Representing structure in a software system design

Michael Jackson, Computing Department, The Open University, Milton Keynes MK7 6AA, England; jacksonma@acm.org

Structure, clearly understood and represented, is a key tool in the design of any complex artifact. Software systems are among the most complex of all human artifacts. The software designer's essential product is not the software itself, but the behaviours that it evokes in the 'problem world' outside the computer and the affordances it provides to its users. The properties of the problem world, and the users' capacities to understand and exploit the system's complex functionality, are therefore vital subjects for the designer's attention. Causality is an important concern in a realistic system, within the computer, within the problem world, and in their mutual interactions. The computer can process and manipulate information; it can therefore embody useful representations of the problem world, of its users, and even of itself, adding further levels of structural complexity. This paper discusses some aspects of structure and its representation arising from these concerns in software system design, and offers some general observations relevant to other design fields. The discussion is illustrated by the design task studied in the SPSD2010 workshop that took place in February 2010 at UC Irvine, CA, USA.

Keywords: causality, initiative, problem, representation, requirements, object-orientation, simulation, span, structure

Introducing a digital computer into a system brings an unprecedented level of behavioural complexity. Anyone who has ever written and tested a small program knows that software, even when apparently simple, can exhibit complex and surprising behaviours. To deal effectively with this complexity the program text must be structured to allow the programmer to understand clearly how the program will behave in execution (Dijkstra 1968). That is: the program's behaviour structure must be clearly represented by its textual structure.

In a system of realistic size, a further level of complexity is due to the combination and interaction of many features. For example, a modern mobile phone may provide the functions of a phone, a camera, a web browser, an email client, a GPS navigation aid, and several more. These features can interact in many ways—a photo taken by the camera can be transmitted in a phone call—and the resulting complexity demands its own clearly understood structuring if the feature interactions are to be useful and reliable. The functional structures created and understood by the designer must be intelligibly related to parallel structures perceived by the users of the system. It is failure in this user-interface structuring that explains the difficulty experienced by many users of digital TV recorders, and the multitude of puzzled customers asking on internet forums how to set the clock on their digitally-controlled ovens.

Yet another demand for clear structure comes from the world outside the computer, where the problem to be solved by the system is located. For some systems, including critical and embedded systems, the essential purpose of the system is to monitor and control this *problem world*: the purpose of an avionics system is to control an airplane in flight, and the purpose of a cruise control system is to control the speed of a car. The designers of such a system must study, structure, analyse and clearly depict the properties and behaviours of the problem world where the software execution will take its effect; they must respect and exploit those properties and behaviours in designing the software to produce the effects required.

Structure, then, is of paramount importance in software design: understanding, capturing, analysing, creating and representing structure furnish the central theme in this paper. The paper arises from a software design workshop (SPSD 2010) sponsored by the US National Science foundation in February 2010. The workshop brought together a multidisciplinary group of participants, all with an interest in the theory and practice of design in its many manifestations. The workshop focused on previously distributed video and transcription records of design sessions, each of one or two hours, in which three teams of software professionals worked separately on a given system design problem. The introduction to the special issue of *Design Studies* in which this paper appears gives a fuller account of the design problem itself and of the capture of the design teams' activities. The time allotted for the work was very short, and none of the teams could be said to have addressed the problem effectively. However, many interesting issues arose in the design sessions and were discussed in the workshop.

In this paper these issues are considered both at the level of software development generally and in the specific context of the given problem, focusing on the discovery, invention and representation of structure in a few of its many dimensions and perspectives. The paper progresses broadly from the task of understanding the initial requirement in Section 1 towards the design of the eventual software system in Section 7. Section 2 considers the designer's view of the reality to be simulated—in this case, a road traffic signalling scheme—and Section 3 discusses the simulation of this reality by objects within a computer program. Sections 4, 5, and 6 explore some general structural concerns as they arose in the problem and its solution. Section 8 discusses some broader aspects of the design case study, and comments critically on current software development practices and notations. Section 9 concludes the paper by offering some more general observations relevant to design practice in other fields.

1 *Understanding the requirement*

The problem, outlined in a two-page *design prompt*, is to design a system to simulate traffic flow in a road network. The sponsor or client is Professor E, a teacher of civil engineering, and the system's intended users are her civil engineering students. The system's purpose is to help them to achieve some understanding of the effects on traffic flow of different schemes of traffic signal timings at road intersections. The design teams were required to focus on two main design aspects: the users' interactions with the system, and the basic structure of the software program.

This traffic simulation problem is not a critical or embedded system: it is no part of the system's purpose to monitor or control traffic in the real world. Nevertheless, the system has a close relationship to the real world because it is required to simulate the behaviour of real traffic under different regimes of traffic signalling at road intersections. An obviously important criterion of a successful design is that the simulation should be sufficiently faithful to the behaviour of real traffic.

The core function of the system is to support the creation and execution of traffic simulations. Each executed simulation has the following basic elements:

- a *map*: a road layout containing several four-way intersections, all with traffic lights;
- a *scheme*: a traffic signalling scheme defining the sequence and duration of the light phases at each intersection; and
- a *load*: a traffic load on the system imposed by the statistical distributions over time of cars entering and leaving the layout.

The student users will specify maps, schemes and loads; they will use them to run simulations and observe their results in a visual display. They must be able to alter the scheme or the load and see how the change affects the traffic flow. By using the system in this way the students are expected to obtain a broad intuitive understanding that the practical engineering of traffic signal timing is a challenging subject, and at least a rough feeling for the variety of effects that a scheme can have. As Professor E points out, “This can be a very subtle matter: changing the timing at a single intersection by a couple of seconds can have far-reaching effects on the traffic in the surrounding areas. Many students find this topic quite abstract, and the system must allow them to gain understanding by experimenting with different schemes and seeing first-hand some of the patterns that govern the subject.” The understanding they have gained will prepare and motivate them for the systematic mathematical treatment in their civil engineering course on the subject of traffic management.

Understanding the requirement—that is, the problem that the software is intended to solve—is a crucial task in developing software (Jackson 2001). Kokotovich (2008) cites the investigation by Matthias (1993) of novice and expert designers in several fields. Matthias found that expert designers treat problem analysis as a fundamental early design activity, but novices typically neglect it. Major misunderstandings of the system requirement were detectable both in some of the recorded design sessions and in some of the subsequent workshop discussions.

One design team was—temporarily—misled by the mention in the design prompt of an “existing software package that provides relevant mathematical functionality such as statistical distributions, random number generators, and queuing theory”. They assumed that the purpose of the software package mentioned was to support a *mathematical analysis* of the traffic flow. The system would take a specified map, scheme and load, and derive a system of equations; by solving these equations the system would determine performance measures such as waiting times at intersections, average speeds over each road segment, the expected number of cars held up at each intersection, and so on. These results would be displayed on the computer screen; when the user changes the map, scheme or load, the effect of these changes on the performance measures would be recalculated and made visible on the screen.

In fact, the design prompt did not call for such a mathematical analysis: instead it called for a *simulation* of the traffic flow. The system would contain program objects corresponding to the parts of the real-world traffic and its environment: vehicles, roads, and signals. The properties and individual behaviours of these program objects would be specified by the user in the map, scheme and load. Execution of the system would cause these program objects to behave and interact in ways closely analogous to their real-world counterparts. The resulting traffic flows would then emerge from the interactions of the program objects just as real-world traffic flows emerge from the individual behaviours and properties of the vehicles, roads, and signals. As the simulation progresses, the flows would be displayed as a pictorial representation of cars moving along roads, the display changing over time just as real traffic flows do.

A related misunderstanding of the requirement was the assumption that the simulation should produce not only a visual representation of the light settings and the resulting traffic flow, but also numerical outputs such as average journey times. In fact, the design prompt document says nothing about numerical outputs, but emphasises visual representation throughout.

In one workshop discussion it was argued that the purpose of the system was to help the student users to find the optimal signalling scheme for a given map and a given load. This, too, was a misconception. The purpose of the software system, clearly described in the design prompt, was to help the students to develop an informal intuitive understanding by running and observing clearly visualised simulations. Finding optimal schemes would be the goal of the formal mathematical treatment that the students would encounter later, in their academic course. The system would prepare the students for this formal mathematical treatment by introducing them to the problem and stimulating their interest by animating some practical examples of the complexities it can exhibit.

2 *Road layout and traffic in the real world*

Simulations are necessarily simplifications, and the design prompt directs the system designer to consider only a simplified area of a real world road layout. Many commonly found features are eliminated. There are no one-way roads and no T-junctions; all intersections are four-way and all are controlled by traffic lights; there are no overpasses and no giratories. The part of the road layout to be considered is bounded; it is quite small, containing only a few—“six, if not more”—intersections.

Even with these layout constraints there are variations that the system must be able to handle, and some that the designer may be free to ignore. In considering the design choices here it is important to bear in mind the purpose of the system, and to consult Professor E whenever the consequences of a design choice are obscure. For example, the simplest road arrangement that satisfies the layout constraints is a rectangular grid; for many reasons the system design will be easier if more complex layouts are ignored. However, as Figure 1 shows, the layout constraints given in the design prompt are also satisfied by a rectangular grid with added diagonal roads. Whether this complication must be handled depends on a question of traffic management theory. All cycles in the rectangular grid are quadrilaterals, while the complex layout also has triangular cycles. Do these triangular cycles give rise to unique characteristic variations in traffic flow effects that the students are expected to learn about from the system? This is a question that Professor E must answer for the designer.

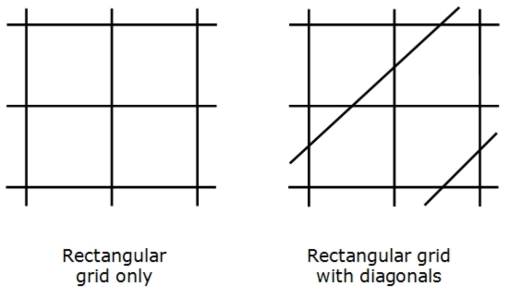

Figure 1. Possible road layouts

Another design choice concerns the way cars enter and leave the traffic flows in the layout. Since there are no one-way roads, cars can enter and leave at any point at which a road crosses the map boundary. If the map contains parking lots, it is also possible for cars to enter or leave the traffic flow at any point at which a parking lot is connected to a road. This possibility seems to introduce a large complication. The system will need to keep track of the number of cars in each parking lot, and the points at which cars enter the traffic flow on leaving the lot will constitute intersections of a new and different kind, requiring special and potentially complex treatment. This additional complication seems to add little or nothing to the value of the system for its proclaimed purpose, and all the teams wisely ignored the possibility.

The real-world traffic signal schemes that must be considered are less constrained. One or more lanes at each intersection may be equipped with sensors: the traffic signal scheme can then take account of the presence or absence of vehicles in a lane, avoiding giving pointless priority to directions in which there is no traffic. The design prompt does not stipulate whether right turn on red is permitted. It states that the system must “accommodate left-hand turns protected by left-hand green arrow lights”, but it is unclear whether unprotected left turns are permitted anywhere. It is also unclear whether ‘lane’ has its usual real-world meaning in which a wide road may have several lanes in each direction, or instead denotes one of the two possible travel directions along a stretch of road. A “variety of sequences and timing schemes” should be accommodated, but the variety is left to the designer to choose.

The design prompt does not mention large trucks or emergency vehicles, and individual vehicle behaviour is implicitly limited to normal lawful behaviour of passenger cars. The design prompt says nothing about speed limits, and the designer may perhaps assume that a uniform speed limit applies everywhere. Cars do not break down or collide; their drivers always obey the traffic signals and never block an intersection. No road segments or lanes are closed for repairs.

3 *Simulation of real-world traffic by programming objects*

The physical elements of a real-world traffic system, and their relationships, are shown in Figure 2. The diagram shows only the physical elements of the real-world traffic, represented by the symbols, and the interfaces at which they interact, represented by the connecting lines. It does not represent such abstract things as signalling schemes or the drivers’ planned journeys through the layout.

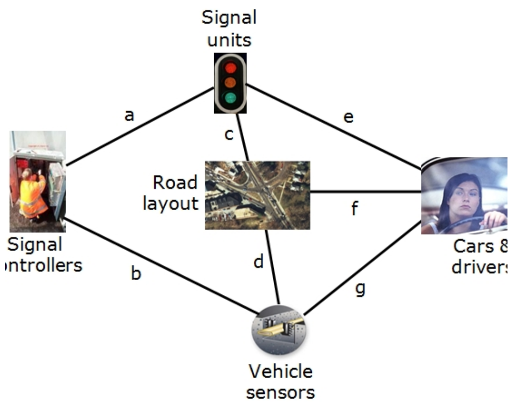

Figure 2. Real-world traffic elements and interactions

Vehicle sensors and signal units are permanently located (c and d) at positions in the road layout. The vehicle drivers see and obey (e) the traffic signals; they move into road space only if it is not occupied (f) by another car; and their movements affect (g) the states of any vehicle sensor they encounter. The sequencing and timing of the signal lights is controlled (a) by one or more signal controllers, which read and take account of (b) the states of any relevant vehicle sensors. It is the behaviour of such real-world traffic systems that is to be simulated by an executable computer program.

All the design teams assumed—realistically, in today’s software development culture—that the executable simulation is to be programmed in an object-oriented language such as Java. So a significant part of the design task is to decide what programming objects should participate in the simulation and how they should interact. In effect, the properties and behaviours of these objects when the simulation is executed will constitute the designer’s chosen model of the properties and behaviours of the simulated reality.

Starting from the depiction in Figure 2, it seems initially that the necessary objects are: one for each car; one for each signal unit; one for each sensor; one for the road layout; and one for each signal controller. However, while it is reasonably clear that one object is needed to represent each car and one for each sensor, it is not yet clear how many individual objects will be needed to represent the road layout, the signal units, and the signal controllers.

The road layout is made up of roads which cross each other at intersections, so we may initially expect to have one object for each road and one for each intersection. These objects must be associated so that each intersection object indicates which roads cross there and each road indicates the sequence of intersections encountered along the road. The signal units might be represented in several different ways. For example: one object for each lamp, one for each set of lamps physically contained in the same casing, or one object for all the casings located at one intersection. The signal controllers too might be represented in several different ways, ranging from one controller for the whole layout to one for each set of lamps or even for each lamp.

We will return later to these more detailed concerns about the numbers of objects of the various classes. Broader concerns demand attention first. They reflect potent sources of confusion and difficulty that were dramatically obvious in some of the recorded design sessions.

4 *Initiative and causality*

The five element classes shown in Figure 2 interact to give the overall behaviour of the real-world traffic. Where does the initiative for this behaviour come from? Which elements cause the behaviour, directly or indirectly, and how?

Initiative in the real world comes from only two sources: the signal controllers and the car drivers. First, the signal controllers. They take the initiative, according to their designed control regime and, perhaps, the states of the vehicle sensors, in changing the visible states of the lamps in the signal units. Even if there were no cars on the roads the signals would, in general, go through their sequences as commanded by the controllers. Second, the car drivers. Each driver is engaged in a purposeful journey from one place to another, driving the car in order to progress along a chosen route at chosen speeds. Progress is constrained by traffic laws that limit the car’s permissible road positions and speeds, by the need to avoid collision with other cars, and by obedience to the signals.

There are no other sources of initiative. The road layout, signal units, and sensors all change their states as a result of the initiatives taken by the signal controllers and the drivers, but they are entirely passive: they take no initiative on their own account. A part of the road may or may not be currently occupied by a car, but the road itself can neither import nor expel a car. A sensor may be on or off, but it can neither permit nor block a car’s movement. Up to an acceptable approximation, the executed simulation must reflect this behaviour faithfully. It will be harder to achieve a sufficient degree of fidelity if the simulation design deviates gratuitously from the behaviour of the real-world traffic.

Setting aside the internal object structures of the signal controllers, the signal units and the road layout, Figure 3 shows an initial software structure for the simulation. The solid rectangles and the solid lines connecting them directly represent the software objects and interactions that model the real-world traffic elements shown in Figure 2. However, these five object classes are not enough to run the simulation, and additional objects are needed.

The *arrivals model* objects, represented by the smaller dotted rectangle, will certainly be needed. Because the road layout is bounded, it is necessary to consider the arrivals and departures of cars: cars arriving from outside the boundary will appear to the simulation to be newly created, and on departure will appear to have been destroyed. The software function of creating a car object cannot be a function of the car itself, so it is necessary to introduce one or more objects to create cars. These objects are represented by the arrivals model object in Figure 3. Like the signals model, arrivals are multiple objects, one located (v) at each entry point to the layout and creating (w) new car model objects there according to statistical distributions representing the frequency of arrivals at that point.

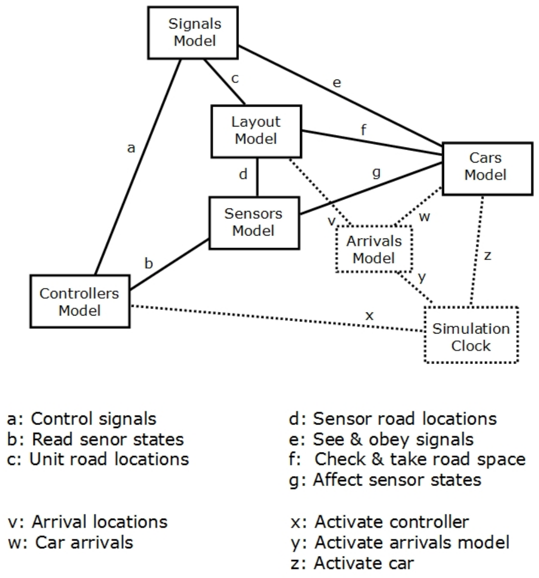

Figure 3. Synchronous simulation elements and interactions

The *simulation clock* object, represented by the larger dotted rectangle, may or may not be needed, depending on a technical design choice. Given the arrivals objects, the simulation could be implemented by a *single-threaded* or a *multi-threaded* design. In a *multi-threaded* design, each independently active object has its own *thread of control*: that is, it behaves as if it were running on an independent processor, concurrently with all other objects, and synchronising its behaviour with other objects only when it interacts with them. The simulation depicted in Figure 3 could then be implemented by the five solid-line objects with the sole addition of the arrivals object. Independent control threads would be provided for each signal controller model, each arrivals model, and each newly created car. This would be an *asynchronous* design for running the simulation. The passage of time—for example, to distribute arrivals over time, to delimit signal phases or to maintain a car's desired constant speed along a road segment—would be represented by *wait* statements in the programmed behaviours of the arrivals objects, signal controllers and cars respectively.

The concurrent behaviours in a multi-threaded design offer many opportunities for subtle programming errors that are hard to avoid and hard to diagnose and eradicate: their effects are difficult to reproduce in program testing. There may also be efficiency penalties in execution. For these reasons, a single-threaded *synchronous* design may be preferred, in which only one object can be executing at a time. The single thread design uses the simulation clock object represented by the larger dotted rectangle. This clock object would be the only independently active object in the simulation. It would emit (x, y and z) periodic ticks to activate the controller objects, arrivals objects and car objects, each one progressing in its simulated lifetime as far as it can progress in the short time associated with the tick. One controller, for example, might count down one tick while it waits for the next signal change to become due, while another makes a signal change that became due immediately after the preceding tick. One arrivals object might make a randomly determined choice to create a new car, while another makes a randomly determined choice to do nothing. One car might continue to wait at a stop signal while another makes a unit of progress along a road.

The key point in such a design is that the simulation clock does nothing except mark the progress of time in units of one tick, for each tick briefly activating each arrivals, cars, and controllers object. It would be a fundamental confusion, therefore, to think of the simulation clock as a kind of master controller, like a global policeman determining the behaviour of every individual part of the simulation. What each controller, arrivals, or car object does in each tick should be determined purely by its own current state, its defined behaviour, and its interactions with neighbouring objects. The simulation clock is merely a technical software implementation mechanism to represent the passage of global time while saving the designer from the well-known programming pitfalls of asynchrony and concurrency in a multi-threaded system.

5 *Causality and determination*

The most notable confusion about initiative and causality displayed in the recorded design sessions centred on the movement of cars in the simulation. In one session a designer suggested that when a signal changes to green the intersection "asks its roads to feed certain numbers of left-turn, right-turn and straight-ahead cars." This surprising suggestion almost exactly reverses the direction of causality in the real-world traffic: it makes the intersection and the road into active objects, while the cars become passive objects, moved by the road's action, which in turn is caused by the intersection.

A related confusion about the movement of cars concerns the path to be taken by each car when it encounters an intersection. This confusion permeated many of the recorded design sessions, and also appeared in discussion sessions in some workshop groups. The confused idea was to regard each incoming road at each intersection as having the property that a certain proportion of the cars arriving at the intersection would turn left, a certain proportion would turn right, and the remainder would continue straight ahead. The confusion here is quite subtle. It is, of course, true that empirical observation of the traffic in a real-world road system will allow these proportions to be measured and recorded. Traffic engineers do make such measurements and use them to determine efficient signal timings. For example, if most northbound and southbound vehicles at a particular intersection continue straight ahead or turn right, it would be obviously inefficient to provide a lengthy phase for protected left turns. The confusion lies in treating these proportions as a *determining cause* of the vehicles' behaviours, when in fact they are an *emergent effect* of the routes the drivers are following. Treating the effect as if it were a cause will give rise to senseless journeys by fragmenting each journey into a series of unrelated choices at successive intersections: in effect each journey becomes a random walk through the road layout. If real-world traffic patterns are to be simulated with a reasonable degree of fidelity, this confusion must be avoided.

A deeper philosophical point is exposed here about object-oriented programming. It was not by chance that the first usable object-oriented programming language (Dahl et al 1970, 1972) was named Simula: it was a language designed for the programming of discrete behavioural simulations. Objects in Simula possessed defined behaviours, independent but interacting, and the goal of the simulation was to reveal the properties that emerged from their interaction. To an important extent the original purpose of such a language is perverted when it is used in design to proceed in the reverse direction—starting from given required global behaviours and properties of a system to derive the interacting behaviours and properties of the objects from which these global properties are required to emerge. In such a design process, the system is decomposed into objects with independent behaviours and properties: but it is then hard to ensure that the global results that emerge will be what is required.

For designing the traffic simulation considered here an object-oriented programming language is an excellent choice: the signal controllers and the cars have given behaviours, and the system's explicit purpose is to reveal the emergent properties of the traffic flow.

6 Description span

The problem of describing the behaviour of individual cars reveals another fundamental and ubiquitous structural concern. In designing, describing or analysing any subject of interest it is necessary to decide on the *span* of the description. The span of a description (Jackson 1995) is the extent that it describes of its subject matter, the extent being defined in time, in space, or in the size of a subset. For example: a map of Europe has a smaller span than a map of the world; a history of the Second World War has a smaller span than a history of the Twentieth Century; a line in a text has a smaller span than a paragraph; a day has a smaller span than a year; and a study of automobiles has a smaller span than a study of road vehicles. The penalty for choosing too small a span for the purpose in hand is that the subject matter of interest is then described only by a number of fragments: obtaining an understanding of the whole from these fragments is difficult and error-prone. In software development, many unnecessary difficulties arise from the choice of an insufficient description span.

The initial focus of two of the three teams provided an early illustration of the consequences of an ill-chosen span: they began their discussion of the problem by considering just one intersection. This probably seemed natural because it is primarily at intersections that traffic control is exercised, and certainly there are aspects of the problem for which one intersection is the appropriate span for discussion and description. However, after considering one intersection, both teams eventually turned their attention to the question: How can departures from one intersection be related to arrivals at a neighbouring intersection? The focus on individual intersections now became a serious disadvantage: the rules were clearly going to become very complicated. It took a little time before the teams realised that they must consider the question in the larger span of a whole road or even the whole layout.

Applying the notion of span to describing the behaviour of an individual car as it encounters successive intersections, it becomes immediately clear that the appropriate span is the car's complete purposeful journey across the bounded layout. Nothing else can adequately represent the behaviour of real cars and drivers. (Of course, in the real world there are exceptions such as learner drivers practising left turns, joy riding teenagers, and mechanics road testing a repaired car: the purpose of their journeys is not to reach a destination. Such exceptions can be ignored in the simulation. They may be assumed to be relatively rare, and since these drivers have no desired destination their efficient and convenient movement is not a matter of significant concern to traffic engineers.) The turning behaviour of a car should therefore be determined by a route assigned to it when it was created by the arrivals model—this route, like the car creation itself, drawn from some statistical distribution defined by the users. Such a design corresponds well to the reality that traffic loads commonly result from such purposeful journeys as commuting between home and work, children's school runs, shopping expeditions, and visits to entertainment centres.

Another example of a span concern is the design of the object structure for the signal controllers. In an earlier section it was suggested that the signal controllers might be represented in several different ways, ranging from one controller for the whole layout to one for each set of lamps or even for each lamp. The choice must be governed by the span of control to be exercised. Evidently, all the signals at one intersection must be brought under unified control. Without such a unified control it would be very hard to ensure, for example, that the signals are never simultaneously green for conflicting flows, and the software to control the signals at a single intersection would become a complex distributed system in its own right. Similarly, if the signals at the intersections along one main road are to be synchronised to present a steady uninterrupted sequence of green signals to cars travelling just below the maximum permitted speed, then some signal control must span the whole road. Signal control objects at different spans will be necessary, interacting in a structure of multiple objects corresponding in total to a single signal control for the whole layout.

Regardless of the synchronisation of signals at multiple intersections, the representation of signal states and sequences at a single intersection raises its own issue of description span. As the design prompt stipulates, “combinations of individual signals that would result in crashes should not be allowed.” Intuitively, this means that only certain combinations of traffic flows can be permitted at any one time, and the allowed signal settings must correspond to these combinations only. For example: “left-turning northbound traffic may not be permitted in combination with straight-ahead southbound traffic;” or “east-west traffic may not be permitted in combination with north-south traffic.” Each permissible combination can be captured by enumerating the associated states of individual signals, and one of the design teams explored representations of signalling sequences in exactly these terms, using lines of coloured dots or stripes to represent the states of individual signals.

A description of larger span for each combination at each intersection could be expressed in terms of permitted traffic flows rather than of individual signals, leaving the individual lamp settings in each case to be implied by the permitted flows. This span leads to a more compact, direct and convenient representation by symbols in a style of the kind shown in Figure 4. Their meanings are immediately obvious, and the complete set of permissible symbols—not all of which are shown in Figure 4—directly represents the set of permissible combinations of signal states.

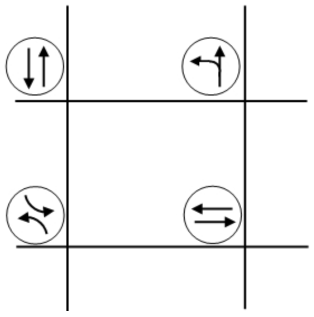

Figure 4. Representing signal states at intersections

The symbols can be used to good effect in design to represent signal sequencing and timing at each intersection, as shown in a state machine diagram in Figure 5.

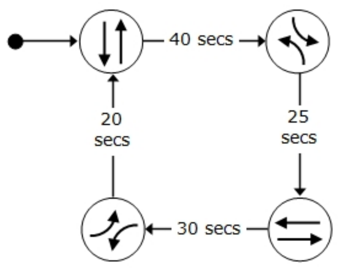

Figure 5. Representing signal sequencing and timing

Further symbols can represent transitional phases in which all lights are red, and no flow is allowed. State machines of this kind could play a useful role in the users’ specification of signalling schemes. The symbols could also be used in the visualisation of the traffic light states, as suggested by Figure 4. Although the users’ own driving experience has accustomed them to recognising configurations of coloured lights as they approach them along the road at ground level, the suggested symbols are more immediately understandable when seen, as it were, from the bird's eye view that is more appropriate to users of the system watching a simulation on the screen. Each symbol directly captures in a single glyph all the currently permitted and forbidden traffic flows at the intersection.

As a general rule a designer who is in doubt should choose a larger rather than a smaller description span. The consequence of choosing an unnecessarily large span is small: the subject matter naturally fragments itself into mutually independent parts among which no relevant relationships or interactions are found. The consequence of an inappropriately small span is large: it is gratuitous complexity with all its attendant difficulties.

7 *Forming simulations*

The student users must define, run and observe many simulations if they are to acquire the desired intuitive understanding of traffic control. To stimulate this understanding how should these simulations be formed? What variations should be supported? How should the various simulations be related to each other? As briefly discussed in Section 1, a single simulation run requires a road layout map, a signalling scheme, and a traffic load. The users must be able to save maps, schemes and loads for later use in repeated or varied simulation runs, and recombine them to observe the effects of different combinations.

One way of structuring the basic elements of each simulation into parts that can be usefully combined is shown in Figure 6:

- the *simulation infrastructure* part is common to all simulations executed by the system, and consists only of the clock object;
- the layout *map* part consists of the layout model and sensors objects;
- the signalling *scheme* part consists of the controllers and signals objects; and
- the traffic *load* part consists of the arrivals and cars objects.

The objects belonging to the traffic load part are positioned (v,f) at specific locations in the layout map; the signalling scheme objects, too, are positioned (a,b) at specific locations. Each load and scheme can therefore be combined only with maps that provide exactly those locations.

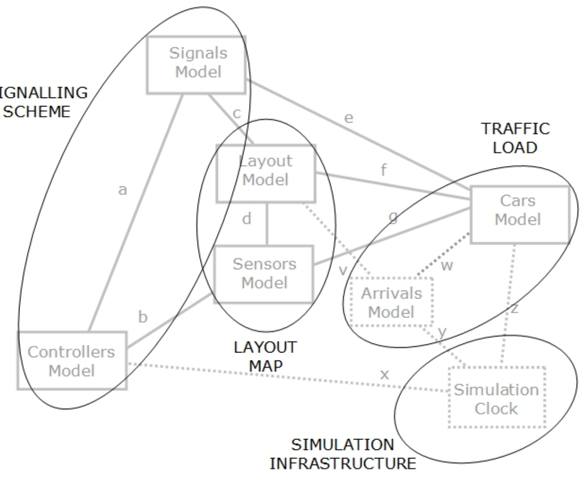

Figure 6. Simulation parts for combinations

Careful software design will allow the relationships e and g to be ignored for purposes of combining simulation parts. Intuitively, the signals that a real-world driver can see and a simulated driver must obey (e) are determined by the driver's location (f) in the layout. In the simulation, therefore, the cars model objects need not communicate directly with the signals model objects. This freedom will make it easier to exploit the treatment of signalling states proposed in the preceding section and illustrated in Figure 4. In the same way, the cars model objects need not communicate directly with the sensors model objects in the simulation. A traffic load part is therefore compatible with any map providing the positions required by the relationships (a, b, v, f), regardless of the presence or absence of sensors at those layout positions.

The ease which simulation parts can be defined and combined to form simulations will play a large part in the convenience and attraction of the system. As one of the teams expressed it, users should be able “to create something without fussing with it too much.” There are many design concerns to be examined here, several of them focused on the dynamics of user interaction at more than one level. For example, defining a signal control scheme will be easier if the system provides what one team called ‘cyclable defaults’. Default local schemes are defined, each with the span of one intersection, and can then be reproduced at as many additional intersections as desired, the user switching between different defaults for different groups of selected intersections. At a larger span, simulation parts can be conveniently defined by modification in the context of a particular simulation. Having run a simulation and observed its effects, the user can modify the scheme or load, defining and storing a new scheme or load for immediate use in the next simulation. In this way both the definition of parts and the comparison of their combined effects can be easy and convenient.

8 *Software system design*

Discrete complexity should be a major determinant of design decisions for software systems. The designer must address the inevitable complexity arising from non-negotiable requirements, and must also aim to minimise additional system complexity by avoiding it wherever reasonably possible. The traffic simulation problem demands decisions where avoidance of complexity may be crucial.

One example is the definition of routes for individual cars. A possible way of defining a route that starts at a given entry point to the map is simply to specify its exit point. The simplest route has its entry and exit at opposite ends of the same road, and no turns are necessary. Slightly less simple is a route which leaves by a road perpendicular to its entry. At least one turn is necessary; if there are intersections at which left turn is altogether prohibited then more turns may be necessary. Even less simple is a route which leaves by a road parallel to its entry, allowing more than one choice of turns for crossing from the entry to the exit road. Several design questions arise. Is the car’s route fully predetermined when the car is created on arrival, or does the car model object execute an algorithm to choose appropriate intersections as turning points? In either case, how can routes be easily worked out if the map is not a simple rectangular grid? In a map in which some left turns are altogether prohibited, may there not be some pairs of entry and exit points between which there is no possible route? These issues must be considered—and addressed or circumvented—to avoid a growth of complexity that could produce an unreliable system, break the project’s resource budget, or both.

An important example of circumventing difficult issues concerns the meaning of ‘lane’ in the map. In real-world traffic the meaning is clear. Wide roads have two or more lanes in each direction. At intersections cars are often funnelled into lanes dedicated to straight-ahead travel or to left or right turns. The signals at an intersection may have a phase of the kind represented by the lower left symbol in Figure 4, in which the left-turning traffic lanes in opposite directions proceed simultaneously while traffic in all other lanes must wait. A driver intending to negotiate such an intersection must be careful to arrive in the appropriate lane. Changing lane between intersections is an unavoidable manoeuvre for many journeys, and is often hard to achieve in dense traffic: it may involve an interaction between the driver changing lanes and another who courteously gives way. This real-world behaviour is hard to formalise and therefore hard to simulate.

The design question, then, is whether this aspect of traffic movement can be neglected, and, if not, how it is to be simulated. One possible option is to circumvent the difficulty altogether, by supposing each road to have only one lane in each direction. Each lane at an intersection necessarily contains both those cars intending to turn left and those intending to turn right or proceed straight ahead. To prevent cars with different turning intentions from blocking each other, signal phases of the kind represented at the lower left in Figure 4 must never occur at any intersection: left turns are permitted only in phases of the kind represented by the symbol in the top right of Figure 4. This design choice ensures that straight-ahead travel is never permitted when left-turns are forbidden, and *vice versa*. The result is that in every signalling phase at an intersection either every vehicle in a lane can proceed or none can proceed. Whether this option is an acceptable solution to the difficulty depends on how far it damages the fidelity of the simulation. Professor E must be consulted once again.

Software system design activity is subject to characteristic distortions of technique. One distortion arises from the widespread tendency to focus attention on the software artifact itself, most often narrowly conceived as an object-oriented computer program. The problem world and the users are then relegated to a secondary position in which they attract no explicit attention but are considered—if at all—only implicitly and indirectly. This distortion was conspicuous in some of the recorded design sessions. It was especially notable that the teams rarely, if ever, discussed the behaviour of real-world traffic: for the most part, if any of the designers thought about it explicitly they did so only dimly through the distorting lens of programming objects.

To a large extent this failure to pay sufficient attention to the problem world is engendered and encouraged by a confusion between things in the problem world and the programming objects that serve to model them. In theory the software developer should consider these two subjects separately, and should also consider explicitly the respects in which each has properties and behaviours that are not reflected by the other. This is a counsel of perfection: in practice most developers consider only the programming objects directly. The UML notations commonly used by most software developers were derived from notations for fragmented descriptions of object-oriented program code. In many respects they are ill-suited to describe the problem world, but most software developers have few others in their toolbox.

The most important single factor in design is relevant previous experience. The three design teams were drawn from companies whose products are very different from the kind of system the teams were asked to design. Inadequacies in their design work were attributable in part to the very short allotted time for the design work, and in part to the artificial conditions of the design sessions; but the teams' evident unfamiliarity with this kind of simulation played a very large part indeed. In the term used by Vincenti (1993) they were engaged in *radical design*:

> "In radical design, how the device should be arranged or even how it works is largely unknown. The designer has never seen such a device before and has no presumption of success. The problem is to design something that will function well enough to warrant further development."

Much—too much—software design is radical design in this sense. One team recognised their situation quite explicitly. They saw their design as a prototype at best. Later versions would have improved features motivated by a better understanding of the requirements to be acquired from users' experience of the system and from continuing discussion with the customer, Professor E. This recognition is more profound than it may first appear. Dependable success in designing software systems can only be a product of specialised experience and the attendant evolution of a discipline of *normal design*. In normal design, in Vincenti's words:

> "... the engineer knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task."

In software development, as in the established engineering disciplines, the existence of firmly established normal design is the essential prerequisite of dependable success. Many software developers hanker after the excitement of daring innovation when their customer would have preferred the comforts of reliable competence based on experience.

9 *Some general observations*

A realistic software system has more complexity than any other kind of design artifact. The complexity arises from the essential purpose of the software: to automate the multiple system features, and their interactions with each other and with the heterogeneous parts of the problem world. An established normal design for artifacts of a particular class embodies a design community's evolved knowledge of the structures by which these complex interactions can be brought under intellectual control.

Although the highest degrees of discrete complexity are uniquely associated with software systems, complexity of interactions more generally seems to be a common theme in many design fields. Certainly this kind of complexity in architecture was recognised by Christopher Alexander (1979):

> "It is possible to make buildings by stringing together patterns, in a rather loose way. A building made like this is an assembly of patterns. It is not dense. It is not profound. But it is also possible to put patterns together in such a way that many patterns overlap in the same physical space: the building is very dense; it has many meanings captured in a small space; and through this density it becomes profound."

Alexander writes here of aesthetic qualities of density and profundity, but this kind of composition—"many meanings captured in a small space"—is also of the greatest practical importance in engineering. The introduction of the tubeless tyre was a major advance in the design of road vehicles. Previously, a tyre consisted of an outer cover providing the interface with the road surface and an inner inflatable tube to absorb shocks; the tubeless tyre combined the two in one component providing both functions. Another example from automobile engineering is the unitary body, acting both as the framework to which the engine, transmission, driving controls, suspension and running gear are attached, and as the coachwork providing seating and weather protection for the driver and passengers.

Even in quite simple design tasks this need to combine different functions in one unified design—though not necessarily in one physical component—plays a major part. A teapot must be easily filled; it must keep the tea hot, and hold it without leaking; it must stand stably on a flat horizontal surface; it must be easy to pick up by the handle and must pour without dripping; the handle must not become too hot to hold comfortably; the teapot must be easy to carry and easy to clean; it must have an attractive appearance. Each function considered in isolation is easily achieved. The virtue of a classic teapot design—that is, a normal design—is to combine all these functions in a way that reliably serves the needs of a significant class of teapot users.

In any design task there must be some combination of radical and normal design. An entirely normal design demands no design work whatsoever: it is merely an exact copy of an existing design. Realistic design work can never be completely normal: even a modest difference in the scale of an artifact can reveal otherwise unsuspected functional factors. In typography the 8-point version of a typeface is not simply a copy of the 12-point version reduced in size, and the teapot designer should not expect the 0.75 litre version to be an exact copy of the 1 litre teapot. At the other end of the spectrum that runs from radical to normal, there can be no entirely radical design. A designer must bring some relevant experience and knowledge to the task: this is necessarily experience and knowledge of existing designs of similar artifacts. An essential aspect of tackling any design problem and its difficulties is to relate it to similar problems already known.

The traffic simulation problem was of a kind unfamiliar to all three design teams. All three quite rightly tried immediately to assimilate the problem to something they already knew. Two teams took the view that the problem was an instance of the Model-View-Controller (MVC) software pattern; the third identified the problem as ‘like a drawing program’. Unfortunately, these hasty classifications were inadequate; but in every case they were accepted uncritically and never explicitly questioned. By locking themselves into these first assumptions, the teams determined the direction, perspective and content of most of their subsequent work. Interestingly, although the MVC software pattern is used in drawing programs, the focus of the third team’s work was very different from the first two. Perhaps motivated by the word ‘drawing’, the third team focused on the user interface for specifying the road map and the signalling scheme, while the first two teams focused on the control of execution during the simulation. Prior knowledge and experience was immediately translated into preconceptions about the design problem and its solution. No team devoted a balanced amount of time and effort to the two design aspects called for by the design prompt: the user interface and the software structure.

This readiness to fix preconceptions is strengthened by a designer’s focus on the artifact that seems to be the obviously direct product of the design work. In software design this means focusing on the software itself—the program texts and the program execution within the computer—at the expense of the problem world and the customer’s requirements. In design more generally it means focusing on the object being directly designed rather than on the experience and affordances that it will provide to its users in the environment for which it is intended. The result is likely to be very unsatisfactory. Donald Norman (1988) presents many classic cases, including the ubiquitous designs for doors that offer no clue to whether they should be opened by pushing, pulling or sliding.

The true product of design is not the designed artifact itself: the true product is the change the artifact brings about in the world, and especially the experience it eventually provides to its users. A fundamental factor for design success is therefore a good understanding of what is expected from this experience. Conveying this understanding is a crucial purpose of a statement of requirements, but it may be difficult to convey this understanding by a written document alone. The design prompt document for the traffic simulation system was skilfully written; yet every team misunderstood some important aspect of the system’s intended purpose. Unsurprisingly, the team that focused on the user interface reached a better understanding than the two teams that focused on the internal mechanisms for executing the simulation. Focusing on the user interface led them naturally to think about how the users would experience the system, and to consider how to make that experience more convenient.

A good understanding of the system’s purpose should inform the design choices, especially those concerned with modelling the road layout and traffic behaviour. The designer must balance costs and benefits. A more elaborate model of the road layout and traffic behaviour would allow simulations to correspond more closely to the complexities of real-world traffic.

So each complication—diagonal roads, vehicle breakdowns, parking lots, complex car routings, realistic treatment of left-turn lanes—brings a benefit. But it also brings many costs. Designing and programming the software will be more expensive. The program text will be larger, and the additional complexities may damage the system by increasing the likelihood of execution failures. More elaborate simulations will also demand more elaborate specification of the maps, schemes and loads, making the system harder to learn and more cumbersome to use. The system's clearly intended purpose is to convey an intuitive understanding: as the design prompt says, "This program is not meant to be an exact, scientific simulation." Most of the additional complications, then, offer little or no benefit to set against their costs. Adapting the words of Antoine de Saint-Exupery, the design of this system will be complete, not when there is nothing left to add, but when there is nothing left to take away.

Acknowledgments

The workshop "Studying Professional Software Design" (SPSD 2010) was partially supported by NSF grant CCF-0845840, this support including the travel costs of the author of this paper. The three design teams, from Adobe Systems Incorporated, Amberpoint and Intuit, provided the indispensable initial material for the workshop by working on the design problem under intense time pressure and in the glare of a video camera. They generously permitted the records of their efforts to be transcribed, and viewed and analysed by the workshop participants before the workshop began. The present paper has benefited from comments on an earlier version by the three reviewers and by the editors of this special issue of *Design Studies*.

References

Alexander, Christopher (1979) *The Timeless Way of Building*; Oxford University Press, 1979.

Dahl, O-J, Myhrhaug Bjorn, and Nygaard, Kristen (1970) *The Simula67 Common Base Language*; Technical report, Oslo, Norway, October 1970.

Dahl, O-J, Dijkstra, E W and Hoare, C A R (1972) *Structured Programming*; Academic Press, 1972.

Dijkstra, E W (1968) *A Case against the GO TO Statement*; EWD215, retitled (*Go To Statement Considered Harmful*) and published as a letter to the editor of Communications of the ACM Volume 11 Number 3 pages 147-148, March 1968.

Jackson, Michael (1995) *Software Requirements & Specifications: A Lexicon of Practice, Principles, and Prejudices*; Addison-Wesley, 1995.

Jackson, Michael (2001) *Problem Frames: Analysing and Structuring Software Development Problems;* Addison-Wesley, 2001.

Kokotovich, Vasilije (2008) *Problem Analysis and Thinking Tools*; Elsevier Design Studies, Volume 29 Number 1, pages 49-69, January 2008.

Mathias, J R (1993) *A study of the problem solving strategies used by expert and novice designers: an empirical study of non-hierarchical mind mapping*; PhD Thesis, University of Aston, Birmingham, UK, 1993.

Norman, Donald A (1988) *The Psychology of Everyday Things*; Basic Books, 1988.

SPSD (2010) International Workshop “Studying Professional Software Design”, February 8th–10th 2010, University of California, Irvine, CA, USA; http://www.ics.uci.edu/design-workshop.

Vincenti, Walter G (1993) *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*; The Johns Hopkins University Press, Baltimore, paperback edition, 1993.

Expanded version of a contribution to the NSF SPSD2010 workshop held in Irvine CA; in special issue of *Design Studies*, Marian Petre and André van der Hoek eds, 2010.