# Understanding Macroscopic Behaviour Patterns in Object-Oriented Frameworks, with Use Case Maps

R. J. A. Buhr
*Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada*
*email: buhr@sce.carleton.ca*

**Abstract**

*Object-oriented frameworks provide a powerful construction technique for software, but understanding the behaviour of applications constructed from them is well known to be difficult, for a number of reasons. 1) The first class representation of the framework in code centers around construction; system structure and behaviour are only visible in a second class way, in widely scattered details. 2) The system view we seek is not a concrete thing, but an abstraction. 3) The use of design patterns, a common practice in frameworks, may obscure the system view by making inter-component interactions more indirect than they would otherwise be. 4) The use of combinations of design patterns obscures the system view because pattern combination is not yet a well defined subject. 5) Both the system view we seek and design patterns are abstractions, so gaining system understanding requires combining multiple abstractions, for frameworks that make serious use of design patterns. 6) Although framework class organizations are fixed, systems constructed from framework are self modifying while they are running, and self modifying software is notoriously difficult to understand. 7) Specific applications are visible only in framework customization details. This paper illustrates by example how a general technique called "use case maps"—invented by the author for describing complex systems of any kind—can help to alleviate these difficulties. To indicate the range of applicability, the examples are drawn from frameworks in very different application domains: HotDraw for GUIs and ACE for communications. The objective is to raise the visibility in the framework community of the UCM approach to alleviating the above problems, not to "prove" UCMs solve them or to provide a cookbook for solving them.*

## 1. Introduction

This paper aims contributes to methods to help humans understand and describe macroscopic behaviour patterns[1] in software systems constructed from object-oriented frameworks. The behaviour patterns we seek to understand are *system* properties exhibited by the running software. A system is a set of components that collaborates to achieve some overall purpose. Examples of components of software systems are objects, threads, processes, groups of these often called modules, and groups of modules often called subsystems. Our interest is in the runtime behaviour of systems of such components constructed from object-oriented frameworks.

The difficulty of understanding macroscopic behaviour patterns exists to some degree for all software, but is particularly troublesome for software constructed from object-oriented frameworks, for the following reasons:

- 1) *The first class representation of the framework in code centers around component construction; system structure and behaviour is only visible in a second class way, in widely scattered details.* The system structure (how components are organized into collaborating groups) may look quite different from the class organization that is explicit in textual object-oriented programming languages such as such as Smalltalk, C++, or Java. For example, multiple instances of a single class may be scattered throughout an executing program, playing different roles in different places in the runtime program, or a an instance of a single class may play different roles at different times in different places in the runtime program. Understanding the system requires understanding how the instances come to play these roles over time, and how the roles are played in the context in which the instances find themselves at a particular time. In other words it requires an understanding of the system structure and how it evolves over time. However, source code written in textual object-oriented programming languages does not describe the system structure and behaviour in a first class way, but leaves them to emerge from widely scattered details in the code.
- 2) *The system view we seek is not a concrete thing, but an abstraction.* Because the system structure and behaviour are not explicit in a first class way in the source code of textual object-oriented programming languages, any view of it is an abstraction relative to code. Object-oriented frameworks implemented with textual object-oriented programming languages do not include such abstractions explicitly. So there is extra work required to understand systems constructed from such frameworks beyond that required to understand the framework itself.

1. The term "pattern" is used in this paper in two ways: a narrow technical one, as in "design pattern", and a broader one that reflects common English usage, as in "behaviour pattern", or "visual pattern". The two meanings may sometimes converge. This overloading of the term is unfortunate, but seems unavoidable when a term with its own useful meaning in common usage is coopted for a narrower technical purpose.

- 3) *The use of design patterns, a common practice in frameworks, may obscure the system view by making inter-component interactions more indirect than they would otherwise be.* This is not a criticism of design patterns, but simply a neutral observation of a fact. The frameworks we shall study in this paper, namely HotDraw [6][8] and ACE [9], embody a number of design patterns that do this [13][17][18][19][20][21][22]. On one hand, design patterns help *program* understanding by enabling humans to assume that details follow standard patterns. On the other hand, design patterns that introduce extra levels of indirection into interactions make the *system* view that is already buried in detail (point 1) even harder to understand. Experts will form intuitive understandings but non-experts approaching a framework cold will have difficulty. For example, we found it quite difficult to abstract the system view from the code and documentation of the frameworks used as examples later in this paper.
- 4) *The use of combinations of design patterns obscures the system view because pattern combination is not yet a well defined subject.* Even relatively small scale frameworks, such as the ones uses as examples later in this paper, use combinations of design patterns. Design patterns are typically described as independent entities, leaving it to the user to understand how to combine them. Pattern combination is not yet a well developed subject, so combinations must be understood informally. Experts will develop an intuitive understanding but non-experts must construct an understanding by painstakingly combining independent descriptions. Even very smart, experienced people may find the problem difficult when approaching a new framework embodying design patterns with which they are not familiar.
- 5) *Both the system view we seek and design patterns are abstractions, so understanding requires combining multiple abstractions, for frameworks that make serious use of design patterns.* In general, abstraction are beneficial for understanding. However, practical frameworks leave it to the user to develop an understanding of this combination of abstractions. This is difficult for the following reasons. First, the system view is not explicit (points 1 and 2) and extracting it can be difficult (point 3). Second, combining design patterns is itself difficult (point 4). Thus difficulty is piled upon difficulty. When the description technique for the abstractions is one such as UML[2] or ROOM [23], we may be overwhelmed with detail for large, complex systems. Used for system description, techniques such as UML or ROOM are very detail-rich. This is because they center around specifying *how* behaviour will be made to emerge at run time (through the interchange of messages between components, driven by internal state transitions of components). Even when we stand back from *how* to use these techniques to describe *what* (in terms of example scenarios expressed as message interchange sequences), the result for large, complex systems will be many diagrams containing much detail. Experts can amortize the work of understanding this detail over many applications of the same framework, but even experts have need of more compact description techniques.
- 6) *Although framework class organizations are fixed, systems constructed from framework are self modifying while they are running, and self modifying software is notoriously difficult to understand.* In self modifying systems, the system structure evolves over time due to actions of its components; in other words, the system modifies itself. This increases the difficulty of understanding macroscopic system behaviour. Descriptions must include not only intercomponent messages for ordinary behaviour but also messages to change the system structure, namely to create new components, destroy old ones, make components visible in new places and invisible in old ones, and move mobile components from place to place. Understanding macroscopic behaviour patterns in systems with fixed structure is difficult enough in terms of detail-rich descriptions such as the ones mentioned in point 5. Adding self modification excacerbates the problem.
- 7) *Specific applications are visible only in framework customization details.* Frameworks are for constructing a range of similar applications. This should mean that a basic system view exists that is common to all applications, only requiring embellishment for particular applications. However, we have seen that, in practical frameworks, no such view is directly available, but must be inferred by the user from code and multiple detail-rich design descriptions. Experts will understand it intuitively, but others may find it difficult.

### 1.1 Use Case Maps

To alleviate the understanding problem, we offer use case maps (UCMs) [3][4][5][6][7][8][9][10][11]. UCMs are characterized in [4] as a simple, visual notation for understanding and architecting the emergent behaviour of large, complex, self modifying systems. UCMs are said to be for *architecting behaviour* to emphasize that they give *form* to macroscopic behaviour patterns without specifying how the actual behaviour will emerge at run time. In common usage, the term *architecting* implies giving form to macroscopic structure. Here the term is extended to included macroscopic behaviour in a way that is consistent with this usage. This paper focuses on only on the *understanding* purpose of UCMs. There are two reasons for this: 1) The examples are from existing frameworks developed without the aid of UCMs. 2) There are as yet no public-domain examples of frameworks developed with the aid of UCMs (the architecting purpose of UCMs has been exercised in industry in at least one framework development project, but this work has not been written up in the public domain). However, the *architecting* purpose of UCMs should be kept in mind.

Explanation of UCMs is left to the examples of Section 2 and Section 3. See Section 4 for further discussion of the properties and uses of UCMs. Features of UCMs for dealing with layering and decomposition of large scale systems are mostly only mentioned in passing in this paper because the examples are relatively small scale (although this does

not mean they are simple); see [4] for a thorough treatment of these features.

The examples are not large scale enough to exhibit in any strong way the problem of detail-rich conventional descriptions swamping the reader in details. A reader can gain insight into this problem from the examples only by exercising some imagination. When looking at the non-UCM diagrams presented as examples here, imagine scaling up the system such that inch-thick stacks of such diagrams are required to describe the system. Then it is not difficult to imagine how UCMs can be useful as a way of condensing the essence of the system into many fewer diagrams.

### 1.2 Use Case Maps and Frameworks

The paper will lead us, step-by-step, to the conceptual model of frameworks indicated by Figure 1, in which UCMs provide a so-called *system* view, high level class relationships (methodless and messageless) provide a so-called *construction* view, and all else is pushed down to the next lower level of abstraction, to be described by techniques such as those offered by UML and ROOM.

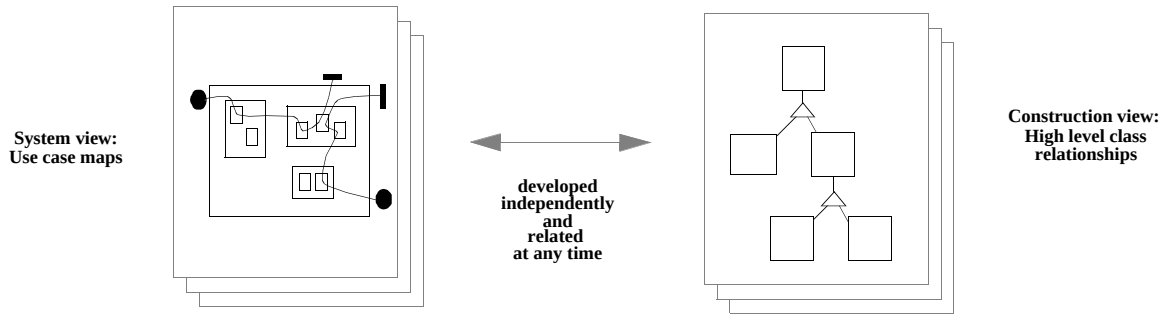



**Figure 1: High level view of frameworks**

The words “system” and “construction” are easily overloaded with different meanings and some readers may find that the meanings intended by Figure 1 conflict with their favourite ones. Finding precisely the right terms to describe these ideas is difficult, so we ask such readers to try to understand the meanings intended here, even if they disagree with the terms. The term *system view* means a description of a collaborating set of components as seen at run time. The term *construction view* means a set of specifications from which system components may be constructed at run time. These two views overlap in source code, but are distinct at this level of abstraction, as follows. On one hand, the system view provided by UCMs includes self modification of system structure, so this is not part of the construction view. On the other hand, in programs, the code that implements the classes of the construction view provides not only the details that produce emergent system behaviour at run time, but also the constructors and destructors needed to achieve system self modification. These aspects are separated at this level of abstraction as follows. The details that produce emergent behaviour are deferred to a *lower level of abstraction*. The constructors and destructors implicitly form part of a *layer* underneath UCMs that provides a component factory[1]. Special UCM notations *imply* usage of this factory, without showing explicitly either its existence or how it will be used. The result is, at this level of abstraction, that only the system view describes runtime structure and behaviour, including self modifying behaviour.

A key property of this way of looking at things is the distinction it makes between the *components* of the system view and the *classes* of the construction view. From a system perspective, at this level of abstraction, classes are specifications that may be used to construct operational components, they are not operational components themselves. Keeping the system and construction views distinct helps to keep design concerns distinct. This is important, because there are often tradeoffs to be made between properties of systems, such as end-to-end response time along paths, and construction properties, such as extensibility.

The general form of this model is relatively independent of whether a framework is white box or black box. In either case, some concrete class in the construction view provides the starting point for constructing a system component. From this perspective a black box framework is simply one that offers specifications for system components in more consolidated form.

This model provides a dramatic reduction in the number of diagrams required to understand the large scale organization and behaviour of systems constructed from object-oriented frameworks.

This model can also help to provide traceability down to details and up to requirements, where *down* and *up* indicate directions of decreasing and increasing abstraction. Upward traceability can be from UCMs to use cases. Downward traceability can be from UCMs to the use of design patterns to implement them (at the level of [13][19][20][21][22]).

Developing this model for existing frameworks is simple on the construction side, because the model is repre-

1. Note the different implied meanings of the terms *level of abstraction* and *layer*: The former pertains to description techniques, the latter to the large scale organization of systems (see [4] for more on how UCMs deal with the latter subject).

sented in a first class fashion in the code. However, on the system side, we need to find our way through widely scattered details and abstract from them. How to do this, as well as the nature of the results, is the subject of the rest of this paper.

## 2. HotDraw

Although this section draws on a case study of HotDraw reported in [6], the design model presented here is actually of an idealized version of Hotdraw that was developed in [6] as part of a reengineering exercise. The presentation here also comes at the model from a very different angle than previous presentations in [6][8], which presented use case maps first. Here we approach the framework first from a conventional perspective, with the objective of illustrating the complexity that can result, and then show the simplification that use case maps produce.

### 2.1 Classes

From a code perspective, HotDraw is a set of Smalltalk classes that extends the standard Smalltalk class library to support a specific style of GUI interface. The nature of the HotDraw class hierarchy is shown in Figure 2, leaving out methods and leaving out the classes of the standard Smalltalk class library used by HotDraw. In this diagram (as elsewhere in this paper), we avoid elaborate class relationship notations, simply to avoid clutter; such notations are useful for detailed design, but we do not need them for the purposes of this paper (labelled arrows suffice). In this particular diagram, boxes with thick lines identify classes used as examples in Figure 4.

Some classes have inheritance relationships between them (shown as *isa* arrows). There may be many runtime relationships among the classes, of which this diagram only shows one example (a *contains* relationship). The runtime relationships are hidden in the details of the classes. In particular, behaviour is generated through message-sending relationships not shown in this diagram. We could try to add more information about runtime relationships to this diagram to try to get insight into behaviour, but such insight is better obtained by going to a more system-oriented view. From a system perspective the classes are specifications that may be used to construct system components, they are not system components themselves.

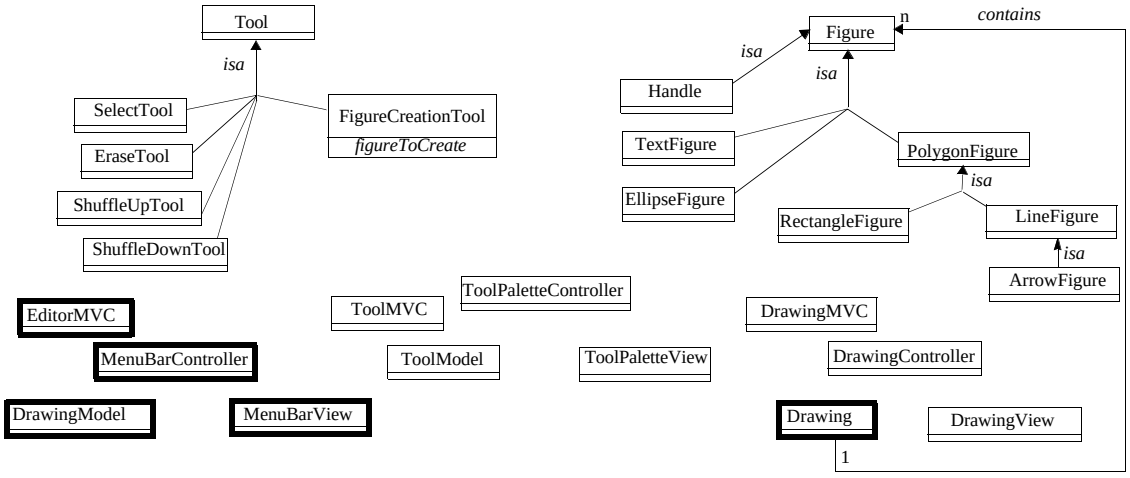



**Figure 2: Nature of the HotDraw Class Hierarchy**

### 2.2 System Components

From a system perspective, the essence of HotDraw is a set of MVC components to control the different areas of the screen (Figure 3). MVC stands for Model-View-Controller, a well known system structure in which the M part (the model) maintains data, the V part (the view) display data, and the C part (the controller) manage interactions with the

human user for different views.

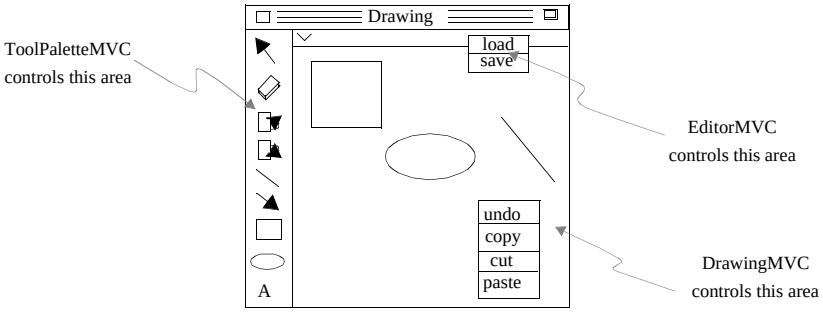



**Figure 3: MVC components control HotDraw screens**

In the multi-MVC system, each MVC group is a system component that contains nested M, V, and C components, which in turn may contain other nested components. Figure 4 shows, on the left, one of HotDraw's MVC components (the Editor MVC) and, on the right, the relationships of its subcomponents to the classes that were highlighted in Figure 2. In this diagram, ME, VE, and CE identify the Model, View, and Controller components, respectively, of the Editor MVC (later, MT, VT, and CT and MD, VD, and CD will identify elements of the ToolPalette and Drawing MVCs, respectively). Such MVC components are peers at the system level, the M, V, and C components are peers within each MVC component, and so on. The diagram illustrates the style of showing system structures in use case maps: sets of peer and nested components, without identification of interface elements, are arranged in fixed relative positions so that the arrangement can be recognized as a visual pattern.

This diagram uses a neutral component notation that is not entirely adequate for the purposes of use case maps, but that will do for now—the use case map of Figure 11 uses a more general notation that will be explained when we come to that diagram.

Showing the cross references from classes to system components as arrows does not constitute a recommendation to document such cross-references this way (a textual list of name cross references would suffice for that); what we are striving for here is a diagram to anchor in the mind's eye the difference between system components and classes.

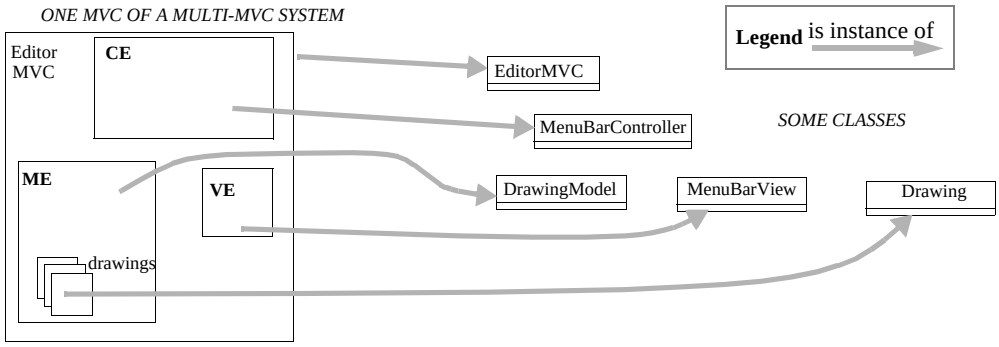



**Figure 4: System components and classes**

This leads us to Figure 5, showing a so-called *component context diagram* of the entire MVC system. The system is self-modifying because some of these components come into existence only when needed and different ones may be needed at different times. It is the behaviour of this set of components as system in which we are interested. However, although Figure 5 shows a system, seeing it as an artifact is difficult because its structure is not explicit in the only declared structures in the code—class hierarchies—and its behaviour is not observable directly with our own eyes, but only through interactions with physical devices such as a screen, a keyboard, and a mouse. The physical devices are part of the system as whole, but seeing how the software affects them does not help us to understand how the software works internally as a system. We want to be able to see behaviour patterns in such systems at a high level of abstraction, but this is difficult by conventional means without becoming swamped in details, as will now be

shown.

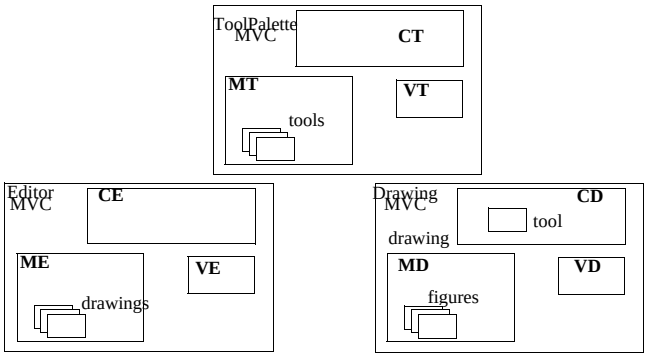



**Figure 5: Component context diagram of the MVC system**

### 2.3 Design Patterns and Messages

We shall now study the message sequences required for one particular example of behaviour: the selection of drawing in the Editor MVC to be displayed by the Drawing MVC, at a time when no drawing is currently displayed.

This example contains two interesting features:

- It uses a design pattern, the observer pattern from [13], to arrange the sending of the required messages (actually the Smalltalk dependency mechanism is used—this is a design pattern, built into the standard Smalltalk class hierarchy, that provided a model for the observer pattern). Use of this pattern makes intercomponent message sequences hard to trace in the code, because they are indirect.
- It illustrates system self-modification through the requirement for a selected drawing containing a new set of figures to move from being data in the Editor MVC to being executable code in the Drawing MVC. When no drawing is currently displayed, there is no drawing component in this position beforehand, so the effect is to put a new component in this position (Figure 6). The fact that only an identifier moves, not the whole component, is an implementation detail. The system effect is analogous to a Java applet moving from being data in one web site to being executable code in another.

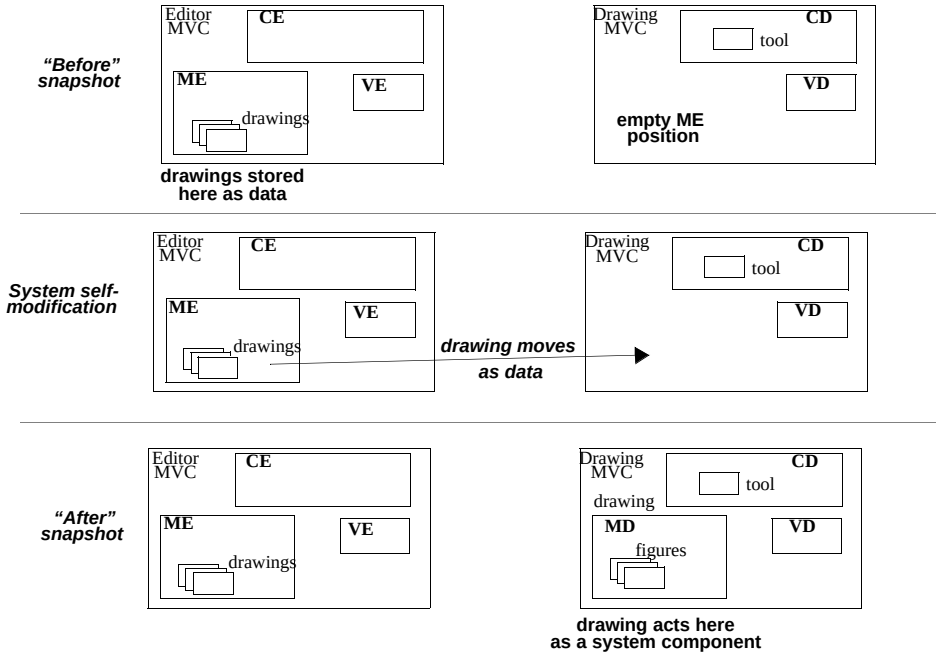



**Figure 6: MVC system self-modification**

Figure 7 displays the message sequences in HotDraw needed to accomplish the purpose of Figure 6 (the observer pattern has more to it than message sequences, but those are all we are interested in for purposes of understanding behaviour). The message sequences in Figure 7(a) are for ME as Subject1, and the ones in Figure 7(b) are for MD as Subject2. Notice that VD is an observer of both subjects and that there can be no Subject2 (no MD) until the message sequence of Figure 7(a) is completed. Figure 6's movement of the drawing as data from the Editor MVC to become

the model MD in the Drawing MVC is accomplished at the end of the message sequence of Figure 7(a), upon VD's return from the getData() message to ME. This enables VD to register itself as an observer of MD (at the top of Figure 7(b)) and thereafter to receive notifications of changes to drawing data. At the code level, we see changing visibility (of a drawing object by VD); at the system level we see this a component movement (of the drawing object into the MD slot).

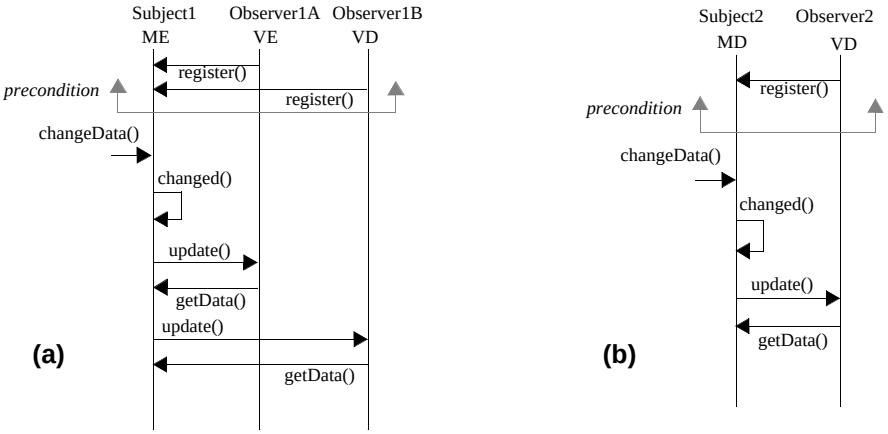



**Figure 7: Message sequences in and between the Editor and Drawing MVCs**

This is a small scale example, but complexity blows up when there are many components and many messages between them. Complexity factors are as follows: One is the sheer number of messages, and the detailed commitments they require, for a system of any size or complexity. Another is the difficulty of keeping the message sequences and the system structure associated in the mind's eye when there are many diagrams of message sequences. Another is the hiding of collaborations behind indirections resulting from the use of design patterns (for example, the changed-update sequences). Another is the lack of differentiation between messages that cause system self-modification (such as getData() at the end of the sequence in Figure 7(a)) from other types of messages, making the existence of self-modification very difficult to separate out as a system issue.

The result is that system meanings are buried under detail. Humans must mentally compose the pieces of the puzzle. In large systems, many diagrams may be required. If the details are known (as with existing frameworks), we get lost in them. If the details are not known (as is the case during forward engineering), there is nothing between requirements prose and the details to help discover the details.

### 2.4 Use Case Maps

Use case maps collapse message sequences into cause effect sequences (Figure 8).

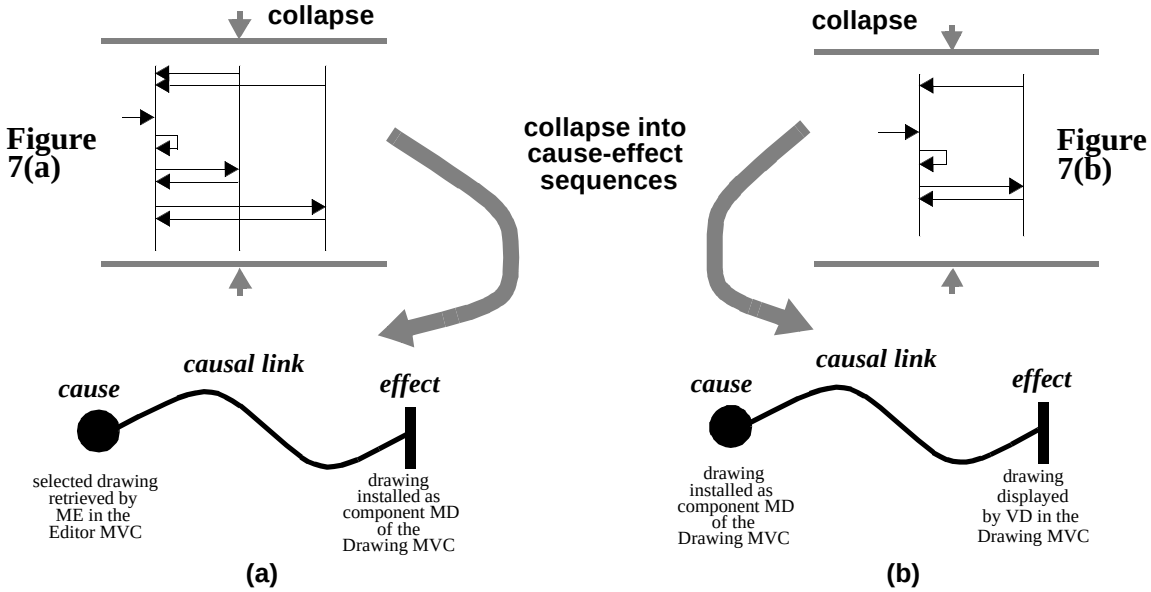



**Figure 8: Collapsing message sequences into cause-effect sequences**

In Figure 8, a cause-effect sequence is represented by a filled circle at the start, a bar at the end, and a path of any shape in between. This notation hides all the complexity factors associated with messages, enabling one to get a higher level perspective. All that remains is a causal link, without any indication of how it is to be implemented. Cause-effect sequences are at a level of abstraction above messages, message directions, multiple back-and-forth messages, and message interchanges required to establish preconditions.

When we want to see the big picture, we simply concatenate cause-effect sequences to form paths (Figure 9), along which *responsibilities* identify intermediate effect/cause points. Result: use case maps.

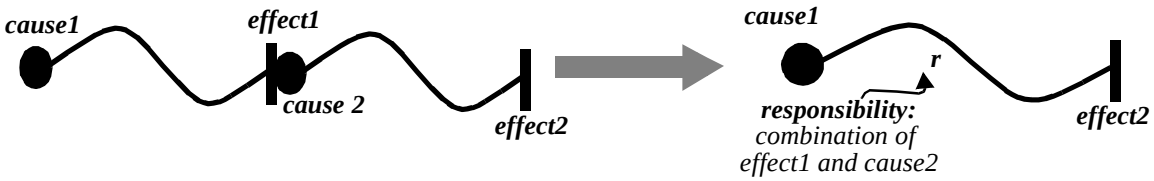



**Figure 9: Use case maps are formed by concatenating cause-effect sequences**

The term *use case map* has its origins in the term *use case* [14]. A *use case* is a prose description of system behaviour that may embody a set of related scenarios of system operation. The word *use* does not necessarily imply human users or users outside the system. In general, a use case map may embody scenarios involving many systems, subsystems, or components that are users of each other. In relation to use case maps, scenarios are just the progression of cause-effect sequences along paths (in general, many such sequences may be in progress concurrently in a map at any time, or even along a single path, constrained only by preconditions). A use case map presents this view as a requirement or explanation; there is no implication that the causal sequence has a first class representation in the software.

Figure 10 (a segment of a path in the use case map for Hotdraw shown in Figure 11) consolidates the ideas in Figure 8-Figure 9 and also introduces some new notation.

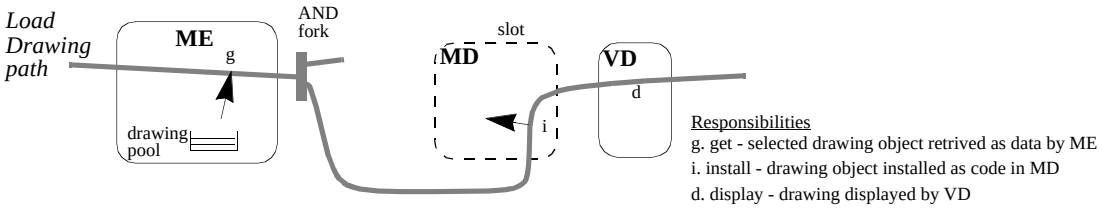



**Figure 10: A fragment of a HotDraw use case map**

The component notation illustrated by Figure 10 is generalized from the neutral one of Figure 5 as follows. It identifies primitive *objects* as boxes with rounded corners. Boxes with sharp corners are known as *teams*. These are concepts for systems, not programs, indicating that, *at the scale of the diagram*, teams are operational groupings (shown nested), and objects are primitive data or procedure abstractions. Here, all are instances of program classes but, in general, they do not all have to be. The generalized component notation also explicitly indicates the dual nature of dynamic components of self modifying systems, as data (in pools, shown by a special symbol) or code (in slots, shown as ordinary boxes with dashed outlines).

Figure 10 has three responsibilities along the path segment, named *get*, *install*, and *display*; these have been abstracted from Figure 8, using the technique of Figure 9. The convention illustrated here of showing shorthand labels along the paths, and listing the actual responsibilities to the side, is a standard one for reducing visual clutter in use case maps.

The path notation in Figure 10 uses small *move* arrows between pools or slots, and paths, to indicate component movement: the arrow at *get* indicates movement of a drawing object, as data, out of a pool in ME; the arrow at *install* indicates movement of the same object into the MD slot to become an operational object there. *Get* and *install* are responsibilities shown with a special notation to highlight system self-modification. The third responsibility along the path, *display*, means that VD displays the new drawing on the screen.

The map notation also provides for path forks and joins. The fork with the bar after ME is called an *AND fork* and means that the paths are, in principle, concurrent after the fork (all this means for HotDraw is that we don't care about the relative order of sequences along the two paths). A mirror-image notation for *AND-joins* is available but not needed by this example. All other forks and joins in maps, such as ones in Figure 11 and Figure 18, are *OR-forks* or *OR-joins* that result from path superposition (different paths are combined into one map and some paths follow the same route, without any implication of synchronization between them).

Observe how much detail this map fragment omits relative to message sequence diagrams like Figure 7, while at the same time giving the essence of behaviour, including both ordinary behaviour and self-modification. The complicated business of making the drawing object visible to VD through the multiple registration, notification, and retrieval messages associated with the observer pattern is hidden, but the effect is expressed directly in a simple and

easy to understand fashion. We can use diagrams like this as traceability tools for identifying where design patterns such the observer pattern are used at a more detailed level, by simply providing traceability pointers in the map documentation.

Figure 11 presents a use case map for three connected use cases for the multi-MVC system of HotDraw: 1. Load Drawing, 2. Select Tool, and 3. Select Figure. These paths could be concatenated to show one large causal sequence, but this is not necessary to get a good system view. All we need to know to understand the larger 1-2-3 sequence is that a precondition of 2 is completion of 1 and a precondition of 3 is completion of 2 (preconditions and postconditions are part of the textual documentation of use case maps). Responsibilities are omitted in this figure because we do not need them to understand its general nature (however, they are implied).

Let us now trace the *LoadDrawing* path in Figure 11. At CE, a user interaction would be decoded as a request to load a drawing. At ME, the requested drawing would be moved from a pool. At VE, the drawing name would be displayed in the editor area of the screen. At MD, the drawing would enter the slot. At VD, the figures of the drawing would be displayed in the drawing area of the screen. This should be enough information to understand the general nature of the map as a whole.

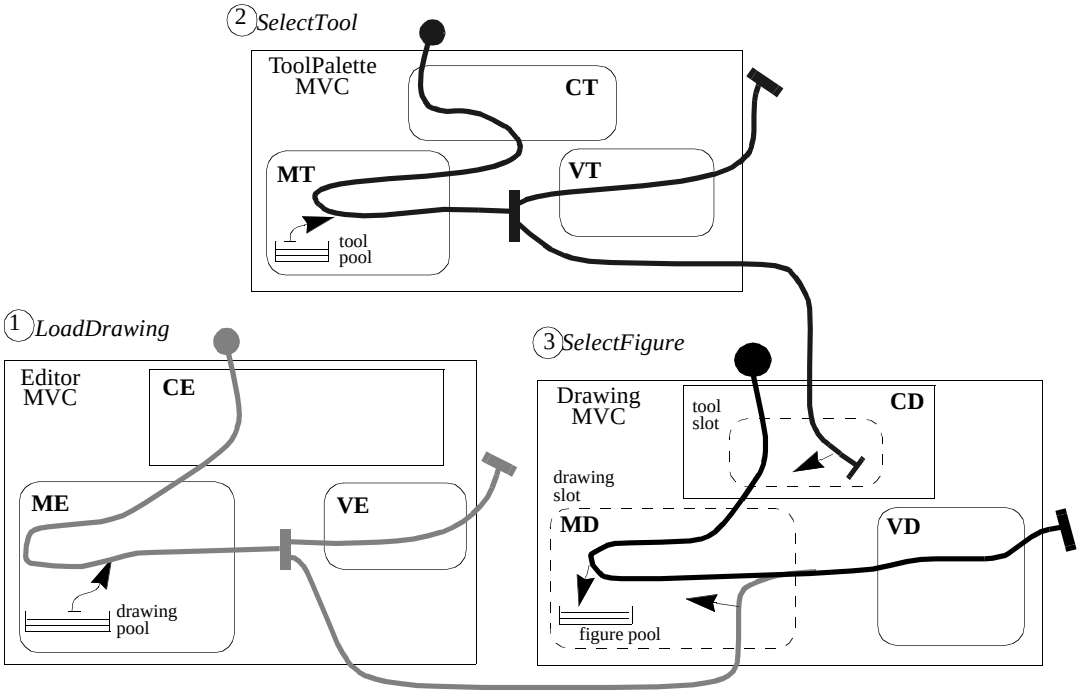



**Figure 11: A use case map for an idealized version of HotDraw**

The use case map of Figure 11 has an elegant regularity about it. The version of HotDraw we studied had more irregular paths. However, this irregularity was only easy to see after the paths had been drawn. When we saw it, we found we could make the paths regular with relatively minor coding changes. It seemed that the irregularity was due to shortcuts taken by coders. The changes made the code immediately more understandable. Regularity seems a desirable property, because it aids understanding, reuse, maintenance, and evolution, but its presence or absence is difficult to see when immersed in details. This suggests that use case maps are useful, not just for understanding frameworks, but also for designing them.

Not only do use case maps eliminate the complexity problems of understanding systems in terms of messages, they also have other advantages, illustrated by Figure 11: They enable us to show behaviour patterns as recognizable visual patterns superimposed on structure, in a way that allows us to combine many behaviour patterns into one diagram. They represent behaviour patterns that could equally well be regarded as design patterns of a rather high level kind (this viewpoint is presented in [8]).

## 3. ACE

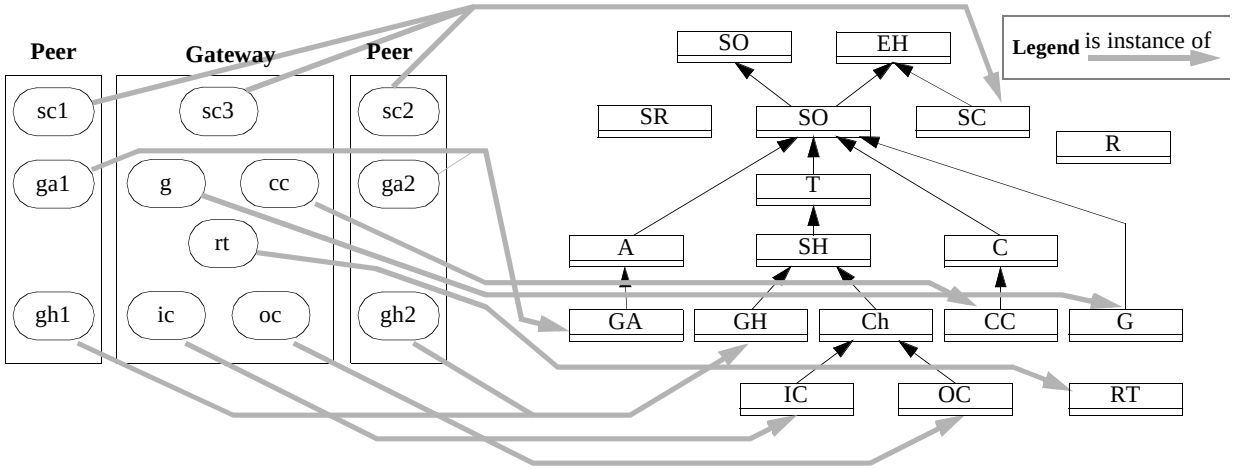
ACE is a public-domain C++ framework for communications applications. It comes with some canned applications, one of which, called the gateway, we shall use as an example here. The gateway application enables peer workstations in a network to interact through a gateway workstation. Some of the peers act as input peers that generate messages and others act as output peers that consume messages. Messages are sent from any one input peer to the gateway

workstation, which then routes the messages to one or more output peers. (The term “message” means two things in this section, depending on context. Here it means textual data sent between workstations. Later it will also refer to intercomponent interactions in the gateway’s system of software components. The meaning will be clear from the context.)

ACE manifests the same complexity factors as HotDraw, but in a more complex application domain, making it even harder to understand. Therefore, in the limited space here, we can only give a few highlights.

### 3.1 Classes and System Components

Figure 12 gives an overview, in condensed form, of the main system components and classes in the gateway application, and the relationships between them.




System Components

| Tag | Instance Name |
|---|---|
| sc1, sc2, sc3 | aService_Config |
| ga1, ga2 | aGateway_Acceptor |
| g | aGateway |
| cc | aChannel_Connector |
| rt | aRouting_Table |
| gh1, gh2 | aGateway_Handler |
| ic | anInput_Channel |
| oc | anOutput_Channel |

Classes

| Tag | Class Name | Tag | Class Name |
|---|---|---|---|
| SO | Shared_Object | GA | Gateway_Acceptor |
| EH | Event_Handler | GH | Gateway_Handler |
| SR | Service_Repository | Ch | Channel |
| SO | Service_Object | CC | Channel_Connector |
| SC | Service_Config | G | Gateway |
| T | Task | IC | Input_Channel |
| A | Acceptor | OC | Output_Channel |
| SH | Svc_Handler | RT | Routing_Table |
| C | Connector | R | Reactor |

**Figure 12: Gateway system components and classes**

### 3.2 Design Patterns and Messages

The main ACE patterns we shall be concerned with here are the reactor, acceptor and connector. Another important ACE pattern, the service configurator, must be left out for lack of space. It supports creation and management of service objects. It is used in both the gateway and peers to provide a generic initialization mechanism. We show only its effects in the use case maps.

The reactor pattern (Figure 13) is used extensively in both gateway and peers as the underlying communication mechanism driving them. It supports event demultiplexing by providing a means of registering event handlers so that when an event occurs the correct event handler is notified. Events include communications events, timers, and UNIX signals. Our use case maps do not make explicit reference to this pattern, but assume is supported by an underlying

layer.

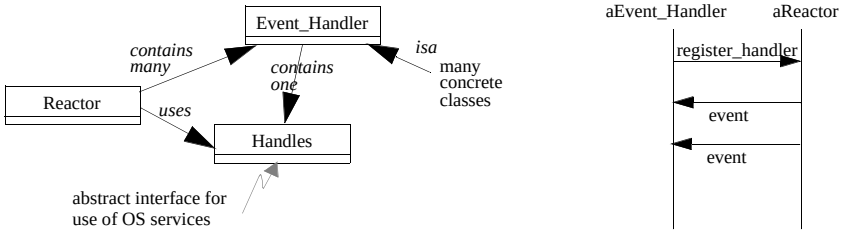



**Figure 13: Some elements of the reactor pattern**

The connector pattern (Figure 14) and the acceptor pattern (the partner of the connector, not illustrated because the nature of both patterns is sufficiently well illustrated by one of them) are used to establish interworkstation soft connections (for example, Unix sockets). The connector pattern is used by the gateway and the acceptor pattern by the peers. The connector pattern supports the making of connect requests to particular network addresses. The acceptor pattern supports listening for connect requests on a particular network address. Figure 14 shows the connector pattern integrated into ACE, not in isolation.

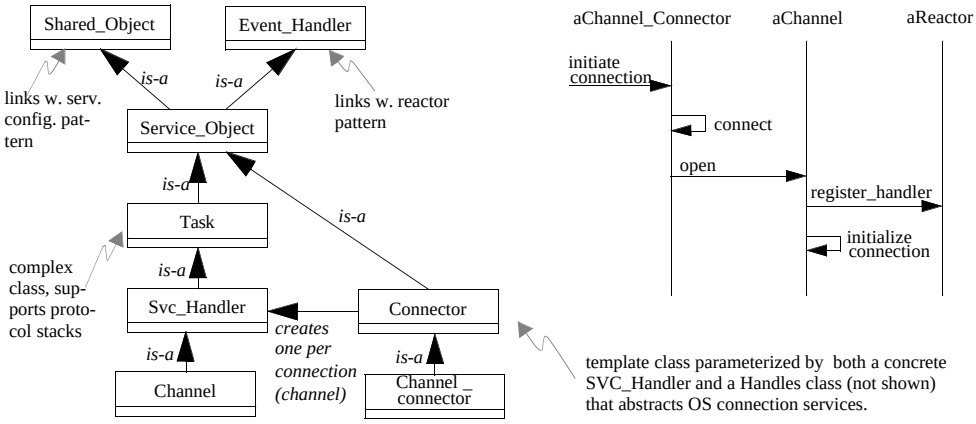



**Figure 14: An overview of the connector pattern in ACE**

Figure 15 shows some message scenarios that occur in the gateway node in relation to establishing a connection. These will not be explained further here; interested readers will no doubt be able to figure them out, but the point of presenting them here is to give a sense of the kind of detail that is required to describe behaviour at the message level. A person has to hold the class and system pictures in the mind's eye while studying different message sequences, and to relate the different diagrams to each other by mentally associating textual names of elements in them. This is a relatively small scale system and these sequences show only part of its operation (for example, they do not show the self modification that occurs during a configuration phase). Therefore the amount of detail in this figure is not yet overwhelming. However it is not hard to visualize how the amount of detail in sets of such figures can easily become over-

whelming for large scale systems.

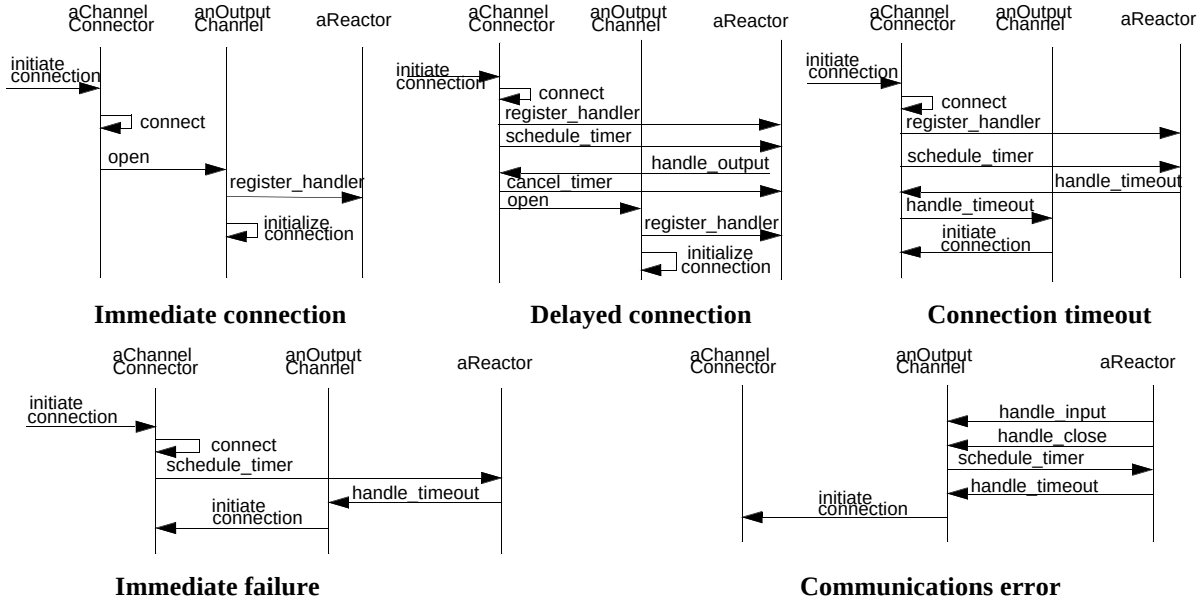


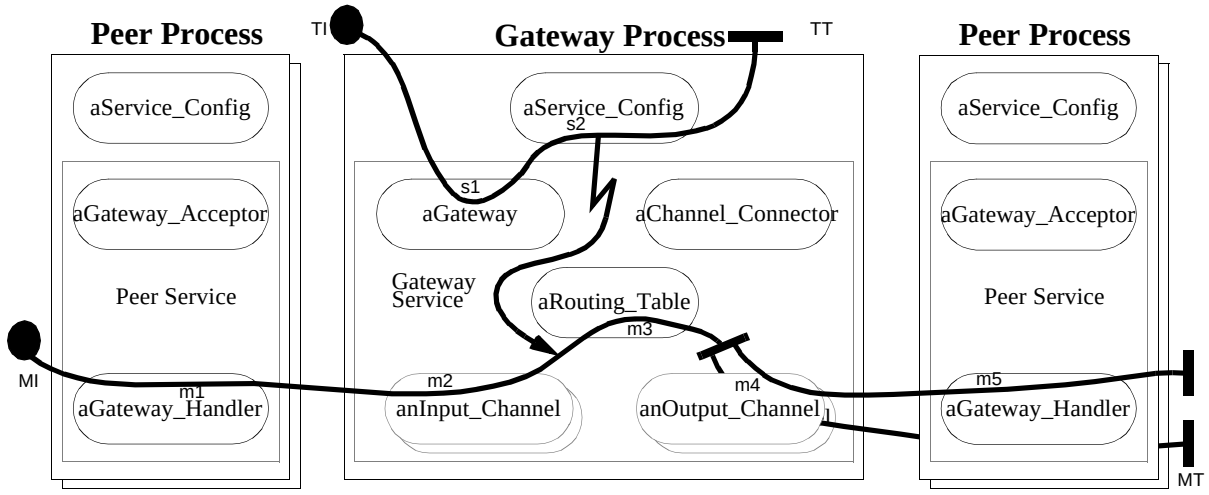

**Figure 15: Some message sequences**

### 3.3 Use Case Maps

The way in which message sequences are collapsed into causal sequences in use case maps was explained in the Hot-Draw example, so here we go directly to use case maps.

Figure 16 gives an overview of the normal operation of the gateway system after connections have been established, including termination behaviour. The slots are populated during initialization and channel connection (see Figure 17 and Figure 18). The lightning strike symbol means abort all activity in progress along all paths to which it points. This figure also illustrates a recommended style of documenting preconditions, postconditions, and responsi-

bilities.




**Preconditions:**

MI: A message is available on the standard input device
TI: A SIGQUIT or SIGTERM signal has been issued

**Postconditions:**

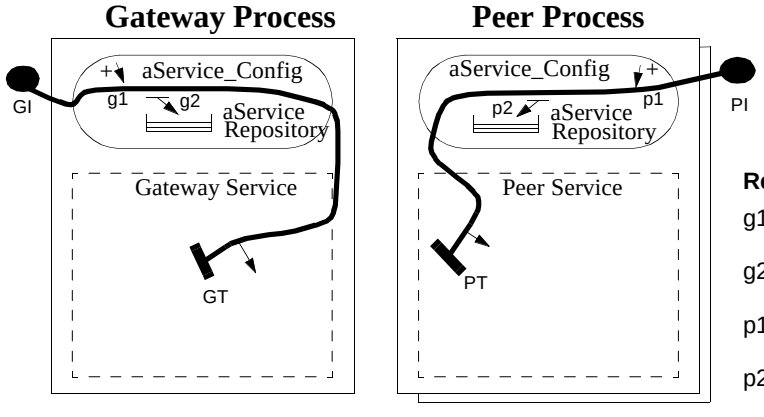
MT: A message is displayed on the standard output device
TT: The gateway process has exited gracefully

**Responsibilities:**

m1: Read a message from a user and send it to the gateway
m2: Read a message from the peer corresponding to the particular input channel
m3: Determine which output channel(s) to use (which peers are to receive the message)
m4: Send the message to the peer corresponding to the output channel
m5: Read the message from the gateway and display it for the user
s1: Interpret the signal received from the system
s2: Shutdown the gateway service and cause the gateway process to exit.

**Figure 16: Basic gateway operation including termination**

Figure 17 gives an incomplete, high level view of initialization, just to indicate what is accomplished by the configurator pattern (without explaining the pattern); postconditions are that service slots are populated.




**Preconditions:**

GI: UNIX command issued to start gateway process
PI: UNIX command issued to start peer process

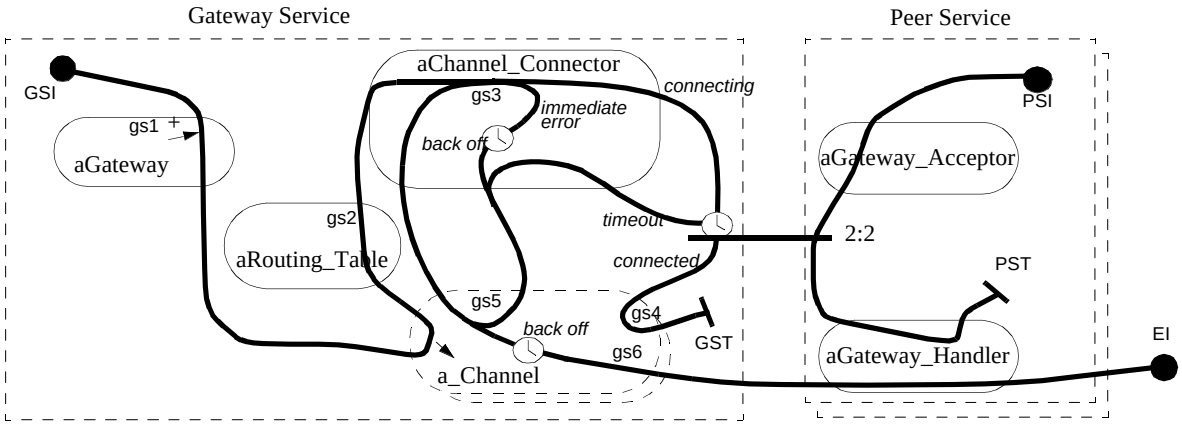
**Responsibilities:**

g1: Create an instance of the Gateway Service
g2: Store the new Gateway Service in the service repository
p1: Create an instance of the Peer Service
p2: Store the new Peer Service in the service repository

**Figure 17: High level initialization of Gateway and Peer processes**

Figure 18 gives, in one diagram, a relatively complete view of the behaviour patterns involved in connecting and error handling. It covers all of the message sequences of Figure 15 in one set of paths superimposed directly on the system components. GSI and PSI continue from the end points GT and PT in Figure 17 (GSI also provides an internal start point in the gateway service for multiple connect scenarios for multiple peers). Figure 18 also adds the error path

EI, which is traversed if a peer that was previously initialized as a receiver mistakenly tries to act as a sender.




**Preconditions:**

GSI: One or more channels needs initialization and this is the first one, or a timer was started, or GST was reached.
PSI: Peer initialization starts.
EI: Message becomes available as input to output peer

**Postconditions:**

GST: Channel initialized and operational
PT: Peer initialized and operational

**Responsibilities (gateway side):**

gs1: Create an instance of Output_Channel or Input_Channel
gs2: Register the channel with the routing table
gs3: Initiate a connection on the newly created channel
gs4: Complete initialization of channel instance
gs5: Reset channel
gs6: Detect error condition

**Figure 18: Connection and error handling patterns**

The only new notational elements are timers and the synchronization bar between the gateway and the peer. Activity along a path entering a timer waits for a timeout period unless some action along another path clears the timer. The 2:2 synchronization bar indicates 2-in-2-out synchronization between the paths, meaning that two incoming scenarios, one along each path, result in two outgoing scenarios, one along each path. The incoming scenarios are synchronized at the bar. Although the reactor pattern is used for much of the messaging that implements this map, it does not appear explicitly in the map, because it is regarded as part of a support layer.

This map is different from the HotDraw one because its preconditions allow concurrent scenarios. Here is how to understand this map. Approach it with a collection of tokens (for example, different coloured beads) to put on the map to represent scenarios. Put a token on each start point from which a scenario may start concurrently, for example, at GSI, and at PSI for each peer. On the peer side, you will need many tokens for many peers, because each peer follows the PSI path independently in its own local context.

Then move each token along its path. The relative start times and rates of progression are undefined, so you may try different combinations to experiment with possible race conditions. When, along the GSI path a token arrives at aChannel_Connector, it must follow one of two possible outgoing routes (indicated by an OR-fork). Let us assume it follows the *connecting* route. Choosing this route assumes that aChannel_Connector has discovered, at gs3, that there are no conditions making the connection attempt with the selected peer impossible, such as an invalid network address. Meanwhile, other tokens continue moving along the PSI path at independent rates of speed.

Now we have many tokens converging on the synchronization bar. What happens depends on the order in which the tokens arrive there. If the GSI token arrives after the PSI token for the selected peer, synchronization will take place immediately for that peer, without starting the timer. This means you can move the GSI token along the path to GST and then return it to the start point to connect another peer. You can also move the PSI token for the now-connected peer to PST, and then remove the token from map.

Otherwise the timer will be started and the GSI token must stop moving until the timer is cleared or runs out. While this token waits, you may start another token down the GSI path to connect another peer.
When the timer is eventually cleared or runs out, you must also continue moving the token that stopped. Now two tokens are tracing independent GSI scenarios. Eventually, many tokens may be moving or stopped at various points along the routes from GSI.

This is enough to give the idea. Many different possibilities exist for routes to be traced through this map, all of which represent possible system behaviours (in some maps routes may be constrained, for example, some combinations of incoming and outgoing routes through an OR-join-fork combination might be forbidden, but not in this example). Thus the map can be used to understand different possible system behaviours, including pathological behaviours such as critical races, or feature interactions.

The use case map view of this application shows a number of objects that the reader might imagine should be threads, for example, the objects that occupy a_Channel slots. ACE supports making such objects either threads or

passive objects but the Gateway software does not exercise the thread option. Instead, the UNIX process handles the multiple channels directly. However, the existing paths in the use case map do not have to be changed (a clarification of the precondition is helpful, to indicate that the controlling Unix process can start a new scenario when it has handed over the current one to a thread). In one case, the timeout context and associated matters must be sorted out by the Unix process, in the other, this sorting out is accomplished automatically by thread support services.

Figure 19 makes the connection back to the message sequences of Figure 15. The highlighted route through the map fragment shown on the left is implemented with the connection-timeout message sequence on the right. Other routes can similarly be related to message sequences. Thus a map can be used as a context for understanding (and documenting) detailed behaviour patterns.

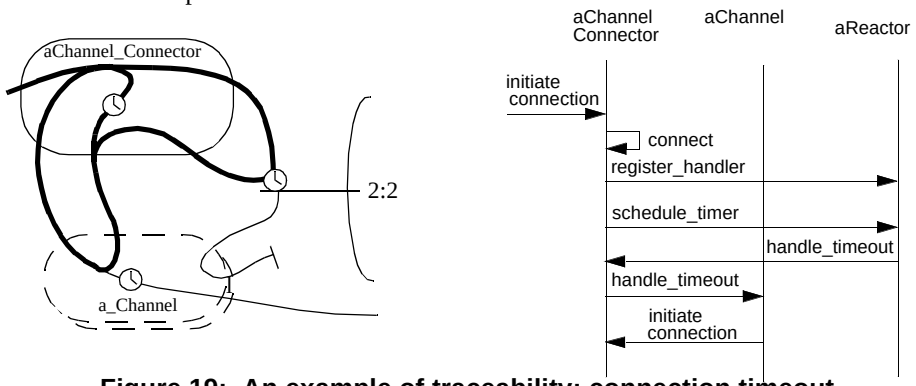



**Figure 19: An example of traceability: connection timeout**

## 4. Discussion

Because this paper is about understanding and documenting existing frameworks, the focus has been on reverse engineering (constructing the view of Figure 1 by studying code). However, we hope that the reader will find it as obvious as we do that any technique that provides helpful system descriptions as a result of reverse engineering is likely also to be helpful for describing systems before the details are decided during forward engineering.

By means of two examples from very different application domains, we have given examples of steps in developing the system side of Figure 1:

- Develop a component context diagram of the system and document the relationships of its components to classes. This may require piecing the system picture together from a study of class relationships, design patterns, and code. Human judgment is required in deciding the level of granularity (for example, all code objects are not necessarily represented as system components at the scale of the particular diagram).
- Identify important use cases that will be used to understand the system.
- Develop an understanding of the use cases in terms of message sequences, by studying the details of the framework (all the examples in this paper did this by documenting the message sequences in diagrams, but actual drawing of such diagrams may not be necessary if the sequences follow standard design patterns).
- Collapse the message sequences into causal sequences and concatenate causal sequences to form use case paths. This is not a mechanical process; human judgment is required, for example, to decide when to collapse details into responsibilities, or when to indicate cyclic behaviour by preconditions instead of path loops. We used preconditions instead of a path loop to indicate repetitive behaviour starting at GSI in Figure 16, to keep the map as simple as possible, without really sacrificing any insight into overall behaviour. The objective is human-understandable paths, committed as little as possible to implementation details, that will give overall insight.
- Document the path preconditions, postconditons, and responsibilities.
- Combine use case paths, as appropriate, into use case maps.
- The result will be a high level model along the lines of Figure 1.
- For traceability back down to details, provide pointers from this model to details required to implement it.

Listed below are some desirable properties (italicized) of any conceptual design model for understanding the operational nature of systems, each followed by a brief comment contrasting the ability of UCMs and conventional software design models to provide these properties (the comments in this list are restricted to well known software methods and CASE tools, and do not cover formal methods, such as Petri nets, which are discussed following the list):

- 1) *Has the primary objective of aiding human reasoning at a high level of abstraction, as opposed to entering details into a computer tool.* UCMs are the only diagramming technique known to the author that was shaped solely by the need for this property. Others, e.g., [15][14][23] [12][2][1][16], are shaped primarily by the need for

machine-executability of design models and/or machine-translatability into code, thus forcing a commitment to details at the level of methods, functions, messages, interprocess communication, interfaces, internal state machines of components, and so forth, that get in the way of reasoning at a high level of abstraction.

- 2) *Is first-class at the macroscopic level, meaning not dependent on details of components or code.* To the author's knowledge, there is only one other system-oriented notation that has this property, the so-called "high level message sequence charts" under development by the Z120 community [12] (this reference covers only detailed message sequence charts, but examples of proposed high level ones are given in [15]). However, this notation does not possess Property 3 and it clouds the mind's eye with boxes in the separate behaviour diagrams that look like components but are not, exacerbating the problem of mentally superimposing behaviour on structure.
- 3) *Combines system behaviour and system structure into a single coherent diagram.* To the author's knowledge, only use case maps possess this property at a high level of abstraction. Other diagramming techniques [12] [15][14][2] require mental combination of information in multiple diagrams.
- 4) *Expresses system self-modification in a compact, lightweight fashion that is easily combined with ordinary behaviour but is also visually separable from it.* To the author's knowledge, only use case maps possess this property strongly.
- 5) *Has diagrams that are easily grasped as visual patterns for a system as a whole.* Use case maps can combine many behaviour patterns in single diagram in a way that enables the mind's eye to sort them out. Recognizing behaviour patterns in superimposed sequence numbers or separate detailed message sequence charts is much more difficult, particularly because many diagrams must be viewed. These visual patterns have been argued elsewhere to be a new kind of design pattern [8], but this is still a controversial view in the object-oriented community.
- 6) *Provides a macroscopic system view for forward engineering, reverse engineering, maintenance, evolution, and reengineering.* Only use case maps and high level message sequence charts provide reference views that are independent of details and so can be used to guide decisions about details. Use case maps do it more compactly and simply (Properties 2-5).
- 7) *Encourages design experimentation.* The lightweight nature and ease of use in back-of-the-envelope style fosters a design process that satisfies requirements by proposing and evaluating design alternatives. This is difficult with techniques that require strong commitment to many details.
- 8) *Scales up.* The big picture remains lightweight because it is not composed from details, and the techniques of factoring, stubbing, and layering provide decomposition techniques that enable large systems to be described.
- 9) *Provides a high level supplement for any detailed model/method.* The concepts are general and insensitive to the details of particular detailed design models/methods.
- 10) *Suggests a new concept of "architecting" behaviour.* A design model along the lines of Figure 1, in which the operational side employs UCMs to achieve the above properties, is a new, useful, and practical form of architectural description. UCMs may be viewed as a technique for "architecting the behaviour" of such systems, in the sense of attributing behaviour to system structure [7]. There is no other technique known to the author that can do this in such a compact and human-understandable fashion.
- 11) *Can be saved for documentation and maintained without unreasonable effort.* The lightweight and compact nature of UCMs contributes to this property. However, tool support is desirable (a UCM editor is currently being developed for this purpose).

It is important to understand that use case maps do not replace the other techniques referred to above, but supplement them to give a higher level view. Work is proceeding on linking use case maps to standard methods and tools in the object-oriented and real time system domains (e.g., [10] links them to the ROOM methodology and toolset [23], and future worked is planned on linking them to UML [2]).

A common remark by skeptics is that anything that can be done with UCMs can also be done with some other technique. For example, causal sequences and self modification can be represented by message sequence charts or Petri nets. This is like saying anything that can be done with higher level programming languages can also be done with assembly language. In the author's opinion, such comments miss the point, which is finding better ways of aiding human understanding in the face of looming complexity brick walls.

Petri nets in particular seem to be a favourite of skeptics, who often suggest they are sufficient to do the job of UCMs. It is true that UCMs can be given execution semantics with formalisms such as Petri nets (students and collaborators have used both Petri nets and LOTOS for this purpose) and that sometimes UCMs can look a bit like Petri nets to the uninitiated. However, there are big differences. UCMs prescribe scenario paths and interpath relationships as first class entities, but not how the scenarios or relationships will emerge at run time; they are a declarative technique (which is why the term "architecting behaviour" is appropriate, rather than "specifying behaviour"). Petri nets are a technique for specifying behaviour. They prescribe how scenarios will be generated but do not identify paths as first class entities; their strength (executability) is a weakness from a UCM perspective (they force the user to become preoccupied with specifying execution details). Petri nets are better viewed as one of several useful techniques for "implementing" UCMs in a high level way when executability is desired, not as replacements for UCMs.

Industrial experience with use case maps may be summarized as follows: The approach always seems to skeptics to be too simple to offer anything new, until the skeptics hit a complexity brick wall. Then the approach appears to be

welcomed by experts, novices, managers, and customers alike. Use case maps reflect a common style of thinking among experts that can be easily communicated to others—a style that has been largely ignored by writers of method textbooks and vendors of CASE tools and, up to now, has lacked a first-class notational representation. Experience is proving the power of the approach in telephony systems, agent systems [11], and banking systems, to name but a few types of systems. Personal communications to the author from users in industry, including students, collaborators, and readers of UCM publications, as well as the author's own experiences working with industry, all tend to support this conclusion.

## 5. Conclusions

We have identified a number of problems that arise in understanding the behaviour of software constructed from frameworks, and offered use case maps to help alleviate the problems. The proposed approach makes coordinated use of use case maps and high level (methodless, messageless), but otherwise conventional, object-oriented descriptions, to provide a model of how the software works and is constructed as a whole. The paper has illustrated, by example, a step-by-step process for developing such a model. A result of using the approach is a dramatic reduction in the number of details required for understanding the software as a whole. The model then can serve to provide traceability down to details and up to requirements, where *down* and *up* indicate directions of decreasing and increasing abstraction.

## Acknowledgments

This work was funded by NSERC and TRIO. We would like to thank Doug Schmidt for useful discussions and the reviewers of this paper for helpful comments. Thanks are due to Alex Hubbard for insights into ACE.

## References

[1] G. Booch, Object-Oriented Design, Benjamin/Cummings, 1994.

[2] Unified Modeling Language (UML) Notation Guide, version 1.1 alpha 6 (1.1 c), 21 July 1997, ftp://ftp.omg.org/pub/docs/ad/97-07-08.ps

[3] R. J. A. Buhr, R. S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996.

[4] R.J.A. Buhr, *Use Case Maps (UCMs) Updated: A Simple Visual Notation for Understanding and Architecting the Behaviour of Large, Complex, Self Modifying Systems*, Carleton report (excerpt of a chapter to appear in a forthcoming ACM book on object-oriented methods), www.sce.carleton.ca/UseCaseMaps/ucmUpdate.ps.

[5] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, D. Pinard, *A High Level Visual Notation for Understanding and Designing Collaborative, Adaptive Behaviour in Multiagent Systems*, Carleton report, April 97, http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/agents-ucms.ps.

[6] R.J.A. Buhr, R.S. Casselman, T.W. Pearce, *Design Patterns with Use Case Maps: A Case Study in Reengineering an Object-Oriented Framework*, SCE 95-17, www.sce.carleton.ca/UseCaseMaps/dpwucm.ps.

[7] R.J.A. Buhr, *Use Case Maps for Attributing Behaviour to Architecture*, Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS), April 15-16, 1996, Honolulu, Hawaii, www.sce.carleton.ca/ftp/pub/UseCaseMaps/attributing.ps.

[8] R.J.A. Buhr, *Design Patterns at Different Scales*, presented at PLoP96, Allerton Park Illinois, Sep 96. www.sce.carleton.ca/ftp/pub/UseCaseMaps/plop.ps.

[9] R.J.A. Buhr, A. Hubbard, *Use Case Maps for Engineering Real Time and Distributed Computer Systems: A Case Study of an ACE Framework Application*. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/hicss.ps.

[10] F. Bordeleau and R.J.A. Buhr, , *The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines*, paper submitted to the Conference on the Engineering of Computer-Based Systems, Monterey, March 97. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/UCM-ROOM.ps.

[11] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, D. Pinard, *Understanding and Defining the Behaviour of Systems of Agents, with Use Case Maps*, submitted to the Conference on Practical Application of Intelligent Agents and Multi-Agent Technology, London, April 97. http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/4paam97.ps.

[12] CCITT Recommendation Z120: Message Sequence Charts (MSC), undated document.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissades, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[14] I. Jacobson et. al., *Object-Oriented Software Engineering (A Use Case Driven Approach)*. ACM Press, Addison-Wesley, 1992.

[15] B. Regnell, M. Andersson, J. Bergstrand, *A Hierarchical Use Case Model with Graphical Representation*, Proc. ECBS96, IEEE Second International Symposium and Workshop on Engineering of Computer Based Systems, March 1996.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson, *Object-Oriented Modelling and Design*, Prentice Hall 1991.

[17] D. C. Schmidt, *ACE: an Object-Oriented Framework for Developing Distributed Applications*, in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[18] D. C. Schmidt, *A System of Reusable Design Patterns for Application-level Gateways*, in *The Theory and Practice of Object Systems* (Special Issue on Patterns and Pattern Languages) (S. P. Berczuk, ed.) Wiley and Sons, 1995

[19] D. C. Schmidt, *Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching*, in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt eds.), Reading, MA: Addison Wesley, 1995

[20] D. C. Schmidt, T. Suda, *The Service Configurator Framework: An Extensible Architecture for Dynamically Configuring Concurrent, Multi-Service Network Daemons*, in *Proceedings of the IEEE Second International Workshop on Configurable Distributed Systems*, Pittsburgh, PA, March 1994.

[21] D. C. Schmidt, *Acceptor: A Design Pattern for Passively Initializing Network Services*, *C++ Report*, SIGS, Vol 7, No. 8, November/December 1995.

[22] D. C. Schmidt, *Connector: A Design Pattern for Actively Initializing Network Services*, *C++ Report*, SIGS, Vol 8, No. 1, January 1996.

[23] B. Selic, G. Gullickson and P.T. Ward, *Real-time Object-Oriented Modeling, Wiley*, 1994.

[24] Shaw and Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.