Use Case Maps: A New Model to Bridge the Gap Between Requirements and Design

R.J.A. Buhr,

SCE 95-

Contribution to the OOPSLA 95 Use Case Map Workshop

Use Case Maps: A New Model to Bridge the Gap Between Requirements and Detailed Design

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

Abstract

Use case maps provide a new design model to bridge the gap between use cases and detailed design. This paper explains why a new model is needed, summarizes the highlights of the model as presented in a new book, and provides a vision for future development of the model.

1.0 The Main Ideas

There is an ever increasing demand on software for more of everything—functionality, flexibility, reusability, extensibility, performance, robustness, distributed operation. Increasingly therefore, programs of all kinds must be viewed as *systems* (sets of components that collaborate to achieve some common purpose). In general, understanding "how it works", and communicating it to others, is a very difficult problem for systems. This problem is excacerbated by the fact that popular techniques for implementing software can look so different in detail that a common system view can be hard to achieve. We need better design models to deal with this issue.

Use case maps [1] (refer to Figure 1) provide a first-class design model for the "how it works" aspect of both object-oriented and real time systems [3]. Use case maps give a road-map-like view of the cause-effect paths traced through a system by scenarios ("use cases" [5] are prose descriptions of scenarios). The maps have a compactness—resulting from their high-level of abstraction—that enables many scenario paths to be expressed in a single diagram in a way that reveals patterns and saves them for reuse. Use case maps have four main elements: *paths* that trace the progression of causes and effects from points where stimuli occur (usually, but not necessarily, in the environment), through the components of a system, to points where responses are felt; *responsibilities* that link paths to components; *couplings* between paths that connect them together into larger patterns; and *components* that perform responsibilities. Components include familiar ones like objects and teams of objects, but also unfamiliar ones called *slots* that may be occupied in a dynamic fashion by objects as scenarios unfold (these are *not* the same as the slots of prototype-based languages). Slots enable us to express, with fixed use case maps, the essence of structural dynamics (the creation, changing visibility, and destruction of objects as execution proceeds) at a level of abstraction above the intricate details that are used to achieve it in software.

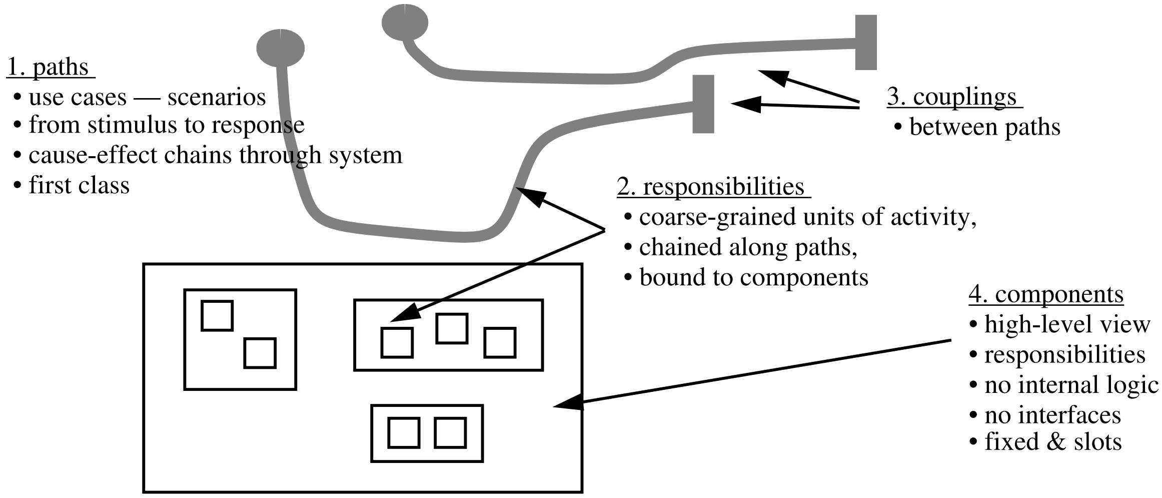

Figure 1 Four main elements of use case maps.

Use case maps offer a different view of systems from the more familiar component-centric view that is supported by popular tools (see Figure 2). Use case maps view a system in terms of causal paths that are followed through the components as a result of the occurrence of stimuli. Ultimately the segments of all such paths that traverse an individual component must be supported by that component by means such as methods, messages and state machines; this is where tools come in—to define the component and its protocols with the outside world. However, use case maps provide a complementary view, above the level of such details.

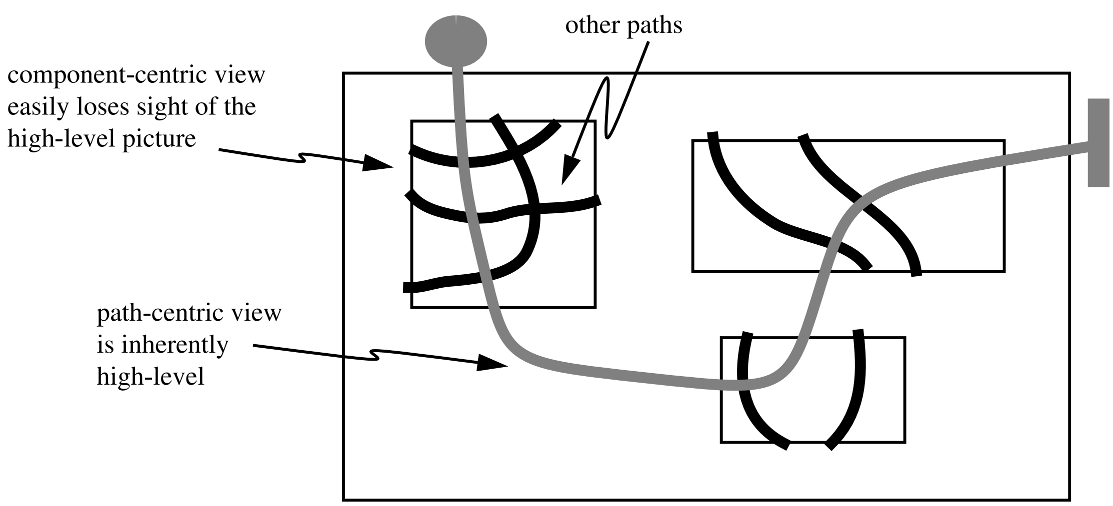

Figure 2 Use case maps provide a path-centric view of systems.

Why do we need yet another design model? Why not make do with a familiar models, such as class relationship diagrams and message sequence charts (a.k.a. interaction diagrams), that are widely used in popular textbooks, e.g., [5], [6], by stretching the models a bit to express the things we said above that use case maps express? While this is possible to a degree, we do not think it is the best use of models that already have other purposes. In general, stretching a model too far to cover too many things may not be the best use of a model.

Class relationship diagrams are routinely stretched in object-oriented practice by annotating them to indicate intended run-time relationships between instances. However, we suggest that carrying this kind of thing too far overloads a model that already has enough to express, making it more difficult to understand and maintain. Furthermore, it does not solve the problem of understanding "how it works" as a whole, which may requires mentally chaining together many relationships. We have heard of use case paths being superimposed on class relationship diagrams to indicate chaining of relationships, but think it may be better to separate concerns by separating the model of "how it works" (use case maps) from the model of "how it is constructed" (class relationship diagrams).

Message sequence charts can be stretched by reinterpreting arrows in them as indicating causality, rather than messages and reinterpreting segments of timelines as indicating responsibililties rather than message processing. However, the linear form of message sequence charts is less useful for expressing composite patterns than the use case map notation. We could superimpose arrow chains indicating causal sequences on diagrams showing components, with a result that would approximate use case maps. However, the use case map notation is simpler, less cluttered, and does not stretch an existing model (arrows indicating messages) beyond its original intent. The meaning of use case maps is so different that there is value in recognizing them as a different model with a simpler notation that is appropriate to its higher level of abstraction.

Why do we need another design model when other models are formally sufficient? Yes, *if we have other design models defined*, use case maps are formally redundant, because the information in them can be derived from the details in the other models. However, the term "redundant" should not be interpreted to mean "not needed". We need redundant models precisely *because* they are redundant. Think of the relationship between code and design models like collaboration graphs, interaction sequence diagrams, visibility graphs, and class relationship diagrams. These design models are, in a formal sense, redundant relative to code because an expert can infer them from code. However, the redundancy is useful because it provides a separate description of important issues that otherwise tend to get lost when implementing or modifying code. Yes, once you have the implementation, the models are formally redundant. No, the implementation does not replace the models, because the latter have a different purpose, namely helping humans to reason above the level of details.

We need redundant design models to improve the probability that software will be correct when it is finally put into service, to shorten the overall time required to make it correct, and to reduce the chances of introducing errors when changes are made. Few would argue that we should abandon all design models that are formally redundant relative to code. Everyone would surely agree that a balance must be struck between the time and effort needed to learn, use and maintain redundant models and the expected pay-off in terms of reduced errors and shortened overall development time. We think that this trade-off is in favour of adopting use case maps as a high-level design model.

2.0 Use Case Maps with Other Design Models

Use case maps are intended to be used in a coordinated way with other, more familiar, design models. Figure 3 sets the stage by identifying a missing model in existing suites of design models. The figure identifies four levels of abstraction (vertically), including two design levels: high-level design and detailed design. The line between high-level design and detailed design is fairly sharp. Below the line are most of the familiar, detailed techniques we mentioned above, such as message sequence charts and collaboration graphs, plus some others like visibility graphs that help us to understand systems that change form over time (i.e., are “structurally dynamic”). Detailed class relationship diagrams (ones that define methods and instance variables) also fall below this line.

The figure also identifies three domains of separate design concerns that we call *operation*, *manufacturing* and *assembly* (the names are intended to identify different design concerns, not to suggest anything about programming languages or code generation). Class relationship diagrams in the manufacturing domain describe “how it is constructed”. The operation and assembly domains describe “how it works”. The operation domain is concerned with collaborative behaviour of teams of components. The assembly domain is concerned with how the teams are assembled at run time (i.e., how the members are created and made visible to each other). At the detailed-design level, models in these domains are all separate and their combination can be quite complex.

Above this line, there is an absence of models of “how it works” to act as first-class partners of the model of “how it is constructed” provided by class relationship diagrams. This often results, in current practice, in class relationship diagrams being annnotated with relations that indicate “how it works”, such as communicates-with, creates, and manages, thus overburdening a model that is important for other purposes.

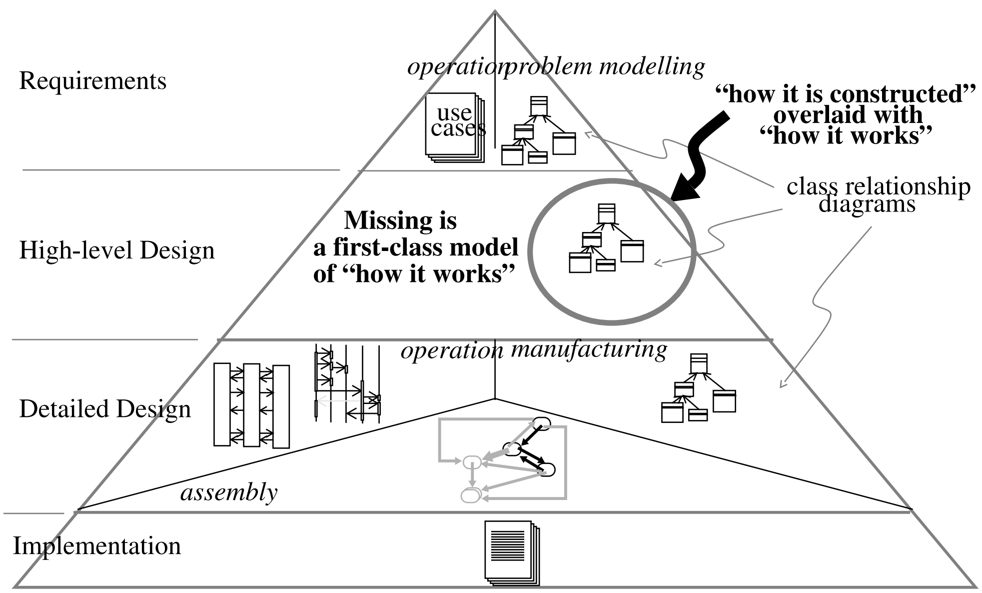

Figure 3 A missing model in existing suites of design models.

Figure 4 extends the idea of operation, manufacturing, and assembly domains into high-level design. It indicates that use case maps supply the missing model of “how it works” in a way that covers both operation and assembly aspects at this level. The “how it is constructed” model is still class relationship diagrams, but now not overloaded with other information. The two models are equal first-class models that may be developed in any order relative to each other and related when convenient. Both models may extend into requirements; they also overlap slightly in relation to assembly (because classes provide the instances that are “assembled” into collaborating teams).

Use case maps cover both the operation and assembly domains to describe “how it works” in a unique way at a high level of abstraction, as follows:

- The paths of use case maps provide both the routes for the progression of stimuli through a system of components (operation) and the loci for creating new components and moving them around to different places in the system at different times (assembly). Because the movement of components along paths is not different in kind from the progression of stimuli along paths, the model easily covers both operation and assembly in the same fixed map.
- The places to which components may move are called *slots*. Slots are boxes with dashed outlines that are analagous to positions in human organizations that exist independently of their occupancy or occupants.

The result is that we need only two models for high-level design, whether a system is structurally static or structurally dynamic. This enables us to come to grips with the “how it works” implications of the pervasive structural dynamics that often exists in object-oriented programs. This is in strong contrast to the detailed-design level, where multiple models may be needed and descriptions may become very complex in structurally dynamic cases.

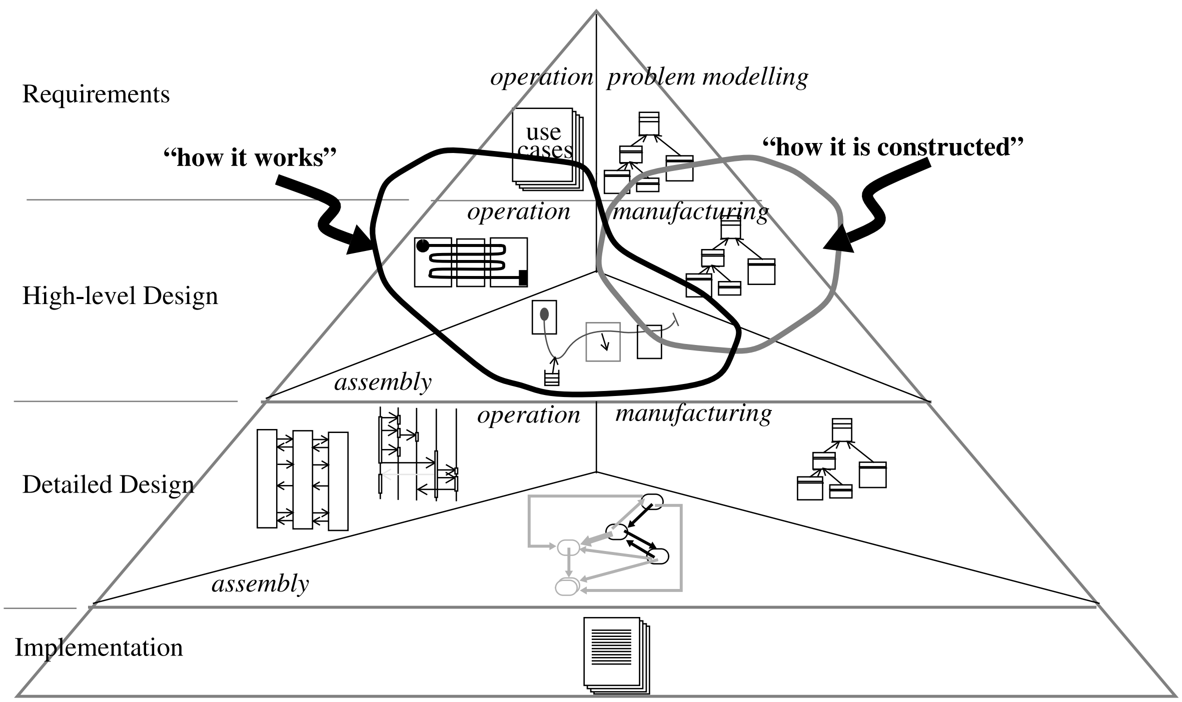

Figure 4 A context for use case maps.

3.0 Vision and Work in Progress

With reference to Figure 5, we see:

- use case map tools that support not only the creation and manipulation of maps as first-class models in their own right, but also the maintenance of relationships to other models in the same and different design levels (possibly also at multiple different levels of recursive decomposition in all models);
- tools that support opportunistic design by allowing development of use case maps in any order in relation to other models;
- formalization of both the structure and the excution semantics of maps (even though many

details are undefined, they have execution semantics); consequential support for executing and analyzing maps;

- formal techniques hidden as internal details in tools;
- support for entering timing estimates through multiple different models that may accumulated along paths to give performance estimates (to avoid having to tune the performance by changing the design after it is implemented);
- support for analyzing tradeoffs between design alternatives;
- repositories of design patterns expressed with use case maps;
- components characterized not just by the protocols or contracts required to interact with them, but also by their intended roles in relation to larger use case maps (satisfying the need to understand how some component offered by a supplier is intended to fit into a larger context);
- tools for visualizing complex systems and software in terms of use case maps, using combinations of fisheye and 3D techniques;
- semiautomatic progression from use case maps to detailed designs and code;
- semiautomatic reverse engineering of code into use case maps;
- use case map front ends for popular detailed design tools;
- and more.

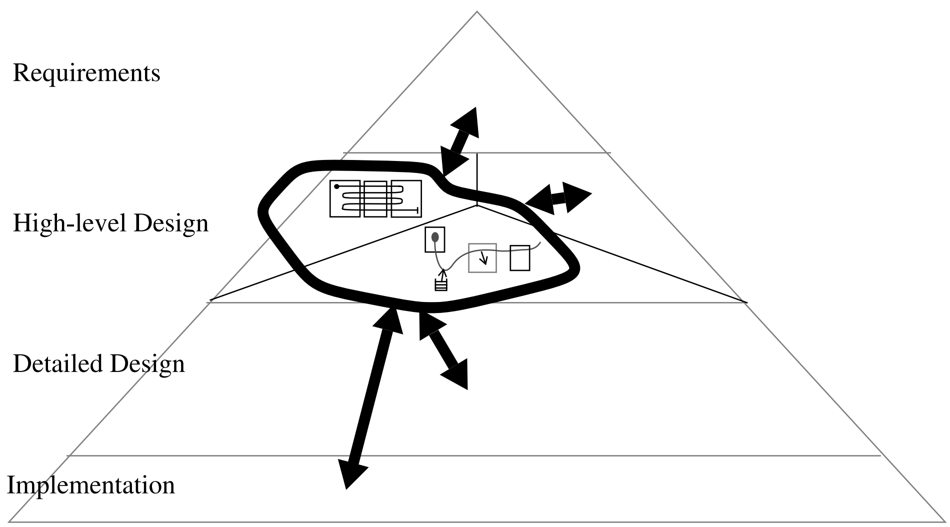

Figure 5 Use case map tools linked to other models.

Progress has been made on formalizing execution semantics of use case maps, e.g., [2], on describing design patterns with use case maps, e.g. , [1] , [4], and on fisheye and 3D visualization techniques (student theses), but much more remains to be done in all of these areas. Work has started on a use case map editor based on hypergraphs and on techniques for employing use case maps to validate detailed designs (e.g., expressed with coupled-state-machine models). Other work is still a gleam in the eye.

4.0 Conclusions

Better techniques for improving human understanding and communication about "how it works" will result in better software built more quickly with fewer design errors in it and with fewer design-violating errors creeping into it through implementation changes made during maintenance, reengineering, and evolution. This body of work offers the prospect of having a significant impact on industrial practice by providing a new way of ensuring that requirements are captured in designs and high-level design knowledge is saved and reused.

References

[1] R.J.A. Buhr, R.S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996 (available October 95), 302 pages .

[2] D. Amyot, F. Bordeleau, R.J.A. Buhr, L. Logrippo, *Formal Support for Design Techniques: a Timethreads-Lotos Approach*, FORTE 95, Montreal, October 95. (“Timethreads” is an old term that we no longer use for paths in use case maps.)

[3] R.J.A. Buhr, *Issues in Bringing Object-Orientation and Real Time Together,* contribution to the OOPSLA 95 real time workshop, Austin, TX, October, 95.

[4] R.J.A. Buhr, R.S. Casselman, T.W. Pearce, *Design Patterns with Use Case Maps: A Case Study of Reengineering an Object-Oriented Framework,* SCE Report 95-17, November 1995.

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

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