SOFTWARE DESIGN STRATEGY is about organizing design activities that reflect the designers’ approach to creating a design, and the approach is often opportunistic.  

Studies of design techniques and software designers at work indicate two important activity types (see the “Related Work in Software Design Strategies” sidebar):

- design space exploration is about scoping the issues and deciding how to approach the design, and
- problem-solving is about identifying problems and finding solutions.

Designers can perform these activities in different combinations and sequences. For instance, they can begin by exploring the whole design space broadly or a particular design area in depth. They can proceed during the actual design according to a problem- or solution-driven approach. Figure 1 shows the iterative application of these activities.

Designers can switch strategically between activities during the design phase. However, the way they do so can influence the final design’s effectiveness. Choosing an approach that doesn’t fit the problem will have a negative impact on the design outcome. For instance, it doesn’t make sense to start designing a nationwide e-government system with a depth-first approach because many high-level issues must be considered up front. Designers must understand the scope of the design problem to plan which activities to tackle early.

In a study based on a 2010 workshop at the University of California, Irvine, we observed the design activities and strategies employed by two teams to design a traffic simulator. Team M and Team A consisted of two designers each, and each team employed different design strategies. Here, we evaluate their effectiveness and identify four archetypal design strategies to use under different circumstances.

Software Design Exploration

Design exploration can be explicit or implicit, elaborate or brief, and performed before or during the actual design. Depending on factors such as designer experience and system complexity, the way a design is planned can significantly influence the end results.

As Figure 2 shows, Team M used a breadth-first exploration strategy early in the traffic simulator design. They spent 9 minutes exploring the design scope, using a drawing tool as a metaphor and concluding that they needed to visualize the simulator. The tool calls for a drawing area, which led to questions about which editing modes to distinguish and what objects to visualize (such as cars, intersections, and sensors). At this point, the team made no decisions on how to model these elements; they merely explored the issues broadly.

In minutes 21, 53, and 80, Team M
investigated the scope and evaluated how to meet the design goals. The first time, they discussed the availability of mathematical packages; the second time, how to measure traffic flow; and the third time, whether statistics would be available to help them in the design. These points represent a depth-first exploration strategy.

Team A, on the other hand, spent the first 1.5 minutes investigating the scope of the problem (see Figure 2b), which they framed in terms of intersections and cars. They quickly noted the problem’s time dimension—cars flow through intersections, and both cars and intersections must be modeled—and spent the next hour discussing their solution possibilities, and developing assumption-consequence scenarios.

Software Design Problem-Solving

Designing software requires an understanding of domain-specific design problems. Users and designers typically specify goals at a high level. As the design proceeds, new and implicit goals emerge, along with subgoals, and become problems that designers must identify and solve to achieve the goals.

Teams A and M used different problem-solving strategies. Team A used a solution-driven strategy. In minute 5, they were already making solution statements such as, “I’m thinking in terms of MVC.” Then they decided to use a queue to represent car movements. With this data structure, cars could pop from the queues when a light turns green. They also decided to use a controller to look after clock ticks.

Anchoring on a real-time, event-driven solution, they continued to enhance this solution over the next hour to handle intersections, left turns,
traffic light timing, sensors, and traffic density. Accommodating these new issues more and more complex, and they spent much time discussing the solution but little time discussing the problems (see Figure 2b). In minute 77, they saw that time was running out and decided to abandon their approach and revisit the design scope, where they decided to focus on code design and user interaction.

Kees Dorst and Nigel Cross show that solutions are discovered and evolve based on problem identification. They suggest that a solution can evolve with a problem in a process called problem-solution coevolution (PS-coevolution). Software design methods such as twin peaks and problem frames have a similar idea of co-evolving problems and solutions.

For the most part, Team M followed a PS-coevolution strategy. Throughout the design session, they alternated between raising design issues and proposing solutions. They finally spent as much time exploring the problems as proposing solutions, asking questions such as, “What are the effects of X?” In minute 54, Team M had a burst of questions lasting about 10 minutes (see Figure 2a). During this time, they analyzed issues relating to timing, traffic density, and traffic flow. In the PS-coevolutionary approach, such questioning appeared to have given Team M new insights, and the solutions that followed addressed initiating traffic into the simulation model.

Team M identified many more
problem statements than Team A. Team M’s problem statements represented 31.9 percent of the total number of statements they made during the study, compared to Team A’s 13.4 percent. Basically, Team M formulated their design problems thoroughly while creating their solution. This facilitated a focused design discourse that reduced design context switching.

**Design Strategies**

We’ve so far considered design exploration and design problem-solving separately. The study teams performed these activities without explicit discussion or any conscious strategies. We propose combining these two dimensions to support conscious strategic design choices.

Figure 3 is a matrix of four archetypal design strategies that address breadth-first versus depth-first approaches to design space exploration and problem-driven versus solution-driven approaches to solving design problems in the contexts of design complexity and designer experience.

The upper-left quadrant is a *scoping/questioning strategy*. It applies to green-field and complex systems in which designers have no prior experience. In these cases, designers need to explore the design’s goals and scope first to understand the issues broadly. This helps them appreciate high-level relationships between issues, so they can plan and prioritize activities. When they begin searching for a solution, they will identify new problems that require frequent backtracking as the solution evolves. This fits a context of high system complexity and low design experience in the problem domain.

The upper-right quadrant is a *scoping/solving strategy*. It involves using packaged or known solutions in application domains where the problems are well known and a solution is readily available. Designers start with exploring and setting the scope to ensure that no major surprises exist in deploying a solution. If the target system falls within the solution’s known capabilities, the designers must explore few design problems. This strategy fits a context of high system complexity and low design experience in the problem domain.

The lower-left quadrant is a *no-scoping/questioning strategy*. It applies when designers are familiar with the design’s overall scope but not certain aspects of it. Examples include technical and algorithmic problems such as the security implementation. While searching for a solution, designers must regularly backtrack to explore the problem space.

Finally, the lower-right quadrant is a *no-scoping/solving strategy*. It applies when the design problem and solution are well-known. It fits a context of low system complexity and high designer experience.

Selecting the right combination of strategies given the nature of a particular problem type at the right moment is important in positioning how best to design, but selecting a wrong strategy can give rise to design problems.

**Strategy and Design Effectiveness**

Although the concepts of traffic lanes and lights, intersections, and car queuing are common and appear to be simple, building a traffic simulator to represent these relationships and events in real time is complex. The two design teams took different strategies in their design.

Initially, Team A worked from a no-scoping/solving strategy. Because they didn’t have a broad scope of design issues from the beginning, they couldn’t prioritize all the problems. These issues arose as the team developed their solutions and forced them to switch design topics. Team A context-switched design topics in 1 out of every 4 discussions. As the design problems grew more complicated over time, they had to revisit the same problem again and again. Team A’s ratio of problem-to-solution statements is 1 to 3, indicating their strong preference for the solution-driven strategy that led them to anchor on a solution prematurely. The initial solution wasn’t as suitable as it first appeared, so the problems gradually became more difficult to tackle.

On the other hand, Team M worked from a scoping/questioning strategy. Using a PS-coevolution strategy, Team M’s ratio of problem-to-solution
statements was almost 1 to 1, showing that they explored the problem space more thoroughly. The combination of these strategies meant that they context-switched design topics less frequently, 1 in every 7.7 discussions, making their discussions more effective. A depth-first strategy like this is effective when the issues in that area are well understood, and the designers don’t have to branch off to tackle side issues.

Traffic simulation is complex, and its design issues were new to all four designers. Under these circumstances, the scoping/questioning strategy is more effective because the in-depth design that follows a systematic problem exploration helps minimize context switching. A no-scoping/solving strategy potentially postpones problem identification, but in Team A’s case, the solution-anchoring effect appeared to encourage the team to stay with their solution. In other words, having settled on a solution prematurely, they were reluctant to abandon it and backtrack to a better option.3

Strategy and Design Context
A design problem’s complexity is relative to both the problem’s difficulty and the designers’ experience in the relevant application and technical domains. A designer might have worked for many years but can still be relatively inexperienced with a new domain and technology. Moreover, a design problem that appears complex to one designer might be familiar to another. Mismatches between the contexts and strategies can reduce design effectiveness and quality. If we consider study teams designing the traffic simulator, the design is complex, and none of the designers has experience in the domain. Team M’s choice of a scoping/questioning strategy was more effective than Team A’s no-scoping/solving strategy.

If designers are experienced in a certain application domain and a packaged solution is available, then using a questioning strategy to revisit solved problems would be a waste of time. Instead, designers should choose a scoping/solving strategy. As new information and design issues are discovered, a seemingly simple design can become more complex than first anticipated. Newly discovered design issues may impact the design strategy. Designers need to choose a suitable strategy conscientiously and adopt new strategies as the context changes.

Our study shows that the misuse of a design strategy can result in less effective designs. The reasons lie in frequent switching of design contexts and in revisiting design solutions that don’t clearly articulate the design problems. Deciding what strategies to use, implicitly or explicitly, dictates the course of design actions and thus the design’s effectiveness. More understanding of these “meta” decisions and their impact on design should help reduce mistakes and wasted time.

ACKNOWLEDGMENTS
The US National Science Foundation funded the design sessions and corresponding workshop under award CCF-0845840. We thank the designers who participated in the sessions that provided the input data to this project. We thank the organizers, André van der Hoek, Marian Petre, and Alex Baker, for granting access to the transcripts and video data. We also thank the anonymous reviewers for their invaluable suggestions in improving this article. The Dutch Regeling Kenniswerkers partially sponsors our work under project KWR09164.

References