Software Metrics

Sudipto Ghosh
CS314

Questions

☐ How big is the program?
  ■ Huge!!

☐ How close are you to finishing?
  ■ We are almost there!!

☐ Can you, as a manager, make any useful decisions from such subjective information?

☐ You need quantitative information like, cost, effort, size of project.
Metrics

- Quantifiable measures that could be used to measure characteristics of a software system or the software development process
- Required in all phases
- Required for effective management
- Managers need quantifiable information, and not subjective information
  - Subjective information goes against the fundamental goal of engineering

Cyclomatic complexity

- Number of binary decisions + 1
- The number of branches in a module
- Proposed by McCabe
- Lower the value of this number, the better
- Only control complexity, no data complexity
- For OO, cyclomatic complexity is usually low because methods are mostly small
  - also, data component is important for OO, but ignored in cyclomatic complexity
Cyclomatic complexity (contd)

- Essentially the number of branches in a module
- Number of tests needed for branch coverage of a module
- Easily computed
- In some cases, good for predicting faults
- Validity questioned
  - Theoretical grounds
  - Experimentally

OO Metrics

- Number of Classes and Interfaces
- Afferent Couplings (Ca): The number of other packages that depend upon classes within the package is an indicator of the package's responsibility.
- Efferent Couplings (Ce): The number of other packages that the classes in the package depend upon is an indicator of the package's independence.
Chidamber-Kemerer Metrics
Weighted Methods per Class (WMC)

- A class $C$, with methods $M_1, \ldots, M_n$, each with complexity $c_1, \ldots, c_n$.
  - $WMC = \sum c_i$
- Number of methods and complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.
- Larger the number of methods, greater is the potential impact on the children.
- Classes with large number of methods are likely to be more application specific, lowering reuse possibilities.

Chidamber-Kemerer Metrics
Depth of Inheritance Tree (DIT)

- Depth of inheritance of the class.
  - For multiple inheritance, DIT is the max length from the class to the root of the tree.
- Measures how many ancestors can potentially affect this class.
- Deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it harder to predict its behavior.
- Deeper trees constitute greater design complexity.
- Deeper a class, greater the reuse potential of the inherited methods.
Chidamber-Kemerer Metrics
Number of Children (NOC)

- Number of immediate subclasses subordinated to a class in the class hierarchy
- Greater the number of children, greater the reuse
- Greater the number of children, greater the likelihood of improper abstraction of the parent class
- A class with a large number of children may require more testing of the methods in the class

Chidamber-Kemerer Metrics
Coupling between object classes (CBO)

- Number of other classes to which a class is coupled
- Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class it, the easier it is to reuse.
- Higher CBO means the class is sensitive to changes in other parts of the design, and thus maintenance is difficult
- Higher CBO requires more rigorous testing
Chidamber-Kemerer Metrics
Response for a class (RFC)

- RFC = |RS|, where RS is the response set for the class.
  - If $M$ = set of all methods in the class, and
  - $R_i$ = set of methods called by the $i$th method of the class, then
  - $RS$ = Union of all the $R_i$'s with $M$

- A large value of RFC indicates that testing and debugging becomes more complicated.
- A large RFC indicates greater class complexity.

Chidamber-Kemerer Metrics
Lack of cohesion in Methods (LCOM)

- Consider a class $C$, with $n$ methods $M_1$, ..., $M_n$.
- Each method $M_i$ used a set of instance variables $I_i$.
- There are $n$ such sets $I_1$, ..., $I_n$. Consider them pairwise.
- Let $P = \{ (I_i, I_j) \mid I_i \cap I_j = \phi \}$
- Let $Q = \{ (I_i, I_j) \mid I_i \cap I_j \neq \phi \}$
- $LCOM = |P| - |Q|$, if $|P| > |Q|$
  - $= 0$ Otherwise
Kinds of software metrics

- **Product metrics**
  - quantify characteristics of the product being developed
    - size, reliability

- **Process metrics**
  - quantify characteristics of the process being used to develop the software
    - efficiency of fault detection

Issues [1]

- **Cost of collecting metrics**
  - Automation is less costly than manual method
  - CASE tool may not be free
    - Development cost of the tool
    - Extra execution time for collecting metrics
  - Interpretation of metrics consumes resources

- **Validity of metrics**
  - Does the metric really measure what it should?
  - What exactly should be measured?
Issues [2]

- Selection of metrics for measurement
  - Hundreds available and with some cost
- Basic metrics
  - Size (like LOC)
  - Cost (in $$$)
  - Duration (months)
  - Effort (person-months)
  - Quality (number of faults detected)

Selection of metrics

- Identify problems from the basic metrics
  - high fault rates during coding phase
- Introduce strategy to correct the problems
- To monitor success, collect more detailed metrics
  - fault rates of individual programmers
Utility of metrics

- LOC
  - size of product
  - take at regular intervals and find out how fast the project is growing
- What if # defects per 1000 LOC is high?
  - Then even if the LOC is high, most of the code has to be thrown away.

Applicability of metrics

- Throughout the software process, like
  - effort in person-months
  - staff turnover
  - cost
- Specific to a phase
  - LOC
  - # defects detected per hour of reviewing specifications
Planning: Cost estimation

- Client wants to know:
  - How much will I have to pay?

- Problem with
  - underestimation (possible loss by the developer)
  - overestimation (client may offer bid to someone else)

- Cost
  - internal (salaries of personnel, overheads)
  - external (usually cost + profit)

Cost estimation

- Other factors:
  - desperate for work - charge less
  - client may think low cost => low quality, so raise the amount

- Too many variables
  - Human factors
    - Quality of programmers, experience
    - What if someone leaves midway
  - Size of product
Planning: Duration estimation

- Problem with underestimation
  - unable to keep to schedule, leading to
    - loss of credibility
    - possible penalty clauses

- Problem with overestimation
  - the client may go to other developers

- Difficulty because of similar reasons as for cost estimation

Metrics: planning - size of product

- Units for measurement
  - LOC = lines of code
  - KDSI = thousand delivered source instructions

- Problems
  - creation of code is only a part of the total effort
  - effect of using different languages on LOC
  - how should one count LOC?
    - executable lines of code?
    - data definitions
    - comments? What are the pros and cons?
Problems with lines of code

- More on how to count
  - Job control language statements?
  - What if lines are changed or deleted?
  - What if code is reused?
- Not all code is delivered to clients
  - Code may be for tool support
- What if you are using a code generator?
- Early on, you can only estimate the lines of code. So, the cost estimation is based on another estimated quantity!!!

Techniques of cost estimation

- Take into account the following:
  - Skill levels of the programmers
  - Complexity of the project
  - Size of the project
  - Familiarity of the development team
  - Hardware
  - Availability of CASE tools
  - Deadline effect
Techniques of cost estimation

- Expert judgment by analogy
- Bottom up approach
- Algorithmic cost estimation models
  - Based on mathematical theories
    - resource consumption during s/w development obeys a specific distribution
  - Based on statistics
    - large number of projects are studied
  - Hybrid models
    - mathematical models, statistics and expert judgement

Metrics: requirements phase

- Number of requirements that change during the rest of the software development process
  - if a large number changed during specification, design, ..., something is wrong in the requirements phase
- Metrics for rapid prototyping
  - Are defect rates, mean-time-to-failure useful?
  - Knowing how often requirements change?
  - Knowing number of times features are tried?
Metrics: design phase

- Number of modules (measure of size of target product)
- Fault statistics
- Module cohesion
- Module coupling
- Cyclomatic complexity
- Fan-in, fan-out

Coupling

- Coupling occurs when there are interdependencies between one module and another
  - When interdependencies exist, changes in one place will require changes somewhere else.
  - A network of interdependencies makes it hard to see at a glance how some component works.
Example of content coupling

```java
public class Line {
    private Point start, end;
    ...
    public Point getStart() { return start; }
    public Point getEnd() { return end; }
}

public class Arch {
    private Line baseline;
    ...
    void slant(int newY)
    {
        Point theEnd = baseline.getEnd();
        theEnd.setLocation(theEnd.getX(),newY);
    }
}
```

Example of control coupling

```java
public routineX(String command) {
    if (command.equals("drawCircle"){
        drawCircle();
    } else {
        drawRectangle();
    }
}
```
Example of stamp coupling

```java
public class Emailer {
    public void sendEmail(Employee e, String text)
    {
        ...
    }
}
```

Using simple data types to avoid it:

```java
public class Emailer {
    public void sendEmail(String name, String email, String text)
    {
        ...
    }
}
```

Example of stamp coupling

Using an interface to avoid it:

```java
public interface Addressee {
    public abstract String getName();
    public abstract String getEmail();
}

public class Employee implements Addressee {...}

public class Emailer {
    public void sendEmail(Addressee e, String text)
    {
        ...
    }
}
```
Cohesion

- A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things
  - This makes the system as a whole easier to understand and change
  - Type of cohesion:
    - Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

Metrics: implementation phase

- Intuition: more complex modules are more likely to contain faults
- Redesigning complex modules may be cheaper than debugging complex faulty modules
- Measures of complexity:
  - LOC
    - Assume constant probability of fault per LOC
    - Empirical evidence: number of faults related to the size of the product
  - Cyclomatic complexity
  - Chidamber-Kemerer OO metrics
Metrics: implementation and integration phase

- Total number of test cases
- Number of tests resulting in failure
- Fault statistics
  - Total number of faults
  - Types of faults
    - misunderstanding the design
    - lack of initialization
    - inconsistent use of variables
- Statistical-based testing:
  - zero-failure technique

Metrics: inspections

- Purpose: measure effectiveness of inspections
  - may reflect deficiencies of the development team, quality of code
- Measure fault density
  - Faults per page - specs and design inspection
  - Faults per KLOC - code inspection
  - Fault detection rate - #faults / hour
  - Fault detection efficiency - #faults/person-hour
Metrics: maintenance phase

- Metrics related to the activities performed. What are they?

- Specific metrics:
  - total number of faults reported
  - classifications by severity, fault type
  - status of fault reports (reported/fixed)