The Design Checklist: A Method for Creating Programs

Norman Ramsey

Fall 2010

Design checklist for creating programs

This design checklist is intended to help you solve problems that are larger in scope than the creation of a single abstract data type: a whole program, not just a single abstraction. *The course staff will not answer substantive questions for students without checklists.*

1. *What problem are you trying to solve?*
2. *What example inputs will help illuminate the problem?*
3. *What example outputs go with those example inputs?*
4. *Into what steps or subproblems can you break down the problem?*
5. *What data are in each subproblem?*
6. *What code or algorithms go with that data?*
7. *What abstractions will you use to help solve the problem?*
8. *If you have to create new abstractions, what are their design checklists?*
9. *What invariant properties should hold during the solution of the problem?*
10. *What algorithms might help solve the problem?*

And once you have a design,

11. *What are the major components of your program, and what are their* ***interfaces****?*

    Components include functions as well as abstract data types. An interface includes contracts as well as function prototypes.

12. *How do the components in your program interact? That is, what is the* **architecture** *of your program?*
13. *What test cases will you use to convince yourself that your program works?*
14. *What* **arguments** *will you use to convince a skeptical audience that your program works?*[1]

What to submit with your program

The design checklist is a tool to help you create working programs. It is not a means of *explaining* a finished program. However, when you submit a program, certain elements of the checklist should be used to explain your work:

11. Components and their interfaces
12. Architecture (how components interact)
9. Invariants
13. Summary of what testing you've done
14. Explanation of why it works

[1] A narrative description of an algorithm is *not* an argument! A convincing argument usually involves invariants that hold during execution and reasoning that once execution is over, the invariant implies the desired result.