December 2006 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

LET'S DO THE NUMBERS
The annual salary survey results are in!

SEMPER FI
Lead like a US Marine

Changing The Hand You're Dealt

Better Designs Through Problem Redefinition

Changing The Hand You're Dealt

Better Designs Through Problem Redefinition

By Payson Hall

For many of us, design is the most enjoyable part of software development. Oh sure, debugging can be fun and it *is* wonderful when everything comes together at the end, but if you're like me, you prefer to spend a little more time "up front" during design to minimize the complexity of debugging and maximize the likelihood that it *will* all come together.

Design is the all-too-brief period during an application's evolution when we consider what we want to create and speculate about how we want to do it but are not yet fully committed to any particular approach to the problem. A shortcoming of some of the best programmers I know is the tendency to minimize this "pre-coding" thinking time and begin implementing the first viable solution that occurs to them. The trouble is that there are several ways to solve most problems, each exhibiting various levels of efficiency, ease of implementation, maintainability, and portability. If a designer stops with the first design that "works," there may be a needless sacrifice of quality. In this rush it is easy to confuse a functional design with a quality design. The distinction can be subtle, particularly if both are implemented professionally, because both will get the job done. The difference in development and maintenance costs, however, can be remarkable.

"Wait!" I imagine you thinking as you read this. "The 'Best' is the enemy of the 'Good.' If we worked until we had the perfect design, we would never accomplish anything!" Wise words, but frequently applied prematurely to software design. As a profession, software development seems to be in little danger of overemphasizing design or seeking perfection.

This article focuses on an aspect of the problem definition portion of the design process using two examples—a parable and a program fragment—that illustrate how seemingly small changes to the problem definition or representation can have a simplifying effect on the solution.

Redefining the Problem

From the perspective of the system developer, complexity is the enemy of the development process. There is a limit to the amount of complexity that any of us can deal with effectively, and when our capacity is exceeded, system quality is a frequent casualty. One goal of the skilled designer is to minimize solution complexity in order to maximize the quality of the resulting system (by "quality" I mean a system that can be developed efficiently; meets its requirements; and is reliable, portable, and maintainable).

Good designers use a number of techniques to seek simplicity. One method that may seem counterintuitive is redefining the problem to be solved—sometimes even adding complexity to the original problem definition—in order to reduce the complexity of the solution.

A Puzzling Parable

I was first exposed to the problem redefinition strategy in a math puzzle. A shepherd died and was survived by three sons and twenty-three sheep. In his will, the shepherd left half of the sheep to the eldest child, one-third to the middle child, and one-eighth to the youngest child. Since the sheep were to be used for their wool, the sons weren't thrilled with the notion of cutting up sheep to account for fractions. As they stood by the road arguing about how the division should occur, a wise woman came along, leading a sheep of her own to market. When the problem was explained to her, she considered it for a moment, and then praised the boys for not being too quick to butcher the sheep, which she said was not really necessary. They looked at her curiously, and she said, "Let me lend you this lamb as a tribute to my friendship with your father." She then led her lamb into the herd. "Now the division should be simpler." The eldest child smiled as he took away half (twelve) of the sheep to begin his herd. The middle child breathed a sigh of relief as he took one-third (eight) of the sheep from the pen and led them home to his farm. The youngest child watched as the old woman led her lamb from the pen and continued down the road toward the market, leaving him with one-eighth (three) of the herd.

Too often during the design process we find ourselves worrying about the complex implementation of messy details (like dividing live sheep), without first seeking ways to minimize complexity by redefining the problem. Subtle changes to the problem being solved can have surprising benefits in terms of simplification.

A "Solitary" Programming Example

Solitaire is the name given to a family of card games that are played by a single player. I enjoy solitaire, and I have always wondered which strategies of play are the most effective. I once designed and built a simulation program to play a solitaire game called "Klondike," probably the most widely played variant of solitaire (see the sidebar for game rules and visit the StickyNotes to see the simulation program). I wanted to observe the effects of different strategies over a large number of games. During my initial thinking about the design, I was dismayed by the number of special cases that seemed to exist when I tried to describe in algorithmic terms how to play the game. The initial placement of aces is a special case, as is some of the logic required to handle depleting one of the seven face-up piles on the board:

```
                              .
                              .
                              .
X.0   If the face-up pile is exhausted
      X.1   If there are face-down cards
                  turn one face up
      X.2   Else (no face-down cards)            /*Special Case*/
                  a king can go here
                              .
                              .
                              .
```

The possibility that the table might be left bare requires yet another special case in the card playing logic (the second special case below):

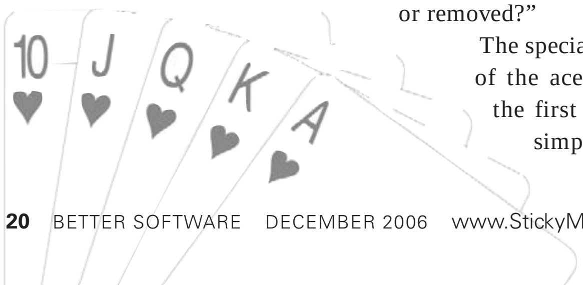
```
1     If value of card_to_play = ace               /*Special Case*/
            move to suit row for score
2     Elseif value of card_to_play = (top card in its suit row) + 1
            move to suit row for score
3     Else
            Do for each face-up pile (until placed)
      3.1         If value of card_to_play = king  /*Special Case*/
                  andif (face-up pile exhausted)
                        start a new face-up pile with the king
      3.2         If color of card_to_play is red
                  andif (value of top face-up card) = card + 1
                  andif (color of top face-up card) = black
                        move card_to_play to face-up pile
      3.3         If color of card_to_play is black
                  andif (value of top face-up card) = card + 1
                  andif (color of top face-up card) = red
                        move card_to_play to face-up pile
            End do
4     If not played by an earlier step
            card_to_play is not playable (place in discard pile)
```

I didn't like it. It was too messy, and it violated one of the great truths of system design: "Nature abhors the special case."

Changing the Problem to Eliminate the Special Cases

I set out to see if I could factor out the special cases to simplify my task. I realized that kings and aces were special cases because they were boundary conditions. It is common to have special cases at the boundaries of a problem; the question is "Can the boundary be moved or removed?"

The special case of the aces was the first to be simplified.



I asked myself, "Is there a way to avoid the special test for aces?" (step 1 in the algorithm previously described). I observed that the reason aces were a special case is that they comprised the foundation of the suit row and, unlike all other cards in the deck, had no ordinal predecessors. This meant that they could not be placed according to step 2. Until the aces were placed, no deuces would pass the test in step 2 either. "If only there were a zero of hearts," I thought. "Then aces would no longer be a special case." Eureka! I imagined placing zeros of hearts, diamonds, clubs, and spades on the table as placeholders for the suit row at the start of each game. Now step 1 was no longer necessary, and aces could be placed according to the rules already defined in step 2.

The problem definition was changed slightly (solitaire is not normally played with a fifty-six card deck that includes zeros of hearts, diamonds, clubs, and spades), but the underlying problem (i.e., accurately simulate a game of Klondike) was not changed, and the solution was simpler. All I had to do was add four cards to my fifty-two card deck, which would be "nailed to the table" during initialization and not affect play in any substantive way. Once the special case of the aces was solved, the kings were just a step away.

The king boundary problem also arose because kings do not have predecessors (actually ordinal successors; remember that you count down from high to low when playing cards on the board). Kings could only be moved to the board when the face-up and face-down piles were exhausted, exposing the bare table below. Having already overcome the conceptual hurdle of inventing cards and adding them to the deck, I imagined a card larger than a king (an emperor?) that I could place on the table to attract kings when the face-up and face-down piles had been exhausted (currently handled by step 3.1). This would allow me to eliminate step 3.1 altogether, since, if kings were playable, they would be played during step 3.2 or 3.3. The trouble was that steps 3.2 and 3.3 required that the emperors be both black *and* red if any king were to go on any bare table space. I realized that I was being too literal about representing the deck of cards—assuming that it could only contain two colors, red and black. If I allowed a third color (green), then I could make my emperors that color and combine steps 3.2 and 3.3 into a single step:

```
If (value of top face-up card) = card_to_play + 1
andif (color of top face-up card) ≠ card_to_play
      move card_to_play to face-up pile
```

All I needed to do was add a green emperor face down at the bottom of each face-down stack during initialization, and the rules became simpler still.

I assigned all of the emperors a suit of "circles." Emperors were now the boundary condition, but they could never be

Klondike Rules

In Klondike, a regular deck of fifty-two cards is used. To start the game, seven face-down piles of cards are dealt in a row to form the "board." The first pile of the board contains one card, the second has two cards, the third has three, and so on. The top card of each board pile is turned face up and placed on top of the remaining cards in that pile. The rest of the deck comprises the "stock."

The goal of the game is to maximize the number of points scored during play by making a series of legal moves (described below) between piles of cards.

Whenever an ace is revealed during play, it is placed in a separate pile in a suit row above the board. On the aces you may build an ascending sequence of cards of similar suit (e.g., on the ace of spades you may place the two of spades, followed by the three). The cards placed on the aces may be moved individually from the top of the face-up piles of the board or from the discard pile. A point is scored for each card played on the suit piles. To "win" the game and achieve a perfect score, all four aces must be placed in the suit row and the ascending sequence must be built completely to the king of each suit (a total of fifty-two points). Once a card has been placed on a suit pile, it may not be moved for the rest of the game.

To play the game, the cards are dealt onto the board and the player makes legal moves of cards between the piles. On the face-up piles of the board you may place a card of alternate color that occurs next in descending sequence (e.g., if the top face-up card of a pile is a black nine [clubs or spades] then a red eight [hearts or diamonds] can be placed on that pile). Cards placed on top of the face-up pile may be the card or cards comprising another face-up pile or a card from the discard pile. When one face-up pile is moved to another, all of the face up cards in the original pile must be moved as a group. The highest value card (the bottom) of the face-up group being moved must be legally playable on top of the receiving pile. When the face-up cards are removed from a pile, it may reveal a pile of face-down cards (if the face-down pile has not been exhausted), and a face-down card may be turned face up on that pile to replenish the face-up pile. When a pile is completely exhausted, the vacant place on the table is eligible to receive any king that is on the board or is revealed later during play. If a king is not immediately available, play continues, leaving the vacant place on the table unused until the game ends or a king becomes available.

Cards from the stock are turned face up one at a time and placed on a separate discard pile. The top card in the discard pile may then be played onto any of the other piles (either the board or the suit row) at any time. Once the top card from the discard pile has been played, any legal moves created by the play may be taken. When a card is played from the discard pile, the top card remaining in the discard pile may then be played. When the stock has been exhausted and no legal moves remain, the game is over.

moved (there was no card bigger that would cause them to relocate on the board, and there was no "king of circles" to attract them to the suit row). This meant that the test in step X.1 was no longer necessary; if the face-up pile was exhausted, then there had to be a face-down card until you turned up the emperor, and once you turned up the emperor it could not be moved to exhaust the stack. This also eliminated the special case for king processing described in step X.2, because once the emperor in a face-down pile was exposed it would attract kings according to the modified rule 3 described previously. All that remained was an administrative detail: It was necessary to create a zero of circles to avoid adding complexity into step 2 when testing whether cards in face-up piles were playable.

The end result of adding these twelve additional cards (five zeroes and seven emperors) to the deck was a minor change to the dealing (initialization) routine and a tremendous reduction in the complexity of play, without altering the "real" game in any way.

The simplified rules of play were:

```
X   When a face-up pile is exhausted
        turn a face-down card face up
                •
                •
                •
1   /* step eliminated */
2   If value of card_to_play = (top card in its suit pile) + 1
        move to its suit pile for score
3   Else
        Do for each face-up pile (until placed)
            If (value of top face-up card) = card_to_play + 1
            andif (color of top face-up card) ≠ card_to_play
                move card_to_play to face-up pile
        End do
4   If not played by an earlier step
        card_to_play is not playable (place in discard pile)
```

By changing aspects of the problem I reduced the complexity of the solution without sacrificing functionality or changing the underlying system behavior. Eliminating the special cases increased both the simplicity and quality of the solution.

Identifying and avoiding needless complexity is part of building quality systems. Changing the definition or internal representation of a problem may significantly reduce solution complexity, but the analysis consumes time and resources. Is the investment worthwhile?

A designer or project manager with limited resources and an aggressive schedule might reasonably question the benefit of searching beyond the first "good enough" solution identified, but finding a simpler solution can be extremely valuable. Simpler solutions are usually less expensive and time consuming to build, less error prone, easier to test, easier to maintain, and easier to modify. When design simplification results in efficient, simpler, and smaller programs, real productivity is improved. **{end}**

Payson Hall (payson@catalysisgroup.com) is a consulting systems engineer and project manager from Catalysis Group, Inc. in Sacramento, CA. Formally trained as a software engineer, Payson has performed and consulted on a variety of hardware and software systems integration projects in both the public and private sectors throughout North America and Europe during his twenty-six-year professional career. He is a regular columnist on StickyMinds.com.

Sticky Notes

For more on the following topic, go to www.StickyMinds.com/bettersoftware.

- Klondike simulation program