# The Design Checklist: A Method for Problem-Solving

Norman Ramsey

Fall 2010

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

If you get stuck on a problem, 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 data are in the problem?*
2. *What code or algorithms go with that data?*
3. *What abstractions will you use to solve the problem?*

And for each individual abstraction,

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

## A detailed example

Let's suppose you're given the following problem:

> *Simulate the real-time behavior of an analog circuit.* We'll assume that the circuit is a set of elements, and each element provides an *event stream*. An event stream is one of the following:
>
> - A pointer to a structure containing a time, a string, and a function which, when applied to the pointer, returns an event stream.
> - The NULL pointer
>
> You are given a set of event streams; your job is to run a simulation that prints out the time of each event and the string associated with that event.

Here's the checklist:

1. *What data are in the problem?*

   We have these data:

   - Event streams, which are sequences
   - Times, which we'll assume are numbers indicating how many nanoseconds have passed since the start of the simulation
   - A set of events

2. *What code or algorithms go with that data?*

   A wide variety of algorithms go with these data. I'll focus on finding the event stream with the smallest time.

3. *What abstractions will you use to solve the problem?*

   We'll need an abstraction that maintains a set of events and allows us quick access to the next event, which is the one with the smallest time. Let's call this abstraction an *event queue*.

4. *What is the abstract thing you are trying to represent?*

   A set of event streams, each of which has a time.

5. *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_stream get_next_event(eventq q)` returns the event stream in `q` with the smallest time. It removes the stream from `q` and returns the stream. 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 stream 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.

6. *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
```

7. *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 stream, 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` $\geq$ `p->time`
     - Either `p->r` is `NULL` or `p->r->time` $\geq$ `p->time`

8. *What test cases have you devised?*

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

9. *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

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