3.2 INSTRUMENTATION SOFTWARE

Our second case study describes the industrial development of a software architecture at Tektronix, Inc. This work was carried out as a collaborative effort between several Tektronix product divisions and the Computer Research Laboratory over a three-year period [DG90, GD90].

The purpose of the project was to develop a reusable system architecture for oscilloscopes. An oscilloscope is an instrumentation system that samples electrical signals and displays pictures (called *traces*) of them on a screen. Oscilloscopes also perform measurements on the signals, and display them on the screen. Although they were once simple analogue devices involving little software, modern oscilloscopes rely primarily on digital technology and have quite complex software. It is not uncommon for a modern oscilloscope to perform dozens of measurements, supply megabytes of internal storage, support an interface to a network of workstations and other instruments, and provide a sophisticated user interface, including a touch-panel screen with menus, built-in help facilities, and color displays.

Like many companies that have had to rely increasingly on software to support their products, Tektronix was faced with a number of problems. First, there was little reuse across different oscilloscope products. Instead, different oscilloscopes were built by different product divisions, each with its own development conventions, software organization, programming language, and development tools. Moreover, even within a single product division, each new oscilloscope typically had to be completely redesigned to accommodate changes in hardware capability and new requirements on the user interface. This problem was compounded by the fact that both hardware and interface requirements were changing increasingly rapidly. Furthermore, there was a perceived need to address "specialized markets," which meant that it would have to be possible to tailor a general-purpose instrument to a specific set of uses, such as patient monitoring or automotive diagnostics.

Second, performance problems were increasing because the software was not rapidly configurable within the instrument. These problems arose because an oscilloscope may be configured in many different modes, depending on the user's task. In old oscilloscopes reconfiguration was handled simply by loading different software to handle the new mode. But the total size of software was increasing, which led to delays between a user's request for a new mode and a reconfigured instrument.

The goal of the project was to develop an architectural framework for oscilloscopes that would address these problems. The result of that work was a domain-specific software architecture that formed the basis of the next generation of Tektronix oscilloscopes. Since then the framework has been extended and adapted to accommodate a broader class of systems, while at the same time being better adapted to the specific needs of instrumentation software. In the remainder of this section, we outline the stages in this architectural development.

3.2.1 AN OBJECT-ORIENTED MODEL

The first attempt at developing a reusable architecture focused on producing an object-oriented model of the software domain, which clarified the data types used in oscilloscopes:

waveforms, signals, measurements, trigger modes, and so on (see Figure 3.6). While this was a useful exercise, it fell far short of producing the hoped-for results. Although many types of data were identified, there was no overall model that explained how the types fit together. This led to confusion about the partitioning of functionality. For example, should measurements be associated with the types of data being measured, or represented externally? Which objects should the user interface interact with?

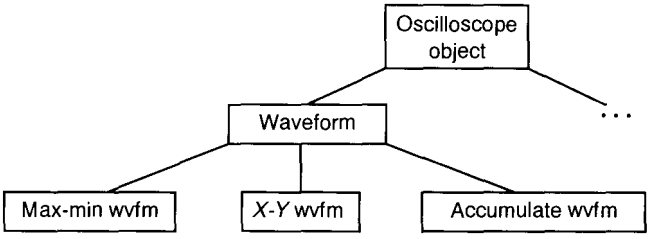

FIGURE 3.6 Oscilloscopes: An Object-Oriented Model

3.2.2 A LAYERED MODEL

The second phase attempted to correct these problems by providing a layered model of an oscilloscope (see Figure 3.7), in which the core layer represented the signal-manipulation functions that filter signals as they enter the oscilloscope. These functions are typically implemented in hardware. The next layer represented waveform acquisition. Within this layer signals are digitized and stored internally for later processing. The third layer consisted of waveform manipulation, including measurement, waveform addition, Fourier transformation, and so on. The fourth layer consisted of display functions, which were responsible for mapping digitized waveforms and measurements to visual representations. The outermost layer was the user interface. This layer was responsible for interacting with the user and for deciding which data should be shown on the screen (see Figure 3.7).

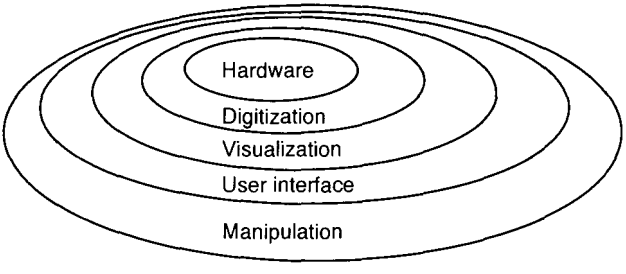

FIGURE 3.7 Oscilloscopes: A Layered Model

The layered model was intuitively appealing since it partitioned the functions of an oscilloscope into well-defined groups. Unfortunately it was the wrong model for the application domain. The main problem was that the boundaries of abstraction enforced by the layers conflicted with the needs for interaction among the various functions. For example, the model suggests that all user interactions with an oscilloscope should be in terms of the

visual representations. In practice, however, real oscilloscope users need to directly affect the functions in all layers, such as setting attenuation in the signal-manipulation layer, choosing acquisition mode and parameters in the acquisition layer, or creating derived waveforms in the waveform-manipulation layer.

3.2.3 A PIPE-AND-FILTER MODEL

The third attempt yielded a model in which oscilloscope functions were viewed as incremental transformers of data. Signal transformers are used to condition external signals. Acquisition transformers derive digitized waveforms from these signals. Display transformers convert these waveforms into visual data (see Figure 3.8).

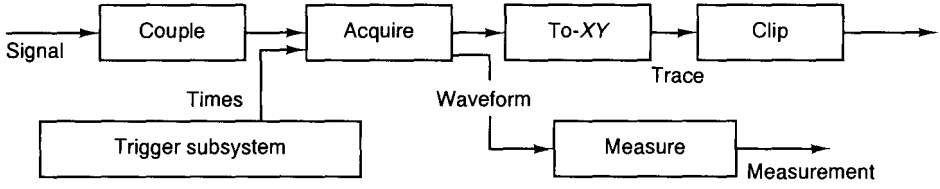

FIGURE 3.8 Oscilloscopes: A Pipe-and-Filter Model

This architectural model was a significant improvement over the layered model because it did not isolate the functions in separate partitions. For example, nothing in this model would prevent signal data from feeding directly into display filters. Further, the model corresponded well with the engineers' view of signal processing as a dataflow problem, and allowed the clean intermingling and substitution of hardware and software components within a system design.

The main problem with the model was that it was not clear how the user should interact with it. If the user were simply at the visual end of the system, this would represent an even worse decomposition than the layered system.

3.2.4 A MODIFIED PIPE-AND-FILTER MODEL

The fourth solution accounted for user inputs by associating with each filter a control interface that allowed an external entity to set parameters of operation for the filter. For example, the acquisition filter could have parameters that determined sample rate and waveform duration. These inputs serve as configuration parameters for the oscilloscope. One can think of such filters as having a "control panel" interface that determines what function will be performed across the conventional input/output dataflow interface. Formally, the filters can be modeled as "higher-order" functions, for which the configuration parameters determine what data transformation the filter will perform. (In Section 6.2 we sketch this formal model.) Figure 3.9 illustrates this architecture.

The introduction of a control interface solves a large part of the user interface problem. First, it provides a collection of settings that determine what aspects of the oscilloscope can be modified dynamically by the user. It also explains how the user can change

oscilloscope functions by incremental adjustments to the software. Second, it decouples the signal-processing functions of the oscilloscope from the actual user interface: the signal-processing software makes no assumptions about how the user actually communicates changes to its control parameters. Conversely, the actual user interface can treat the signal-processing functions solely in terms of the control parameters. Designers can therefore change the implementation of the signal-processing software and hardware without affecting the implementation of the user interface (provided the control interface remained unchanged).

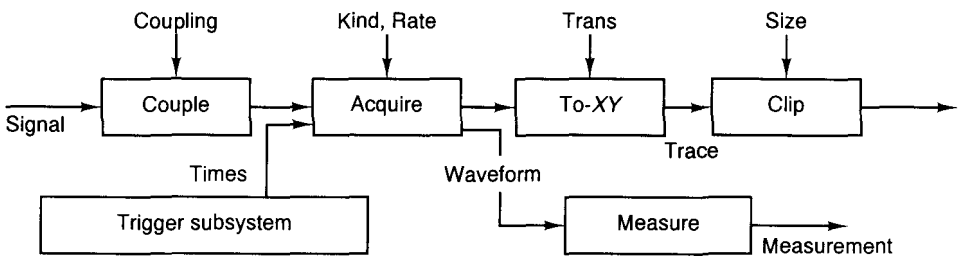

FIGURE 3.9 Oscilloscopes: A Modified Pipe-and-Filter Model

3.2.5 FURTHER SPECIALIZATION

The adapted pipe-and-filter model was a great improvement, but it, too, had some problems. The most significant problem was that the pipe-and-filter computational model led to poor performance. In particular, because waveforms can occupy a large amount of internal storage, it is simply not practical for each filter to copy waveforms every time they process them. Further, different filters may run at radically different speeds: it is unacceptable to slow one filter down because another filter is still processing its data.

To handle these problems the model was further specialized. Instead of using a single kind of pipe, we introduced several "colors" of pipes. Some of these allowed data to be processed without copying. Others permitted slow filters to ignore incoming data if they were in the middle of processing other data. These additional pipes increased the stylistic vocabulary and allowed the pipe/filter computations to be tailored more specifically to the performance needs of the product.

3.2.6 SUMMARY

This case study illustrates some of the issues involved in developing an architectural style for a real application domain. It underscores the fact that different architectural styles have different effects on the solution to a set of problems. Moreover, it illustrates that architectural designs for industrial software must typically be adapted from pure forms to specialized styles that meet the needs of the specific domain. In this case, the final result depended greatly on the properties of pipe-and-filter architectures, but adapted that generic style so that it could also satisfy the performance needs of the product family.

3.3 MOBILE ROBOTICS

By Marco Schumacher

A mobile robotics system is one that controls a manned or partially manned vehicle, such as a car, a submarine, or a space vehicle. Such systems are finding many new uses in areas such as space exploration, hazardous waste disposal, and underwater exploration.

Building the software to control mobile robots is a challenging problem. These systems must deal with external sensors and actuators, and they must respond in real time at rates commensurate with the activities of the system in its environment. In particular, the software functions of a mobile robot typically include acquiring input provided by its sensors, controlling the motion of its wheels and other moveable parts, and planning its future path. Several 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 it moves; the robot may manipulate hazardous materials; and unpredictable events may demand a rapid response.

Over the years, many architectural designs have been proposed for robotic systems. The richness of the field permits interesting comparisons of the emphases chosen by different researchers. In this section we consider four representative architectural designs.

3.3.1 DESIGN CONSIDERATIONS

To help create a framework for comparison of the architectural approaches, we enumerate below some of the basic requirements (Req1 through Req4) for a mobile robot's architecture.

- **Req1:** The architecture must accommodate *deliberative and reactive behavior.* The robot must coordinate the actions it undertakes to achieve its designated objective (e.g., collect a rock sample) with the reactions forced on it by the environment (e.g., avoid an obstacle).
- **Req2:** The architecture must allow for *uncertainty.* The circumstances of a robot's operation are never fully predictable. The architecture must provide a framework in which the robot can act even when faced with incomplete or unreliable information (e.g., contradictory sensor readings).
- **Req3:** The architecture must *account for dangers* inherent in the robot's operation and its environment. By considering fault tolerance, safety, and performance, the architecture must help maintain the integrity of the robot, its operators, and its environment. Problems like reduced power supply, dangerous vapors, or unexpectedly opening doors should not lead to disaster.
- **Req4:** 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 in a given situation 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.

We now examine four architectural designs for mobile robots: Lozano's control loops [LP90], Elfes's layered organization [Elf87], Simmons's task-control architecture [Sim92], and Shafer's application of blackboards [SST86]. We will use the four requirements listed above to guide the evaluation of these alternatives.

3.3.2 SOLUTION 1: CONTROL LOOP

Unlike mobile robots, most industrial robots only need to 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. (Rather, the environment 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 [LP90].

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 according to the monitored information. Figure 3.10 illustrates the control-loop paradigm. Let us consider how this model handles the four requirements listed above.

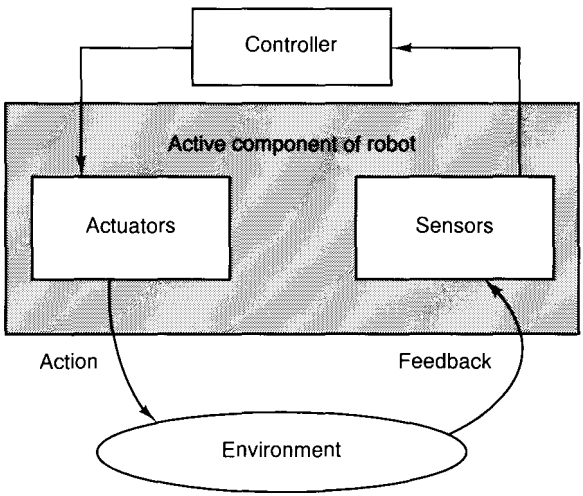

FIGURE 3.10 A Control-Loop Solution for Mobile Robots

- **Req1:** An advantage of the closed-loop paradigm is its simplicity: it captures the basic interaction between the robot and the outside. However, its simplicity is also a drawback in the more unpredictable environments. One expert [LP90] notes that the feedback loop assumes that changes in the environment are continuous and require continuous reactions (e.g., the control of pressure through the gradual opening and closing of a valve). Robots, though, are mostly confronted with disparate, discrete events that require them to switch 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 suggest how such mode changes should be handled.

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

- **Req2:** 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.
- **Req3:** 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.
- **Req4:** The major components of a robot architecture (supervisor, sensors, motors) are separated from each other and can be replaced independently. More refined tuning must take place inside the modules, at a level of detail that the architecture does not show.

In summary, the closed-loop paradigm seems most appropriate for simple robotic systems that must handle only a small number of external events and whose tasks do not require complex decomposition.

3.3.3 SOLUTION 2: LAYERED ARCHITECTURE

Figure 3.11 shows Alberto Elfes's definition of the idealized layered architecture [Elf87], which influenced the design of the Dolphin sonar and navigation system, implemented on the Terregator and Neptune mobile robots [CBCD93, PDB84].

- At level 1, the lowest level, reside the robot control routines (motors, joints, etc.).
- Levels 2 and 3 deal with 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.
- 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.
- **Req1:** Elfes's 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 indicates the concerns that must be addressed (e.g., sensor integration). Furthermore, it defines abstraction levels (e.g., robot control vs. navigation) to guide the design.

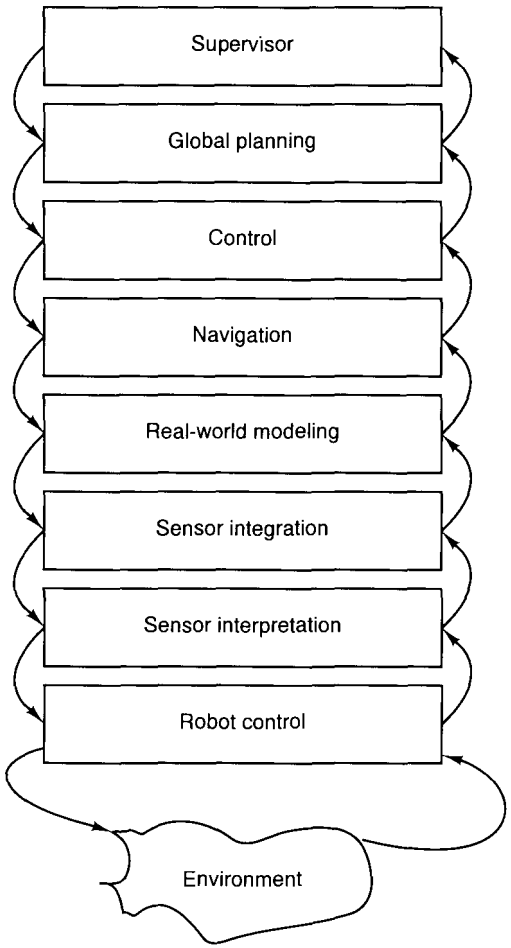

FIGURE 3.11 A Layered Solution for Mobile Robots

While it nicely organizes 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 acknowledges, information exchange is less straightforward. For instance, data requiring 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 problem in the model is that it does not separate 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).

(Some other layered architectures, such as NASREM, do a better job in this respect. We will mention some of these later.)

- **Req2:** 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.
- **Req3:** Fault tolerance and passive safety (when you strive not to do something) are also served by the abstraction mechanism. 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.
- **Req4:** The interlayer dependencies are an obstacle to easy replacement and addition of components. Further, complex relationships between the layers can become more difficult to decipher with each change.

In summary, the abstraction levels defined by the layered architecture provide a framework for organizing the components that succeeds because it is precise about the roles 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 communication patterns in a robot are not likely to follow the orderly scheme implied by the architecture.

3.3.4 SOLUTION 3: IMPLICIT INVOCATION

The third solution is based on a form of implicit invocation, as embodied in the Task-Control Architecture (TCA) [Sim92]. Figure 3.12 sketches the architecture. It has been used to control numerous mobile robots, such as the Ambler robot [Sim90].

The TCA architecture is based on hierarchies of tasks, or *task trees*. Figure 3.13 shows a sample task tree, in which parent tasks initiate child tasks. The software designer can define temporal dependencies between pairs of tasks; for example, a common temporal constraint is that *A* must complete before *B* starts. These features permit the specification of selective concurrency. The implementation of TCA includes many operations for dynamic reconfiguration of task trees at run time in response to changing robot state and environmental conditions.

TCA uses implicit invocation to coordinate interactions between tasks. Specifically, tasks communicate by multicasting messages via a message server, which redirects those messages to tasks that are registered to handle them.

In addition to the communication of messages, TCA's implicit invocation mechanisms support three functions:

1. **Exceptions:** Certain conditions cause the execution of an associated exception handler. Exceptions override the currently executing task in the subtree 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 archi-

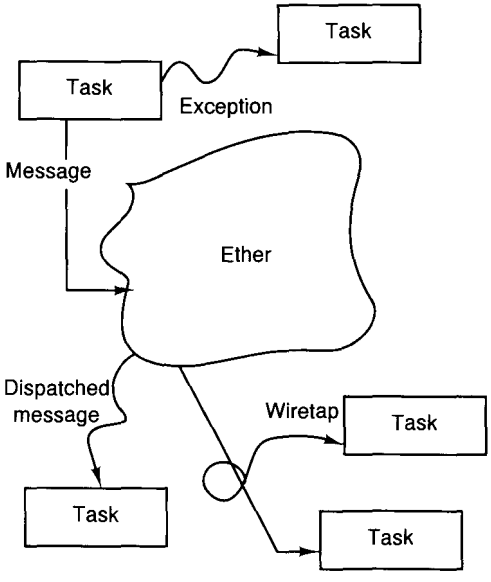

FIGURE 3.12 An Implicit Invocation Architecture for Mobile Robots

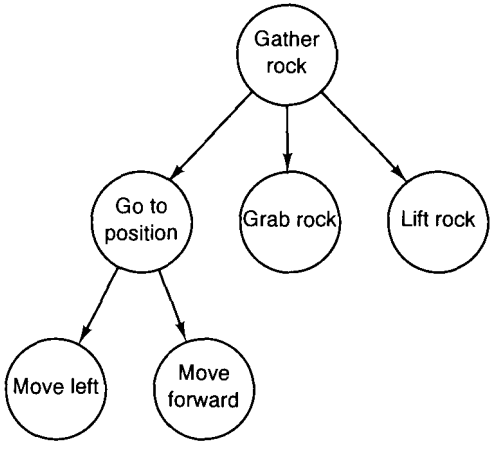

FIGURE 3.13 A Task Tree for Mobile Robots

tecture. Exception handlers have at their disposal all the operations for manipulating the task trees: for example, they can abort or retry tasks.

2. **Wiretapping:** Messages can be intercepted by tasks superimposed on an existing task tree. For instance, a safety-check component can use this feature to validate outgoing motion commands.
3. **Monitors:** Monitors read information and execute some action if the data fulfills 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.

We turn now to the requirements.

- **Req1:** Task trees on the one hand, and exceptions, wiretapping, and monitors on the other, permit a clear-cut separation of action (the 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 explicitly 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 explicitly address concurrency.

  In practice, the amount of concurrency of a TCA system 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.
- **Req2:** How TCA addresses uncertainty is less clear. If imponderables exist, a tentative task tree can be built, to be modified by the exception handlers if its fundamental assumptions turn out to be erroneous.
- **Req3:** 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.
- **Req4:** 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 components are affected.

In summary, TCA offers a comprehensive set of features for coordinating the tasks of a robot while respecting the requirements for quality and ease of development. The richness of the scheme makes it most appropriate for more complex robot projects.

3.3.5 SOLUTION 4: BLACKBOARD ARCHITECTURE

Figure 3.14 describes a blackboard architecture for mobile robots. This paradigm was used in the NAVLAB project, as part of the CODGER system [SST86]. The *whiteboard* architecture, as it is termed in [SST86], works with abstractions reminiscent of those encountered in the layered architecture. The components of CODGER are the following:

- 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.

Let us consider the requirements.

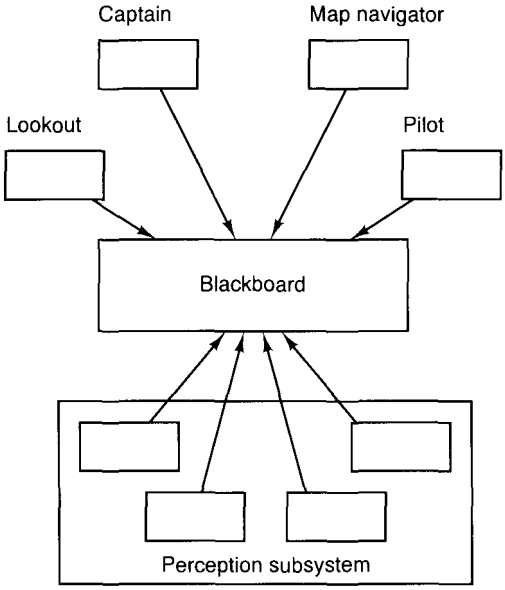

FIGURE 3.14 A Blackboard Solution for Mobile Robots

- **Req1:** The components (including the modules inside the perception subsystem) communicate via the shared repository of the blackboard system. Modules indicate their interest in certain types of information. The database returns them such data either immediately or when some other module inserts it 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 control flow has to be coerced to fit the database mechanism, even under circumstances where direct interaction between components would be more natural.
- **Req2:** 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 sets of data are stored in the database. The modules responsible for resolving the uncertainty register with the database to obtain the necessary data. (A good example of this activity is sensor fusion, performed by the perception subsystem to reconcile the input from its diverse sensors.)
- **Req3:** 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 signs of unexpected occurrences or the beginnings of troublesome situations.
- **Req4:** As with TCA, the blackboard architecture supports concurrency and decouples senders from receivers, thus facilitating maintenance.

In summary, the blackboard architecture is capable of modeling the cooperation of tasks, both for coordination and resolving uncertainty in a 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.

3.3.6 COMPARISONS

In this section we have examined four architectures for mobile robotics to illustrate how architectural designs can be used to evaluate the satisfaction of a set of requirements. The four architectures differ substantially in their control regimes, their communications, and their specificity of components, which affects the degree to which they satisfy the requirements, as shown in Figure 3.15.

	Control Loop	Layers	Implicit invocation	Blackboard
Task Coordination	+ –	–	+ +	+
Dealing with uncertainty	–	+ –	+ –	+
Fault intolerance	+ –	+ –	+ +	+
Safety	+ –	+ –	+ +	+
Performance	+ –	+ –	+ +	+
Flexibility	+ –	–	+	+

FIGURE 3.15 Table of Comparisons

Naturally, these architectures are not the only possibilities; dozens of other architectures exist. Many of these are hybrids. For example, the The NASA/NBS Standard Reference Model for Telerobots (NASREM) [LFW90] is an architecture that combines control loops with layers, while Hayes-Roth's architecture combines layers with data flow [HRPL+95].

3.4 CRUISE CONTROL

In this section we illustrate how to apply the control-loop paradigm to a simple problem that has traditionally been cast in object-oriented terms. As we will show, the use of control-loop architectures can contribute significantly to clarifying the important architectural dimensions of the problem.