Technique for Everyone

Norman Ramsey

COMP 40
Tufts University

Here is a summary of your work on the technique exercise. I've organized and classified your work, and I have written some responses in square brackets.

1. Technique outside of programming

Great direction and level of detail

- Breakdancing: when feeling body backward, pressure against fingers
- Graphic design: Step back and look at the whole canvas; study the relationship of the elements.
- Piano: tap foot to keep in rhythm[1]
- Piano: know scales and hit them consistently
- Rockclimbing: keep weight as close to wall as possible
- Running: when getting tired, keep arms swinging, will carry feet
- Rugby: keep low to the ground

Good detail with some direction

- Climbing: visualize your moves before you make them
- Fire poi: know where both chains are going to be in each rotation [note similarity with climbing]
- Lacrosse: step when throwing
- Music: in ensemble playing, listen as well as play out
- Photography: know the rule of thirds [which is?]
- Piano: sit up straight with relaxed shoulders
- Soccer: know your role on the team [how do you identify your role? what if there are no coaches?]

Great goal, but not given technique to achieve it

- DJ: turn off mic when playing music [what habits or cues do you develop or use to get the mic off and then back on at the right time?]
- Crew, breakdancing: keep back straight [without a mirror or a video feed, how do you know if your back is straight?]
- Archery: fix posture [same questions as "back straight"]
- Singing: take care of your voice [how?]
- Tennis: deliver a strong serve the lands in the opposite corner box [how?]
- Ultimate: making "good cuts" [what's good or bad about a cut? *how* do you make good ones?]
- Ultimate: stretch and warm up [stretch what, for how long? warm up how?]

Vague goal, learner could easily go wrong

- Basketball: be unselfish [what does it mean for a basketball player to be unselfish?]
- Design: have a plan [where do plans come from?]
- Guitar, basketball: practice [what? how?]
- Guitar: strengthen fingers [Robert Schumann, phone your office. . .]
- Sailing: watch the water

Motherhood and apple pie, not useful guidance

- Tennis, backing, volleyball, frisbee: Balance, timing, precision, accuracy [Nobody wants to be off-balance, ill-timed, imprecise, or inaccurate. But to pick one example (volleyball), nobody can be supremely precise all the time. Do your digs, sets, and spikes all have to have the same level of precision? Or did you mean to talk about foot placement? Hand placement?]
- Polo: be aware of your surroundings

Personal qualities

Valuable personal or character qualities useful in many fields of endeavor, but not necessarily helpful to improve in any particular field:

- Organization
- Patience
- Tenacity

Nobody wants to be known as disorganized, impatient, or someone who easily gives up. But to develop these qualities requires guidance and practice.

[1] I was taught to count—out loud if necessary.

2. Technique for programmers

Great direction and level of detail

- Comment consistently
- When you get frustrated, go to dinner, take a break to think about it. [Excellent advice. If you've eaten, go for a 15-minute walk outside.]

Good detail with some direction

- Change of location, get out of room. [Programming is one field of endeavor where the "geographical cure" actually works, but more guidance would be useful here, like *when* and *how often*.]
- Draw diagrams (3 groups).

  Good advice, but *when* do you draw diagrams? I suggest:
  - Whenever a problem seems hard
  - When you are designing a representation in the world of code and your representation uses pointers (C or C++)
  - When you are stuck
  - When your code does not work

  Diagrams often make excellent documentation!
- Have the design planned out before coding [A lot goes unsaid here, but almost any *written* (or diagrammed) design before coding will pay significant dividends. Soon you'll get handouts on what kinds of properties to write down about designs.]
- Iteration: make a change and test [Helpful, but needs development. How big a change before you test?]
- Not waiting until the last minute [good advice, but needs more: what do you do *instead*?]
- Share ideas with others (3 groups)
- To avoid getting distracted by syntax, use pseudocode. [Very good advice; to be great advice, we need to be told when and how to do this.]
- Turn off the screen, visualize. [Very good advice, but *when* should you do this?]

Great goal, but not given technique to achieve it

- Compiling often
- Familiarity with your own code [how do you advise a programmer to achieve this goal?]
- Incremental testing [what's the increment?]
- Readability [Most of you will know this when you see it, but I will add that readability in *all* media demands *no tab characters* and a *width of at most 80 columns*.]
- Typing. [This skill is supremely important. For technique to acquire it, I recommend learning to touch-type by using a typing-tutor program three or four times a day, for 5 minutes each sesssion, for 3 weeks.]

Vague goal, learner could easily go wrong

- Abstraction [when and how?]
- Deliberate design [suggests that something happens before code, which is good, but not enough guidance]
- Do little piece by little piece [good as far is it goes, but what's a "piece"? what kinds of pieces are there? how big is "little?" when is a piece "done"?]
- Not coding all at once [Excellent advice but does not go far enough. What do you code? When? What activities are interleaved with coding? How? How often?]
- Know how to utilize tools (or using the right tools) [a good goal, but learners need more guidance: which tools are most important? which do you learn first? which tools can you live without, at least at first? how deeply do you have to know each tool? when in your career do you start revisiting tools to learn them more deeply?]
- Knowlege of libraries [libraries are like tools: it's good to know them, but there are way too many libraries, and to learn successfully you must prioritize. A good place to start is with standard C library as covered in Kernighan and Ritchie and with the parts of the Hanson library that are recommended for 40.]
- Outlining [What goes into an outline of software? What distinguishes a good outline from a bad outline? When do you do an outline? For example, do you outline before or after writing the code (or both)? How long do you keep it? Do you revise it in light of experience?]
- Rationing your time [it seems like there might be a kernel of a useful idea here, but by itself it's not enough. Robert Boice's research shows *work in "brief daily sessions" of 45–90 minutes* is a successful technique. To learn to apply it, I had to read several chapters of Boice's book—it's not so easy.]
- Spreading out coding over time [similar comments]
- Reuse code [how? why? when? how do you know if it's possible?]
- Refactoring [how? when?]

Well meant, but there is a better way

Here are some techniques that are OK, but in my experience there is a better alternative:

- Think of general functions, name them, set aside to organize.

  There is a trap for beginners: You Aren't Going to Need It (YAGNI). Programmers with less than five years of experience often spend time and effort trying to build general solutions to problems in which generality never arises.

  A better technique for arriving at general functions is not to create them *ab initio*, but to review code just written (or older code) and look for generality that could be

there but isn't. Then refactor the code to bring out the generality. Two examples from the course software:

- Raoul and I started with a function to check that standard output is not empty. We generalized to a function that can check any condition checkable by `/bin/test`.
- Raoul and I started with a function to check to be sure that given a particular test image, `brightness` writes to standard output, as required in the assignment. We generalized to a function that will check any executable binary on any input.

Motherhood and apple pie, not useful guidance

All these things sound good, but I can't tell how to put them into practice successfully:

- Comments [where? how? how much?]
- Clean comments [what's clean? how can you tell?]
- Good debugging skills; smart debugging (3 groups) [Sounds good, but what does it mean? How do you acquire them? What distinguishes good debugging from bad debugging?]
- Good documentation (3 groups) [What parts of a program are available to be documented? Is each part documented in the same way? What parts *should* be documented? What parts *must* be documented? For each part, what constitutes good documentation? What's bad documentation?]
- Modularity [an ill-defined goal with no guidance about how to achieve it] (4 groups)
- Planning [what do you plan? what do you *do* when planning?] (6 groups)
- Reusability [a goal with no guidance about how to achieve it] (2 groups)
- Teamwork
- Testing
- Time management; managing time

Personal qualities

Valuable qualities, but it may not always be obvious how to develop those qualities in yourself:

- Attention to detail
- Commitment
- Creativity
- Focus
- Foresight
- General problem solving
- An open mind
- Patience (2 groups)
- Start early
- Think out of the box

Platitudes

These are ideas that sound good and are uttered by people who mean well, but which have no agreed-upon definition in the profession. In some cases I have no idea what's being proposed. In other case I can suggest a related technique.

- Compartmentalization
- Cost-effective [compared to what? what's your cost? suggested replacements: *have a time budget* and *have a performance target*]
- Keep macro perspective
- Testing as you go. [This idea sounds really good, but the moment I try to put it into practice, I realize I don't have enough information. It's too big a topic to tackle in this handout.]
- User-friendly [suggested technique: *do a discount usability study*]
- Using resources

High-risk techniques

These are techniques that need to be used judiciously. Perhaps the cost/benefit ratio is not favorable, or perhaps there is a risk that the techniqe won't pay off at all.

- Unit testing; testing units as you write them (3 groups)

  Maybe half of the projects in COMP 40 will benefit from unit testing. For many projects, however, unit tests don't have much payoff, and you are better off going straight to integration testing. For some projects, like image compression, unit testing can have a very high payoff. I will try to guide you.

- Constant testing/revisions; lots of small testing

  Sometimes this idea goes under the name "continuous integration." It's great for big projects, but many of the projects in 40 are too small to benefit much. For many projects you'll benefit more from a half dozen tests that are very well thought out—and perhaps created *before* you begin coding.

- Understanding efficiency.

  Understanding is good, but there's a very great risk that an *understanding* of efficiency leads to paying too much *attention* to efficiency—and as noted below, that can lead to disaster.

Bad or misguided ideas about technique

- Clear goals.

  Sadly, in software construction as in life, goals are seldom clear. It's much more common for programmers to say we don't know what we're trying to build until we