CS1530, LECTURE 9

DETAILED SOFTWARE DESIGN

CHARACTERIZING DESIGN

- **Consistency** - does the design use common terminology/patterns/logic across the system? Is there re-use and centralized locations for common functionality (e.g. math helper functions, error display, etc.)? Are similar functions treated in a similar way across the program?

- **Completeness** - Have all the requirements (or at least those agreed upon) been completed? Has the design of the software been carried out to the proper level of detail?

EXAMPLES OF POOR CONSISTENCY

- Run-tracking app - during run, shows average speed in miles/hour; after run, shows average speed in millimeters/fortnight
- Multiple, unconnected “help” subsystems - clicking “help” on one screen takes you to a different system than clicking “help” on another
- Two different message queueing systems which perform similar functions

EXAMPLE OF POOR COMPLETENESS

- System should use an architecture and do stuff
- Trade-offs: favor good things over bad
- Design constraints: should run on a computer

CAN WE QUANTIFY OUR SOFTWARE DESIGN?

- Difficult, but there have been attempts at developing **metrics** of system complexity
  - **Metric** - quantification of some aspect of the system
- Early attempts focused on implementation at level of code or methods, not system as a whole
- Often because systems were not at a much higher level than the code or a few modules!

HALSTEAD COMPLEXITY METRIC

- Four fundamental metrics:
  - n1 = Number of distinct operators (e.g. +, %)
  - n2 = Number of distinct operands (variables, constants)
  - N1 = sum of all occurrences of n1
  - N2 = sum of all occurrences of n2
- Two directly derived metrics:
  - n = n1 + n2 (Program vocabulary)
  - N = N1 + N2 (Program length)

HALSTEAD COMPLEXITY METRIC

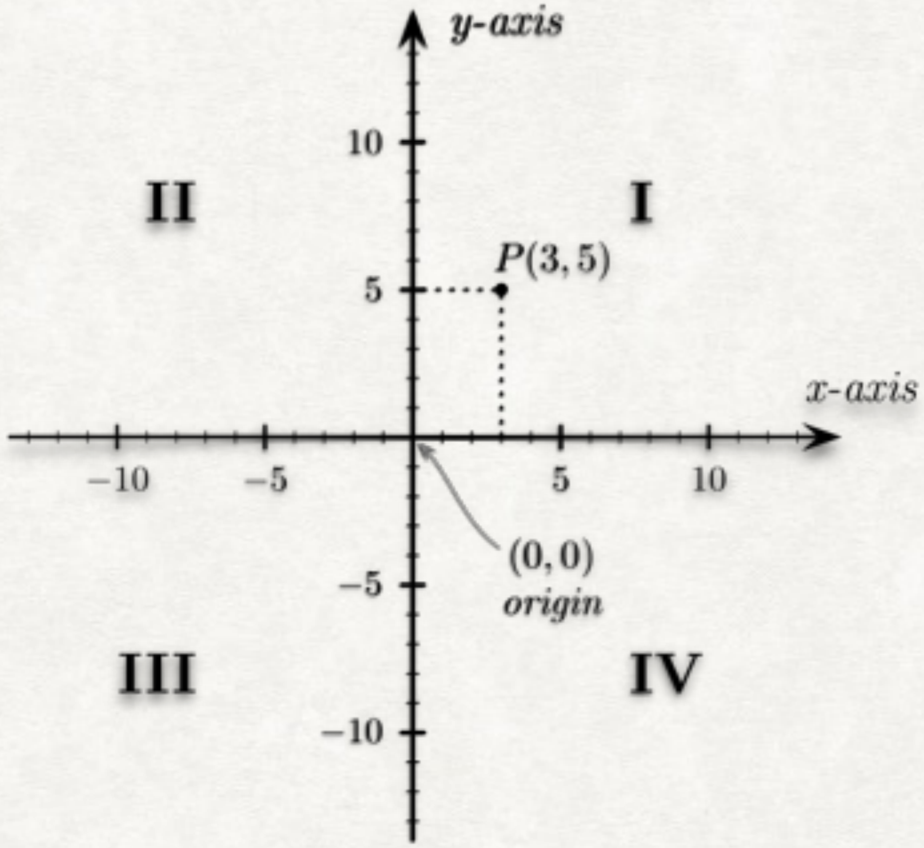
- Can derive effort, “volume”, etc. from these values
- Pro: Simple to calculate
- Con: Tell us about lexical complexity, not program complexity

MCCABE'S CYCLOMATIC COMPLEXITY

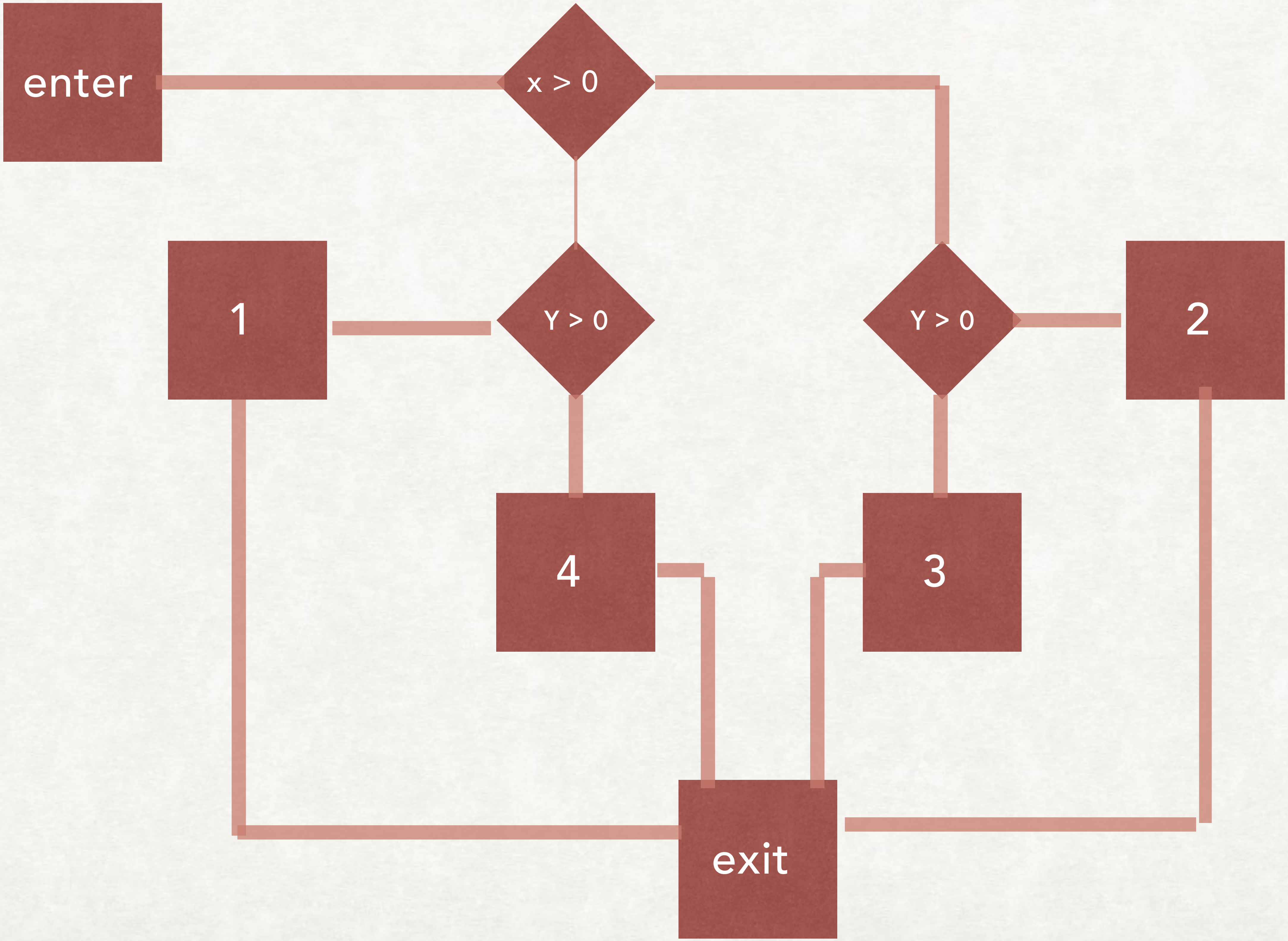
- Views a program's control flow through the lens of graph theory
- Given a method's control flow, calculate:
  - E = number of edges of graph
  - N = number of nodes of graph
  - p = number of connected components (usually 1)
  - Cyclomatic complexity = E - N + 2p
  - Also equal to the number of possible paths through a method

CYCLOMATIC COMPLEXITY - EXAMPLE

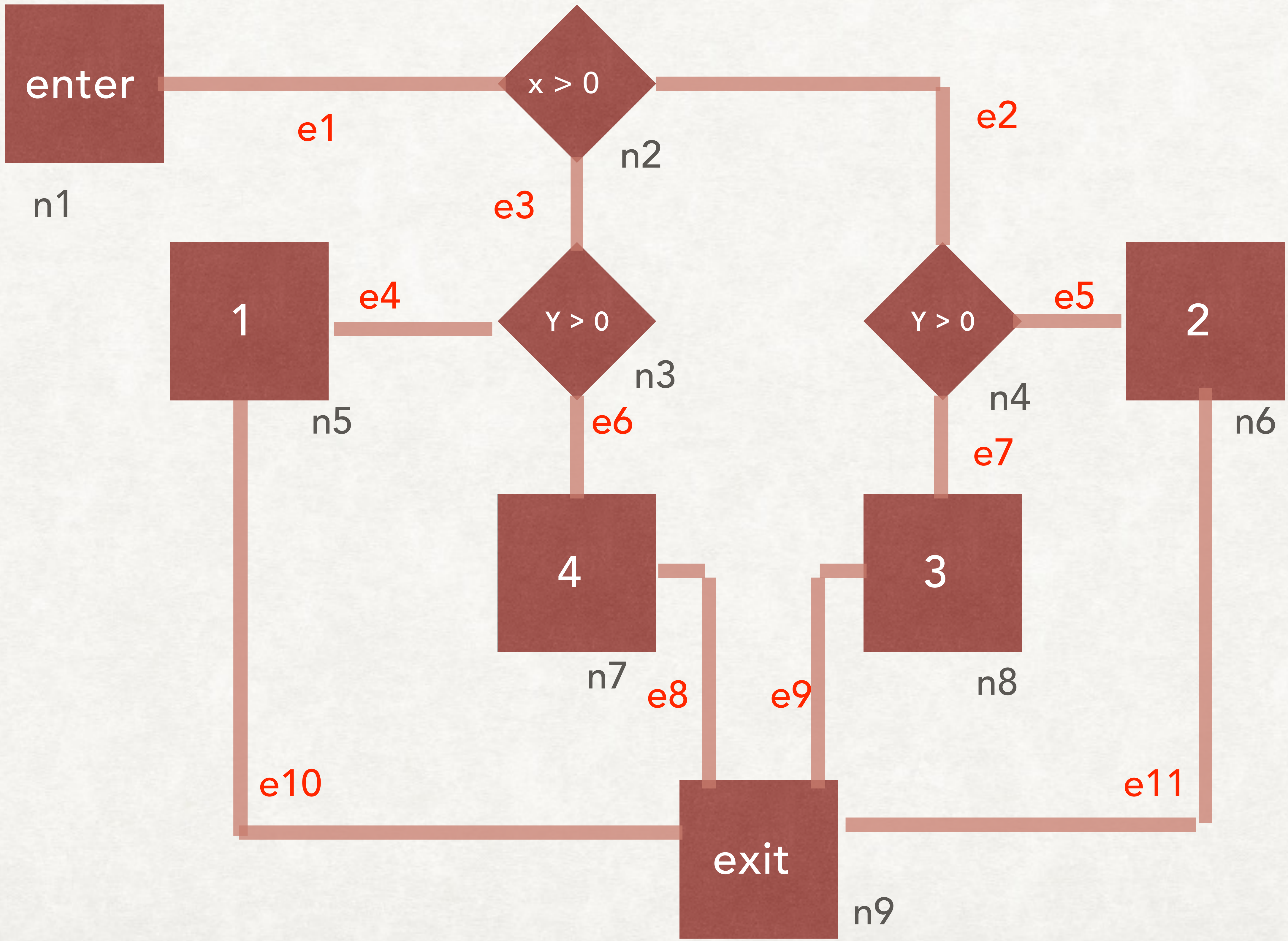
```
public int whichQuadrant(int x, int y) {
  int toReturn = -1;
  if (x > 0) {
     if (y > 0) {
        toReturn = 1;
     } else {
        toReturn = 4;
     }
  } else {
     if (y > 0) {
        toReturn = 2;
     } else {
        toReturn = 3;
     }
  }
  return toReturn;
}
```



CYCLOMATIC COMPLEXITY EXAMPLE



CYCLOMATIC COMPLEXITY EXAMPLE



CYCLOMATIC COMPLEXITY EXAMPLE

Edges = 11

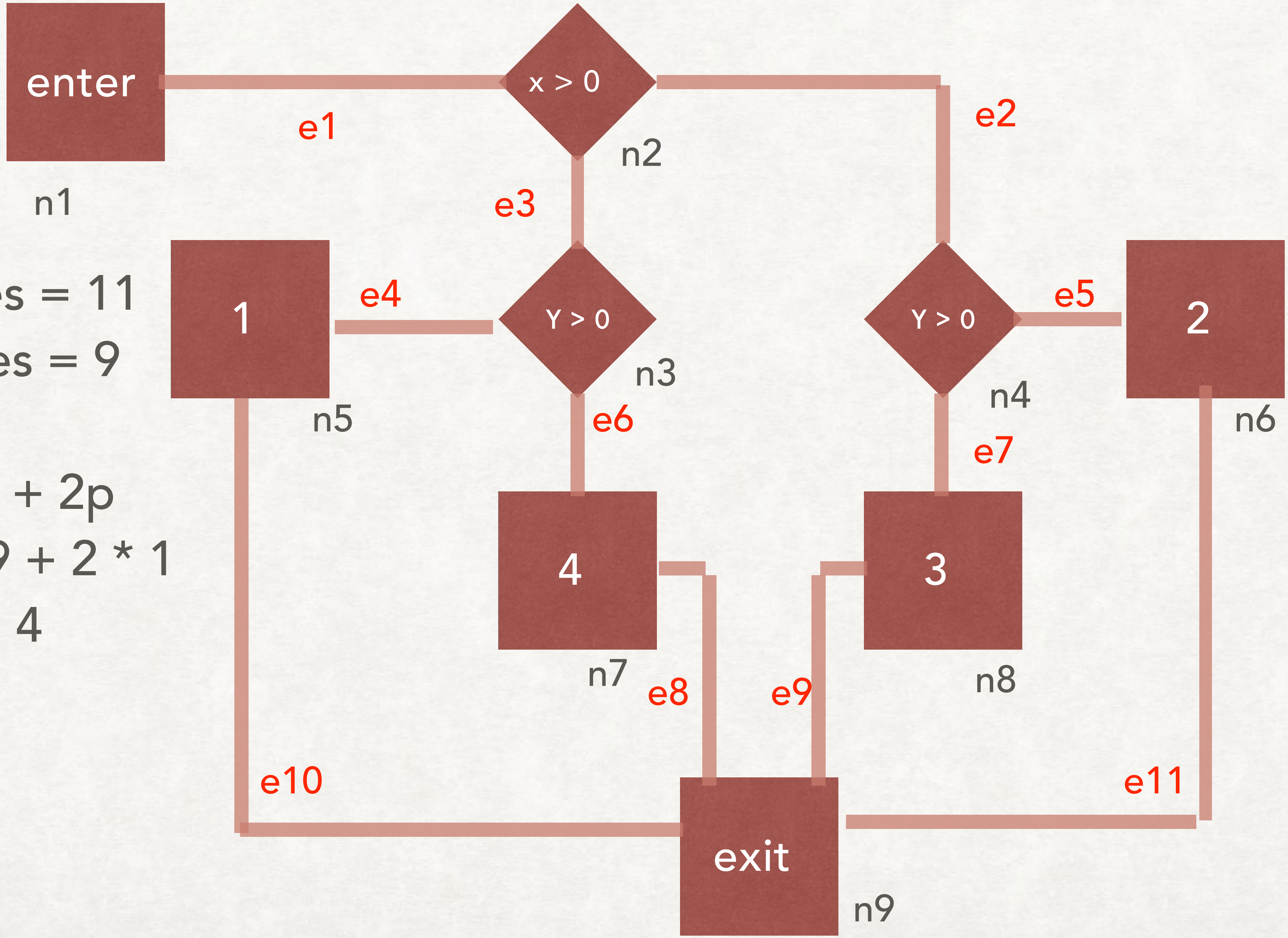
Nodes = 9

p = 1

E - N + 2p

11 - 9 + 2 * 1

CC = 4



UNDERSTANDING CYCLOMATIC COMPLEXITY

- The maximum number of linearly independent paths through the control flow of the method
- Lower cyclomatic complexity = lower risk, easier to understand
- < 10 = very simple, low risk
- > 50 = very complex, high risk

HENRY-KAFURA INFORMATION FLOW

DONEC QUIS NUNC

- Measures intramodular data flow (intramodular = depends on definition of module. Class, method, function, etc.)
- All information that goes IN to or OUT of a module, via
  - Parameter passing (arguments)
  - Reading / writing a global variable
  - Input (from user or external system)
  - Output (to display or external system)

FAN-IN VS FAN-OUT

- **Fan-in:** total count of information flow into a module
  - Accepting an argument
  - Accepting an input from a user
  - Reading a global variable
- **Fan-out:** total count of information flow out of a module
  - Returning a value
  - Sending data to another system
  - Writing to a global variable

CALCULATING HENRY-KAFURA INFORMATION FLOW

- C_p = structural complexity of an individual module (in this case, class)
- $C_p = (\text{fan-in} * \text{fan-out})^2$
- Calculate C_p for each module, then sum up for total structural complexity (ΣC_p)

HENRY-KAFURA EXAMPLE CALCULATION

- Assume a system with three methods:
  - Method 1: Accepts two arguments, reads one global variable, returns one integer (Fan-in: 3, Fan-out: 1. $C_p = (3 * 1)^2 = 9$)
  - Method 2: Accepts one argument, returns one integer (Fan-in: 1, Fan-out: 1. $C_p = (1 * 1)^2 = 1$)
  - Method 3: Accepts one argument, reads two global variables, writes one global variable, writes one output to file, returns one value (Fan-in: 3, Fan-out: 3. $C_p = (3 * 3)^2 = 81$) *Note dramatic growth in Cp here! Does not increase linearly!*
- System structural complexity = 9 + 1 + 81 = 91

HENRY-KAFURA INFORMATION FLOW: HENRY-SELIG MODIFICATION

- Multiply initial Henry-Kafura structural complexity by the internal complexity of the module (as measured by Halstead or McCabe's cyclomatic complexity)

- $HC_p = C_{ip} * (\text{fan-in} * \text{fan-out})^2$

- Calculate HCp of all modules and sum up (ΣHC_p)

- For either version (Henry-Kafura or Henry-Kafura-Selig) number, lower number should lead to a more understandable and simple design

HENRY-KAFURA-SELIG EXAMPLE CALCULATION

- Assume same system as before, with Method 1 with $C_p = 9$, Method 2 with $C_p = 1$, Method 3 with $C_p = 81$
    - Method 1 cyclomatic complexity = 4, $HC_p = 4 * 9 = 36$
    - Method 2 cyclomatic complexity = 2, $HC_p = 2 * 1 = 2$
    - Method 3 cyclomatic complexity = 10, $HC_p = 10 * 81 = 810$
- Total Henry-Kafura-Selig structural complexity = 36 + 2 + 810 = 848

CARD-GLASS COMPLEXITY MEASURES

- Three different complexity measures for some module *x*:
  - Structural complexity
    - $S_x = (\text{fan-out}_x)^2$
  - Data complexity (where P_x = number of variables passed to and from *x*)
    - $D_x = P_x / (\text{fan-out}_x + 1)$
  - System complexity (sum of structural complexity and data complexity)
    - $C_x = S_x + D_x$

A GOOD DESIGN SHOULD BE..

- Easy to understand
- Easy to change
- Easy to reuse
- Easy to test
- Easy to integrate
- Easy to code

Do Halstead complexity, McCabe cyclomatic complexity, Henry-Kafura, Card-Glass or other metrics really give us a good way to quantify these?

COUPLING AND COHESION

- A more fundamental (and perhaps more accurate) assessment might be measured by two characteristics:
  - *Cohesion* - Each module/component/class should be focused on a very specific purpose (*strong cohesion*), not dealing with a variety of functions (*weak cohesion*).
  - *Coupling* - Each module/component/class should interact with other modules/components/classes without being highly dependent on its implementation (*loose coupling*), not being bound to implementation decisions of other modules/components/classes (*tight coupling*).
  - Strive for strong cohesion and loose coupling - why?

STRONG COHESION - WHY?

- *Easier to understand* - A module which does one thing and does it well is going to be easier for new people to understand
- *Allows for good divisions of functionality* - No "god classes" which handle all sorts of things
- *Fixes are localized* - Changes can be made in one place, instead of having to deal with changes all over the codebase
- *Easy to change* - Modifications are in a known location
- *Easy to test* - Tests of a given module will be focused on a particular piece of functionality
- *Easier to code* - Modules line up with functionality (requirements)

LOOSE COUPLING - WHY?

- Easier to re-use code - Can have different modules accessing same functionality
- Reduces scope of errors - If there is a problem, will have more difficulty causing issues in other parts of the code
- Code easier to understand - Minimizes complex interactions between modules. Allows developers to maintain “myopia”

SPECTRUM OF COHESION - WEAK TO STRONG

- Coincidental - Code is in a module, but performs entirely unrelated tasks
- Logical - Code in module performs similar tasks
- Temporal - Code in module is performed at same or similar time
- Procedural - Code in module is related in terms of a control sequence
- Communicational - Design is related procedurally and targeted on the same data or kind of data
- Sequential - Module performs one specific goal or activity (vaguely defined)
- Functional - Module performs one very specific goal or activity (well-defined and delimited)

DEGREES OF COUPLING - TIGHT TO LOOSE

- *Content coupling* - Tightest level of coupling. Modules have access to each other's internal data and procedures. Modifying any aspect of one will generally involve modifications to the other.

- *Common coupling* - Two modules which refer to the same global variable, which may modify program flow or data.

- *Control coupling* - One module explicitly influences the logic of another software unit (e.g., passing in which method should be called by the receiver)

- *Stamp coupling* - Data - both necessary and superfluous - is passed between software modules (e.g. passing a Student object to a module which only needs the major)

- *Data coupling* - Only necessary data ia passed from one module to another (e.g., passing Student.major attribute to module)

MEASURING COUPLING

- Assign each level of coupling an equivalent value, lowest is best (1 = data coupling, 5 = content coupling)
- For each pair of modules *x* and *y*, identify the tightest coupling between the two. Assign this value to the variable *i*.
- Identify all the different ways that the modules are coupled and assign this value to the variable *n*.
- For each pairwise coupling, $C(x,y) = i + (n / n + 1)$
- Coupling of a system C(S) = median of the set $\{ C(x_n, x_m) \}$ for all n,m from 1 to j, where j = number of modules

OO DESIGN METRICS

- Design metrics specific to object-oriented design
- Previously covered metrics and issues - functional decomposition, strong cohesion, loose coupling - are still important!
- We will cover three of the six metrics covered in the book

WEIGHTED METHODS PER CLASS (WMC)

- Given a class *C*, assume *n* methods, $m_1..m_n$
- WMC = $\boldsymbol{\Sigma}(w_i)$ where $i = 1..n$, and *w* is the weight of each method
  - Can weight based on complexity measures, but often just 1
- More methods per class = higher WMC = more complex

DEPTH OF INHERITANCE TREE (DIT)

- Maximum length of a given class to its “root” class (superclass of its superclass of its superclass etc)
- Example: java.lang.Integer -> superclass java.lang.Number -> superclass java.lang.Object (DIT = 2)
- Research shows mixed results on how DIT relates to code quality!
  - Too deep - perhaps very complex, difficult to find and re-use code
  - Too shallow - perhaps code is not written in a good OO way, code not being re-used well

NUMBER OF CHILDREN (NOC)

- Number of immediate subclasses of a class
- Example: java.lang.Number -> subclasses: AtomicInteger, AtomicLong, BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short (NOC = 10)
- Research still ongoing on if/how this impacts quality