



SENG 421: Software Metrics

Object-Oriented Metrics (Chapter 11)

Department of Electrical & Computer Engineering, University of Calgary

B.H. Far (far@ucalgary.ca)

<http://www.enel.ucalgary.ca/People/far/Lectures/SENG421/11/>



Contents

- ▶ Object-Oriented measurement concepts
- ▶ Basic metrics for OO systems
- ▶ OO analysis and design metrics
- ▶ Metrics for productivity measurement
- ▶ Metrics for OO software quality
- ▶ Experience-based guidelines



What is Object-Orientation?

- *Structure-oriented programming* is not enough to represent (dynamic) software behaviour.
- *Data-oriented programming* is not enough to represent (dynamic) software behaviour, too.
- *Object-Oriented programming* is a methodology for system analysis, design and implementation that supports integration of functional and data-oriented programming and system development.



Terminology /1

| | |
|--|---|
| Class | A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. |
| Object | An instantiation of some class which is able to save a state (information) and which offers a number of operations to examine or affect this state. |
| Attribute Variable | An attribute is a named property of a class that describes a range of values instances of the property may hold. |
| Operation Responsibility Method | An operation is the implementation of a service that can be requested from any object of the class to affect behaviour. |





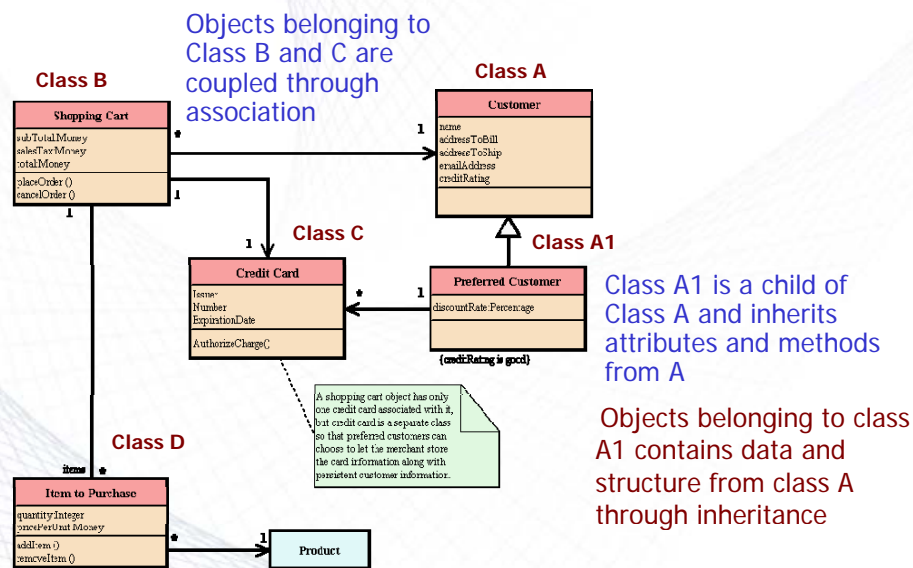
Terminology /2

| | |
|--------------------|--|
| Package | A package is a general purpose mechanism for organizing elements into groups. Packages group functionally related classes. |
| Cohesion | The degree to which the methods within a class or classes in a package are related to one another. |
| Coupling | Object X is coupled to object Y if and only if X sends a message to Y. |
| Association | A semantic relationship between two or more classes that specifies connections among their instances. |
| Inheritance | A relationship among classes, wherein an object in a class acquires characteristics from one or more other classes. |



Terminology /3

CLASS DIAGRAM : ELECTRONIC SHOPPING CART





Similarities & Differences

- Why Object-Oriented Software Engineering (OOSE) metrics are different from the conventional software metrics?
- OOSE metrics are different because of:
 1. Localization
 2. Encapsulation
 3. Information hiding
 4. Inheritance
 5. Reuse



1. Localization /1

- Localization means placing items in close (physical) proximity to each other.
 - Functional decomposition (localize information around functions).
 - Data localization (localize information around data).
 - Object-oriented approaches (localize information around objects).
- In conventional software engineering, localization is based on functionality. Therefore:
 - Metrics gathering has traditionally focused on functions and functionality
 - Units of software were set to be functional, thus metrics focusing on component relationships emphasized functional interrelationships, e.g., module coupling and cohesion.





1. Localization /2

- In object-oriented software, however, localization is based on *objects*. This means:
 - Metrics identification and gathering effort must recognize the “object” as the basic unit of software.
 - Within systems of objects, the localization between functionality and objects is not a one-to-one relationship.
 - For example, one function may involve several objects, and one object may provide many functions.



2. Encapsulation

- Encapsulation is the packaging (or binding together) of a collection of items:
 - Low-level examples of encapsulation include records and arrays, procedures, subroutines.
 - OO programming languages allow higher-level encapsulating, e.g., classes in C++ and Java.
- Encapsulation has two major impacts on metrics:
 - The basic unit will no longer be the program, but rather the object.
 - We have to modify our thinking on characterizing and estimating systems.





3. Information Hiding

- Information hiding is the suppression (or hiding) of details.
- The general idea is that we show only that information which is necessary to accomplish our immediate goals.
- There are degrees of information hiding, ranging from partially restricted visibility (e.g., public or private operations) to total invisibility (e.g., subsystems).
- Encapsulation and information hiding may not be the same thing, e.g., an item can be encapsulated but may still be totally visible (e.g., a package).
- Information hiding plays a direct role in such metrics as object coupling and the degree of information hiding.



4. Inheritance

- Inheritance is a mechanism whereby one object acquires characteristics from one, or more, other objects.
 - Some OO languages support single inheritance (e.g., Java), some support multiple inheritance (e.g., C++).
- Many OO software engineering metrics are based on inheritance, e.g.:
 - Number of children (number of immediate specializations)
 - Number of parents (number of immediate generalizations)
 - Class hierarchy nesting level (depth of a class in an inheritance hierarchy)





5. Reuse

- In OO development, reuse is a central issue
 - Reuse of libraries or frameworks
 - Reuse through inheritance
 - Meta-code level reuse:
 - Patterns
 - Business objects
- Reuse changes development process
 - Build reusable components
 - Find and reuse components



1. OO Project Metrics

What we want to measure in an *OO project*?

- Number of Classes, Operations (Methods), Attributes (Variables)
 - Lines Of Code (LOC) and Statement Count
Total and/or Averaged by class and/or method
- Structural measurement:
 - Coupling, Cohesion





2. OO Package Metrics

What we want to measure for a *package*?

- Number of Classes, Operations (Methods), Attributes (Variables)
Averaged by class and/or method
- Structural measurement:
 - Coupling, Cohesion
 - Maximum Inheritance Depth



3. OO Class Metrics

What we want to measure for a *class*?

- Number Attributes and Operations
- Lines of code (LOC) and statement count
- Inheritance related metrics
- Collaborators (Cohesion and Coupling related metrics)



4. OO Attribute Metrics

What we want to measure for an *attribute*?

- Instance variables
- How many times used



5. OO Operation Metrics

What we want to measure for an *operation*?

- Number of local variables
- Lines of code (LOC) and statement count
- Cyclomatic Complexity



OO Metrics Suite

- Object Oriented Metrics seek to measure the unique attributes of Object Oriented design as opposed to software developed using other methods.
- Chidamber and Kemerer felt that software metrics developed with traditional methods in mind did not readily lend themselves to Object Oriented notions such as classes, inheritance, encapsulation and message passing (Chidamber & Kemerer, 1993).
- They proposed 6 metrics unique to Object Oriented systems. These metrics measure various attributes including size and complexity and are constructed with a strong degree of theoretical and mathematical rigor.



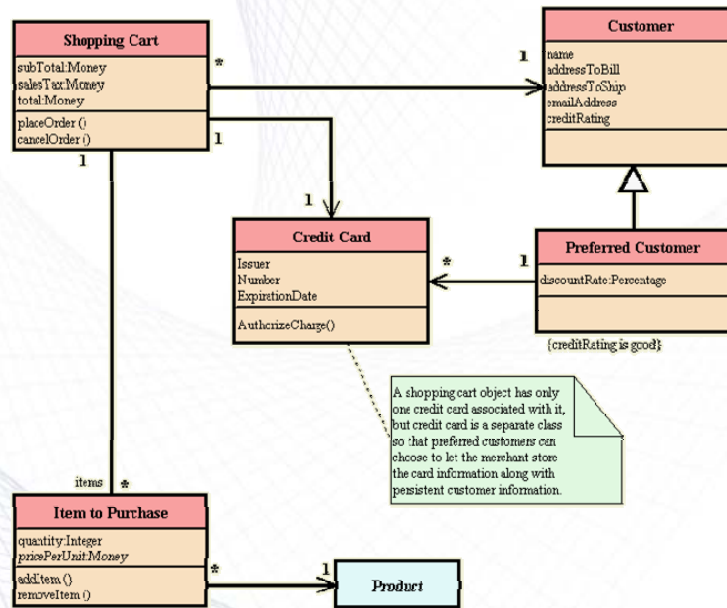
Basic Metrics for OO Systems

- Proposed originally by [Chidamber & Kemerer 1993](#), expanded by Soft Assurance Technology Center (SATC) at NASA. [[Rosenberg, et al.](#)]

| SOURCE | METRIC | O-O CONSTRUCT |
|-------------|----------------------------------|-----------------|
| Traditional | Cyclomatic Complexity (CC) | Operation |
| Traditional | Lines of Code (LOC) | Class/Operation |
| Traditional | Comment Percentage (CP) | Class/Operation |
| NEW | Weighted Methods per Class (WMC) | Class/Operation |
| NEW | Response For a Class (RFC) | Class/Operation |
| NEW | Lack of COhesion (LCOM) | Class/Operation |
| NEW | Coupling Between Objects (CBO) | Coupling |
| NEW | Depth of Inheritance Tree (DIT) | Inheritance |
| NEW | Number of Children (NoC) | Inheritance |



Working Example: Shopping Cart



Weighted Methods per Class (WMC) /1



- WMC is a metric of size and complexity. It is defined as: $WMC = \sum c_i$
- where c_i is the complexity of each different method c_1, c_2, \dots, c_n in class C .
- If each method were assigned a complexity of 1 then the WMC for class C would equal the number of methods in the class.
- Chidamber and Kemerer deliberately did not define “complexity” more specifically in order to permit the most general application of the metric possible.



Weighted Methods per Class



(WMC) /2

- The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class.
- The larger the number of methods in a class, the greater the potential impact on children; children inherit all of the methods defined in the parent class.
- Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.
- **Example:** WMC is calculated by counting the number of methods in each class, therefore:

WMC for *Shopping_Cart* = 2

WMC for *Credit_Card* = 1



Depth of Inheritance Tree (DIT)



- The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes.
- The deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit making it more complex to predict its behavior.
- Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods.
- **Example:**

Customer is the root and has a DIT of 0. The DIT for *Preferred_Customer* is 1.

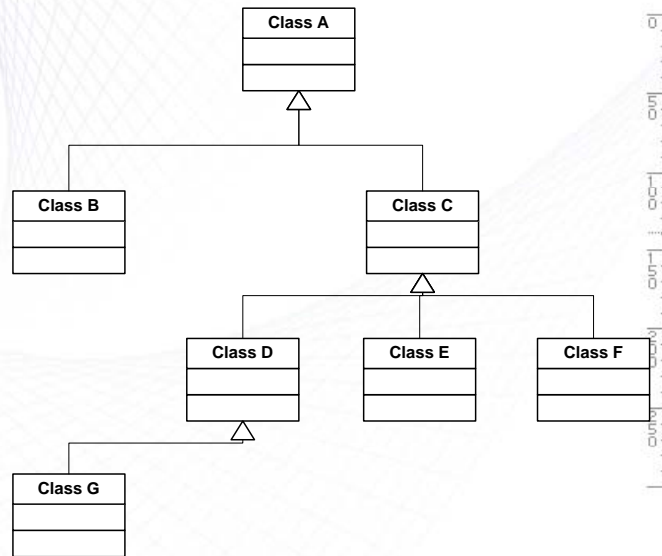




Depth of Inheritance Tree (DIT)

Another Example

- $DIT(A)=0$
- $DIT(B,C)=1$
- $DIT(D,E,F)=2$
- $DIT(G)=3$



Number of Children (NoC)

- The number of children is the number of immediate subclasses subordinate to a class in the hierarchy.
- It is an indicator of the potential influence a class can have on the design and on the system.
- The greater the NoC, gives an idea of the potential influence a parent class has on design. If a class has a large number of sub-classes, it could require more testing of its methods.
- The greater the NoC, the greater the reuse since inheritance is a form of reuse.

- **Example:**

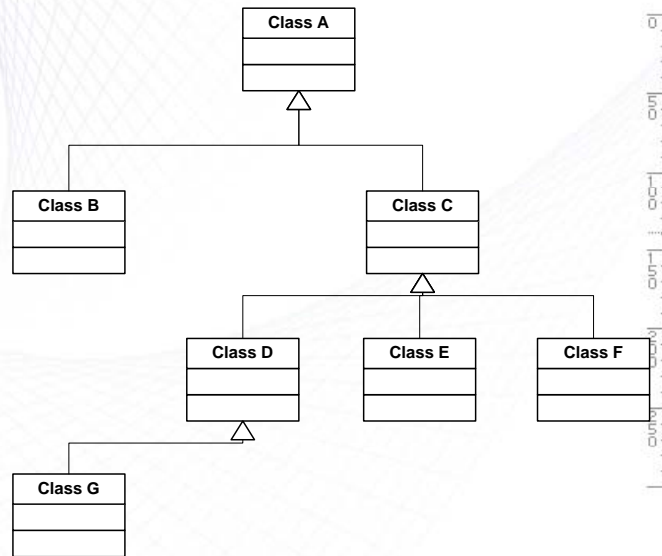
Customer has an NOC of 2. NOC for *Preferred_Customer* is 0 since it is a terminating or leaf node in the tree structure.



Number of Children (NoC)

Another Example

- $NoC(A)=2$
- $NoC(C)=3$
- $NoC(D)=1$
- $NoC(B,E,F,G)=0$



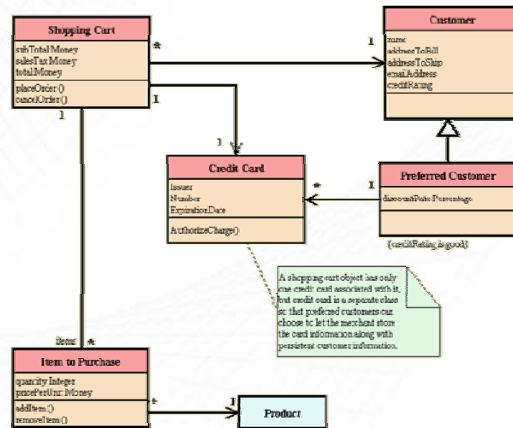
Coupling Between Objects (CBO)

- CBO metric therefore is a measure of non inherited interactions between classes.
- CBO is a count of the number of other classes to which a class is coupled.
- It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends.
- Excessive coupling is detrimental to modular design and prevents reuse.
- The more independent a class is, the easier it is reuse in another application.



Example: CBO

- Two classes may have excessive coupling (too many messages passing between them). This implies that those classes should be combined into one class.



Response for a Class (RFC)

- RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class.
- This metric looks at combination of the complexity of a class through the number of methods and the amount of communication with other classes.
- The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class.
- Example:**

$$\text{RFC for Preferred_Customer} = 0 \text{ (self)} + 0 \text{ (Customer)} + 1 \text{ (Credit_Card)} = 1$$



Lack of Cohesion (LCOM)

- Lack of Cohesion (LCOM) measures the dissimilarity of methods in a class by instance variable or attributes.
- If a class has different methods performing different operations on the same set of instance variables, the class has cohesion.
- A highly cohesive module should stand alone; high cohesion indicates good class subdivision.
- Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.
- High cohesion implies simplicity and high reusability.



Example: LCOM

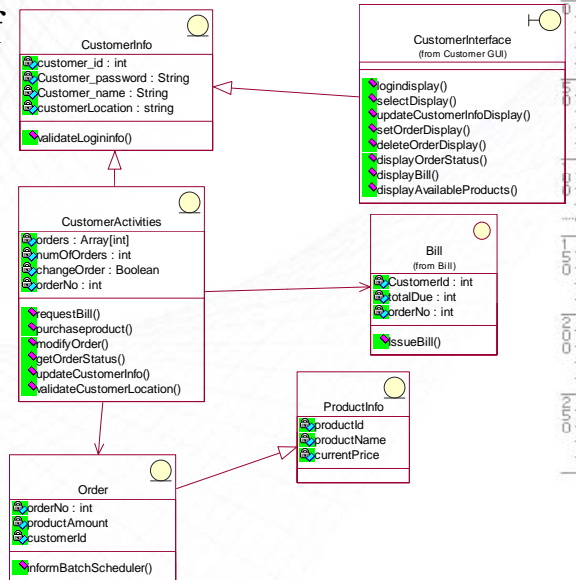
- Alternative design with high LCOM: Assume that the *Credit_Card* and *Preferred_Customer* classes are merged. There will be relatively few common attributes and methods among the objects that may belong to this class.



Example

- Determine the value of
 - (1) “Average method per class”;
 - (2) “Response for a Class (RFC)” for CustomerActivities and CustomerInterface classes.

Average method per class = 17/6 = 2.83
For CustomerActivities RFC = 9
For CustomerInterface RFC = 9

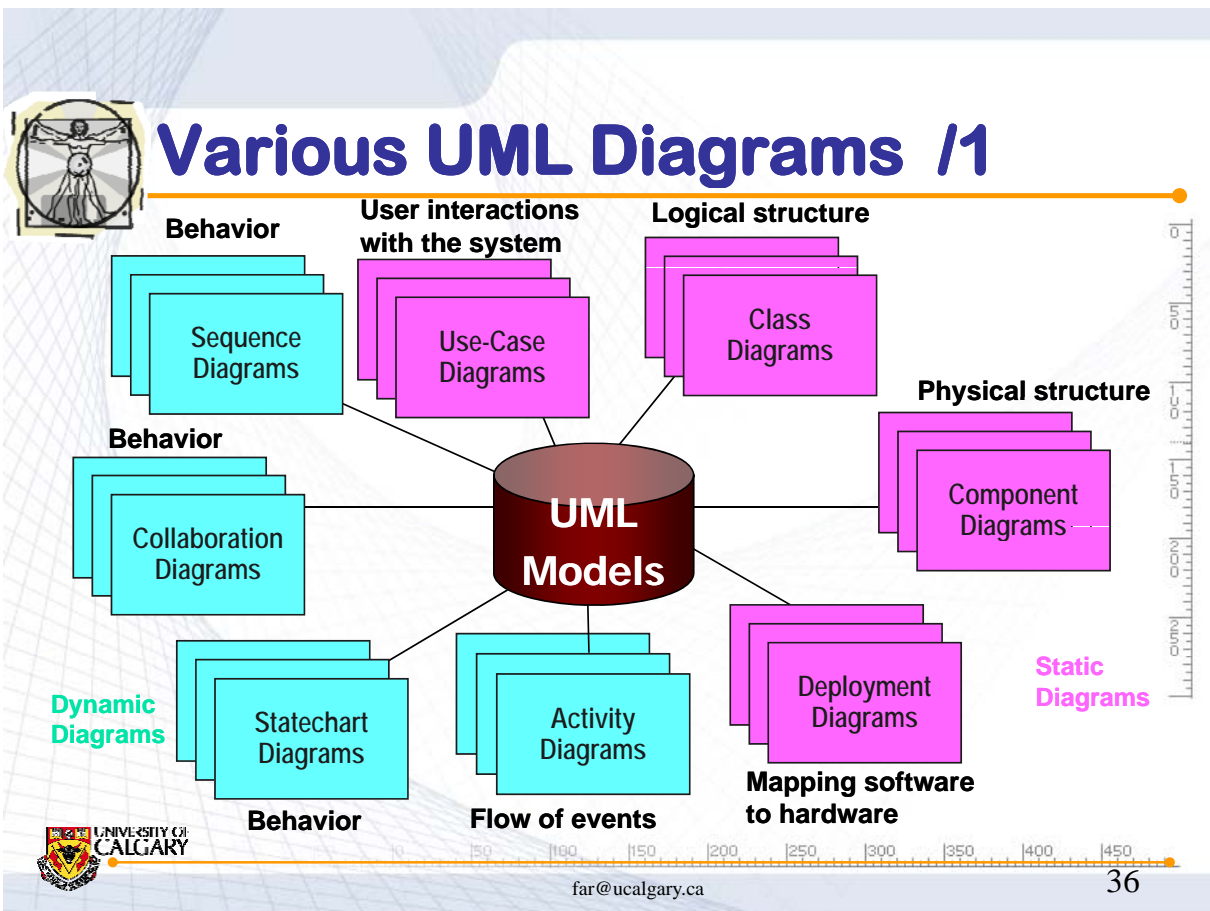


Interpretation

| METRICS | OBJECTIVE |
|-------------------------------------|-----------------|
| Cyclomatic Complexity | Low |
| Lines of Code/Executable Statements | Low |
| Comment Percentage | ~ 20 – 30 % |
| Weighted Methods per Class (WMC) | Low |
| Response for a Class (RFC) | Low |
| Lack of Cohesion of Methods (LCOM) | Low |
| Cohesion of Methods | High |
| Coupling Between Objects (CBO) | Low |
| Depth of Inheritance (DIT) | Low (trade-off) |
| Number of Children (NoC) | Low (trade-off) |

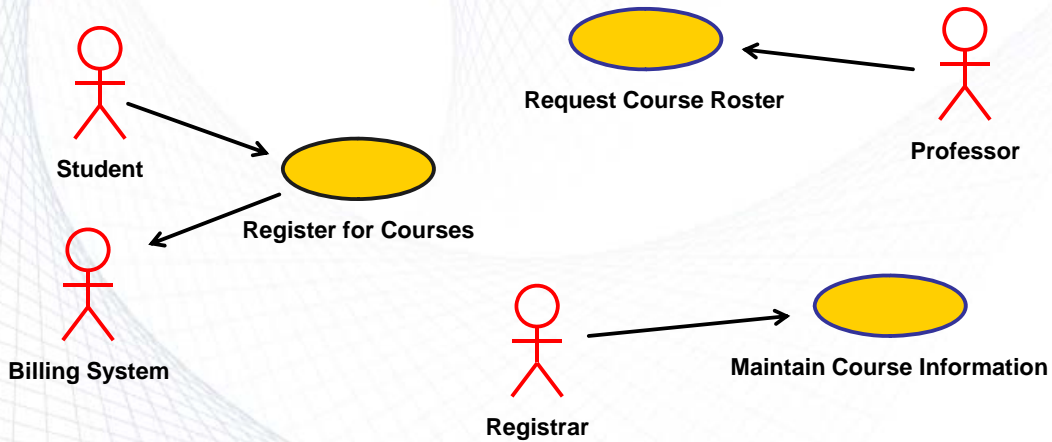


OO Metrics: Metrics Using UML



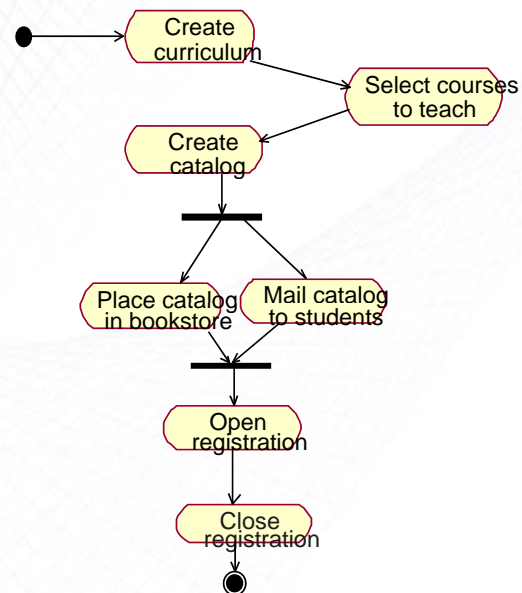


Example: UML Use-Case Diagram



Example: UML Activity Diagram

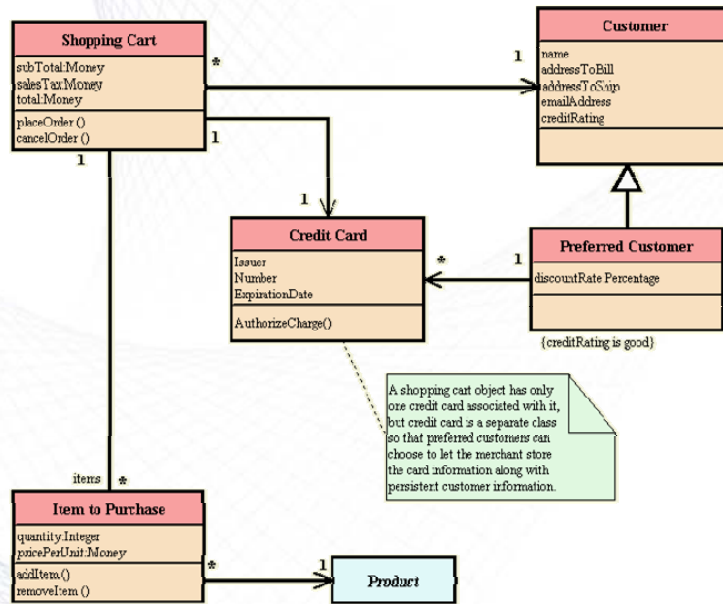
- An activity diagram shows the flow of events within the use-case.





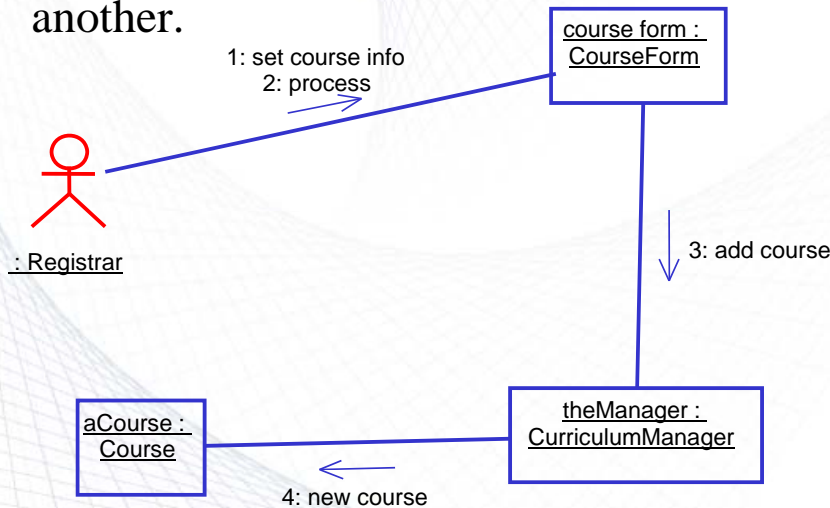
Example: UML Class Diagram

- A class diagram shows the structure of the software.



Example: UML Collaboration Diagram

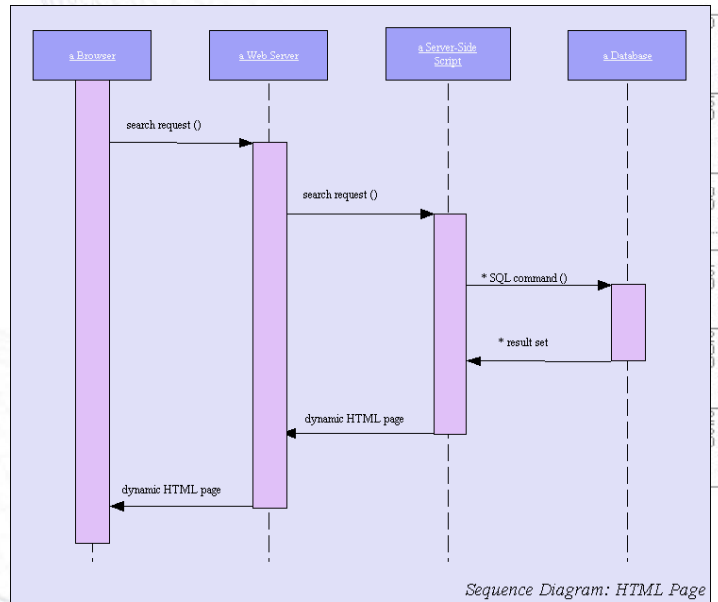
- A collaboration diagram displays object interactions organized around objects and their links to one another.





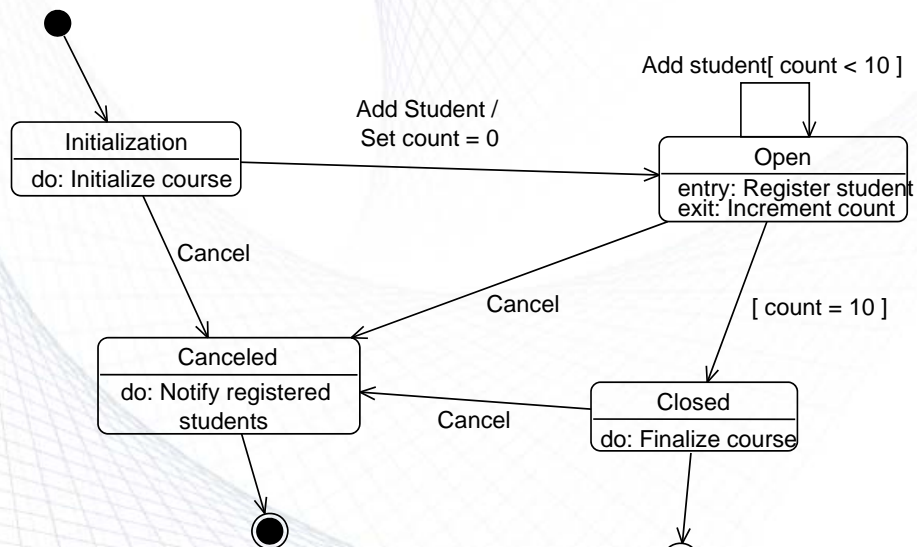
Example: UML Sequence Diagram

- A sequence diagram shows step by step what must happen to accomplish a piece of functionality provided by the system.



Example: UML Statechart Diagram

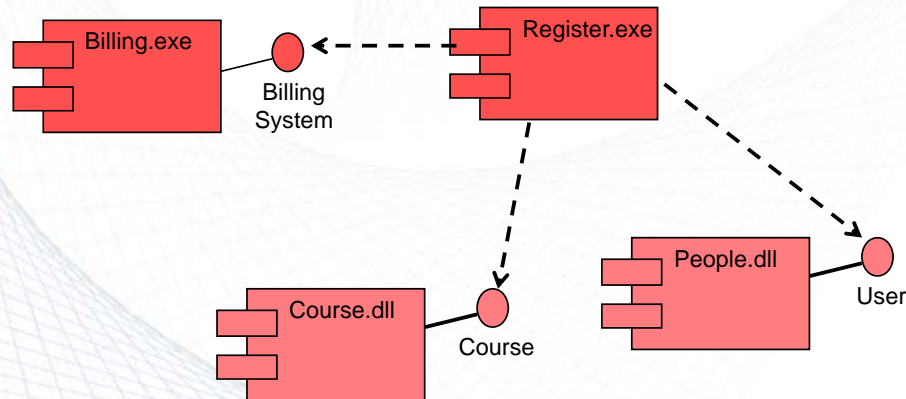
- A statechart diagram shows the lifecycle of a single class.



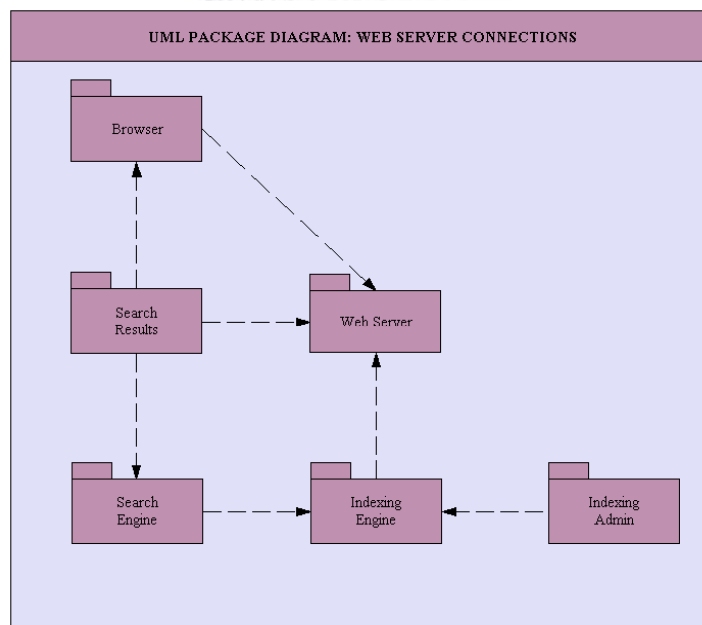


Example: UML Component Diagram

- Component diagrams illustrate the organization and dependencies among software components.



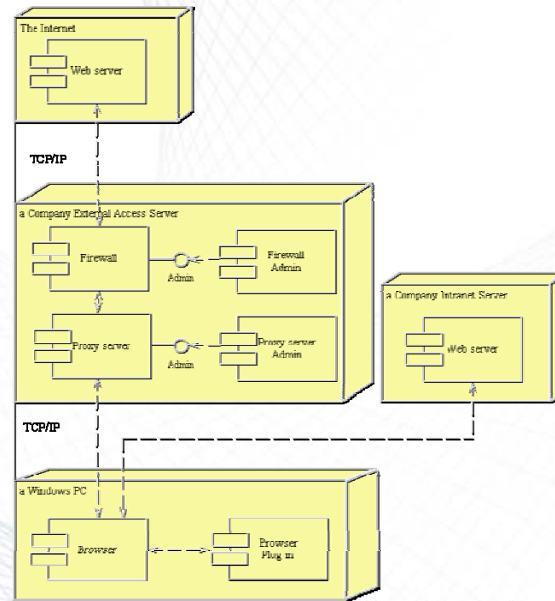
Example: UML Package Diagram



Example: UML Deployment Diagram



UML Deployment: TCP/IP Layout



Number of Domain Classes (#DC)



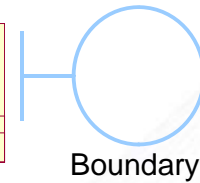
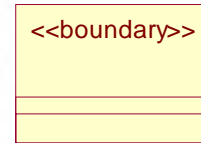
- **Name and origin:** #DC, first discussed in (Lorenz & Kidd, 1994).
- **Measurement:** #DC can be easily measured, even without tool support.
- **Usage:** #DC can be used for tracking the overall software process. It can only reflect major changes to the system (creation, deletion, or replacement of domain classes).
This metric ignores the associations or relationships between the classes.





Domain Classes (#DC)

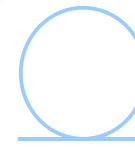
- The analysis classes can be further broken down to class stereotypes.
- Stereotyped class is still a class, but has additional information that is defined by the stereotype.



Boundary



Control



Entity



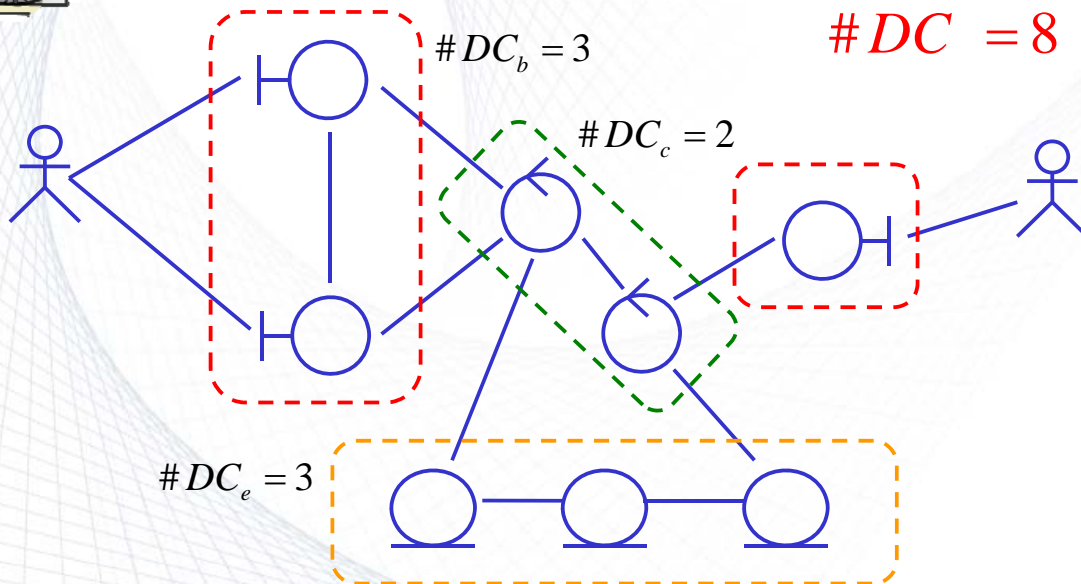
Domain Classes (#DC)

- **Boundary class** models the interaction between the system's surroundings and its inner workings.
 - #DC_b is the number of boundary classes.
- **Control class** controls the behavior of a use case and delegates the work of the use case to other classes
 - #DC_c is the number of control classes.
- **Entity class** models the key concepts of the system (usually models persistent information)
 - #DC_e is the number of entity classes.





Example: #DC



From Analysis to Design Classes

- Can we estimate the number of *design classes* from the *analysis classes*?
- An analysis class maps directly to a design class or
- More complex analysis classes may
 - Split into multiple classes
 - Become a package
 - Become a subsystem
 - Any combination of the above

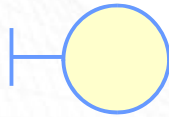
This depends
on
development
technology



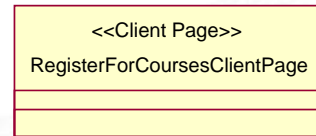
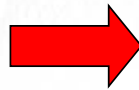


Expansion: Boundary Class

- In a **Web application**, each boundary class representing a form becomes a JSP, servlet or client page, or a combination of the three.
 - Example: RegisterForCoursesForm as a client page



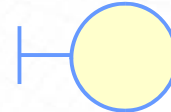
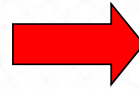
RegisterForCoursesForm



- In a **GUI environment**, each boundary class continues to represent a form.



RegisterForCoursesForm



RegisterForCoursesForm

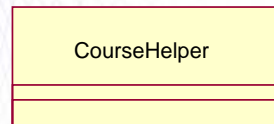


Expansion: Boundary Class

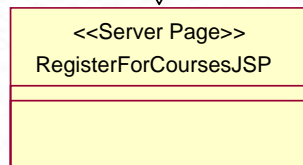
- Web Example: Boundary classes as server pages**



RegisterForCoursesForm



Helper Object:
Provides logic to manipulate and return data from an EJB



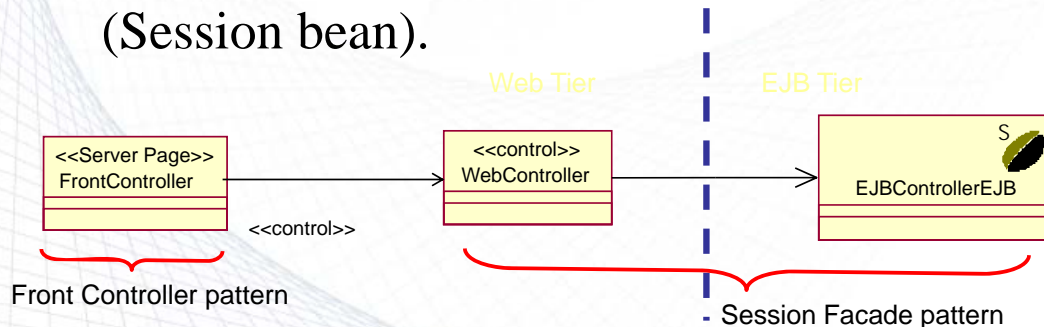
JSP: Contains presentation logic and content





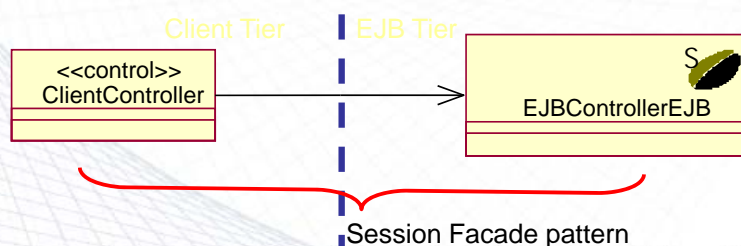
Expansion: Control Classes

- In a **Web application**, control classes become **use-case dispatchers**, which are composed of three elements: a front controller (Servlet or JSP), Web controller, and EJB controller (Session bean).



Expansion: Control Classes

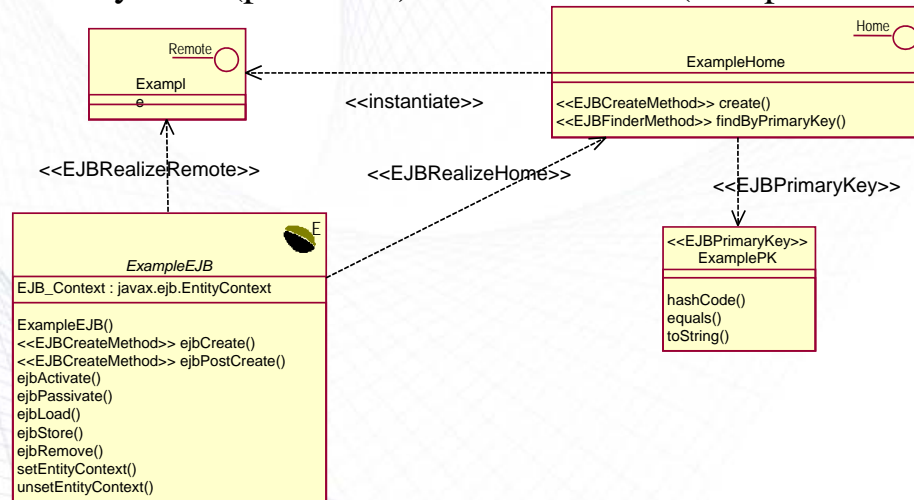
- In a **GUI environment**, control classes become use-case dispatchers which are composed of two elements: a client controller and EJB controller (Session bean).
- The **client controller** acts as a proxy for the EJB controller and is responsible for forwarding events from the boundary class to the EJB controller.
- An **EJB controller** accepts the events and makes calls on the enterprise beans affected by the event.





Expansion: Entity Classes

- Each entity class becomes an enterprise bean:
 - Entity bean (persistent) or session bean (non-persistent)



Expansion Rate

- Typical expansion rate for converting analysis classes to design classes, using Java technology:
 - Boundary Class \times 1 or 2
 - Control Class \times 2 or 3
 - Entity Class \times 4





Example: #Design Classes

- The total estimated number of design classes are:

$$\#DC_b = 3 \times 2$$

$$\#DC_c = 2 \times 3$$

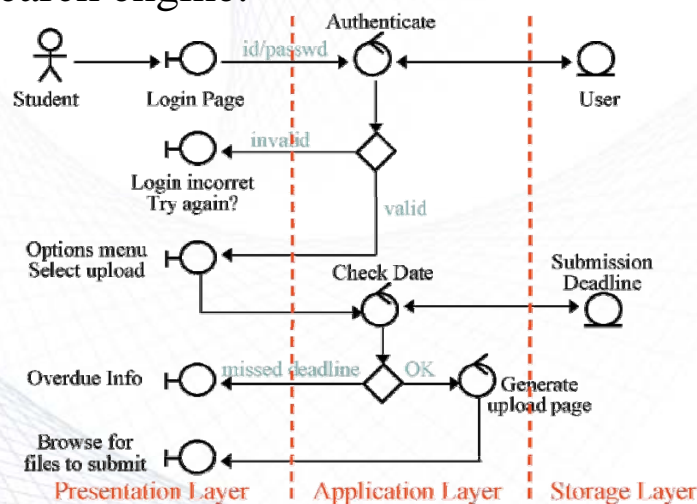
$$\#DC_e = 3 \times 4$$

$$Total = 24$$



Example

- The following figure depicts analysis class model for a search engine.





Example (cont'd)

- The system will be implemented as a Web application using EJB technology. In the design phase, boundary classes are expanded by the ratio of 1 to 2; control classes are expanded by the ratio of 2 to 3, and each entity class is expanded to a subsystem having 4 to 8 design classes. A design class in Java on average has 20 methods, each method has 10 to 20 lines of code, and the productivity of an average Java programmer is 200 LOC per week for a salary of 5,000\$ per month (4 weeks).
- Estimate the minimum and maximum for the
 - number of design classes; number of methods; total software size in LOC (assuming total size is 1.5 times methods LOC) and
 - total development cost.



Example (cont'd)

| <i>Analysis Classes</i> | <i>No.</i> | <i>Total classes</i> | <i>Total methods</i> | <i>Total LOC</i> | <i>Development cost</i> |
|--|------------|--------------------------------|----------------------------------|--|---|
| Boundary Class | 5 | Min 19 Max 35 | Min 380 Max 700 | $1.5 \times 380 \times 10 = 5,700$ (minimum) | $(5,700/200) \times (\\$5,000/4) = 35,625\\$ (minimum) |
| min ($\times 1$) max ($\times 2$) | 5 10 | | | | |
| Control class | 3 | | | $1.5 \times 380 \times 20 = 11,400$ | $(21,100/200) \times (\\$5,000/4) = 131,875\\$ (maximum) |
| min ($\times 2$) max ($\times 3$) | 6 9 | | | | |
| Entity class | 2 | | | $1.5 \times 700 \times 10 = 10,500$ | $1.5 \times 700 \times 20 = 21,100$ (maximum) |
| min ($\times 4$) max ($\times 8$) | 8 16 | | | | |





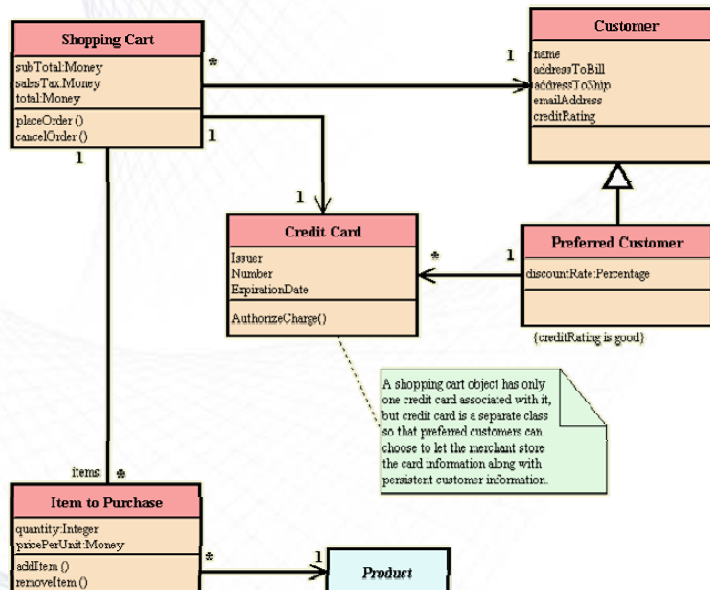
Number of Classes (#c)

- **Name and origin:** #c (Lorenz and Kidd, 1994).
- **Measurement:** #c is the number of classes that make up a system.
- **Usage:** The number of design classes can be used to track the process of design.
It ignores the association and aggregation among classes. Does not bear any structural meaning.



Example: #c

#c = 5





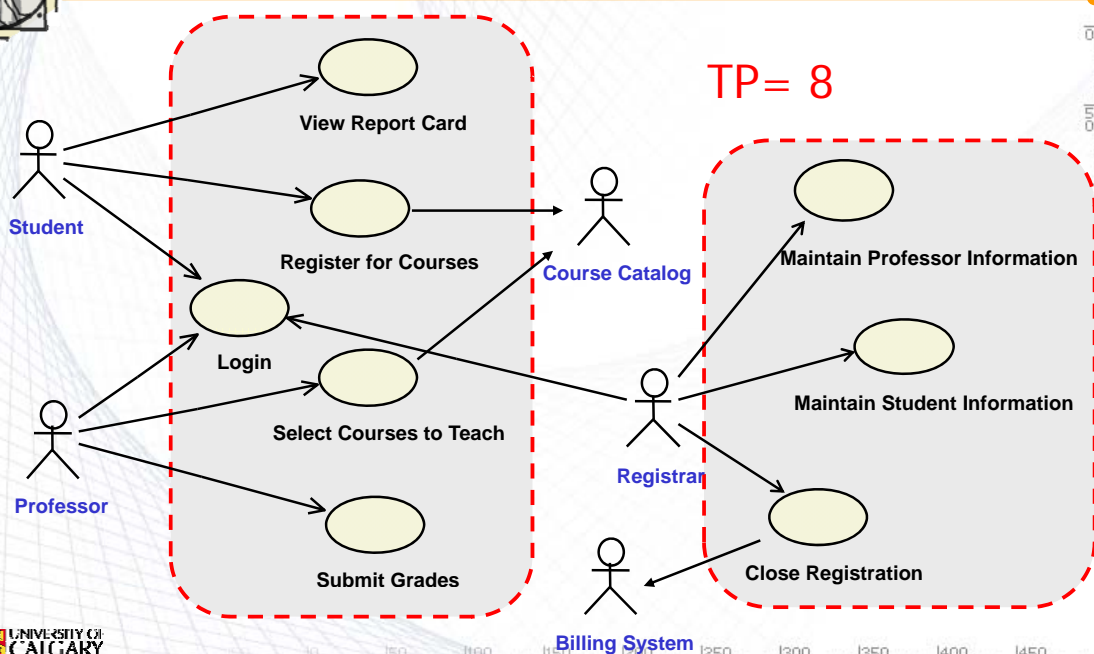
Number of Atomic Scenarios (#TP)

- **Name and origin:** Task Points (Graham, 1995).
- **Measurement:** Task Points are the simple count of the so-called “atomic” task scripts (a more formal form of use cases) in an object-oriented analysis model.
- **Usage:** Task Points are suggested for tracking and estimating the overall software process. No empirical data has been reported on the practical use of it.

Similar to #DC, it does not include any structural information about the system.



Example: #TP





Number of Instance Methods (#im)

- **Name and origin:** #im (Lorenz and Kidd, 1994).
- **Measurement:** #im is the number of methods that siblings of a class can understand (regardless of eventual method access restrictions such as “private” or “protected”).
- **Usage:** Upper threshold value of 20 for #im may be used to indicate alarm for individual classes. Use the thresholds only for non-interface and non-generated classes.
Interface classes, usually require higher thresholds (e.g., upper threshold of 40 for individual GUI-classes).
A system-wide summation of all #im values can be used as a measure of design size (and be used for tracking the technical design process).



Number of Class Methods (#cm)

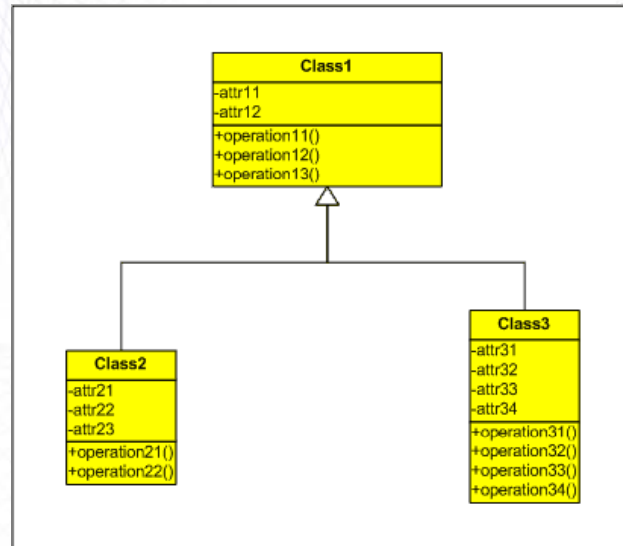
- **Name and origin:** #cm (Lorenz and Kidd, 1994).
- **Measurement:** #cm the number of methods that a class can understand (regardless of eventual method access restrictions such as “private” or “protected”).
- **Usage:** Upper threshold value of 3 for #cm may be used to indicate alarm for individual classes.
A system-wide summation of all #cm values can be used as a measure of design size (and be used for tracking the technical design process).





Example: #im and #cm

- For Class1
#cm= 3
- For Class3
#im= 7



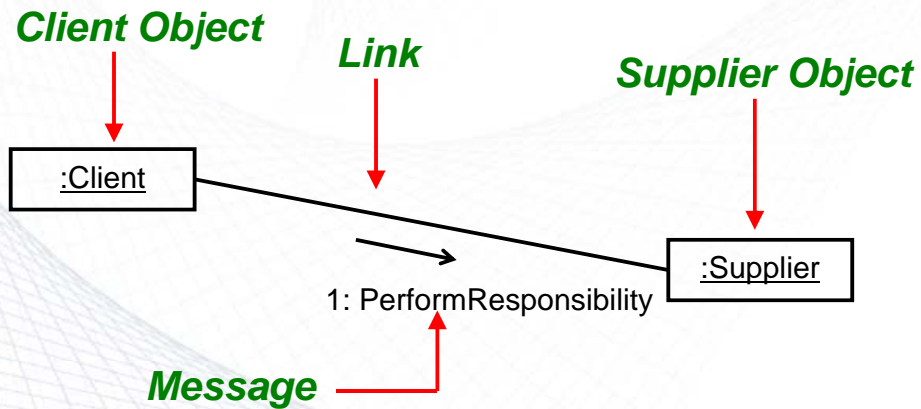
Coupling

- Coupling describes how strongly one element (class / package) relates to another element
- The goal is to achieve “loose coupling”
- **Coupling between objects** is a count of the number of “links” between them.
- **Coupling between classes** (CBO) is a count of the number of other classes to which a class is related. It is measured by counting the number of distinct “associations” with the other classes.
- **Coupling between Packages** is a count of the number of distinct “associations” between packages.

What are Links and Associations?



- An object is an instance of a class
- In the same way, a “link” is an instance of an “association”

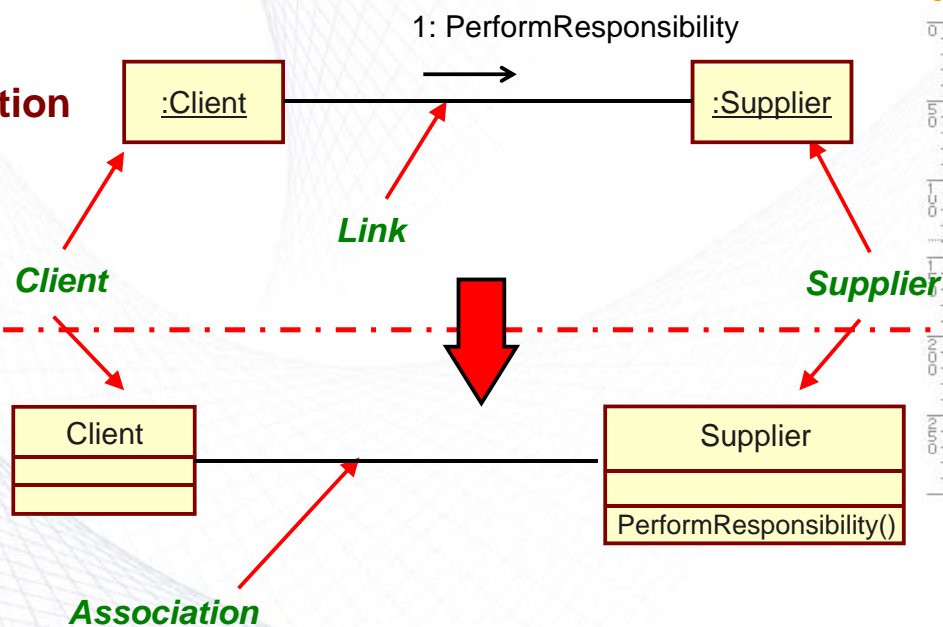


What are Links and Associations?



Collaboration Diagram

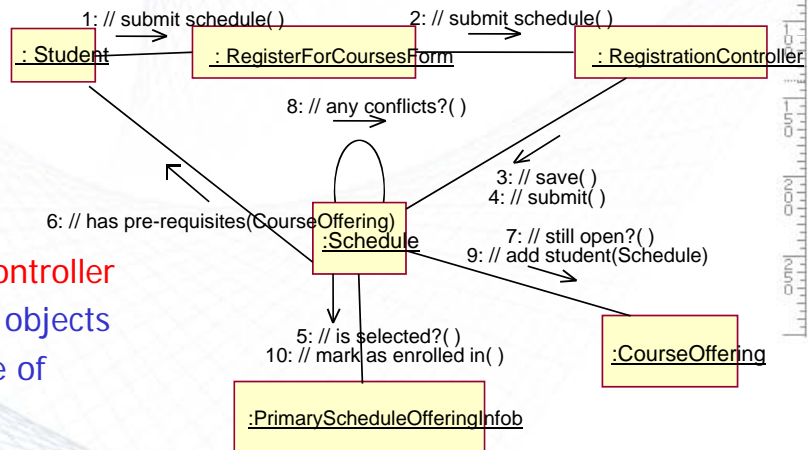
Class Diagram





Coupling: How to Measure?

- Both collaboration and class diagrams can be used to measure coupling.
- Example:

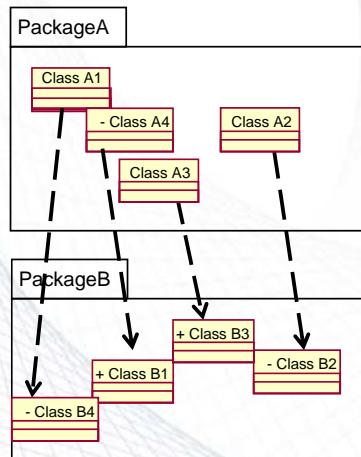


RegistrationController and Schedule objects have a degree of coupling of 2

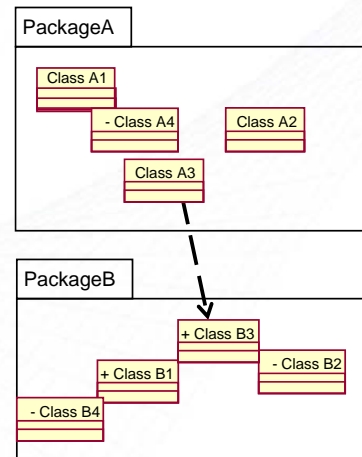


Package Coupling: Class Relationships

- Strive for the loosest coupling possible



Strong Coupling



Loose Coupling





Cohesion

- Cohesion describes how strongly related the responsibilities between design elements can be described
 - The goal is to achieve “high cohesion”
 - High cohesion between classes is when class responsibilities are highly related

A class should do one thing, and do that well.



Cohesion: How to Measure?

- Find “similarities” and “dissimilarities” between operations (methods) in a class.
- Remember a “link” will eventually be translated into an “operation.”
- What diagrams can be used?
 - Class diagrams





Example: Cohesion

| Account |
|--------------------|
| -Balance |
| -Name |
| -Number |
| +withdraw() |
| +createStatement() |

Good class cohesion

| Scholar |
|-------------------------|
| -Name |
| -Status |
| -Employee_ID |
| -Student_ID |
| -Hire_Date |
| -Max_Load |
| +getSchedule() |
| +createSchedule() |
| +addSchedule() |
| +submitGrade() |
| +setMaxLoad() |
| +takeSabbatical() |
| +acceptCourseOffering() |
| +withdraw() |
| +getTuition() |

Bad class cohesion



Example: Cohesion

| Scholar |
|-------------------------|
| -Name |
| -Status |
| -Employee_ID |
| -Student_ID |
| -Hire_Date |
| -Max_Load |
| +getSchedule() |
| +createSchedule() |
| +addSchedule() |
| +submitGrade() |
| +setMaxLoad() |
| +takeSabbatical() |
| +acceptCourseOffering() |
| +withdraw() |
| +getTuition() |

| Professor |
|-------------------------|
| -Name |
| -Status |
| -Employee_ID |
| -Hire_Date |
| -Max_Load |
| +getSchedule() |
| +createSchedule() |
| +addSchedule() |
| +submitGrade() |
| +setMaxLoad() |
| +takeSabbatical() |
| +acceptCourseOffering() |

| Student |
|-------------------|
| -Name |
| -Status |
| -Student_ID |
| +withdraw() |
| +getSchedule() |
| +createSchedule() |
| +addSchedule() |
| +getTuition() |

- Cohesion can help split and merge the classes



OO Metrics: Other Metrics



Size: Lines of Code (LOC)

- **Project LOC** = Import's + Classes LOC
- **Class LOC** = Class Declaration + Variable Declarations + Methods LOC + Inner (sub) Classes LOC
- **Method LOC** = Method Declaration + Local Variable Declarations + Statements



Average Methods Per Class

- The number of methods and the complexity of methods involved is an indicator of how much time and effort is required to develop and maintain a class.
- The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class.
- Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

$$\text{Average no. of Methods per Class} = \frac{\text{Total no. of methods}}{\text{Total no. of classes}}$$

[Chidamber & Kemerer, 1994]



Depth of Inheritance Tree

- The deeper a class in the hierarchy, the greater the number of methods it is likely to inherit, hence, more complex behaviour.
- Deeper trees constitute greater design complexity, since more methods and classes are involved.
- The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

$$\text{Inheritance Tree Depth} = \text{Max}\{\text{Inheritance Tree Path Length}\}$$

[Chidamber & Kemerer, 1994]





Number of Used Methods

- If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.
- The larger the number of methods that can be invoked from a class, the greater the complexity of the class.

[Chidamber & Kemerer, 1994]



Object Library Effectiveness

- If objects are actually being designed to be reusable beyond a single application, then the effects should appear in object library usage statistics.

Average Number of Times
a Library Object is Reused

$$= \frac{\text{Total Number of Object Reused}}{\text{Total Number of Library Objects}}$$





Degree of Reuse of Inheritance Methods

- Simply defining methods in such a way that they can be reused via inheritance does not guarantee that those methods are actually reused.
- Percent of Potential Method Uses Actually Reused (PP):

$$PP = \frac{\text{Total no. of Actual Method Uses}}{\text{Total no. of Potential Method Uses}}$$

- Percent of Potential Method Uses Overridden (PM):

$$PM = \frac{\text{Total Number of Methods Overridden}}{\text{Total no. of Potential Method Uses}}$$



Average Method Complexity

- More complex methods are likely to be more difficult to maintain.
- Greater method complexity is likely to lead to a lower degree of overall application comprehensibility.
- Greater method complexity is likely to adversely affect application reliability.
- More complex methods are likely to be more difficult to test.

$$\text{Average Method Complexity} = \frac{\sum V(G)}{\text{Total no. of application Methods}}$$

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





Application Granularity

- An application constructed with more finely granular objects (i.e. a lower number of functions per object) is likely to be more easily maintained because objects should be smaller and less complex.
- More finely granular objects should also be more reusable.

$$\text{Application Granularity} = \frac{\text{Total Number of Objects}}{\text{Total Function Points}}$$



MOOD: Metrics for Object Oriented Design Toolkit

- The MOOD metrics set refers to a basic structural mechanism for the OO measurement.
- The set includes:
 - *Encapsulation* (MHF and AHF)
 - *Inheritance* (MIF and AIF)
 - *Polymorphism* (PF)
 - *Message-passing* (CF)





Method Hiding Factor (MHF)

- MHF is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration.
- The invisibility of a method is the percentage of the total classes from which this method is not visible.

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)} \quad V(M_{mi}) = \frac{\sum_{j=1}^{TC} is_visible(M_{mi}, C_j)}{TC - 1}$$

$$is_visible(M_{mi}, C_j) = \begin{cases} 1 & \text{iff } j \neq i \wedge C_j \text{ may call } M_{mi} \\ 0 & \text{otherwise} \end{cases}$$

- $M_d(C_i)$ is the number of methods declared in a class.
- TC is the total number of classes.

rather nonsense!
does not distinguish between operations visible to a package, to a subsystem, to a component and to a class



Attribute Hiding Factor (AHF)

- AHF is defined as the ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes defined in the system.
- $A_d(C_i)$ is the number of attributes declared in a class.
- TC is the total number of classes.

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{a=1}^{A_d(C_i)} (1 - V(A_{ai}))}{\sum_{i=1}^{TC} A_d(C_i)} \quad V(A_{ai}) = \frac{\sum_{j=1}^{TC} is_visible(A_{ai}, C_j)}{TC - 1}$$

$$is_visible(A_{ai}, C_j) = \begin{cases} 1 & \text{iff } j \neq i \wedge C_j \text{ may access } A_{ai} \\ 0 & \text{otherwise} \end{cases}$$

rather nonsense!
does not distinguish between private and protected attributes.





Method Inheritance Factor (MIF)

- MIF is defined as the ratio of the sum of the inherited methods in all classes of the system under consideration to the total number of available methods (locally defined plus inherited) for all classes.

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} \quad M_a(C_i) = M_d(C_i) + M_i(C_i)$$

$M_d(C_i)$: number of methods declared in a class

$M_a(C_i)$: number of methods invoked in association with C_i

$M_i(C_i)$: number of methods inherited (not overridden) in C_i



Attribute Inheritance Factor (AIF)

- AIF is defined as the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes (locally defined plus inherited) for all classes.

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)} \quad A_a(C_i) = A_d(C_i) + A_i(C_i)$$

$A_d(C_i)$: number of attributes declared in a class

$A_a(C_i)$: number of attributes accessed in association with C_i

$A_i(C_i)$: number of attributes inherited in C_i





Polymorphism Factor (PF)

- PF is defined as the ratio of the actual number of possible different polymorphic situation for class C_i to the maximum number of possible distinct polymorphic situations for class C_i .

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]} \quad M_d(C_i) = M_n(C_i) + M_o(C_i)$$

$M_n(C_i)$: number of new methods declared in a class

$M_o(C_i)$: number of overriding methods

$DC(C_i)$: number of classes descending from C_i



Coupling Factor (CF)

- CF is defined as the ratio of the maximum possible number of couplings in the system to the actual number of couplings not imputable to inheritance.

$$CF = \frac{\sum_{i=1}^{TC} \left[\sum_{j=1}^{TC} is_client(C_i, C_j) \right]}{TC^2 - TC}$$


$$is_client(C_i, C_j) = \begin{cases} 1 & \text{iff } C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 & \text{otherwise} \end{cases}$$

$C_c \Rightarrow C_s$ is relationship between client class (C) and supplier class (S)






OO Metrics: Quality and Risk Metrics for OO Software



High Risk Components Identification

- Empirical studies show that most faults are found in only a few of a system's components. If these few components can be identified early, then actions such as focusing defect detection activities on high-risk components can be taken.
- How to identify high-risk components?
 1. Devise quality models
 2. Define practical thresholds



far@ucalgary.ca 94



Object-Oriented Quality Models

- A quality model is usually developed using a statistical modeling or machine learning technique, or a combination of the two techniques, using historical data.



OO Quality Models: Example

Probability of a Class Having a Fault

$$p = \frac{1}{1 + e^{-(3.97 + 0.464NAI + 1.47OCMEC + 1.06DIT)}}$$

- p is the probability of a class having a fault
- NAI is the total number of attributes defined in the class
- $OCMEC$ is the number of other classes that have methods with parameter types of this class (i.e., a form of export coupling)
- DIT is the depth of the inheritance tree which measures how far down an inheritance hierarchy a class is.
- If the predicted probability of a fault is greater than 0.33, then the class should be considered as high risk!

[El-Emam, 2001]





2. Object-Oriented Thresholds

- What is threshold?
- The range of values on the software product metrics that delineate between *acceptable* and *unacceptable* values.
- The practical utility of object-oriented metrics would be enhanced if meaningful thresholds could be identified.
- **Example:** The average number of attributes (public or private variables) per class should be less than 6. More attributes indicate that the classes are doing more than they should.

Most of the threshold values are experience-based!



Experience-Based Size Threshold

- The average method size should be less than 8 LOC for Smalltalk and 24 LOC for C++. Bigger averages indicate OO design problems (i.e., function-oriented coding).
- The average number of methods per class should be less than 20. Bigger averages indicate too much responsibility in too few classes.
- The average number of attributes (public or private variables) per class should be less than 6. More attributes indicate that one class is doing more than it should.

[Lorenz & Kidd, 1994]





Experience-Based Inheritance Threshold

- The class hierarchy nesting level should be less than 6. Start counting at the level of any framework classes that you use or the root class if you don't.

[Lorenz & Kidd, 1994]



Experience-Based Coupling Threshold

- The number of package-to-package relationships should be less than the average number of class-to-class relationships within a package.
- The number of class-to-class relationships within a package should be relatively high.

[Lorenz & Kidd, 1994]

- Coupling between objects (CBO) is a useful indicator of fault-prone classes.

[El Emam, et al. 1999]





Experience-Based Effort Threshold

1. A prototype class has 10 to 15 methods, each with 5 to 10 lines of code (in Java), and takes 1 person-week to develop.
2. A production class has 20 to absolute maximum of 30 methods, each with 10 to 20 lines of code, and takes 8 person-weeks to develop. In both these cases, development includes documentation and testing.
3. The number of classes and methods thrown away should occur at a steady rate throughout most of the development process.



Tools for OO Measurement

- JMetric: A tool that calculates a set of object-oriented metrics from Java code.
- Requires Java 1.2 or better.
- Free Download.

[Andrew Cain & Rajesh Vasa]

URL:

<http://www.it.swin.edu.au/projects/jmetric/>





References

- S. Chidamber and C. Kemerer: “A Metrics Suite for Object-Oriented Design,” *IEEE Transactions on Software Engineering*, 20(6):476-493, 1994 (highly recommended).
- M. Lorenz and J. Kidd: “*Object-Oriented Software Metrics*,” Prentice-Hall, 1994.
- F. Brito Abreu and R. Carapuca: “Object-Oriented Software Engineering: Measuring and Controlling the Development Process (MOOD),” Proceedings of the 4th Int. Conf. on Software Quality, ASQC, 1994.
- K. El Emam: “*Object-Oriented Metrics: A Review of Theory and Practice*,” NRC-CNRC, ERB-1085.



UNIVERSITY OF
CALGARY

far@ucalgary.ca

103



UNIVERSITY OF
CALGARY

Summary

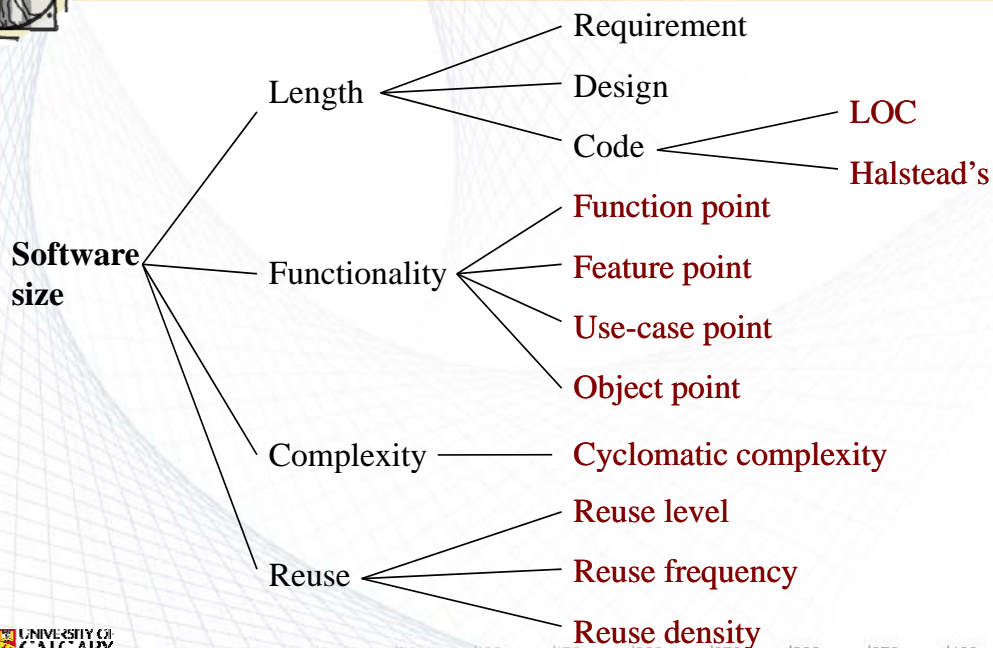


Measurement Theory

- Overview of software metrics
- The science of measurement
- Goal directed software measurement
- Empirical investigation in software engineering

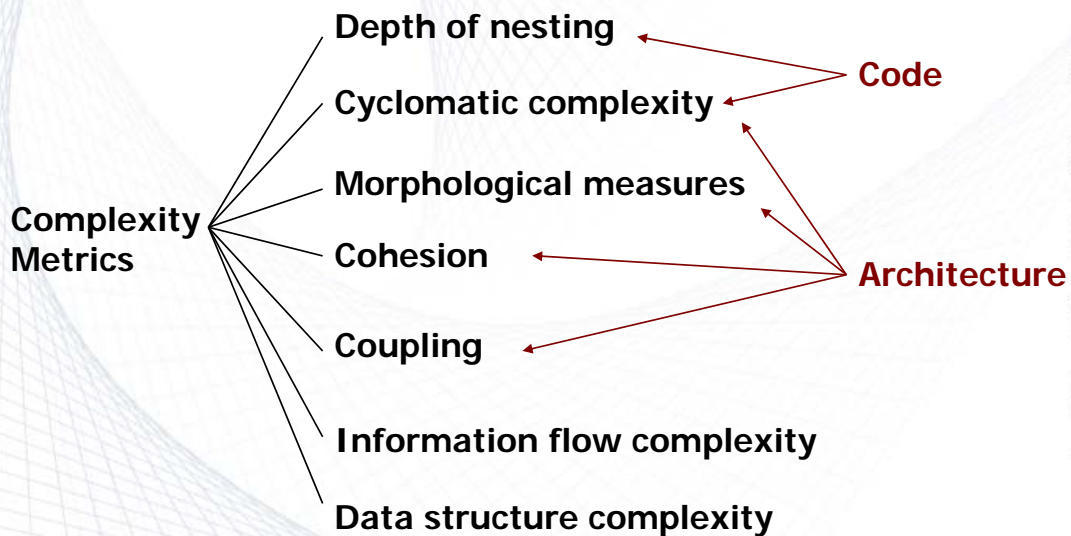


Software Size Metrics: Summary





Software Complexity Metrics



Software Cost

Software cost

Cost model

Effort vs. cost drive (size)

COCOMO

$$E = a(KLOC)^b \times EAF$$

$$T_{dev} = cE_{nom}^d$$

Constraint model

Effort & cost vs. time

SLIM

$$size = C \times B^{1/3} \times T^{4/3}$$

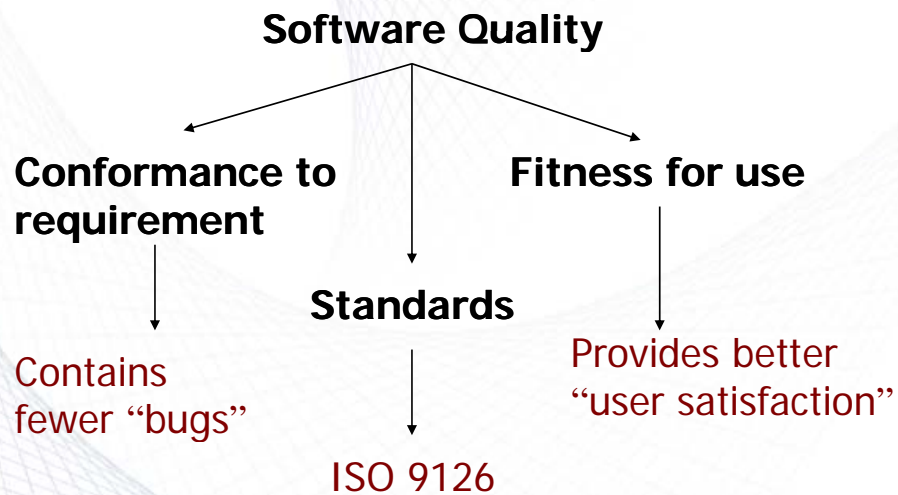
$$B = \left(\frac{1}{T^4}\right) \left(\frac{size}{C}\right)^3$$

$$E = 0.3945B$$





Software Quality



Software Quality: ISO 9126

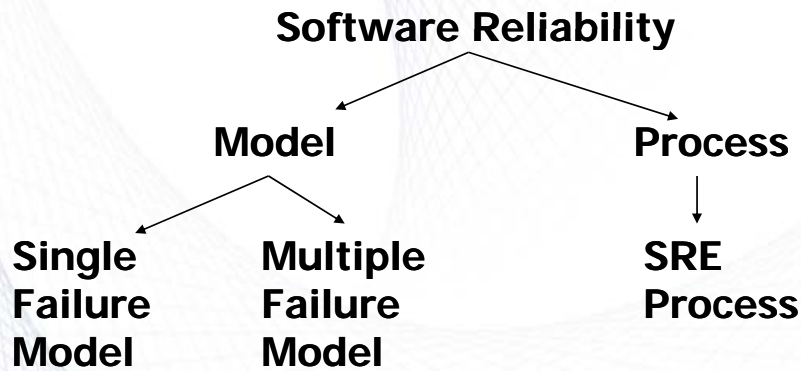
- Original ISO 9126 does not provide sub-characteristics and metrics, nor the method for measurement, rating and assessment.

| Characteristics | Attributes | | |
|------------------------|-------------------|--------------------|-----------------|
| Functionality | Suitability | Interoperability | Accuracy |
| | Compliance | Security | |
| Reliability | Maturity | Recoverability | Fault tolerance |
| | Crash frequency | | |
| Usability | Understandability | Learnability | Operability |
| Efficiency | Time behaviour | Resource behaviour | |
| Maintainability | Analyzability | Stability | Changeability |
| | Testability | | |
| Portability | Adaptability | Installability | Conformance |
| | Replacability | | |

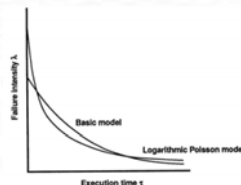
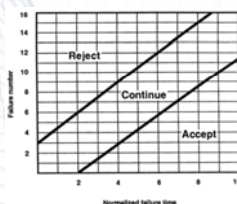
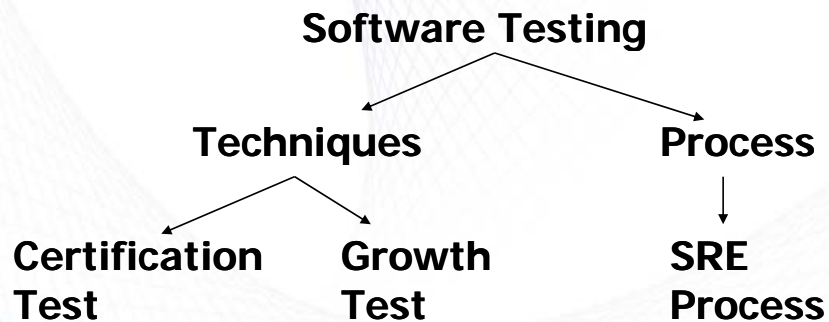




Software Reliability



Software Testing





OO Metrics

| SOURCE | METRIC | O-O CONSTRUCT |
|-------------|----------------------------------|-----------------|
| Traditional | Cyclomatic Complexity (CC) | Operation |
| Traditional | Lines of Code (LOC) | Class/Operation |
| Traditional | Comment Percentage (CP) | Class/Operation |
| NEW | Weighted Methods per Class (WMC) | Class/Operation |
| NEW | Response For a Class (RFC) | Class/Operation |
| NEW | Lack of COhesion (LCOM) | Class/Operation |
| NEW | Coupling Between Objects (CBO) | Coupling |
| NEW | Depth of Inheritance Tree (DIT) | Inheritance |
| NEW | Number of Children (NoC) | Inheritance |



Now You Should Know...

- What is software measurement about?
- Why software measurement is important?
- What does empirical investigation mean in the SE context?
- What are common software metrics?
- What attributes of the software you suggest to be measured?
- What is software measurement process?
- How to implement software measurement?
- What are challenges and difficulties of applying software metrics?





That Is All!

Good Luck!



Remember:

**Not everything
that counts can
be counted ...**

And

**Not everything
can be counted
counts ...**