IEEE Software, 1995, vol. 12, no. 6, pp. 27-41

Making Choices: A Comparison of Styles for Software Architecture

Mary Shaw
School of Computer Science and
Software Engineering Institute
Carnegie Mellon University
Pittsburgh Pa 15213

mary.shaw@cs.cmu.edu

May 1994

Abstract

Good engineering practice solves problems not only by applying scientific techniques but also by making design choices that reconcile conflicting requirements. We are interested here in design of the overall organizations and system-level properties of software systems—that is, their architectures. Early decisions about design strategies can have far-reaching consequences, because they shape the analysis of the problem and the expression of the design. This paper explores the consequences of one of the earliest decisions, the choice of architectural style and its associated notations. The paper shows how different architectural styles lead not simply to different designs, but to designs with distinctly—and significantly—different properties. I examine eleven different published designs for "the same" problem (automobile cruise control), classify and compare the approaches, and discuss major differences among the resulting solutions. Although all the designers nominally designed automobile cruise controls, they actually produced a wide range of solutions to somewhat different problems. The issues addressed in the designs depend on the choice of architectural style, and most styles use multiple models in the design. The comparison illustrates some of the relative advantages and shortcomings of the styles and provides some guidance for selection.

Table of Contents

1. Design Idioms for Software Architectures

Engineering is the creation of cost-effective solutions to practical problems, usually by applying well-organized scientific knowledge. Since most engineering problems require a designer to resolve conflicting constraints, engineers often explore a space of possible designs for the system organization. Explicit patterns, or idioms, increasingly guide the organization of modules and subsystems into complete systems. This stage of the design is usually called the architecture, and a number of common patterns are commonly, though quite informally, used [Garlan&Shaw 93]. The choice of an architecture, or organizational principle, for a software system is made early in the life of the system. Although strong advocates of some architectural idioms tout each as the best choice for all problems, designers should select an architecture to match the needs of each problem. The choice affects not only the system description and the decomposition into components, but also functionality, performance, and other important properties.

Designers often fail to explain their architectural decisions, and the architectures are often not permanently retained with the code. Although many design idioms are available, they are not clearly described or distinguished; further, the consequences of a decision are not well understood. Just as Wing found in a dozen specifications of the library problem [Wing 88], the most interesting aspects of the comparison have to do with what the architectural styles elicit about the systemd being designed , This paper contributes to our understanding of the relative advantages of different architectural idioms by examining eleven designs developed by nine design groups for a single example, automobile cruise control.

1.1. The Example: Automobile Cruise Control

Disciplines often work out the details of their methods through *type problems*, common examples used by many different people to compare their models and methods [Shaw et al 94]. The cruise control problem serves this purpose for software architecture and related issues. One commonly-used version [Booch 86, Birchenough&Cameron 89] is:

> 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 increase and decrease speed as directed by the driver. Figure 1 gives 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.
> - **Incr/Decr 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 engine throttle setting.

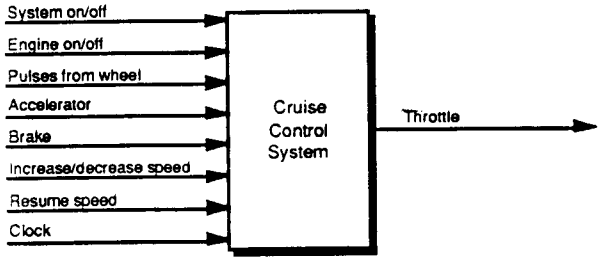

Figure 1: Booch block diagram for cruise control

Another, slightly more complex, version includes monitoring functions as well [Brackett 87, Kirby 87].

1.2. Design Considerations

Designs can be evaluated in any number of ways. The precise criteria should depend on the requirements of each application. However, certain considerations apply generally:

- *Locality and separation of concerns:* Does the design separate independent parts of the system, group closely related parts, and avoid redundant representation of information? How easy will it be to make modifications?
- *Perspicuity of design:* Does the expression of the design correspond clearly to the problem being solved? Is the design's response to the most significant requirements easy to identify and check?
- *Analyzability and checkability:* Is the design easy to analyze or check for correctness, performance, and other properties of interest? If more than one model or notation is used, how easy is it to understand their interaction?
- *Abstraction power:* Does the design highlight and hide the appropriate details? Does it help the designer avoid premature implementation decisions?

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 issues 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?
- *Integration with vehicle:* How well do inputs and outputs match the actual controls. How do the manual and automatic modes interact? What are the characteristics of real-time response? How rapidly and smoothly does the vehicle respond to control inputs? How accurately does the system maintain speed?

1.3. Solutions

At least a dozen groups have used cruise control as an example, presenting over twenty designs. Some of these are presented as requirements but are sufficiently concrete to serve as architectures. Eleven of these examples are summarized in Section 2 through 5. Most use either Booch's [Booch 86] or Brackett's [Brackett 87] formulation.

Section 2: Booch uses the example to motivate object-oriented programming [Booch 86, adapted from Ward 84] and includes a functional definition as a strawman. Yin and Tanik present an object-oriented solution to demonstrate reusability in Ada [Yin&Tanik 91]. Birchenough and Cameron show how the Jackson System Development Method (JSD) complements object-oriented design [Birchenough&-Cameron 89]. Kirby applies the NRL information-hiding technique [Kirby 87].

Section 3: Smith and Gerhart illustrate the use of Statemate; their design is, of course, based on states and activities [Smith&Gerhart 88]. Atlee and Gannon use a state-based formulation to illustrate model-checking for requirements [Atlee&Gannon 93].

Section 4: Higgins emphasizes feedback control models in showing how Data Structured Systems Development can be extended for real-time process control systems [Higgins 87]; his architecture . Shaw also bases a solution on feedback control, using other architectures for subsystems [Shaw 94].

Section 5: 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 [Ward&Keskar 87].

Additional designs: Wasserman and others present an object-oriented design to illustrate a proposed new methodology [Wasserman 89]. Gomaa uses this example illustrate a new design method for real-time systems [Gomaa 89]. Gomaa's text includes designs in several styles [Gomaa 93]. Wang and Tanik develop a dataflow solution to illustrate Process Port Analysis and XYZ/E [Wang&Tanik 89]. Jones considers the testing problem for an Ada program but is not explicit about the character of the software [Jones 90].

2. Object-Oriented Architectures and Information Hiding

Object-oriented architectures decompose systems into discrete objects that encapsulate state and definitions of operations. These objects interact by invoking each others' operations. Booch defines an object as "an entity that has state; is characterized by the actions that it suffers and that it requires of other objects; is an instance of some class; is denoted by a name; has restricted visibility of and by other objects; may be viewed either by its specification or by its implementation" [Booch 86] The first two terms of this definition are structural; the others affect the way objects are defined.

A number of methods for object-oriented design have been proposed. They differ in their strategies for defining objects; for defining timing, sequencing and other dynamic properties; and for establishing that the design satisfies the requirements. As these examples show, other models may complement the object-oriented part of the design.

2.1. Booch: Object-Oriented Programming

Booch begins by modeling the problem space in a data flow diagram (Figure 2). This shows how information passes from its various sources through computations and internal states to the output value. From this model of the requirements, he presents a functional decomposition of the design in which modules correspond to major functions in the overall process (Figure 3). This serves primarily as a foil for the object-oriented design.

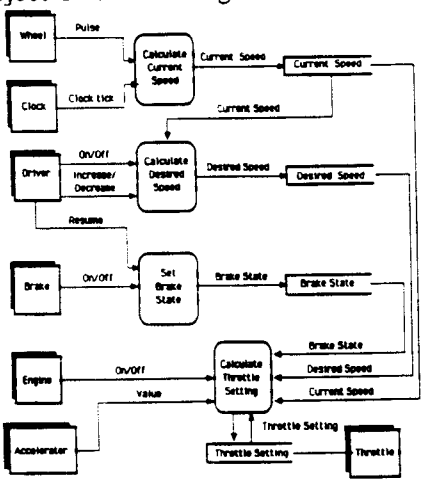

Figure 2: Booch's data flow diagram

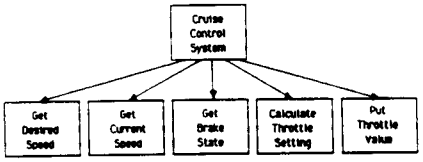

Figure 3: Booch's functional design

From the same data flow model of the requirements, Booch then structures an object-oriented decomposition around objects that exist in the task description (Figure 4). This yields an architecture whose elements correspond to important quantities and physical entities in the system. The blobs of Figure 4 correspond approximately to the inputs and internal states of Figure 2. The arrows of Figure 4 are derived from the data paths of Figure 2. Note that the object-oriented designs of Sections 2.2 and 2.3 show slightly different dependencies.

Note that the design emphasizes the objects in the problem domain and the dependencies they have with major elements of internal state. The design makes no distinction between objects of the problem domain and internal objects, nor does it show the nature of the dependencies.

2.2. Yin & Tanik: Reusability

Yin and Tanik use cruise control to demonstrate software reusability [Yin&Tanik 84]. They chose the example because it requires parallel processing, real-time control, exception handling, and unique input/output control. They begin from Booch's problem specification, using essentially the same data flow diagram to model the requirements. Following Booch's development technique, they derive an object-oriented design (Figure 5). This differs from Booch's (Figure 4) in that they create objects for all the external elements and one single object for the entire cruise control system; thus Figure 5 shows just the dependencies of the core solution on the external elements and any interactions between external elements. The system architecture of Figure 6 elaborates Figure 5 by showing the operations for the major objects. It also rearranges the design substantially: all other objects are (evidently) internal to the engine, and the relation of throttle to everything else is much different. Although the problem was chosen because of its real-time characteristics, the design does not address timing questions. Yin and Tanik comment on the need for additional facilities for dealing with timing constraints.

2.3. Birchenough & Cameron: Complementarity of JSD and OOD

Jackson System Development (JSD) and Object-Oriented Design (OOD) both support the principle that the structure of the software system should match the structure of the problem it solves; both rely on identifying discrete entities (which correspond to objects) and the operations they commit on each other.

Birchenough and Cameron use cruise control, with essentially the same formulation as Booch, to show how the object view can complement JSD [Birchenough&Cameron 89].

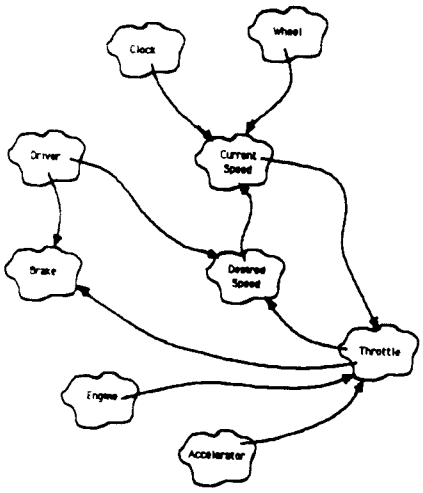

Figure 4: Booch's object-oriented diagram

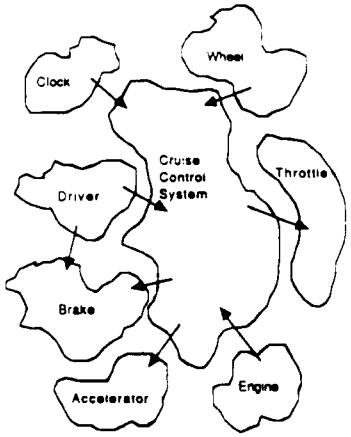

Figure 5 Yin & Tanik's object-oriented design

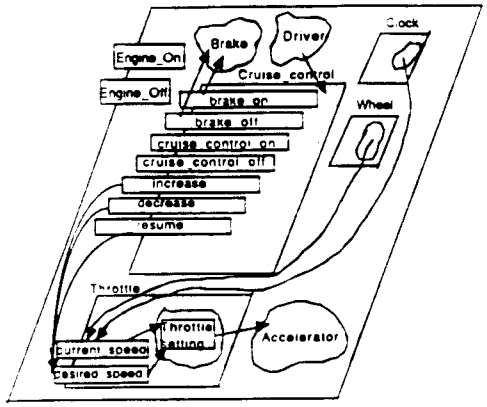

Figure 6: Yin & Tanik's system architecture

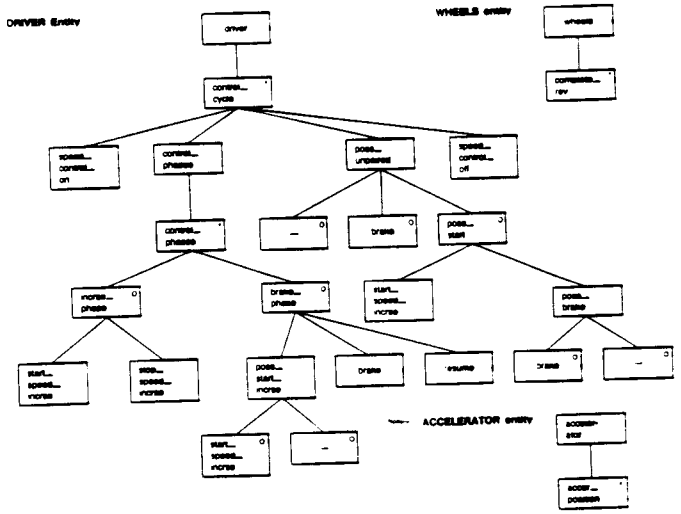

Figure 7: JSD entity structures

JSD has three stages: modeling, network, and implementation; this discussion deals with the modeling stage (JSD separates model from function). JSD encourages analysis of actions, or events, before identifying entities whereas OOD is the opposite. In JSD, the modeling stage only considers time-ordered or state-changing operations when identifying objects. As a result it is more conservative in object identification than OOD. For cruise control, Booch finds eight objects (Figure 3) whereas JSD finds three: the *wheels* entity, the *accelerator* entity, and the *driver* entity; the driver. These three, with their substructure, are shown in Figure 7. The substructure shows time-ordering among the operations through notations in the corners of the boxes. JSD's functional components are not chosen to adhere to an object discipline but for a variety of problem-specific reasons. Birchenough and Cameron conclude that a more object-oriented approach would add to the comprehensibility of later stages.

2.4. Kirby: Information Hiding

The Naval Research Laboratory/Software Cost Reduction (NRL/SCR) approach is based on information hiding, precision, and completeness. It provides separate definitions of inputs and outputs, the modes of operation, functionality, timing, accuracy, and undesired events. Each of these comprises a set of discrete definitions in uniform format; tables are used heavily but architectural diagrams are not.

Kirby uses Brackett's formulation [Brackett 87]. Because this version includes monitoring, the definition includes 20 input and output items; Figure 8 shows the Cruise Control Request. The modes of operation and the state transition table are essentially the same as provided by Atlee and Gannon (Section

3.2). Definitions of function rely on decision tables; Figure 9 gives the definition of the throttle setting function. Timing, accuracy, and undesired events are enumerated and tabulated in a similar manner.

Cruise Control Request

Input Data Item: Cruise control request

Acronym: /Lever/

Hardware: Cruise control lever

Description: The value of /lever/ is determined by the cruise control lever. The lever has five positions. Four of the five positions are labeled CONST, OFF (two instances), and RESUME. The lever may be pushed and held in one of these positions at a time. When released, the lever will always return to the unlabeled RELEASE position. The value of /lever/ is described by table.

Characteristics of Values:
Values: $inactivate$, $activate$, $start incr$, $stop incr$, $resume$

Inputs from Cruise Control Lever

Event	/Lever/
@T(enter *on*)	$inactivate$
@T(!const pos!)	$activate$
@T(!const pos! for > 500 msec)	$start incr$
@T(!release pos!) when !const pos! for > 500 msec	$stop incr$
@T(!off pos!)	$inactivate$
@T(!resume pos!)	$resume$

Figure 8: Kirby's Input Data Item

Determine Throttle Setting Function

Modes in which function is required: *Cruise*

Output data item: //Throttle//

Initiation Event: @T(enter mode)

Termination Event: @T(exit mode)

Output description: Table shows required value of //throttle//

	Throttle Setting			
Mode	Condition			
cruise	!too slow! OR !start incr!	!speed ok! AND NOT !start incr! AND NOT !driver accel!	!too fast! AND NOT !start incr! AND NOT !driver accel!	!driver accel! AND NOT !too slow! AND NOT !start incr!
//Throttle//	!throttle accel!	!throttle maintain!	!throttle decel!	!throttle off!

Figure 9: Kirby's Throttle Function

3. State-Based Architectures

State-based architectures focus primarily on the major modes, or states, of the system and the events that cause transitions between states. As with object-oriented designs, other models may complement the state analysis, for example to identify the major modes. State machines also appear as secondary views in designs by Kirby (Section 2.4), Shaw (Section 4.2) and Ward & Keskar (Section 5.1).

3.1. Smith & Gerhart: STATEMATE

Statemate uses a state transition formalism supported by two graphical notations: activity charts (a form of functional decomposition) and statecharts (a representation of finite-state machines). It is particularly well-suited to reactive systems. It can be used with many methodologies: functional methods focus on activity charts, whereas behavioral methods focus on statecharts. Smith and Gerhart [Smith&Gerhart 88] use Brackett's formulation of cruise control [Brackett 87] to compare Statemate to Brackett's use of the Hatley method, which uses a graphical notation for a functional view. They abstract from physical devices such as "brake pedal" to the actions selected by the driver such as "inactivate system".

In order to compare their result to Brackett's, Smith and Gerhart choose the functional decomposition design method. The top-level functions emerge directly from the system requirements; they are the boxes visible at the boundary of the system as shown in the activity chart of Figure 10. Since all the functions at this level operate concurrently, no statechart is required. The granularity of the activities is larger than Booch's functional elements, which are simple constructors and selectors.

The second level decomposes the internal activities of the first level. Smith and Gerhart illustrate this step with the function for speed control. The requirements again lead to the functions in the decompositions shown in Figure 11. Now, however, ordering dependencies affect execution, so a controlling statechart is required. This appears as a shaded box in Figure 11 and is elaborated in Figure 12.

Smith and Gerhart observe that the advantages of Statecharts include good notations for showing concurrency and restrictions on concurrency, mechanisms for showing state changes, notations for certain timing constraints, and the simulation and analysis capabilities of the tool. However, this formulation required more notation than others for the same level of detail and large (or variable) numbers of similar

activities are hard to represent. They also note that behavioral and functional approaches will lead to different designs.

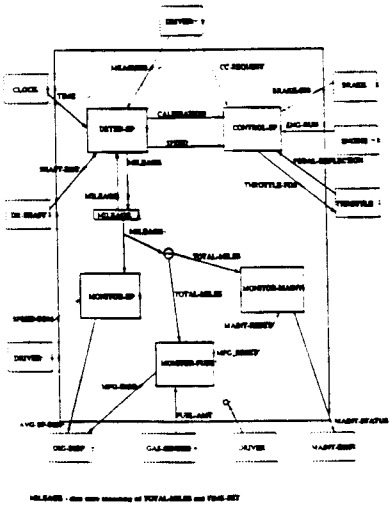

Figure 10: Smith & Gerhart's top-level activity chart

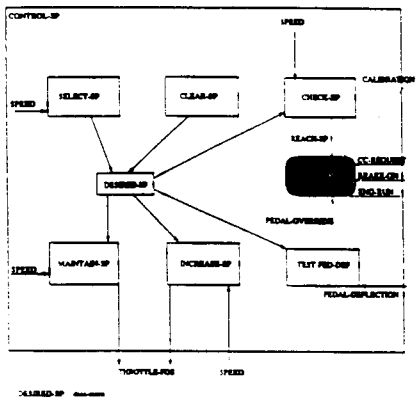

Figure 11: Smith & Gerhart's speed control activity chart

3.2. Atlee & Gannon: State-Based Model Checking

Atlee and Gannon are primarily interested in using a model checker on requirements for large systems. Their state-based formulation of cruise control is therefore presented as a requirement, but it is close enough to a design for consideration here. They use Brackett's formulation as presented by Kirby [Kirby 87]. The basic formulation says that a cruise control system can be in one of four modes (which correspond to intuitive states): off, inactive, cruise, and override (on but not in control). The requirements are given as a table (Figure 13) that shows how events (flagged "@") and conditions cause mode transitions. Their analysis considers interactions among conditions ("what if the brake is on when the system is activated?") and identifies important invariants among conditions and events. Their definition is essentially the same as Kirby's for this aspect of the problem.

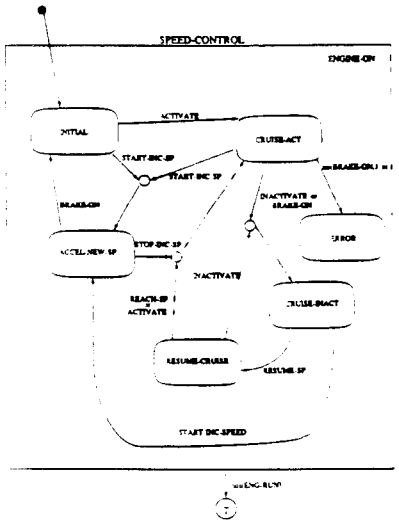

Figure 12: Smith & Gerhart's speed control state chart

MODE TRANSITIONS FOR AUTOMOBILE CRUISE CONTROL

Current Mode	Ignited	Running	Too Fast	Brake	Activate	Deactivate	Resume	New Mode
OFF	@T	—	—	—	—	—	—	INACTIVE
INACTIVE	@F	—	—	—	—	—	—	OFF
	t	t	—	f	@T	—	—	CRUISE
CRUISE	@F	—	—	—	—	—	—	OFF
	—	@F	—	—	—	—	—	INACTIVE
	—	—	@T	—	—	—	—	
	—	—	—	@T	—	—	—	OVERRIDE
	—	—	—	—	—	@T	—	
OVERRIDE	@F	—	—	—	—	—	—	OFF
	—	@F	—	—	—	—	—	INACTIVE
	t	t	—	f	@T	—	—	CRUISE
	t	t	—	f	—	—	@T	

Initial mode: OFF.

Figure 13: Atlee & Gannon's mode transition table

4. Feedback Control Architectures

Feedback control architectures are a special form of dataflow architecture adapted for embedded systems where the process must be regularly tuned to compensate for external perturbations. These architectures are modeled on analog process control systems, and they model the current values of the outputs and the value that controls the process as continuously available signals.

4.1. Higgins: Extending DSSD for Real-Time Software

Higgins extends the Data Structured System Development (DSSD) method for real-time embedded systems by adding a standard template for analyzing feedback/control models [Higgins 87]. The conversion from a standard feedback control block diagram to a DSD entity diagram is straightforward; it includes a summing point for feedback and explicit recognition of the external disturbance and the source of the target value of the process variable as well as the more obvious processing elements (Figure 14). This entity diagram is static, so it is complemented by a functional flow diagram to establish the communication dynamics (Figure 15). This diagram shows how the input signals produce the output signals: the inputs to each "level of flow" are on the right and the output is on the left; the lowest element on the input list is the name of the transformation. When the system flow is defined, the final step is to convert the functional flow diagram to a data structure diagram showing the hierarchy of system data .

Higgins uses cruise control to illustrate the method. A simple form of the example is shown here; additional levels of control such as whether the engine is on, the system is activated, etc are handled as secondary control loops that embed this example as the controlled system in a surrounding control loop.

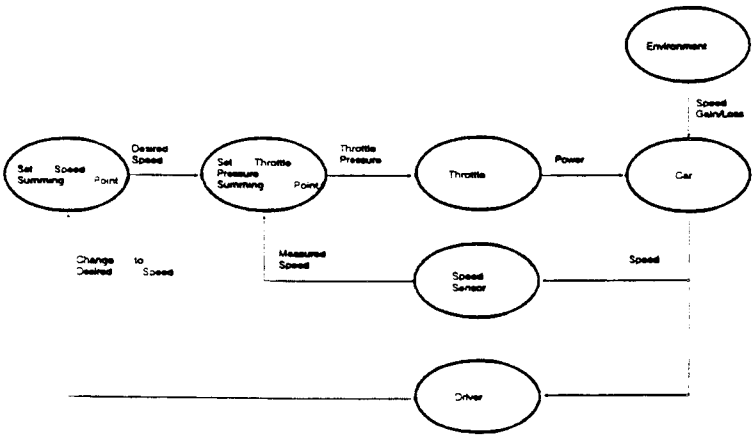

Figure 14: Higgins' feedback entity diagram

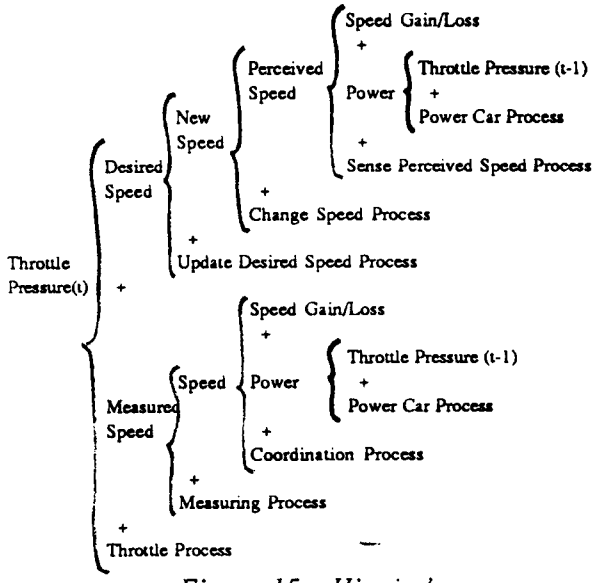

Figure 15: Higgins' functional flow diagram

4.2. Shaw: Process-Control Paradigm

Shaw explores a software idiom based on process control loops [Shaw 94] using Booch's formulation of the problem [Booch 86]. Unlike object-oriented or functional designs which are characterized by the kinds of components that appear, control loop designs are characterized both by the kinds of components and the special relations that must hold among the components. The elements of this pattern incorporate the essential parts of a process control loop, and the methodology requires explicit identification of these parts. The parts include two computational elements, the process definition and the control algorithm; three data elements, the process variables, the set point or reference value, and sensors; and the control loop paradigm that establishes how the control algorithm drives the process. She characterizes the result as a specialized form of data flow architecture.

The essential control problem establishes a control relation over the engine (Figure 16). This does not, however, solve the whole of the problem, as most of Booch's inputs are actually used to determine whether the system is active or inactive and to set the desired speed. The former is a state transition problem, and the solution (Figure 17) is much like that the state-based designs. The latter problem lends itself to a decision table as in Figure 18.

Figure 19 shows how to compose the control architecture, the state machine for activation, and the event table for determining the set point into an entire system.

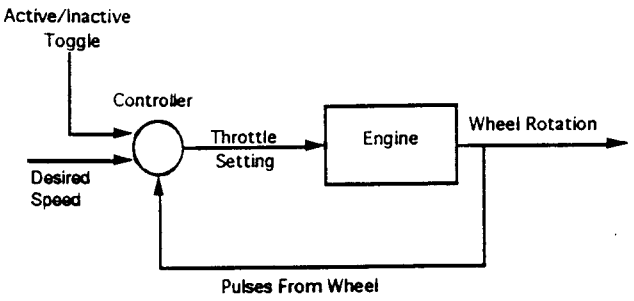

Figure 16: Control Architecture for Cruise Control

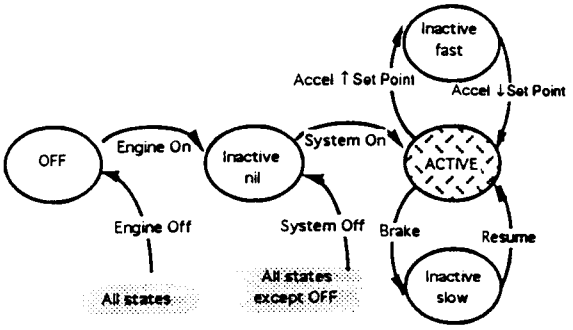

Figure 17: State Machine for Activation

Event	Effect on desired speed
Engine off, system off	Set to "undefined"
System on	Set to current speed as estimated from wheel pulses
Increase speed	Increment desired speed by constant
Decrease speed	Decrement desired speed by constant

Figure 18: Event Table for Determining Set Point

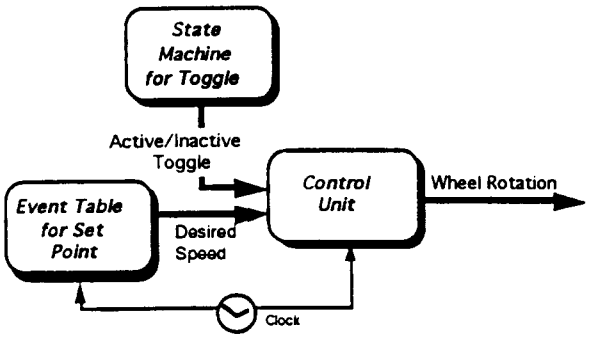

Figure 19: Complete Cruise Control Design

Shaw concludes that it is appropriate to consider a control loop design when the task involves continuing action, behavior, or state, the software is *embedded*; that is, it controls a physical system, and when uncontrolled, or open loop, computation does not suffice, usually because of external perturbations or imprecise knowledge of the external state.

5. Real-time System Analysis

Real-time systems must meet stringent response-time requirements. Extensions to several methods add various features for the special demands of real-time processing. Interestingly, these systems deal with event ordering but not absolute time. Higgins' process control model is motivated by real-time problems but is discussed above.

5.1. Ward & Keskar: Ward/Mellor and Boeing/Hatley Real-Time Methods

Ward and Keskar compare two extensions of DeMarco Structured Analysis (DMSA) that support real-time system models: Ward/Mellor (WM) and Boeing/Hatley (BH) [Ward&Keskar 87]. DMSA was developed for commercial business applications, and these extensions were developed because pure structured analysis couldn't effectively capture the sorts of time-dependent actions that appear in process control, avionics, medical instrumentation, communications, and similar domains. The basic requirements are stated informally and resemble Booch's.

The WM approach extends the basic structured analysis data flow diagram (solid lines, Figure 20) by adding event flows (dotted lines, Figure 20) to show time-dependent behavior. The control transformation "Control Speed" receives events from the external interface and enables, disables, or triggers the basic functions. The logic of the control transformations are described by the state transition diagram of Figure 21. In a given state, only certain events are recognized; when they occur, they change state (>) and enable (>>), disable (<<), or trigger ([ ) the indicated transformations. This design is meaningful only when the engine and cruise control system are both on. Ward and Keskar show a hierarchical extension with an additional control state to handle this. Some of the interactions examined by Atlee and Gannon can arise here; this can be prevented by adding explicit detail to handle them.

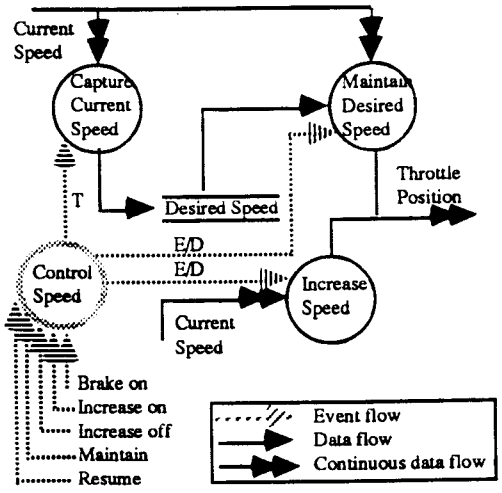

Figure 20: Ward & Keskar's WM control transformation diagram

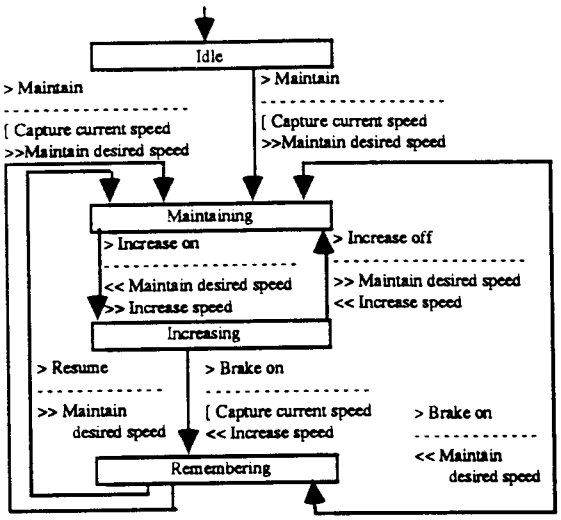

Figure 21: Ward & Keskar's WM state transition diagram

The BH extension begins with two context diagrams that show data and control flow between physical components of the system. The highest level of the design consists of a data flow diagram (DFD) (Figure 22) and a variant on the DFD to show control flow (Figure 23). These are based on the same entities, Next, control specifications show hot to activate or deactivate processes on the Data Flow Diagram. They may be combinatorial (no state) or sequential (internal state). The cruise control is sequential, so its control is described in a decision table (Figure 24) that converts combinations of input signals to output signals, a state-transition diagram (essentially similar to Figure 20), and an activation table (Figure 25) that relates the transition actions of the state machine to the processes of the DFD.

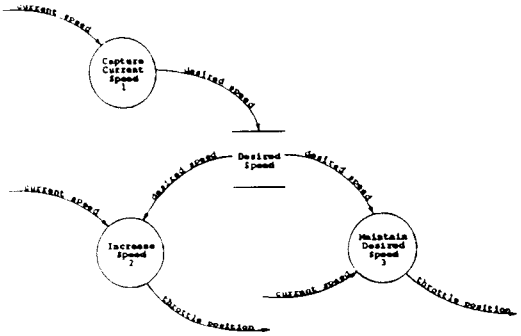

Figure 22: Ward & Keskar's BH data flow diagram

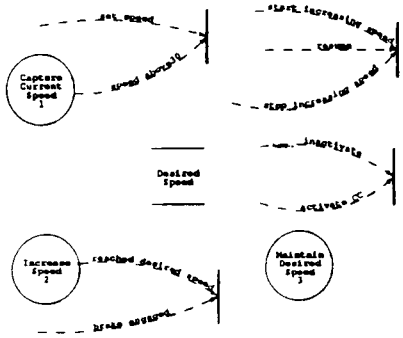

Figure 23: Ward & Keskar's BH control flow diagram

Speed_ Above 50	Set_ Speed	Hold_ Speed
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

Figure 24: Ward & Keskar's BH decision table

Process / Action	Capture Current Speed 1	Increase Speed 2	Maintain Speed 3
Stop_Accelerating	0	0	0
Accelerate	0	1	0
Maintain Speed	0	0	1
Accelerate_to_ Desired_Speed	0	1	0
Get_and_ Maintain_Speed	1	0	1
Stop_Maintaining_ Speed	0	0	0
Turn_CC_Off	0	0	0
Cruise_Control_ Ready	0	0	0

Figure 25: Ward & Keskar's BH activation table

6. Discussion

All these designs started from a single task; indeed, most started from one of two problem statements. Nevertheless, the solutions differ substantially, even within a single architectural style. Some of the differences can be attributed to variations among individual designers, but others arise from the way each architecture leads the designer to view the world.

Designs based on different architectures attend to different aspects of the problem. Some focus on interpreting the driver controls; some focus on internal state; some focus on the actual control of automobile speed. The ability to focus on issues of interest is important, but the designer should consciously match the technology to the needs of the client.

Even when the design is declared to be of a particular style, the designer usually appeals to two or three different design representations or models. These models are used in many combinations and provide different views of the design—the only real point of consistency is the model that matches the declared style. Within a single design, different models may decompose the task into different entities. Needless to say, not all of these can persist to runtime. Table 1 identifies the models used in these examples.

The design criteria of Section 1.2 provide a basis for comparing the design approaches, and the remainder of this section is organized correspondingly.

Design strategy [reference]	*Sect*	*Problem Version*	*Functional Decomp*	*Data Flow*	*Object-Oriented*	*State Machine*	*Event-Oriented*	*Process Control*	*Decision Table*	*Data Structure*
Functional [Booch 86]	2.1	Booch	√	for rqts						
Object-oriented [Booch 86]	2.1	Booch		for rqts	√					
Object-oriented [Yin&Tanik 91]	2.2	Booch		for rqts	√					
JSD and O-O [Birchenough&-Cameron 89]	2.3	Booch	JSD activity chart		√					
NRL/SCR [Kirby 87]	2.4	Brackett			info hiding	√			√	
Statemate [Smith&Gerhart 88]	3.1	Brackett	activity chart			state chart		—		
State machines [Atlee-Gannon 93]	3.2	Brackett				√				
Process-control [Higgins 87]	4.1	Higgins	√					√		√
Process-control [Shaw 94]	4.2	Booch				√		√	√	
Ward/Mellor [Ward&Keskar 87]	5.1	Ward-Keskar		mixed with event flow		√	mixed with data flow			
Boeing/Hatley [Ward&Keskar 87]	5.1	Ward-Keskar		√		√	control flows		√	

Table 1: Design models used in examples

6.1. Separation of concerns and locality

All the models used in these designs provide a way to decompose the system into separate parts that localize decisions that are significant *with respect to that model.* Generally speaking,

- Object-oriented designs focus on real-world entities and data/computational dependencies from input entities to output. They are little concerned with internal relations or operation sequencing.
- State-oriented designs focus on the modes in which the system operates and the conditions that cause transitions from one state to another.
- Process-control designs focus on the feedback relation between actual and desired speed.

- Real-time designs focus on events and the orders in which they appear; the real-time designs presented here don't say much about actual timing.

It is the choice of what information to localize and what concerns to separate that leads to the large differences in which aspects of the problem each design addresses.

Locality is motivated not only by design simplification but also by the ability to interchange parts between designs. When multiple architectural styles are used, they often lead to parts with different packaging—different ways of interacting with other parts. This can substantially interfere with exchange or interoperability; failure to recognize these discrepancies may be a major contributor to problems with software reuse.

Locality is also motivated by the prospect of future modifications. Booch, for example, argues that object decompositions are superior to functional decompositions because functional decompositions have global data and future changes may require representation changes. This is sometimes, but not always the case. As Parnas argued two decades ago, locality should hide the decisions *most likely* to be changed. Which decisions these are will vary from one application to another.

Separation of concerns becomes more complex when multiple models are used—as happens in most of these designs. Whenever multiple views are defined, they constrain the design in different ways. It is essential to show how those views are related (see Section 6.3).

6.2. Perspicuity of design

Perspicuity is in the mind of the beholder. Most methodologies exhort designers to make the design match the real world. As these examples show, the real world has many faces. Each of these designs can be defended as matching some view of reality, though some (objects, process control) do so more consciously than others (functional). As the discussion of safety (Section 6.5) shows, the designer needs to understand what aspects of the real world are most important to the client. Many methodologies begin with a domain analysis. This should lead to an initial choice of architecture.

If the client is most concerned with the devices the drive manipulates to indicate desired speed and changes of speed, object-oriented designs are a good match. If the client is most concerned with the possible modes of the system and assurance that obscure interactions will not make the system unsafe, a state-based design will bring out the information of interest. If the client is most concerned with the embedded feedback problem of actually controlling the speed, a process control design will show the necessary relations without extraneous detail.

Note that the problem statements do not quite capture real cruise control systems. For example, some do not fully specify how to determine the desired speed–only how to increase and decrease it, and real cruise control systems command changes to current throttle settings, not absolute settings. It is hard to determine to what extent these discrepancies are accidents of specification and to what extent they crept in to make the problem more tractable for the definition technique at hand.

6.3. Analyzability and checkability

Most of the models used here have associated analysis techniques for the aspects of the design they are intended to bring out. As noted in Section 6.2, the important aspects of the design depend on the client and the problem. The significant problem of analysis and checkability arises when multiple views or models are used in the design.

Multiple views used in many different combinations exacerbate consistency problems—especially when the views use different decompositions at the same level. The first-order problem is, of course, consistency—but the ability to make changes later is also at stake. It's particularly problematic when the different styles decompose the problem into different elements.

Some uses of different models are easy to handle. For example, when one model is used to refine a component that appears in another view, the interaction can be restricted to the interface of the component being expanded. This is the case in Shaw's design, where Figure 19 shows how the models of Figures 17 and 18 provide inputs needed in the feedback process (Figure 16). Another tractable interaction between models occurs when one model is derived from another, as Higgins derives first functional flow (Figure 15) and then data structure from the feedback entity diagram (Figure 14). A more

difficult case arises when multiple kinds of structure are imposed on a single set of base entities; this arises in Figure 20, where Ward and Keskar add event flows to a data flow diagram. Still more difficult is the situation where the problem is decomposed in essentially different ways and the semantics of the models interact. This arises, for example, when state models are added to other designs in Statemate (Figure 12) and the Ward/Mellor analysis (Figure 21). Each of these combinations can, of course, be handled as a special case of the methodology. But as Table 1 shows, the models are used in many combinations, and the task of devising special analyses for each combination is daunting.

6.4. Abstraction power

The essence of abstraction is identifying, organizing, and presenting appropriate details while suppressing details that are not currently relevant. Abstraction is often supported by design discipline, notation or analysis tools; these guide the designer in selecting details to emphasize and suppress. Disciplines do this explicitly, whereas notations and tools do it implicitly, by providing the ability to express some things, by requiring certain details, or by having no means of expressing other things.

The need to decide what's currently relevant is illustrated by the differences between the object and process-oriented designs. The object designs are concerned with the external devices that will be manipulated by the driver. They begin by enumerating objects of the real world and relations among these objects; these designs relegate control of the system to internal entities. The feedback designs, on the other hand, are concerned with the speed of the vehicle and how to control it. They begin by setting up the feedback loop and establishing the relation among the reference speed, the model of the current speed, and the engine (as controlled by the throttle); these designs deal lightly with the conversion of driver actions to the signals of interest.

Abstraction should also serve to suppress implementation decisions. Several of these designs unnecessarily reveal implementation decisions: Booch and Yin & Tanik include a global system clock and the fact that it resolve milliseconds; Kirby includes extensive information about range, resolution, and accuracy of values; Ward & Keskar include details of the definition of events at the beginning of the design. Ward & Keskar, on the other hand, do not include timing information even though they explicitly claim to be real-time.

6.5. Safety

Cruise control systems exercise largely-autonomous control over moving machines. Designers of such systems should consider explicitly whether the machines will be safe in operation. Two particular questions arise here: Can the system fully control the speed of the vehicle, and can the system's model of the world (i.e., the current speed) be sufficiently wrong to be dangerous?

Most automobile cruise controls (and all the designs here) can control the throttle but not the brake. As a result, if the car picks up excess speed (coasting down a hill, for example) the system cannot slow it down. The only designs that recognize this problem are Kirby's NRL design, Atlee & Gannon's state analysis, and Shaw's feedback loop.

Some aspect of the design process should also lead the designer to consider an even more serious problem: the possibility that the cruise control's model of current speed is radically different from the actual speed. This can happen, for example, if the drive wheels start spinning or a sensor goes bad. Brackett's problem formulation calls for a calibration capability; this addresses the problem of gradual drift but not of sudden inaccuracy. The only designs that explicitly call the designer's attention to the way current speed is modeled are Higgins' and Shaw's feedback-loop models.

6.6. Integration with vehicle

Ultimately, the designed system must not only satisfy the problem statement but also integrate with the whole automobile. Brackett's problem statement recognizes this by requiring calibration capability to deal with different tire sizes. However, this problem would not arise in a system that used relative rather than absolute speeds; such a system deals with "faster" and "slower" rather than calculated speeds. In such a system, cruise control can be independent of speedometer display.

The solution differ in the extent to which they address the relation between the user-manipulated controls and the data of the software design. The object-oriented architectures focus on this aspect. The

state-based architectures define this as outside the scope of their problem. The feedback architectures focus on the control question first but include interpretation of user inputs as separate stages of the design. The real-time architectures show the relation as system context.

Most of these designs simply compute a throttle value. Only Shaw and Ward & Keskar consider the rate at which speed should be increased or the response characteristics of the system. Higgins' design has an explicit entity (Set Throttle Pressure Summing Point) whose elaboration might reasonably address this question.

7. Conclusions

Engineering involves making choices, especially design choices. Although we're not very articulate about it, we have a variety of architectural styles for organizing systems. Examining a variety of solutions for a single problems shows some of the comparative advantages of the architectures and some of the remaining problems in making systematic architectural design practical.

Architectural style strongly influences the resulting design but does not fully determine result. It is not surprising that different designers using the same approach get different results, as all design methods allow considerable room for individual judgment. We do, however, see systematic differences in the kinds of questions designers are led to ask by different architectures.

Notwithstanding the label associated with an architectural style, the designer usually develops several models. Sometimes, but not always, the relation among views is well-defined. These examples call attention to the need for systematic means of establishing the relations between multiple views of a system.

Acknowledgments

My interest in cruise control problems began in a reading group on software architecture that I organize with David Garlan. Via Marc Graham's efforts, Will Tracz presented an example that led me to look at cruise control. Suzanne Bell's INSPEC search yielded a rich harvest of papers. Discussions with Bill Wulf, Roy Weil, Jeannette Wing, Gene Rollins, David Garlan and colleagues at Fisher Controls have improved the my understanding of the designs.

This research was supported by the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is sponsored by the U.S. Department of Defense), by a grant from Siemens Corporate Research, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. 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 Wright Laboratory, the Department of Defense, the United States Government, Siemens Corporation, or Carnegie Mellon University.

References

[Atlee&Gannon 93] 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.

[Birchenough&Cameron 89] 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.

[Booch 86] Grady Booch. Object-Oriented Development. IEEE Trans. on Software Engineering SE-12, 2, February 1986, pp. 211-221.

[Brackett 87] John Brackett. Automobile Cruise Control and Monitoring System Example. Wang Institute of Graduate Studies, TR 87-06, 1987. The archives show no trace of this paper, but [Kirby 87] repeats the problem definition.

[Garlan&Shaw 93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora (eds), Advances in Software Engineering and Knowledge Engineering, vol. 2, World Scientific Publishing Company, 1993.

[Gomaa 89] Hassan Gomaa. Structuring Criteria for Real Time System Design. *Proc 11th International Conference on Software Engineering*, 1989, pp. 290-301.

[Gomaa 93] Hassan Gomaa. *Software Design Methods for Concurrent and Real-time Systems.* Addison-Wesley 1993.

[Higgins 87] David A. Higgins. Specifying Real-Time/Embedded Systems using Feedback/Control Models. *Proc SMC XII: Twelfth Structured Methods Conference*, 1987, pp. 127-147.

[Jones 90] 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.

[Kirby 87] J. Kirby. *Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System*. Wang Institute of Graduate Studies. TR 87-07, 1987.

[Shaw 94] Mary Shaw. *Beyond Objects: A Software Design Paradigm Based on Process Control.* Carnegie Mellon University Computer Science Department Technical Report CMU-CS-94-154.

[Shaw et al 94] Mary Shaw, Robert Allen, David Garlan, Dan Klein, John Ockerbloom, and Curtis Scott. *Candidate Model Problems in Software Architecture*. Version 1.1, unpublished manuscript.

[Smith&Gerhart 88] Sharon L. Smith and Susan L. Gerhart. STATEMATE and Cruise Control: A Case^2 Study. Proc. COMPSAC88: *Twelfth Annual International Computer Software and Applications Conference*, 1988, pp. 49-56.

[Wang&Tanik 89] 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.

[Ward 84] P. Ward. Class exercise used at the Rocky Mountain Institute for Software Engineering, Aspen CO, 1984.

[Ward&Keskar 87] 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.

[Wasserman 89] Pircher, Robert J. Muller. An Object-Oriented Structured Design Method for Code Generation. *ACM Software Engineering Notes* vol 14, no 1, Jan 1989, pp. 32-55.

[Wing 88] Jeannette Wing. A Study of 12 Specifications of the Library Problem. *IEEE Software*, July 1988, pp. 66-76.

[Yin&Tanik 91] 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.