

The Design Checklist: A Method for Designing Abstractions

Norman Ramsey

Fall 2009

When faced with a difficult engineering problem or homework assignment, you may find that you don't know how to get started, or you get stuck trying to get your program to work. In these situations, it is helpful to have systematic methods to fall back on. This handout describes a method developed at Rice University and tested at over 20 universities around the world. I have adapted the method for use in COMP 40.

A wise man once said that we don't need to use systematic methods on every problem, just on problems that are hard. An even wiser man said that it is sometimes useful to practice systematic methods on easy problems, so that when confronted with a hard problem, you can focus on the problem, not the method.

The design checklist for abstract data types

If you have difficulty designing an abstract data type, please complete the following checklist. If completing the checklist does not get you unstuck, take your completed checklist to a member of the course staff and ask for help. Similarly, if you find yourself unable to complete the checklist, ask for help. *The course staff will not answer substantive questions for students without checklists.*

1. *What is the abstract thing you are trying to represent?*
Often the answer will be in terms of sets, sequences, and finite maps.
2. *What functions will you offer, and what are the contracts of that those functions must meet?*
3. *What examples do you have of what the functions are supposed to do?*
4. *What representation will you use, and what invariants will it satisfy?* (This question is especially important to answer *precisely*.)
5. *What test cases have you devised?*
6. *What programming idioms will you need?*

A detailed example

Let's suppose you've decided to implement an abstraction that maintains a set of future events with a time for each event, and you want to efficiently find the next event that is scheduled to occur. Here's the checklist:

1. *What is the abstract thing you are trying to represent?*
A set of events, each of which has a time. We'll call it an *event queue*, or `eventq` in C.
2. *What functions will you offer, and what are the contracts of that those functions must meet?*

We'll want these functions:

- `bool isempty(eventq q)` tells if the event queue `q` is empty.
- `event get_next_event(eventq q)` returns the event in `q` with the smallest time. It removes the event from `q` and returns the event. The function's contract says it should be called only when `q` is not empty. It should be fast, e.g., logarithmic in the size of the queue.
- `void add_event(eventq q)` adds an event to the queue. It should be fast, e.g., logarithmic in the size of the queue.
- `eventq new_queue(void)` creates a new, empty event queue.

These functions and contracts make it clear that `eventq` is a *mutable* abstraction.

3. *What examples do you have of what the functions are supposed to do?*

Here are some examples:

- If `get_event` is called on an empty queue, it halts the program with a checked run-time error.
- Here's another example sequence:

```
eventq q = new_queue();
add_event(q, e1); // e1 has time 3.0
add_event(q, e2); // e2 has time 9.0
add_event(q, e3); // e3 has time 2.0
isempty(q); // returns false
get_event(q); // returns e3
get_event(q); // returns e1
isempty(q); // returns false
get_event(q); // returns e2
isempty(q); // returns true
```

4. *What representation will you use, and what invariants will it satisfy?*

I'll use a binary heap, which is either

- The NULL pointer, or
- A pointer `p` to a structure containing a time, a pointer to an event, and two child heaps `p->l` and `p->r`. The children must not only be binary heaps but must satisfy these additional conditions:
 - Either `p->l` is NULL or `p->l->time ≥ p->time`
 - Either `p->r` is NULL or `p->r->time ≥ p->time`

5. *What test cases have you devised?*

Test cases are left as an exercise for the reader.¹

6. *What programming idioms will you need?*

- The idiom for allocating and initializing pointers to structures
- The idiom for creating an abstract type using incomplete structures
- The idiom of recursive functions for recursive types

¹This phrase sounds academically respectable, but unless the exercise is carefully crafted, it's really just a way of dodging work. I'm dodging work.