Measuring Internal Product Attributes

(Structure)

Aspects of Structural Measures

- Size is important as discussed in the previous lecture. However, we must investigate the characteristics of a product structure, and determine how they affect the outcomes.
- A Software module or design (structure) can be viewed from several perspectives. The perspectives depend on
  - The level of abstraction—program unit (function, method, class), package, subsystem, and system
  - The way the module or design is described—syntax and semantics
  - The specific attribute to be measured

Structural Perspective

- We can think of structure from at least two perspectives:
- 1. ***Control flow structure:*** Addresses the sequence in which instructions are executed. This aspect of the structure reflects the iterative and looping nature of programs in a program
- 2. ***Data flow structure:*** Follows the trail of a data item as it is created or handled by a program. Data flow measures depict the behavior of the data as it interacts with the program.

Structure Measure

- We evaluated ***size*** measure in terms of three properties
  - Nonnegativity
  - Null values
  - Additivity
- To evaluate the ***structure*** measure we have to consider the following properties
  - Complexity
  - Length
  - Coupling
  - Cohesion

Structural Complexity Properties

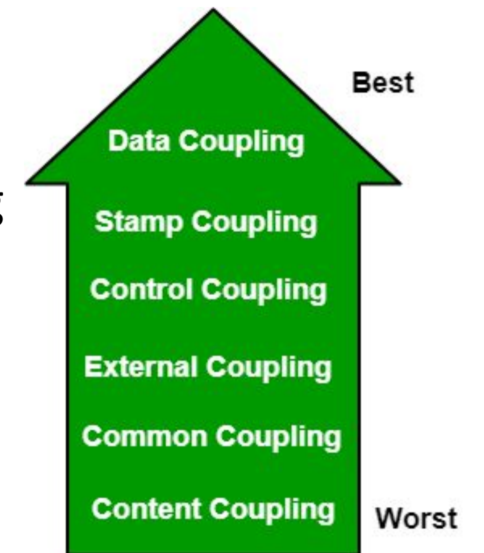
- Complexity refers to the complexity of a system
- The complexity of a system depends on the number of links between elements, and should, at a minimum, satisfy the following properties
  - *Nonnegativity:* complexity cannot be negative
  - *Null values: the* complexity of a system with no links is zero
  - *Symmetry:* The complexity of a system does not depend on how links are represented
  - *Module monotonicity:* System complexity "is no less than the sum of the complexities of any two of its modules with no relationships in common"
  - *Disjoint module additivity:* The complexity of a system of disjoint modules is the sum of the complexities of the modules.

Length Properties

- For software entities, we might be interested in the distance in terms of links from one element to another
  - *Nonnegativity*: System length cannot be negative.
  - *Null value*: A system with no links has zero length.
  - *Nonincreasing monotonicity for connected components*: Length does not increase when adding links between connected elements.
  - *Nondecreasing monotonicity for nonconnected components*: Length does not decrease when adding links between nonconnected elements.
  - *Disjoint modules*: The length of a system of disjoint modules is equal to the length of the module with the greatest length.

Coupling

- Coupling is the measure of the degree of interdependence **between** the modules. Good software will have low coupling.
  - **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent of each other and communicate through data. Example-customer billing system
  - **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module *but uses only a part of it*. E.g., passing structure variable in C or object in C++ language to a module
  - **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. e.g., a control flag, a comparison function passed to a sort algorithm.
  - **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
  - **Content Coupling:** Content coupling, or pathological coupling, occurs when one module modifies or relies on the **internal workings** of another module. This is the worst form of coupling and should be avoided. An example would be a search method that adds an object which is not found to the internal structure of the data structure used to hold information.



Coupling Properties

- 1. *Nonnegativity*: Module coupling cannot be negative.
- 2. *Null value*: A module without links to elements that are external to the module has zero coupling.
- 3. *Monotonicity*: Adding intermodule relationships does not decrease coupling.
- 4. *Merging modules*: Merging two modules creates a new module that has a coupling that is at most the sum of the coupling of the two modules.
- *5. Disjoint module additivity*: Merging disjoint modules without links between them creates a new module with a coupling that is the sum of the coupling of the original modules.

Cohesion

- Cohesion is the indication of the relationship **within** the module. It is the concept of intra-module. Cohesion has many types but usually, high cohesion is good for software.
- Increasing cohesion is good for software whereas increasing coupling is avoided for software.
- Highly cohesive gives the best software whereas loosely coupling gives the best software

Cohesion Properties

1. *Nonnegativity and normalization*: Module cohesion is normalized so that it is between zero and one.
2. *Null value*: A module whose elements have no links between them has zero cohesion.
3. *Monotonicity*: Adding links between elements in a module cannot **decrease** the cohesion of the module.
4. *Merging modules*: Merging two unrelated modules creates a new module with a maximum cohesion no greater than that of the original module with the greatest cohesion.

Object-Oriented structural attributes and measures

- The object management group (OMG) defined the Unified Modeling Language (UML), which includes a set of diagram types for modeling object-oriented systems at various levels of abstraction to describe the structure, behavior, and interactions of a system. The commonly used UML diagram types include the following:
    - *Class diagrams*: Model each class and its connections to other classes.
    - *Object diagrams*: Model a configuration of run-time objects.
    - *Activity diagrams*: Model the steps taken for a system to complete a task.
    - *State machine diagrams*: Model of finite-state machine representations.
    - *Use case diagrams*: Model external actors, the "use cases" that they take part in, and the dependencies between use cases.
    - *Sequence diagrams*: Model the sequences of messages passed between objects in order to complete a task.

Measuring Coupling in Object-Oriented Systems

- In addition to the coupling properties described earlier, several orthogonal coupling properties can help to evaluate coupling measures
  - Type: What kinds of entities are coupled?
  - Strength: How many connections of a particular kind?
  - Import or export: Are the connections import and/or export?
  - Indirect: Is indirect coupling measured?
  - Inheritance: Are connections to or from inherited entities counted?
  - Domain: Are the measures used to indicate the coupling of individual attributes, methods, classes, sets of classes (e.g., packages), or the system as a whole?

Coupling example

The *coupling between object classes* (*CBO*) is metric 4 in a commonly referenced suite of object-oriented metrics (Chidamber and Kemerer 1994). CBO is defined to be the number of other classes to which the class of interest is coupled. Briand et al. identified several problems with CBO (Briand et al. 1999b). One problem is that the treatment of inherited methods is ambiguous. Another problem is that CBO does not satisfy property 5, disjoint module additivity, one of the coupling properties given in Section 9.1.3. If class A is coupled to classes B and C, both classes B and C have CBO = 1, assuming no other coupling connections. Now assume that classes B and C are merged creating new class D. Class D will have CBO = 1, as it is only coupled to class A. Property 5 is not satisfied because the original classes A and B were disjoint, and the property requires that the coupling of the merged classes be the sum of their coupling. Property 5 would be satisfied if the measure counted the number of connections rather than the number of classes that the class of interest is coupled to.

Coupling example

Message passing coupling (*MPC*) was introduced by Li and Henry and formalized by Briand, Daly, and Wüst (Li and Henry 1993, Briand et al. 1999b). The MPC value of a class is a count of the number of static invocations (call statements) of methods that are external to the class. MPC satisfies all of the properties in Section 9.1.3 including property 5.

Coupling example

Robert C. Martin defines two package-level coupling measures (Martin 2003). These indicate the coupling of a package to classes in other packages:

1. Afferent coupling (C_a): "The number of classes from other packages that depends on the classes within the subject package." Only "class relationships" are counted, "such as inheritance and association." C_a is really the *fan-out* (see Section 9.3.8) of a package.
2. Efferent coupling (C_e): "The number of classes in other packages that the classes in the subject package depend on" via class relationships. C_e is a form of the *fan-in* of a package.

These measures satisfy all of the properties in Section 9.1.3, including property 5 as long as a class can only belong to a single package.

Martin suggests that high efferent coupling makes a package *unstable* as it depends on too many imported classes. He defines the following *instability* (I) *metric:*

$$I = \frac{C_e}{C_a + C_e}$$

Measuring Cohesion in Object-Oriented Systems

- *Method cohesion* is conceptually the same as the cohesion of an individual function or procedure.
- *Class cohesion* is an intraclass attribute. It reflects the degree to which the parts of a class—methods, method calls, fields, and attributes belong together. A class with high cohesion has parts that belong together because they contribute to a unified purpose. Most of the proposed cohesion metrics are class-level metrics.

Cohesion metrics

- Researchers have developed many different object-oriented cohesion metrics
- The majority of object-oriented cohesion metrics are calculated by <u>inspecting the syntax</u> of the software.

Cohesion Metrics

- *Lack of cohesion metric* (*LCOM*) is a metric proposed by *Chidamber and Kemerer* (Chidamber and Kemerer 1994). Here, the cohesion of a class is characterized by how closely the local methods are related to the local instance variables in the class.
- LCOM is defined as the number of disjoint (i.e., nonintersecting) sets of local methods. Two methods in a class intersect if they reference or modify common local instance variables.
- LCOM is an inverse cohesion measure; higher values imply lower cohesion.

Cohesion Metrics

- *Tight class cohesion* (*TCC*) and *loose class cohesion* (*LCC*) are based on connections between methods through instance variables (Bieman and Kang 1995). Two or more methods have a *direct connection* if they read or write to the same instance variable. Methods may also have an *indirect connection* if one method uses one or more instance variables directly and the other uses the instance variable indirectly by calling another method that uses the same instance variable. *TCC* is based on the relative number of direct connections:

$$TCC(C) = NDC(C)/NP(C)$$

- where $NDC(C)$ is the number of direct connections in class C and $NP(C)$ is the maximum number of possible connections. *LCC* is based on the relative number of direct and indirect connections:
  - $LCC(C) = (NDC(C) + NIC(C))/NP(C)$

where $NIC(C)$ is the number of indirect connections.

Cohesion Metrics

- *Ratio of cohesive interactions* (*RCI*) is defined in terms of *cohesive interactions* (*CIs*). *RCI* is the relative number of *CI*s:

$$RCI(C) = NCI(C)/NPCI(C)$$

where *NCI*(*C*) is the number of actual *CI*s in class *C* and *NPCI*(*C*) is the maximum possible number of *CI*s. *RCI* satisfies all four cohesion properties

Object-Oriented Length Measures

- In object-oriented systems, length/distances depend on the perspective and the model representing an appropriate view of the system
- Inheritance in a class diagram is represented as a hierarchy or tree of classes.
- The *depth of inheritance tree* (*DIT*) is a metric suite of object-oriented metrics defined by *Chidamber and Kemerer 1994*
- The nodes in the tree represent classes, and for each such class, the DIT metric is the length of the maximum path from the node to the root of the tree.
- DIT is a measure of how many ancestor classes can potentially affect this class

Bieman and Zhao studied inheritance in 19 C++ systems containing 2744 classes (Bieman and Zhao 1995). The systems included language tools, GUI tools, thread software, and other systems. They found that the median DIT of the classes in 11 of the systems was either 0 or 1. Classes in the studied GUI tools used inheritance the most with a mean class DIH of 3.46 and median class DIH values ranging from 0 to 7. The maximum measured DIH for all GUI tool classes was 10.

Object-Oriented Reuse Measure

- One of the key benefits of object-oriented development is its support for reuse through data abstraction, inheritance, encapsulation, etc.
- Two perspectives of reuse:
- (1) *client perspective:* the perspective of **a new system or system component** that can potentially reuse existing components. The potential reuse measures include the number of direct and indirect server classes and interfaces reused.
- (2) *server perspective:* the perspective of the **existing components** that may potentially be reused, for example, a component library or package. From the server perspective, we are concerned with the way a particular entity is being reused by clients.