Transaction versus transform flow

Transaction: command: A is a directive, and the output is not a transform of A, but rather, the result of executing the command A.

Transform: potentially asynchronous operation that takes data A and produces data B.
A -> B -> C

Dominance: B, e.g., describes the whole operation. A is preparation, C is postprocessing.

Lack of dominance: A, B, C are just steps in a process.

In choosing modules and submodules, look for cohesion
Example of cohesion
From last time:

General principles of module decomposition
Segregate the kinds of information flow
transaction flow
data flow
inheritance (condition flow)
Seek one or more "centers" from which to decompose modules
transaction center
transform center
root class
overarching goal (control module)
Build a basic module decomposition outward from the "centers"
(with some judgement calls as to which centers dominate the design)
Refine that decomposition by creating submodules for the centers.

Finally,
Assign modules to workgroups in a manner that achieves balance between workloads.
**How does one do that?**
How does one know a project is reasonable?

- Can it be done in the allocated time?
- Will it do what we want at the end?

But at a deeper level,

- Is value sufficiently greater than cost?
- Is cost similar to those of other similar projects?

Reading: SEPA Chapter 23, MMM Chapter 9.
Concept of a metric

A function $\mu: \{\text{things}\} \rightarrow \text{positive numbers}$

Where for all $X$ in $\{\text{things}\}$,

$\mu(X) \geq 0$

[Also $\mu(X \cup Y) \leq \mu(X) + \mu(Y)$]

If $\mu(X) < \mu(Y)$, then $X$ "is less complex than" $Y$. 
A **product metric** is a measure of the **complexity** of a software product. It is
- a function $\mu()$ satisfying $\mu(P) \geq 0$ for all products $P$.
- When $\mu(P) < \mu(Q)$, product $P$ is "less complex than" product $Q$.
- Common examples:
  - $\text{cost}(P) =$ cost of developing $P$.
  - $\text{LOC}(P) =$ lines of code in some version of $P$ in some language.
  - $\text{Function\_points}(P) =$ complexity factors in $P$'s description.
  - $\text{Branch\_complexity}(P) =$ $\#$ if statements $+ 1$
Some simple claims about metrics

Allow one to compare the complexity of two projects.
Are not easily converted into one another, because conversion requires context:
  How good are the programmers?
  How much experience do they have?
  How big is the organization?
Two kinds of metrics

- "**hard**" metrics can be measured directly, e.g., lines of code, cost, sales.
- "**soft**" metrics can only be measured indirectly, e.g., usability, marketability, etc.
Another characterization of metrics

**White-box:** computed with knowledge of the implementation (source code).

**Black-box:** computed without knowledge of implementation (just requirements)
A key idea in metrics: "covariance".

- Hard and soft metrics often vary together.
- If something is more usable, then it may take more lines of code.
- There is thus a naive "covariance" relationship between hard and soft concepts.
Note the role of statistics
Claims are valid over a space of reasonable projects
Claims I make will be statistically justified.

"If covariance between a hard and soft metric is high for a set of sample projects, and yours is like one of those, then you can predict the soft metric from the hard one and vice versa."
Metrics apply to different phases of the engineering process:

- LOC: only meaningful after program is written.
- "Function points": a way of estimating complexity from requirements.
Kinds of requirements metrics

- how many functions (capabilities) are there?
- how many discernable subsystems comprise the system?
- how many actors are there?
- how many branches in the control-flow diagram?
- how many data exchanges are there?
- how complex are the data structures?
Metrics applicable to designs

- **how many modules** are there?
- **how many total functions** are there?
- **how many libraries** (or library functions) are needed?
Metrics that measure implementations

- **lines of code (LOC):** how many lines of code were necessary to implement the project?
- **cyclomatic complexity:** how many branches were taken in the code? (plus one, see below!)
- **time spent implementing:** person-hours spent actually implementing the project.
Brooks Chapter 2: the Mythical Man-Month

- While people like to think of project complexity in terms of the number of hours to implement it
- Project implementation time is very sensitive to how the project is managed.
- "Adding people to a late project makes it later."
What we want is a method to accurately compute cost from requirements.
What we have are methods that estimate cost are hopelessly inaccurate in early parts of the project. get more accurate as the project progresses.
There is a complex relationship between requirements metrics and the **Jackson duality principle**.

- **branches in the requirements** must lead to **branches in the design**, which lead to **branches in the implementation**.
- **data exchanges** in the requirements must be in the design, and thus in the implementation.
- Thus there is an **imprecise relationship** between the metrics applicable to various stages of the project.
The metrics "game"

- You only know the real cost of the project when it is done.
- But management wants some estimate of the cost long before it is done.
- Cost can be estimated from complexity as it arises in each phase of the process.
- As implementation progresses, cost estimates become more accurate.
- At the end, you know the cost, which is the hardest known measure of complexity.
My approach: hard to soft

Monday, September 28, 2009
12:50 PM

Hard to soft:
- In talking about metrics, I will start with the **hardest known metrics** and proceed to the **softest**.
- As I proceed, I will move from **later** in the process to **earlier**.
- Every step I take back in time will be **less precise** than the one before.
- I will end with the **most imprecise metrics** we have, which are **complexity metrics on requirements**.
Metrics on maintenance

- lifecycle cost of maintaining the program, which varies with
- how many person-hours were spent in total on the project, which varies with
- the number of revisions necessary for the product, which varies with
- the number of deficiencies or changes desired.
Next step back: metrics on testing

- cost of testing varies with
- time spent testing, which varies with
- complexity of test plan and
- number of defects found, which vary with
- complexity of the overall project.
Beginning: what can you tell from the code
how long it took to write it.
how much you paid people.
how long the program is (lines of code)

Claim:
○ LOC varies with how long to implement,
○ which varies with money paid.

Which is wrong. See MMM.
The LOC controversy:

- Lines of code are a poor metric, because:
  - LOC varies greatly by language (100 lines of Perl has a very different complexity than 100 lines of C)
  - Some simple programs have high LOC, low complexity (e.g., matrix operations)
- Need a better metric that
  - is language-independent
  - better models costs
Solution to the LOC controversy: **cyclomatic complexity**
- The cyclomatic complexity of a program is the number of conditional branches in the program + 1.
- measurable from the source code or the assembler output!
- Proposer: McCabe.

McCabe's claims: cyclomatic complexity is
- language independent
- a better measure of the cost of testing and maintenance than LOC, and thus
- a better predictor of lifecycle cost than LOC.
Cyclomatic complexity and "if"

Cyclomatic complexity and C/C++
○ the cyclomatic complexity of
  if (x) y;
  is 2 (there are two paths through the program).
○ The cyclomatic complexity of
  if (x) y; else z;
  is 2 (there are two paths through the program).

Careful:
○ The cyclomatic complexity of
  if (x || y) z;
  is 3! (conditional execution of x || y)
○ The cyclomatic complexity of
  x = (y?z:w);
  is 2! (implicit inline if statement)
Also,
- the cyclomatic complexity of
  \[
  \text{switch (z) \{}
  \text{case a:}
  \text{    \ldots}
  \text{    break;}
  \text{case b:}
  \text{    \ldots}
  \text{    break;}
  \text{default:}
  \text{    \ldots}
  \text{\}}}
\]
  is the same as that of
  \[
  \text{if (z==a) \{}
  \text{    \ldots}
  \text{\} else if (z==b) \{}
  \text{    \ldots}
  \text{\} else \{}
  \text{    \ldots}
  \text{\}}}
\]
  which is 3.
The root of McCabe's claims: "basis testing"
- the cyclomatic complexity of a program is also the number of test cases necessary to execute all paths through the program in all useful contexts.
- Thus the difficulty of testing varies with the cyclomatic complexity.
- Which means that the cost varies likewise.
- We will learn about how to do this using "basis testing" later.
But... what if we just have requirements?

Cyclomatic complexity is based upon having the source (or object) code in hand.

What do we do if all we have is the design or the requirements?
Jackson duality to the rescue

- the **structure of the program** must follow the **structure of the requirements**.
- the number of "if" statements in the program is equal to the number of "if" statements in the requirements!
  - Some "if" statements are explicit in the **control flow diagram**
  - Some are **implicit attributes** of the **data** or **interfaces**.
- If we can **tease out** the number of if statements in the requirements, we can then estimate the cyclomatic complexity of the program and, from that, estimate its cost.
Function points:

- A measure of the complexity of requirements which vary with the complexity of the code that satisfies requirements which can be utilized to predict cost of creating the code.
basic idea of function points

- count the **number of branches** in the requirements
- count the **number of interfaces** with the outside world
  - weight each interface with how complex it is (e.g., implicit branching) to get an estimate of **interface complexity**.
- count the **number of data structures** listed in the requirements.
  - weight each data structure with how complex it is (e.g., variation, unions, etc) to get an estimate of **data structure complexity**.

Sum these up to get an estimate of **project complexity**.
Caveats for function points

- correlations between cost and complexity are actually site-relative.
  - It costs a lot less to develop software if you're Google than if you're a startup!
  - It costs a lot more if you have inexperienced staff.
  - It costs a lot more if you are understaffed or undersupported.
- So
  - the function of branch complexity (that is cost) varies with your environment as a programmer and
  - the weighting factors for program attributes also vary with where the software will be written!
Constructive cost modeling

○ input: a set of requirements and a description of the programming environment in which the project will be written.
○ output: an estimated project cost in programming time and/or dollars.
○ Justification: statistical studies of existing projects predict cost relationships to complexity!

http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html
Parts of COCOMO-II

○ Describe requirements
  ▪ options/branches
  ▪ interfaces to outside world
  ▪ via questionnaires!

  This determines both weighting factors and **overall project complexity.**

○ Describe programming environment
  ▪ programmer capability
  ▪ maturity/experience of staff
  ▪ via questionnaires!

  This determines weighting factors for **staff capabilities.**

○ Plug these into an **overall cost model** to predict costs.
COCOMO-II is very controversial because
- it only models environments that the authors have studied
- it decreases in accuracy as one deviates from one of the studied shops.
- It seldom gets closer than an order of magnitude away from actual project cost, e.g., the best one can say for sure is that
  predicted cost/10 ≤ actual cost ≤ predicted cost*10
- But it is better than nothing!