Studying Professional Software Designers and their Use of Abstraction

Joanne M. Atlee and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON CANADA
{jmatlee, migod}@uwaterloo.ca

***Abstract*—In this paper, we study how three pairs of professional software developers use abstraction in the course of a two-hour design exercise. We devise a scheme for classifying abstractions according domain (e.g., problem domain vs. user-interface domain vs. computer-science domain), which enables us to better compare the developers' different uses of abstraction. We also examine how focusing on a particular domain (e.g., how the real-world really operates, or the formal definitions of computer-science concepts) sometimes hinders the developers' ability or willingness to abstract from those concepts.**

I. INTRODUCTION

Is abstraction the key to computing? This is the title of Jeff Kramer's position paper in *Communications of the ACM*, in which he asks "Why is it that some software engineers and computer scientists are able to produce clear, elegant designs and programs, while others cannot?" [2].

An **abstraction** is a model or representation that omits some details so that it can focus on other details. The definition is intentionally vague about which details are left out of a model because different abstractions, built for different purposes, omit different kinds of details. For example, the architectural blueprints for a building comprise multiple plans, each one focusing on the design details that are pertinent to a particular professional: floor plans depict the skeletal structure of the building, the locations of walls and support beams, and the configurations of rooms; electrical plans map out the electrical circuits, locations of outlets, and the amperage of circuit breakers; plumbing plans similarly show the layout the water pipes and locations of faucets and sinks; heating and cooling plans show the layout of air ducts and locations of vents and heating and cooling units; and so on.

The use of abstraction is promoted as a fundamental design principle in software engineering [5]. Abstraction is critical in requirements specifications and software designs to produce early design artifacts that do not over-constrain or bias subsequent software design or development. *Information hiding* [4] uses abstraction to support software maintenance: each *module* encapsulates a separate design decision that could be changed in the future, and *interface specifications* describe the module in terms of its externally visible properties, avoiding reference to the module's internal design details. *Abstract data types* [3] and *object-oriented programming* support programmer-defined abstractions (separating out details of data representation and implementation of operations) via programming methodology and programming-language constructs, respectively. *Libraries* and *frameworks* allow programmers to extend pre-built functionality and design infrastructure to create customized solutions out of abstract building blocks. *Aspect-orientation* [1] is an attempt to encapsulate and abstract distinct concerns that cross-cut an application. And so on.

As participants of an NSF-sponsored International Workshop on Studying Professional Software Developers, we explore the use of abstraction by professional software developers. The goal of the workshop was to "collect a foundational set of observations and insights into software design" drawing on "the analysis of a common data set" by researchers from "a variety of research disciplines." [6]. The common data set consists of videos and transcripts of three pairs of professional software developers as they work on a common design exercise. The developers were given a two-page problem statement, including broad requirements, for a program that simulates traffic: cars flow through a user-specified network of roads and intersections, whose traffic throughput is controlled by user-specified traffic lights. Each developer team had two hours to produce a conceptual design of the simulation tool, including user interface and basic code structure. Their designs were to be sufficiently complete to be handed over to an implementation team.

In this paper, we present our analysis of these videos and transcripts, focusing on the developers' use of abstraction. We itemize and categorize the abstractions that the developers employ in their discussions and their models. The classification distinguishes between abstractions that are known (e.g., mentioned in the problem description) from those that the developers devised themselves. It also categorizes abstractions according to domain: the real-world domain of driving, the domain of traffic simulation, the domain of user-interface abstractions, and the domain of software design. We then comment on how the use of certain types of abstraction seems to affect the developers' insights into the design problem. Of particular note is how the developers' real-world knowledge of driving (e.g., right-of-way rules and driving conventions) would sometimes hinder their ability or willingness to abstract. We also discuss the distinction between abstraction and simplification — two means of managing complexity in software

systems — and note the various ways that the developers simplified the problem they were solving.

Our analyses are far from conclusive, as we studied only three pairs of developers, who worked for up to two hours on a problem that clearly requires more time than that to produce an adequate design. However, despite these shortcomings, we are able to make some interesting observations about the developers' approach to the design problem.

Throughout the paper, we refer to the developer teams by the names given to the video recordings of their respective design exercises: Adobe, Amberpoint, and Anonymous.

II. Methodology

On our initial viewing of the videos, it seemed to us that there was a clear distinction in how well and how quickly the developers worked through the design exercise. In particular, the Amberpoint team seemed to make further and faster progress towards a final design than the other teams. While we considered a number of reasons why this would be (e.g., their emphasis on requirements, their use of user-oriented design [7], the relationship and interactions between the two developers), we decided to focus on their use of abstraction. The question that we explore is: *Is there any correlation between the teams' use of abstraction and their performance on the exercise?*

We analyzed the videos and transcripts in multiple iterations. First, as a calibration exercise, we both studied the Amberpoint video and transcripts in detail on our own, looking for any abstractions and concepts that the developers used in their design exercise. We then met to compare notes and develop a merged set of identified abstractions. Next, on our own we each performed a similar systematic study of one of the other two videos and transcripts. We met again to discuss our collective results, and try to find commonalities within them. Thereafter, we continued to consult the videos and transcripts, both in their entirety and in ad-hoc queries, as we clustered the abstractions into domains and constructed our final classification.

Despite our best efforts to be thorough and systematic, we cannot claim that our results are authoritative. This is partly because the identification and classification of abstractions is highly subjective, and partly because the analysis was both manual and tedious, and thus likely to be error-prone. Despite this, we believe that the coarse-grained commonalities and differences in the classes of abstractions (e.g., the numbers of abstractions listed in the different tables, or listed in different columns of the same table) tell an interesting story. Not only did the development teams conceive different abstractions, but they spent differing amounts of time discussing the problem with respect to different domains of abstraction.

There has been no attempt to judge the quality of the teams' final designs or to correlate the teams' respective use of abstractions with the quality of their designs. Rather, we have focused on how the use of abstraction relates to each team's ability to make progress during the design exercise.

III. Classification of Software Abstractions

In this section, we itemize and classify the concepts and abstractions that each team spent time discussing. We have attempted to recognize when teams use the same concept or abstraction, even when they use different vocabulary. Analysis of each team's use of the various abstractions is deferred until the next section.

We have used several orthogonal categorizations of the concepts we encountered in our analyses. First, we distinguish between *evident* concepts that the developers were "given" in the program statement versus the concepts that the developers *conceived* of or whose existence they inferred themselves.

Second, we distinguish between the four levels of abstraction domains, from the real world to implementation concepts. We note that the problem specification is unusual, in our experience, in that there are actually two problem domains to consider: (1) The *real-world* problem domain of city driving, and (2) a secondary, or *simulation* problem domain, of traffic simulation. We also identified two more domains that the designers spent time discussing: (3) *user interface (UI)* concepts and (4) *software engineering (SE) / computer science (CS)* concepts.

We also use a third level of categorization of concepts within two scenarios, which we describe below.

A. Evident Concepts

Table I lists the evident concepts that we extracted from the problem statement. We note that all three teams mentioned almost all of these during their sessions, so we list them only to provide context for the discussion of developer-conceived concepts.

B. Developer-Conceived Concepts

Developer-conceived concepts represent entities and abstractions that the developers came up with on their own, based on their knowledge of the problem domain(s), software-design abstractions, computer-science abstractions, and user-interface technology. Tables II, III, IV, and V present the developer-conceived concepts that we identified from the videos, and our classifications of these concepts according to the four abstraction domains listed in the introduction to this section. In each table, a row represents a distinct concept; each concept is listed under every development team (column) that uses that concept/abstraction. To visualize commonalities and variabilities, the tables are separated horizontally into sections, with the top section listing abstractions used by all three teams, the middle section listing abstractions used by two of the three teams, and the bottom section listing abstractions that were used by only one team. Some concepts are listed in multiple tables because they have manifestations within multiple abstraction domains. For example, a queue of cars waiting at a traffic light is a real-world concept that needs also to be simulated by the program. As another example, the simulation domain includes the concept of being able to start and stop the simulation, and the UI domain includes widgets that realize this control over the simulation.

TABLE I
EVIDENT CONCEPTS GIVEN IN THE PROBLEM STATEMENT.

REAL-WORLD (CITY DRIVING) DOMAIN
- time
- traffic signals / colours
- traffic signal timing
- intersection
- car
- road
- lane
- waiting time
- road length
- sensors / sensor input / sensor behaviour
- left-hand turn
- left-hand turn signal
- crash
- direction of travel

(TRAFFIC) SIMULATION DOMAIN
- traffic flow / patterns / density and their specification
- traffic signal timing schemes
- changing / setting signal timing
- traffic simulator
- controlling traffic flow

USER INTERFACE DOMAIN
- visual map / map creation
- traffic flow visualization
- traffic density visualization
- traffic signal visualization
- layout of roads
- intersection design
- control of traffic signal behaviour / user interaction
- visually represent traffic (model individual cars or not)

SOFTWARE ENGINEERING / COMPUTER SCIENCE DOMAIN
- library functions (queuing theory, random number generator, statistical packages)

TABLE II
DEVELOPER-CONCEIVED – REAL-WORLD (CITY DRIVING) DOMAIN CONCEPTS.

ADOBE	AMBERPOINT	ANONYMOUS
direction of traffic	direction of traffic	direction of traffic
speed	speed	speed
distance	distance	distance
# of lanes per road	# of lanes per road	# of lanes per road
road length	block length	block length
queue of cars waiting at light	traffic backup	traffic backup
	block-level speed limit	block-level speed limit
	intersection approach	intersection approach
right-of-way rules		
unprotected left turns (when no oncoming traffic or as light turns red)		
	right on red	
		cars speed up when light is yellow

C. Real-World (City Driving) Domain Concepts

Table II lists the real-world concepts that each of the developer teams used in their discussions or their models. What is most notable about these lists is their degree of similarity. We hypothesize that this is because the developers have a shared understanding of what the real world of city driving is like, at least with respect to the major concepts: roads, lanes, road length, intersections, car speeds, traffic lights, and traffic build-up.

D. (Traffic) Simulation Concepts

Table III lists a number of concepts and abstractions belonging to the domain of traffic simulation. Many of the concepts pertain to the actual simulation of traffic flow, such as the network of roads, the progression of cars along roads and through intersections, and the queueing of cars at traffic lights. A number of other concepts reflect the many ways that a user can configure and control a simulation, including specifying traffic densities at entry points to the simulation, specifying traffic patterns at each light (e.g., percentage of cars that turn left at an intersection), and controlling the simulation speed (e.g., real-time or fast-forward). A third category of concepts is concerned with possible outputs of the simulation, including various analytics, the locations or causes of traffic congestion, or some indication of the "success" of the simulation. Each of the concepts in Table III is labelled as either pertaining to traffic simulation (s), specifying ways to control or configure a simulation (c), or being possible outputs (usually analytics) of the simulation (a).

The greatest degree of commonality is among the teams' use of simulation abstractions. Like real-world concepts, simulation abstractions reflect the teams' knowledge of traffic flows, street geography, traffic lights, and backups. Thus, we hypothesize that the teams' use of similar concepts reflects their shared understanding of the domain of traffic simulation. In contrast, the control, configuration, and analytics concepts are all related to unstated, ambiguous, or imprecise requirements of the program: to what degree should the user be able to control the simulation, and what information *exactly* should the program output? The three teams diverge in their

TABLE III
DEVELOPER-CONCEIVED — (TRAFFIC) SIMULATION CONCEPTS.

ADOBE	AMBERPOINT	ANONYMOUS
(s) network of roads	(s) network of roads	(s) network of roads
(s) car speed (individual speeds)	(s) car speed (common speed)	(s) car speed
(s) flow of cars from one block to another	(s) flow of cars from one block to another	(s) flow of cars from one block to another
(s) rate of cars through intersection	(s) rate of cars through intersection	(s) rate of cars through intersection
(s) queue of cars waiting at light	(s) queue of cars waiting at light	(s) queue of cars waiting at light
(s) simulation clock	(s) simulation clock	(s) simulation clock
(c) rates of input traffic	(c) rates of traffic input	(c) rates of traffic input
(c) block length = road capacity	(c) block length	(c) block length = block capacity
(c) cars enter / leave city	(c) cars enter / leave city	(c) cars enter / leave city
(c) traffic pattern per block	(c) traffic pattern per block (per time of day)	(c) traffic pattern per block
(c) simulation mode (edit, run, pause)	(c) simulation mode (edit, run, pause)	(c) simulation mode (run, pause)
(c) simulation speed	(c) simulation speed	
(s) cars progress down roads	(s) cars progress down roads	
(a) average wait time per intersection	(a) average wait time per approach to intersection	
(a) average wait time per car	(a) average wait time per car	
(c) start / stop simulation		(c) start / stop simulation
	(c) # of cars in simulation	(c) # of cars in simulation
(s) ”on-ramps”, ”off-ramps” for simulation		
	(c) editing mode	
	(c) change settings dynamically, during simulation	
	(c) save: map, sim parameters, traffic config, sim result	
	(c) car types, car destinations	
	(c) program leads user towards optimal light-timing settings?	
	(a) # cars driving the speed limit, moving, waiting	
	(a) min/max # cars waiting at any time	
	(a) avg # cars waiting at each intersection	
	(a) max, avg waiting time, throughput	
	(a) simulation ”result”	
	(a) identify source of bottlenecks?	
		(s) connection points (roads)

Key: (s) simulate traffic, (c) control/configure the simulation, (a) analytics

understandings or opinions about these requirements details.

E. User Interface Concepts

Table IV lists the various user-interface concepts, widgets, and visualizations that the developer teams discussed or modelled. Some of the concepts are concerned with user input and are intended to ease the task of setting up and configuring a simulation. These include drag-and-drop palettes for map creation, pop-up dashboards or sliders for setting traffic-light timings, and default configuration settings. Other table entries describe ways of visualizing results of the simulation, such as highlighting problematic intersections, visualizing the states of the traffic lights (i.e., whether signals are red vs. green), and displaying summaries of analytics. The provided problem statement left the design of the user interface entirely to the developer teams, so there is far greater variability in the teams' conceived UI concepts than in their conceived simulation concepts.

The dearth of UI concepts under the Anonymous heading is striking. However, we defer the discussion of each team's use of abstractions until the next section.

F. Software Engineering and Computer Science Concepts

Perhaps the starkest contrast in the teams' performances is with respect to their use of computer-science and software-engineering abstractions, which are listed in Table V. These include standard computer-science abstractions, such as networks of nodes and edges, state machines, and queues; software-architecture styles, such as model-view-controller and discrete-event simulation; and object-oriented design patterns.

More generally, the degree of variability in the teams' use of abstractions increases as we progress from real-world abstractions to simulation abstractions to user-interface concepts to computer-science abstractions. We hypothesize that this is because computer-science abstractions were used to reason about design and implementation details: in which case it would not be so surprising that there is less commonality among the teams' designs than among their understandings of requirements.

Also noteworthy, the abstractions employed by the Adobe team are predominantly computer-science abstractions (e.g., networks, queues, directed graphs), whereas the abstractions employed by the Anonymous team are predominantly object-oriented concepts (e.g., objects, methods, method calls, delegation). We discuss this contrast in more detail in the next section.

IV. DEVELOPERS' USE OF ABSTRACTION

In this section, we look at how the individual teams approached the design problem. In each case, the deverlopers' overall strategy seemed to greatly affect what abstractions they identified.

A. Amberpoint

The Amberpoint developer team focused intently on the user experience and how students would interact with the

TABLE IV

DEVELOPER-CONCEIVED — USER INTERFACE CONCEPTS.

ADOBE	AMBERPOINT	ANONYMOUS
(i) drag-and-drop palette	(i) drag-and-drop palette	(i) drag-and-drop palette
(i) per intersection settiings: timing, sensor, protected left	(i) per intersection settiings: timing, sensor, protected left	(i) drill down for inputs
(i) roads align with grid	(i) roads align with grid	
(i) intersections derived from road placement	(i) intersections derived from road placement	
(i) road length derived from spatial placement of roads	(i) road length derived from spatial placement of roads	
(i) editor for sensor logic	(i) editor to configure sensor	
(v) visualize traffic as moving dots (along slots in road)	(v) visualize traffic as moving dots	
(v) full roads blink red	(v) roads bolder if they have waiting traffic	
(v) display analytics per car, intersection, road	(v) display analytics per car, approach, road	
(v) dashboard summary of analytics	(v) dashboard summary of analytics	
	(i) road labels / names	(i) road labels / names
(i) start, play, pause, reset buttons		
(i) slider for simulation speed (real-time, fast-forward)		
(i) slider for rate of input traffic		
(i) speed of individual cars adjustable		
(v) highlight specific car, watch it progress		
(v) scroll bars if map too big for window		
	(i) table of roads, entries are int	
	(i) dashboard for setting intersection parameters	
	(i) N/S and E/W roads have same settings by default	
	(i) derive timings for one approach from another's	
	(i) default settings for light timings, left turn, sensors, etc.	
	(i) user-specified global defaults for intersection parameters	
	(i) ability to clone settings	
	(i) static vs. dynamic UI fields	
	(i)(v) timeline of light timings	
	(v) visualize states of traffic lights	
	(v) hide/pop-up intersection details during sim	
	(v) road analytics an aggregation of intersection analytics	
	(v) aggregate analytics for whole grid	
	(v) rank/sort intersections by wait times	

Key: (i) inputs to simulation, (v) visualization of simulation output

TABLE V

DEVELOPER-CONCEIVED — SOFTWARE ENGINEERING AND COMPUTER SCIENCE CONCEPTS.

ADOBE	AMBERPOINT	ANONYMOUS
model-view-controller architecture		model-view-controller architecture
queue (of waiting cars)		bounded queues (of waiting cars)
event-driven (clock ticks)		event-driven (clock ticks)
user stories		use cases
network of nodes and edges		
digraph (plus constraints on edges to reflect "direction")		
graph traversal		
controller (traffic cop)		
state machine of controller		
distributed controller		
dequeuing logic		
visitor pattern		
code: `main()`		
		OO design: objects, attributes, methods
		object interface (API)
		"a big container for everything" (map)
		master/meta controller object
		delegation
		state machine per intersection
		persistence of data / DB
		loop / simulation loop
		polling
		push vs pull model
		sensor-driven
		multi-thread
		drawing package

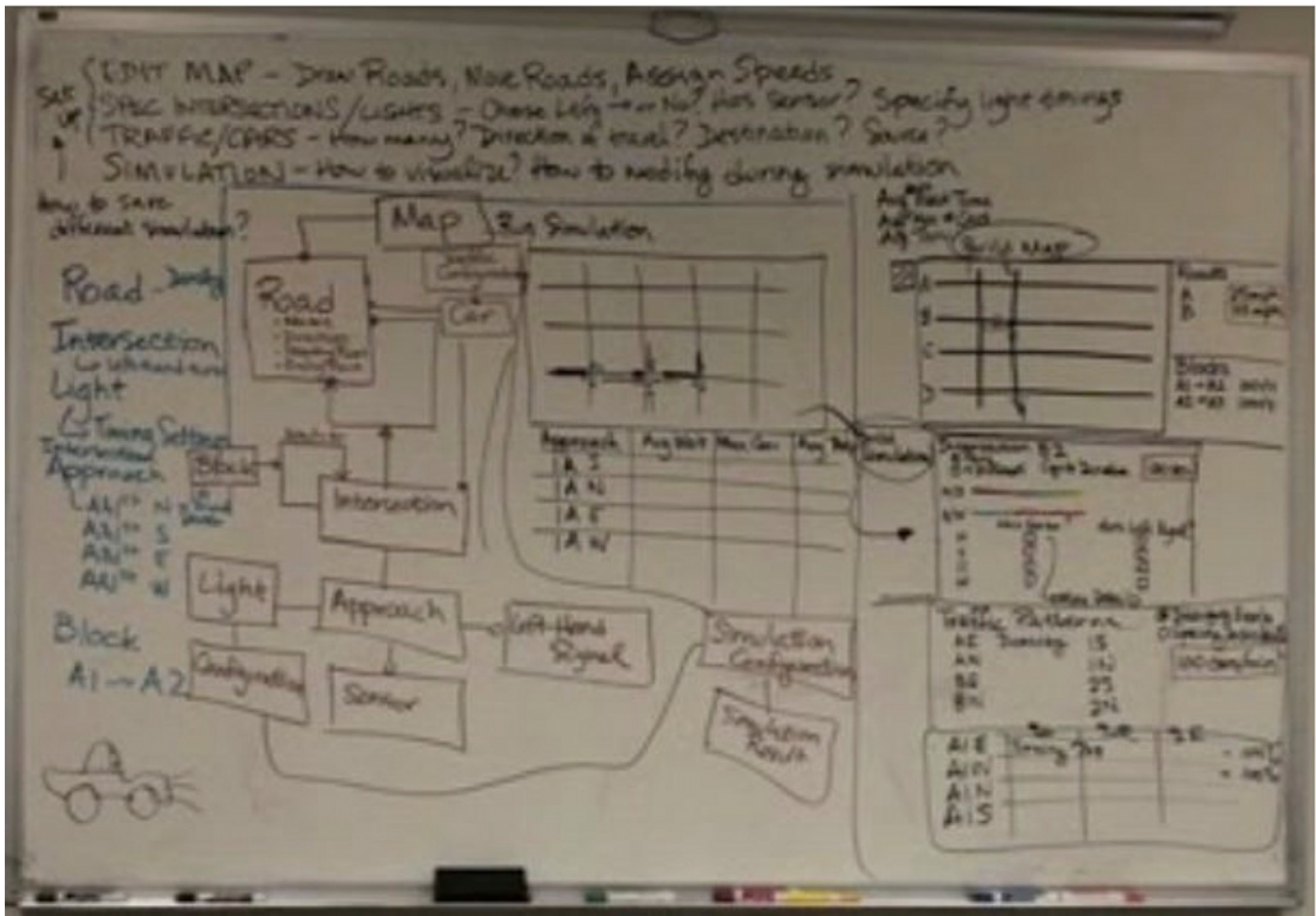

Fig. 1. Final design of the Amberpoint team.

simulation tool:

1) the different modes of operation
2) the information needed to set up a simulation
3) the easiest ways of entering user-provided information
4) the information/analytics that the system should output
5) the most effective ways to visualize that information
6) the level of support the system should provide (if any) in setting acceptable traffic-signal timings

As such, the team seemed engaged in more of a requirements exercise than a design exercise. In particular, questions 1, 2, 4, and 6 are, inherently, questions about requirements: what is the purpose of the program, what degrees of configurability are supported, and to what degree does the program help the user to configure an "acceptable" simulation? As a result of their focus on requirements and on the user experience, the Amberpoint team conceived many more simulation concepts and user-interface concepts than the other teams. These extra concepts are listed in Tables III and IV, and are depicted in Figure 1.

The extra simulation concepts concern the system's inputs and outputs rather than the traffic simulation itself. For example, the developers spent a considerable amount of time discussing the different ways that a student would use the tool: creating and editing the street map, configuring the traffic patterns and traffic signals, running the simulation and tuning it dynamically, and saving the "results." They debated whether traffic patterns should be expressed in terms of global destinations (i.e., each car exits the map at particular point) versus local destinations (i.e., each car travels straight through the next intersection or turns right or left). They considered a number of different analytics that the program might output, and deliberated whether the program should try to be helpful or should simply simulate whatever the user inputs. They were particularly perplexed about what constituted "success": how would the program or the user determine that the traffic-signal timings were acceptable? The developers recognized that a number of these issues were open questions: requirements details to be clarified by the client rather than design decisions to be made by them. They collected a number of questions for the client, and were the only team to do so.

The extra user-interface concepts were mostly aimed at simplifying the task of setting up a simulation. The Amberpoint developers were particularly concerned about the amount of work needed to initiate even a simple simulation: creating

a map, specifying the traffic inputs (i.e., event distributions) at every entry point on the map, specifying the traffic-signal settings (timings, protected left-turn, sensor) for each intersection approach, and configuring the traffic pattern of cars for each intersection approach. The traffic-signal settings alone would involve decisions about 24 signals, for a simulation model of 6 intersections. The Amberpoint developers worried that the configuration of a simulation would be a "big-ass dialogue", and they contemplated ways of reducing the user effort, through the use of default settings, derived settings, and cloning of settings.

The other extra user-interface concepts concern the visualization of the extra analytics that the Amberpoint team identified, such as summarizing analytics in a dashboard, enabling the user to drill-down to more simulation details on demand, visualizing the status of the traffic-signal lights, and dynamically ranking intersections according to wait time.

Interestingly, according to our analysis, the Amberpoint team did not spend any time on design-level concepts and abstractions, as indicated in Table V. As a result, their final design, shown in Figure 1, is more a conceptual design — an entity-relationship diagram with no assignment of responsibility to entities — than a software design, as asked for in the problem statement. That said, the construction of the diagram was spared some of the snags experienced by the other teams, who wondered, for example, whether roads should own intersections or vice versa. Moreover, the Amberpoint final design is much more comprehensive with respect to the user-interface and use cases than the other teams' designs.

B. Adobe

The pair of developers from Adobe approached the design problem from the perspective of useful computer science abstractions. Although they spoke in terms of wanting to identify useful "data structures", their actual focus was on standard computer science concepts (e.g., networks, nodes, and edges; bounded queues, and rules for enqueuing and dequeuing; state-machine controller; diagraphs and graph traversal) and software-engineering design patterns (e.g., model-view-controller design pattern, visitor pattern). The vast majority of their discussions were in terms of these abstractions.

One positive consequence of this perspective is that, by trying to work out the "dequeuing logic" — that is, the rules for when a car moves from one block to another — the Adobe team developed a better understanding than the other teams of the intricacies of traffic flows and backups. These complexities are essential to the design problem. The Amberpoint team did not discuss these details because they focused mostly on the requirements of the system; they got only as far as recognizing that the simulation would have to deal with congestion and with traffic not being able to flow. The Anonymous team recognized that the design rules for simulating traffic flow would be complicated, but they never explored what the rules would look like.

On the downside, because the Adobe team focused on design concepts and object-oriented design, the developers never considered the separation of the system from its environment. More specifically, they do not separate the system phenomena being simulated (e.g., the flow of traffic, the sequencing of the traffic lights) from the environment phenomena being simulated (e.g., the creation of new cars, the destinations of cars). Instead "there's a master intelligence that needs to be looking at the whole state of the world", where their notion of the "world" encompasses all aspects of the simulation. This is a major source of the complexity of their controller. Once the developers realize that their controller is responsible for all aspects of the simulation, as well as updating the analytics, they look for ways of distributing control. They settle on employing the Visitor design pattern, but this is an unusual application of that pattern, and may be a convoluted solution to their problem.

C. Anonymous

The Anonymous team was, to our minds, the most surprising of the three. They spent only an hour on the exercise — compared to two hours for the other teams — and seemed pleased with their results in the subseqeunt debriefing interview, yet they appeared to miss large sections of the problem space in their discussions. More concretely, we observed that the Anonymous team spent significant time discussing the real-world domain of city driving and how that might be implemented as a simulation, but spent relatively little time discussing user interactions or simulation analytics.

In the first few minutes of the discussions, the Anonymous team sketched out what they saw as being the main problem space, unfolding the problem into a user interface — which they mostly did not discuss — and the underlying engine for simulating city traffic. In the following 20 or so minutes, they examined the real-world domain in some detail, considering fundamental modelling questions such as *"What is an intersection?"*, *"Does* traffic==cars*?"*, and *"Should we model individual lanes?"*. A lot of their efforts were spent on abstraction: deciding which real-world details were important to include in their model, and which could be safely ignored.

In the second half of the session, the Anonymous team began to address how these real-world domain concepts might be implemented in a software simulation. In their design, they saw the "map" as the central unifying concept: the underlying "model" in a Model-View-Controller architecture. Concretely, they described it as a container of interconnected objects: cars, roads, intersections, traffic signals, etc. They next addressed simulation-domain concepts such as adding cars into the system, modelling time, tracking cars' progress through the system, and implementing roads as a collection of interconnected queue data structures. And then they discussed the logic of traffic signals, signal timing, co-ordination of cars as they pass through an intersection, and (briefly) the presence of road sensors.

Having finished a fairly thorough investigation into the real-world domain and how to model it in a software simulation, they concluded with a few quick remarks about the user interface: that it should support building maps by drag-and-drop

TABLE VI
SIMPLIFICATIONS MADE BY DEVELOPMENT TEAMS

Simplification	Developmemt Team
all roads are straight (no curves)	Adobe, Amberpoint
no car enters/exits the simulation from an interior block	Adobe, Amberpoint
the speed limit is the same on all streets	Amberpoint
a fixed number of cars in the simulation	Amberpoint
protected left-turn cars have their own lane	Adobe
all traffic input (i.e., new cars, directions) is random	Adobe
all roads have three queues: left-turn, straight, right-turn	Anonymous
all roads have two lanes: left-turn and other	Adobe
all roads have single lane	Amberpoint
all cars have the same length	Anonymous

of components, and that it should be possible to "drill down" on individual components to define configuration settings.

Overall, we felt that the Anonymous team made good progress on modelling the real-world domain concepts, and how they could be implemented in a simulation but did not seriously address either the user interactions or the simulation analytics. The underlying simulation engine seemed to be the only piece of the problem that they considered to be worthy of serious discussions. Presumably, they felt that the user interface was either straightforward enough not to warrant explicit discussion or was simply outside of the scope of the problem. Indeed, at the end of the session, one of the team suggests that they were now ready to start coding, with one engineer on the model and another doing the user interface. Presumably, the user-interface engineer would somehow know what needed to be done.

V. ABSTRACTIONS VS. SIMPLIFICATIONS

Another method commonly used to tackle complexity is to **simplify** the problem, either by decomposing it into subproblems that are easier to address individually or by reducing the extent of the problem. We focus on the latter definition because, like abstraction, it involves omitting information from the model.

Consider the different simplifying assumptions that the three development teams made regarding the number of lanes on a road. The issue that they were grappling with is: whether cars turning left at an intersection should have their own lane, so that they do not hold up traffic if they cannot proceed; and if so, where the separate left-turn lane should begin. The Amberpoint team assumed that there are no separate left-turn lanes, and that all cars advancing towards the same approach to an intersection are aligned in a single queue. The Adobe team assumed that every intersection has a left-turn lane, and that the left-turn lane extends the length of the block (so that there is no issue with left-turning cars sitting in the other lane waiting to get into a full left-turn lane). The Anonymous team abstracted away from lanes, and decided to maintain separate queues for cars wanting to turn left, turn right, or travel straight through each intersection — effectively simulating the case where there are always three lanes of traffic. The three different assumptions would result in very different simulations, with correspondingly different points of congestion. Other simplifying assumptions that the teams made, listed in Table VI, would similarly have observable effects on the simulation.

Are some of the assumptions more acceptable than others? It depends on whether the client cares about the differences in the simulations. Recall that the "program is not meant to be an exact, scientific simulation, but aims to simply illustrate the basic effect that traffic signal timing has on traffic."[1]

An intrinsic characteristic of abstraction is that the resulting model *preserves* the details and properties of interest. A great example of property-preserving abstraction is *program slicing* [8], in which one creates a projection of a program and its state space by considering only a subset of its variables and the program statements that affect the values of those variables. Analyses on the variables of a program slice are as accurate as analyses on the same variables with respect to the entire program. Another example is pseudocode, which is a language-agnostic representation of a program's algorithm. Analyzing the asymptotic complexity of an algorithm expressed in pseudocode produces the same result as analyzing an implementation of the algorithm.

In contrast, a simplification can be a more extreme omission of information: such as deferring features, reducing variability, excluding difficult cases, or making simplifying assumptions about the context in which the system will execute. There is no expectation that the simplification is equivalent to the original. Rather, the expectation is that the simplification will be a "good enough" *approximation* of the original. Thus, the difference between abstraction and simplification centres on the properties that are to be retained in the reduced model: an abstraction preserves the properties of interest, and a simplification need not. Returning to our example about lanes in roads, whether the teams' different simplifying assumptions are simplifications or abstractions depends on whether the client's notion of a protected left includes separating left-turning traffic out from other traffic.

This distinction between abstraction and simplification reflects a recurring problem in software development. Given a design task, a project team makes simplifying assumptions

[1]The problem statement is otherwise silent on the question of whether roads have lanes, so the document cannot be used to judge whether any or all of the teams' assumptions are valid.

TABLE VII
MISSED OPPORTUNITIES TO ABSTRACT, BY DEVELOPMENT TEAM

Missed Opportunities to Abstract	Developmemt Team
unprotected left turns	Adobe
unprotected left-turning cars go when light turns yellow/red	Adobe
right-of-way rules, when multiple cars enter intersection or block	Adobe
cars have distinct speeds	Adobe
distinct types of cars	Amberpoint
overlapping red lights to avoid collision	Amberpoint
right turn on red	Amberpoint
traffic patterns that vary by time-of-day	Amberpoint
cars speed up when light turns yellow	Anonymous

under the guise of abstractions. But the developers may not be in the best position of judging which simplifying assumptions are truly abstractions. If a simplifying assumption leads to an observable effect in the eventual program, then whether the effect is acceptable should be determined by all of the program's stakeholders.

VI. REAL-WORLD KNOWLEDGE AN OBSTACLE TO EFFECTIVE ABSTRACTION

Despite the fact that all three teams actively sought out simplifying assumptions, there were a number of instances where teams got caught up in the details of the problem and did not take the opportunity to abstract away those details. Table VII lists the details that were never simplified.

In many of these cases, the details reflect the developers' attempts to realistically simulate traffic flows. For example, in addition to left-hand turns protected by left-turn-only signals, the Adobe team discusses accommodating *unprotected* left-turns as well, in which cars can turn left as long as there is no on-coming traffic. The Amberpoint team discusses allowing right-hand turns when the signal is red. Abstracting away these behaviours would be comparable to the simplifying assumptions discussed in the previous section: their inclusion or exclusion affects the accuracy of the simulation, and only the client can judge whether the extra degree of fidelity is important — whether exclusion would be an abstraction or an over-simplification.

Other details reflect real-world driving habits and conventions, such as drivers who speed up when a traffic signal turns yellow, or drivers who turn left at the end of a yellow signal after the last oncoming car crosses the intersection. While it is true that simulating these details would result in an extra car (or two) making it through the intersection each signal cycle, these specifics seem less relevant and would have less affect on the simulation than the many simplifying abstractions that the teams did make. The most extreme case was the Amberpoint team's preoccupation with the timings of red signals: ensuring that, when a traffic signal changes, there would be a period of time when the signal in all directions would be red, to avoid collisions. This overlapping of red signals is a trick that is employed in real-world traffic signals to counteract drivers who cannot be trusted, or who misjudge, and do not stop before a signal turns red. A great feature of simulated drivers is that they can be programmed to always obey the traffic lights, so there is no need for any overlap in red signals at an intersection.

We hypothesize that the developers may have missed these opportunities for abstraction and simplification *because* they are experts in the real-world domain of this design problem — that is, experts on city driving. There are times during the design exercise when it seems that this expertise gets in the way of their ability or willingness to abstract. At these times, their efforts are centred is on emulating specific attributes of real-world behaviour, at the possible expense of the main goals of the design problem. Would the developers have been more open to simplifying some of these behaviours if they had been weaker experts on real-world driving and stronger experts on simulation?

There may also have been one or two instances in which developers became fixated on an inappropriate computer-science abstraction. The best example of this is when the Adobe team wanted to model a map as a directional graph, but then imposed a number of constraints in order to make the abstraction work: zero or two edges between each pair of nodes (representing zero or one road between intersections); if there are two edges, their directions must be opposites (to reflect two-way traffic) and their weights (road capacities) must agree. These instances may not represent obstacles to abstraction, but rather confirm the principle that designers should consider multiple abstractions and designs in order to identify an appropriate one. In this specific case, bidirectional edges would have eliminated some of the above constraints.

VII. CONCLUSIONS

This paper has presented some observations about the use of abstraction by professional software developers in the course of a two-hour design exercise. In particular, we have itemized and classified the concepts and abstractions that they employed, and have explored correlations between the teams' design approaches and the resulting lists of concepts and abstractions. We have also commented on the simplifying assumptions that the teams made (and did not make).

We did not attempt to correlate the teams' use of abstraction with the quality of their designs, but rather focused on the correlation between abstraction and progress made on the design task. If other workshop attendees have looked at design

quality, then we can explore the former correlation as well.

REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M.Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[2] J. Kramer. Is Abstraction the Key to Computing? *Communications of the ACM*, 50(4):37–42, April 2007.

[3] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Langauges*, pages 50–59, 1974.

[4] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[5] S. L. Pfleeger and J. M. Atlee. *Software Engineering: Theory and Practice*. Prentice Hall, 4ed edition, 2009.

[6] A. van der Hoek, M. Petre, and A. Baker. NSF-sponsored international workshop on studying professional software designers. "[Online document], December 2009, Available: http://www.ics.uci.edu/design-workshop/index.html".

[7] R.W. Veryzer and B.B. de Mozota. The impact of user-oriented design on new product development: an examination of fundamental relationships. *Journal of Product Innovation Management*, 22(2):128–143, 2005.

[8] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, pages 439–449, 1981.