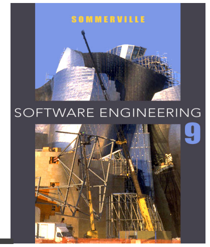


Software Engineering

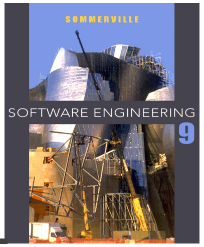
Prof. Dr. Stefan Kramer

(lecture slides based on Ian Sommerville)

Chapter 24 - Quality Management

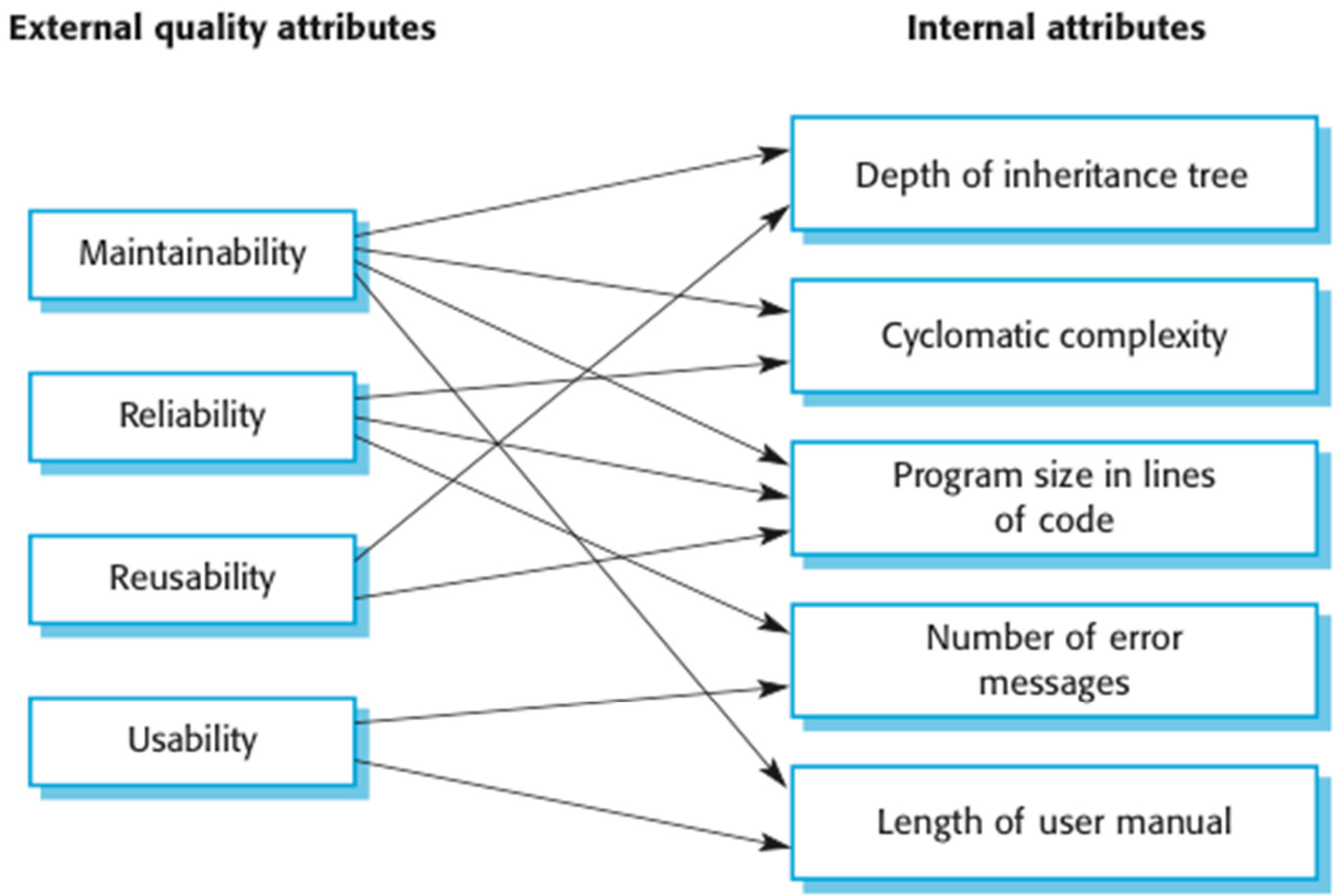
Lecture 2

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.

Figure 24.10 Relationships between internal and external software



Dynamic and static product metrics

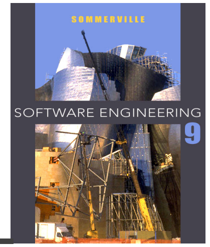
✧ **Dynamic metrics:** collected during **execution**

- closely related to **software quality attributes** like **efficiency** and **reliability**
- it is relatively easy to measure the **response time** of a system (performance attribute) or the **number of failures** (reliability attribute).

✧ **Static metrics:** computed from **system representations**

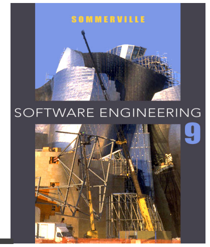
- **indirect relationship** with quality attributes
- help assess **complexity**, **understandability** and **maintainability**

Static software product metrics



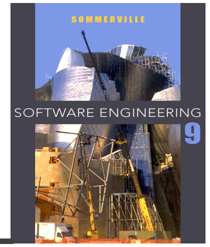
Software metric	Description
Fan-in/Fan-out	Fan-in is a measure of the number of functions or methods that call another function or method (say X). Fan-out is the number of functions that are called by function X. A high value for fan-in means that X is tightly coupled to the rest of the design and changes 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.
Cyclomatic complexity	This is a measure of the control complexity of a program. This control complexity may be related to program understandability. I discuss cyclomatic complexity in Chapter 8.

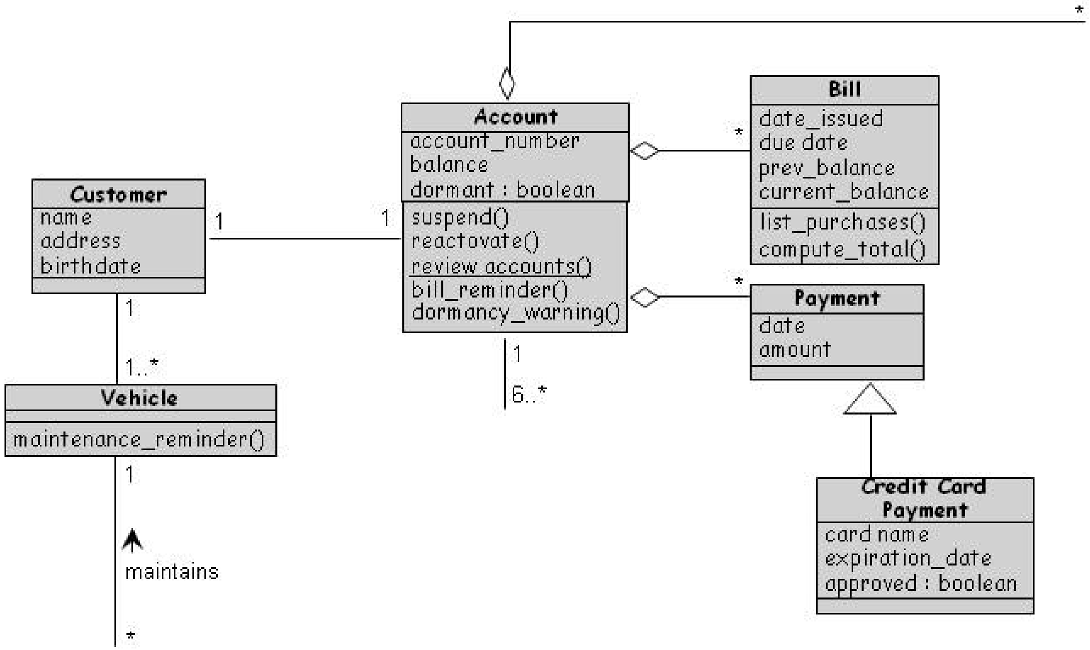
The Chidamber and Kemerer object-oriented metrics suite



Object-oriented metric	**Description**
Depth of inheritance tree (DIT)	This represents the number of discrete levels in the inheritance tree where subclasses inherit attributes and operations (methods) from superclasses. The deeper the inheritance tree, the more complex the design. Many object classes may have to be understood to understand the object classes at the leaves of the tree.
Number of children (NOC)	This is a measure of the number of immediate subclasses in a class. It measures the breadth of a class hierarchy, whereas DIT measures its depth. A high value for NOC may indicate greater reuse. It may mean that more effort should be made in validating base classes because of the number of subclasses that depend on them.

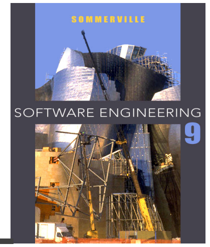
CK Suite Applied to Royal Service Station's System Design





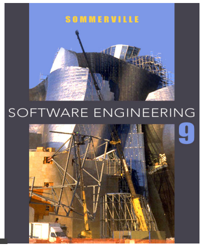
Metric	Bill	Payment	Credit Card Payment	Account	Customer	Vehicle
Weighted Methods / Class	2	0	0	5	0	1
Number of Children	0	1	0	0	0	0
Depth of Inheritance Tree	0	0	1	0	0	0

The Chidamber and Kemerer object-oriented metrics suite

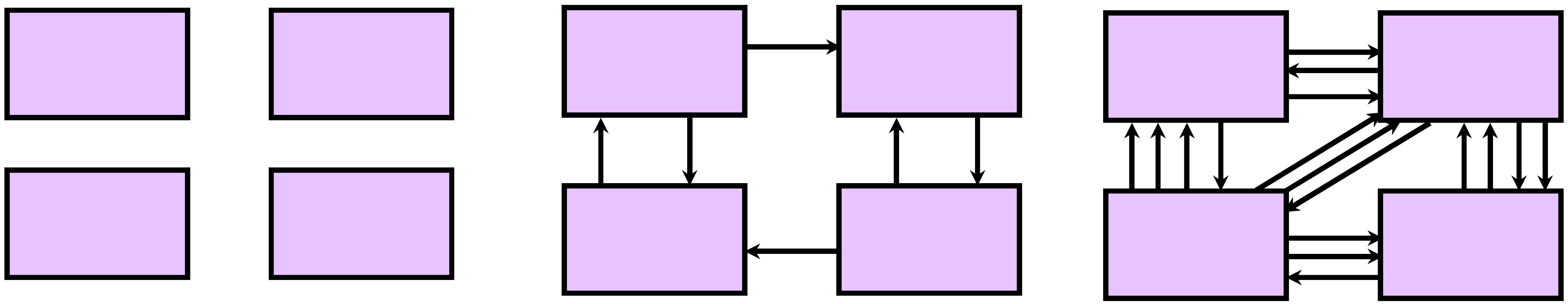


Object-oriented metric	Description
Coupling between object classes (CBO)	Classes are coupled when methods in one class use methods or instance variables defined in a different class. CBO is a measure of how much coupling exists. A high value for CBO means that classes are highly dependent, and therefore it is more likely that changing one class will affect other classes in the program.
Lack of cohesion in methods (LCOM)	LCOM is calculated by considering pairs of methods in a class. LCOM is the difference between the number of method pairs without shared attributes and the number of method pairs with shared attributes. The value of this metric has been widely debated and it exists in several variations. It is not clear if it really adds any additional, useful information over and above that provided by other metrics.

Coupling



- Two modules are **tightly coupled** when they depend a great deal on each other
- **Loosely coupled** modules have some dependence, but their interconnections are weak
- **Uncoupled** modules have no interconnections at all; they are completely unrelated

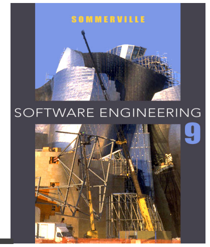


Uncoupled - no dependencies

Loosely coupled - some dependencies

Tightly coupled - many dependencies

Coupling: Types of Coupling



- Content coupling: component modifies an internal data item in another or branches into the middle of another component
- Common coupling:

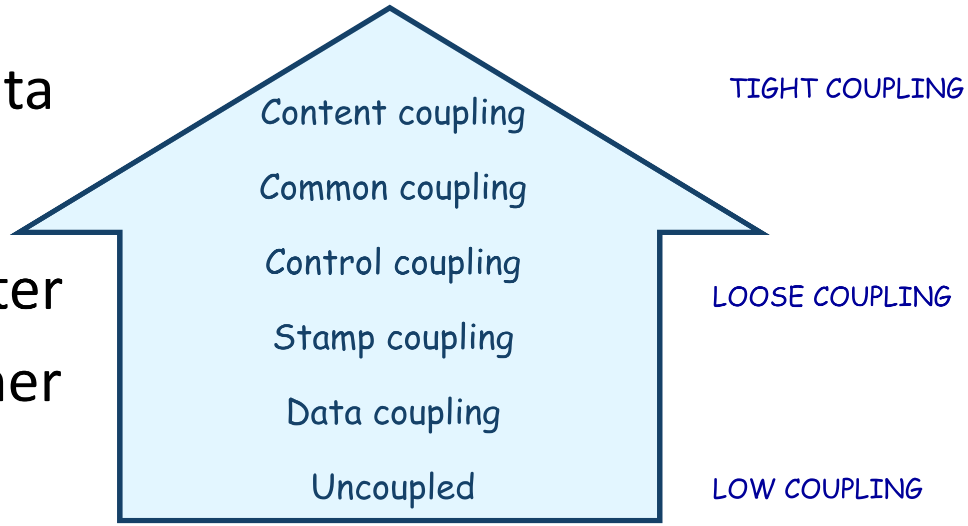
  modules access common data
- Control coupling:

  one module passes parameter

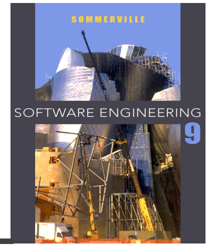
  to control behavior of another
- Stamp coupling:

  complex data types are passed between modules
- Data coupling:

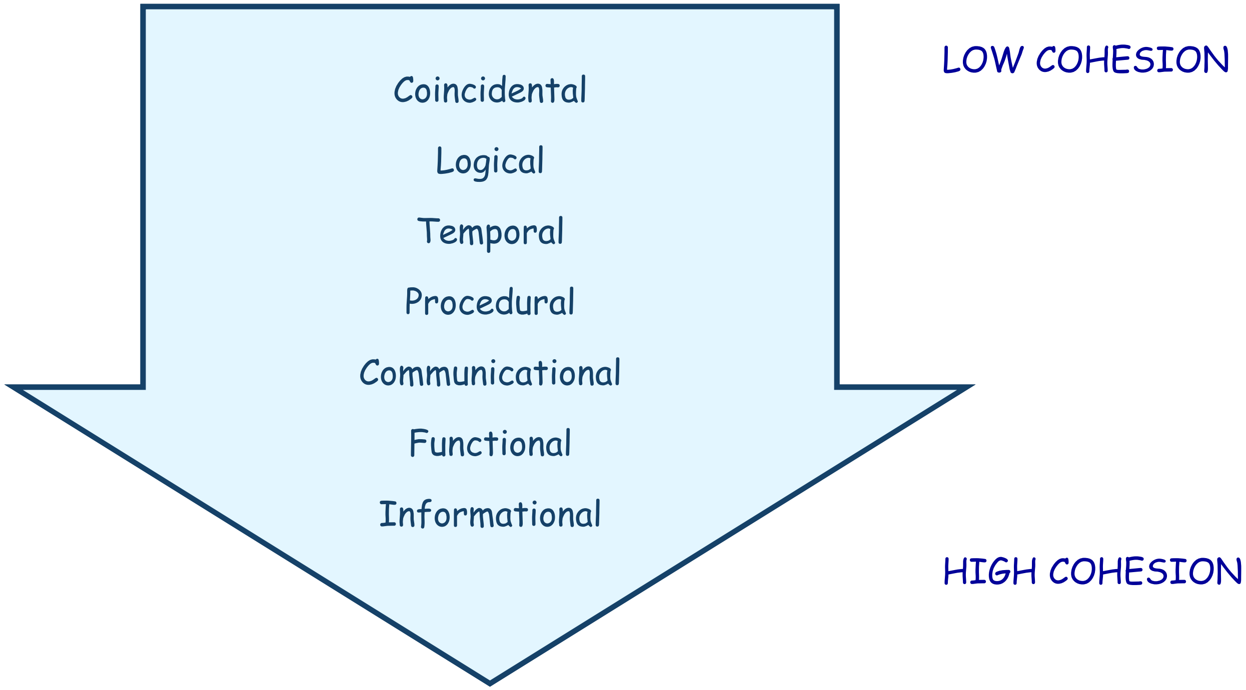
  only atomic data types are passed



Cohesion



- **Cohesion** refers to the dependence within and among a module's internal elements (e.g., data, functions, internal modules)



Cohesion (continued)

- Coincidental (worst degree)

  Parts are unrelated to one another

- Logical

  Parts are related only by the logic structure of code

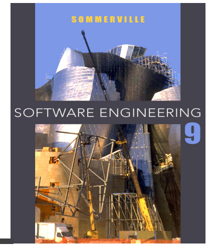
- Temporal

  Module's data and functions related because they are used at the same time in an execution

- Procedural

  Similar to temporal, and functions pertain to some related action or purpose

Cohesion (continued)



- Communication

  Operates on the same data set

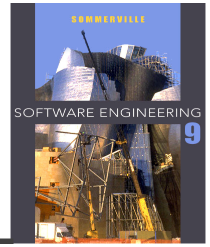
- Functional (ideal degree)

  All elements essential to a single function are contained in one module, and all of the elements are essential to the performance of the function

- Informational

  Adaptation of functional cohesion to data abstraction and object-based design

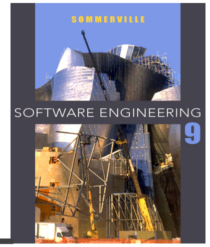
Coupling and Lack of Cohesion According to Chidamber and Kemerer: Notation

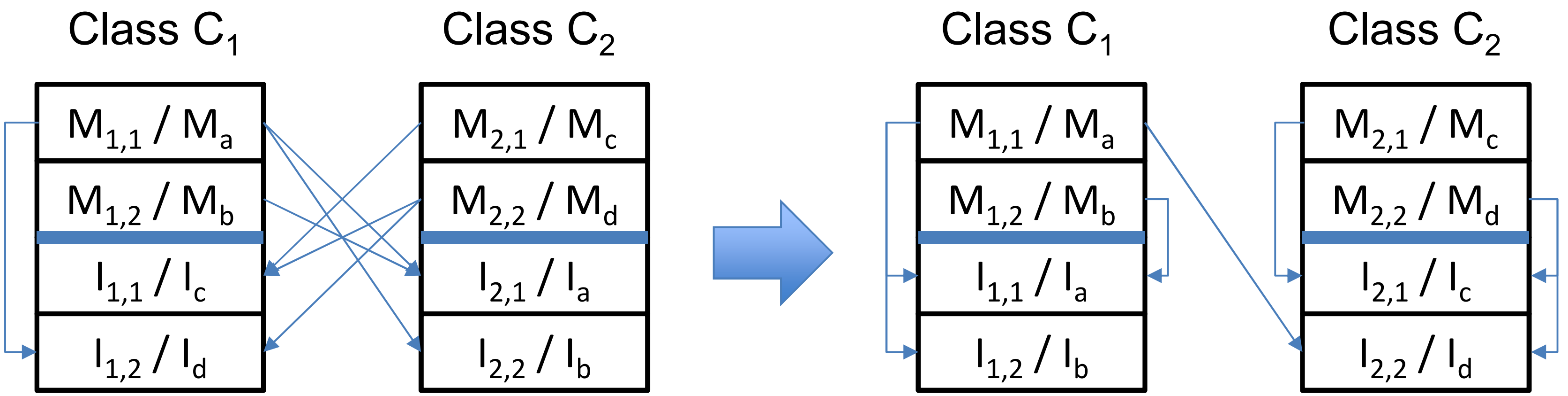


- Classes of software system: $C_1, \ldots, C_k$
- m_C methods of a class C: $M_{C,1}, \ldots, M_{C,m_C}$
- n_C instance variables of a class C: $I_{C,1}, \ldots, I_{C,n_C}$
- Set of all methods $M = \cup_{i,j} M_{i,j}$ and all instance variables in a system $I = \cup_{i,j} I_{i,j}$

- Relation of accesses of methods to instance variables: $R_{M,I} \subseteq M \times I$
- Relation of method calls (one method calling another): $R_{M,M} \subseteq M \times M$

Toy Example: Before and After Re-Design

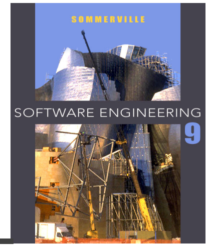




$CBO(C_1) = 1$	$CBO(C_2) = 1$	$CBO(C_1) = 1$	$CBO(C_2) = 0$
$LCOM(C_1) = 1$	$LCOM(C_2) = 1$	$LCOM(C_1) = 0$	$LCOM(C_2) = 0$

Note there is only one distinct pair of methods to go into the calculations!

Coupling and Lack of Cohesion According to Chidamber and Kemerer: Auxiliary Functions



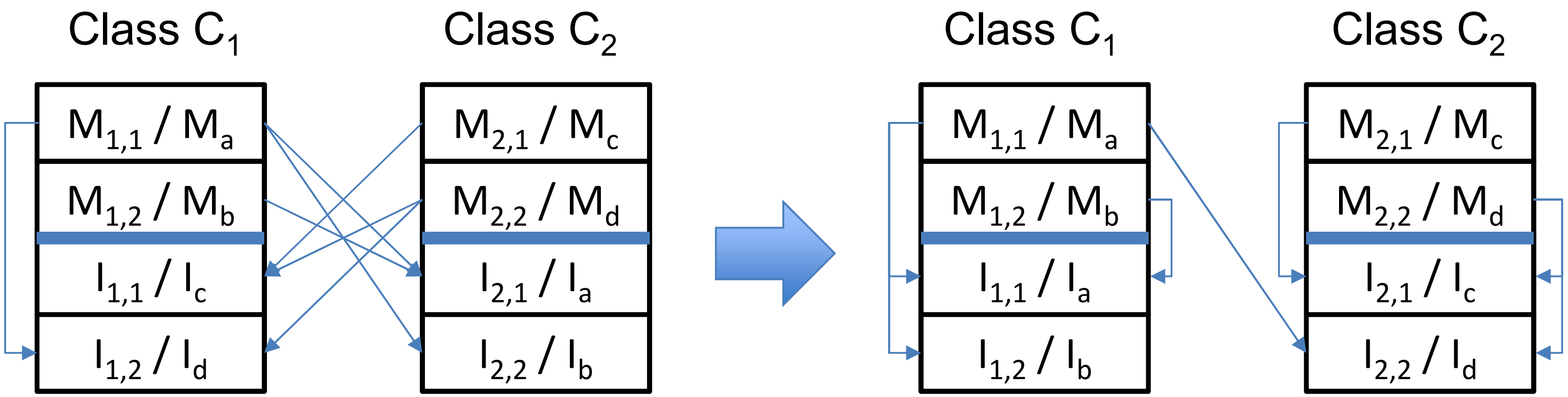
- Auxiliary function f returning all instance variables in a given set of classes C_1 to C_l that are accessed from a given method $M_{i,j}$:

$$f(M_{i,j}, \{C_1, \ldots, C_l\}) = \{I_{r,s} \mid (M_{i,j}, I_{r,s}) \in R_{M,I} \land r \in \{1, \ldots, l\}\}$$

- Auxiliary function g returning all methods in a given set of classes C_1 to C_l that are called from a given method $M_{i,j}$:

$$g(M_{i,j}, \{C_1, \ldots, C_l\}) = \{M_{r,s} \mid (M_{i,j}, M_{r,s}) \in R_{M,M} \land r \in \{1, \ldots, l\}\}$$

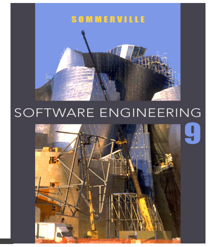
Toy Example: Before and After Re-Design



$CBO(C_1) = 1$	$CBO(C_2) = 1$	$CBO(C_1) = 1$	$CBO(C_2) = 0$
$LCOM(C_1) = 1$	$LCOM(C_2) = 1$	$LCOM(C_1) = 0$	$LCOM(C_2) = 0$

Note there is only one distinct pair of methods to go into the calculations!

Coupling and Lack of Cohesion According to Chidamber and Kemerer



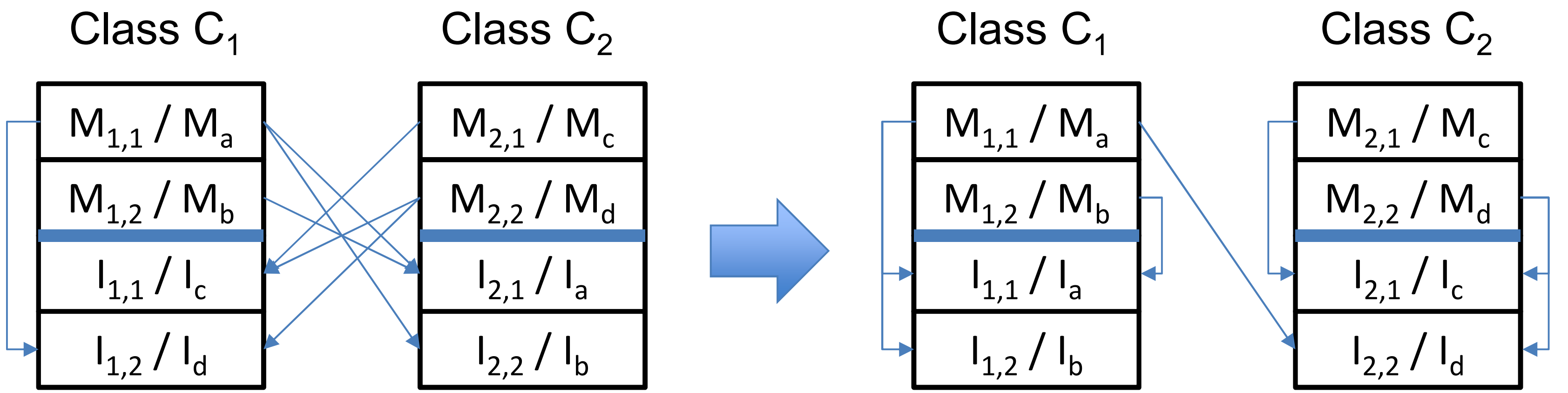
Coupling between objects: set of objects that are related to a given object by either access of instance variables or method calls:

$$\begin{aligned} CBO(C_i) = |\{o \mid\ & I_{o,m} \in \textstyle\bigcup_j f(M_{i,j}, C \setminus \{C_i\}) \lor \\ & M_{o,n} \in \textstyle\bigcup_j g(M_{i,j}, C \setminus \{C_i\})\}| \end{aligned}$$

Lack of cohesion: For each distinct pair of methods from a class we *add one if they do not share a single used instance variable* and *subtract one if they do share one or more used instance variables* (**1** returns one if the argument evaluates to true and zero otherwise). If it less than zero, it is by definition set to zero.

$$LCOM(C_i) = \Sigma_{j,\ k,\ j < k} (2 \times \mathbf{1}[f(M_{i,j}, \{C_i\}) \cap f(M_{i,k}, \{C_i\}) = \emptyset] - 1)$$

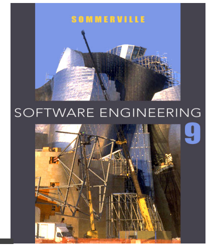
Toy Example: Before and After Re-Design



$CBO(C_1) = 1$ $CBO(C_2) = 1$ $CBO(C_1) = 1$ $CBO(C_2) = 0$

$LCOM(C_1) = 1$ $LCOM(C_2) = 1$ $LCOM(C_1) = 0$ $LCOM(C_2) = 0$

Note there is only one distinct pair of methods to go into the calculations!



Remarks and Conclusions

- ✧ Software metrics make certain characteristics of software systems objective and quantifiable
  - possible for use in predictive models
- ✧ Coupling and cohesion metrics measure two aspects of **modularity**:
  - trade-off between coupling and cohesion
  - "abstract model" of modularity
  - application not straightforward (how about inheritance?, how do you count getters/setters?, etc.) and language-dependent (which OO language constructs are available and in which form?)
  - classes should also have a clear semantics (“why are you here?!”)
- ✧ No good proposal for *one* metric for modularity yet