Even Larger Programs

As the size of our programming problems continues to grow, so does the need to break the task down into manageable pieces. Writing small pieces of code, then testing them is the only way to make solid progress. The opposite approach (which is a poor choice) is the "big bang" approach, in which the programmer tries to write all of the code, and only then start testing and debugging.

One important way to split larger problems into smaller problems is to start with a minimal set of features—implement, test and debug those—then add new features one at a time. After you add each new feature, re-test the entire system, and make sure that everything still works. This model of development requires more discipline (resisting the temptation to write everything all at once—and hope it will all just work), but pays off in terms of increased overall productivity and software quality. Discipline (especially under pressure) is one of the key features of good programmers—a good programmer will work (whether implementing, debugging, and testing) in a disciplined manner, especially under time pressure. Remember that "haste makes waste."

As an example of how we might apply this principle, consider writing a program to let someone play chess. The first thing we might do is just write the program so that it displays the board, but not even any pieces. This may not sound like much, but is the smallest piece we can make, thus something we can and should stop and test before we proceed. Once we have this part working, we might add displaying the pieces on the board, and test again. A next step might be to allow the user to move pieces around, but without worrying about the legality of the moves—just letting the user pick up any piece and put it down anywhere else.

At this point, it is likely quite useful to add code to save and load a board position. Whether or not we intend to have this feature in the final game, it will prove incredibly useful for testing—letting us set up and use test cases in an efficient manner. As with all other functionality, we should test this out thoroughly—we would not want our later testing confused and complicated by poor infrastructure.

After testing this functionality, our next step would be to add the rules of the game one-by-one. We can add the basic rules for each piece moving, and after each one, test the code again. After all the basic moves, we can add castling, and other complex rules (and test each as we add them). We can then add code to check for winning (and test) and drawing (and test). If we want our program to have other features (*e.g.*, playing across a network), we can then add them one at a time (and for large features, break them down into smaller pieces) and test as we go.

Completed