Design Patterns at Different Scales

R.J.A. Buhr

SCE-96-3: June 8, 1996

Department of Systems & Computer Engineering
Carleton University
Ottawa Canada K1S 5B6
email: buhr@sce.carleton.ca
web: http://www.sce.carleton.ca/faculty/buhr
tel: (613) 520-5718
fax: (613) 520-5727

Design Patterns at Different Scales

R.J.A. Buhr

Department of Systems & Computer Engineering

Carleton University, Ottawa, Canada

Keywords: architecture, design, patterns, scale, use case maps

Abstract

Patterns in the style of the "gang-of-four" book or the ACE framework are becoming popular as a way of enabling developers to reuse good design solutions to implementation problems. However, systems constructed from combinations of many general patterns of these kinds, in which the specifics are in the code-customization details, are not easy to explain or understand as a whole. In particular, it can be hard to understand how stimuli propagate through a specific implemented system as a whole, through many components that may come from many different, overlapping patterns. This is an unsatisfactory state of affairs for real-time and distributed systems, for which stimulus-response behaviour is very important. This paper proposes that a solution to this problem requires a paradigm shift in description techniques at some scale, similar to the paradigm shift patterns provide from code. In the new conceptual domain above this paradigm shift, description techniques would provide a coherent view of a system as a whole and an organizing framework for understanding the combination of patterns from which it is constructed. The prospect also exists of discovering patterns of a different scale and kind in this domain. This paper proposes that a diagramming technique called use case maps provides a suitable paradigm shift and illustrates the proposal with examples that span the range from an object-oriented GUI to a distributed client-server system that processes end-to-end transactions across a network. The motivation for this work is high-level understanding and design of real-time and distributed systems.

1.0 Introduction

To set the stage, imagine the difficulty of trying to understand how a complex physical system, such as an automobile, will behave as a whole, by piecing together information from parts manuals for its subsystems, such as gearbox, transmission, steering, engine, and frame (to name but a few). Imagine how much more difficult it would be if you had never seen or driven an automobile, so that you had no physical model of the look and feel of a whole automobile in your mind's eye to provide context. This seems analagous to trying to piece together an understanding of a software system from details at the level of programming-language classes and methods, because the details combine in complex ways and there are no physical models of software systems to hold in the mind's eye to provide context.

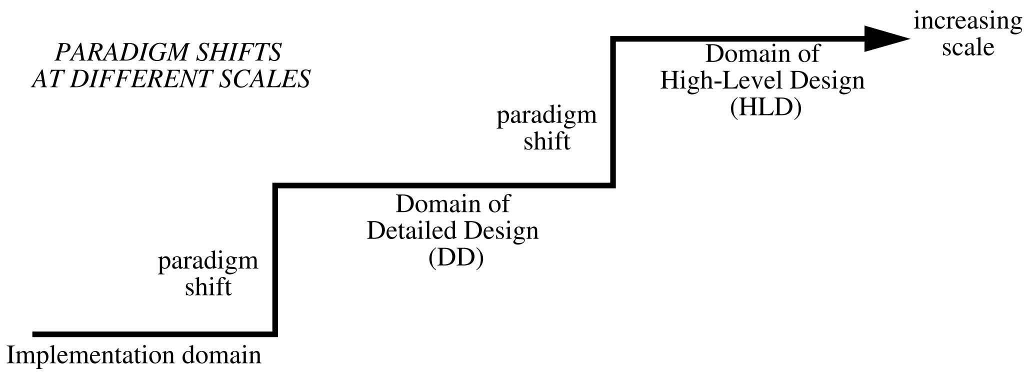
Continuing with the automobile analogy, the behaviour of an automobile as a whole is generally understood—by people who have seen and driven them—in terms of generic high-level behaviour patterns that are universal for all automobiles, not by piecing together details of parts. Examples of such patterns are (1) maintaining *steady progress* (speed and direction) on a hilly, uneven road by manipulating the steering wheel and accelerator and (2) *recovering from a skid* by turning the steering wheel in the opposite direction to the skid and applying power judiciously. Experienced drivers easily understand such generic patterns without having any knowledge of automotive machinery. The more mechanically inclined among them can also understand in a high-level way how the patterns are related to the workings of automative machinery by understanding relationships in causal terms instead of mechanical terms, for example, pushing the accelerator *causes* the engine to generate more power, which *causes* additional force to be applied to the wheels through the transmission, which *causes* the automobile to maintain speed going up a hill. Causal relationships like these are easy to understand without understanding all the mechanical details that realize them. An experienced automative engineer will, of course, work with these details, but will also work with the patterns and how they relate to the details.

Thus the behaviour of physical systems such as automobiles can be understood in terms of generic behaviour patterns that may be seen in the mind's eye in terms not only of the system as a black box, but also in terms of causal relationships that produce purposeful behaviour from the joint functioning of its parts. They are held in the mind's eye in a high-level way based on observation of and experience. They are easily held in the mind's eye because mechanical systems are concrete, physical things that behave according to familiar physical laws and constraints. The term *patterns* seems justified as consistent with ordinary English usage of the term, even though the patterns are different in scale and kind from software design patterns such as those catalogued by the "gang of four" in [6] or embodied in the ACE framework [12].

There are also analogies in the automotive world to detailed design patterns such as those of [6] and [12]. For example, a *multispeed gearbox* pattern may be used to provide a generic solution to the design problem of matching a narrow range of engine speeds to a much wider range of automobile speeds. Such detailed patterns contribute to achieving higher level patterns such as *steady progress* and *recovering from a skid*, but one would not normally try to understand the higher level patterns in terms of pieced-together details except as a means of verifying that the details achieve the desired higher-level results.

We suggest that high-level behaviour patterns like *steady progress* and *recovering from a skid* are also part of how we think about, understand, and design software systems. For example, *recovering from a skid* might be analagous to *recovering from a communications failure* in a distributed software system while trying to process an end-to-end transaction. There are generic patterns for such recovery that span the software components just as there are generic patterns for recovering from a skid that span the components of an automobile. However, such patterns are not as easy to hold in the mind's eye for software systems because of the lack of high-level models to provide context. Although experts manage to do it (that is why they are experts), they cannot easily pass on their knowledge to other people and they take their knowledge with them—and eventually lose it—when they move on to other things.

We need some terminology to relate these ideas more specifically to software. The following staircase diagram identifies a concept of different domains of description of systems at different scales, with paradigm shifts in between. The horizontal steps indicate domains in which certain description techniques apply that are suitable for a particular scale. The vertical risers indicate paradigm shifts when the scale gets large enough that understanding in a particular domain is hindered by too much detail. The HLD domain is at the level of automotive behaviour patterns such as *steady progress* and *recovering from a skid*, or software system behaviour patterns such as *recovering from a communications failure* in the context of processing a transaction. The HLD domain could also be referred to as the *system architecture* domain, assuming behaviour patterns are seen as part of "architecture". The DD domain is at the level of detailed patterns such as *multispeed gearbox* or of software patterns such as those of [6] and [12].



The purpose of this paper is to propose and illustrate *use case maps* (UCMs)[2][3][4] as (1) a means of describing HLD patterns in a way that can be held in the minds eye and easily described to others, above the level of implementation details, and (2) a context for understanding how certain kinds of DD patterns fit into a system as a whole. Applying the term *pattern* to UCMs seems justified if for no other reason than that they provide a means of representing generic system scenarios as *visual patterns* superimposed on diagrams of system structure. However, there are also other reasons (see Section 4.0). UCMs are maps of cause-effect paths taken through systems by the scenarios of use cases [7]. The term *use case* is not restricted to human users interacting with system externals—in general, the term includes the possibility of scenerios involving many systems or subsystems that are "users" of each other. UCMs have two forms, bound and unbound. The bound form shows paths over structure, with labelled responsibility points along paths bound to components by visual superposition. This form combines structure and behaviour in compact visual form; we say that this form provides a seamless bridge between structure and behaviour at the HLD level. The unbound form shows only responsibilities, not components. This form provides a means of expressing behaviour requirements without necessarily committing to HLD solutions. Because the same paths may be used for both requirements and HLD solutions, we say that this form provides a seamless bridge between requirements and solutions. This twofold seamlessness—between structure and behaviour in the HLD domain and between requirements and HLD solutions—distinguishes UCMs from other scenario notations [5][7][11] and makes them particularly appropriate for the purposes we propose here.

For software systems, the DD domain uses familiar description techniques such as detailed class relationship diagrams, message sequence charts, object visibility diagrams, and state transition diagrams.

Of course, systems may be described in each of the DD and HLD domains, independently of whether or not they are implemented with design patterns; however, the focus of this paper is patterns. We shall call design patterns in the DD domain *DD patterns* and ones in the HLD domain *HLD patterns*. We classify the familiar design patterns of the "gang-of-four" book [6] or the ACE framework [12] as DD patterns. Claims have been made elsewhere[2][4] that some UCMs can be viewed as HLD patterns (see also Section 4.0).

The issues to be addressed in the HLD domain for software design are different in kind and require a different perspective than the issues that drive design in the DD domain, for example:

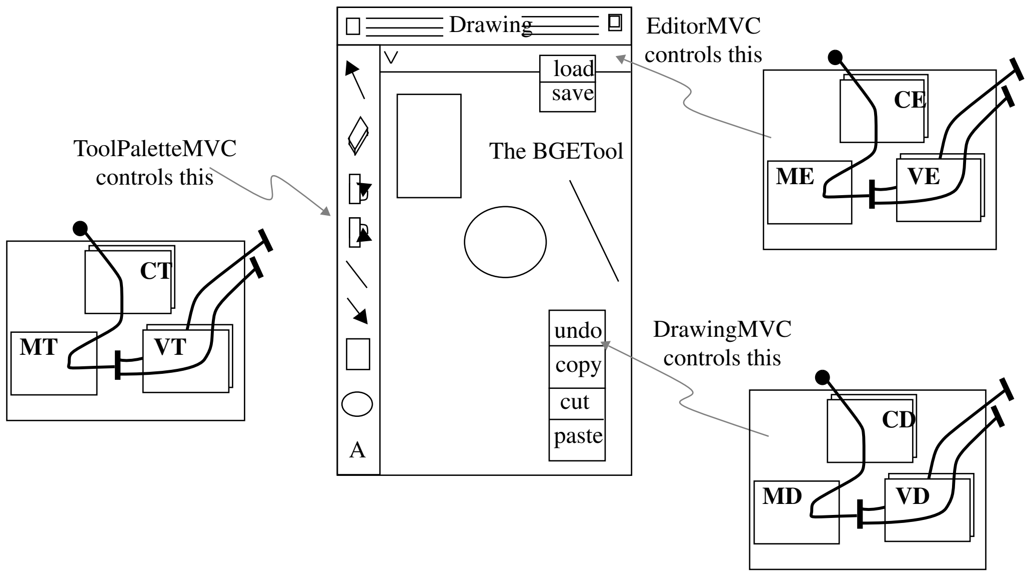
- How can critical stimulus-response scenarios at the requirements level, such as might be expressed by use cases [7], be linked to possible solution architectures in the HLD domain, to make design tradeoffs before committing to details in the DD domain?
- How can trade-offs be made between the reusability-extensibility benefits and the performance overheads of DD patterns without being forced to use to cut-and-try approaches that bounce back and forth between the the DD and Implementation levels?
- If we can speak of patterns in the HLD domain for end-to-end behaviour in real-time and distributed systems, then how can such patterns be linked to DD patterns for implementation?
- How can software architecture be described in the HLD domain to express not only structure, but also behaviour, to guide forward engineering, reeingineering and evolution?
- How can whole-system behaviour patterns be included as first-class elements of architecture, above the level of details in the DD domain, to evaluate qualities such as performance and robustness, for example, in architectural assessment techniques such as SAAM [8]?

Dealing with such issues is made easier by making a paradigm shift from DD description techniques to HLD ones such as UCMs. Although this adds more concepts to an already-overloaded concept set, this is not as troublesome as it may seem at first, because UCMs offer a common-sense notation that is quite easy to learn and use in a coordinated way with other notations, once its purpose is clearly understood.

The ideas are developed in this paper through two examples that span the range from object-oriented programs to real-time and distributed systems: an object-oriented GUI and a distributed client-server system that processes end-to-end transactions. The UCM notation needed to understand the examples will be explained as we go along.

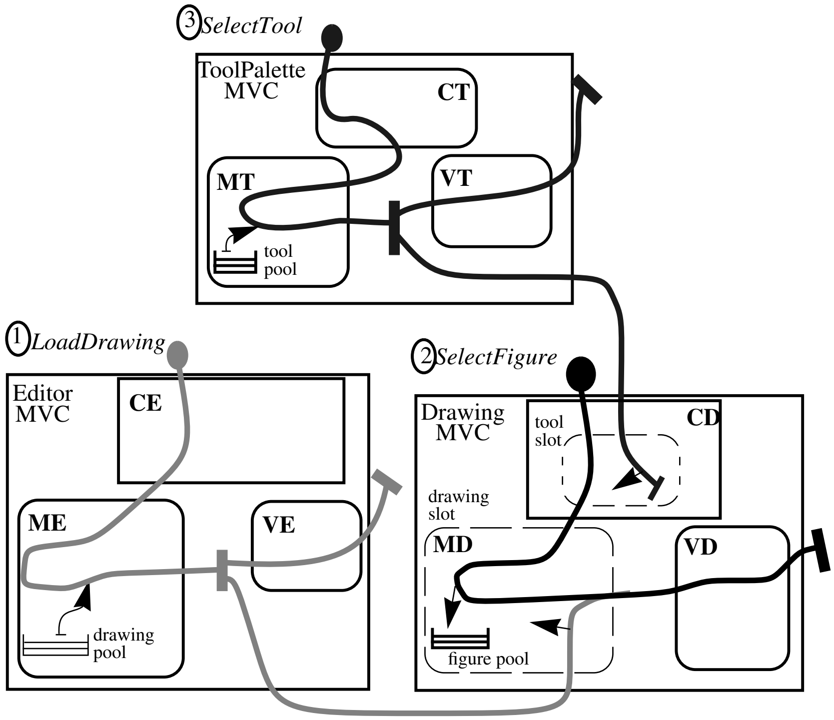
2.0 A Familiar Example from an Unfamiliar Perspective

Although the motivation of this work comes from real-time and distributed systems, it will be helpful to introduce the ideas through a simple, familiar OO example, namely a GUI controlled by a set of MVC teams (*teams* are collaborating sets of objects or other runtime components). MVC stands for well known Model-View-Controller organization (we could say *architecture* but do not for reasons explained earlier), in which the model maintains data, the views display data, and the controllers manage interactions with the human user for different views. The diagram below is intended to convey this idea in a stylized fashion. Each MVC team controls a different area of the screen. Each MVC is characterized by a UCM of the same form (a triad of components with a forked path superimposed).



Let us now understand this basic UCM. Its form is sufficient to gives the general idea of UCMs, although later specialization to the menu bar, the tool pallette, and the drawing areas of the screen, plus requirements for coupling the UCMs, will modify this form somewhat. In the basic UCM, a scenario starts at some point of stimulus outside the team, traverses the controller to analyze the interaction, traverses the model to update the data, and traverses a parallel fork (called an AND fork) to all views to display the result (the responsibilities along the paths are not shown in this diagram, but are implied, and the AND fork means here don't-care ordering of responsibilities along the path segments following it). Notice that the UCM does not show any back-and-forth interaction between models and views of the kind that typically occurs in programs at the message level. This is because the UCM is only concerrned with indicating that display follows update, not with indicating message sequences. How the views are notified by messages of the need for a new display and how they get the data to perform the display are deferred as lower-level details.

While this basic UCM is somewhat simpler than the view that we would get in terms of messages, the real simplification with UCMs comes at the larger scale of a system as a whole. The next diagram specializes and combines these individual UCMs into a composite UCM for a set of coupled MVCs. As before, responsibilities along paths are not shown, only implied.



The notation in this diagram is as follows:

- *Teams* (boxes with sharp corners) are operational groupings of system-level components. UCM teams may or may not represent instances of programming-language classes.
- *Objects* (boxes with rounded corners) are data or procedure abstractions that need to be seen as system-level components to understand the system at the scale of interest. UCM objects may or may not represent instances of programming-language classes.
- *Slots* (boxes with dashed outlines) are analagous to positions in human organizations that exist independently of their occupancy or occupants. They are used to indicate the possibility of different components occupying organizational positions at different times (e.g., drawings in the MD slot and tools in the tools slot in CD). The slot concept simplifies diagrams of systems in which visibility relationships change over time by raising the level of abstraction above specific components. The slot notation does not show when slots are empty or not, one must consult the paths for that. From a static structure perspective, slots are *potential* components: when a slot is empty, there is nothing there to do anything, not even a control shell (there is nothing to stop an implemention from making a conceptual slot into a literal component that provides a control shell, but this is not the general concept). From another perspective, slots are places where different components may play the same role at different times.
- *Pools* (boxes open at the top with lines across them symbolizing storage for many internal quantities) are places that hold components in readiness to occupy slots (e.g., a set of drawing objects or a set of tool objects).
- *Move arrows* (small arrows between paths and pools or slots) are used to indicate the possibility of component movement that may cause slots to become occupied or empty. For example, the *LoadDrawing* path shows the possibility of a drawing object moving from a drawing pool in ME to the drawing slot MD (implicitly, the object moves along the path). Movement is a metaphor for changing visibility. Moving a component into a slot means making the component visible to those who must interact with it at the slot location. For example, the drawing object becomes visible at the MD location; the figures pool inside the MD slot means each drawing object brings its own pool of figures with it to the slot.

Coupling between MVCs in this diagram is accomplished by one path setting up preconditions for the next path (this is only one of many possible forms of coupling in UCMs). For example, a precondition of the *SelectFigure* path would be that the MD slot is occupied, which would be a postcondition of the *LoadDrawing* path. Preconditions and postconditions of paths are part of prose map documentation, not of the visual presentation.

Let us now trace the *LoadDrawing* path, remembering that responsibilities along the path, although not shown, are implied. At CE, a user interaction is decoded as a request to load a drawing. At ME, the requested drawing is moved from a pool. At VE, the drawing name is displayed in the editor area of the screen. At MD, the drawing enters the slot, meaning becomes visible in the DrawingMVC (in other words, is ready to play its role in the DrawingMVC). At VD, the figures of the drawing are displayed in the drawing area of the screen.

This should be enough information to understand the general nature of the map as a whole. The important point here is that the map gives an overview in a single diagram of some rather complex behaviour that is not easy to keep in the mind's eye when working at the level of, say, observer patterns, and message sequences, both of which are used to implement this behaviour.

The above UCM has an elegant regularity about it. A case study of an actual GUI framework [3] turned up a much more irregular UCM to accomplish the same purpose that seemed to come from detailed coding decisions performed by programmers. Regularity seems a desirable property, because it aids understanding, reuse, maintenance, and evolution, but regularity or irregularity at the HLD level of abstraction is difficult to see when looking at details in the DD domain. UCMs expose it directly in the HLD domain.

The other side of the coin is that the DD details are needed for implementation. UCMs can be used to indicate where DD patterns are to be used for implementation, as will be shown in Section 3.0.

2.1 UCMs and Object-Oriented Design

Although somewhat off the topic of design patterns at different scales, it may be helpful to note in passing that UCMs offer the prospect of changing some object-oriented design practices for the better in cases where overall behaviour of a system is too important to leave as a detail to be adjusted by cut-and-try methods that bounce back and forth between the DD and Implementation domains (for example, real-time and distributed systems).

UCMs and class relationship can be developed independently in the HLD domain and associated with each other at any time to evaluate tradeoffs between construction and behaviour in a qualitative manner. The high-level association is not onerous to establish or maintain. All that is required is identifying the *existence* of programming-language classes that will be instantiated to give the components in the UCMs. For example, in the GUI system: each team might or might not have an associated class (if not, teams simply identify sets of collaborating objects); each object would have one associated class (in general, map objects might be abstractions that aggregrate many program classes, and there might be many other programming classes that are simply not shown because they are too small a scale to be of interest); each slot could have many different associated classes (the only constraint at this level being the ability to perform the required responsibilities); and each pool would probably have one associated class. The class picture for the GUI example was omitted in the GUI example as something that needs no explanation for the intended audience of this paper, without in any way meaning to diminish its importance for object-oriented design.

3.0 UCMs as Conceptual Frameworks for DD Patterns

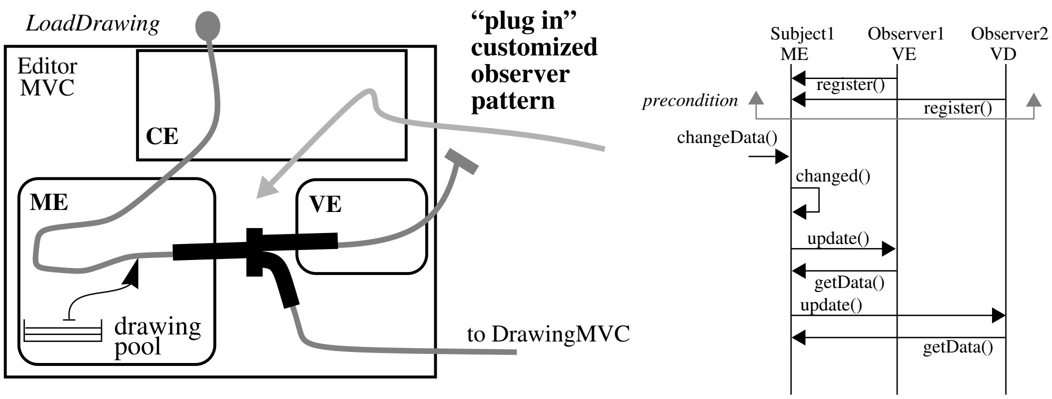
Let us use the observer pattern of [6] to implement the some of the causal relationships in the GUI UCM (which has several places where this may be done). The result will be to make the UCM into a conceptual framework for understanding how different instances of the observer pattern come into play in different places in the system. Although we show this for only one DD pattern, the observer, the principle is the same for many. Of course, DD patterns that do not have a behavioural aspect are orthogonal to UCMs, so we are only talking here about DD patterns with a behavioural component, of which the observer pattern is an example.

We present this use of the observer pattern through its message sequences, not its classes, because we are concerned here primarily with behaviour. The associations that will be made will apply to the classes of the pattern in an obvious way that we shall omit describing, for brevity. The concept is one of "plugging in" the pattern's message sequences, as follows: identify the affected path segments in the map (done below by highlighting them with thick lines); identify the DD pattern to be plugged in; customize the message sequences of the DD pattern by associating its components with those of the UCM; treat slots as participants in messag sequences only at points where they are occupied, otherwise customize the message sequences to capture the meaning of moving a component into a slot, namely that components that must interact with the one in the slot are notified that the slot is now occupied (remember that slot occupancy is a metaphor for visibility).

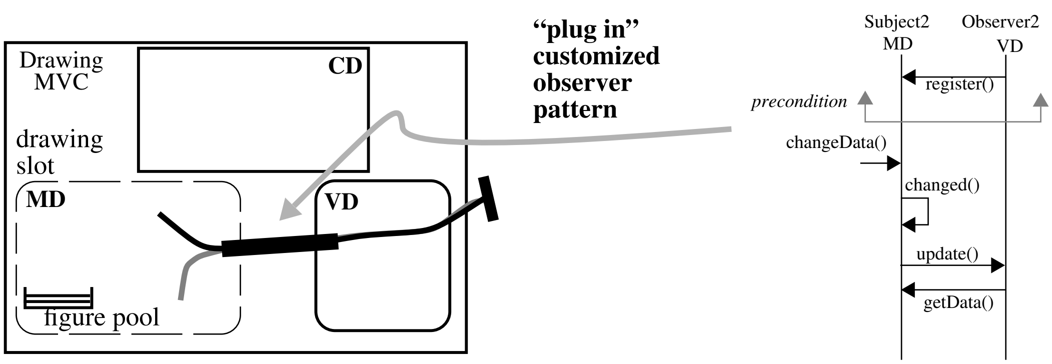
Thus may be established relationships between the HLD and DD domains that are useful for many purposes: to provide a path from the HLD domain to the DD domain that leads to implementation (the following example illustrates this); to understand, at a higher level of abstraction, combinations of DD patterns that appear in a particular system (the following example also illustrates this); and to provide a path from implementation to high-level understanding in the HLD domain (an example is given in [3]).

The next figure illustrates this idea for a part of the UCM in the Editor MVC. The arrow labelled “plug-in” symbolizes the idea that a tool that would enable a user to highlight the portion of the UCM shown here, select the observer pattern from a patterns library, and enter associations between the library pattern and the UCM to customize the pattern, with the result that the UCM and the DD patterns would be cross-referenced to each other.

This customized messaging pattern has a precondition, namely that components along the branches of the AND fork must have previously registered themselves as observers of ME. Given this precondition, the AND fork may be implemented with the message sequences shown. The back-and-forth message interactions are not shown in the UCM because the UCM is concerned only with the causal sequence that display follows update. VD (in DrawingMVC) is identified as the observer along the branch of the AND fork that goes to DrawingMVC. At first glance, it seems that MD should be the observer, but it cannot be because it is an empty slot at this point along this path; in other words there is nothing there to act as a observer. We have to look to the DrawingMVC map to understand why VD is the appropriate customization (next figure).

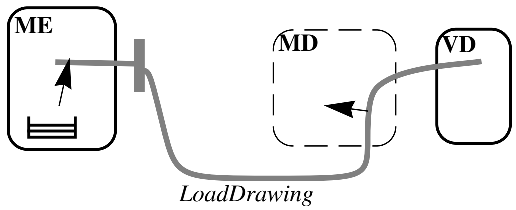


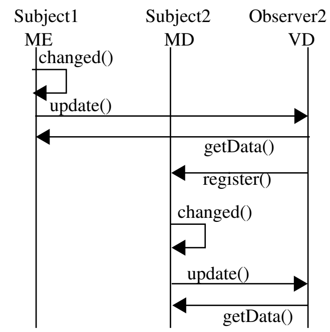
The next figure shows how a customized observer pattern may be plugged in to the Drawing MVC in a similar fashion. This customization requires only one observer because there is no AND fork in the map. At this point, the MD slot is occupied, so its occupant can participate in the message interchanges shown. The precondition that VD is registered as an observer of the occupant of the MD slot must be established *after* a drawing object is moved into the slot (otherwise there would nothing with which to register). Now the question is, what is the message equivalent of moving a drawing object into the MD slot? The answer is, it is the above message interchange between ME and VD to notify VD of the identity of the new drawing object. This interchange enables VD to register itself as an observer of the occupant of MD, which satisfies the precondition for the plugged-in pattern.



The next figure shows how the UCM may be used as a framework for understanding combinations of DD patterns. It uses two instances of the DD pattern as an example, but the principle would be the same for different patterns. This figure puts the above plugged-in pieces together along the *LoadDrawing* path: the fragment of this path on the left below is implemented by the message sequence on the right (the changeData() message is missing because, along the *LoadDrawing* path, the new drawing is the only change to be displayed). A programmer could easily optimize this message sequence by cutting out some of the back-and-forth messaging required by the standard pattern, but at the expense of making the code nonuniform and less easy to understand in terms of standard patterns.

Note that the idea is not that the plug-in process generates message sequence diagrams like this to replace the UCM but that the UCM provides a context for understanding the message sequences.





A customized observer pattern could also be plugged into the ToolPaletteMVC, but there is nothing new in doing it, so we omit doing it here.

In passing, note that this concept of plugging in DD patterns opens up an interesting possibility for performance estimation during early design. Associate performance estimates with elements of DD patterns to be plugged in; then peformance effects can be summed along UCM paths to get response time estimates for a system as a whole. This might give advance warning of unacceptable performance that otherwise would not be discovered until much later in the development process.

4.0 UCMs as HLD Patterns

We may think of some UCMs as providing HLD patterns. Thus, the staircase diagram of Section 1.0 may be thought of as representing, not just a hierarchy of description techniques at different scales, but also the possiblity of a hierarchy of design patterns at different scales, as the title of the paper suggests. The term *pattern* is appropriate for the following reasons:

- *UCMs provide a way of representing system scenarios as visual patterns that include both system behaviour and system structure.* Thus UCMs exploit human visual pattern recognition ability, as is appropriate for a description technique that aims to aid human understanding.
- *UCMs can be reusable design artifacts.* For example, the basic UCM given above for a single MVC is reused with variations in several different places to design a system of coupled MVCs. The UCM for the system of coupled MVCs given above looks like it might itself be reusable in essentially the same form in different applications. A general coupled-MVC pattern would have more paths, to cover the possiblity of more different kinds of behaviour, but this would add nothing new in principle.
- *UCMs can represent HLD solutions to HLD problems.* For example, such a problem is coupling a set of MVCs. Solutions to such problems are at the level of system architecture and are related to the idea of architectural style [10], although expressed in a novel way with UCMs. The driving forces leading to choices of system architecture patterns or styles are many, varied, and beyond the scope of this paper. Our purpose here is only to point out that UCMs provide a high-level way way of describing and understanding them.
- *UCMs allow us to see higher-level patterns in combinations of DD patterns* by providing a notational framework for placing DD patterns in a system context.

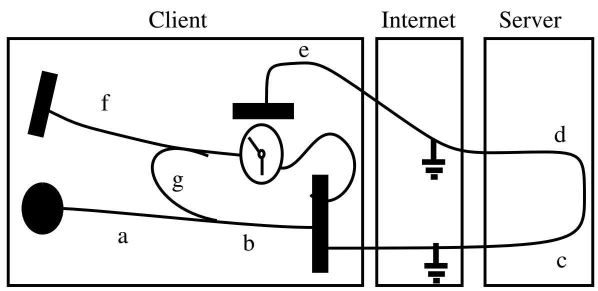
HLD patterns described by UCMs have a structural aspect and a behavioural aspect. The structural aspect is given by a UCM diagram without the paths (in other words, by a diagram showing sets of components that are positioned in standard ways relative to each other to suggest operational relationships).The behavioural aspect is given by a UCM with paths superimposed on the structural aspect. (In general there may also be another aspect, given by high-level class relationship diagrams, but that aspect is well understood in the object-oriented community and is therefore outside the scope of this paper).

5.0 Distributed Transactions in a Client-Server System

Let us now apply UCMs to an example from the domain that motivated the invention of UCMs in the first place: real-time and distributed systems. The example is a client and a server interacting over the internet to process transactions reliably in the presence of unreliability in the internet. This example is at a much larger scale than the GUI example treated so far. We can imagine that there might be layers of UCMs required to describe it, starting from UCMs for the internal details of the clients and servers that might be similar in kind to those for the coupled MVCs in the GUI. However, at the scale in which we are interested, these things are details and so are left implicit here. The point is to illustrate how large-scale issues can be dealt with by UCMs and how there may just as easily be design patterns at this large scale as at the smaller scales with which we have been concerned up to now.

We set out to develop this example without making any commitment to the nature of clients and servers: they might be human beings interacting with a plain ordinary email system, who maintain the transaction concept in their minds, or they might be sophisticated computer systems with full blown GUIs and sophisticated internal mechanisms for preserving the integrity of transactions. A strength of UCMs is that deferring such commitments is easy and natural.

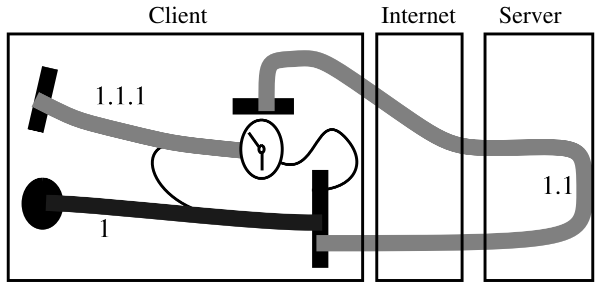
The figure below presents, without any preamble, a UCM that can be regarded as an HLD pattern, because the behaviour it implies is general and widely used. The named points along the paths identify responsibilities, which will be left out of the remaining diagrams of this section (although assumed to be present). The basic pattern in this map is the responsibility sequence a-b-c-d-e-f, which represents a successful transaction. Possible failure points are identified by ground symbols borrowed from electrical engineering, symbolizing that scenario along paths may end abnormally at these points. The rest of the map is concerned with detecting and recovering from failures at these points.



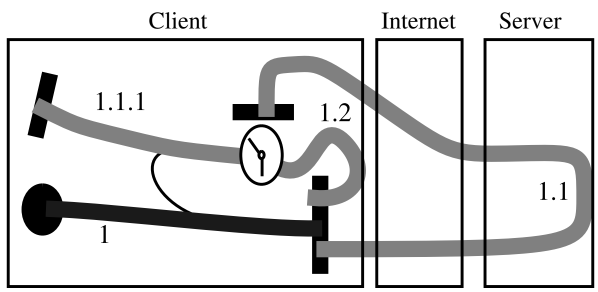
RESPONSIBILITIES
a. Start transaction
b. Produce
c. Consume
d. Acknowledge consumption
e. Observe acknowledgement
f. Conclude transaction.
g. Retry after timeout

The principle is to use a map like this as a visual pattern that can (1) be understood by those familiar with it as embodying a set of different scenarios and (2) be used to explain the scenarios to others by pointing and tracing. The map *documentation* must include enumeration of legal and illegal segment sequences that make up scenarios, but the visual form of the map remains uncluttered by these things (presenting this information visually would detract from the understandibility of maps as visual patterns). With this principle in mind, let us now examine the meaning of this map in terms of segment sequences that make up scenarios.

To explain the possibilities in the above map, we shall use thick, shaded lines to identify path segments that are involved in scenarios and indented sequence numbers to represent causal sequences of segments that make up scenarios. For example, the map below explicitly identifies segments 1, 1.1, and 1.1.1 with a scenario. This scenario is easily recognized as the fundamental success scenario of this map, which produces the responsibility sequence a-b-c-d-e-f.

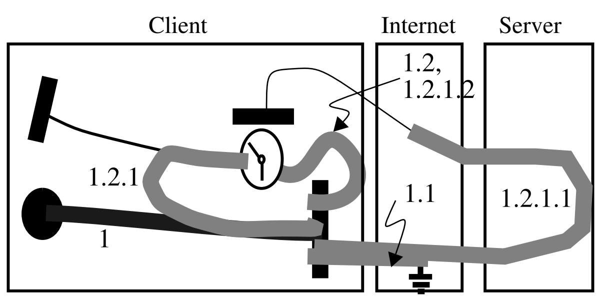


In order to understand this map fully, one needs to understand that there is actually an additional segment involved in the above scenario, as shown below. The additional segment 1.2 proceeds in parallel with 1.1. Segment 1.2 sets up a timed waiting condition in the client (the clock symbol is a timed waiting place that indicates a scenario is blocked along this path until either the timer runs out or a triggering scenario arrives along another path). This segment has no further effect in the success scenario because, by definition, the success scenario triggers the timed waiting place from segment 1.1 before timeout can occur. Segment 1.1.1 actually follows from *both* 1.1 and 1.2, because it is a continuation of the 1.2 path. However, it is a continuation without timeout that is caused by 1.1, so we identify it as 1.1.1 and reserve the sequence number 1.2.1—not needed here—for a continuation of path 1.2 caused by timeout.

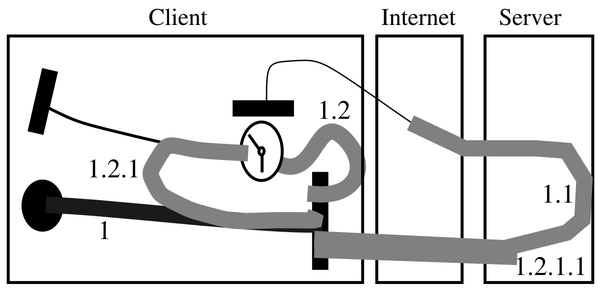


Note that multiple scenarios could proceed concurrently through this map. The notation tells us that paths exist to make it possible, but not the effects of congestion at points like the timed waiting place. In other words, the notation is not for specifying behavioural details.

Now we can see other possibilities. For example, in the diagram below, segment 1.1 fails, resulting in the segment sequence 1.2 followed (after timeout) by 1.2.1. This sequence would continue with 1.2.1.1 and 1.2.1.2 proceeding concurrently (the same segment may appear in different places in a scenario sequence). The highlighting and labelling are now beginning to be visually cumbersome, but, as explained above, they are not part of the visual presentation of maps, they are only being used here for explanation. To the initiated, the sequences are part of the fabric of the map.

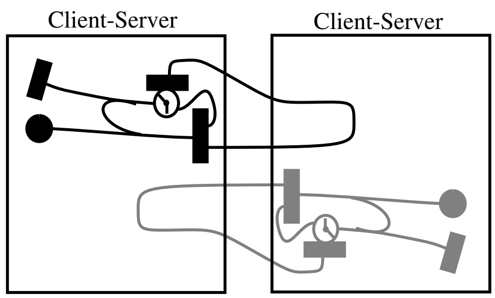


Another possibility is a race between a slow success scenario and the timer that is won by the timer, as suggested (incompletely) below. Segment 1.2.1 would cause segment 1.2.1.1 to proceed to the server, possibly confusing the server, who would see the “same” transaction twice (timestamps could be used to disambiguate). To the initiated, the possibility of such a race is implicit in the fabric of the map. Observe how the UCM fosters analysis of concrete issues like races and the need for timestamps at a high level of abstraction.

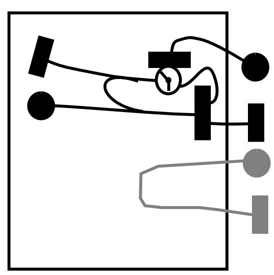


To illustrate the idea that UCMs can be first-class HLD patterns that can be understood and manipulated independently of their relationship to details, let us examine the implications of the same map applying in both directions, instead of only one. The UCM below conveys the idea, leaving out the internet box (thus, in effect, treating it as a layer). This map could be tidied up visually by redrawing it to overlay some paths (e.g., the ones through the internet that go in the same direction); however, this is a detail, not a matter of principle.

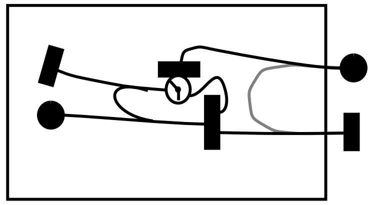
This map shows two mirror-image patterns that we intend here to be concurrent. However, there is nothing in the visual form of the map that says “concurrent”. This map is not different in kind from the earlier ones for the sequential GUI example, which also had multiple paths in them. There was nothing in the visual form of the GUI map that said “not concurrent”. Concurrency or not is determined by the preconditions associated with the paths, not by the visual form of the map. In the GUI example, sequentiality was imposed by making the preconditions of one path the postconditions of another. In this client-server example, concurrency would be indicated by a precondition that a scenario may start down any path at any time. The effect is that clients may be servers and servers clients concurrently, which is why we have renamed each box Client-Server.



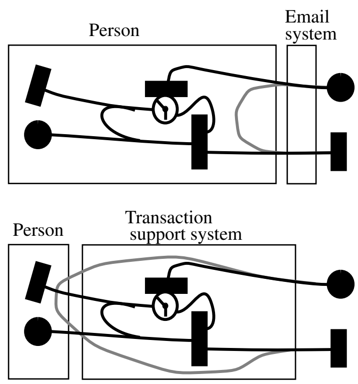
Now let us zero in on one Client-Server box and treat it as a system in its own right with its own UCM. This is done by cutting the above UCM between the two boxes, and terminating the ends as shown below (the shading helps to keep the original larger-scale UCM in mind).



A little bit of manipulation makes this into a composite pattern that combines the above paths, a pattern that is useful to guide design. This UCM is unbound relative to the internal components of the Client-Server box.



This unbound UCM may now be used to explore different designs for the Client-Server box at the HLD level, as suggested by the next diagram, which shows two different bindings. In one binding, a person keeps all the transaction information in mind and uses an email system as a communication mechanism. In the other, a person uses a transaction support system that manages all aspects of transactions. The different bindings can be used to help evaluate tradeoffs. Thus the unbound map is a pattern that can be customized by binding it to different component organizations (the customization requires allocating responsibilities to componentsl, possibly distorting some paths to make this possible, as illustrated by the grey path in the figure).



The possibility exists of a hierarchy of UCMs in the HLD domain. For example, if the email system in the above diagram was implemented with a GUI, it might have internal MVCs that could be represented by UCMs along the lines illustrated in Section 2.0.

The concept of plugging DD patterns into UCMs was illustrated in Section 3.0 and will therefore not be illustrated again here (an example from ACE given elsewhere [1] illustrates it for a real-time and distributed system example). This concept is not altered by the facts that the example of Section 3.0 is fundamentally sequential and localized in one computer and the example here is fundamentally concurrent and distributed, because the essential nature of use case maps is the same in both cases.

6.0 Discussion

The use of the term *pattern* for UCMs may be controversial in the OO community. However, we hope that enough arguments and examples have been given to justify it. The point is that these are patterns at the HLD level that are different in kind from familiar DD ones. A number of other UCM patterns are presented in [2], [3], and [4] and others are in our minds or await discovery. The ones presented in this paper are only a small sampling to suggest possibilities. Discovery of additional UCM patterns, development of systematic techniques for documenting them, inspired by [6] and [12], development of catalogs of them, inspired by [6] and [12], and development of the tools and supporting techniques for using a hierarchy of HLD and DD patterns in a coordinated way, all remain in the future.

The examples of this paper have attempted to illustrate that UCMs are widely applicable to different types of applications and implementation techniques. Some nuances are useful for real-time systems, such as a component notation that includes processes [4], so that in addition to plugging in messaging patterns between objects, we can also plug in IPC patterns between processes, but these are refinements that are outside the scope of this paper. We see UCMs as potentially useful for describing large-scale behaviour patterns in object-oriented, real-time frameworks like ACE [12] and in applications built from ACE; the concept of plugging in detailed patterns from ACE would be essentially unchanged ([1] gives an example).

Let us now propose answers to the questions we asked in the introduction (recall that we said that the questions provided examples of issues that justified a DD-HLD paradigm shift):

- How can critical stimulus-response scenarios at the requirements level, such as might be expressed by use cases [7], be linked to possible solution architectures in the HLD domain, to make design tradeoffs before committing to details in the DD domain? We propose that the answer is by applying UCMs as invariant HLD patterns for different system structures, as was illustrated at the end of Section 5.0.
- How can trade-offs be made between the reusability-extensibility benefits and the performance overheads of DD patterns without being forced to use to cut-and-try approaches that bounce back and forth between the the DD and Implementation levels? We propose that the answer is twofold. First, sum along paths estimated execution times from plugged-in DD patterns, as was suggested at the end of Section 3.0. Then, if necessary, change the class organization to reduce the times. Because both UCMs and class hierarchies are first-class models relative to each other (meaning that the two can be developed in either order), this can be done before major detailed commitments have been made in the class hierarchy.
- If we can speak of patterns in the HLD domain for end-to-end behaviour in real-time and distributed systems, then how can such patterns be linked to DD patterns for implementation? We propose that the answer is to express real-time patterns in forms like the timeout-recovery pattern of Section 5.0 and then to use such HLD patterns as frameworks for plugging in DD patterns, as was illustrated in Section 3.0 for a different example.
- How can software architecture be described in the HLD domain to express not only structure, but also behaviour, to guide forward engineering, reeingineering and evolution? We propose that the answer is to attribute behaviour to architecture with UCMs [2] that can be used as invariant guides for developing or changing details in the DD domain. Without such the HLD view, one is faced with both understanding the system in DD terms and changing it in DD terms at the same time.
- How can whole-system behaviour patterns be included as first-class elements of architecture, above the level of details in the DD domain, to evaluate qualities such as performance and robustness, for example, in architectural assessment techniques such as SAAM [8]? We propose that the answer is to begin as above and then to associate quantitative measures with paths, for example, measures of response time, path interference, and regularity (on the principle that robustness is related to regularity).

Pointing to industry successes following from the use of UCMs is difficult, even though they have become popular in some quarters in industry for HLD reviews and understanding/documenting system architecture, for both object-oriented programs and real-time and distributed systems. This is because they are too new for tool support to have emerged and lack of such support limits their mandated presence in projects once serious implementation starts. People may still think in terms of them, but maintaining diagrams in a systematic way becomes onerous. The application of UCMs as design patterns is so far mainly a gleam in the author's eye. Achieving the full potential of the UCMs as suggested by this paper must await the development of UCM support tools and HLD patterns catalogues.

7.0 Conclusions

A concept of a coordinated hierarchy of behaviour-centered design patterns of different scales has been presented. The concept requires a paradigm shift from the bottom to the top of the hierarchy to focus attention on different issues. At the top of the hierarchy, use case maps (UCMs) are proposed as a means of expressing high-level design (HLD) patterns that draw together scenarios and structure in compact, visual form. UCMs are also proposed as a framework for plugging in detailed design (DD) patterns one level down in the hierarchy and for understanding combinations of DD patterns in a system context. In the spite of the paradigm shift adding new concepts to an already overloaded concept set, limited industrial experience so far indicates that the approach adds value and would add even more value if more support was available, such as tools, and catalogs of standard HLD patterns. Two examples illustrated that the approach spans the range from object-oriented programs to real-time and distributed systems: an object-oriented GUI and a distributed client-server system that handles end-to-end transactions. The approach was motivated by real-time and distributed systems, and the current focus of our efforts to extend and apply it are in this domain, but the examples illustrate that it has wider applicability.

Acknowledgments

NSERC and TRIOSOFT are providing current financial assistance for aspects of this work. BNR provided financial support for some of the initial development of the ideas. Colleagues, industrial collaborators, and many students, too numerous to list, have helped greatly by performing interesting case studies of significant applications, doing work to extend the ideas, and asking difficult questions. Grateful acknowledgment is made of helpful discussions about this paper with Jim Coplien and Doug Schmidt.

References

[1] R.J.A. Buhr, A. Hubbard,*Understanding the Behaviour of Real-Time and Distributed Ssytems Constructed from Object-Oriented Frameworks*, SCE report—http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/hiccs.ps

[2] R.J.A. Buhr, *Use Case Maps for Attributing Behaviour to Architecture*, SCE-96-2: June 8, 1996, Contribution to the Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS), April 15-16, 1996, Honolulu, Hawaii, http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/attributing.ps.

[3] 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, http://www.sce.carleton.ca/ftp/pub/UseCaseMaps/dpwucm.ps.

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

[5] 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.

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

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

[8] R. Kazman, L. Bass, G. Abowd, M. Webb, SAAM: A method for Analyzing the Properties of Sofware Architectures, Proc. ICSE 16, Sorrento, Italy, May 1994, 81-90.

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

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

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

[12] D.C. Schmidt, *The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software*, updated version of a paper appearing in the 11th and 12th Sun User Group conferences, San Jose, CA, Dec 7-9, 1993 and San Fransisco, CA, June 14-17, 1993.