and return functions also need the reservation function, at least to locate the reservation to be bought or returned. Any reservation operations need to select a trip and view

the seat status. All the relations are compulsory. Therefore, the default "uses" relation is utilized without having to name the arrows. If, for example a return could be conducted without having to consult to the reservation function, the arrow could be named as "extends." That would imply an optional reference, where the reservation function could sometimes be requested.

How detailed the ovals could be is a matter of defining the activities as system functions. If, for example, viewing the locations can be considered as a system function and also selecting a trip may need to view the locations, and then in Figure 8.1, the "trip selection" use-case could connect to an additional use-case to be named as "view locations" through an "extends" connection.

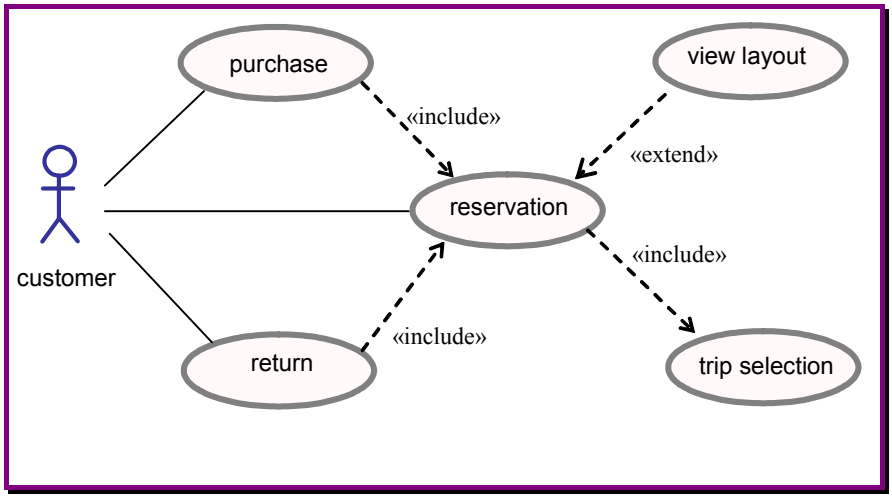


Figure 8.1. Reservation Capability

A system function is a high-level function immediately within the scope of system capability. All the system functions must be included in some use-case diagram. How important this is, is again a decision of the requirements task force. One strategy we try to avoid here is a hierarchical decomposition of use-case diagrams. So, for example to show what is going on in a trip selection activity should not be presented through another use-case diagram that defines the internals of the "trip selection" use-case. There are approaches though that allow for at least two levels of use-case diagram hierarchy.

The detailed explanation of how a system function represented by one use-case is achieved through interaction diagrams. At this point, there is the choice of breadth-first or depth-first refinement. First, all the use-case diagrams can be drawn, or the interaction diagrams corresponding to one use-case diagram can be finished before moving to the next use-case

diagram. The next activity in the case study will be the modeling of the interactions, which are first textually defined as "scenarios". Then a collaboration diagram will be presented for the interaction. The scenario for the "purchase system" function is described below:

1. Customer is identified
2. If not existing, customer information is entered
3. If not existing, a new reservation is made
4. Reservation is modified
5. A payment option is selected
6. Transaction is completed
7. Ticket is issued

Messages need to represent the actions in the scenario. A message will be constructed considering its sender and receiver objects. This is the first time in the case study an object will be explicitly introduced. Actually, actors in the use case diagrams are also objects that can take place in the interaction diagrams. Also their class definitions will be included in the class diagrams. The following paragraphs will define new messages corresponding to the scenario steps, and objects that will contribute to the interaction diagram.

1. Customer is identified: This action requires the locating of a specific customer hinting that there should be a list of customers. This list is our first object candidate. Also, when found, the information about a single customer can be another good candidate. Any message requires a sender who is the initiator of the action. The client-server paradigm is a practical guidance when determining the sender and the receiver of the message. The initiator is the client, requesting a service. The other object will assume the role of a server (only for the processing of this message, the same object may act like a client for other messages). The entity to start the search needs to be identified; this is the client object for the search operation. In our application, this object could be an office clerk or a customer. If we assume the customer as the client, we may end up with the same kind of an object trying to locate the same kind of an object, which is not a problem. Actually, the user-interface items appearing on the screen relate to a customer object that contains the service request in the form of "buttons" or something similar. Also, the same customer will have information the mentioned search operation is trying to match. The two objects do not have to be the same objects, only they are of the same type (class). The operation in the scenario/step can be further detailed. More than once, messages might do the

job. Also, the search could be carried out with respect to the customer name or the telephone number. All such decisions are subject to refining the requirements. More information may need to be polled from the customers of the project or future users. Figure 8.2 presents the first step in building the collaboration diagram corresponding to the purchase use-case.

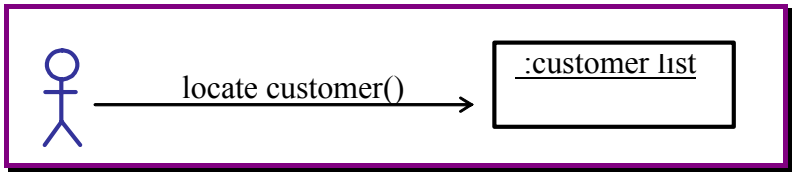


Figure 8.2. Constructing the first message for the "identify customer" action in the scenario

A good way to indicate the search by name is to include the word "name" as a parameter sent with the message. In that case, the word "name" should appear in parenthesis. Whereas, the "locate customer" message in Figure 8.1 does not carry any parameter.

2. If information does not exist, customer information is entered. This action is an example of conditional operations in the collaboration models. The condition is treated as a guard that allows the transaction of the message. Guards in UML are written in angled braces before the name of the message. Actually, the new customer creation and its information entry may take two messages. They both should be guarded by the same condition. Such an "if" block can be displayed in a more visual manner through a sequence diagram. In a collaboration diagram, the guard can be repeated in the related messages or shown once in the initial message for the conditional group. Figure 8.3 presents the messages related to creating a new customer and entering related information.

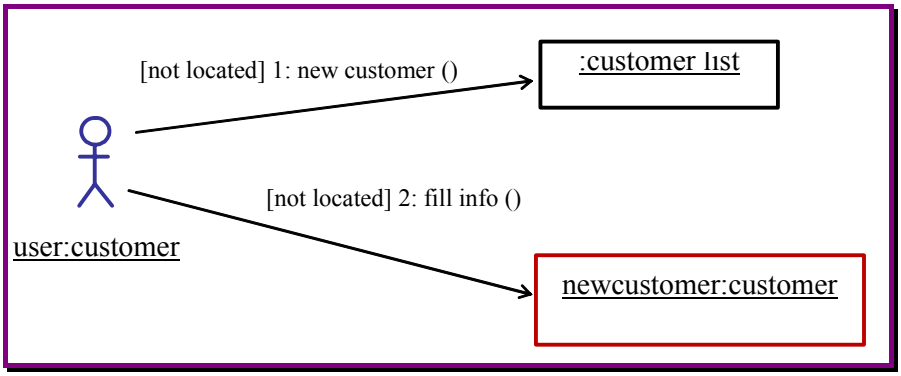


Figure 8.3. Creating a new customer and filling it with information

3. If information does not exist, a new reservation is made

4. Reservation is modified: The modification is actually optional. An existing reservation can just be viewed in a modification window and accepted. If a new reservation is being made, the modification is really conducted and any modification needs to be saved. Actually, we do not want to show the details of a reservation operation since it is represented as a different oval in the use-case diagram. This collaboration (purchase) is like referring to a next collaboration (reservation) diagram. Now, it is the collaboration diagrams acting in a client/server mode. One requests a service from the other. Although we want to analyze interaction models in isolation, there are cases where an object participating in a different collaboration diagram provides the initiation. The initiating object might also participate in the current collaboration diagram. Especially, if this client object is totally external to the collaboration, it might be considered to be an "actor" for the current collaboration without physically being represented as an actor (unless it is already an actor kind of an object). The purchase operations are using the reservation operations. Therefore, it is better to converge all reservation operations into a single request that will be carried out in the reservation use-case (or the corresponding collaboration). Our collaboration diagram will generate the request so that an object relating to the reservation operation (to impersonate an actor) is pertained in the server mode. It would be best to observe the call to reservations, in the whole collaboration diagram for the purchase system-function. One point to be noted is that no object is owned by collaboration diagrams. A specific object can participate in different diagrams.

5. A payment option is selected

6. Transaction is completed: These steps in the scenario can again be refined with the definition of the payment types such as cash, check, and credit. Completing the transaction could be different per payment type and may be conducted as internal operations for objects corresponding to these payment types. Figure 8.4 depicts the collaboration logic for the payment operation. The messages 3.a, 3.b, and 3.c are the first examples of reflexive messages in the case study. These messages originate and terminate at the same object.

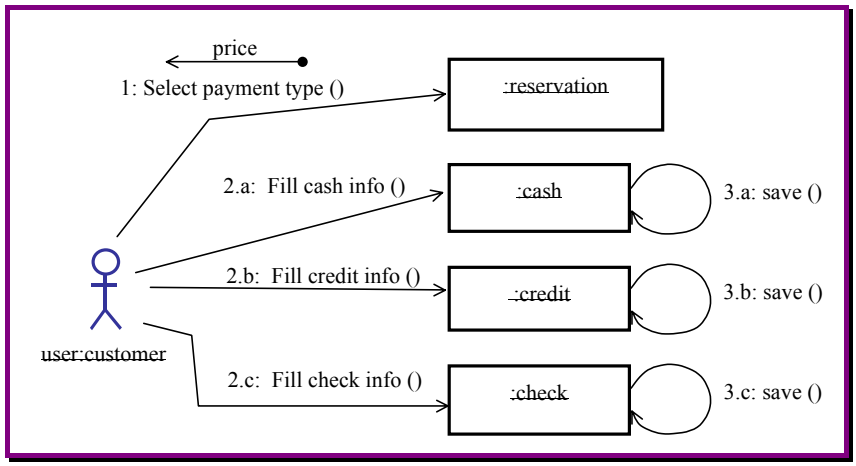


Figure 8.4. Payment scenario

Most of the purchase operation is defined through partial collaboration diagrams presented in figures 2 through 4. The complete diagram can now be drawn for the "purchase system". Figure 8.5 depicts the collaboration diagram. Here, the messages are re-numbered with respect to the complete collaboration model. Also, the flow of this case study assumes decision changes. The three different objects corresponding to different payment types in Figure 8.3 are later decided unnecessary and a single object "payment" does the job in Figure 8.5.

After the construction of the first collaboration diagram, it may be a good time to start defining the participating objects in a class diagram. Later the diagram can be grown with the addition of other classes defining the objects to be used in the other collaboration diagrams.

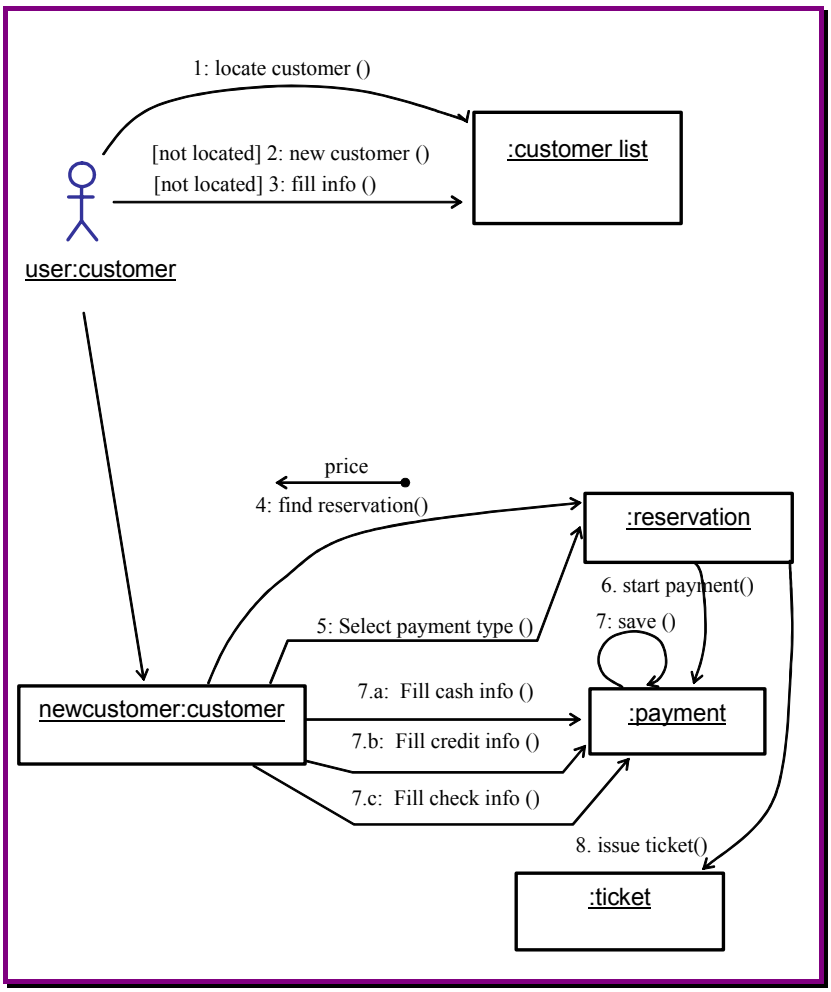


Figure 8.5. Purchase scenario

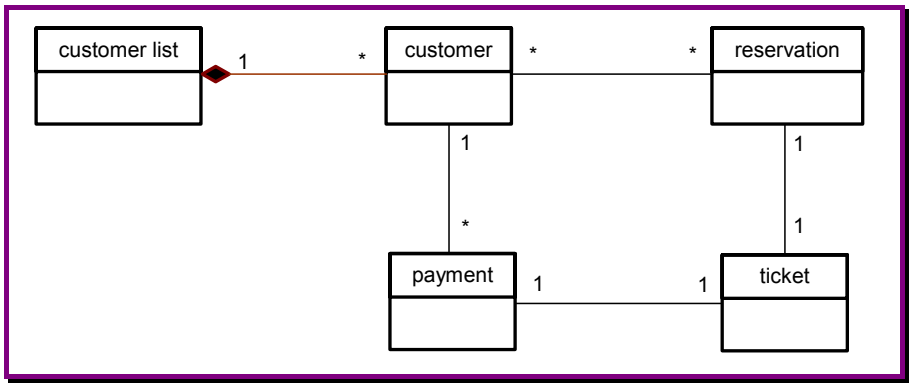


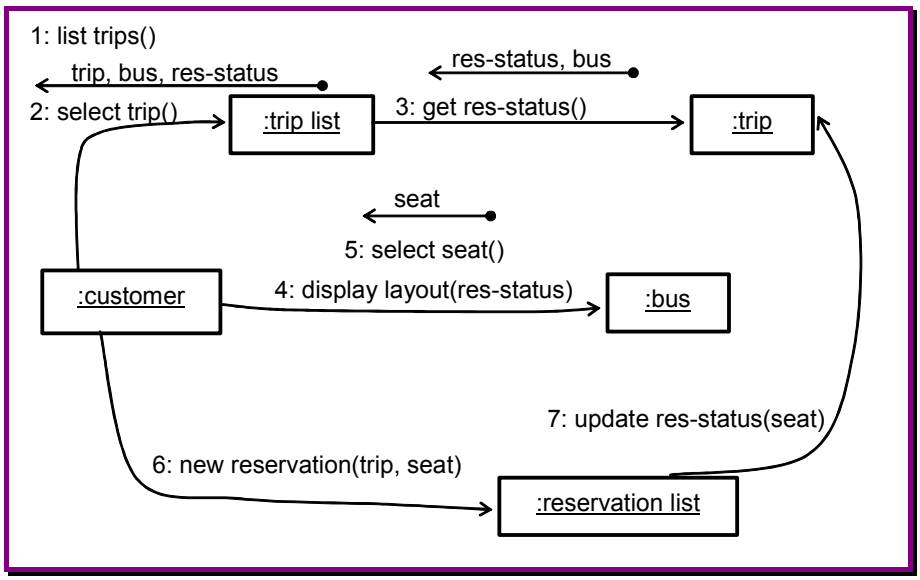
Figure 8.6. Initial class diagram

The class diagram in Figure 8.6 does not include any inheritance relation. A composition is present, indicating that "one" customer list contains many customer objects. The other relations are simple associations. Names of the associations are not given because they would not add to the understandability of the diagram. The pluralities are included such as the ones on the association between the customer and the reservation classes. This association has a "many-to-many" relation. To discuss these pluralities, the customer class can be taken first and the question of "how many reservations a customer can be related to?" can be answered. Since a customer can have many reservations at the same time, the plurality at the reservation end is a "*". Likewise, a single reservation can correspond to a set of customers, such as in a group travel. This suggests that the customer end of the relation also has a "*" kind of plurality.

A specific reservation can correspond to only one ticket object. Also, a specific ticket is for only a single reservation. So the association between the reservation and the ticket classes is "one-to-one." A customer can relate to many payments, if different reservations and corresponding tickets are bought by the same customer, while one payment can only be conducted by one customer. We also assume that one payment can only be for a single ticket and a single ticket can only be paid by a specific payment. These assumptions also indicate some constraints on the requirements. This problem could be defined in a different way. For example, it sounds natural to allow a payment to pay for more than one ticket at a time. Also, a payment could correspond to more than one customer if this is related to a group reservation. Our example model suggests that even if the reservation were for a group, only one person can make the payment.

Reservation system-function

The analysis activity continues with another use case presented in Figure 8.1. This oval was provided as part of modeling the reservation related capability. Later there will be other use-case diagrams drawn for other capabilities (such as "trip management") and their ovals will also be explained by interaction diagrams. Although some of the reservation services were utilized in the purchase activities explained above, their internal explanations might have been missing. Figure 8.7 presents the interaction logic for the activities related to reservations. It may be easier to feel the flow of operations, first ignoring the data-flows in the diagram. The numbered messages will indicate the order of the events.



If the initial attempt on the reservation modeling is analyzed, the assumed scenario can be summarized as:

1. A customer asks for a listing of all trips from the "trip list" object.
2. The customer selects one trip in the list and waits for the information about the specific trip, the bus assigned to it, and the reservation status for all the seats for that trip.
3. The trip list object, having a trip selected by the customer, asks for the reservation status from the specific trip and receives the bus ID and the reservation status. All the result information is relayed by the "trip list" back to the customer, also the trip ID.
4. Now, the specific bus assigned to the trip can draw its seat layout. No other object should assume this internal operation. Meanwhile, the bus cannot know the reservation status, as it is only responsible for the "trip." Therefore, after the selection of the trip and acquiring the reservation status, the customer provides the bus such information to draw its seats with different colors according to the individual reservation status.
5. The customer can pick a vacant seat and its ID will be returned to the customer.
6. The customer finalizes the reservation request, by creating a new reservation record. This operation in turn, should update the reservation status for the trip.

There are a few points to articulate in this interaction model. The customer makes a "select trip" request to the "trip list" object and assumes three parameters to be returned. Also, the "trip list" object plays like an intermediary, and asks the specific trip to send its reservation status back, knowing that a trip selection will naturally require such information, although this information is not necessary to complete the selection operation. So, the "trip list" acts on the behalf of the customer to do this reservation request and relays the return parameters (reservation status and bus) to the customer, without any local need for them. This kind of intermediary functions imposed on objects point to an option in the approach:

1. Either the initiator (actor) assumes all the requests and directs them to the corresponding object directly, or
2. The initiator delegates an object that is central to the kind of request. This object initiates the actions to complete the request and holds the required intermediate information.

The above example employs both options by not delegating a central object to control the reservation related actions. At the same time, it assumes the central role by itself, keeping the parameters and directing them to other objects. Also, the "trip list" object assumes some central role. In most cases it may be desirable to prevent intermediary objects relaying parameters and creating secondary requests unless these secondary requests are vital for the completion of the primary service request. But then, care must be taken so that all the responsibility for conducting the service does not belong to a single object that is not meant for the service. In other words, when the user as an actor only asks for a service, it should not do most of the service. This discussion leads us to a variation in the reservation collaboration, by allocating the control of the message traffic within a "reservation" object. Figure 8.8 displays the revision in the "reservation function".

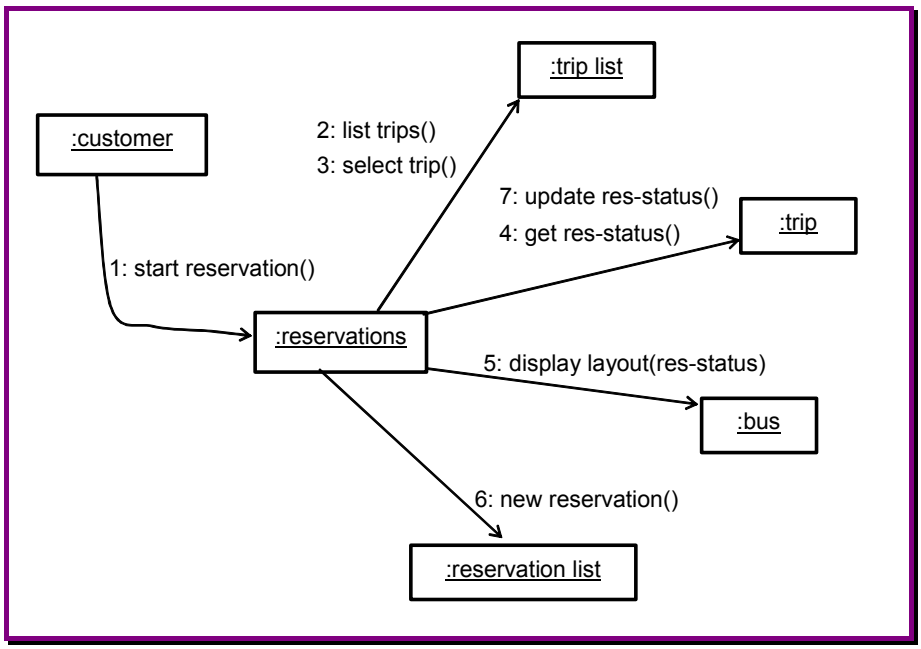


Figure 8.8. Reservation model with a specific control object

Next use case

The "view layout" system function is selected for the next interaction modeling. Meanwhile, the class diagram was not updated for the reservation use-case. There are new classes to declare and this task has to be completed. It is better to update the class diagrams so that the methods used in the interaction diagrams are declared inside the classes. A class diagram was included already and all update actions would take too much space to include. The next class diagram will include the introduced classes and the methods yet waiting to be declared. This scenario can be summarized in the following statements that correspond to the collaboration diagram in Figure 8.9.

1. Initiation comes from the reservations
2. The bus object first draws the frame and the background
3. For each seat to be drawn, the status of the single seat is accessed
4. The bus requests a seat to draw itself, sending it the reservation status

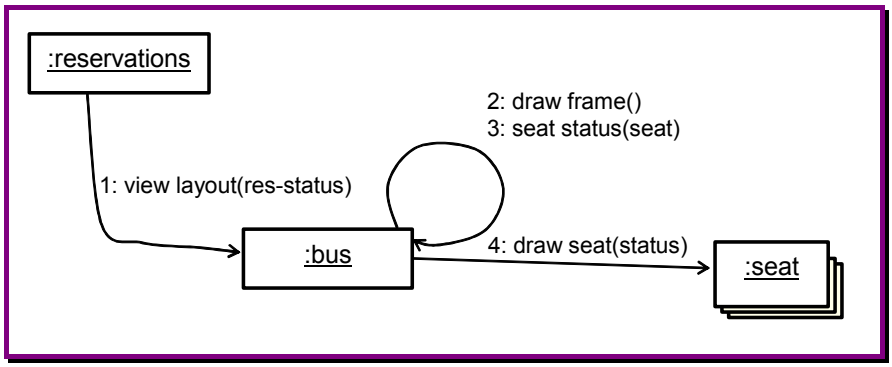


Figure 8.9. Collaboration Diagram for the view layout system-function

The new aspect of this diagram is a multiple object. The set of seats are represented with the "seat" object that is drawn as three overlapping object icons. Although the sequencing logic through all the seats is not specified in the collaboration diagram, elements of the loop body are represented. The loop block could be indicated using a comment box.

Return system function

The return operation is similar to the purchase operation. There is a reservation to be located instead of being created. Then the buy action is replaced by a return action that may be very similar to the buy action. First the textual explanation for the return scenario is provided.

1. Customer initiates the return operation: the reservation list is searched for the previously ticketed reservation
2. the reservation is displayed: payment type is displayed as at the purchase time
3. reservation is modified: the returned status is recorded
4. trip is updated with the new reservation status
5. payment is done back to the customer

The collaboration diagram in Figure 8.10 graphically models this scenario.

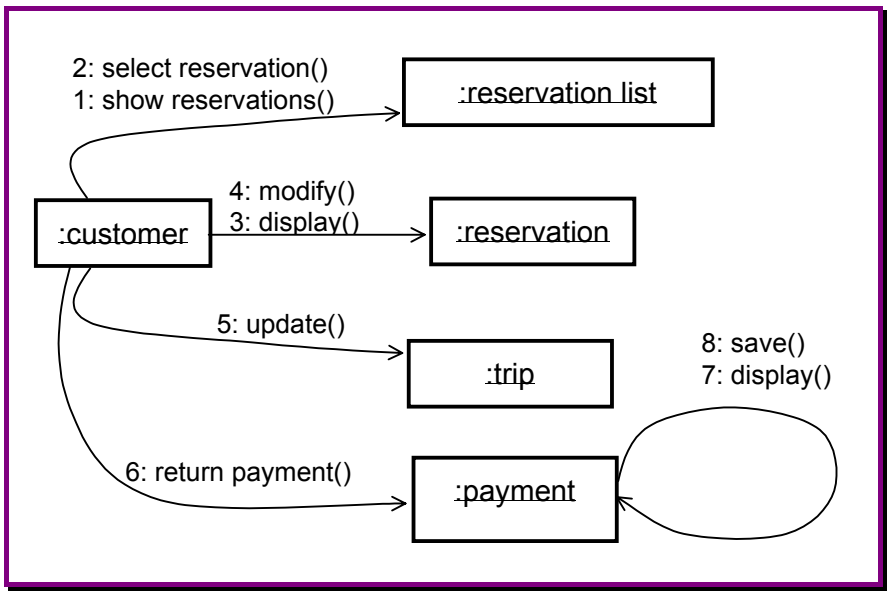


Figure 8.10. Collaboration diagram for the return system-function

List trips system function

The final oval in the use case diagram is the "trips list" use case. The interaction model again is a textual description of the scenario followed by the collaboration diagram.

1. Reservation requests a display from the trip list
2. trip list asks for the individual trips to provide summary information
3. the trip list displays the information for each trip on a single line

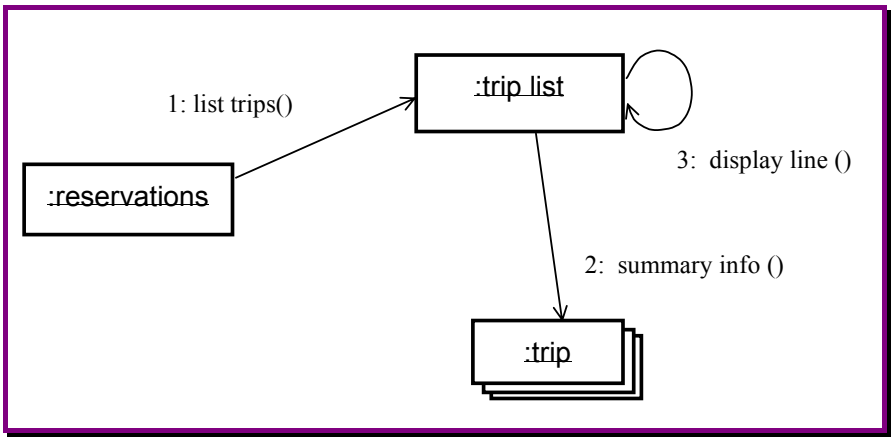


Figure 8.11. Collaboration diagram for list trips system-function

Next Capability: trip management

Although the problem as represented here is not very complex, it helps in the organization of the model to have more than one capability. A use-case diagram will be drawn for this capability as well. The trip's information needs to be set and modified by the users or administrators and used by reservation operations. A list of locations is the key to maintaining trips. Locations as well as departure and arrival times should be assigned to the trips, beside busses. Management of the bus list is assumed to be a part of this capability. Figure 8.12 presents a use-case diagram for tip management.

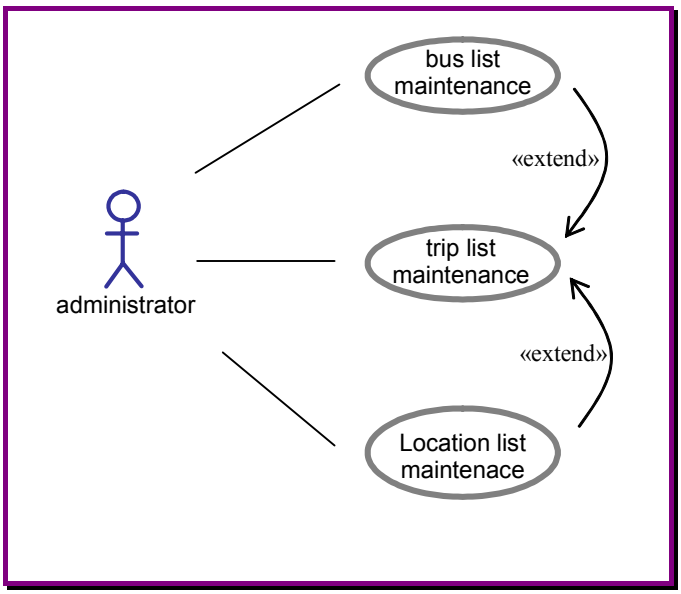


Figure 8.12. Use case diagram for the trip management capability

Bus List Maintenance

Actually, there are similar list management functions for different kinds of lists. Three such lists are implied in the use case diagram for trip management (Figure 8.12). The modeling for bus list maintenance will provide guidance for the other lists. Any list will save the records of the list elements, display them on the screen in the list form, allow adding, deleting, and modifying items. Also printing the list or an item and the display of a detailed view for an item should be possible.

The scenario for bus list maintenance should be divided into separate operations such as add, delete, and modify. Most of those functions will have little detail, if they are all represented in one interaction model. However, the scenario will have so many conditional sections and that is not a preferred case.

So many applications include many lists to be managed so a list manager module can be a good candidate for a pattern. Consistency is important especially in the user interfaces. If a delete operation is taken for example, first an item may be sought and displayed before the delete request can be made. Or, the delete operation may be selected in the beginning and then a search will be conducted and the located item will be deleted. So, the decision for all the lists is to whether allow listing/searching in the beginning and select an operation (delete, modify etc.) after locating an item, or to initially ask for the operation and do the search for the item. The latter choice may be easier for enforcing access rights per user, localized at the

main menu. The former option may appeal to some users as being more versatile. The bus list management scenario is listed as below:

1. If an add operation is requested then
2. a new bus object is created
3. modify operation is started with an empty bus detail screen
4. else:
5. User requests a list or a search function from the bus list
6. the individual bus is located and its detailed information is displayed
7. the user asks for a modify, delete, or a print operation
8. if “delete,” a confirmation message is displayed and the bus is deleted
9. if “modify,” the item is displayed in a detail screen and edits are saved
10. if “print,” then the bus detail screen is printed

This scenario can be modeled in two separate interaction diagrams, for the add operation and for the others. Figure 8.13 depicts a collaboration diagram for the “add a new bus to the list” scenario. Figure 8.14 presents the collaboration diagram that accounts for the other functions.

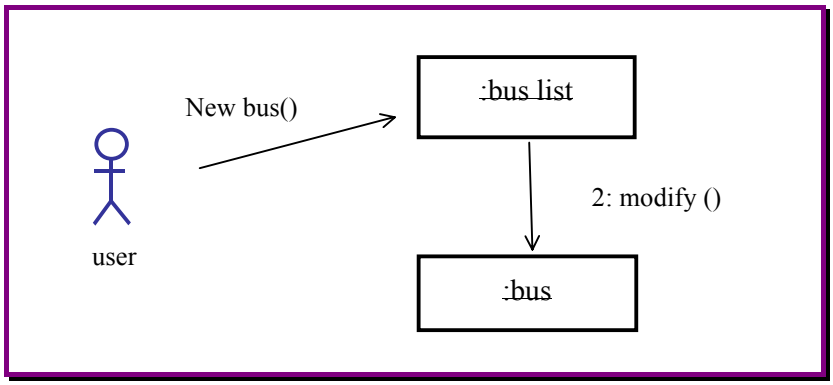


Figure 8.13. Collaboration diagram for add new bus system function

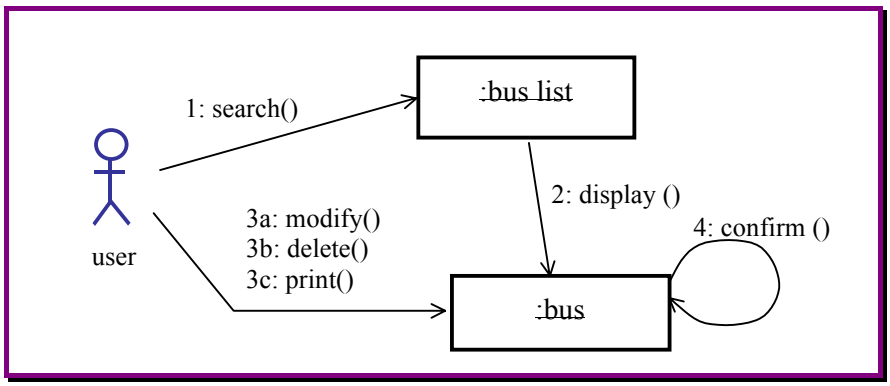


Figure 8.14. Collaboration diagram for the bus list management system functions

Trip list maintenance system function

Among the similar list related functions, the trip list maintenance is probably the most involved. It is worth providing the scenario and the graphical model for it. Locations, busses, and drivers are involved. A calendar element is also necessary. Adding a trip and modifying a trip are similar operations. Apart from the difference as the initial creation of a new trip or searching for an existing trip, all the rest of the functions are identical. Also, for the creation of a trip, a trip number needs to be determined. Only including the modify trip list function will be sufficient for the chapter: in a commercial docum ent, it is preferred to present all the functions. A trip modification sequence including all the possible information accesses is presented:

1.	User requests a listing of the trips
2.	user selects one trip
3.	modification on the fields of the trip information is conducted
4.	origination location is selected from a list of locations
5.	take off date and time are entered
6.	destination location is selected
7.	arrival date and time are entered
8.	a bus is assigned to the trip
9.	a bus driver is assigned
10.	new trip record is saved

This scenario is first modeled using a collaboration diagram shown in Figure 8.15. Looking at the number of messages between some pairs of objects and

their sequence numbers, one can prefer a sequence diagram for this collaboration.

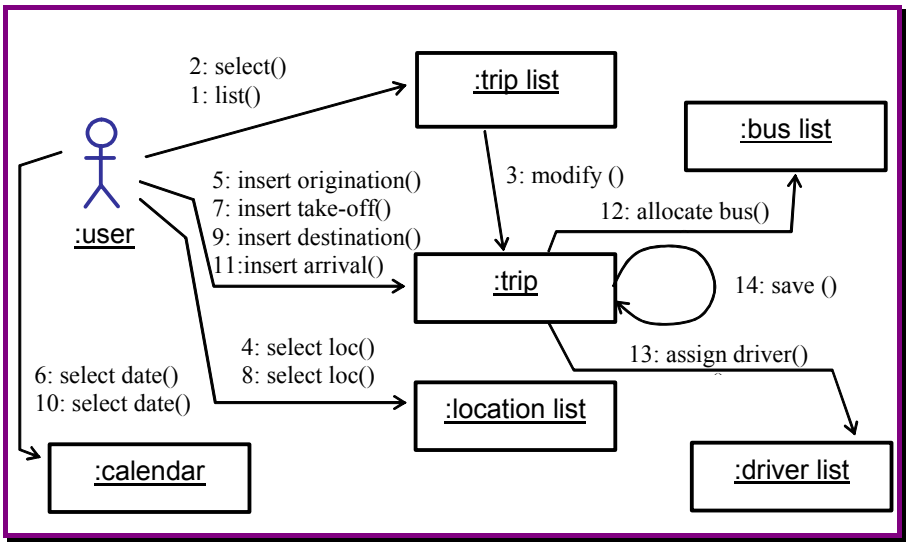


Figure 8.15. Collaboration diagram for the modify trip list system function

Figure 8.16, presents the sequence diagram corresponding to the collaboration diagram in Figure 8.15. A sequence diagram helps the visualization of the timely ordering of messages in a better way.

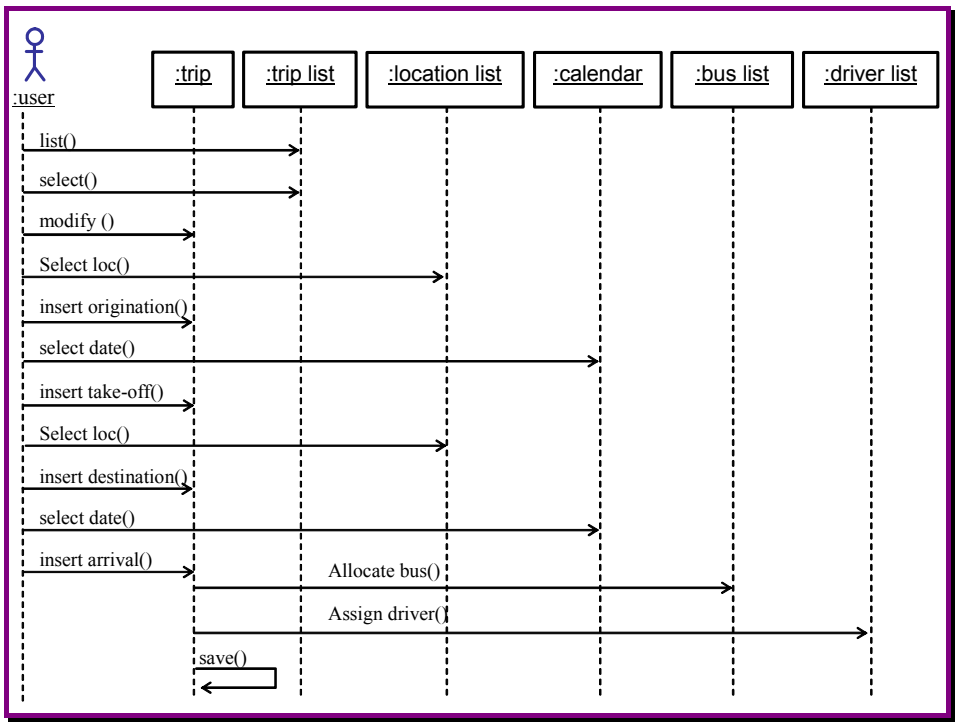


Figure 8.16. Sequence diagram for the "modify trip list" system function

The information provided for the trip list is sufficient for the case study. The process continues with the specification of the next capability.

Business automation capability

The business related services are contained in this section. Sales related functions are left outside of the responsibilities as the reservations capability is handling such services. Human resources, reporting and accounting abilities are actually what is meant here. Figure 8.17 depicts the use case diagram for business automation capability. Here we assume that the reservation function uses human resources and accounting services. The reservation use case was included in another use-case diagram before. Actually, it is not part of this capability, so it may be modeled as an actor.

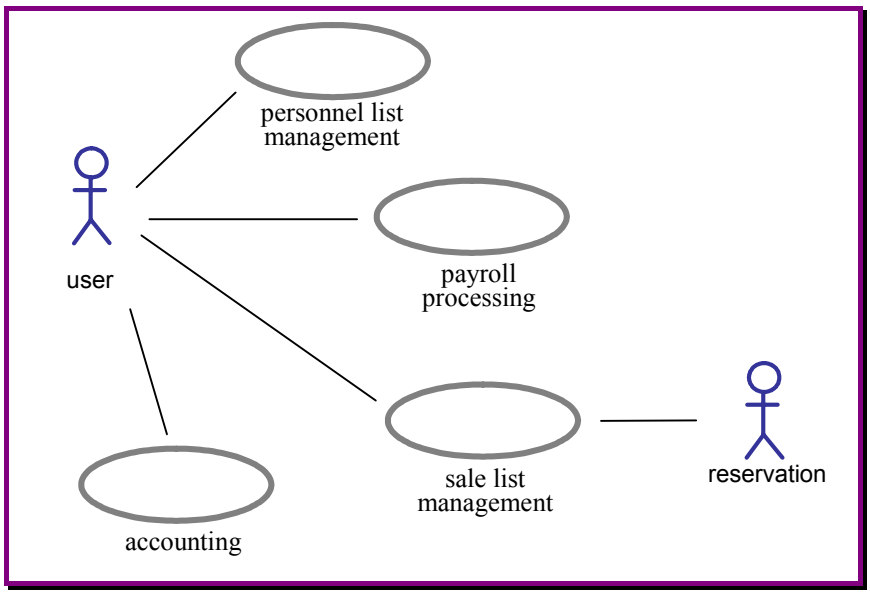


Figure 8.17. Business automation capability

Once again, there are lists. Detailed modeling about the personnel list and sale list management functions will not be included because similar models have been demonstrated in the previous sections. The reader is reminded again that this is only for the demonstrative context of this text; a commercial development should include specification for any aspect, whether or not similar to other aspects.

The sale list is automatically modified when there is a sale or a return operation. Accounting is interpreted as the ability to analyze money and other payment related transactions and produce reports. Payroll processing calculates the monthly salaries for the personnel that include a fixed amount plus sale percentages for tickets.

Payroll processing system function

The payroll processing system function operates per personnel. The fixed amount for the salary is accessed through the personnel records as well as the sale percentage for the person. Then a search in the sale list is conducted to find the total sales this specific person has completed for the period of concern and percentages for all those sold tickets are added. A periodic payroll report is generated that shows the different constituents of the monthly pay:

1.	A person is selected.
2.	Fixed salary is taken.
3.	Sale list is searched for the sales done by the person.
4.	Sale percentages for every located ticket are added.
5.	The payroll report is prepared

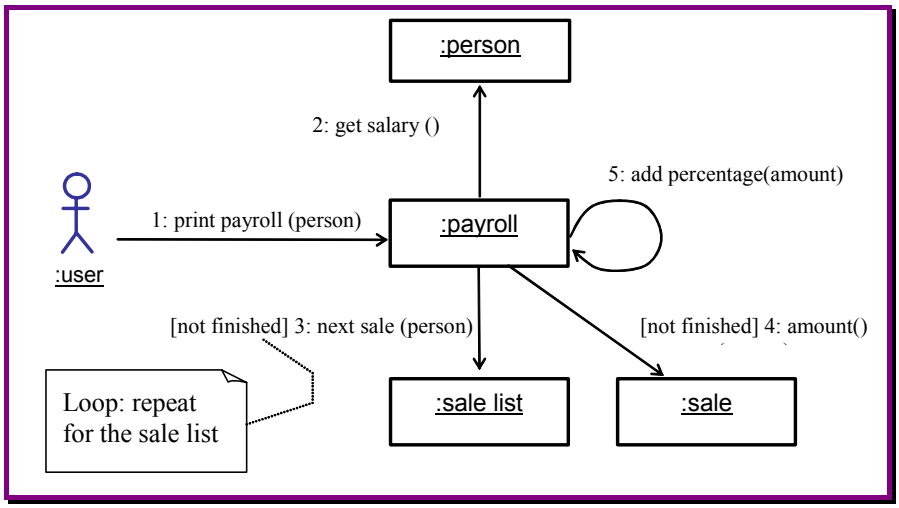


Figure 8.18. Collaboration diagram for the payroll processing system function

Accounting system function

The output of this function is reports. For any report, a date period is required. First, the starting and ending dates need to be entered. Then a report type is selected. To prepare a report, the sales list is polled and, if necessary, the personnel list: some reports may require detailed sections based on different personnel. Finally, the gathered information is printed within the format for the selected report type:

1.	Report period is entered by the user
2.	Report type is selected
3.	If needed, personal information is taken from the personnel
4.	For every sale item in the sale list:
5.	Required information is retrieved
6.	Report information is prepared
7.	Report is printed

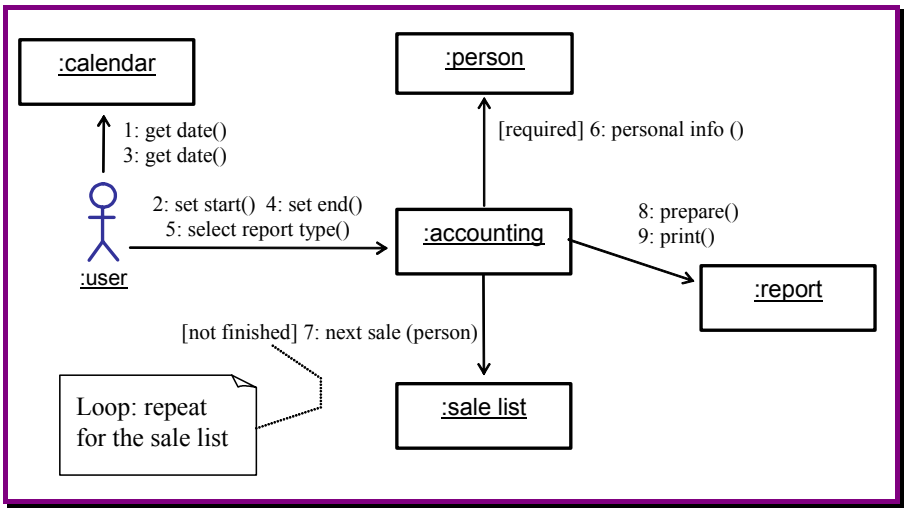


Figure 8.19. Collaboration diagram for the accounting system function

Final capability: client list

Although this is another list related capability, more specific requirements exist for the client list management. Besides routine maintenance such as add/delete/modify operations, other facilities are desired to keep in touch with the clients. One such facility is the tracking of the birthdays of the customers and mailing birthday cards. In addition, the activity frequency of a customer is monitored to keep track of new promotions after a long period of missing contact. Figure 8.20 presents the use case diagram for the "client list management" capability.

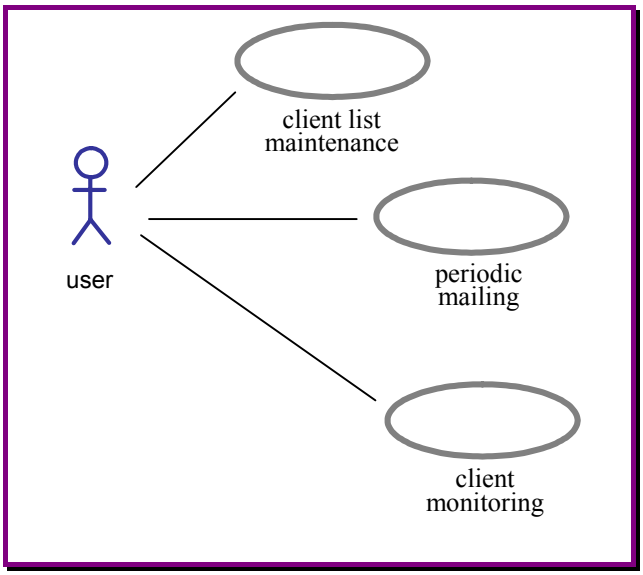


Figure 8.20. Use case diagram for the client list management capability

The client list maintenance system function is similar to any other list management functions and therefore will not be specified for further detail. The periodic mailing function is responsible for checking the client list to find birth dates that are coming soon, or clients who have been passive for a long period. Client monitoring function is responsible for determining the passive clients.

Periodic mailing system function

The user may request a birthday mail function. The list of clients will be searched for the clients whose birthday falls within the coming week. For those, a birth-date card will be printed for mail. Another request could initiate mails to the customers who have not done business for six months or for some other period that would be set by the client monitoring function. Figure 8.21 displays a collaboration diagram that corresponds to the periodic mailing scenario:

1. User requests birth-date mail.
2. Beginning and ending dates are calculated.
3. The clients list is traversed for birth-dates within those dates.
4. A card is printed for each client satisfying the dates.

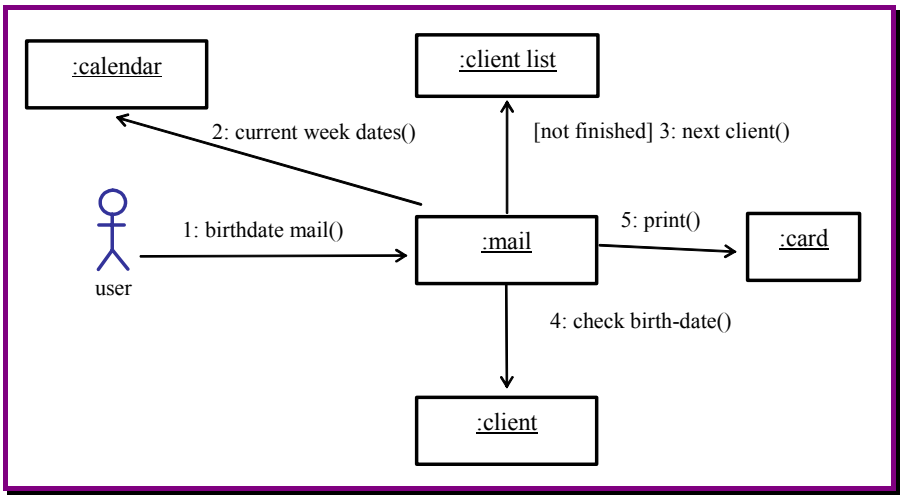


Figure 8.21. Collaboration diagram for the periodic mailing system function

Client Monitoring System Function

Whenever a client asks for a service, the system date is recorded as the client's last activity. An "aging" value can be changed if a different "passive period" for the clients is needed. Then a request like in the case of a birth-date mail will cause letters to be printed for the clients. Figure 8.22 depicts the collaboration logic for the client monitoring function. This function can be described as:

1. A client conducts a reservation.
2. System date is recorded as the last activity date, in client.
3. Administrator changes the aging value.

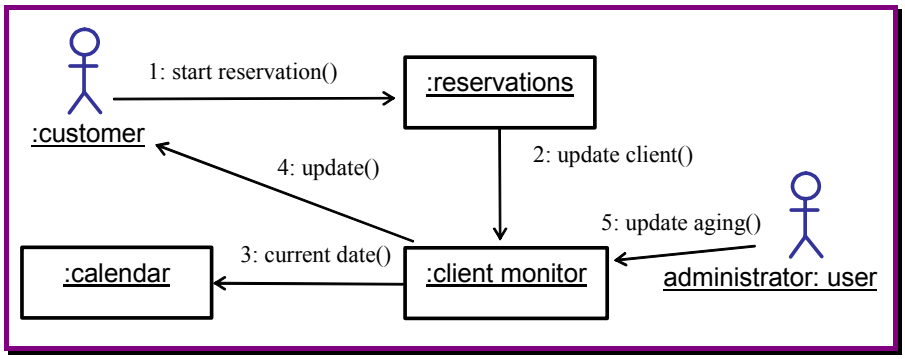


Figure 8.22. Collaboration diagram for the client monitoring system function

Class diagrams

A class diagram was presented before, in Figure 8.6. It is possible to partition the class diagrams to correspond to individual scenarios, including only the classes involved in the scenario with their interrelations. It is preferred here to present classes as the global resource for the whole problem. Also, different relations or types of classes may be considered for partitioning the class model. If partitioned, such model allows the repetitive representation of a class in different diagrams. Figure 8.23 depicts a class diagram accounting for most of the model introduced so far. Not all of the relations are included. In this class diagram, the classes are colored for indicating the type of the classes: Table 8.1 lists the symbols that are used to differentiate the different types of classes. These symbols will be located in the upper-right corners of the class symbols. UML allows extension to its symbols through this mechanism and refers to it as "stereotyping."

Table 8.1. Colors used in Figure 8.23 to indicate the class types

Class Type	Symbol
Base	B
List	L
List item	LI
Entity	E
Control	C
Composed item (Part)	P

Another missing aspect in the class diagram is the class details: apart from the base classes (List and Person), no class is presenting its attributes or methods. The classes should be including such information for especially representing the messages used in the collaboration diagrams. The messages need to be declared as methods in the classes. Although such details could as well be represented in the general class diagram, because such an expansion would render the diagram too large to fit in a book page several class diagrams will be presented each detailing a partition of the total class set.

One issue to resolve is embedded in Figure 8.23. The base classes include properties that are inherited, but actually cannot be used in the development. The "table" attribute in the "list" class needs to be overridden through re-declarations in the inherited classes. Actually, even in the logical model (let alone the implementation) these tables are of different types. Other operations declared in the list class also need to be re-declared in the derived classes. On the methods end, however, this is less of a problem because the overriding or even polymorphic declarations of the same methods are well-accepted trends. The virtual declaration of the "table" in the "list" class is for the logical description of the inheritance: All list type of classes will have a (database) table, despite being very different.

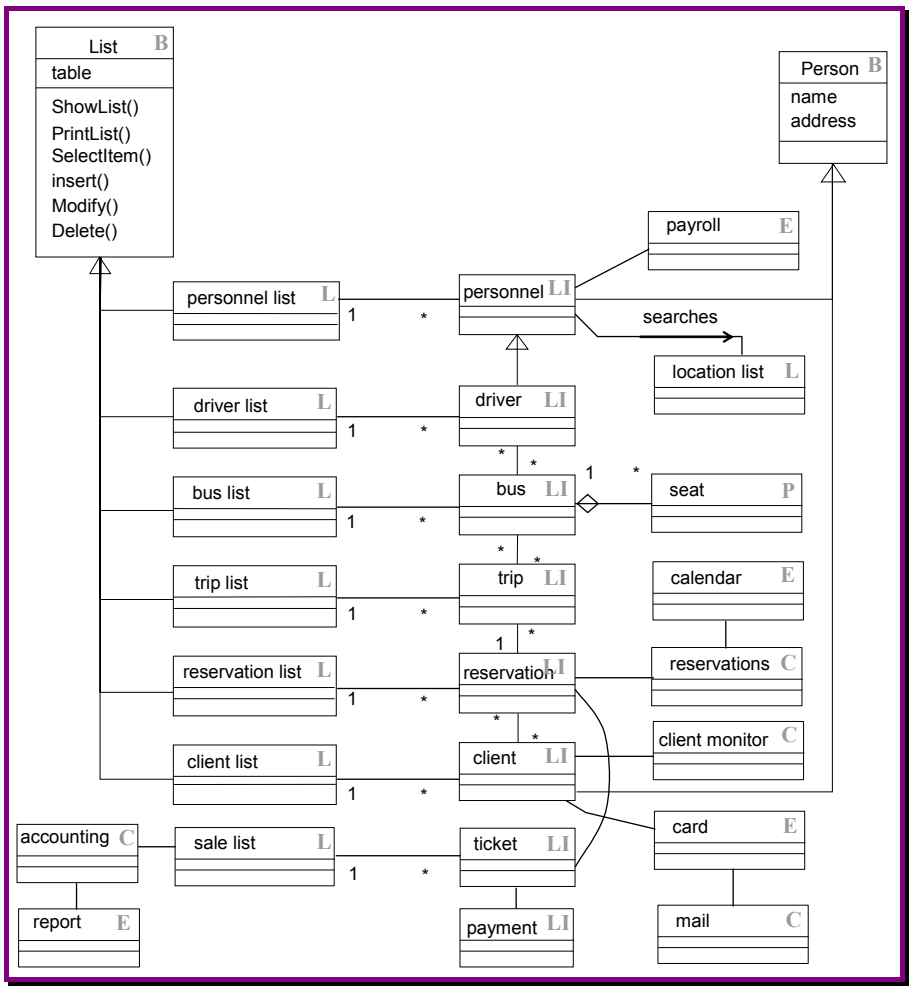


Figure 8.23. General Class Diagram

A class diagram that includes the attribute and method details is provided in Figure 8.24. This diagram is based on Figure 8.6, the first class diagram that contained classes related to a reservation scenario. Now that more classes and system functions are defined, the more classes can be included while declaring the internals as well.

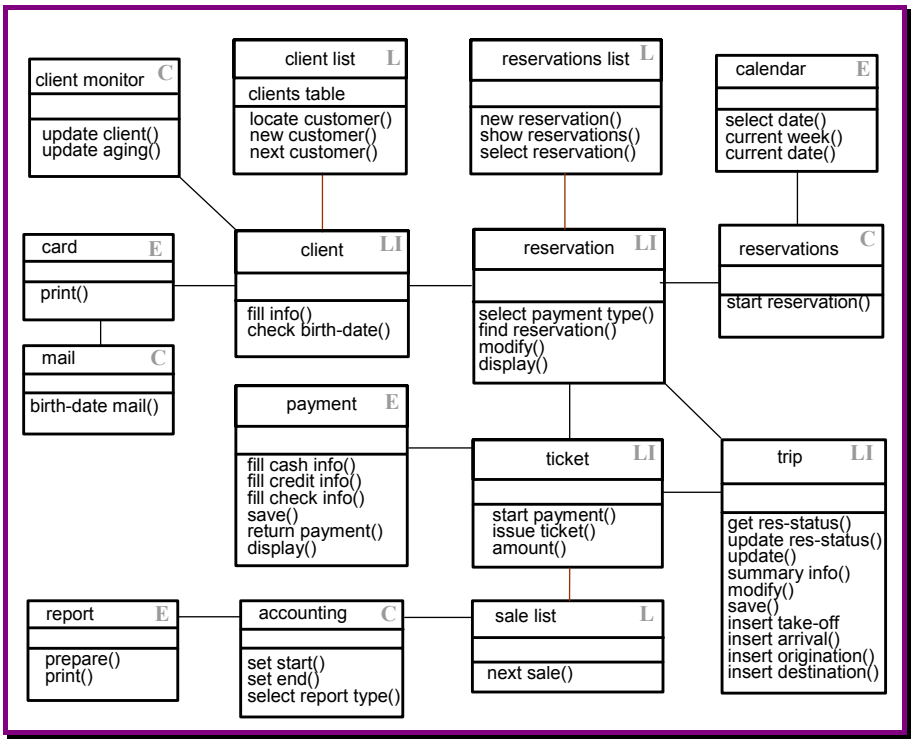


Figure 8.24. Class diagram containing reservation related classes with internal declarations

It should be noted that the class diagrams introduced to represent the internals, do not include all of the relations defined before; especially pluralities are omitted. In addition, the virtual functions declared in the "list" class of Figure 8.23 are replaced by method names more consistent with specific classes. For example, "new reservation" method is declared in the "reservation list" class rather than using the "insert" method existing in the "list" class. Since there is very little chance that different list objects will be used for polymorphic invocations, better naming is preferred for the methods per specific class. The declarations in Figure 8.23 are now outdated, and they only represent conceptual information as to what this class stands for. Figure 8.23 requires an update before it is presented in the requirements report. Figure 8.25 displays the class diagram that includes the details for the classes that are related to trip operations.

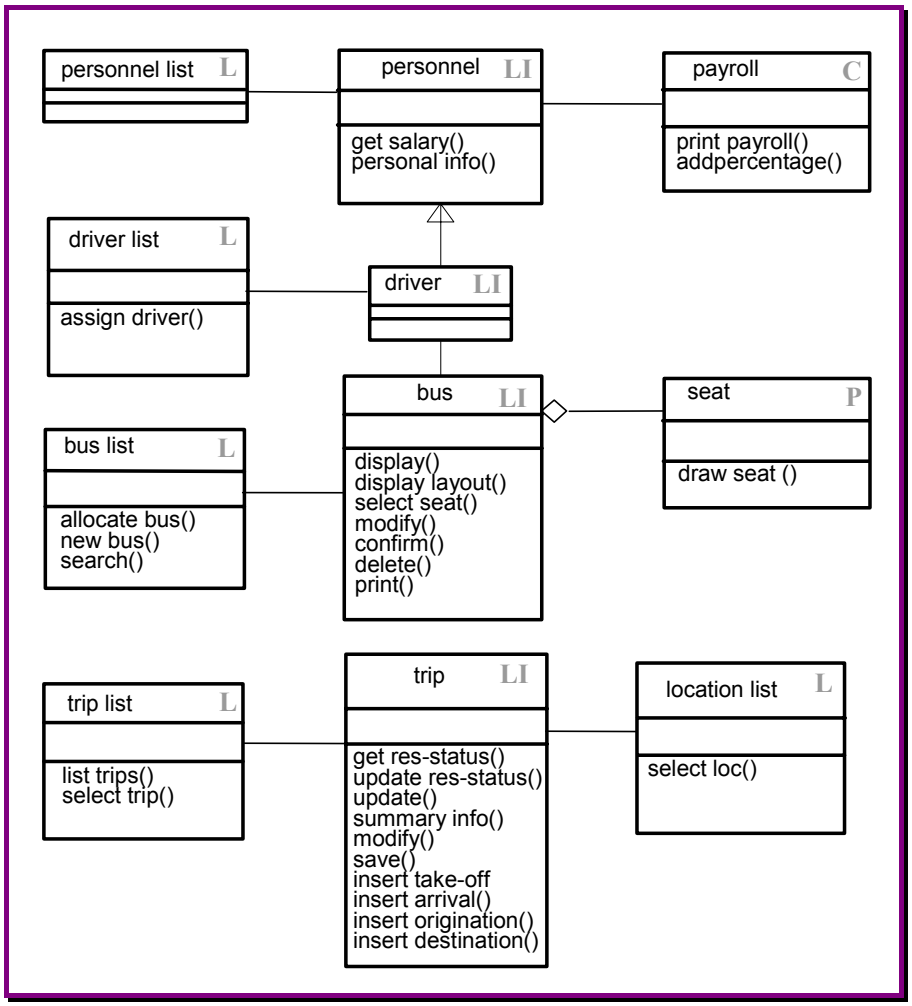


Figure 8.25. Details for classes involved in trip related functions

Final comments on requirements model

The class diagrams were the last graphical models included in the requirements modeling. If the problem suggests, state chart or activity diagrams could as well be incorporated and a logical grouping of the classes could be realized by allocating them in different "packages." A similar approach will be utilized in design where "subsystems" (very close relatives of packages) are used to physically separate the groups of classes/objects.

The order of the activities presented so far is based on a synthetic approach where first objects were introduced in interaction diagrams. Later commonalities were drawn to figure out the classes representing the objects.

This is a matter of methodology. Different approaches may suggest to start with the most general classes and refine them towards more specific classes, and finally objects.

Design

The approach taken for design can be summarized as loops consisting of subsystem design, object design, and requirements revision. The subsystem design is actually a decomposition of the solution into chunks. Object design introduces new properties and methods to the classes that were introduced in the requirements model, and adding new classes and objects. Further, the interaction models are updated to reflect the solution definitions. As a result, the requirements model is both refined towards implementation related details and revised with respect to surfacing change needs with developer additions to the problem definition.

Initial step is to partition the defined classes in subsystems. First separation comes to mind with respect to the distributed nature of the application: There will be offices and a special location that holds the database. Further, the classes that could be placed in the center can be further separated into two subsystems: one for the resources and another for the reservation related classes. Once the top-level subsystems are defined, further subsystems can be positioned inside the former ones. Figure 8.26 depicts the subsystems. It is noticeable that the subsystems correspond to distributed nodes therefore for this case study it may be appropriate to present a deployment diagram early. Otherwise, drawing deployment diagrams usually are left as one of the latest activities.

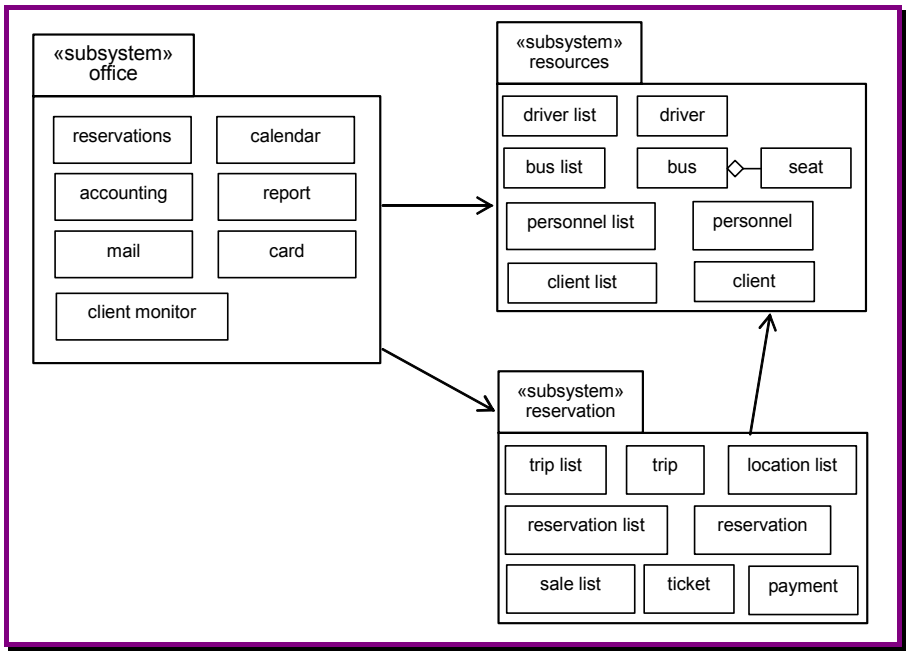


Figure 8.26. Subsystems

The deployment diagram presented in Figure 8.27 indicates that the reservation and the resources packages are contained in the server (center) and the office package is contained in the office node.

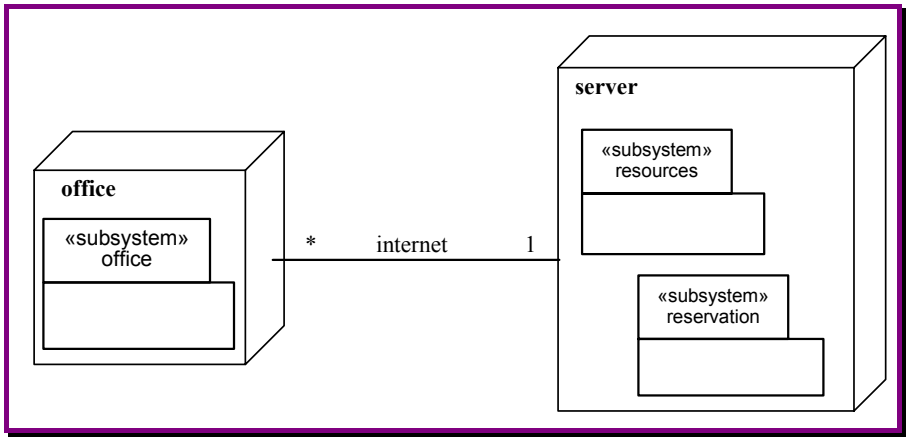


Figure 8.27. The deployment diagram

Objects revisited

Previously, objects appeared in the interaction diagrams. Probably these are the best places to investigate the objects for refinement. The responsibilities are better viewed in such dynamic modeling diagrams. In design, issues such as efficiency are also considered. The refinement can correspond to

necessary inclusions for the functionality, or for other aspects such as quality factors. The birthday card mailing function can be taken as an example. It can be observed from Figure 8.21 that the corresponding collaboration diagram suggests the ": mail" objects accessing the ": client list" to receive client records, one at a time. Now we have access to the subsystem structure, and even to the distributed allocation of the objects through the deployment diagram. Consequently, it can be noticed that for each record a request will be issued over the Internet connection, to a remote site. Surely, this is a costly operation in terms of time. For increasing the efficiency, we may opt for reading all the records at once and saving a copy of the clients that satisfy the criteria (those who have birthdays in the coming week). This will introduce the first item to be defined during design, for the case study, such as another "client list object" of the same class "client list." Then, it is the objects we are concerned with their locations, not necessarily classes for this issue. Further, the methods need some rework, in order to implement the new mode of the client list traversal. Figure 8.28 reflects the necessary changes on the previously drawn collaboration diagram in Figure 8.21. In addition, user interface objects may accompany the classes that need to display some information or that require human interaction. After the refinement, new and modified methods necessitate an update on the class diagrams. Figure 8.29 depicts the classes that need modification because of the mentioned refinement.

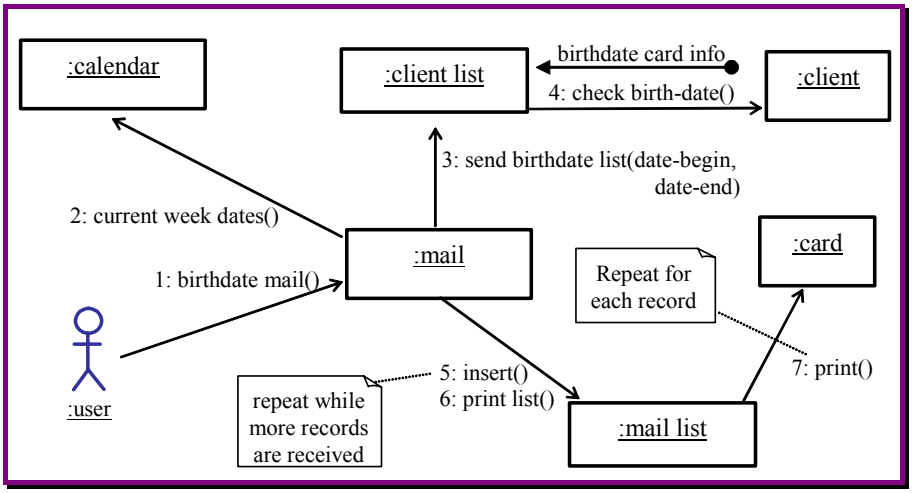


Figure 8.28. Collaboration diagram for the periodic mailing system function

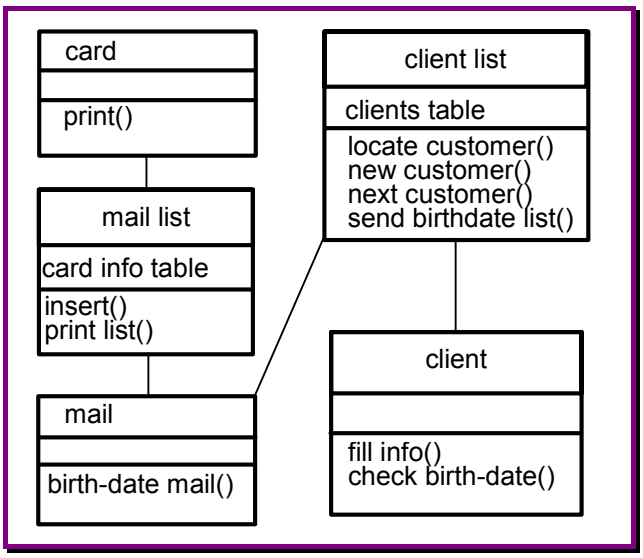


Figure 8.29. Class diagram for the modified classes

After cycling through the decomposition (subsystem definitions), object refinement, and analysis model revision, the set of objects/classes can be tested for their assumed responsibilities. Actually not included in the UML methods, a Class Responsibility Collaboration model can help with this. The system functions will be realized by the dynamic behavior represented in the interaction diagrams, through responsibilities assigned to the objects. Such responsibilities can be taken one at a time and a walk through can be conducted to see if the existing methods can meet the responsibility.

Database interface

It is a good idea to define interface whenever there are different technologies or chunks of the system having to work together. In the case of databases, however, the data management is so much coupled to the class varieties that a localized interface structure to serve the database connection is often not utilized. The idea should not be categorically ruled out though. For the case study, the common tendency will be followed and the cooperation with the data base management system will be regulated through policies of design. For example, any table in the database should correspond to a class where queries will be modeled by the methods of the class. This policy finds its way in the example through the "list" classes. A list represents the same kind of data with a database table: every element of the list will be stored as a record and the fields (columns) on the table correspond to the attributes of the element. Here the term element is used for the list element that should also be represented as an "item" kind of an object. An item object will hold the value corresponding to the last read record from the database table (or the record to be written soon). Relations among the classes can also be implemented by the relations among the tables.

The interface inevitably introduced the structural mapping between the object oriented model and the relational database tables. How can tables represent the classes that inherit from others, or compose others? These problems happen to have particular solution patterns, for the base class and for the derived class tables will be declared. For any object of the derived class, records will be generated in both of the tables and these records will point to each other through external keys fields. A similar technique holds for the composition relation: the container type object will also require a record for the composed object, in the table corresponding to the composed class. In general, the association kind of relations in the class diagrams can also be represented as pairs of mutual pointers.

Graphical user interface

Whenever there is interaction between a human user and the system, there will be Graphical User Interface (GUI) modules, assuming the most widely used software types and platforms. The GUI classes will undertake most of the responsibility for a human actor. Sometimes these classes will conduct a kind of controller task, managing the interaction traffic. In the logical levels of modeling, such activities can be regarded as access functions for the information kept inside a class. In design level modeling, such an access function needs to regard the data to be displayed and/or input, in formats that require graphical resources on the screen. Practically, windows environment displays information in message boxes and accepts information in dialog boxes (or forms). We may need a GUI class for the interaction that in turn may activate further classes responsible for such display or input screen elements.

Since there are different kinds of users with different interaction expectancies, the special package for the user kind should be initiated in the beginning, or a single program will configure its menus and access rights based on the login information for the user. For example, the administrator will need different functions for updating the bus list, location list, etc. Whereas the travel agent (office employee) is interested in reservations, sales, returns, and sometimes the client update and mail operations.

The screens should be designed and then class structures to implement them can be constructed. Actually, graphical compiler environments allow a quick deployment of a series of screens. To achieve such an executable GUI, the designers need to introduce some classes. At this point, it can be assumed that the GUI related classes are designed together with the screens. A windows style interface suggests an opening screen with a menu bar on the top that includes the names of the individual “pull-down menus.” Figure 8.30 describes the main menu for the application. A login option is provided, and the menus presented assume that a travel agent has logged in. The menus change depending on the user: some menu items disappear and some others

are inserted. Actually, the scene in Figure 8.30 will never be available on the screen; one pull-down-menu at a time will be activated whereas Figure 8.30 displays them all together as if they were active in the same time. This is to provide all the main menu information in one figure.

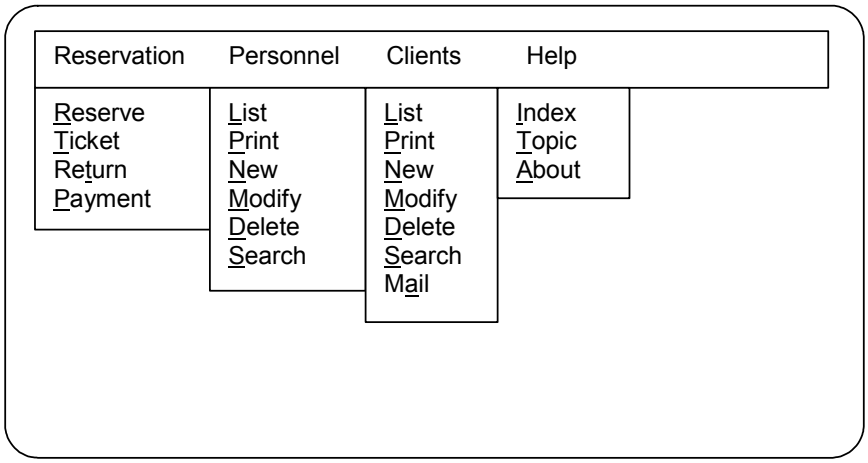


Figure 8.30. The main menu

Coding

So far, models have been used to specify and understand what and how to build. It is time to convert that information to code and develop the required system. C++ is selected as the implementation language for the case study. First step is to utilize the class descriptions. The information included in the class models can be converted to directly corresponding code segments. However, this is not enough to completely code a class. More need to be added. This is valid especially for the body content for the methods. If the previous specification included comments or some textual description of how the code should proceed for the functionality, the programmers have a very useful guide.

The recently displayed class “client” can be the first to demonstrate its code. The client inherits from class “person.” To understand the client fully, it is better to view the code for both of the classes.

```
class person {
                    string          name;
                    string          address;
                    string          getname();
                    string          getaddress();
          void      setname(string name-p);
          void      setaddress();
};

class client: person {
                    void            fill_info();
                    int             check_birth-date();
};
```

This code only contains what was provided in the design model. More work is needed. The following code segment defines the internals of some of the methods declared above.

```
string              person::getname() {
                    return name;
};

void                person::setname(string name-p) {
                    name = name-p;
};
```

There exist certain methods that are more involved than the simple get/set operations. If the fill-info method for the client class is considered, the potential need for a GUI class may be noticed. Displaying a dialog box for the user to enter the information is a widely applied way of implementing this kind of an action. Then to continue with the coding example, we must assume the presence of such GUI related classes together with some variables for accessing those new classes. Let us define the client class again with some additions and continue with its methods:

```
class client: person {
        f* fill_message-box;
        void    fill_info();
        int     check_birth-date();
};

void client::fill_info() {
        f = new(fill_message-box);
        f ->.show(self);  // the dialog box will fill the attributes of the client object
};
```

Chapter 9: Component Oriented Development of A Travel Reservation System

Introduction

For an effective component oriented development, a domain for the concepts and the component set should be mature. The bus travel reservation domain is not available to us and will be assumed to be present, during the presentation of the development process. The early activity is the logical modeling that could be independent from existing components despite the fact that this independence hinders the development efficiency. There could be two major paths in demonstrating such an approach depending on the utilization of domain orientation or not. The case study will present a slight taste of a related domain and continue with CO development where some information is implicitly taken from the domain model.

The Domain

No matter how formally defined, the existence of a bus travel reservation field for software development is intuitive. There must be some expertise, some requirements or design work already conducted, and some software pieces developed before. Existence of geographical location modeling such as cities, their coordinates etc. can be assumed. Trips connecting locations for some date/time and containing seat information regarding the free/reserved/sold kind of reservation status should also be in the picture. Such speculative thinking about what may be available will affect even the early decisions in partitioning the logical model. Figure 9.1 presents the context of the assumed domain.

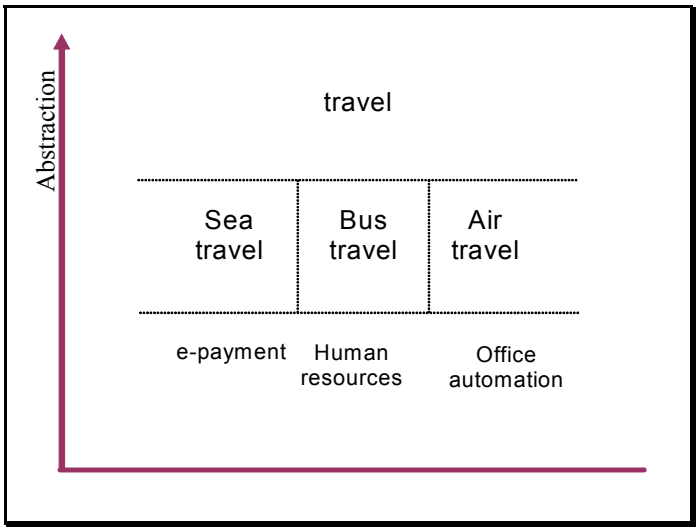


Figure 9.1. The bus travel domain context

Basic entities are locations, busses, trips, clients, personnel, and perhaps facilities such as buildings. On the procedural side, there should be the reservation operation as the most important one, maintenance of the lists of busses, locations, clients, personnel etc. Although the architecture should not be involved in the logical modeling (closer to requirements than design), we might want to articulate on this dimension since component orientation considers structure as the fundamental modeling dimension. The early architecture considerations could relate to top-level entities as well as components at the bottom of the abstraction levels. If we start thinking in lines of "what existing components could be present?" then it is the bottom. Alternatively, we may start decomposing the system definition with respect to logical entities or physical constraints such as:

- Parts of the software that will reside in the front offices,
- Distributed, Mobile sub-systems, etc.

A function oriented start could as well be considered top-down: thinking about for example, the reservation operation, the decomposition could start at the outset with functional or data oriented criteria.

In an encompassing domain model, different alternatives should be supported with architectural, logical, and structural elements. The following sections contain such elements as samples.

Domain Dictionary

This structure is a blackboard resource. It can be built in parallel with all the other model constituents. However, presentation-wise it is more convenient

to locate the terminology in the beginning. Some kind of semantic net will help in the utilization of the dictionary: Navigating across the terminology with respect to relations that bind different terms is a useful approach for understanding the foundations. The medical terminology dictionaries for example, have found their ways into the medical field, in such semantic structures and there are quite a number of matured application examples. A category of terms relates to organs. From any organ, one can follow the "diseases" link to arrive at possible diseases related to this specific organ. A disease may be linked to treatments, as well as medicine. This chapter contains the case study for bus travel domain to support the rest of the book therefore a simple and hypothetic example will be constructed for the domain of our interest. The small set of terminology is listed below to take part in the semantic net representation of the dictionary that may be in the form of an entity-relationship model represented in the UML class diagram syntax:

- bus
- date
- location
- seat
- trip

Additional structures can support the utilization of the dictionary. Table 9.1 is an example to such structures where a *data dictionary* format is followed. Figure 9.2 presents the entity-relationship model for the domain dictionary.

Table 9.1. Domain Data Dictionary

entity	explanation	relates with	context	format
bus		trip, seat	sub-trip	
date		trip	elementary	
location		trip	elementary	
seat		bus, seat	sub-trip	
trip		bus, location, trip	ticketing	

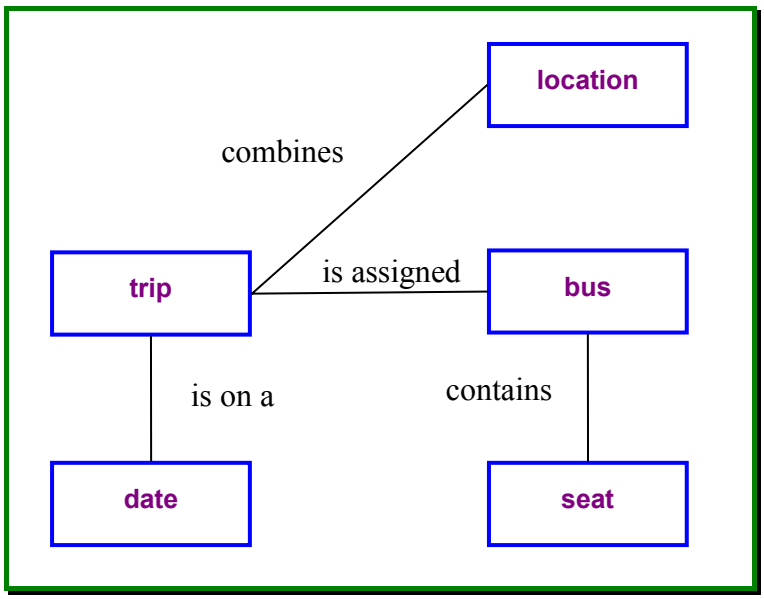


Figure 9.2. The partial domain relational dictionary model

In contrast to the limited set presented here, a domain model should be as much covering as possible. What differentiates these models from being specific to the early modeling of a specific project (rather than a domain model) is that domain models should be generic and should support a variety of applications, and further, a variety of approaches for the development of those applications.

Design Patterns

Intermediate-level knowledge for the domain repository can be demonstrated by some Design Patterns (DPs) in this section. A pattern can be strictly abstract, as the definition of a DP suggests. Also allowed in the COSE approach are the instantiated versions of DPs, which actually become super components after instantiation. A generic collaboration among a few objects can represent a DP.

Two different design patterns will be demonstrated to represent a distributed and a central control in the collaboration of abstract components for conducting the reservation system-function. In central control, a dedicated object assumes the arbitration for the message traffic. The initiating request and the external interactions address this component, which, in return, manages the message transactions among the "worker" objects. Figures 3 and 4 display the central and distributed control-type collaborations, respectively, for the reservation operation.

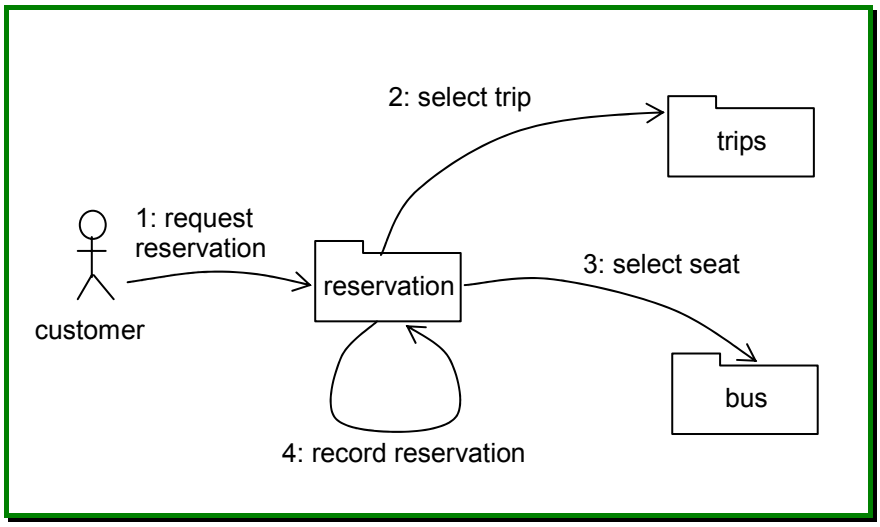


Figure 9.3. The reservation design pattern with central-control

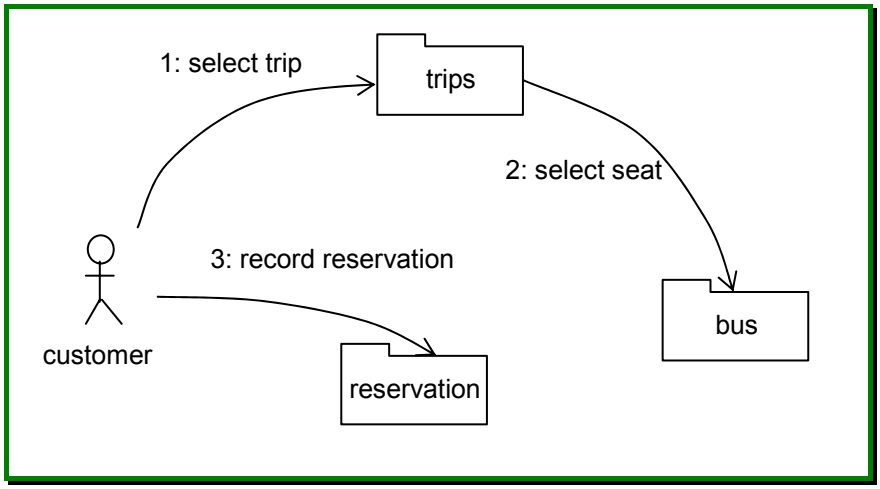


Figure 9.4. The reservation design pattern with distributed-control

There is so much that could be part of the domain model. High-level concepts could be offered with their inter-relations. Architectural descriptions are another aspect of a domain model where alternatives for the complete or partial solutions can be viewed and adopted. An example high-level architecture is presented in Figure 9.5.

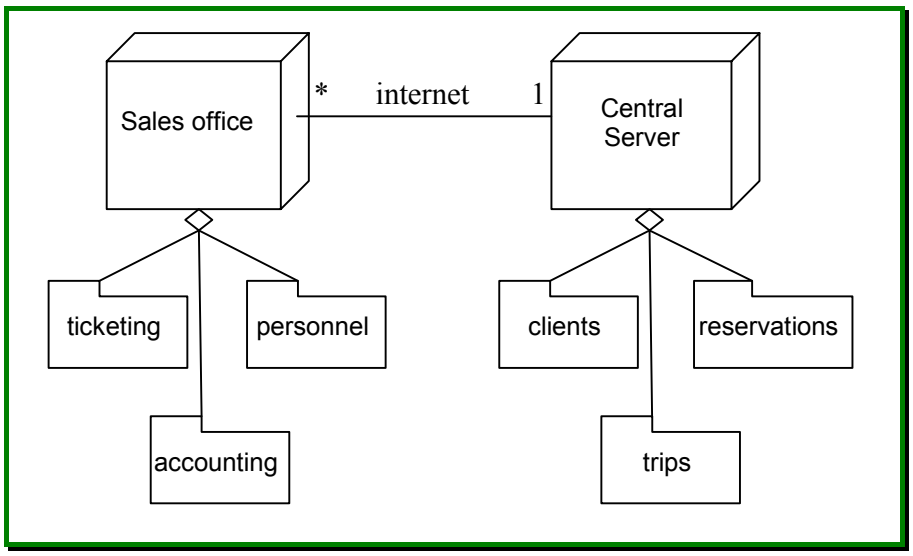


Figure 9.5. A distributed architecture for bus reservation systems

DPs were incorporated as intermediate granularity solution parts in this section. Finally, a set of components can be part of the domain environment to support the integration of the final product. A set of hypothetical components is presented in Table 9.2.

Table 9.2. Components for the Bus Travel Domain

reservation	reservations	trip	trips	bus	busses
client	clients	employee	employees	ticket	sale
accounting	credit-card	cash	cash-register	shopping-cart	mail
e-mail	reports	drivers	calendar	appointment	groups

The rest of the case study makes use of the concepts introduced in the domain environment. Utilization of the domain model is not mentioned explicitly but the reader will recognize similar concepts, patterns, or components.

A Bus Reservation System

This section presents a case study that presents the modeling of a reservation system for bus travel. The system allows customers to select a destination, time, and seats for the trip. Trips are important concepts, like flight-numbers in air travel domain. Origin and destination as well as the time and date for the travel define a trip in this case study. Further, a specific bus has to be

assigned to each trip and seats will be offered for selection on a graphical display that has to be different per bus model. A customer should be allowed to reserve, buy, and return a ticket.

So far, very little has been defined but still developers have a lead to try their initial decomposition. Even with this early start the separation of units should go parallel with the considerations about the chances that any unit defined in the decomposition should better correspond to a component. Even with expected very large granularities, the top-level abstractions should be confined with contexts to match components. It is more unlikely to locate a component for an abstraction this high-level; one should expect to correlate lower-level and hence smaller "requirements modules" to components easier. Nevertheless, it is assumed that this practical criterion should be in mind rather than logical criteria while identifying the players in the decomposition.

It would be very efficient to run into existing components to represent higher-level abstractions. Knowledge of the developers about the available component spectrum is a key asset for effective decomposition activity. Those less experienced or knowledgeable would be advised to ask at any refinement step, "what kind of decomposition yields abstractions that has better chances to match existing components." This is even if there is no available match. It is assumed that this approach may increase the chances in the successive decomposition for further sub components. Another view to this judgment is to consider being a component developer and decide what could define a general purpose component that others may have a chance to use, and also can be partitioned out of our requirements space. The generality of a component is very high if it can be used in any domain. If such a generic component will not work, then a more domain specific one should be imagined.

The case study progresses with the definition of the first-level decomposition to yield the sub-systems as:

- Customer
- Reservation
- Office
- The Bus fleet.

At this highest level partitioning if it is likely to locate components, they would be of general and large-grained kinds that would probably require considerable tailoring.

Initial decomposition is illustrated in Figure 9.6 corresponding to the items determined and listed above. The immediate next step after any decomposition is the definition of connectors. If any abstract component would require communications with others, the communication channels are

defined and available in the forms of components. In Figure 9.6, the office is connected to every package; the office personnel will at least need to change the configuration of the system. The reservation is assumed to require the trip information and the seat layouts from the busses. That is why connectors are drawn between the reservation and the bus packages.

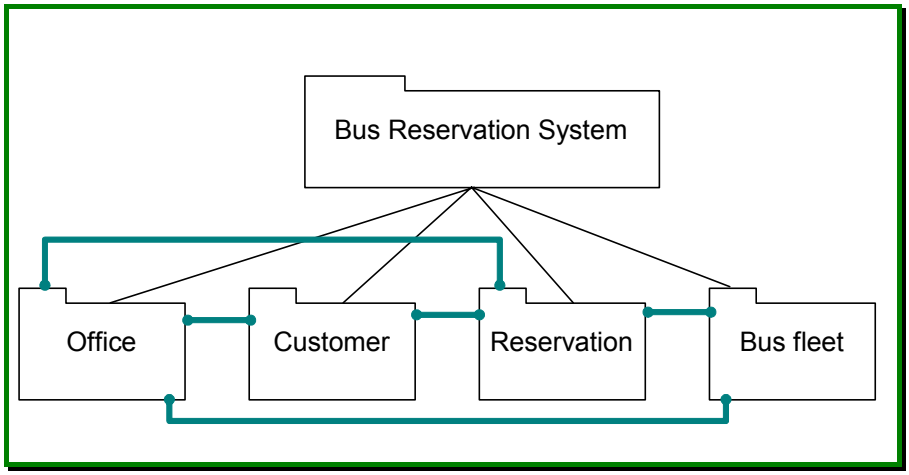


Figure 9.6. First level decomposition for the Bus Reservation System

The case study continues with further decomposition and reviewing the current specification. Packages defined in the initial decomposition step will be further divided. The Office package declares a data abstraction (personnel list). A specification does not have to be complete to provide useful feedback to the developers. The partial specification that is about the office will be left like this for now: currently only one abstraction stands for the office abstraction. In such a situation, there is no need to have two abstractions. Only one would do the representation. Anyway, let us continue the refinement with other sub-systems. By the way, a sub-system is one of the highest-level components of a system. Reservation package maintains lists for locations (origination and destination for trips), trips, and reservation records. Sale, return, and reservation operations all require reservation records. Actually, even if sold a ticket is still said to correspond to a reservation in this domain contrary to the immediate feeling that a reservation becomes obsolete once a ticket is bought, ticket information is valid rather than reservation. On the Bus Fleet package side a list of busses should be maintained where every bus is capable of drawing its own seat layout. Further, somewhere there should be the capability to save the assignment information of a driver to a bus.

The modularity principles set forth even before OO modeling, are universal. Another important concept a component developer and consequently, a CO developer should keep in mind is encapsulation. With such consciousness,

the decomposition would naturally determine modules that are both cohesive, thereby readily wrapping related items together.

In this case study, the reservation package should save the seat status information for any trip. Since the reservation concept is a bundling of the origin, destination, date, time, and seat number, all this information should be accessible from a reservation record. Starting with reservation point of view, one might suggest then, all this information should be modeled as sub-components, or attributes of a reservation entity. Now the coupling principle would imply a similar avenue because if all the required pieces of information are encapsulated, there will not be a need for communications between a reservation component and others that carry such information: all are contained locally. On the other hand, the idea of structuring such a big package is against the divide-and-conquer understanding. If this works, a big part of the problem will be resolved but there are little chances that this big granularity component will work exactly as the specific problem requires. The developers will try to match a big component as such and if it does not work, continue the decomposition so that smaller-grained components may be connected according to the specific needs. Further, it may be a single sub-component such as a "seat" that is offered in versions through different components: if the incompatibility of the large-grained "reservations component" is due to the differences in the "seat" requirements, selecting the fittest "seta component" may be the suitable solution. Let us take another component for our modularity analysis that is the bus. Who will draw the seat layout is a good question to guide the modularity here. The drawing should exert different colors for the reservation status (free/reserved/sold) so that it can be related with the "reservation" item. On the other hand, "guarded variables" principle indicates that it should be the only methods in the bus object that access the seats in a bus – no other entity than the bus should be allowed to draw the interior of its self. At the same time, it is the duty of a bus to draw its layout. Although drawing depends on the reservation status, the bus utilizes this information while painting a seat layout.

The case study moves towards decomposition, trying to observe the considerations discussed in the preceding paragraphs. Figure 9.7 displays the subs-systems and their initial decomposition. Here, reservation is a major component that contains a data abstraction, which records reservation items with other data abstractions to maintain the lists of trips and locations. The ticketing process at the office is modeled as a function abstraction inside the reservations package. On the other hand, it is unlikely that different bus types and models would be part of such a reservation package. Although we might want those two entities to be closer for reducing coupling, we may be disturbing cohesion. The bus fleet is modeled as a separate entity that contains a collection of busses and the "drawLayout" functional abstraction. Considering the specific mentioning of the drawing of the seat layout and

knowing the importance of this action for the domain, a specific abstraction is dedicated for layout drawing.

Of course, it would be better to have more defined before the starting of the decomposition. This chapter gave a very brief description to enable the presentation of the approach as early as possible. The details are explained as the decomposition progresses. Actually, it is also possible to continue requirements gathering with fast prototyping tools that would allow the drawing of the decomposition while the requirements are being acquired and analyzed.

At this time, the office contains only one abstraction that is the personnel data abstraction. We could opt for representing either the package or the data abstraction under the system but it is obvious that office will have more components later. Further, we want to indicate that among the components of the office, personnel structures are an important issue to represent. All decisions made so far assume that a good set of corresponding components will be located.

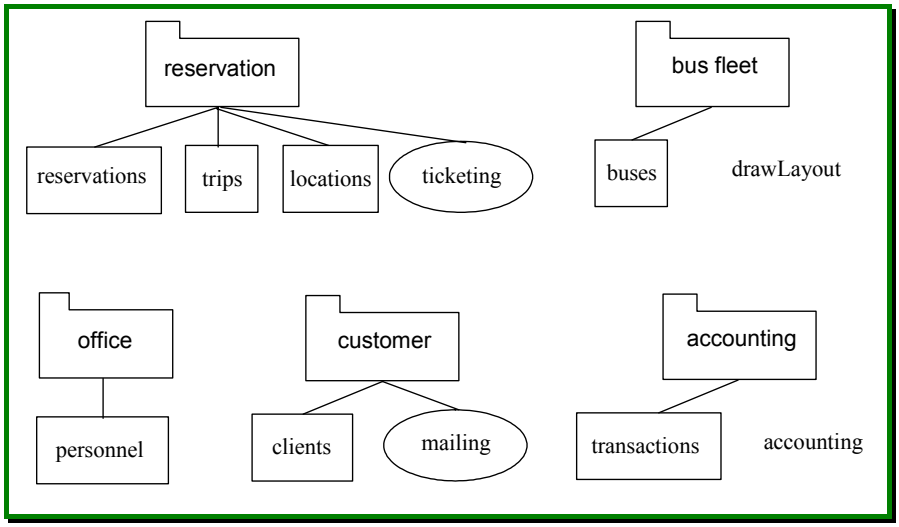


Figure 9.7. Decomposition after the second step

The higher-level main building blocks have been defined, so far, in a static structure. It may be a good time to stop and see if the specification complies with the requirements. Further, any investigation at the early stages is valuable because discovering errors now, is very important to make a better start. Besides, it is the suggestion of COSE to define connections when a partition is made. The connectors will help in enacting scenarios and hence tracing the functional requirements through the bundled messages. For this kind of an articulation, a dynamic model is required. Collaboration analysis, as defined in UML will be applied in the following sections. An example system function will be selected and dynamic model analysis will be

conducted on it. The reservation operation seems to be the most involved, it will be traced using collaboration models in both logical and component (presented in the next section) levels.

The logical level collaboration model includes the abstractions and messages that were not declared yet. The connectors have not been declared yet. Neither have been any messages. Perhaps it is time for experimenting with a set of new messages in an effort to enact the system function. With the given set of abstractions, if different message insertions cannot solve the problem, it is worth reconsidering the decomposition (set of involved abstractions). Figure 9.8 presents the logical-level collaboration diagram for the reservation operation. Actually displaying connections internal to a package is optional and especially not encouraged in a greater picture. Since our focus is within the context of the reservations package, we need them for understandability. The reader is reminded of the fact that selective displaying of any declared elements including connectors and messages is suggested for various views of the same model.

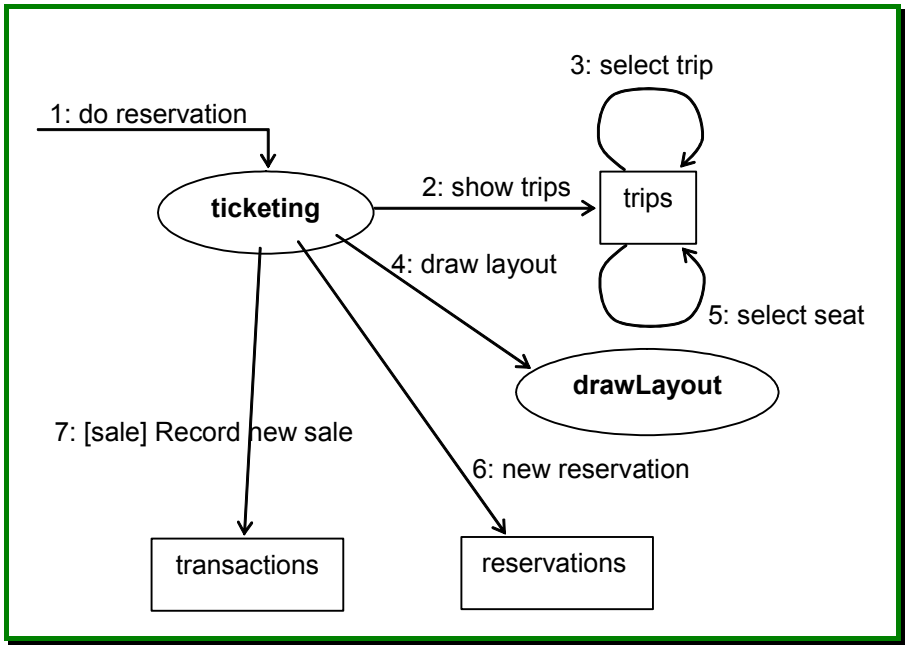


Figure 9.8. Logical-level collaboration model for the reservation operation

The objective of this case study is not a complete specification of the bus travel reservation system. Such a model could easily grow out of the limits for a chapter. For that reason, we will leave the refinement on the reservation scenario and continue with a different category of development that is the inclusion of components. After the components are identified, it will also be possible to demonstrate the above presented collaboration

through physical level modeling – a run time collaboration view that includes messages among the components and their interfaces.

Implementation by Components

It is assumed in the initial decomposition attempt for this case study that there exists components with names resembling the packages introduced before in Figures 6 and 7. With this optimistic expectation on components, it is further assumed that there are interfaces owned by components per connection. Finally, messages declared in the logical collaboration diagram presented in Figure 9.8 are also declared in the interfaces of these available components. Figure 9.9 depicts the components that correspond to the specification done so far.

A more realistic development would definitely confront many problems. Rather than the simplistic case assumed above, there might be many levels of decomposition and, even then, so many abstractions may not be satisfied with the available component set. Although it will not be possible to fit a realistic development here, some more complexity is not difficult to include in the case study. We may pick the "office" package for example, and further refine it to include sub components. Later development scenario in this chapter includes more entities inside the office (as shown in Figure 9.12). By the way, currently, an independent accounting package is also included inside the office later.

Figure 9.9 introduces the components for the first time, in the case study. They declare interfaces (per connection) with specific names to relate with the connection. Such assumed naming and interface structure is for a smooth presentation of the case study. It is not difficult to figure out that reality may be very different; so much refinement may be required to make the components fit to the need. Details of the interfaces introduced in Figure 9.9 are presented in Figure 9.12. These interfaces declare "request methods" as well as "service methods." A message is represented as a directed connection between a requesting and a serving pair of methods. The origination of the message optionally can be from the component – without indicating where exactly in the component the call is initiated. Figures 10, 11, and 12 include real messages at component-level. These messages can be numbered if collaboration information is also being presented. Of course, the overall model corresponds to all system functions hence all possible collaboration models. We prefer representing collaboration on a separate diagram, still better on a selected set of items rather than the whole decomposition. Names of the messages are the same as the method names at the end of the message arrows; that is the serving methods.

The issue of excluding messages or connections internal to a package has been discussed before. Here, the similar issue is also reflected on a component rather than a package. In a context where the model is supposed

to visualize the connection of the components, an intra-component message cannot be thought of representation. However, if a collaboration is being studied, to follow the sequence of events in order to trace a scenario, the developers might want to see all events – messages between component pairs and also messages originating and terminating on the same component. The so- called reflexive messages indicate an action taken by a component mostly as a result of receiving another message. Figure 9.9, presenting a collaboration view includes reflexive messages while Figure 9.12, representing the overall composition hides them.

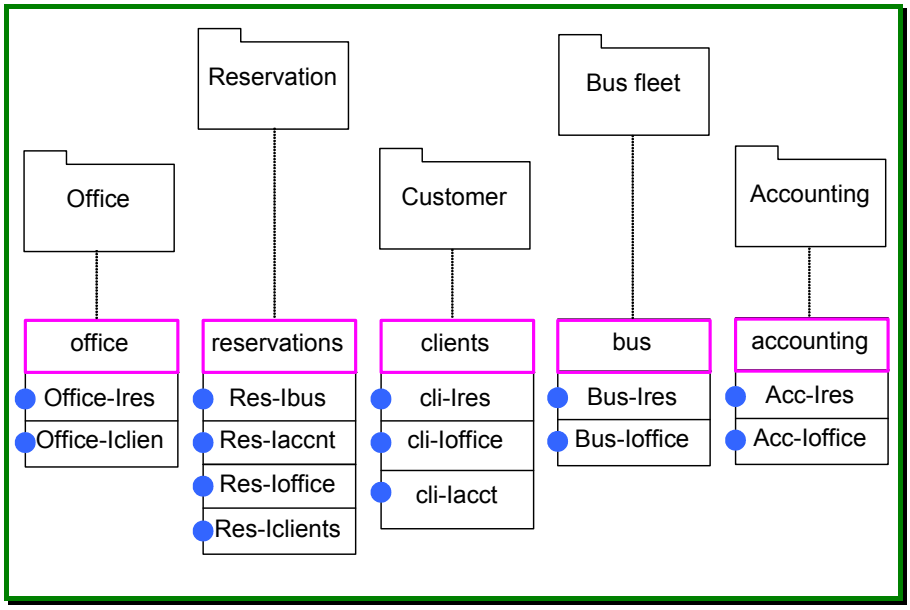


Figure 9.9. Components corresponding to abstractions

For any view, any message that is actually between the included decomposition of a component has to be represented as a reflexive message, if the decomposition is hidden. If the component displays further sub-components then messages can be shown to leave a sub-component (or its interface) and arrive at another sub-component (or its interface). The reflexive messages shown in Figure 9.10 are:

- show trips,
- select trip, and
- select seat.

If the reservations component represents the internal entities of the reservation abstraction, this is the only way. If the sub-components of the reservation abstraction were also represented by separate components, then some of those messages could as well be represented non-reflexive. The

show trips message would not be a reflexive message if ticketing function and trips data abstractions were implemented through independent components. None of these messages crosses the boundary of the reservation component and this can be visualized easily in Figure 9.10.

It is also possible to draw a component-only version of the model shown in Figure 9.12 will be useful for information hiding during the composition activity: once the abstractions are determined in the decomposition and mapped to the components, we can focus on the components and how they compose to form the executable system.

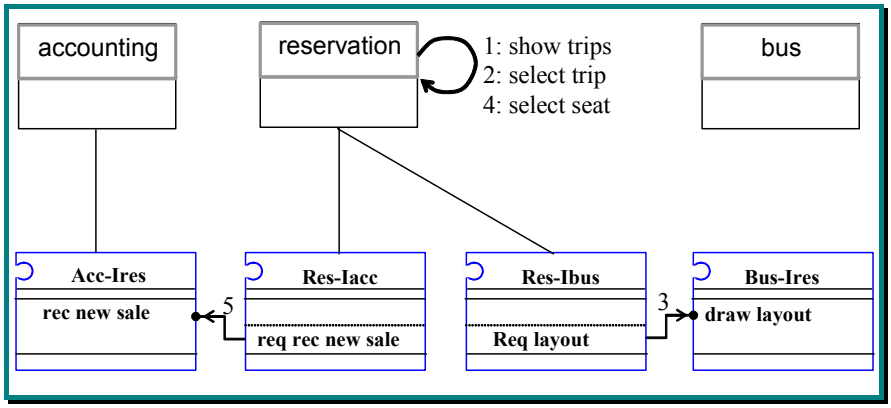


Figure 9.10. Component-level collaboration diagram for the reservation operation

The case study is about finished as far as reaching the lowest-level specifications. Interfaces and their connections are the lowest-level entities. There may be other work to refine the design. Starting with the next section some possible refinement to the initial model is being investigated. The problem has not been represented anywhere close to a complete specification but a taste of different kinds of development is contained. Selective representation of the defined units has been illustrated in different figures for collaboration analyses and also for limited abstraction level views for higher and lower levels.

Figure 9.10 contains some reflexive messages that originate and terminate at a component, rather than interfaces. This is fine but also it hints that there could be a more specific model for better understandability, better specification, or better correspondence to the reality. Actually, origination of a message is of secondary importance, but destination is of more. Not showing the originations means that activities are handled within the component without having to interact with the environment. In this example, the activities indeed involve the customer. It may be better to provide an interface for the customer, specifically of a Graphical User Interface (GUI) kind. Originators of the messages on a GUI would naturally be external "human" kind of entities. This is another place where a Use-Case element

namely an actor can be imported from the UML graphics. Figure 9.11 displays the refined alternative for the reservations component already shown in Figure 9.10, with the added GUI interface and with the modified collaboration. Since only the modified parts are selected, destinations of some messages cannot be represented in this diagram, which are drawn as round interface ports.

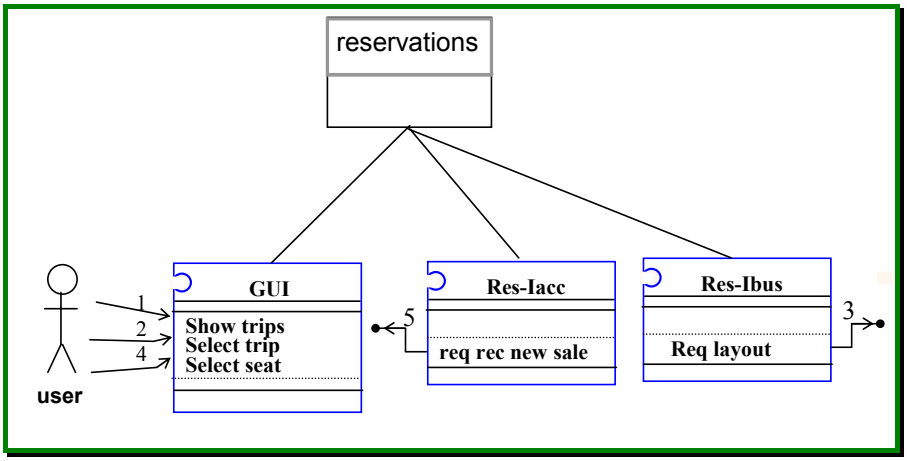


Figure 9.11. Refined reservations component in the same collaboration

No matter how crowded, it is usually desired that the complete picture be present some place, at some time. Usually, the developers paste sheets on a wall and draw connections across the sheets, for even various structured or Object-Oriented models. The inclusive model for the information supplied so far is represented in Figure 9.12. When the development is converging towards the complete system, this figure can easily get over crowded.

Documentation is a very important issue in any kind of development. Standards set forth formats for requirements, design or other documents. Such documents contain mainly text but they now heavily depend on the graphical models. To fit different graphics into documents page boundaries, extra measures are needed. Hierarchical organization of a model and representing a portion of the hierarchy in one page of course helps. Even if it is possible to decompose the model format, there will be problems at least for connectors that cross the boundaries for such portions (or pages corresponding to such model sections). A picture distributed across pages can be linked with connectors. The connectors will discontinue at the page boundaries exactly at the same location a special icon can be drawn to act like a terminal that connects to a matching terminal on a different page.

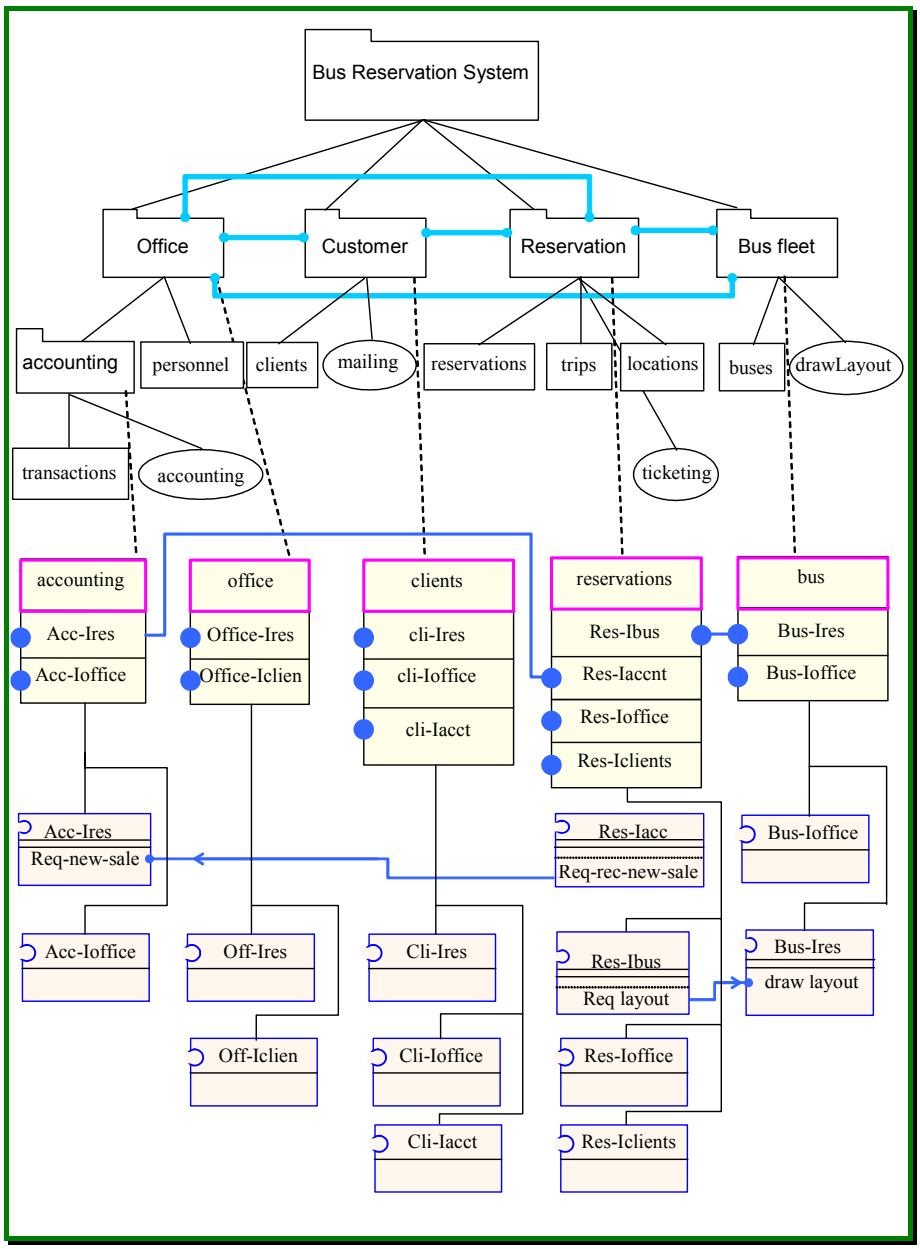


Figure 9.12. Complete model for the specification done so far

Scenario Changed

So far, a small example has been demonstrated displaying a history, which records, no surprises. Following the experience gained through decades of evolutionary processes, we should anticipate some change. In the case study

this change will occur in the development related stage (that could as well have happened in the requirements). In this section, an alternative (especially further detailed) decomposition of a segment in the previous model will be investigated; the decomposition will not match the available components. Thus, consequently it will be revised. As can be seen in Figure 9.13, the bus fleet package is decomposed in more detail where finally the developers expect to find components (busses, b1, seats, seat, and b1-reg) available.

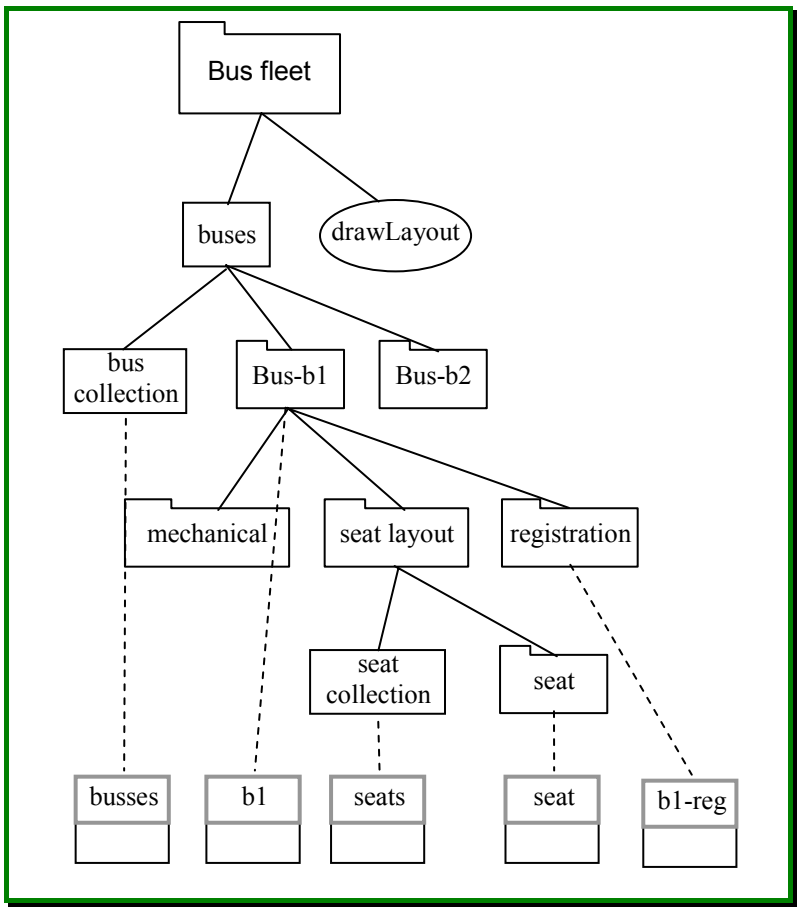


Figure 9.13. Alternative decomposition of the bus fleet package.

The busses component is a collection data structure, especially adapted to hold a list of busses. Through this component, a new bus can be created, existing busses can be deleted or selected for modification and it is possible to view and print the list of busses. The b1 component is to represent a "b1" brand of a bus. The b1 component actually represents the Bus-b1 package, which also is represented by "seats" and the "seat" components. The meaning of this compound represents relation is as follows: All the

requirements related with the Bus-b1 package except for those represented by “seats” and “seat” components, will be represented by the “b1” component.

According to the challenging new scenario, the set of components that are expected to exist are found to be non-existent. Instead, a different set is found to be available:

- b1-list,
- b1-draw,
- registration.

The b1-list component is capable of replacing the “busses” and the “b1” components together. The b1-draw component is also a specific component for the b1 brand busses and is responsible for any visualizing functions; it accommodates all the features represented in the “seats” and the “seat” components, together. It can also draw the external view of this particular vehicle, which can be used in animation pictures; but this later ability is not of our concern. Hence, this component may be more complex and expensive than what we need but this additional overhead is acceptable when compared to the cost and risks of developing from scratch. Finally, there seems to be no component developed to address the registration related information for this specific bus, rather the generic registration component can assume the mission for any kind of a vehicle.

Considering this existing set of components, a bottom-up adjustment is made to the decomposition shown in Figure 9.13. The new decomposition relates to the part of the model that is below the “busses” package. Figure 9.14 displays only the changed part that was listed under the “busses” data abstraction.

To start with the modification, it must be noticed that the changes relate to different levels in the previous decomposition: The “bus collection” data abstraction and the “bus-b1” package in the previous decomposition are not practical any more because a component that represents both definitions is available. Actually, it is legal in COSEML to allow one component to implement the functionality of more than one abstraction. Such a picture may be preserved if the developers need to represent the relations among such detailed abstractions. Conversely, the abstractions may be combined to achieve consistency with the component-level representation. Further, it is highly unlikely to desire to represent the information internal to a component, at higher abstraction levels.

The next change addressed in this paragraph is the unification of the definitions represented by the “seats” and the “seat” components. Immediately above those components, “seat collection” data abstraction and

“seat” package can also be united. If this uniting is decided, the container package above (seat layout) will be rendered redundant.

Finally, the newly found “registration” component corresponds to all brands of busses. Therefore, its peer modules are bus-b1 and bus-b2. In the abstract levels, this component should be represented either at this peer level or may be the one above.

This bottom-up modification only considers the mentioned components. The b2 brand busses are not mentioned. A similar problem could occur for them also. Here, the “bus-b2” package is not modified. If it were also decomposed inefficiently and exact components would not match then a similar revision could be carried out for this region in the decomposition diagram.

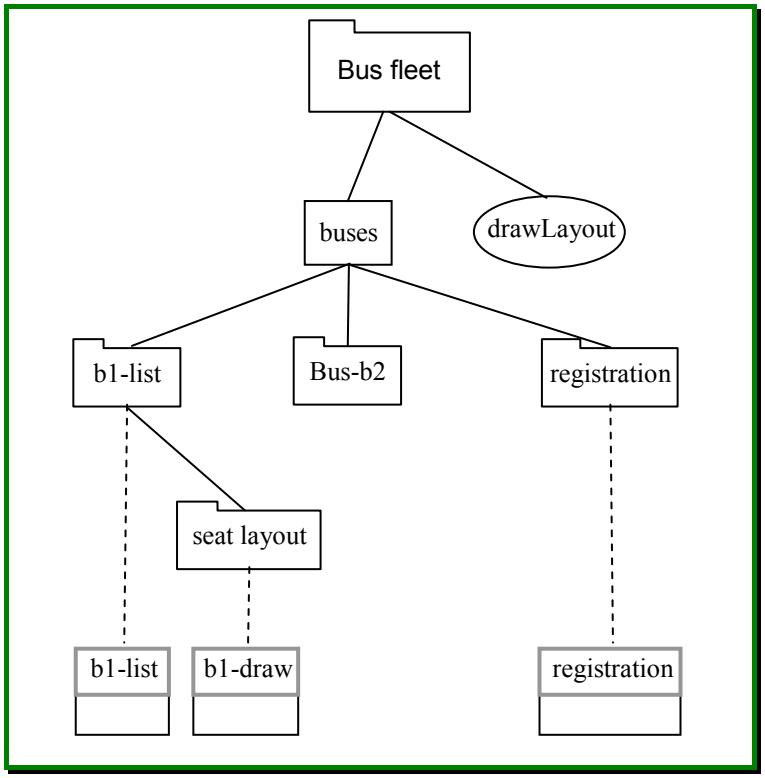


Figure 9.14. The bottom-up modification of the decomposition to match existing components

Index

References

Aktas, 1987	Ziya Aktas, 1976, *Structured Analysis and Design of Information Systems,* Prentice Hall.
Albrecht 1979	A.J. Albrecht, "Measuring Application Development Productivity," *IBM Application Development Symposium,* Monterey, California, October 1979.
Albrecht 1983	A.J. Albrecht and J.E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," *IEEE Transactions on Software Engineering*, November 1983, pp. 639-648.
Altintas, 2001	Ilkay Altintas, 2001, A Comparative Study for Component Oriented Design Modeling, M.S. Thesis, Computer Engineering Department, Middle East Technical University, May, Ankara, Turkey.
Arrango 1994	G. Arrango, "Domain Analysis Methods," in *Software Reusability*, W. Shcaefer, R. Prieto-Diaz, M. Matsumoto (editors), Ellis Horwood, 1994.
Avkaroğulları, 2004	Okan Avkaroğulları, 2004, Representing Design Patterns in Component Oriented Design, M.S. Thesis, Middle East Technical University.
Baker 1972	F.T. Baker, "Chief Programmer Team Management of Production Programming," *IBM Systems Journal*, Vol 11, No. 1, 1972.
Bayar, 2001	Bayar V., 2001, A Component Oriented Process Model, M.S. Thesis, Middle East Technical University.
Boehm 1981	Barry Boehm, *Software Engineering Economics,* Prentice Hall, 1981.

Booch 1994 — G. Booch, *Object-Oriented Analysis and Design*, 2nd Edition, Benjamin Cummins, 1994.

Booch et al. 1999 — Booch G., Rumbaugh J., Jacobson I., 1999, *The Unified Modeling Language User Guide*, Addison-Wesley.

Brown and Wallnau, 1998 — Alan W. Brown and Kurt C. Wallnau, 1998, "The Current State of CBSE," *IEEE Software*, September-October.

Chen 1977 — P. Chen, *The Entity-Relationship Approach to Logical Database Design,* QED Information Systems, 1977.

Christiansen 1989 — M. Christiansen, Integrating Domain Knowledge into Software Components, Ph.D. Dissertation, Southern Methodist University, Dallas, Texas, 1989.

Coad and Yourdon 1991 — P. Coad and E. Yourdon, *Object Oriented Analysis*, 2nd edition, Prentice-Hall, 1991.

Coleman et al. 1994 — D. Coleman et al., *Object-Oriented Development: The Fusion Method*, Prentice-Hall, 1994.

D'Souza, 1998 — Desmond Francis D'Souza, Alan Cameron Wills, 1998, *Objects, Components, and Frameworks With UML: The Catalysis Approach*, Addison-Wesley.

Dennis 1973 — J.B. Dennis, "Modularity" in *Advanced Course On Software Engineering,* F.L. Bauer (ed.), Springer-Verlag, New York, 1973.

Diaz 1987 — Ruben, Prieto-Diaz, "Domain Analysis for Reusability," *COMPSAC 87: The Eleventh Annual Computer Software and Applications Conference*, October pp: 23-29, 1987.

Dogru 1999 — Ali Dogru, 1999, "Component Oriented Software Engineering Modeling Language: COSEML," *Technical Report TR-99-3,* Computer Engineering

Department, Middle East Technical University, Ankara, Turkey.

Dogru and Altintas 2000 — Ali. H. Dogru, Ilkay. Altintas, "Modeling Language for Component-oriented Software Engineering: COSEML," *The Fifth World Conference on Integrated Design and Process Technology*, June 4-8, Dallas, Texas, 2000.

Dogru and Tanik 2003 — Dogru A., Tanik M.M., 2003, "A Process Model for Component Oriented Software Engineering," *IEEE Software*, Vol 20, No. 2, March/April, pp. 34-41.

Dogru et al. 1992 — A. H. Dogru, S. N. Delcambre, C. Bayrak, Y. T. Chen, E. S. Chan, W. Yin, M. G. Christiansen, and M. M. Tanik, "An Integrated System Design Environment: Concepts and a Status Report," *Journal of Systems Integration*, October, 2(4), pp. 317-347, 1992.

D'Souza, D.F. and Wills 1998 — D'Souza, D.F. and Wills, A.C. *Objects, Components, and Frameworks With UML: The Catalysis Approach*. Reading, Massachusetts: Addison-Wesley, 1998.

Dursun and Dogru 1995 — Huseyin Dursun, Ali H. Dogru, "Prototyping Specifications through Visualization," *The First World Conference on Integrated Design and Process Technology*, December 8-9, Austin, Texas, Vol. 1 pp:362-368, 1995.

Fayad 2000 — Mohamed E. Fayad, "Introduction to the Computing Surveys' Electronic Symposium on Object-Oriented Application Frameworks," *ACM Computing Surveys,* Vol. 32, No. 1, March: pp. 1-11, 2000.

FunSoft 2001 — Funsoft users manual, Funsoft, 2001, Austin, Texas.

Gamma et al. 1995	Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, Reading, Massachusetts, 1995.
Hatley and Pirbhai 1987	D.J. Hatley and I.A. Pirbhai, *Strategies for Real-Time System Specification,* Dorset House, 1987
Heineman and Councill 2001	George T. Heineman and William T. Councill, *Component-Based Software Engineering,* Addison Wesley, 2001.
Herzum and Sims, 2000	Peter Herzum and Oliver Sims, *Business Component Factory*, Wiley, 2000.
Holibaugh 1993	Robert Holibaugh, Joint Integrated Avionics Working Group (JIAWG) Object-Oriented Domain Analysis Method (JODA), CMU/SEI-92-SR-3, November, Pittsburgh, Philadelphia: Software Engineering Institute, Carnegie Mellon University, 1993.
Hopcroft and Ullman, 1979	John E. Hopcroft, Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 1979.
Itoh et al. 1998	Kiyoshi Itoh, Toyohiko Hirota, Satoshi Kumagai, Hiroyuki Yoshida (editors), *Domain Oriented Systems Development: Principles and Approaches*, Information Processing Society of Japan, Gordon and Breach Science Publisher, Japan, 1998.
Jacobson, 1992	I. Jacobson, *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
Kang et al. 1990	Kyo C. Kang, Sholom C. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, CMU/SEI-90-TR-21, ADA 235785, Pittsburgh, Philadelphia: Software Engineering Institute, Carnegie Mellon University, 1990.

Kang et al. 1998 — K. Kang, Kim, S., Lee, J., Kim, K., Shin, E., Huh, M., "FORM : A Feature Oriented Reuse Method with Domain-Specific Reference Architectures", *Annals of Software Engineering*, Volume 5, J. C. Baltzer AG Science Publishers, Red Bank, NJ, USA, pp. 143-168,1998.

Krieger and Adler 1998 — Krieger D., Adler R.M., "The Emergence of Distributed Component Platforms", *IEEE Computer,* March, 1998.

Manzer 2002 — Ayesha Manzer, Formalization of Core-Competency Processes for Integration of Value-add Chains, PhD. Dissertation, Middle East Technical University, July 2002.

McCabe 1976 — T. McCabe, "A Software Complexity measure," *IEEE Transactions on Software Engineering*, Vol. 2, December 1976.

McCall 1977 — J. McCall, P. Richards, G. Walters, Factors in Software Quality, NTIS AD-A049-014, 015, and 016, November 1977.

Muller 1997 — P.A. Muller, *Instant UML*, Wrox Press, Birmingham, Canada, 1977.

Neighbors 1989 — J.M. Neighbors, "DRACO: A Method for Engineering Reusable Software Systems," *Software Reusability,* Vol. 1, pp: 295-320, ACM, 1989.

Paulk et al. 1994 — M.C. Paulk, C.V. Weber, B. Curtis, M.B. Chrissis, *The Capability Maturity Model: Guidelines for Improving the Software Process,* Carnegie Mellon University Software Engineering Institute, Addison-Wesley, 1994, Reading, Massachussetts.

Prather 1997 — R. Prather, "Regular Expressions for Program Computations," *The American*

Mathematical Monthly, Vol. 104., No. 2, pp. 120-130, 1997.

Pressman 1997 — R.S. Pressman, *Software Engineering: A Practitioner's Approach*, 4th Edition, Mc-Graw Hill, 1997.

Rambaugh et al. 1991 — J. Rambaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

Riebisch 2003 — Matthias Riebisch: "Towards a More Precise Definition of Feature Models." Position Paper. In: M. Riebisch, J. O. Coplien, D, Streitferdt (Eds.): *Modelling Variability for Object-Oriented Product Lines*. BookOnDemand Publ. Co., Norderstedt, 2003. pp. 64-76.

Salman 2002 — Salman, N. "Extending object oriented metrics to components." *The 6th World Conference on Integrated Design and Process Technology*. Pasadena, California, June 23-28, 2002.

Simon 1969 — H.A. Simon, *Sciences of the Artificial*, MIT Press, Cambridge, Massachusetts, 1969.

Simos 1996 — M. Simos, "Organization Domain Modeling (ODM): Extending Systematic Domain Analysis and Modeling beyond Software Domain, *IDPT,* 1996.

SPC 1990 — Software Productivity Consortium, *A Domain Analysis Process* Domain_Analysis-90001-N, January, Herndon, Virginia, 1990.

Szypersky 1998 — Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming,* Addison Wesley, New York, 1998.

Tanik and Chan 1991 — Murat M. Tanik and Erik S. Chan, 1991, *Fundamentals of Computing for*

Software Engineers, Van Nostrand Reinhold, New York.

Tanik and Ertas 1992 — M.M. Tanik and A. Ertas, "Design as a Basis for Unification: System Interface Engineering," *ASME PD-Vol. 43*, pp: 113-114, 1992.

Tanik and Ertas 1997 — M.M. Tanik and A. Ertas, "Interdisciplinary Design and Process Science: A Discourse on Scientific Method for the Integration Age," *Journal of integrated Design and Process Science,* September, Vol. 1 No. 1: pp. 76-94, 1997.

Wallnau et al. 2002 — Kurt C. Wallnau, Scott A. Hissam, and Robert C. Seacord, *Building Systems from Commercial Components*, Addison Wesley, 2002.

Ward and Mellor1985 — P.T. Ward and S.J. Mellor, *Structured Development for Real-Time Systems,* Yourdon Press, 1985.

Yin 1988 — Weiping Yin, An Integrated Software Design Paradigm, Ph.D. Dissertation, Southern Methodist University, Dallas, Texas, 1988.

Yourdon 1989 — E.N. Yourdon, *Modern Structured Analysis,* Prentice-Hall, 1989.