CSE 470
Software Engineering

Testing - SIX

Imran Zahid
Lecturer
Computer Science and Engineering, BRAC University

Recap

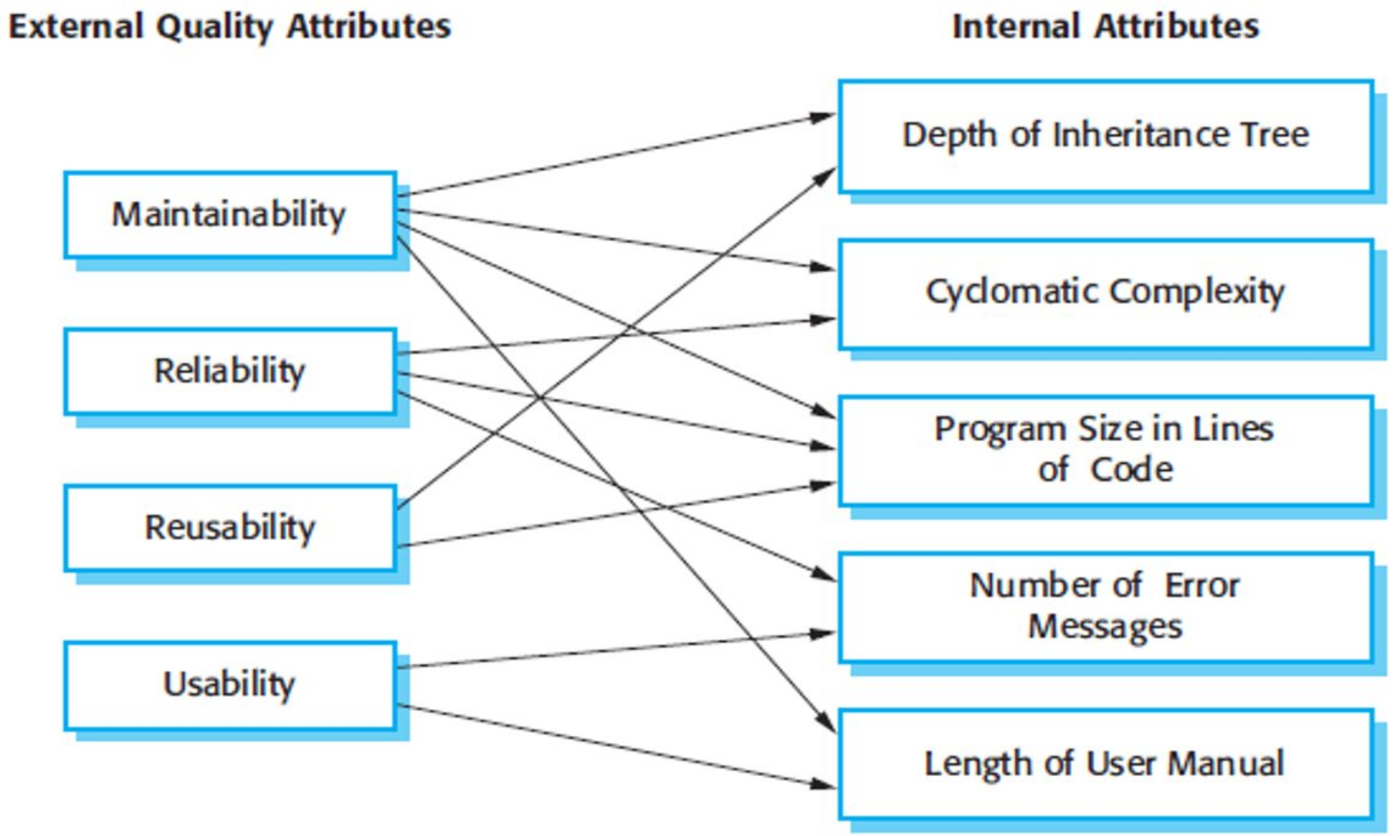
Software measurement and metrics

- Software measurement is concerned with deriving a numeric value for an attribute of a software product or process.
- This allows for objective comparisons between techniques and processes.
- Although some companies have introduced measurement programmes, most organisations still don’t make systematic use of software measurement.
- There are few established standards in this area

Software metric

- Any type of measurement which relates to a software system, process or related documentation
- Lines of code in a program, the Fog index, number of person-days required to develop a component.
- Allow the software and the software process to be quantified.
- May be used to predict product attributes or to control the software process.
- Product metrics can be used for general predictions or to identify anomalous components.

Relationships between internal and external software



Product metrics

- A quality metric should be a predictor of product quality.
- Classes of product metric
  - Dynamic metrics which are collected by measurements made of a program in execution;
  - Static metrics which are collected by measurements made of the system representations;
  - Dynamic metrics help assess efficiency and reliability
  - Static metrics help assess complexity, understandability and maintainability.

Other Metrics

Fan-in/Fan-out | Length of code

- Fan-in/Fan-out
  - Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes to X will have extensive knock-on effects. A high value for fan-out suggests that the overall complexity of X may be high because of the complexity of the control logic needed to coordinate the called components.
- Length of code
  - This is a measure of the size of a program. Generally, the larger the size of the code of a component, the more complex and error-prone that component is likely to be. Length of code has been shown to be one of the most reliable metrics for predicting error-proneness in components

Defect Removal Efficiency

- A defect is found when the application does not conform to the requirement specification.
- A mistake in coding is called Error
- An average DRE score is usually around 85% across a full testing program.
- DRE = E / (E + D) where:
- E is the number of errors found before delivery of the software to the end-user
- D is the number of defects found after delivery.
- We found 100 defects during the testing phase and then later, say within 90 days after software release (in production), found five defects,
-   DRE = 100/(100+5) = 95.2%

Specialization Index (SIX)

Specialization Index (SIX)

The Specialization Index metric measures the extent to which subclasses override their ancestors classes. This index is the ratio between the number of overridden methods and total number of methods in a Class, weighted by the depth of inheritance for this class

The metric provides a percentage, where the class contains at least one operation. For a root class, the specialization indicator is zero. Nominal range is between 0 % and 120 %.

Specialization Index (SIX)

Calculated from a number of variables determining the level at which subclasses override their ancestor classes.

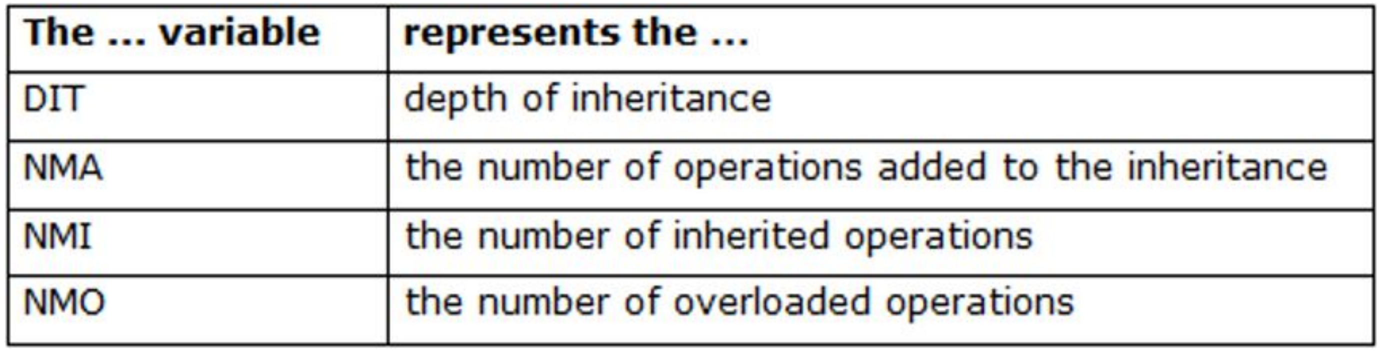
DIT = Depth of Inheritance

NMA = Number of Methods Added (to the inheritance)

NMI = Number of Inherited Methods

NMO = The number of Overridden Operation

Specialization Index (SIX)



The ... variable	represents the ...
DIT	depth of inheritance
NMA	the number of operations added to the inheritance
NMI	the number of inherited operations
NMO	the number of overloaded operations

SIX is determined from the above variables using the following equation:

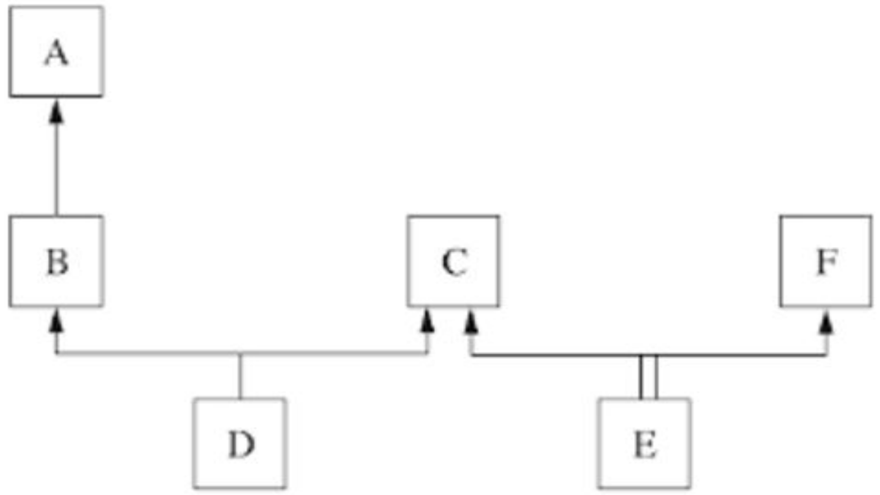
$$SIX = \frac{NMO \ * DIT}{NMI + NMA + NMO} * 100\%$$

Depth of Inheritance Tree (DIT)

Depth of Inheritance Tree (DIT) is the length of the longest path from a class to a root class in the inheritance tree. It measures how deep a class is located within the inheritance tree.

DIT(D) = 2

DIT(E) = 1



Number of Methods Added (NMA)

- The number of methods that are uniquely defined in the subclass and do not exist in its parent class.
- These methods provide new functionality that is specific to the subclass and are not part of the inheritance from the parent class.

Number of Methods Inherited (NMI)

- The number of methods directly inherited by the subclass from its parent class without any modification.
- These methods are used in their original form as defined in the parent class.
- Overridden methods will be included once (**don’t include them in NMI**). NMI will only include methods that were directly inherited, but NOT overridden.

Number of Methods Overridden (NMO)

- The number of methods in the parent class that are redefined (overridden) in the subclass to provide a different or specialized behavior.
- These methods exist in both the parent and subclass, but the subclass provides its own implementation.

Calculating SIX

```
class Animal:
    def eat(self): ...

    def sound(self): ...

class FlyingThing:
    def fly(self): ...

class Bird(Animal, FlyingThing):
    def sound(self): ...

    def hunt(self): ...
```

$$SIX = \frac{NMO * DIT}{NMI + NMA + NMO} * 100\%$$

Overridden methods will be included once (don’t include them in NMI)
So, the SIX of the class **Bird:**

$$SIX = \frac{1 * 1}{2 + 1 + 1} * 100\%$$

$$= (1/4) * 100\%$$

$$= 25\%$$

Calculating SIX

```
class LivingBeing:
    def grow(self): ...

    def breathe(self): ...

class Animal(LivingBeing):
    def eat(self): ...

    def sound(self): ...
```

```
class Machine:
    def start(self): ...

    def stop(self): ...

class RobotDog(Animal, Machine):
    def sound(self): ...

    def recharge(self): ...

    def walk(self): ...
```

Calculating SIX

```
class LivingBeing:
    def grow(self): ...

    def breathe(self): ...

class Animal(LivingBeing):
    def eat(self): ...

    def sound(self): ...
```

```
class Machine:
    def start(self): ...

    def stop(self): ...

class RobotDog(Animal, Machine):
    def sound(self): ...

    def recharge(self): ...

    def walk(self): ...
```

Calculating SIX

```
class LivingBeing:
    def grow(self): ...

    def breathe(self): ...

class Animal(LivingBeing):
    def eat(self): ...

    def sound(self): ...
```

```
class Machine:
    def start(self): ...

    def stop(self): ...

class RobotDog(Animal, Machine):
    def sound(self): ...

    def recharge(self): ...

    def walk(self): ...
```

So, the SIX of the class **RobotDog:**

$$SIX = \frac{NMO \times DIT}{NMI + NMA + NMO} \times 100\%$$

$$SIX = \frac{1 \times 2}{5 + 2 + 1} \times 100\%$$

$$SIX = \frac{2}{8} \times 100\% = 25\%$$

Calculating SIX - Overriding in Parent

```
class LivingBeing:
    def grow(self): ...

    def breathe(self): ...

class Animal(LivingBeing):
    def breathe(self): ...

    def sound(self): ...
```

```
class Machine:
    def start(self): ...

    def stop(self): ...

class RobotDog(Animal, Machine):
    def sound(self): ...

    def recharge(self): ...

    def walk(self): ...
```

We are only concerned with which methods the class itself overrides. Not those overridden by a parent class.

Calculating SIX - Overriding in Parent

```
class LivingBeing:
    def grow(self): ...

    def breathe(self): ...

class Animal(LivingBeing):
    def breathe(self): ...

    def sound(self): ...
```

```
class Machine:
    def start(self): ...

    def stop(self): ...

class RobotDog(Animal, Machine):
    def sound(self): ...

    def recharge(self): ...

    def walk(self): ...
```

So, the SIX of the class **RobotDog:**

$$SIX = \frac{NMO \times DIT}{NMI + NMA + NMO} \times 100\%$$

$$SIX = \frac{1 \times 2}{4 + 2 + 1} \times 100\%$$

$$SIX = \frac{2}{7} \times 100\% \approx 28.57\%$$

Ideal Values for SIX

- The SIX value should ideally fall within a moderate range to ensure a balance between specialization and reuse, providing reliability and reusability.
- A low SIX (close to 0%) may indicate excessive reliance on inherited methods, potentially reducing the distinct functionality of the subclass.
- Conversely, a high SIX (above 120%) may suggest over-specialization, which could lead to poor reusability and increased maintenance effort.

Thank you