CSED332: Software Design Methods

Lecture 5: Metrics; Debugging; Assertions

Kyungmin Bae

Department of Computer Science and Engineering
POSTECH

Next Week: Lectures on Software Design and Analysis

http://sevlab.postech.ac.kr/facs18/invited-speakers

- POSCO International Center, 1st floor (large seminar room)
  - You need to attend at least one lecture (No class on Oct 11)
  - You can make up for "missed classes" by attending multiple lectures
- Edward A. Lee (University of California, Berkeley)
  - 9:30am, Wednesday, Oct 10
  - "What Good Are Formal Models?"
- Grigore Rosu (University of Illinois at Urbana-Champaign)
  - 9:30am, Thursday, Oct 11
  - "Design, Implementation and Verification of Blockchain Languages"
- Moonzoo Kim (KAIST)
  - 9:30am, Friday, Oct 12
  - "Lessons from Automated Analysis of Industrial SW for 15 Years"

What Can We Measure?

- Process
  - man hours, bugs reported, stories implemented, ...
- Product
  - size and complexity of code

What Can We Measure?: Process

- Number of people on project
- Time taken, money spent
- Bugs found or reported (by testers, developers, or users)
- Bugs fixed
- Features added

What Can We Measure?: Product

- Size of code
    - Number of files
    - Number of classes
    - Number of processes
- Complexity of code
    - Dependencies / Coupling / Cohesion
    - Depth of nesting
    - Cyclomatic complexity

Product or Process?

- Number of tests
- Number of failing tests
- Number of classes in design model
- Number of relations per class
- Size of user manual
- Time taken by average transaction

Measuring Size of System

- Lines of code (Source Lines of Code – SLOC)
- Number of classes, functions, files, . . .
- Function Points
- Questions
  - are they repeatable?
  - do they work on requirements model, design model, or code?

Lines of Code

- Easy to measure
- All projects produce it
- Correlates to time to build
- Not a very good standard

> *Measuring software productivity by lines of code is like measuring progress on an airplane by how much it weighs.*
>
> *– Bill Gates*

Lines of Code: Example

- Blanks? Comments?

```
void copy(char *p,*q) {while(*p) *q++ = *p++;}
```

```
void copy(char *p,*q) {
    while(p) {
        *q++ = *p++;
    }
}
```

Lines of Code: Language Matters

- Assembly code may be 2–3 times longer than C code
- C code may be 2–3 times longer than Java code
- Java code may be 2–3 times longer than ML code
- . . .

Lines of Code: When is It Valid?

- Same language
- Standard formatting
- Code has been reviewed

- Other types of size?

Halstead Volume

- Introduced by Maurice Howard Halstead in 1977

$$\text{Halstead Volume} = \text{total number of operators/operands} \\ * \log_2(\text{number of distinctoperators/operands})$$

- Approximates size of elements and vocabulary
  - less depending on formatting (still depending on languages)
  - linearly connected to LoC, but less intuitive

Example: Halstead Volume

```
void main() {
    int a, b, c, avg;
    scanf("%d %d %d", &a, &b, &c);
    avg = (a + b + c) / 3;
    printf("avg = %d", avg);
}
```

- Operators: `main`, `()`, `{}`, `int`, `scanf`, `&`, `=`, `+`, `/`, `printf`, `,`, `;`
- Operands: `a`, `b`, `c`, `avg`, `"%d %d %d"`, 3, `"avg = %d"`
- Halstead Volume $= 42 * \log_2(19) = 178.4$

Complexity

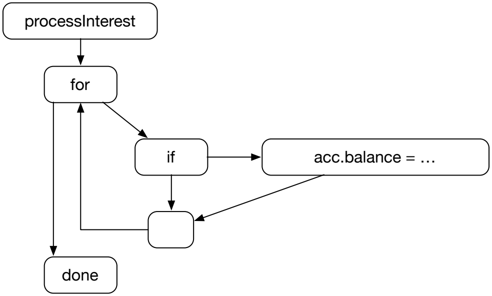
- Complex systems are
  - hard to understand
  - hard to change
  - hard to reuse
- Some metrics for complexity
  - Cyclomatic complexity
  - Cohesion and coupling
  - Function points
  - . . .

Cyclomatic Complexity

- A measure of logical complexity
    - Proposed by McCabe in 1976
- Tells how many tests are needed
    - to execute every statement of program
- Number of branches (if, while, for) + 1

Cyclomatic Complexity: Example

```
void processInterest() {
    for (Account acc : accounts) {
        if (hasInterest(acc)) {
            acc.balance =
                acc.balance + acc.balance * acc.interest;
        }
    }
}
```



Cyclomatic Complexity: How to Compute

- Number of predicates (branches) + 1
- Number of edges - number of nodes + 2
- Number of regions of the control flow graph

Cyclomatic Complexity: Testing View

- Cyclomatic complexity
  - the number of independent paths through the procedure
- Gives an upper bound on the number of tests
  - necessary to execute every edge of the control flow graph

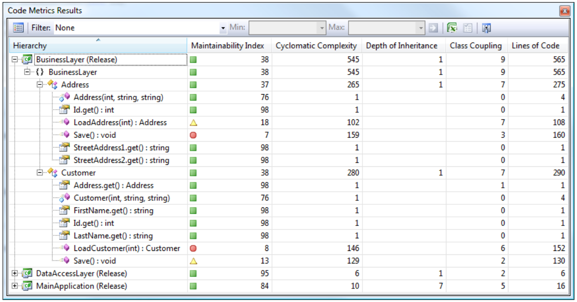
Cyclomatic Complexity: Metrics View

- Modules with a cyclomatic complexity greater than 10 were
  - hard to test and error prone (McCabe, 1976)
- Look for procedures with high cyclomatic complexity
  - rewrite them, focus testing on them, or focus reviewing on them

Maintainability Index

Visual Studio since 2007

Maintainability Index calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability. . . . A green rating is between 20 and 100 and indicates that the code has good maintainability. A yellow rating is between 10 and 19 and indicates that the code is moderately maintainable. A red rating is a rating between 0 and 9 and indicates low maintainability.



Code Metrics Results

Filter: None Min: Max:

Hierarchy		Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
BusinessLayer (Release)	🟩	38	545	1	9	565
{ } BusinessLayer	🟩	38	545	1	9	565
Address	🟩	37	265	1	7	275
Address(int, string, string)	🟩	76	1		0	4
Id.get() : int	🟩	98	1		0	1
LoadAddress(int) : Address	△	18	102		7	108
Save() : void	●	7	159		3	160
StreetAddress1.get() : string	🟩	98	1		0	1
StreetAddress2.get() : string	🟩	98	1		0	1
Customer	🟩	38	280	1	7	290
Address.get() : Address	🟩	98	1		1	1
Customer(int, string, string)	🟩	76	1		0	4
FirstName.get() : string	🟩	98	1		0	1
Id.get() : int	🟩	98	1		0	1
LastName.get() : string	🟩	98	1		0	1
LoadCustomer(int) : Customer	●	8	146		6	152
Save() : void	△	13	129		2	130
DataAccessLayer (Release)	🟩	95	6	1	2	6
MainApplication (Release)	🟩	84	10	7	5	16

Maintainability Index

- Maintainability Index =

$$\begin{aligned}&\max(0, (171 - \\ &\quad 5.2 * \log(\text{Halstead Volume}) - \\ &\quad 0.23 * (\text{Cyclomatic Complexity}) - \\ &\quad 16.2 * \log(\text{Lines of Code})) * 100/171)\end{aligned}$$

- Origines
  - Introduced in 1992 by Paul Oman and Jack Hagemeister
  - Developers rated a number of HP systems in C and Pascal
  - Statistical regression analysis to find key factors among 40 metrics

Discussion

- Metric seems attractive
- Easy to compute
- Often seems to match intuition

- Parameters seem often arbitrary
- Size dominates many metrics
- Rarely rigorously validated

Coupling and Cohesion

- Coupling
  - dependence among modules (often bad sign)
- Cohesion
  - dependence within modules
- Dependence
  - Call methods, refer to class, share variable
- Metrics for design model

Coupling and Cohesion: Examples

- Number and complexity of shared variables
  - functions in a module may share variables
  - functions in different modules should not
- Number and complexity of parameters
- Number of functions/modules that are called (fan-out)
- Number of functions/modules that call me (fan-in)

Dhama's Coupling Metric

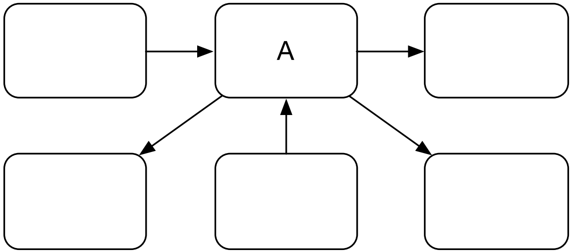
$$\begin{aligned}\text{Module coupling} = 1/(\, & \text{number of input parameters} + \\ & \text{number of output parameters} + \\ & \text{number of global variables used} + \\ & \text{number of modules called} + \\ & \text{number of modules calling})\end{aligned}$$

- 0.5 is low coupling
- 0.001 is high coupling

Martin's Coupling Metric

- Ca = afferent coupling (fan-in)
  - the number of classes outside this module that depend on classes inside this module
- Ce = efferent coupling (fan-out)
  - the number of classes inside this module that depend on classes outside this module
- Instability = $\mathrm{Ce} \,/\, (\mathrm{Ca} + \mathrm{Ce})$
  - 1 means highly unstable
  - 0 means stable (but hard to change)

Example: Martin's Coupling Metric

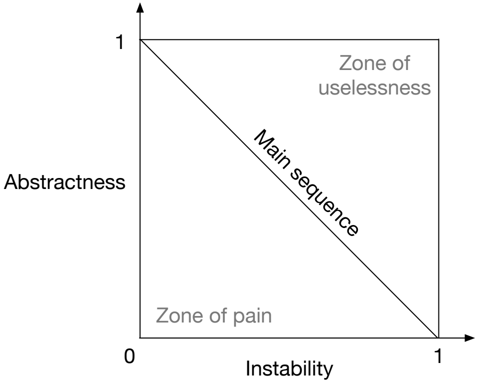


- $\mathrm{Ce} = 3$
- $\mathrm{Ca} = 2$
- $I = 3/(2+3) = 0.6$

Abstractness

Abstractness = number of abstract classes in module / number of classes in module

- 0 means completely concrete
- 1 means completely abstract



Function Points

- Way of measuring "functionality" of system
  - measure of how big a system ought to be (used to predict size)
  - metrics for the requirements model
- Several methods of computing function points
  - mostly complicated
  - most are proprietary

Function Points: How to Compute

- Count numbers
  - inputs, outputs, algorithms, tables in database, ...
- "Function points" is a function of above
  - plus fudge factor for complexity and developer expertise
- Need training to measure function points

Example: Function Points (1)

Information Domain Value	Count		Weighting factor				
			Simple	Average	Complex		
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
Count total							

$$FP = \text{count total} \times (0.65 + 0.01 \times \sum F_i)$$

R. S. Pressman. Software Engineering: A Practitioner's Approach

Example: Function Points (2)

- F_i $(1 \leq i \leq 14)$: value adjustment factors
- Responses to 14 questions (scale from 0 to 5), including
  - Does the system require reliable backup and recovery?
  - Are there distributed processing functions?
  - Is performance critical?
  - Does the system require online data entry?
  - Is the internal processing complex?
  - Is the code designed to be reusable?
  - ...

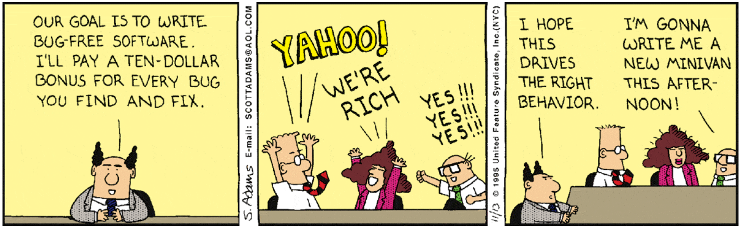
Product Metrics

- Requirements
  - function point, . . .
- Design
  - coupling, instability, abstractness, . . .
- Code
  - lines of code, cyclomatic complexity, halstead volume, maintainability index, . . .

One Way to Use Metrics

- Measure amount of code per month written by each programmer
- Give high producers big raise

⇒ Is this good or bad?


OUR GOAL IS TO WRITE BUG-FREE SOFTWARE. I'LL PAY A TEN-DOLLAR BONUS FOR EVERY BUG YOU FIND AND FIX.
S. Adams E-mail: SCOTTADAMS@AOL.COM
YAHOO!
WE'RE RICH
YES!!! YES!!! YES!!!
11/3 © 1995 United Feature Syndicate, Inc.(NYC)
I HOPE THIS DRIVES THE RIGHT BEHAVIOR.
I'M GONNA WRITE ME A NEW MINIVAN THIS AFTERNOON!

Another Way to Use Metrics

- Measure complexity of modules
- Pick the most complex and rewrite it

$\Rightarrow$ Is this good or bad?

Yet Another Way to Use Metrics

- Use function points
- Determine how many lines of code a system will require
- When you write that many lines, stop and deliver the system

⇒ Is this good or bad?

Yet^2 Another Way to Use Metrics

- Track progress
- Code reviews
- Planning

$\Rightarrow$ Is this good or bad?

Track Progress

- Manager review: manager periodically makes a report
  - Growth in SLOC
  - Bugs reported and fixed
  - SLOC per function point (user story)
  - SLOC per programmer (user stories per programmer)
- Information radiator: letting entire group know the state of project
  - green/red test runner (green/red lava lamp or traffic light)
  - wall chart of predicted/actual stories
  - wall chart of predicted/actual SLOC
  - webpage showing daily change in metrics (computed by daily build)

Code Reviews

- Look at metrics before a code review
    - Coverage - untested code usually has bugs
    - Large methods/classes
    - Code with many reported bugs

Planning

- Need to predict amount of effort
  - Make sure you want to do it
  - Hit delivery dates
  - Hire enough people
- Predict components and SLOC
- Predict stories & months (XP)
- Opinion of developers/tech lead/manager

Use Metrics Carefully

- Most software metrics are controversial
  - rarely rigorously validated (usually only plausibility arguments)
  - cyclomatic complexity was repeatedly refuted and is still used
- Know why you are measuring
  - what is the purpose of the measure?
  - what is the relationship of the attribute to the metric?

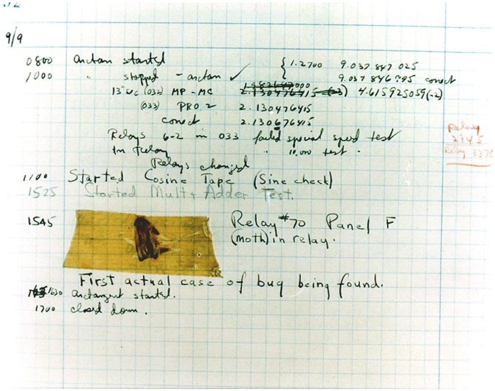
Debugging

Debugging

- Find out why a program is not functioning as intended
- Identify the root cause of an error and correct it
- Grace Hopper first used the word bug to describe computer glitch

Debugging

- Finding an actual moth in the computer (Mark II, 1947)



9/9

0800 arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 025
9.037 846 995 correct
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415 (-3)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 arctangent started.
1700 closed down.

Relay 2145
Relay 3370

Debugging vs. Testing

- What is the difference?
- How would you describe it in terms of failures and faults?

Debugging vs. Testing

- Testing finds a failure
  - failure = symptom: observe wrong result
  - produces bug reports, i.e., tests that fail
- Debugging finds/corrects the fault
  - fault = cause: mistake in the code
  - produces changes to the code

Debugging Process

- Reproduce failure (write automated test)
- Find and understand cause
- Fix the bug
- Add test case to regression suite

Debugging Process: Reproduce Failure

- Clarify symptom (simplify input), create minimal test
- Find small, repeatable test case that produces the failure
- Write automated test

$\Rightarrow$ Help identify the defect and gives you a regression test.

Debugging Process: Find and Understand Cause

- Traditional
  - Rubber duck debugging
  - Tracing and backtracking
  - Cause elimination
- Automated fault localization
  - Search (e.g., delta debugging)
  - Static analysis
  - Program slicing
  - ...

Rubber Duck Debugging

- Explain the problem to a friend who love to listen to you
  - or just to a rubber duck

https://pbs.twimg.com/media/B75vnDaIgAAXGDV.jpg:large

Tracing

- Watch execution of program
- Determine when it goes wrong
- Example
  - print statements
  - single stepping in debugger
  - log or trace execution (e.g., log4j)
  - (conditional) breakpoints
  - assertions / watchpoints

Backtracking

- Determine a failure
- Find fault by backward tracing from the failure
- Example
  - Repeatedly run program with new print statements
  - What could have caused this? Put in prints to test each possibility
  - Remove unneeded print/trace statements

Cause Elimination

- What could cause this bug?
- List all possibilities
- Check each one
  - by examining program
  - by performing experiments (tracing or testing)
- Requires knowledge of system and the errors that people make

Delta Debugging

- Figuring out which program change caused a bug
- Want to find a minimal set of changes that cause failure
- Search on (sets of) changes!

Zeller A. Yesterday, my program worked. today, it does not. why? ESEC/FSE '99.
G. Misherghi and Z. Su HDD: Hierarchical Delta Debugging. ICSE '06

Delta Debugging: Problem

- Given a set of changes $C = \{c_1, \ldots, c_n\}$ that causes a failure

$$\boxed{\checkmark} \Rightarrow \underbrace{\boxed{} \Rightarrow \cdots \Rightarrow \boxed{}}_{n \text{ changes}} \Rightarrow \boxed{\text{X}}$$

- Formalized as a function $hasFailure : 2^C \rightarrow Bool$

$$hasFailure(C) = true$$
$$hasFailure(\emptyset) = false$$

- Identify a minimal set of changes $D \subseteq C$ that causes a failure

$$hasFailure(D) = true$$
$$hasFailure(D') = false \quad \text{if } D' \subsetneq D$$

- This is in fact algorithmically very difficult (NP-complete)

Delta Debugging: Single Change

- There is a single change $d \in C$ that caused failure
  - given a set of changes $C = \{c_1, \ldots, c_n\}$
- Algorithm: Binary search! (logarithmic complexity)

$$DD(\{c_1, \ldots, c_n\}) = \begin{cases} c_1 & \text{if } n = 1 \\ DD(\{c_{\lfloor n/2 \rfloor + 1}, \ldots, c_n\}) & \text{if } \mathit{hasFailure}(\{c_{\lfloor n/2 \rfloor}, \ldots, c_n\}) \\ DD(\{c_1, \ldots, c_{\lfloor n/2 \rfloor - 1}\}) & \text{otherwise} \end{cases}$$

- Tool
  - `git bisect` (https://git-scm.com/docs/git-bisect)

Delta Debugging: 1-Minimal Set of Changes (1)

- A 1-minimal set of changes $D \subseteq C$ that caused a failure

$$\begin{aligned} \mathit{hasFailure}(D) &= \mathit{true} \\ \mathit{hasFailure}(D') &= \mathit{false} \quad \text{if } D' \subseteq D \ \wedge \ |D'| = |D| - 1 \end{aligned}$$

- (Naive) Algorithm:

```
function DD($D$) {
    for each $c \in D$:
        if hasFailure($D \setminus \{c\}$):
            return DD($D \setminus \{c\}$)
    return $D$
}
```

- Complexity: $O(n^2)$, even for a single change
  - better algorithm?

Delta Debugging: 1-Minimal Set of Changes (2)

- Try dividing change set into k subsets (initially, $k = 2$)
  - increase k if we cannot make progress
- Algorithm

```
function DD($D$, k) {
    split $D$ into $D_1 \cup \ldots \cup D_k$      // initially k = 2
    for each $1 \leq i \leq k$:
        if hasFailure($D_i$):
            return DD($D_i$, k)
        if hasFailure($D \setminus D_i$):
            return DD($D \setminus D_i$, k)
    return DD($D$, 2 * k)
}
```

- Complexity
  - worst case is still $O(n^2)$, but for a single failure, $O(\log n)$
  - there are several other improved versions of algorithms

Debugging Process: Fix the Bug and Add Test Case

- Fix the fault
  - fix the problem, not the symptom
  - you need understand the “true” cause
- Check your fix
  - whether the fix causes another bug
  - make sure that it does not occur elsewhere
- Add the test case to the regression test suite
  - store the input that elicited the bug
  - store the correct output

Example: Fix the Problem, Not the Symptom

- Failure: when *client* $= 45$, sum turns out to be wrong by 3.45.

```
for ( index = 0; index < numClaims[ client ]; index++ ) {
   sum[ client ] = sum[ client ] + claimAmount[ index ];
}
```

- Fix ??

```
for ( index = 0; index < numClaims[ client ]; index++ ) {
   sum[ client ] = sum[ client ] + claimAmount[ index ];
}
if ( client == 45 ) {
   sum[ 45 ] = sum[ 45 ] + 3.45;
}
```

- What if another failure is found for *client* $= 47$??

Debugging Expertise

- Expert debuggers know the system model
  - Can imagine how the system executes
  - Can perform mental experiments
  - Know their limits
- They know kinds of errors people make
  - Not closing a file
  - Not initializing an object correctly
  - Sending message to wrong object
- Error catalog
  - Testing and debugging depend on knowing common errors
  - Other people make the same kinds of mistakes that you do
  - Learn the kinds of mistakes people make (e.g., Testing Catalog)

Time Spent for Debugging

- Developers Spend Half Their Time Fixing Bugs[1]

Coding (50%)
Debugging(50%)

- How to decrease the time spent for debugging?
  - Find defects in earlier stages
  - Maintain high code quality

Find Defects in Earlier Stages

- During unit tests
- During integration tests
- During production (much harder to debug or fix)

Phase in Which Found	Cost Ratio
Requirements	1
Design	3–6
Coding	10
Development testing	15–40
Acceptance testing	30–70
Operation	40–1000

Boehm's analysis of 63 software development projects
(IBM, GTE, TRW, etc.), 1981

Maintain High Code Quality

- Continuous integration
- Code reviews
- Reduce cyclomatic complexity
- Emphasize simplicity (delete unnecessary code)

  Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. *– Brian Kernighan*

Assertions

Assertions

- Statements about program execution that are true or false, e.g.:
  - variable is an integer between 1 and 10
  - variable is not null
  - v1 $<$ v2
- Different than assertions in testing
  - a part of program code, not test code
- Reduce time spent for debugging
  - errors can be detected close to when/where they occur
  - easier to track down the source of a bug

Assertions in Java

- Assertion statements (since version 1.4)
  - `assert` Exp_1 `;`
  - `assert` Exp_1 `:` Exp_2 `;`
- If *Exp* evaluates to false, throw an `AssertionError` exception
  - Exp_2 is a string for a message of the exception
- Assertions in unit tests are completely different
  - `assertTrue`, `assertEquals`, ...

Example: Assertions in Java

```
public int pop() {
   assert !isEmpty() : "Stack is empty";
   return stack[--num];
}
```

```
public int abs(int number) {
    if (number < 0) {
        number *= -1;
    }
    assert number >= 0;
    return number;
}
```

Types of Assertions

- Checkpoint (like above)
- Pre-conditions
- Post-conditions
- Invariants

Pre and Post Conditions

- Some functions only work under certain conditions
- Preconditions
  - assertion that must be true before a function is executed
- Postconditions
  - assertion that is true after a function

Invariants

- Invariant is an assertion that should be always true
- Class invariant
    - precondition and postcondition of every method in the class
- Loop invariant
    - assertion that is true before, after, and every time through the loop

Example: What is the Class Invariant?

```
class ValueRange {
  int lower = Math.minInt();
  int upper = Math.maxInt();

  void setLower(int i) {
      if (i < upper) lower = i;
  }

  void setUpper(int i) {
      if (i > lower) upper = i;
  }
}
```

Example: Pre/Post Consitions and Invariants?

```
int sum(int n) {
    // precondition: ???
    int s = 0;
    int i = n;
    while(i > 0) {
        // invariant: ???
        s = s + i;
        i = i - 1;
    }
    // postcondition: ???
    return s;
}
```

Example: Pre/Post Consitions and Invariants?

```
int sum(int n) {
    assert n > 0; // precondition
    int s = 0;
    int i = n;
    while(i > 0) {
        assert s == sumFrom(i+1, n);  // invariant
        s = s + i;
        i = i - 1;
    }
    assert s == sumFrom(1, n); // postcondition
    return s;
}
```

Assertions as Programming Tool

- Execute assertions during testing and debugging
  - do not execute assertions in production code
- In Java, assertion is disabled by default
  - use option `-ea` or `-enableassertions` when running java
- Write preconditions for procedures
  - unit tests are preferred to postconditions
- Write assertions
  - to match original specifications
  - for places that errors occur
  - where you need them during debugging

Assertions as Specification

```
float sum(float[] array, int length) {
    float sum = 0.0;
    int i = 0;
    while (i < length) {
        sum = sum + array[i];
        i = i + 1;
    }
    return sum;
}
```

- Precondition:

$$\texttt{length} > 0 \quad \wedge \quad \texttt{array.length} = \texttt{length}$$

- Postcondition

$$\textit{result} \quad = \sum_{j=0}^{\texttt{length}} \texttt{array}[j]$$

Behavioral Subtyping: Liskov Substitution Principle

Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

— Barbara Liskov

- Principle of protection against variations in
  - different implementations of an interface
  - subclass extensions of a superclass
- Object of type T may be substituted with any object of its subtype
  - without altering any of the desirable properties of T
  - class invariants and contracts

Example: Car is Behavioral Subtype of Vehicle

```
abstract class Vehicle {
    int speed, limit;
    //invariant: speed < limit;







    //pre:  speed > 0
    //post: speed < \old(speed)
    void brake();
}
```

```
class Car extends Vehicle {
    int fuel;
    boolean engineOn;
    //invariant: speed < limit;
    //invariant: fuel >= 0;

    //pre:  fuel > 0 && !engineOn
    //post: engineOn
    void start() { ... }

    //pre:  speed > 0
    //post: speed < \old(speed)
    void brake() { ... }
}
```

- Subclass fulfills the same invariants (and additional ones)
- Overridden method has the same pre- and postconditions

Example: Hybrid is Behavioral Subtype of Car

```
class Car extends Vehicle {
    int fuel;
    boolean engineOn;
    //invariant: speed < limit;
    //invariant: fuel >= 0;

    //pre:  fuel > 0 && !engineOn
    //post: engineOn
    void start() { ... }

    //pre:  speed > 0
    //post: speed < \old(speed)
    void brake() { ... }
}
```

```
class Hybrid extends Car {
     int charge;
    //invariant: charge >= 0;

    //pre:  (charge > 0 || fuel > 0)
    //        && !engineOn;
    //post: engineOn;
    void start() { ... }

    //pre:  speed > 0
    //post: speed < \old(speed)
    //      && charge > \old(charge)
    void brake() { ... }
}
```

- Subclass fulfills the same invariants (and additional ones)
- Overridden method start has weaker precondition
- Overridden method brake has stronger postcondition

Question: Is Square Behavioral Subtype of Rectangle?

```java
class Rectangle {
    int h, w;
    //invariant: h > 0 && w > 0

    Rectangle(int h, int w) {
        this.h = h;
        this.w = w;
    }
}
```

```java
class Square extends Rectangle {
    //invariant: h > 0 && w > 0
    //invariant: h == w

    Square(int w) {
        super(w, w);
    }
```

Question: Is Square Behavioral Subtype of Rectangle?

```java
class Rectangle {
    int h, w;
    //invariant: h > 0 && w > 0

    Rectangle(int h, int w) {
        this.h = h;
        this.w = w;
    }

    //pre:  factor > 0
    void scale(int factor) {
        w = w * factor;
        h = h * factor;
    }
}
```

```java
class Square extends Rectangle {
    //invariant: h > 0 && w > 0
    //invariant: h == w

    Square(int w) {
        super(w, w);
    }
```

Question: Is Square Behavioral Subtype of Rectangle?

```
class Rectangle {
    int h, w;
    //invariant: h > 0 && w > 0

    Rectangle(int h, int w) {
        this.h = h;
        this.w = w;
    }

    //pre:  factor > 0
    void scale(int factor) {
        w = w * factor;
        h = h * factor;
    }

    //pre: neww > 0;
    void setWidth(int neww) {
        w = neww;
    }
}
```

```
class Square extends Rectangle {
    //invariant: h > 0 && w > 0
    //invariant: h == w

    Square(int w) {
        super(w, w);
    }
```

Summary

- Software metrics
  - process, requirements, design, code, . . .
- Finding bugs is only half the battle
  - write assertions and tests incrementally
- Homework 5 (Due 10/18)
  - Eclipse plugin development
- Reading
  - `https://en.wikipedia.org/wiki/Software_metric`
  - `https://en.wikipedia.org/wiki/Design_by_contract`

Question?