COSC345 Week 12

Working backwards

3 October 2017

Richard A. O'Keefe

Where does this fit?

Elsewhere we consider testing and static checking.

They are ways of dealing with errors.

But how can we **avoid** them?

How do we think about programming?

Coding is a form of problem solving

Given

—**1**— the postcondition to establish,

—**2**— the preconditions you may assume,

—**3**— the language to use, and

—**4**— the components you may use;

Construct

— a component in that language which establishes that postcondition provided that precondition is true.

How to Solve It (Polya)

1. You have to *understand* the problem.

2. Find the connection between the data and the unknown. You may have to consider auxiliary problems if an immediate connection cannot be found. You should obtain a *plan* of the solution.

3. *Carry out* your plan.

4. *Examine* the solution obtained.

(Taken from "How to Solve it", by G. Polya. Put this book at the top of your "to buy" list. I mean it.)

Understanding the Problem

— What is to be found/constructed/proved (component)?

— What are the data (precondition, language, library)?

— What is the condition (postcondition, constraints)?

— *Can* the condition be satisfied?

— Is the condition redundant, inconsistent, insufficient?

— Draw diagrams. Find or invent a suitable notation.

— Separate the parts of the condition.

Devising a plan I

— Have you seen it before? (Schema/Pattern)

— Have you seen the problem in a slightly different guise?

— Do you know a related problem?

— Do you know a schema or transformation that could be useful?

— Look at the unknown (postcondition/behaviour)!

— Try to think of a familiar problem that is like it.

— Here is a solved problem like yours. Can you use it?

Devising a plan II

— Could you use the *method* of that solved problem?

— Should you introduce an auxiliary element to adapt it?

— Could you restate the problem? Go back to definitions.

— Try to solve a related problem instead:

- a more special problem?
- a more general problem?
- an analogous problem?

Devising a plan III

— Can you solve part of the problem?

— Can you derive something useful from the givens?

— Can you think of other information that would be helpful?

— Can you change the problem to make it easier to solve?

— Did you use all the data?

— Have you considered all the essential notions involved in this problem?

Carrying out the plan

-**P**- Check each step

-**P**- Can you see that the step is correct?

-**P**- Can you prove that the step is correct?

-**O**- Check the connections between steps.

-**O**- Can you see that each connection is correct?

-**O**- Can you prove that each connection is correct?

-**O**- Can you see how to test this component?

Looking back

-**P**- Can you check the result?

-**P**- Can you check the argument?

-**P**- Is the result obvious at a glance?

-**O**- Is the code well laid out?

-**O**- Are words and comments spelled correctly?

-**O**- Is there prunable detail, now you understand it?

-**P**- Can you use the result, or method, for another problem?

Alain Fournier's strategies

* Start at the beginning
* Visualise
* Take it apart
* Look for angles
* Don't dismiss foolish ideas right away
* Restart often
* Sweat the details
* **Do not assume**
* Try to solve again

Working forwards

The natural way to start from a problem is to work *forwards* from the data, floundering around until you find a solution.

This is particularly true for coding.

"Start at the beginning, keep going until you reach the end, and then stop" is a *very bad* way to code.

This is related to, but not quite the same as, "bottom-up coding".

Working backwards I

"exceptionally able people, or people who had the chance to learn in their mathematics classes something more than mere routine operations, do not spend too much time in such trials, but turn around, and start working backwards." (Polya, p227).

This is related to, but not quite the same as, "top-down coding".

You start from the *bottom* of the procedure, and work back towards the top. You start at the *top* of the module hierarchy, and work down towards the leaves. You *postpone decisions* about low level issues like data representations.

Working backwards III

— Visualise the final situation.

— From what forgoing situation could that be obtained?

— But how could we reach *that* situation?

— The order in which we *design* the parts of a component does not have to be the order in which they are *assembled* (LP) or *executed*.

Working backwards IV

Sometimes there are several ways to solve a (sub)problem, but none of them works all the time.

That's what IF and CASE statements are for. They are a way of gluing together solutions each of which can handle some situations but not all. You must check that each case that can arise is handled by at least one arm of the IF or CASE.

The books by Dijkstra, Gries, and Reynolds explain this method in detail, for the case of coding.

Documentation

Assertions are not decorative

They are the framework which supports the code

Write the assertions first and keep them

They are also good for testing and debugging

C, C++, Java (since 1.4), and Python have them.

Eiffel really does them thoroughly, Ada is nearly as good and has SPARK.

AI Planning I

— Classic STRIPS/WarPlan-like planner.

1. A state is a set of facts.

2. An initial state is given.

3. The desired final state is (partially) specified.

4a. An operator requires some facts to be true.

4b. An operator makes some facts true.

4c. An operator makes some facts false.

— Use the method of working backwards.

AI Planning II

WarPlanC extended to conditional actions

Manna and Waldinger computed functions this way

— inference engine computes proof that function doable

Rich, Waters, and Shrobe "Programmer's Apprentice"

— used catalogue of program fragments

Exponentially hard in general