3.1 COUPLING: AN OVERVIEW

3.1.1 Measuring Coupling

Coupling is one of the most significant attributes affecting the overall quality of the design. No generally accepted metric exists for coupling; however, generally, it is accepted that too much coupling in a design leads to increased system complexity Harrison et al. (1998); therefore, high coupling is considered as undesired property. Following describes some of the known efforts that are made for measuring coupling.

Yourdon and Constantine define the coupling as a degree of interdependence between modules. Bansiya also defines coupling as a dependency of an object on other objects in a design. He uses DCC (Direct Class Coupling) metric that counts the number of classes that a class is directly related to. This metric includes the classes directly related by attribute declaration and message passing (parameter list) in methods Bansiya et al. (2002). Chidamber and Kemerer have also discussed the coupling in the context of OO paradigm, in their opinion; two classes are coupled; if the method of one class uses any method or instance of other class. CBO (Coupling between object classes) metrics counts the number of coupled classes. In CBO metric, a class is coupled to other class if it uses the method or attribute defined in other class. However, CBO metric does not distinguish among different types of interactions between two classes Briand et al. (1997). Whereas, Briand and his colleagues have presented a detailed suite for
measuring the C++ class coupling by distinguishing different types of interactions among the classes in C++. Fenton and Pfleeger recognize coupling as a pair-wise measurement of the modules. They have discussed about measuring the coupling on ordinal scale and they have classified the coupling in six pair-wise module relationships on ordinal scale Fenton et al (1998). To measure coupling, Fenton have presented an idea of set of classification of pair-wise relationships between modules x and y; starting from relation \( R_0 R_1 R_2 \) to \( R_n \). Relations are subscripted from the least dependent at the start and the most dependent at the end, so that \( R_i > R_j \) for \( i > j \). Modules x and y are said to be the loosely coupled if \( i \) value is somewhere in the start (near to zero) and modules x and y are said to be tightly coupled if \( i \) value is somewhere in the end (near to n).

### 3.1.2 Coupling in Procedure-oriented Systems

High quality software design, among many other principles, should obey the principle of low coupling. Stevens et al., who first introduced coupling in the context of structured development techniques, define coupling as “the measure of the strength of association established by a connection from one module to another.” Therefore, the stronger the coupling between modules, i.e., the more integrated they are, the more difficult these modules are to understand, change, and maintain and thus the resulting software system will be more complex. While software reuse demands for separation of the subsystems, the software development process demands for integration of subsystems. Consequently, achieving both effective software reuse and an efficient software development process seem hard to unite. Hence, higher the coupling between two packages, the reuse capability of those packages is lower as it is hard to separate two highly coupled packages. Similarly, higher the coupling, more complex is the system is and it is hard to maintain such system.
Myers, Glenford J defined six distinct degrees of coupling to measure the interdependence among the modules.

- **Data Coupling** - Two modules are data coupled if they pass data through scalar or array parameters.

- **Stamp Coupling** - Two modules are stamp coupled if they pass data through a parameter that is a record. Stamp coupling is perceived as worse than data coupling because any change to the record will affect all of the modules that refer to that record, even those modules that do not refer to the fields that are changed.

- **Control Coupling** - Two modules are control coupled if one passes a flag value that is used to control the internal logic of the other.

- **External Coupling** - Two modules are external coupled if they communicate through an external medium such as a file.

- **Common Coupling** - Two modules are common coupled if they refer to the same global data.

- **Content Coupling** - Two modules are content coupled if they access and change each other's internal data state or procedural state.

Offutt et al (1993) includes ‘no coupling’ as the zeroth level. If two modules are coupled in more than one way, they are considered to be coupled at the highest level. Page-Jones also introduced the notion of tramp coupling, where data flows through many intermediate modules i.e. from the module where the data are defined to where they are used. This differs from the other coupling levels in that it measures the coupling among many modules instead of just two modules. Tramp coupling is a mostly form of data coupling, but
can also be stamp or control coupling. Though the use of structured design is outdated by the object-oriented designs, these coupling levels are not limited to the structured design. They are modified by the researchers to accommodate the object oriented design.

### 3.1.3 Coupling in Object-oriented Systems

Object-oriented design and analysis incorporate the data and its functionality into objects thereby reducing the coupling between objects. But still object-oriented mechanisms does not guarantee minimum coupling. Coad and Yourdon, in their books, introduced the concept of coupling in the Object Oriented design. As opposed to procedure-oriented systems where the module is the only one subject of interest, in object-oriented systems there are several subjects of interest, such as methods and object classes. Coupling properties of the various subjects of interest are not independent from each other i.e. the coupling properties between methods of different object classes highly influence the coupling properties between these object classes. Budd et. al. (1997) gives the classical concepts of coupling in terms of object-oriented languages. The terms internal data coupling and Global data coupling seem to fit into the object-oriented paradigm. But in the literature, it is believed data coupling is not of much important in object-oriented paradigm as the main communication mechanism is message passing rather than data sharing.

In OO design, the coupling of a class means the measurement of the interdependence of class with the other classes. In a design of reasonable size (say design size is ten classes); normally classes do not exist in absolute isolation. By going through any OO source code of a working system, one can see that nearly all classes have some kind of relationships with other classes in the design. These relationships, among the classes, create pair-wise interdependencies. Such pair-wise relationships among the classes are the results of design decisions which are made on the specifications of the
system. A good design decision may create a good relationship and a bad
design decision may create a bad relation. Here, by good design decision, we
mean a decision that makes the OO design easy to reuse, understandable and
flexible for modification and adoption in future. Gurp and Bosch, based on an
industrial case study, have presented five reasons for software erosion, which
revolves around the design decisions made on different stages Gurp Bosch
et.al.(2002). During such stages designer team of the system decides which
relationship should be used to fulfill the goals (specifications and constraint)
of a particular system. Based on the goals of the system and the design skills,
a team may design a system that exhibits low or high coupling among the
classes, they have not described their model for measuring coupling in terms
of OO paradigm. Most of OO metrics to measure coupling are the counting
metrics, which count the number of times a class establishes an OO
relationship with other class. Counting all types of OO relationships without
distinguishing different types of interactions among the classes is not the best
way to reflect to class coupling with other classes. However, class coupling
can also be defined by distinguishing UML relationships (i.e. association,
inheritance and dependency) among the classes.

3.1.4 UML Relationships

As mentioned earlier, very few classes in OO design stand alone.
Relatively a large number of classes collaborate with others in OO design.
Therefore, while modeling OO design, the relations are also modeled based
on how classes stand to each other. According to Booch, in OO modeling
there are three types important relationships among the classes, namely
dependency, inheritance and association relationships. Dependency represents
the using relationships among the classes; inheritance relationship connects
generalized classes to their specialized classes; and association relationship
shows the structural relationship among the objects.
Each of these coupling dimensions induces that the behavior of a class $C$ depends on the behavior of $C'$ if $C$ is related to $C'$ by one of the relationships mentioned before. Or, to put it in other words, $C$ has to have some information about $C'$ such that in case $C'$ changes, $C$ knows what to change, too. The degree of coupling can be described as how much, how complex and how explicit this information has to be. On one end of the scale, low coupling is described by a small, simple and explicit inter-relationship between methods and between object classes. In general, low coupling correlates to good software quality in terms of better maintainability and reusability. On the other end, high coupling is described by a large, complex and implicit inter-relationship making maintenance a nightmare and reuse even impossible.

3.2 COUPLING FRAMEWORKS

Several different authors describe frameworks to characterize different approaches to coupling and to assign relative strengths to different types of coupling. A framework defines what constitutes coupling. This is done in an attempt to determine the potential use of coupling metrics and how different metrics complement each other.

Eder identifies three different types of relationships. These are interaction relationships between methods, component relationships between classes, and inheritance between classes. These relationships are then used to derive three different dimensions of coupling which are classified according to different strengths:

1. **Interaction coupling**: Two methods are said to be interaction coupled if i) one method invokes the other, or ii) they communicate via the sharing of data. There are seven types of interaction coupling.
2. **Component coupling**: Two classes \( c \) and \( d \) are component coupled, if \( d \) is the type of either i) an attribute of \( c \), or ii) an input or output parameter of a method of \( c \), or iii) a local variable of a method of \( c \), or iv) an input or output parameter of a method invoked within a method of \( c \). There are four different degrees of component coupling.

3. **Inheritance coupling**: two classes \( c \) and \( d \) are inheritance coupled, if one class is an ancestor of the other. There are four degrees of inheritance coupling.

Hitz and Montazeri derive two different types of coupling, object and class-level coupling Hitz et al (1995). These are determined by the state of an object (value of its attributes at a given moment at runtime) and state of an object's implementation (class interface and body at a given time in the development cycle).

Class level coupling (CLC) results from state dependencies between two classes in a system during the development cycle. This can only be determined from a static analysis of the design documents or source code. This is important when considering maintenance and change dependencies as changes in one class may lead to changes in other classes which use it. Object level coupling (OLC) results from state dependencies between two objects during the run-time of a system. This depends on concrete object structure at runtime, which in turn is determined by actual input data. Therefore, it is a function of design or source code and input data at run-time. This is relevant for run-time oriented activities such as testing and debugging.
3.2.1 Criteria for Coupling Framework

In the framework by Briand et al, coupling is constituted as interactions between classes Briand et al (1997). The strength is determined by the type of the interaction (Class-Attribute, Class-Method, Method-Method), the relationship between the classes (Inheritance, Other) and the interaction's locus of impact (Import/Client, Export/Server). They assign no strengths to the different kinds of interactions. There are three basic criteria in the framework which are as follows:

1. **Type of interaction**: This determines the mechanism by which two classes are coupled. A class-attribute interaction is present if aggregation occurs, that is, if a class \( c \) is the type of an attribute of class \( d \). A class-method interaction occurs if a class \( c \) is the type of a parameter of method \( m_d \) of a class \( d \), or if a class \( c \) is the return type of method \( m_d \). A method-method interaction occurs if a \( m_d \) of a class \( d \) directly invokes a method \( m_c \), or a method \( m_d \) receives via parameter a pointer to \( m_c \), thereby invoking \( m_c \) indirectly.

2. **Relationship**: An inheritance relationship occurs if a class \( c \) is an ancestor of class \( d \) or vice versa. Friendship is present if a class \( c \) declares class \( d \) as its friend, which grants class \( d \) access to the non-public elements of \( c \). There is another relationship when no inheritance or friendship relationship is present between classes \( c \) and \( d \).

3. **Locus**: If a class \( c \) is involved in an interaction with another class, a distinction is made between export and import coupling. Export is when a class \( c \) is the used class or server in the interaction. Import is when a class \( c \) is the using class or client in the interaction. Briand et al. outlines a new unified
framework for coupling in object-oriented systems Briand et al (1999). It is characterized based on the issues identified by comparing existing coupling frameworks. There are six different criteria in the framework and each criterion determines one basic aspect of the resulting measure.

The criteria are as follows:

1. **The type of connection**: This determines what constitutes coupling. It is the type of link between a client and a server item which could be an attribute, method, or class.

2. **The locus of impact**: This is import or export coupling. Import coupling analyses attributes, methods, or classes in their role as clients of other attributes, methods, or classes. Export coupling analyses the attributes, methods, and classes in their role as servers to other attributes, methods or classes.

3. **The granularity of the measure**: This is the domain of the measure that is, what components are to be measured and how to count coupling connections.

4. **The Stability of server**: Should both stable and unstable classes be included? Classes can be, a) stable which are those that are not subject to change in the project at hand, for example classes imported from libraries, or b) unstable which are those which are subject to development or modification in the project at hand.

5. **Direct or indirect coupling**: Should only direct connections be counted or should indirect connections also be taken into account?
6. **Inheritance**: Inheritance-based versus non-inheritance-based coupling. Also how to account for polymorphism and how to assign attributes and methods to classes.

### 3.2.2 Unified Framework for Coupling

Briand et al (1999) introduces a unified framework for defining coupling measurement in object-oriented systems. They review three other previous attempts at defining such a framework and attempt to improve and unify the terminology. The previous frameworks have been proposed by Eder et al (1994), Ritz et al (1995) and an earlier attempt by Briand et al (1997). The framework utilizes mathematical notation to specifically define the different types of relationships. There are many definitions which are stated within the framework, for brevity, the definitions necessary for understanding the proposed metrics will be introduced.

**System, Classes, Inheritance Relationships**: An object-oriented system consists of a set of classes \( C \). There can exist inheritance relationships between classes such that for each class \( c \in C \) let

- **Parents\((c)\) \( \subseteq C \)** be the set of parent classes of \( c \).
- **Children\((c)\) \( \subseteq C \)** be the set of children classes of \( c \).
- **Ancestors\((c)\) \( \subseteq C \)** be the set of ancestor classes of \( c \).
- **Descendants\((c)\) \( \subseteq C \)** be the set of descendent classes of \( c \).

A class has a set of methods. A method can be virtual or non-virtual and either inherited, overridden, or newly defined, all of which have implications for measuring coupling.

- **Methods of a class**: For each class \( c \in C \) let \( M(c) \) be the set of methods of class \( c \).
- **The set of all Methods:** \( M(C) \) is the set of all methods in the system and is represented as

\[
M(C) = \bigcup_{c \in C} M(c)
\]

- **The set of Methods Implemented in a Class:** \( M_I(C) \subseteq M(c) \) be the set of methods implemented in \( c \) i.e. methods that \( c \) inherits but overrides or non-virtual non-inherited methods of \( c \).

- **Polymorphic Identification:** \( P(m) \) is the function to identify which class the method \( m \) is dynamically bound to. \( P(m) = c \in C \) where \( m \in M(c) \)

To measure coupling of a class \( c \), it is necessary to define the set of methods that \( m \in M(c) \) invokes and the frequency of these invocations. Method invocations can be either static or dynamic. For static invocations, the invoked method is determined by the type of the variable that references the object. For dynamic invocations, the invoked method is determined by a late-binding at run-time to the polymorphic type. Now we discuss about the transitive relation upon method invocations. A method invocation may possibly invoke another method and so on.

**The set of Statically Invoked Methods of \( m \):** Let \( c \in C \), \( m \in M_I(c) \), and \( m' \in M(c) \). Then \( m' \in \text{SIM}(m) \iff \exists d \in C \text{ such that } m' \in M(d) \) and the body of \( m \) has a method invocation where \( m' \) is invoked for an object of static type class \( d \).

**The set of Polymorphically Invoked Methods of \( m \):** \( \text{PIM}(m) \) is the set of all polymorphically invoked methods on \( m \). Let \( c \in C \), \( m \in M_I(c) \), and \( m' \in M(c) \). Then \( m' \in \text{PIM}(m) \iff \exists d \in C \text{ such that } m' \in M(d) \) and the
The body of m has a method invocation where m' may, because of polymorphism and dynamic binding, be invoked for an object of dynamic type d.

The Transitive Closure on a Set of Invoked Methods of m: T(m) is the transitive closure on a set of invoked methods. Let m be a method, whether it be statically or polymorphically invoked. Let m be defined to be m₀, where m₀ can invoke m₁, m₁ can invoke m₂, and so on. Let

\[ T = \bigcup_{m \in M(C)} m_i \]

Classes have attributes which are either inherited or newly defined.

The set of all Attributes: A(C) is the set of all attributes in the system and is represented as

\[ A(C) = \bigcup \{ A(c) \mid c \in C \} \]

Methods may reference attributes. These attributes may not be part of the encompassing class, therefore coupling it to the referenced encompassing class.

The set of Attributes referenced by the method m: For each m ∈ M(C) let AR(m) be the set of attributes reference by method m. To ensure proper usage between terms, a uses predicate has to be defined.

Uses: Let \( c \in C, d \in C, \text{uses}(c, d) \iff (\exists m \in M_l(c): \exists m' \in M_l(d): m' \in PIM(m)) \lor (\exists m \in M_l(c): \exists a \in A_l(d): a \in AR(m)) \)

A class c uses a class d if a method implemented in class c references a method or an attribute implemented in class d.
3.2.3 Interaction Coupling

Briand et al (1996) proposes a set of properties that defines the concept coupling of a modular system where a module is a subsystem. Given a modular system $MS$, and its intermodule edges sub graph $S$, the intermodule coupling of modular system is the minimum description length of the relationships in $S$.

$$\text{Coupling}(MS) = \sum_{i=1}^{n} I(S_i) - I(S)$$

Intermodule coupling, discussed above, measures attributes of intermodule relationships. Intramodule coupling of a modular system is closely related to the coupling family above.

Intramodule coupling is applicable to the same software engineering abstractions as intermodule coupling. The intramodule coupling of a modular system $MS$, is the minimum description length of the relationships in $S'$.

$$\text{Intramodule Coupling}(MS) = \sum_{i=1}^{n} I(S'_i) - I(S')$$

The definitions are based on information theory, taking patterns of relationships into account, rather than just counts of graph attributes.

Certain properties are expected to be contained for coupling of a modular system. They are:

1. Non negativity: Coupling of a modular system is non negative.
2. Null value: Coupling of a modular system is null if its set of intermodule edges is empty.
3. Monotonicity: Adding an intermodule edge to a modular system does not decrease its coupling.

4. Merging of modules: If two modules, $m_1$ and $m_2$ are merged to form a new module $m_{12}$, that replaces $m_1$ and $m_2$, then the coupling of the modular system with $m_{12}$ is not greater than the coupling of the modular system with $m_1$ and $m_2$.

5. Disjoint module additivity: If two modules $m_1$ and $m_2$ which have no intermodule edges between nodes in $m_1$ and nodes in $m_2$, are merged to form a new module $m_{12}$, that replaces $m_1$ and $m_2$, then the coupling of the modular system with $m_{12}$ is equal to the coupling of the modular system with $m_1$ and $m_2$.

Methods are coupled by interaction in terms of invocation of each other and/or sharing of data. Since interaction coupling is most similar to the classical definition of coupling between modules, we shall adopt the various degrees of classical coupling to describe interaction coupling. Here we discuss how interaction coupling in OO systems differs from the classical notion of coupling. The difference mainly stems from two interrelated facts. Firstly, methods belong to object classes. This implies that object classes may be interaction coupled, too. This implies that we have to distinguish interaction between different classes from interaction within a single class. Here all degrees of interaction coupling is considered – from worst to best.

1. Content

Content coupling is the worst form of coupling. It means that one method directly accesses parts of the internal structure, i.e., the implementation of another method. Thus one method has to know exactly all internals of the other methods, and any change in one method may influence the other. The object-oriented paradigm in general, and encapsulation and
information hiding in particular prohibit that a method directly accesses the implementation of another method or hidden instance variables of a different class. However, content coupling may occur if the programmer uses features of some OO languages which break the information hiding property.

2. **Common**

Coupling is rated common if methods communicate via an unstructured, global, shared data space. Common coupling is better than content coupling since all implicit communication channels are collected in the common area. Nevertheless, it is still a problem since the number of possible connections between methods is polynomial, and the locality principle of good software design is not considered at all. Encapsulation and information hiding prohibit common coupling.

3. **External**

External coupling improves common coupling by structuring the global, shared data space. However, the locality principle is still violated, thus most deficiencies of common coupling remain. Encapsulation and information hiding also prohibit external coupling between methods of different classes. Nevertheless, it may occur in OO systems based on languages which provide globally visible variables. For methods of the same class, we may find external coupling in the interaction between methods of the same class as they may access the same instance variables which are used similar to global variables in modules. Passing of data may be implemented through these shared instance variables instead of using explicit parameters. In general, the passing of information between different invocations of methods of the same object in instance variables is not considered as external coupling.
Coupling is rated as external, however, if instance variables do not represent the state of the object. Such instance variables contain transient data which is relevant only during the execution of a method and is not relevant at the next invocation of a method. Such data may be reinitialized at each invocation from outside. Transient data should be represented by local variables of the method and passed to other methods as parameters. Like global data in modules transient data in instance variables should be avoided.

The set of instance variables of an object class is minimal if and only if they contain data representing the (static) state of an object of that class. Methods of such a class cannot be external coupled to each other. Methods of different classes, which also do not inherit from each other, may use public instance variables of these classes for passing transient data, which is an even worse form of external coupling. Both kinds of external coupling can and should be avoided.

For external coupled methods we can further distinguish methods and instance variables implemented at the same class from those implemented at a classes $C$ and a super class $C'$ of $C$. The coupling from method $m$ to method $m'$ is defined as inherited external

- if $m$ implemented at $C$ and $m'$ defined at $C$ exchange data through instance variables inherited from $C'$ through explicit parameters. This kind of coupling particularly occurs if the class from which information is inherited has external coupled methods.
- if $m$ implemented at $C'$ and $m'$ defined at $C$ communicate via public instance variables inherited by $C$ from $C'$ where $C \neq C'$ and $C$ is neither a super class nor a subclass of $C'$. 
It is obvious that inherited external coupling between methods is worse than external coupling between methods as it further complicates maintenance. Since inherited variables are directly accessed, inherited external coupling not only uses instance variables to pass transient data but it also breaks encapsulation and information hiding between an object class and its super classes.

4. Control

Methods are control coupled if they communicate exclusively via parameter passing, which implies that they are not content, common, or external coupled, but one method controls the internal logic of the other method. With control inversion, the worst form of control coupling, the called method determines the future execution sequence of the calling method. Control coupling is not prohibited by object-oriented concepts. Therefore, interaction between methods of the same as well as of different classes may be control coupled. Control coupling should be avoided since the change of the implementation of a method may cause hidden changes to the behavior of the control coupled methods. Although control coupling is not prohibited by the object-oriented paradigm, polymorphism and dynamic binding aid in avoiding control coupling. Instead of passing a flag which controls method execution, polymorphism and dynamic binding can be employed.

5. Stamp

Two methods are stamp coupled if, in analogy to classical coupling, they are not control coupled but whole data structures are passed as parameters although only parts of the data structure would suffice. The essence of stamp coupling is as follows: a method depends on some externally defined data structure and has to be changed if this data structure changes, although the change would otherwise not influence the method. Stamp
coupling has to be rephrased for OO systems since there exist two kinds of stamp coupling.

The first kind of stamp coupling is similar to the classical definition of stamp coupling. According to that definition, a method depends on the domain of its parameters. The domain of a parameter may either be an object class, or a basic data type, or a complex data type based on type constructors such as tuple, array, and set. Depending on the domain of the parameter, either basic data values, or complex data values, or objects may be passed as parameters. If a complex data value is passed as parameter stamp coupling occurs if already parts of the complex data value would suffice. This case is analogous to the classical definition of stamp coupling. If objects are passed as parameters a similar problem may occur. This object may again consist of references to other objects. Such an object is also called composite object since it is constructed out of component objects. Thus we should analyze whether the object passes as parameter or merely some of its components are relevant for a method. If an object is passed, and the method uses just some of the object’s components but not the object itself, this interaction shall be classified as stamp coupling. But if the object passed as parameter is used as a whole, then it is called data coupling.

To improve stamp coupling to data coupling an object should be replaced by its components whenever possible, in particular, if only some but not all of its components are necessary. There are rare cases where the replacement of an object by its components may leave extensibility more difficult. Such a situation occurs if only some components of an object are currently needed by some method \( m \) but the object is extended with an additional component and this component is also requested by \( m \). If the object would have been passed as parameter no change of the interface of \( m \) would have been necessary.
The second kind of stamp coupling uncovers dependencies between a method and the domain of instance variables of the same class. The definition of instance variables is external for a method. At first sight it may look strange to consider interaction between methods and instance variables. However, it leads to rules for a better organization of methods. The value of an instance variable is either a basic data value, or a complex data value, or a reference to some object depending on the domain of the instance variable. If a method directly accesses an instance variable although it needs only parts of its value, the method has to be changed if the domain of this instance variable is changed due to optimization purposes. The key idea for improving this kind of stamp coupling is to distinguish between methods which directly access instance variables i.e. internal data structure of a class, and methods which do not. It is good design to hide the internal data structure whenever possible – not only from the outside of an object class but also from methods inside of the object class. Therefore, it is always good to design read methods and write methods, called access methods, for each instance variable and use these methods as only means to access these variables. If the internal data structure of an object class is changed only the access methods have to be updated.

The idea of restricting the access to instance variables via explicit access methods is not new. There exist design rules in software reuse leading to a factoring out of methods from some object class if they do not directly access instance variables of that class. This might increase, however, interaction coupling between classes. Thus there is some trade-off between various design goals and the designer has to decide which goal to prefer on a case by case basis. Considering inheritance of methods and instance variables into account, stamp coupling between a method and inherited instance variables is called inherited stamp coupling. It is worse than stamp coupling between a method and instance variables defined within the same class. This is due to the commonly accepted understanding that directly accessing
inherited instance variables in subclasses breaks encapsulation and information hiding.

6. **Data**

Two methods are data coupled if they communicate only by parameters and these parameters are relevant as a whole. Data coupling is the best form of coupling whenever two methods have to interact. Data coupled methods minimize maintenance effort due to a great restriction of change propagations.

7. **No direct coupling**

The theoretical optimum of interaction coupling is no direct coupling. Two methods do not directly depend on each other, and thus also their object classes are not interaction coupled. A change in one method does not directly demand a change in the other method, and hence no change in that method’s object class is necessary.

3.2.4 **Component Coupling**

As opposed to interaction coupling where object classes and methods are involved component coupling concerns only object classes. The component relationship between classes is defined by the use of a class as domain of some instance variable of another class. In the context of component coupling the object class \( C' \) is considered to be a component of the object class \( C \) if and only if \( C' \) appears in \( C \). \( C' \) appears in \( C \) if and only if:

1. \( C' \) is the domain of an instance variable of \( C \), or
2. \( C' \) is the domain of an input or output parameter of a method of \( C \), or
3. \( C \) is the domain of a local variable of some method of \( C \), or

4. \( C \) is the domain of input or output parameter of some method invoked within a method of \( C \)

Whereas component coupling reveals the coupling from a class \( C \) to a class \( C' \) during compile time it might happen during run-time that \( C \) is component coupled with any subclass of \( C \). This is known as potential coupling. If \( C \) is component coupled with \( C' \) then \( C \) is potentially component coupled with all subclasses of \( C \). Component coupling usually implies interaction coupling. In interaction coupling, the focus should be on how much information is exchanged between methods and classes and on how complex such information is. With component coupling we shall analyze how explicit the coupling between classes.

The first case of component relationship is realized via instance variables. It is made explicit in OO languages at the class level, but only in the implementation part and not in the specification part of the class definition. Coupling of the second case is made explicit in the specification part of a class definition by specifying the signature of the methods. Component coupling of the third case is based on local variables. It is only explicit within a method through the declaration of local variables but it is not explicit at the class level. The fourth case is even worse. In cascading messages, the object returned by a method is used immediately as receiver of another message. The object class of this receiver might not even be declared anywhere in the actual class. Based on these, the degrees of component coupling shall be explained:
1. Hidden

The coupling between two classes $C$ and $C'$ is rated hidden if $C'$ shows up neither in the specification nor in the implementation of $C$, although an object of $C'$ is used in the implementation of a method of $C$. As an example, let us consider the cascading message problem. A similar problem is encountered if the return value of a method invocation is immediately used as input parameter in another method invocation. Most languages do not require that the class of this object is declared anywhere within the actual class.

Hidden coupling causes problems since this coupling between classes is implicit. Hidden coupling can be compared with the use of global variables in procedure-oriented systems, which is responsible for common coupling between modules. Consider a change of a class in a maintenance process, i.e. the change of the signature of a method. In the presence of hidden coupling the programmer has to search through all implementations of all methods of all classes to detect where this change may have influence, and where this change has to be propagated to, respectively. A possibility to avoid hidden coupling is to disallow the use of cascading messages and to disallow the use of return values as parameters if their domains are not declared. A less restrictive way to overcome hidden coupling is to declare all those classes in the specification part of the actual class definition.

2. Scattered

Two classes $C$ and $C'$ as scattered coupled, if $C'$ is used as domain in the definition of some local variable or instance variable in the implementation of $C$ yet $C'$ is not included in the specification of $C$. To detect whether $C$ and $C'$ are component coupled it is necessary to check the implementation of classes to get the domains of instance variables, and even worse to check the implementation of all methods to detect the domains of
local variables. If a class is changed the implementations of all other classes have to be checked in order to discover which classes may be influenced.

3. Specified

Two classes $C$ and $C'$ are specified coupled if $C'$ is included in the specification of $C$ whenever it is a component of $C$. Specified coupling overcomes the problems of hidden and scattered coupling by specifying all related component classes of some class in a single place. Thus it is possible to determine whether two classes are coupled without browsing through the whole implementation. Browsing the implementation might be even impossible if the source code is not available.

In most OO languages only the signatures of the methods provided by some class $C$ are shown in the specification of $C$. This set of methods provided by $C$ is also called suffered interface of $C$. Those classes which are used as domains of input parameters and return values of the methods of $C$ are the only ones which are specified coupled with $C$. In addition to the suffered interface, the required interface also becomes part of the specification of a class. The required interface of some class $C$ compromises all classes which are used as components of $C$.

4. Nil

The theoretical optimum is no direct component coupling between classes and thus no interaction coupling. It is an advantage to recognize that two classes are completely independent such that one class can be maintained without any knowledge of the other class.
3.2.5 Inheritance Coupling

Similar to component coupling, inheritance coupling only concerns object classes. Two classes are inheritance coupled if one class is a direct or indirect subclass of the other. Inheritance is one of the most important features of OO methods and languages. It supports re-use both through subclassing i.e. specialization, and through factoring out i.e. generalization, of common information from independent classes into a common super class. At a first glance it seems contradictory to use inheritance for gaining better reusability and to have the goal of low coupling.

The key idea to resolve this seeming contradiction is twofold: Firstly, inheritance may be used to lower coupling in an OO system through factoring out. Given a class D which invokes the same method m on objects of class C and C', D is component coupled with C and C'. If the method m is factored out into a common superclass C of C and C' and not overridden in C' and C', respectively, D is component coupled with C only. Thus the coupling is improved since the number of classes with which D is coupled has been reduced. Secondly, there exist different degrees of inheritance coupling. The lowest degree of inheritance coupling coincides with better reusability. Furthermore, considering inheritance coupling is necessary for improving the overall quality of the system to be implemented.

Inheritance coupling is also different from interaction coupling and component coupling in that it does not only exhibit the coupling property between sub classes and super classes but implicitly also the coupling property between an interaction coupled object class and the inheritance hierarchy. In other words, if class D is interaction coupled to some class C being the root of an inheritance hierarchy and the inheritance hierarchy is changed, i.e. subclasses are added, and inherited instance variables and
methods are modified, the degree of inheritance coupling reveals to which extent changes in the inheritance hierarchy might impose changes in D.

The following is the discussion on various degrees of inheritance coupling from worst to best degree. The assumption is for simplicity and without loss of generality that access methods for each instance variable exist. Thus any change to an instance variable in some subclass is reflected by the corresponding change of the signature and/or implementation of the access methods.