Candidate Model Problems in Software Architecture

Mary Shaw, David Garlan, Robert Allen, Dan Klein, John Ockerbloom, Curtis Scott, Marco Schumacher

The Software Architecture Group
Computer Science Department
Carnegie Mellon University
Pittsburgh PA 15213

Version 1.3: January 1995

Invitation: The software architecture community would benefit from sharing a set of standard example problems. These would improve our ability to work out ideas, exhibit techniques, and compare results. The Software Architecture group at Carnegie Mellon has been assembling such a collection of problems. With this draft report we would like to open a discussion about suitable problems: what characteristics they should have, what specific problems would serve us well. To start that discussion, we present ten candidate problems and sketches of several distinct architectural approaches to two of them. We invite refinements and discussion of the problem list, the solution sets, and the criteria for choosing problems.

1 Introduction

It is common for a discipline, especially one that is just getting its wits about itself, to adopt some shared, well-defined problems for teaching and study. Often known as *model systems* or *type problems*, they provide a way to compare methods and results, work out new techniques on standard examples, and set a minimum standard of capability for new participants. In time, a reasonable approach to some of these problems becomes the price of admission to get serious consideration of a new technique. Model problems also provide a pre-debugged source of educational exercises.

Biology, for example, has

- *Drosophila melanogaster* (the fruit fly)
- *Rattus rattus Norwegicus* (the lab rat)
- *Escherichia coli* (the digestive bacterium)

Each of these is part of the common language of discourse in the field. Each provides a familiar concrete instance that illustrates an important set of issues. This allows discussions to start from shared knowledge of the basic example and proceed expeditiously to the result, theory, or technique of current interest.

Closer to home, computer science has model problems in many areas. Familiar examples include

- *Algorithms and Data Structures:* Sort, search, greatest common divisor, prime integers, set, stack, queue
- *Synchronization:* Reader/writer, producer/consumer, dining philosophers, cigarette smokers
- *Programming Methodology:* Eight queens, tower of Hanoi
- *Formal Specifications:* Telegraph, lift (elevator, on the west side of the Atlantic), library
- *Combinatoric Optimization:* Travelling salesman

In this report, we propose several model problems for software architecture, discuss the interesting design problems they raise, and show how some of the work in this group addresses each of them.

Our intention is to stimulate a discussion about these problems, potential additional problems, and the criteria for choosing problems and evaluating or comparing solutions. To that end, this is a living document. We are distributing it informally and encourage informal redistribution. We have made it available via anonymous FTP. We include a version number on the first page, and we do not plan any kind of "permanent" publication anytime soon. We will attempt to incorporate comments and suggestions, along with short sketches of solutions. We are open to suggestions about how longer solutions or comparison of alternative solutions should be handled.

Before moving on to the problems, we clarify what we mean by *software architecture* [GarlanShaw93, Shaw93; see also Perry-Wolf92]. System design takes place at many levels. It is useful to make precise distinctions among those levels, for each level appropriately deals with different design concerns. Software design includes at least the following:

- *Architecture*, where the design issues involve overall association of system capability with components.
- *Code*, where the design issues involve algorithms and data structures.
- *Executable*, where the design issues involve memory maps, call stacks, and so forth.

Software architecture is concerned with design at the system level. Certainly this includes system structure (or topology), discriminations among different kinds of structures, and abstractions or generalizations about structures and families of similar structures. It also includes identification, specification, and analysis of the properties that are related to these structures, either because they influence the selection of a structure or because they are consequences of that structure.

At the architecture level, the components of interest are modules and the interconnections among modules. Architectural styles guide the selection of kinds of components and of the strategies for composing them. As a result, the kinds of components and interconnections can differ substantially between architectural styles. The properties of interest include system structure, gross performance, component consistency, and other aggregate properties such as security and reliability.

Model problems for software architecture should help us focus on specific architectural issues. Such issues include

- Describing system organizations, and describing specific kinds of system organization (architectural styles)
- Distinguishing among templates, instances, and invocations
- Distinguishing among different kinds of system organization -- not only structural differences, but the implications of those differences
- Selecting among different architectural alternatives
- Using different models concurrently, or at different refinements of a design; establishing consistency among such different views
- Defining families of systems
- Defining families, or styles, of architecture
- Describing dynamic behavior of systems with fixed structure and desribing dynamic changes in system structure
- Measuring, evaluating, or testing properties of systems such as overall performance, reliability, or security
- Measuring, evalutaing, or testing properties of designs such as ease of extension or subsetting

Different problems may, of course, be selected in response to different issues. We have not tried to make the problems independent or orthogonal. It's fine if they overlap, but as the set is refined, each should include a description of the specific issues it helps to clarify.

The remainder of the paper has three parts. First, it presents brief statements of all the problems. Second, it presents sketches of solutions based on different architectures for two problems, *Keyword in Context* and *Mobile Robot*. These examples focus on the choice of an overall architecture for the problem; they identify several candidates architectures and compare the merits of the alternatives. They attempt to provide enough detail to compare designs but not so much as to drown the reader. A companon paper [Shaw94] provides an extended comparison of published solutions for *Cruise Control*. Third, it gives an extended specification of the Calendar Scheduler problem [vanLamsweerde92,93]. This specification comes to us much in the manner of a requirement definition: it is the result of an exercise in the specification community.

The problems are:

- *Keyword in Context (KWIC):* Given a set of lines, create an alphabetized list of the rotations of those lines.
- *Sea Buoy:* Collect and transmit weather data both automatically and on demand; allow preemption for emergency services.
- *Cruise Control:* Maintain the speed of a vehicle.
- *Conference Refereeing:* Solicit, referee, and select papers for a conference.
- *Mailing List Handler:* Merge address information from multiple sources, eliminating duplicates and observing reader preferences.
- *Printer Spooler:* Manage print jobs within a printer network.
- *Library:* Automate traditional library tasks, such as check-in and check-out of books.
- *Automated Teller Machine (ATM):* Provide the usual banking functions with a remotely-located machine.
- *Calendar Scheduler:* Organize a meeting schedule.
- *Compiler:* Translate source code for a programming language to executable form.
- *Mobile Robot:* Design a mobile robot capable of executing tasks while monitoring the environment, e.g., avoiding obstacles.

1.1 Keyword In Context (KWIC)

From Parnas [Parnas72] we have a concise definition of the *Keyword in Context* problem:.

> The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

1.1.1 History

Contextual indices have been used for many years. For example, Biblical concordances have approximately this form, except for the rotations. The usual source for the problem as now known, however, is the Parnas definition.

In his paper of 1972, Parnas used the problem to contrast different criteria for decomposing a system into modules [Parnas72]. He describes two solutions, one based on functional decomposition with shared access to data representations, and a second based on a decomposition that hides design decisions. The latter was used to promote information hiding, a principle that underpins the use of abstract data types and of object-oriented design. Since its introduction, the problem has become well-known and is widely used as a teaching device in software engineering. Garlan, Kaiser, and Notkin also use the problem to illustrate modularization schemes based on data-driven tool invocation [Garlan92]—sometimes referred to as reactive integration.

While KWIC can be implemented as a relatively small system it is not simply of pedagogical interest. Practical instances of it are widely used by computer scientists. For example, the "permuted" [sic] index for the Unix Man pages is essentially such a system.

We use the problem in a course on software architecture to give students experience with software development in different architectural styles [GarlanShaw94]. We give three separate assignments. Each starts with a simple KWIC indexer, for which we supply code, and asks for modifications. By providing an initial implementation, we give them an example of a small system in the style of interest and get them started in the right way. Each exercise requires the modifications to be done in a way that preserves the style. As part of the assignments, students analyze the suitability of different styles for different variants on the basic problem.

1.1.2 Design Considerations

From the perspective of software architecture, the problem derives its appeal from the fact that it can be used to illustrate the effect of changes on software design. Parnas shows that different problem decompositions vary greatly in their ability to withstand design changes. Among the changes he considers are:

- Changes in algorithm: For example, line shifting can be performed on each line as it is read from the input device, on all the lines after they are read, or on demand when the alphabetization requires a new set of shifted lines.
- Changes in data representation: For example, lines can be stored in various ways. Similarly, circular shifts can be stored explicitly or implicitly (as index and offsets).

Garlan, Kaiser, and Notkin [Garlan92] extend Parnas' analysis by including enhancements to system function. For example:

- Have the system eliminate circular shifts that start with certain noise words (such as "a", "an", "and", etc.).
- Make the system interactive, and allow the user to delete lines from the lists.

Finally, it is worth considering differences in architectural solutions based on considerations of:

- Performance: Both space and time.
- Reuse: To what extent can the components serve as reusable entities.

1.1.3 Solutions

In section 2.1 on page 6, we outline four architectural designs for the KWIC system. All four are grounded in published solutions. The first two are those considered in Parnas' original article. The third solution is based on the use of "reactive integration" and represents a variant on the solution examined by Garlan, Kaiser, and Notkin. The fourth is a pipeline solution inspired by the Unix index utility.

1.1.4 Contributors

Two of the solutions are derived from [Parnas72]. Curtis Scott and David Garlan provided the other two solutions and arranged the presentation.

1.2 Sea Buoy

Sea buoys support navigation at sea. Here is the problem statement from [Booch86]:

> There exists a collection of free-floating buoys that provide navigation and weather data to air and ship traffic at sea. The buoys collect air and water temperature, wind speed, and location data through a variety of sensors. Each buoy may have a different number of wind and temperature sensors and may be modified to support other types of sensors in the future. Each buoy is also equipped with a radio transmitter (to broadcast weather and location information as well as an SOS message) and a radio receiver (to receive requests from passing vessels. Some buoys are equipped with a red light, which may be activated by a passing vessel during sea-search operations. If a sailor is able to reach the buoy, he or she may flip a switch on the side of the buoy to initiate an SOS broadcast. Software for each buoy must:
>
> - maintain current wind, temperature, and location information; wind speed readings are taken every 30 seconds, temperature readings every 10 seconds and location every 10 seconds; wind and temperature values are kept as a running average.
> - broadcast current wind, temperature, and location information every 60 seconds.
> - broadcast wind, temperature, and location information from the past 24 hours in response to requests from passing vessels; this takes priority over the periodic broadcast
> - activate or deactivate the red light based upon a request from a passing vessel.
> - continuously broadcast an SOS signal after a sailor engages the emergency switch; this signal takes priority over all other broadcasts and continues until reset by a passing vessel.

1.2.1 History

Booch used the sea buoy example to illustrate object-oriented development [Booch86]. He adapted his version from a study by Boehm-Davis and Ross [Boehm84].

From an architectural standpoint, the interesting problem lies in the different levels from which it can be analyzed. As the next section illustrates, maintainability, real-time factors, and hardware questions are all important considerations.

1.2.2 Design Considerations

The problem statement defines a set of separate functions with relatively little in common. They share the communications equipment and a number of current sensor readings.

The software architecture must permit the *integration of these loosely coupled functions* (requirement R1).

At the same time, it must *respect their priorities and timing constraints* (R2).

Clearly the system may be extended further by additional functions (e.g., more sensors) or that the priorities and timing constraints may be modified. The architecture should

therefore *allow modifications to the overall system parameters* (R3).

Finally, sea buoys must operate for long periods without maintenance, and they are numerous enough for cost to be a major consideration. As a result, the architecture should provide *hints for its implementation on the most basic platform* (R4).

1.2.3 Solutions

Booch provided an object-oriented solution in the same paper as the problem statement [Booch86].

1.2.4 Contributors

Marco Schumacher organized the presentation and drafted a solution (not included here).

1.3 Cruise Control

Cruise control has been used by a number of authors to illustrate software design methodologies. This problem statement is derived from the one Booch used to describe object-oriented programming [Booch86] and the one Birchenough and Cameron later used to compare JSD to OOD:

> A cruise-control system exists to maintain the speed of a car, even over varying terrain, when turned on by the driver. When the brake is applied, the system must relinquish speed control until told to resume. The system must also steadily increase or decrease speed to reach a new maintenance speed when directed to do so by the ddriver. Below (Figure 1.5.1), we see the block diagram of the hardware for such a system. There are several inputs:
>
> - *System on/off:* If on, denotes that the cruise-control system should maintain the car speed.
> - *Engine on/off:* If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.
> - *Pulses from wheel:* A pulse is sent for every revolution of the wheel.
> - *Accelerator:* Indication of how far the accelerator has been pressed.
> - *Brake:* On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.
> - *Increase/Decrease Speed:* Increase or decrease the maintained speed; only applicable if the cruise-control system is on.
> - *Resume:* Resume the last maintained speed; only applicable if the cruise-control system is on.
> - *Clock:* Timing pulse every millisecond.
>
> There is one output from the system:
>
> - *Throttle:* Digital value for the engineer throttle setting.

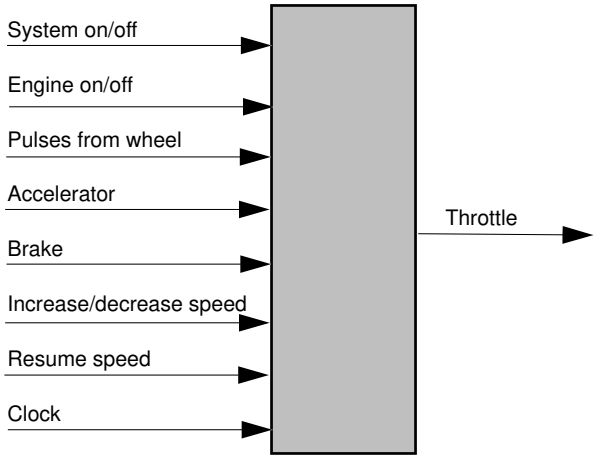

Figure 1.3.1: Block Diagram for Cruise Control.

1.3.2 History

1.3.3 Design considerations

A cruise control system provides autonomous (but casually supervised) control of the speed of a motor vehicle moving at highway speeds. For such a system, important design considerations include

- *Safety:* Can the system fully control the vehicle, and can it ensure that the vehicle will not enter an unsafe state as a consequence of the control?
- *Simplicity of use:* Can a driver with no prior experience with the system use it correctly?
- *Characteristics of real-time response:* How rapidly does the vehicle respond to control inputs?
- Other?

1.3.4 Solutions

The most familiar presentation is probably Booch's use to motivate object-oriented programming [Booch86]. Booch adapted his version from Ward [Ward84]. Yin and Tanik do an object-oriented solution to cruise control to demonstrate reusability in Ada [YinTanik91]. Wasserman and others also do an object-oriented design [Wasserman89]. Jones considers the testing problem for an Ada program but is not explicit about the character of the software [Jones90].

Birchenough and Cameron compare the Jackson System Development Method (JSD) to object-oriented design using a formulation similar to Booch's [Birchenough-Cameron89].

Smith and Gerhart use a slightly more elaborate forumulation to illustrate the use of Statemate. The design is, of course, based on states and activities [SmithGerhart88]. Their problem statement is based on one used by Bracket [Bracket87].

Ward and Keskar use cruise control as an example for comparing the Ward/Mellor and Boeing/Hatley Structured Methods techniques for modeling real-time systems. Both add time and control information to DeMarco Structured Analysis [WardKeskar87]. Gomaa also this example for studying real-time systems. He compares Structured Design and the NRL Software Cost Reduction methods [Gomaa89].

Higgins uses cruise control to show how Data Structured Systems Development can be extended for real-time [Higgins87]; his architecture emphasized feedback control models. Shaw also bases a solution on feedback control, with other architectures used for subsystems [Shaw95].

Wang and Tanik develop a dataflow solution to illustrate Process Port Analysis and XYZ/E [WangTanik89].

Atlee and Gannon use cruise control as the basis of a specification study [AtleeGannon93].

1.3.5 Contributors

Mary Shaw organized the presentation and prepared one of the solutions. She also prepared a comparison of published solutions [Shaw94].

1.4 Conference Refereeing

Professional conferences are held in order to announce and discuss new results. The core activity of organizing a conference centers on selecting the papers to be presented. Usually this is done by making an open invitation calling for papers to be submitted, circulating the submitted papers to a (geographically distributed) panel of reviewers, then selecting the best papers to appear on the program. A system to automate conference refereeing should do the following:

1. The program committee announces "call for papers."

2. Authors receive the call for papers and decide to will submit papers on their work.They write papers and send them to the program committee. A given paper may have several authors, but only one reply address.

3. The program committee registers the contributed papers upon receipt.

4. At a certain point in time the program committee distributes the papers among the panel of referees. Each paper is sent to three distinct referees, none of whom is an author of the paper.

5. The program committee continuously collects reports from the referees.

6. At a certain point in time the program committee selects papers for inclusion in the program andnotifies the authors about the selection. This may involve obtaining additional opinions from the referees.

7. The program committee advises the authors of the selection results.

1.4.1 History

This is a slight rewording and elaboration of the OOPSLA Conference Registration Problem proposed by Høydalsvik and Sindre at OOPSLA '93 [HøydalsvikSindre93]. They created it by simplifying an information system problem posed by Rumbaugh [Rumbaugh92].

1.4.2 Design Considerations

1.4.3 Solutions

Høydalsvik and Sindre provide an object-oriented solution [HøydalsvikSindre93].

1.4.4 Contributors

Mary Shaw brought the problem statement in from OOPSLA.

1.5 Mailing List Handler

We are all plagued with multiple or unwanted copies of catalogs and other mass mailings. These arise largely from merging multiple mailing lists, clerical errors in data collection, and raw information generated by individuals in different forms at different times. Ideally, a mailing list system would collect (even propagate) corrections, merge variant forms, and recognize reader preferences about receipt.

> The Mailing List Handler accepts address entries, corrections, and preferences to create one or more mailing lists. It generates mailing labels from the lists.
>
> An address entry contains a name, mailing address, and reader/supplier information. Corrections include updates to individual address entries and guidance about merging variants. Preferences update the reader/supplier information. A mailing list is a collection of address entries plus perhaps control information.
>
> Address entries may be original (collected from raw sources such as reader requests), or they may be derived from other mailing lists. Address entries may also be received as external mailing lists (not necessarily in the desired format). Corrections may come from internal consistency checks, post office correction procedures, reader information, or other sources. Preference information may come from readers, suppliers, or other sources (e.g., suppression information from Direct Marketing Association or USPS objectionable-mail procedures).
>
> The mailing list handler must maintain a set of mailing lists. It should eliminate duplicate entries and correct errors. When generating mailing labels it must take reader/supplier information into account.

1.5.1 History

This problem was proposed within the CMU group, so it has as yet no history.

1.5.2 Design Considerations

Costs are prime drivers of mailing list handling. Costs to consider include

- acquiring and using addresses
- eliminating duplicates
- sending duplicates
- violating mandatory suppression orders

Mailing lists from other sources may safely be assumed to be in an undesired format, incorrect, and incomplete.

1.5.3 Solutions

1.5.4 Contributors

Mary Shaw developed the problem statement after an extended discussion in the Software Architecture Reading Group at CMU.

1.6 Printer Spooler

Local area networks provide services for their users. Often the services are replicated for throughput, reliability, or physical convenience. Access for these services can be provided in a number of different ways that differ in such details as where the queues reside, how explicitly each user needs to specify the service, and the consequences of local failures.

> A network connects multiple computers and printers. Each printer is driven by one of the computers, provides service to the entire network, and is equipped with multiple paper trays. A program running on any computer may specify any paper tray on any printer for its print requests.

1.6.1 History

This problem was proposed within the CMU group, so it has as yet no history.

1.6.2 Design Considerations

This model problem raises configuration and fault tolerance issues. Site administrators may disable the use of paper trays for maintenance purposes. If a printer fails, one may conceive that its pending requests are rerouted to other printers and the originators notified of the destination change. It also raises issues of heterogeneity. Differ-

ent printers may have different capabilities, such as large paper, high resolution, or color. Further, some printers may be located in private space and hence have special status.

A software architecture appropriate for this network must, at the least:

- Support the distribution of the print services.
- Allow the reconfiguration of both hardware and software.
- Enable the fault tolerance permitted by the duplication of the hardware.

1.6.3 Solutions

1.6.4 Contributors

Dan Klein developed the problem statement.

1.7 Library

The library problem has served the formal specification community well [Wing88]. To use it as a software architecture problem, we'll focus on the possible structure of solutions rather than the specification of functionality.

A library requires an information system that provides the following on-line operations for library users and staff:

1. Check out (or return) a copy of a book.
2. Get a list of books by a particular author or on a particular subject.
3. Find out what books a particular borrower currently has checked out (users can only look up themselves).
4. Find out which borrower last checked out a particular copy of a book (staff only).
5. Record the addition (or removal) of a copy of a book to (from) the library (staff only).

The system must be able to search and update the catalog quickly (to avoid long check-out lines, and to make on-line book search a viable alternative to card catalogs), and easily handle updates and corrections by staff users to an potentially large collection.

The system must also enforce the following integrity constraints:

a. All copies in the library must be available for checkout or be checked out.

b. No copy may be both available and checked out at the same time.

c. Borrowers can't have more than a predefined number of books checked out at once.

d. Borrowers can't have more than one copy of a given book checked out at once.

1.7.1 History

The existing history of this problem has been with the specification community [Wing88].

On-line library systems like the one described above have been envisioned since at least the 1960s, when the US Library of Congress embarked on its MARC project. The concise statement of the the problem above is due to Kemmerer, who first published this problem as a specification exercise in [Kemmerer 85]. A variant of Kemmerer's problem statement was posed for the Fourth International Workshop on Software Specification and Design in 1986, and twelve of the published papers considered it. Wing summarizes their specifications in [Wing 88]. The workshop specification made some changesfrom the specification, such as limiting transaction 1 to staff users as well. (Presumably they would do it on the behalf of ordinary borrowers.)

1.7.2 Design Considerations

Given the focus of this problem on searching and incrementally updating information on individual books in a large, mostly static collection, the obvious architectural choice for this problem is a database-oriented system.

Building on this premise, we can consider many interesting design variations:

- How should the applications interact with the database? While some results must be produced in real-time, some transactions could be bundled for batch processing.
- How centralized should the system be? Both the database and the applications could be distributed over multiple machines.
- Does the type of the database influence the choice of the software architecture? It is conceivable that object-oriented database systems are biased towards different implementations than relational databases.

It would also be interesting to consider designs that do not localize circulation information in a database.

1.7.3 Solutions

1.7.4 Contributors

John Ockerbloom refined the problem statement and discussion.

1.8 Automated Teller Machine (ATM)

The ATM (Automated Teller Machine) problem has cropped up in several papers. Here is the problem as it was originally posed by Rumbaugh in his book on object-oriented design [Rumbaugh91], as described in [Lubars92].

> Design the software system to support a computerized banking network including both human cashiers and automatic teller machines (ATMs) to be shared by a consortium of banks. Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter account and transaction data. Automatic teller machines communicate with a central computer which clears transactions with the appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires appropriate record keeping and security provisions. The system must handle concurrent accesses to the same account correctly. The banks will provide their own software for their own computers; you are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.

The architecture in this case will have to address issues such as:

- Where should the security mechanisms be located, in the ATMs or a central network controller?
- How should the presence of heterogeneous systems (each bank has its own software) be managed?

1.8.1 History

1.8.2 Design Considerations

1.8.3 Solutions

1.8.4 Contributors

1.9 Calendar Scheduler

Calendar management is one of the beastly problems of computing. Many people have attacked it, but as yet no fully satisfactory solution has appeared.

> The calendar scheduler maintains consistent meeting schedules for a number of people. These schedules record at least the time, duration, and participants in each meeting. Some of the meetings may include people whose schedules are not maintained by the calendar scheduler. Meetings may be added or dropped at any time (up to the moment when they occur), and participants to meetings can be added or removed. A meeting may be scheduled at any time which is convenient for all (or enough) of the meeting participants, except that some of the meetings may need to occur in a particular order. The scheduler may maintain information about the scheduling preferences of the people it serves.

1.9.1 History

This is a standing problem that is often "solved" badly. Existing products are able to record simple scheduling decisions and share databases, but they fall far short of being able to handle personal preferences.

This problem has, obviously, had paper and pencil solutions as long as there have been paper and pencil, and there were undoubtedly other solutions to it before that. In the computer arena, there are individual calculator size machines to take the place of a calendar notebook (e.g. ...? there are a ton of these), as well as many scheduling programs on multi-user systems which are able to take over some of the time-selection and consistency checking task.

«Describe current products and their shortcomings»

The problem has been used to focus discussion of requirements and specification. Axel van Lamsweerde provided

the results of those discussions as an extended problem statement [vanLamsweerde92, vanLamsweerde93].

1.9.2 Design considerations

The challenge arises from two source: the multiparty, distributed, heterogeneous, asynchronous nature of calendars; and the need to accommodate personal preferences, some of which are either private or poorly articulated.

This problem may face considerable hardware and environmental constraints. For example, personal electronic notebooks do not yet communicate freely, so it is not possible to assume that all calendars of interest will be either instantly or simultaneously accessible.

Users' expectations are also a factor in considering alternatives; it is probably not acceptable to completely reschedule everyone whenever a meeting is changed: there must be some stability as meetings are added and removed. This problem may be made arbitrarily more complex by considering what it means for a time to be "convenient" for a participant or group of participants. What kind of constraints may a user place on the allowable schedules?

Some key considerations affecting the architecture are:

- *Individual flexibility:* How rich a set of individual preferences can be expressed and accommodated?
- *Heterogeneity:* How well are different personal calendar representations handled?
- *Priorities and Conditions:* How well can the system resolve conflict when degrees of intensity about preferences can be provided?
- *Ease of use:* How easy is it for a person to define and manipulate a set of meetings to attend? How will the information from multiple machines be consolidated? How well are regular meetings handled? Can quorums be defined?
- *Optimality:* If there is a schedule, will the system find it? Can the architecture support contingency strategies, e.g., in the absence of complete information?

1.9.3 Solutions

1.9.4 Contributors

Rob Allen stated the simple problem and organized the discussion. Axel van Lamsweerde provided the extended specification to be found in the solution section.

1.10 Mobile Robot

This problem focuses on embedded real-time systems. These systems must deal with external sensors and actuators, and they must respond in time commensurate with the activities of the system in its environment.

Consider the following activities a mobile robot typically has to accomplish:

- Acquiring the input provided by its sensors.
- Controlling the motion of its wheels and other moveable parts,
- Planning its future path.

A number of factors complicate the tasks:

- Obstacles may block the robot's path.
- The sensor input may be imperfect.
- The robot may run out of power.
- Mechanical limitations may restrict the accuracy with which the robot moves.
- The robot may manipulate hazardous materials.
- Unpredictable events may leave little time for responding.

1.10.1 History

Over the years, the field of mobile robots has yielded many architectural proposals. In the solutions we present in section 2.2 on page 8, we will consider four proposals ranging from the layered paradigm [Elfes87] to the blackboard structure [Shafer86].

The richness of the field permits interesting comparisons of the emphases different researchers have chosen for their robotic projects and between the trade-offs the choices entail. The next section surveys the factors to consider.

1.10.2 Design Considerations

We state the following requirements for the robot's architecture.

R1: The architecture must *accommodate deliberative and reactive behavior.* The robot has to coordinate the actions it deliberately undertakes to achieve its designated objective (e.g., collect a sample of rocks) with the reactions forced on it by the environment (e.g., avoid an obstacle).

R2: The architecture must *allow for uncertainty.* Never will all the circumstances of the robot's operation be fully predictable. The architecture must provide the framework

in which the robot can act even when faced with incomplete or unreliable information (e.g., contradictory sensor readings).

R3: The architecture must *account for the dangers* inherent in the robot's operation and its environment. By incorporating consideration of fault tolerance (R3a), safety (R3b), and performance (R3c) attributes, the architecture must help in maintaining the integrity of the robot, its operators, and its environment. Problems like reduced power supply, dangerous vapors, or unexpectedly opening doors should not spell disaster.

R4: The architecture must give the designer *flexibility.* Application development for mobile robots frequently requires experimentation and reconfiguration. Moreover, changes in tasks may require regular modification.

The degree to which these requirements apply depends both on the complexity of the work the robot is programmed to perform and the predictability of its environment. For instance, fault tolerance is paramount when the robot is operating on another planet as part of a space mission; it is still important, but less crucial, when the robot can be brought to a nearby maintenance facility.

1.10.3 Solutions

In section 2.2 on page 8, we examine four major architectures that have been implemented on robots. These include Lozano's control loops, Elfes' layered organization, Simmons' task control architecture, and Shafer's application of blackboards. The requirements listed above guide the evaluation of these alternatives.

1.10.4 Contributors

Marco Schumacher refined the problem and described the solutions.

1.11 Compiler

Compilers translate programming languages to machine language. They also interact with other programming tools such as interactive editors anddebuggers.

> A compiler translates source code in a programming language to object code that can be linked with other object code and executed on a computer.

1.11.1 History

Compilers are among the oldest well-understood non-trivial software systems. The compiler is the example of choice for the undergraduate course that introduces multi-module software organizations, yet high-performance incremental distributed compilers continue to offer design challenges.

1.11.2 Design Considerations

Simple compilers can be class exercises. However, production compilers must respond to concerns about performance and usability.

The architecture must *respond to the usage profile of its environment*. For example, student compilers must support rapid turnaround of small programs but need not be much concerned with the quality of the code. For production compilers, however, code speed may be paramount.

The architecture must be compatible with its *associated software development enviroment*. This might, for example, be batch or interactive.

1.11.3 Solutions

Many compiler design textbooks present solutions. Seshadri [Seshadri88] shows how to create a parallel version. Perry andWolf [PerryWolf92] and Garlan and Shaw [GarlanShaw93] examine some of these solutions from an architectural standpoint.

1.11.4 Contributors

Alex Wolf, Dewayne Perry, and Bill Griswold pointed out that this collection would be deficient without a compiler example.

2 Solutions

We now present sample solutions for the first two model problems, KWIC and the Mobile Robot. In each case, the presentation begins with sketches of several alternative architectures. Each sketch describes the architecture and identifies some of its strengths and weaknesses. Then a summary section compares the merits of the alternatives, emphasizing the design considerations of the problem statement.

2.1 Solutions to KWIC

This section contains four solutions to the Key Word in Context (KWIC) architectural model problem. All four are grounded in published solutions. The first two are those considered in Parnas' original article [Parnas72]. The third solution is based on the use of "reactive integration" and represents a variant on the solution examined by Garlan, Kaiser, and Notkin [Garlan92]. The fourth is a pipeline solution inspired by the Unix index utility [ref??].

2.1.1 Solution 1: Main program/subroutine with shared data.

The first solution decomposes the problem according to the four basic functions performed: input, shift, alphabetize, and output. These computational components are coordinated as subroutines by a main program that sequences through them in turn. Data is communicated between the components through shared storage ("core storage"). Communication between the computational components and the shared data is an unconstrained read-write protocol. This is made possible by the fact that the coordinating program guarantees sequential access to the data.

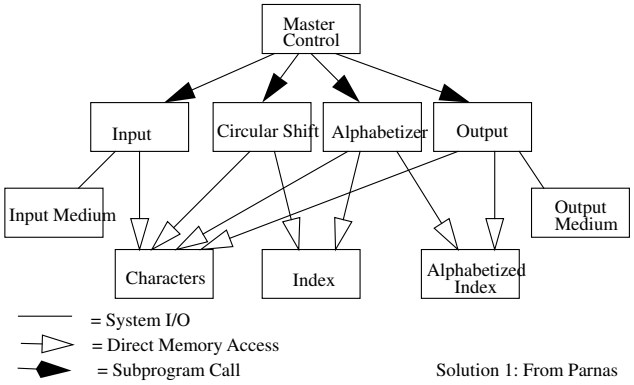

Figure 2.1.2: Hierarchical Subroutine Architecture with Shared Data

In this solution, computations can share the same storage. This allow efficient data representation. The solution also has a certain intuitive appeal, since distinct computational aspects are isolated in different modules.

However, as Parnas argues, it has a number of serious drawbacks in terms of its ability to handle changes. In particular, a change in data storage format will affect almost all of the modules. Similarly changes in algorithm and enhancements to system function are not easily handled.

Finally, reuse is now well-supported because each module of the system is tied tightly to this particular application.

2.1.3 Solution 2: Abstract data types.

The second solution decomposes the system into a similar set of five modules. However, in this case data is no longer directly shared by the computational components. Instead, each module provides an interface that permits other components to access data only by invoking procedures in that interface.

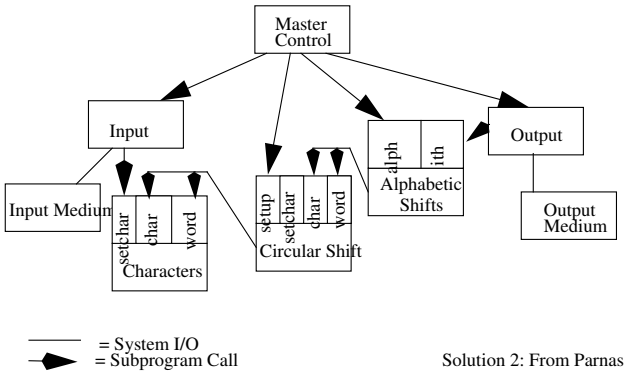

Figure 2.1.4: Abstract Data Type Architecture

This solution is composed of the same processing modules as the first. However, it has a number of advantages over the first solution when design changes are considered. In particular, both algorithms and data representations can be changed in individual modules without affecting others. Moreover, reuse is better supported than in the first solution because modules make fewer assumptions about the others with which they interact.

On the other hand, as discussed by Garlan, Kaiser, and Notkin, the solution is not particularly well suited to enhancements. The main problem is that to add new functions to the system, the implementor must either modify the existing modules -- compromising their simplicity and integrity -- or add new modules that lead to performance penalties. (See [Garlan92] for a detailed discussion.)

2.1.5 Solution 3: Reactive integration.

The third solution uses a form of component integration based on shared data similar to the first solution. However, there are two important differences. First, the interface to the data is more abstract. Rather than exposing the storage

formats to the computing modules, data is accessed abstractly (for example, as a list or set). Second, computations are invoked implicitly as data is modified. For example, the act of adding a new line to the line storage causes an event to be sent to the shift module. This allows it to produce circular shifts (in a separate abstract shared data store). This in turn causes the alphabetizer to be implicitly invoked so that it can alphabetize the lines. Additional discussion of this integration paradigm can be found elsewhere [GarlanNotkin91].

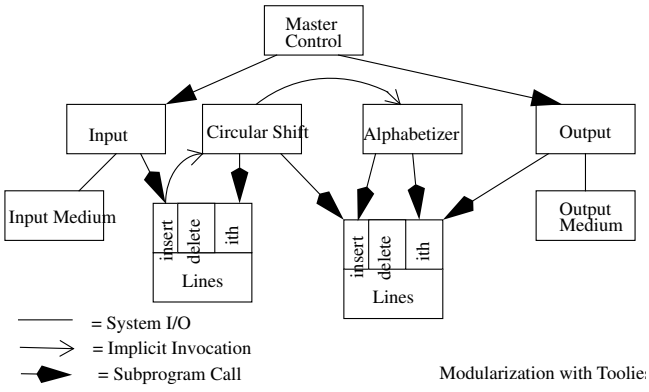

Figure 2.1.6: Reactive Architecture

This solution easily supports functional enhancements to the system: additional modules can be attached to the system by registering them to be invoked on certain events. Because data is accessed abstractly, it also insulates computations from changes in data representation. Reuse is also supported, since the implicitly invoked modules only rely on the existence of certain externally triggered events.

However, the solution suffers from the fact that it can be difficult to change the order of processing of the implicitly invoked modules. Further, because invocations are data driven, the most natural solutions using this kind of decomposition tend to use more space than the previously considered decompositions.

2.1.7 Solution 4: Dataflow.

The fourth solution uses a pipeline. A pipeline is composed of a sequence of filters, connected by streams of data. In this case there are four filters: input, shift, alphabetize, and output. Each filter processes its data, sending it to the downstream filter. Control is distributed: each filter can run whenever it has data on which to compute. Data sharing between filters is strictly limited to that transmitted on pipes [AllenGarlan92].

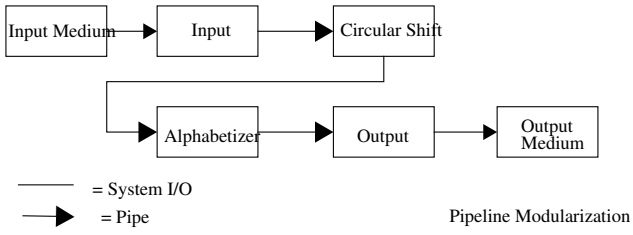

Figure 2.1.8: Dataflow Architecture

This solution has several nice properties. First, it supports the intuitive flow of processing. Second, it supports reuse, since each filter can function in isolation (provided upstream filters produce data in the form it expects). New functions are easily added to the system by inserting filters at the appropriate point in the processing sequence.

On the other hand, it has a number of drawbacks. First, it is virtually impossible to modify the design to support an interactive system. For example, in order to delete a line, there would have to be some persistent shared storage, violating a basic tenet of this approach. Second, the solution is inefficient in terms of its use of space, since each filter must copy all of the data to its output ports.

2.1.9 Summary

To a rough approximation, the solutions can be compared by tabulating their ability to address the design considerations itemized in the following table:

	Shared Data	Abstract Datatype	Reactive Integration	Dataflow
Change in Algorithm	-	-	+	+
Change in Data Representation	-	+	-	-
Change in Function	+	-	+	+
Performance	+	+	-	-
Reuse	-	+	-	+

Table 2.1.1. Strength and Weaknesses of KWIC Architectures

2.2 Solutions for Mobile Robot

For sample solutions, we examine four major architectures that have been implemented on robots. These include Lozano's control loops [Lozano90], Elfes' layered organization [Elfes87], Simmons' task control architecture [Simmons92], and Shafer's application of blackboards [Shafer86].

2.2.1 Solution 1: Control Loop

Figure 2.2.1 models the control loop paradigm.

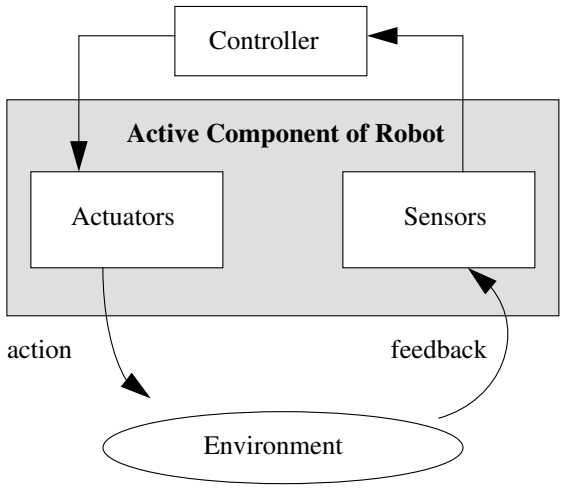

Figure 2.2.2: A Control Loop Architecture

Most industrial robots support minimal handling of unpredictable events: the tasks are fully predefined (e.g, welding certain automobile parts together), and the robot has no responsibility with respect to its environment (it is rather the environment that is responsible for not interfering with the robot). The *open loop* paradigm applies naturally to this situation: the robot initiates an action or series of actions without bothering to check on their consequences [Lozano90].

Upgrading this paradigm to mobile robots involves adding feedback, thus producing a *closed loop* architecture. The controller initiates robot actions and monitors their consequences, adjusting the future plans based on this return information.

(R1) An advantage of the closed loop paradigm is its simplicity: it captures the basic interaction between the robot and the outside.

Its simplicity is also a drawback in the more unpredictable environments. One expert [Lozano90] comments on the fact that the feedback loop assumes that changes in the environment are linear and require linear reactions (e.g., like the control of pressure through the gradual opening and closing of a valve); robots, though, are mostly confronted with disparate, discrete events that demand switches between very different behavior modes (e.g., between controlling manipulator motions and adjusting the base position, to avert loss of equilibrium). The model

does not provide any hints as to how different kinds of events may be managed.

For complex tasks, the control loop gives no leverage for decomposing the software into cooperating components. If the steps of sensing, planning, and acting have to be refined, other paradigms have to provide the nuances the control loop model lacks.

(R2) For the resolution of uncertainty, the control loop paradigm is biased towards one method: reducing the unknowns through iteration; a trial-and-error process with action and reaction eliminates possibilities at each turn. If more subtle steps are needed, the architecture offers no framework for integrating these with the basic loop or for delegating them to separate entities.

(R3) Fault tolerance and safety are supported by the closed loop paradigm in the sense that its simplicity makes duplication easy and reduces the chance of errors creeping into the system.

(R4) The major components of a robot architecture (supervisor, sensors, motors) are separated from each other and can be replaced independently. More refined tuning has to take place inside the modules, at a level of detail the architecture does not show.

In summary, the closed loop paradigm seems most appropriate for simple robotic systems which have to handle only a small number of external events and whose tasks involve no complicated decomposition.

2.2.3 Solution 2: Layered Architecture

Figure 2.2-2 shows Alberto Elfes' definition of the idealized layered architecture [Elfes87] that influenced the design of the Dolphin sonar and navigation system, implemented on the Terregator and Neptune mobile robots [Champeny93, Podnar84].

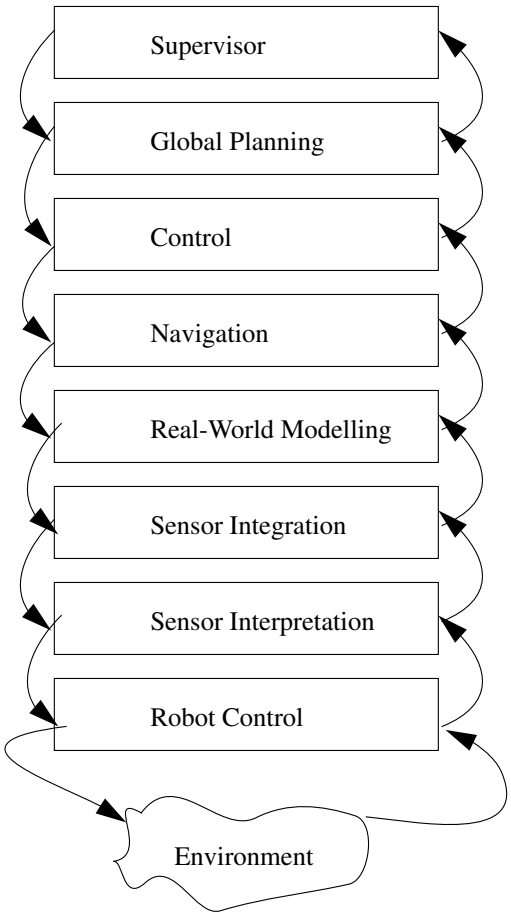

Figure 2.2.4: A Layered Architecture

At level 1, the lowest level, reside the robot control routines (motors, joints,...).

Levels 2 and 3 deal with the input from the real world. They perform sensor interpretation (the analysis of the data from one sensor) and sensor integration (the combined analysis of different sensor inputs).

Level 4 is concerned with maintaining the robot's model of the world.

Level 5 manages the navigation of the robot.

The next two levels, 6 and 7, schedule and plan the robot's actions. Dealing with problems and replanning is also part of the level-7 responsibilities.

The top level provides the user interface and overall supervisory functions.

(R1) Elfes' model sidesteps some of the problems encountered with the control loop by defining more components to which the required tasks can be delegated. Being specialized to autonomous robots, it points to the concerns that have to be addressed (e.g., sensor integration). Furthermore, it defines abstraction levels (e.g, robot control vs. navigation) to guide the design.

While it organizes well the components needed to coordinate the robot's operation, the layered architecture does not fit the actual data and control flow patterns. The layers suggest that services and requests are passed between adjacent components. In reality, as Elfes readily admits, the information exchange is less straightforward. For instance, data necessitating fast reaction may have to be sent directly from the sensors to the problem handling agent at level 7, and the corresponding commands may have to skip levels to reach the motors in time.

Another imprecision in the model is that it does not separate the two abstraction hierarchies that actually exist in the architecture:

- The data hierarchy with raw sensor input (level 1), interpreted and integrated results (2 and 3), and finally the world model (4).
- The control hierarchy with motor control (level 1), navigation (5), scheduling (6), planning (7), and user-level control (8).

The NASREM architecture mentioned in the conclusion is more precise in this respect.

(R2) The existence of abstraction layers addresses the need for managing uncertainty: what is uncertain at the lowest level may become clear with the added knowledge available in the higher layers. For instance, the context embodied in the world model can provide the clues to disambiguate conflicting sensor data.

(R3) Fault tolerance and passive safety (when you strive *not* do something) are served by the abstraction mechanism too. Data and commands are analyzed from different perspectives. It is possible to incorporate many checks and balances into the system.

As already mentioned, performance and active safety (when you have to do something rather than avoid doing something) may require that the communication pattern be short-circuited.

(R4) The fudged dependencies are an obstacle to easy replacement and addition of components. The fragile relationships between the layers can become more difficult to decipher with each change.

In summary, the abstraction levels defined by the layered architecture provide what constitutes the goal for software architectures in general: a framework for organizing the components. It achieves this objective by being precise about the role of the different layers.

The major drawback of the model is that it breaks down when it is taken to the greater level of detail demanded by an actual implementation. The communications patterns in a robot do most probably not follow the very orderly scheme implied by the architecture.

2.2.5 Solution 3: Implicit Invocation

Figure 2.2.3 summarizes the Task Control Architecture (TCA) [Simmons92] which uses implicit invocation. It was applied, among others, to the Ambler robot [Simmons90].

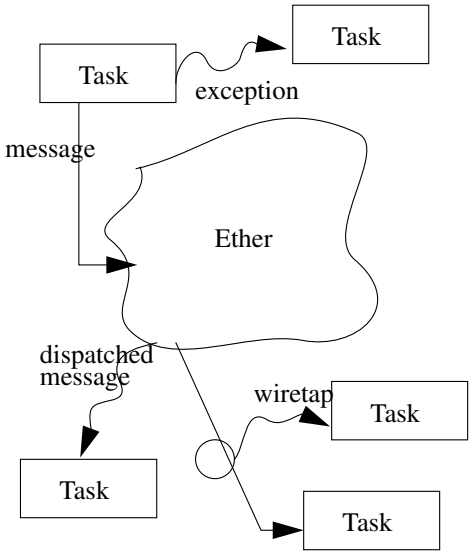

Figure 2.2.6: An Implicit Invocation Architecture

TCA is not only an architecture; it also provides a sophisticated tool box for building robots: a library of communication and control routines that implement the TCA philosophy. The following discussion focuses on task trees and the implicit invocation features. For a complete overview, see the references.

The TCA architecture is based on hierarchies of tasks, the task trees. Figure 2.2.4 shows a sample task tree. Parent tasks initiate child tasks. The software designer can define temporal dependencies between pairs of tasks. An example temporal constraint is: "A must complete before B starts." These features permit the specification of selective concurrency.

TCA's routines include many operations on task trees for dynamically reconfigure them at run-time.

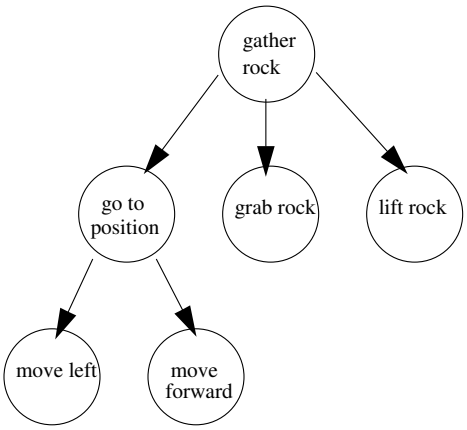

Figure 2.2.7: A Task Tree

In TCA, tasks communicate by sending messages to a central server, which redirects the messages to tasks that have registered to handle them. This scheme, where the sender does not need to know the receiver, is the basic characteristic of implicit invocation.

Three more implicit invocation mechanisms are part of TCA's features:

- *Exceptions:* Certain conditions cause the execution of an associated exception handler. Exceptions override the currently executing task in the subtler that causes the exception. They quickly change the processing mode of the robot and are thus better suited for managing spontaneous events (such as a dangerous change in terrain) than the feedback loop or the long communication paths of the pure layered architecture.

  Exception handlers have at their disposal all the operations for manipulating the task trees: e.g., they can abort or retry tasks.
- *Wiretapping:* Messages can be intercepted by routines superimposed on an existing architecture, i.e., task tree. For instance, a safety check procedure can use this feature to validate all outgoing motion commands.
- *Monitors:* Monitors read information and execute some action if the data fulfill a certain criterion. An example from the TCA manual is the battery check: if the battery level falls below a given level, the actions necessary for recharging it are invoked. This feature offers a convenient way of dealing with fault tolerance issues by setting aside agents to supervise the system.

(R1) Task trees on one hand, and exceptions, wiretapping, and monitors on the other permit a clear-cut separation of action (the nominal behavior embodied in the task trees) and reaction (the behavior dictated by extraneous events and circumstances).

TCA also distinguishes itself from the previous paradigms by incorporating concurrent agents in its model. In TCA it is evident that multiple actions can proceed at the same time, more or less independently. The other two models do not show the presence of concurrency.

The amount of concurrency is limited by the capabilities of the central server. In general, its reliance on a central control point may be a weak point of TCA.

(R2) How TCA addresses uncertainty is less clear. If imponderables exist, a tentative task tree can be built, to be adapted by the exception handlers when the assumptions it is based on turn out to be erroneous.

(R3) As illustrated by the examples above, the TCA exception, wiretapping, and monitoring features take into account the needs for performance, safety and fault tolerance.

Fault tolerance by redundancy is achieved when multiple handlers register for the same signal; if one of them becomes unavailable, TCA can still provide the service by routing the request to another. Performance also benefits since multiple occurrences of the same request can be handled concurrently by multiple handlers.

(R4) The use of implicit invocation makes incremental development and replacement of components straightforward: it is often sufficient to register new handlers, exceptions, wiretaps or monitors with the central server; no existing component feels the impact.

In summary, TCA offers a comprehensive set of features for coordinating the tasks of a robot while respecting the

quality and ease of development requirements. The richness of the scheme makes it most appropriate for more complex robot projects.

2.2.8 Solution 4: Blackboard Architecture

Figure 2.2.5 describes a blackboard architecture for mobile robots. This paradigm was used in the NAVLAB project, as part of the CODGER system [Shafer86].

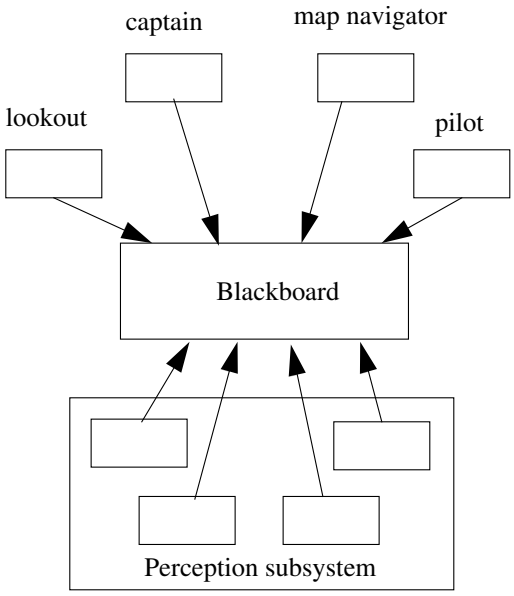

Figure 2.2.9: A Blackboard Architecture

The "whiteboard" architecture, as it is named in [Shafer86], works with abstractions reminiscent of those encountered in the layered architecture. The components of CODGER are:

- The "captain", the overall supervisor.
- The "map navigator", the high level path planner.
- The "lookout", a module that monitors the environment for landmarks.
- The "pilot", the low level path planner and motor controller.
- The perception subsystem, the modules that accept the raw input from multiple sensors and integrate it into a coherent interpretation.

(R1) The components (including the modules inside the perception subsystem) communicate via the characteristic central database of the blackboard systems. Modules indicate their interest in certain types of information. The database returns them such data either immediately or when some other module inserts them into the database.

For instance, the lookout may watch for certain geographic features; the database informs it when the perception subsystem stores images matching the description.

One difficulty with the CODGER architecture is that all control flow has to be coerced to fit the database mechanism, even under circumstances where direct interaction between components would be more natural.

(R2) The blackboard is also the means for resolving conflicts or uncertainties in the robot's world view. For instance, the lookout's landmark detections provide a reality check for the distance estimation by dead-reckoning, both stored in the database. The modules responsible for the uncertainty resolution register with the database to obtain the necessary data.

The main example of this activity is *sensor fusion*, performed by the perception subsystem to reconcile the input from its diverse sensors.

(R3) The communication via the database is similar to the communication via TCA's central message server. The exception mechanism, wiretapping and monitoring - guarantors of reaction speed, safety, and reliability - can be implemented in CODGER by defining separate modules that watch the database for the tell-tale signs of unexpected occurrences or the beginnings of troublesome situations. TCA's safety mechanism of double-checking messages through wiretaps cannot be fully duplicated because it may be too late to prevent an action once it manifests itself in the database. (TCA holds the message while the wiretap processes it.)

(R4) As with TCA, the blackboard architecture offers support for concurrency and decouples senders from receivers, thus gaining flexibility for maintenance.

In summary, the blackboard architecture is capable of modeling the cooperation of tasks, both for coordination and uncertainty resolution in a very flexible manner, thanks to an implicit invocation mechanism based on the contents of the database. These features are only slightly less powerful than TCA's equivalent capabilities.

2.2.10 Conclusion

We have seen four architectures, of which two (layered architecture and blackboard) are very specific and give

precise indications as to the components expected in a robot. The other two (control loop and implicit invocation) define no functional components and concentrate on the mechanisms.

Specificity is helpful for getting a grasp on the basic abstractions and tasks involved in an autonomous robot. It would be interesting to research the value of a TCA architecture (which is the most powerful in its mechanisms) combined with a functional decomposition of robot tasks (planning, sensor integration, ...).

Other hybrid architectures have been proposed. The NASA/NBS Standard Reference Model for Telerobots (NASREM) [Lumia90] can be seen as a combination of the control loop and the layered architectures (Figure 2.2.6).

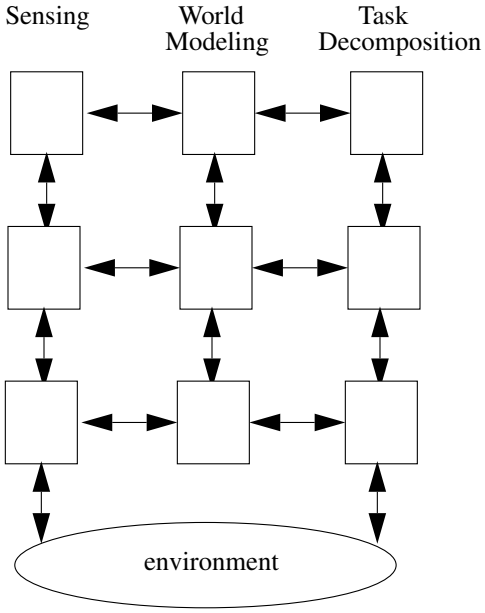

Figure 2.2.11: The NASREM Architecture

The layers from top to bottom are defined by the time frame in which they perform their tasks. Seen from this perspective, the architecture is a hierarchy of control loops with increasingly tighter response time constraints.

The layers from left to right represent the functional abstractions.

To conclude, table 2.2.1 summarizes the strengths and weaknesses of the reviewed software architectures.

	Control Loop	Layers	Impl. Invoc.	Black Board
Task Coordination	+-	-	++	+
Dealing with Uncertainty	-	+-	+-	+
Fault Tolerance	+-	+-	++	+
Safety	+-	+-	++	+
Performance	+-	+-	++	+
Flexibility	+-	-	+	+

Table 2.2.1. Strengths and Weaknesses of Robot Architectures

3 Fuller Definitions of Problems

3.1 Extended Definition of Meeting Scheduler

Axel van Lamsweerde provides an extended specification of the meeting scheduling problem from October 1992 [vanLamsweerde92] and an extension to cover conflict resolution from November 1993 [vanLamsweerde93].

We include it in this document because it shows how problem complexity emerges as you consider the problem statement in more detail, and because the original source is relatively inaccessible to this commumnity. Because of its length it is set up as a separate section rather than being included in the short introductions. Think of this section as a solution imported from a requirements/specification exercise.

For more information about the preliminary definition, contact

> Axel van Lamsweerde, Robert Darimont and Philippe Massonet

UCL - Unite d'Informatique
B-1348 Louvain-la-Neuve (Belgium)
avl@info.ucl.ac.be

For more information about the extension for conflict resolution, contace

Axel van Lamsweerde, Charles Christoph and Philippe Massonet
University of Louvain,
Unite d'informatique,
B-1348 Louvain-la-Neuve (Belgium)

3.1.1 The Meeting Scheduler System: Preliminary Definition

3.1.1.1 Foreword

This preliminary description is deliberately intended to be sketchy and unprecise. Acquisition, formalization and validation processes are needed to complete it and lift the many shadow areas.

A number of features of the Meeting Scheduler System were inspired from various experiences in organizing meetings (faculty meetings, ESPRIT project meetings, Program Committee meetings, etc.) and from various discussions with Steve Fickas' group at the University of Oregon.

3.1.1.2 Scheduling Meetings: Domain Theory

Meetings are typically arranged in the following way. A *meeting initiator* asks all potential meeting attendees for the following information based on their personal agenda:

- a set of dates on which they cannot attend the meeting (hereafter referred as *exclusion set*);
- a set of dates on which they would prefer the meeting to take place (hereafter referred as *preference set*).

A *meeting date* is defined by a pair (calendar date, time period). The exclusion and preference sets are contained in some time interval prescribed by the meeting initiator (hereafter referred as *date range*).

The initiator also asks *active participants* to provide any special equipment requirements on the meeting location (e.g., overhead-projector, workstation, network connection, telephones, etc.); he/she may also ask *important* participants to state preferences about the meeting location.

The proposed meeting date should belong to the stated date range and to none of the exclusion sets; furthermore it should ideally belong to as many preference sets as possible. A *date conflict* occurs when no such date can be found. A conflict is strong when no date can be found within the date range and outside all exclusion sets; it is weak when dates can be found within the date range and outside all exclusion sets, but no date can be found at the intersection of all preference sets. Conflicts can be resolved in several ways:

- the initiator extends the date range;
- some participants remove some dates from their exclusion set;
- some participants withdraw from the meeting;
- some participants add some new dates to their preference set.

A meeting room must be available at the selected meeting date. It should meet the equipment requirements; furthermore it should ideally belong to one of the locations preferred by as many important participants as possible. A new round of negotiation may be required when no such room can be found.

The meeting initiator can be one of the participants or some representative (e.g., a secretary).

3.1.1.3 System Requirements

The purpose of the *meeting scheduler system* is to support the organization of meetings - that is, to determine, for each meeting request, a meeting *date* and *location* so that most of the intended participants will effectively participate. The meeting date and location should thus be as convenient as possible to all participants. Information about the meeting should also be made available as early as possible to all potential participants. The intended system should considerably reduce the amount of overhead usually incurred in organizing meetings where potential attendees are distributed over many different places. On another hand, the system should reflect as closely as possible the way meetings are typically managed (see the domain theory above).

The system should assist users in the following activities.

- Plan meetings under the constraints expressed by participants
- Replan a meeting dynamically to support as much flexibility as possible. On one hand, participants should be

allowed to modify their exlusion set, preference set and/or preferred location *before* a meeting date/location is proposed. On the other hand, it should be possible to take some external constraints into account *after* a date and location have been proposed - e.g., due to the need to accommodate a more important meeting. The original meeting date or location may then need to be changed; sometimes the meeting may even be cancelled. In all cases some bound on replanning should be set up.

- Support conflict resolution according to resolution policies stated by the client.
- Manage all the interactions among participants required during the organization of the meeting - to communicate requests, to get replies even from participants not reacting promptly, to support the negotiation and conflict resolution processes, to make participants aware of what's going on during the planning process, to keep participants informed about schedules and their changes, to make them confident about the reliability of the communications, etc.
- Keep the amount of interaction among participants (e.g., number and length of messages, amount of negotiation required) as small as possible.

The meeting scheduler system must in general handle several meeting requests *in parallel*. Meeting requests can be competing by overlapping in time or space. Concurrency must thus be managed.

The following aspects should also be taken into account.

- The system should accomodate decentralized requests; any authorized user should be able to request a meeting independently of his whereabouts.
- Physical constraints may not be broken - e.g., a person may not be at two different places at the same time, a meeting room may not be alloc
- The system should provide an appropriate level of performance, for example:
  - the elapsed time between the submission of a meeting request and the determination of the corresponding meeting date/location should be as small as possible;
  - the elapsed time between the determination of a meeting date/location and the communication of this information to all participants concerned should be as small as possible;
  - a lower bound should be fixed between the time at which the meeting date is determined and the time at which the meeting is actually taking place.
- Privacy rules should be enforced; a non-privileged participant should not be aware of constraints stated by other participants.
- The system should be usable by non-experts.
- The system should be customizable to professional as well as private meetings. These two modes of use are characterized by different restrictions on the time periods that may be allocated (e.g., meetings during office hours, private activities during leisure time).
- The system should be flexible enough to accommodate evolving data - e.g., the sets of concerned participants may be varying, the address at which a participant can be reached may be varying, etc.
- The system should be easily extendable to accommodate the following typical variations:
  - handling of explicit status and priorities among participants;
  - handling of explicit priorities among dates in preference sets;
  - handling of explicit dependencies between meeting date and meeting location;
  - participation through delegation - a participant may ask another person to represent him/her at the meeting;
  - partial attendance - a participant can only attend part of the meeting;
  - variations in date formats, address formats, interface language, etc.
  - partial reuse in other contexts -e.g., to help establish course schedules.

This ends the problem description. The following extends the system

3.1.2 Extending the Meeting Scheduler System to Support Conflict Resolution

3.1.2.1 Foreword

This note aims at suggesting a useful extension to the Meeting Scheduler System. The objective is to incorporate knowledge about participant status and about various kinds of priorities among participants and meetings.

3.1.2.2 Finding Best Meetings and Resolving Conflicts

Context

The purpose of the Meeting Scheduler System is to support the organization of meetings--that is, to determine, for each meeting request, a meeting date, location and equipment so that the expected participants can attend, the meeting date and location are most convenient to important participants, etc. The Meeting Scheduler System should also minimize the overhead usually incurred in organizing meetings.

When there is no common date within all preference sets or no common date outside all exclusion sets, the Meeting Scheduler System will not be able to find a date which is perfectly suitable to everybody. It is then necessary to negotiate a solution to resolve conflicts. This may be done in several ways (see preliminary description above).

Clients and analysts came to the conclusion that knowledge about participant status and about priorities among users and meetings should help in resolving conflicts by determining a "best" way to resolve a conflict. Even when there is no conflict, the participant status may be useful in determining a "best" meeting date and location.

Status and priorities

The following notions should be incorporated in the proposed extension. They capture the hierarchical importance of participants, the importance for a participant to attend a particular meeting relatively to other participants or to other meetings, and the ease with which a participant can make a particular date interval free. These various notions will be used in the conflict resolution process.

Participant Status

The participant *status* captures the hierarchical importance of a participant with respect to others independently of any specific meetings he is expected to participate in.

The participant *status* might be used, e.g., to determine a "best" compromise on date and location whenever several ones are possible.

The participant *status* is typically determined by some super user.

For instance, in the context of scheduling Faculty meetings the Departement Head would have a higher *status* than normal professors. The latter would have a higher *status* than student representatives.

Participant Importance

The participant *importance* captures the importance for a specific person to attend a particular meeting *relatively to other participants*.

Participant *importances* are typicaly determined by the meeting initiator.

For instance, the meeting chairman and secretary must be present; they have the highest participant *importance*. In a project meeting where specific tasks are discussed, the task leaders would have a higher participant *importance* than normal project members and a lower importance than the meeting chair, the task speakers or the project reviewers.

Meeting Significance

The meeting *significance* represents the importance for a specific person to attend a particular meeting *relatively to other meetings or meeting requests*.

Meeting *significances* are typically determined by the participants concerned.

For instance, participants to a specific task in a research project would assign a greater significance to a project meeting where their task will be discussed.

This information must be kept confidential.

Participant Flexibility

The participant *flexibility* is intended to indicate how easily a user can make a particular date interval free to allow meetings to be scheduled within that interval. Dates in exclusion sets and/or preference sets can thus be weighted accordingly.

The participant *flexibility* is typically determined by the participants concerned.

For instance, professors cannot move lecture periods easily; their participant *flexibility* for the corresponding date intervals should be low. A date interval which is not in the exclusion set of a participant should have a high {\it flexibility} for that participant.

This information must be kept confidential.

Using Knowledge about Status and Priorities

The following tactics illustrate some typical uses of the various kinds of priorities suggested above.

- Best meeting dates and locations should be determined by considering participants with higher participant *status* first
- If no date can be found to organize a meeting, the Meeting Scheduler System could propose a person having low participant {\it importance} to withdraw from the meeting.
- If no date can be found to organize a meeting, the Meeting Scheduler System could propose a participant to cancel (or to withdraw from) another meeting having a lower meeting {\it significance}.
- A meeting date within some exclusion set (or outside some preference set) coud be considered if the corresponding participant has a high {\it flexibility} for it.

4 Administrative Matters

4.1 Acknowledgments

This work has been funded variously by the Department of Defense Advanced Research Project Agency under grants MDA972-92-J-1002 and F33615-93-1-1330, by the National Science Foundation Grants CCR-9109469 and CCR-9112880, by a grant from Siemens Corporate Reseaerch, and by the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is a Federally Funded Resarch and Development Center sponsored by the US Department of Defense and operated by Carnegie Mellon University under Contract F19628-90-C-0003). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the above organizations (or any others -- how can an organization have an opinion?). The US government has a royalty-free government purpose license to use, duplicate, or disclose the work. Not only that, we encourage the rest of you to do the same -- just give us credit.

We thank Ralph Johnson of the University of Illinois at Urbana-Champaign and Bill Griswold of the University of California at San Diego for their comments.

4.2 Bibliography

[AllenGarlan 92] Robert Allen and David Garlan. A Formal Approach to Software Architectures. *Proceedings of the 1992 IFIP Congress*, September 1992.

[AtleeGannon93] Joanne M. Atlee and John Gannon. State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, vol 19, no 1, Jan 1993, pp.24-40.

[BirchenoughCameron89]JSD and Object-Oriented Design. In John R. Cameron (ed), *JSP and JSD: The Jackson Approach to Software Development*, IEEE Press 1989, pp.293-304.

[Boehm84] D. Boehm-Davis and L. Ross. *Approaches to Structuring the Software Development Process.* General Electric Company Report GEC/DIS/TR-84-B1V-1, October 1984, p.14.

[Booch86] Grady Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, vol. 12, no. 2, February 1986, pp. 211-221.

[Bracket87]

[Champeny93] Lee Champeny-Bares, Syd Coppersmith, and Kevin Dowling. *The Terregator Mobile Robot*. Technical Report CMU-RI-TR-93-03, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[Elfes87] Alberto Elfes. Sonar-Based Real-World Mapping and Navigation. *IEEE Journal of Robotics and Automation*, no.3, 1987, pp. 249-265.

[Garlan92] David Garlan, Gail E. Kaiser, and David Notkin. Using Tools to Compose Systems. *IEEE Computer*, vol.25, no.6, June 1992.

[GarlanNotkin91] David Garlan and David Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. *VDM '91:*

Formal Software Development Methods, 1991, pp. 31-44.

[GarlanShaw93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tiortora (ed), *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing, 1993, pp.1-39.

[GarlanShaw94] David Garlan and Mary Shaw. Software Development Assignments for a Software ARchitecture Course. *Proc ICSE-16 Workshop on Software Engineering Education*, to appear 1994.

[Gomaa89] Hassan Gomaa. Structuring Criteria for Real Time System Design. *Proc 11th International Conference on Software Engineering*, 1989, pp.290-301.

[Higgins87] David A. Higgins. Specifying Real-Time/Embedded Systems using Feedback/Control Models. *Proc SMC XII: Twelfth Structured Methods Conference*, 1987, pp.127-147.

[HøydalsvikSindre93] Geir Magne Høydalsvik and Guttorm Sindre. On the Purpose of Object-Oriented Analysis. *Proc OOPSLA'93*, 1993, pp. 240-255.

[Jones90] Do-While Jones. Software Testing. *Ada-Info column, Journal of Pascal, Ada, and Modula-2*, vol 9, no 2, March-April 1990, pp.53-64.

[Kemmerer85] *Oops, citation missing*.

[Lozano90] Tomás Lozano-Pérez. Preface to *Autonomous Robot Vehicles*. L. J. Cox and G.T. Wilfong, eds. Springer Verlag, New York, NY, 1990.

[Lubars92] M. Lubars, G. Meredith, C. Potts, and C. Richter. Object-Oriented Analysis for Evolving Systems. *Proceedings of ICSE*, May 1992.

[Lumia90] R. Lumia, J. Fiala, and A. Wavering. The NASREM Robot Control System and Testbed. *International Journal of Robotics and Automation*, no.5, 1990, pp. 20-26.

[Parnas72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, vol.15, no.12, December 1972, pp. 1053-1058.

[PerryWolf92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM Sigsoft Software Engineering Notes*, vol 7, no 4, October 1992, pp.40-52.

[Podnar84] Gregg Podnar, Kevin Dowling, and Mike Blackwell. *A Functional Vehicle for Autonomous Mobile Robot Research*. Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1984.

[Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Rumbaugh92] James Rumbaugh. Designing Bugs and Dueling Methodologies. *Journal of Object-Oriented Programming*, Jan 1992.

[Seshadri88] V. Seshadri et al. Semantic analysis in a concurrent compiler. *Proc. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 1988.

[Shafer86] Steven A. Shafer, Anthony Stentz, and Charles E. Thorpe. An Architecture for Sensor Fusion in a Mobile Robot. *Proceedings of the IEEE International Conference on Robotics and Automation*, San Francisco, CA, April 7-10, 1986, pp. 2002-2011.

[Shaw93] Mary Shaw. *Software Architecture for Shared Information Systems.* Technical Report CMU/SEI-93-TR-3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1993.

[Shaw94] Mary Shaw. *Making Choices: A Comparison of Styles for Software Architectures*. Unpublished manuscript.

[Shaw95] Mary Shaw. Beyond Objects: A Software Design Paradign Based on Process Control. *ACM SIgsoft Software Engineering Notes*, to appear January 1995.

[Simmons90] Reid Simmons. *Concurrent Planning and Execution for a Walking Robot*. Technical Report CMU-RI-90-16. Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[Simmons92] Reid Simmons. Concurrent Planning and Execution for Autonomous Robots. *IEEE Control Systems*, no. 1, 1992, pp.46-50.

[SmithGerhart88] Sharon L. Smith and Susan L. Gerhart. STATEMATE and Cruise Control: A Case^2 Study. *Proc COMPSAC88: Twelfth Annuan International Computer Software and Applications Conference*, 1988, pp.49-56.

[vanLamsweerde92] Axel van Lamsweerde, R. Darimont and Philippe Massonet. *The Meeting Scheduler System: Preliminary Definition*. University of Louvain, Unit\'e d'informatique, B-1348 Louvain-la-Neuve (Belgium), October 1992.

[vanLamsweerde92] Axel van Lamsweerde, Charles Christoph, and Philippe Massonet. *Extending the Meeting Scheduler System to Support Conflict Resolution*. University of Louvain, Unit\'e d'informatique, B-1348 Louvain-la-Neuve (Belgium), November 1993.

[WangTanik89] Jianbai Wang and Murat M. Tanik. Describing Real Time Systems Using PPA and XYZ/E. *Proc. 22nd Annual Hawaii International Conference on System Sciences, Vol II: Software Track*, Jan 1989.

[WardKeskar87] Paul. T. Ward and Dinesh A. Keskar. A Comparison of the Ward/Mellor and Boeing/Hatley Real-Time Methods. *Proc SMC XII: Twelfth Structured Methods Conference*, 1987, pp.356-366.

[Wasserman89] Pircher, Robert J. Muller. An Object-Oriented Structured Design Method for Code Generation. *ACM Sigsoft Software Engineering Notes* vol 14, no 1, Jan 1989, pp.32-55.

[Ward84] P. Ward. Class exercise used at the Rocky Mountain Institute for Software Engineering, Aspen CO, 1984.

[Wing88] Jeannette Wing. A Study of 12 Specifications of the Library Problem. *IEEE Software*, July 1988, pp. 66-76.

[YinTanik91] W.P. Yin and M.M. Tanik. Reusability in the real-time use of Ada. *International Journal of Computer Applications in Technology*, vol 4, no 2, 1991, pp.71-78.