[Handout for L1P1]

Software Engineering Rocks!

The following description of the Joys (and Woes) of the Programming Craft was taken from Chapter 1 of The *Mythical Man-Month*, by Frederick P. Brooks.

The Joys of the Craft

Why is programming fun? What delights may its practitioner expect as his reward?

First is the sheer joy of making things. As the child delights in his mud pie, so the adult enjoys building things, especially things of his own design. I think this delight must be an image of God's delight in making things, a delight shown in the distinctness and newness of each leaf and each snowflake.

Second is the pleasure of making things that are useful to other people. Deep within, we want others to use our work and to find it helpful. In this respect the programming system is not essentially different from the child's first clay pencil holder "for Daddy's office."

Third is the fascination of fashioning complex puzzle-like objects of interlocking moving parts and watching them work in subtle cycles, playing out the consequences of principles built in from the beginning. The programmed computer has all the fascination of the pinball machine or the jukebox mechanism, carried to the ultimate.

Fourth is the joy of always learning, which springs from the nonrepeating nature of the task. In one way or another the problem is ever new, and its solver learns something: sometimes practical, sometimes theoretical, and sometimes both.

Finally, there is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by the exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures....

Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.

Programming then is fun because it gratifies creative longings built deep within us and delights sensibilities we have in common with all men.

The Woes of the Craft

Not all is delight, however, and knowing the inherent woes makes it easier to bear them when they appear.

First, one must perform perfectly. The computer resembles the magic of legend in this respect, too. If one character, one pause, of the incantation is not strictly in proper form,

the magic doesn't work. Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.

Next, other people set one's objectives, provide one's resources, and furnish one's information. One rarely controls the circumstances of his work, or even its goal. In management terms, one's authority is not sufficient for his responsibility. It seems that in all fields, however, the jobs where things get done never have formal authority commensurate with responsibility. In practice, actual (as opposed to formal) authority is acquired from the very momentum of accomplishment.

The dependence upon others has a particular case that is especially painful for the system programmer. He depends upon other people's programs. These are often maldesigned, poorly implemented, incompletely delivered (no source code or test cases), and poorly documented. So he must spend hours studying and fixing things that in an ideal world would be complete, available, and usable.

The next woe is that designing grand concepts is fun; finding nitty little bugs is just work. With any creative activity come dreary hours of tedious, painstaking labor, and programming is no exception.

Next, one finds that debugging has a linear convergence, or worse, where one somehow expects a quadratic sort of approach to the end. So testing drags on and on, the last difficult bugs taking more time to find than the first.

The last woe, and sometimes the last straw, is that the product over which one has labored so long appears to be obsolete upon (or before) completion. Already colleagues and competitors are in hot pursuit of new and better ideas. Already the displacement of one's thought-child is not only conceived, but scheduled.

This always seems worse than it really is. The new and better product is generally not *available* when one completes his own; it is only talked about. It, too, will require months of development. The real tiger is never a match for the paper one, unless actual use is wanted. Then the virtues of reality have a satisfaction all their own.

Of course the technological base on which one builds is *always* advancing. As soon as one freezes a design, it becomes obsolete in terms of its concepts. But implementation of real products demands phasing and quantizing. The obsolescence of an implementation must be measured against other existing implementations, not against unrealized concepts. The challenge and the mission are to find real solutions to real problems on actual schedules with available resources.

This then is programming, both a tar pit in which many efforts have floundered and a creative activity with joys and woes all its own. For many, the joys far outweigh the woes....

Worked examples

[Q1]

(a) Compare Software Engineering to Civil Engineering in terms of how CE work products (i.e., buildings) differ from SE work products (i.e., software).

(b) Comment on this statement: Building software is cheaper and easier than building bridges (all we need is a PC!).
(c) Justify this statement: Coding is still a 'design' activity, not a 'manufacturing' activity. You may use a comparison/analogy of Software engineering vs Civil Engineering to prove this point.

[A1]

(a)

Buildings	**Software**
Visible, tangible	Invisible, intangible
Wears out over time	Does not wear out
Change is limited by physical restrictions (e.g., difficult to remove a floor from a high rise building)	Change is not limited by such restrictions. Just change the code and recompile.
Creating an exact copy of a building is impossible. Creating a near copy is almost as costly as creating the original.	Any number of exact copies can be made with near zero cost.
Difficult to move.	Easily sent from one place to another.
Many low-skilled workers following tried-and-tested procedures.	No low-skilled workers involved. Workers have more freedom to follow their own procedures.
Easier to assure quality (just follow accepted procedure).	Not easy to assure quality.
Majority of the work force has to be on location.	Can be built by people who are not even in the same country.
Raw materials are costly, costly equipments required.	Almost free raw materials and relatively cheap equipment.
Once constructions started, it is hard to do drastic changes to the design.	Building process is very flexible. Drastic design changes can be done, although costly.
A lot of manual and menial labor involved.	Most work involves highly-skilled labor.
Generally robust. E.g., removing a single brick is unlikely to destroy a building.	More fragile than buildings. A single misplaced semicolon can render the whole system useless.

(b) Depends on the size of the software. Manpower required for software is very costly. On the other hand, we can create a very valuable software (e.g., an iPhone application that can make million dollars in a month) with a just a PC and a few days of work!

(c) Arguments to support this statement:

* If coding is a manufacturing activity, we should be able to do it using robotic machines (just like in the car industry) or low-skilled laborers (like in the construction industry).

* If coding is a manufacturing activity, we wouldn't be changing it so much after we code software. But if the code is in fact a "design", yes, we would fiddle with it until we get it right.

* Manufacturing is the process of building a finished product based on the design. Code is the design. Manufacturing is what is done by the compiler (fully automated).

However, the type of 'designing' that happens during coding is much lower level than the 'designing' that happens before we start coding.

---End of Document---