Software Evolution

Prof. A. M. Calvagna

Software Evolution

- Program evolution dynamics
- Software maintenance
- Complexity and object oriented metrics

Software change

- Software change is inevitable
  - New requirements emerge when the software is used;
  - The business environment changes;
  - Errors must be repaired;
  - New computers and equipment is added to the system;
  - The performance or reliability of the system may have to be improved.
- A key problem for organisations is implementing and managing change to their existing software systems.

Program evolution dynamics

- Program evolution dynamics is the study of the processes of system change.
- After major empirical studies, Lehman and Belady proposed that there were a number of 'laws' which applied to all systems as they evolved.
- There are sensible observations rather than laws. They are applicable to large systems developed by large organisations. Perhaps less applicable in other cases.

Lehman's laws

Law	Description
Continuing change	A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.
Organisational stability	Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.

Applicability of Lehman's laws

- Lehman's laws seem to be generally applicable to large, tailored systems developed by large organisations.
- It is not clear how they should be modified for
  - Shrink-wrapped software products;
  - Systems that incorporate a significant number of COTS components;
  - Small organisations;
  - Medium sized systems.

Software maintenance

- Modifying a program **after** it has been put into use.
- Maintenance does not normally involve major changes to the system's architecture.
- Changes are implemented by **modifying** existing components and **adding** new **components** to the system.
- Studies have shown that most maintenance effort is spent on a relatively small number of system components.

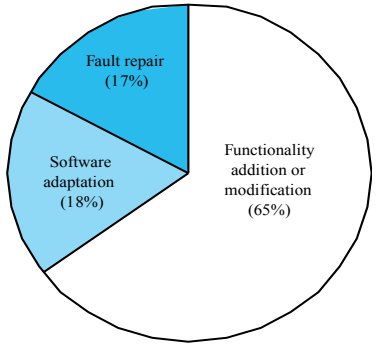
Maintenance is inevitable

- The system requirements are likely to **change while** the system is being **developed** because the environment is changing. Therefore a delivered system won't meet its requirements!
- Systems are tightly coupled with their environment. When a system is installed in an environment **it changes that environment** and therefore changes the system requirements.
- Systems MUST be maintained therefore if they are **to remain useful** in an environment.

Types of maintenance

- Maintenance to **repair** software faults
  - Changing a system to correct deficiencies in the way meets its requirements.
- Maintenance to **adapt** software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- Maintenance to **add** to or **modify** the system's functionality
  - Modifying the system to satisfy new requirements.

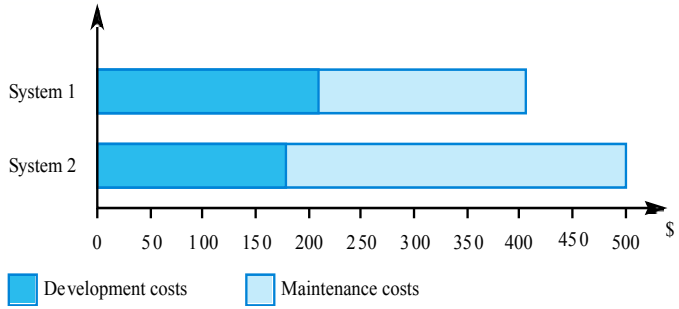
Distribution of maintenance effort



Maintenance costs

- Usually greater than development costs (2* to 100* depending on the application).
- Affected by both technical and non-technical factors.
- Increases as software is maintained. **Maintenance corrupts** the software structure so makes further maintenance more difficult.
- Ageing software can have high support costs (e.g. old languages, compilers etc.).

Development/maintenance costs



Complexity metrics

- Predictions of maintainability can be made by assessing the complexity of system components.
- Studies have shown that most maintenance effort is spent on a relatively small number of system components.
- Complexity depends on
  - Complexity of control structures;
  - Complexity of data structures;
  - Object, method (procedure) and module size.

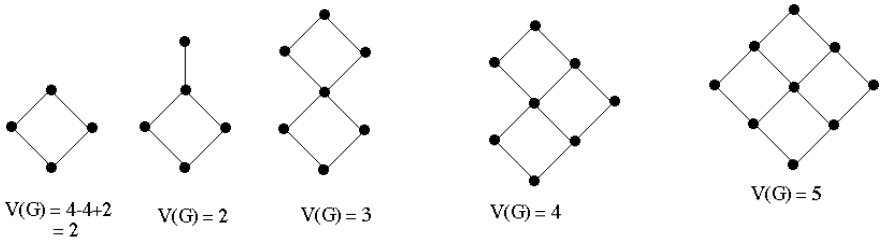
McCabe's Cyclomatic Complexity

- Logic Structure Metric
- Let G be a connected, directed acyclic graph (DAG) with e edges and n nodes

The **cyclomatic complexity (**or cycle rank) of G is defined as:

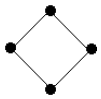
$$V(G) = e - n + 2$$

McCabe's Cyclomatic Complexity



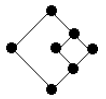
McCabe's Cyclomatic Complexity

```
if ( A< B ) then
      W;
else
      X;
endif
A = 3;
```



V(G)=2

```
if ( A< B ) then
      W;
else
    if ( B < C ) then
        X;
    else
        Z;
    endif
endif
```



V(G)=3

Standard CFGs

```
1 int f()
2 {
3     int x,y;
4     x = 10;
5     y = x+3;
6     return y;
7 }
```

Figure 7.11: CFG for Sequence

```
1 int f(int a)
2 {
3     int x=0;
4     if (a>10)
5         x=a;
6     return x;
7 }
```

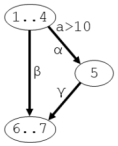


Figure 7.12: CFG for Selection (if-then)

```
1 int f(int a)
2 {
3     int x=0;
4     if (a>10)
5         x=a;
6     else
7         x=-a;
8     return x;
9 }
```

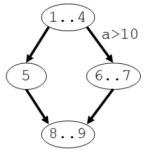


Figure 7.13: CFG for Selection (if-then-else)

```
   ...
1 switch (a) {
2     case 0:
3         b=33;
4         break
5     case 1:
6         b=-44;
7         break;
8     default:
9         ok=false;
10        break;
11 }
   ...
```

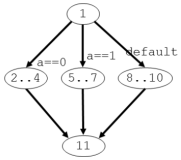


Figure 7.14: CFG for Selection (switch)

More Standard CFGs

```
1 int f(int a)
2 {
3     int x=a;
4     while (x>10)
5         x=x/2;
6     return x;
7 }
```

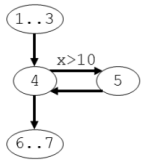


Figure 7.15: CFG for Iteration (while)

```
1 int f(int a)
2 {
3     int x=a;
4     do {
5         x=x/2;
6     } while (x>10);
7     return x;
8 }
```

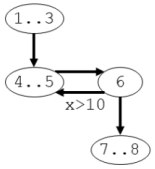


Figure 7.16: CFG for Iteration (do-while)

```
1 int f(int a)
2 {
3     int x=0;
4     for (int i=0; i<a; i++)
5         x = x+a+i;
6     return x;
7 }
```

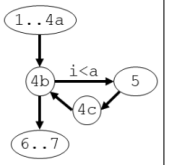


Figure 7.17: CFG for Iteration (for)

Altre metriche comuni

Dimensione

LOC o SLOC (lines of source code), oppure NCNB (non comment non blank)

*"A line of code is **any line** of program text that is **not a comment or blank** line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements."*

Comment Percentage

Percentuale di commenti rispetto a LOC (consigliato 30%)

How do you measure data complexity?

Data Complexity Measures

1. Measures of overlaps in feature values from different data classes.
2. Measures of separability of classes.
3. Measures of geometry, topology, and density of manifolds.

Complexity analysis is a technique to characterise the time taken by an algorithm with respect to input size (independent from the machine, language and compiler).

Chidamber & Kemerer OO Metric Suite

- WMC (Weighted Methods per Class)
- DIT (Depth of Inheritance Tree)
- NOC (Number of Children of a Class)
- CBO (Coupling Between Object Classes)
- RFC (Response for a Class)
- LCOM (Lack of Cohesion of Methods)

Metric 1: (WMC) Weighted Methods Per Class

Let C be a class, with methods M_1 , M_2 ,... , M_k

Let c_1 , c_2 ,... , c_k be the complexity of the methods

Then WMC = $c_1 + c_2 + ... + c_k$

indicates how much time and effort is required to develop and maintain the object

The larger WMC…

the greater the potential impact on the children

Objects are likely to be more application specific, limiting the possible reuse

Metric 2: (DIT) Depth of Inheritance Tree

The **depth of inheritance** of the class is the DIT metric for the class

The deeper a class is in the hierarchy, the more methods it is likely to inherit, making it more complex

Deeper trees indicate greater design complexity (but also reuse)

A high DIT has been found to increase faults. A recommended DIT is 5 or less.

Metric 3: (NOC) Number of Children

NOC = the number of immediate sub-classes of the class

Depth is generally better than breadth in class hierarchy, since it promotes reuse of methods through inheritance

A high NOC can indicate several things:

- High reuse of base class. Inheritance is a form of reuse.
- Base class may require more testing.
- Improper abstraction of the parent class.
- Misuse of sub-classing. In such a case, it may be necessary to group related classes and introduce another level of inheritance.

High NOC has been found to indicate fewer faults. This may be due to high reuse, which is desirable.

Metric 4: (CBO) Coupling between Objects

CBO = number of classes to which a class is coupled

Two classes are coupled when methods declared in one class use **methods or instance variables** of the other

Fan Out = # classes used by me

Fan In = # classes using me

CBO = FanOut - FanIn

High CBO is undesirable. Excessive coupling (>14) between object classes is detrimental to modular design and prevents reuse.

The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult.

A high coupling has been found to indicate fault-proneness.

Metric 5: (RFC) Response for a Class

RFC = NLM + NRM

where:
NLM = number of local methods in the class
NRM = number of remote methods called by methods in the class

How much code (methods) is executed by a class in response to calls

A given remote method is counted only once

The larger the RFC figure:
testing and debugging becomes more complicated
the greater the complexity of the object
the harder it is to understand

Metric 6: (LCOM1) Lack of Cohesion in Methods

Chidamber and Kemerer (1993). LCOM or LOCOM or LCOM1

Let C be a class, with methods M1 , M2 ,... , Mm

Let Ik = the set of instance variables used by method Mk

Let P = { (Ik, Ij) | Ik and Ij do not intersect }

Let Q = { (Ik, Ij) | Ik and Ij do intersect }

If | P | > | Q | then

LCOM = | P | - | Q |

else

LCOM = 0

Metric 6: (LCOM) Lack of Cohesion in Methods

Henderson-Sellers LCOM*.

Class has a set of m methods, M1 , M2 ,... , Mm

And a set of data attributes, A1 , A2 ,... , Aa

Let m(Ak) = number of methods that access data Ak

Then:

$$LCOM^* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a} m(A_j)\right) - m}{1 - m}$$

NASA Object-Oriented systems study

NASA STUDY	System analyzed #1	System analyzed #2	System analyzed #3
Language	**Java**	**Java**	**C++**
Classes	**46**	**1000**	**1617**
Lines	**50,000**	**300,000**	**500,000**
Quality	**"Low"**	**"High"**	**"Medium"**
CBO	2.48	1.25	2.09
LCOM1	447.65	78.34	113.94
RFC	80.39	43.84	28.60
NOC	0.07	0.35	0.39
DIT	0.37	0.97	1.02
WMC	45.7	11.10	23.97

Laing, Victor & Coleman, Charles: Principal Components of Orthogonal Object-Oriented Metrics. White Paper Analyzing Results of NASA Object-Oriented Data. SATC, NASA, 2001. http://satc.gsfc.nasa.gov/support/OSMASAS_SEP01/Principal_Components_of_Orthogonal_Object_Oriented_Metrics.pdf

Metrics Case Study

Sharble & Cohen (1993)

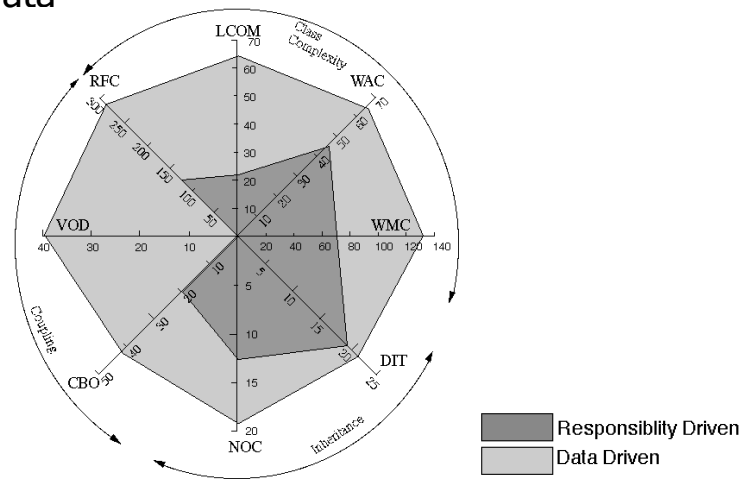
Two OO designs of same system were studied

One design was produced using responsibility-driven methodology

One design using data-driven methodology

Data Driven	Respon. Drive		
130	71	WMC	Weighted Methods Per Class
21	19	DIT	Depth of Inheritance Tree
19	13	NOC	Number of Children
42	20	CBO	Coupling between Objects
293	127	RFC	Response for a Class
64	21	LCOM	Lack of Cohesion in Methods
40	0	VOD	Violations of Demeter
64	44	WAC	Weighted Attributes per Class

Case Study Data



15 min. break