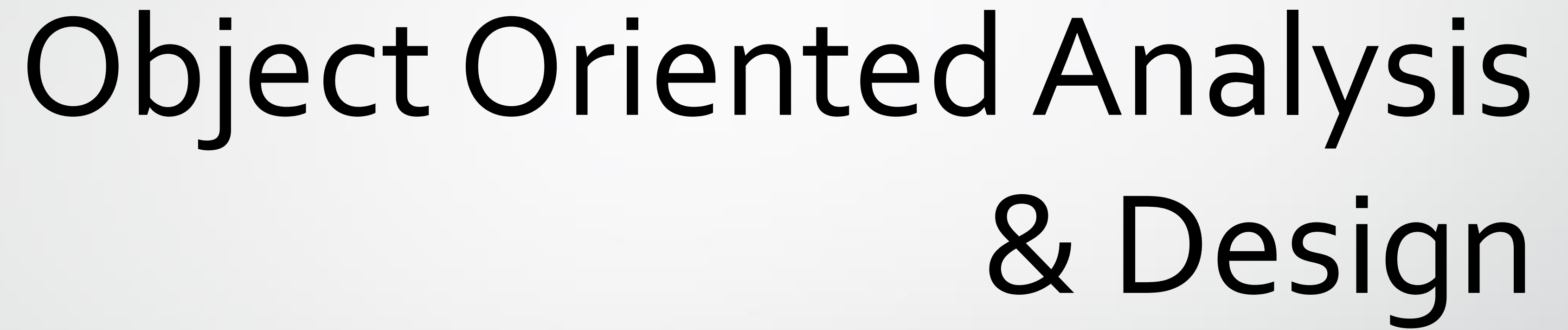


Week 11

Reuse, Reusability and Portability

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 10

- In UML 2 packages are logical groupings and thus the elements of a package are semantically related; they are not physical.
- Packages may contain use cases, analysis cases, or use case realizations.
- Generalization can be applied in packages in the same way as the class diagram.
- The principle behind interfaces is to separate the specification of functionality from its implementation by a classifier such as a class or subsystem.
- Provided interfaces refers to the set of interfaces realized by a classifier; required interfaces refers to the interfaces needed by a classifier to realize its operations
- A component is a logical part of a system usually larger than a single class, which can (in principle) be replaced, reused, or sold separately.
- The deployment diagram shows how the software will be deployed in the system, and how that hardware is connected.
- There are two forms of deployment diagrams – descriptor form and instance form.

Content

- The Design challenge
- Reuse and reusability
- Portability

The Design Challenge

Introduction

- So far we have learnt about the SDLC and its phases.
- We have also learnt in the last 10 weeks the challenges involved in the pivotal analysis and design phases.
- OOAD is focused on these phases.
- In this lesson we examine the issues of reuse and reusability. We can already guess from what we have learnt from first principles where reuse can be implemented in the SDLC.
- Reuse is first introduced in the design phase using OO principles before getting to the coding phase where we can take chunks of code and reuse them in different parts of the final solution.
- These chunks of code exist in the respective modules where they were first written.
- It is thus impossible to talk about reuse without revisiting the modularity issue.
- In this section we revisit modularity in the context of two strong terms used: cohesion and coupling.

1.1 Modularity

- In week one we described modularity as breaking down a complex system into smaller units that are more manageable. These smaller units are usually built around functionality and are fully functional on their own; thus they can be managed independently.
- This concept was explored further using the concept of monolithic vs modular programming. By breaking down a program into smaller parts (modules) it makes it that much easier to understand, and the modules can each be developed independently.
- Ashrafi, N and Ashrafi, H (2013) show the relationship between modularity and reuse by describing it as follows: "the concept that an application can be constructed from a set of modules, conceived around a certain functionality—database module, calculation module, reporting module, etc., so that the code would become more reusable and reliable, teams of programmers could work on different parts of the application, and redundancy would decrease."
- The whole idea of modularity is a well thought out idea especially from a reuse perspective; however, modularity also goes hand in hand with two other concepts that need discussion: coupling and cohesion.

1.2 Cohesion

- Cohesion is a measure of the degree to which elements of a module are related. In OO context cohesion is examined in the context of classes, objects and methods (these are the individual parts of the module).
- Cohesion is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion. (geeksforgeeks.org)
- Essentially this means that all these individual elements should perform singular tasks; for example a class (or object for that matter) should do only one thing, and a method should solve only one problem.
- Coad and Yourdon (as cited by Dennis et al., 2015) identified three general types of cohesion for OO systems: method, class and generalization/specialization.

1.2.1 Method cohesion

- This is a measure of how single minded a method is. It is desirable for a method to perform one and only one task.
- If a method performs more than one task it is that much harder to understand it (and by extension to implement and maintain).
- It follows therefore that the higher method cohesion is, the better; it should thus be maximized.
- There are seven types of method cohesion:
- Informational cohesion - A module has informational cohesion if it performs a number of operations, each with its own entry point, with independent code for each operation, all performed on the same data structure. Fig 1 provides an example of a module with informational cohesion; each piece of code has exactly one entry point and exactly one exit point. A module with informational cohesion is an example of separation of concerns. (Schach, 2017).

1.2.1 Method cohesion

- Functional cohesion – this is the best form of method cohesion. It means that the method performs one and only one task; for example it calculates the area of a circle.
- Sequential cohesion – this one combines two functions such that the output of one is the input of the other; for example, a method that calculates the area of a rectangle in square meters and converts it to hectares.
- Communicational cohesion – two functions use the same attribute to execute; for example compute current mean and cumulative mean for a student.
- Procedural cohesion - The method supports multiple weakly related functions; for example, the method could calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.
- Temporal (classic) cohesion - The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time (geeksforgeeks.com); for example, the initialiazation of all attributes.

1.2.1 Method cohesion (cont'd)

- Logical cohesion - The method supports multiple related functions, but the choice of the specific function is chosen based on a control variable that is passed into the method. For example, the called method could open a checking account, open a savings account, or calculate a loan, depending on the message that is sent by its calling method. (Dennis et al., 2015)
- Coincidental cohesion – this is the weakest (worst) form of method cohesion. The purpose of the method can't be defined; or it performs multiple unrelated functions; for example, calculate area of a circle, calculate the mean grade of a student, and print out the members of a faculty department.
- Notice the use of function, method, and module in defining the different types of method cohesion? It is not coincidence. In general terms a module can be a method or a function; the key word is independence, meaning it completes one task wholly.
- Fig 2 shows the different types of method cohesion, from best to worst, while fig 3 presents an example of different types of method cohesion.

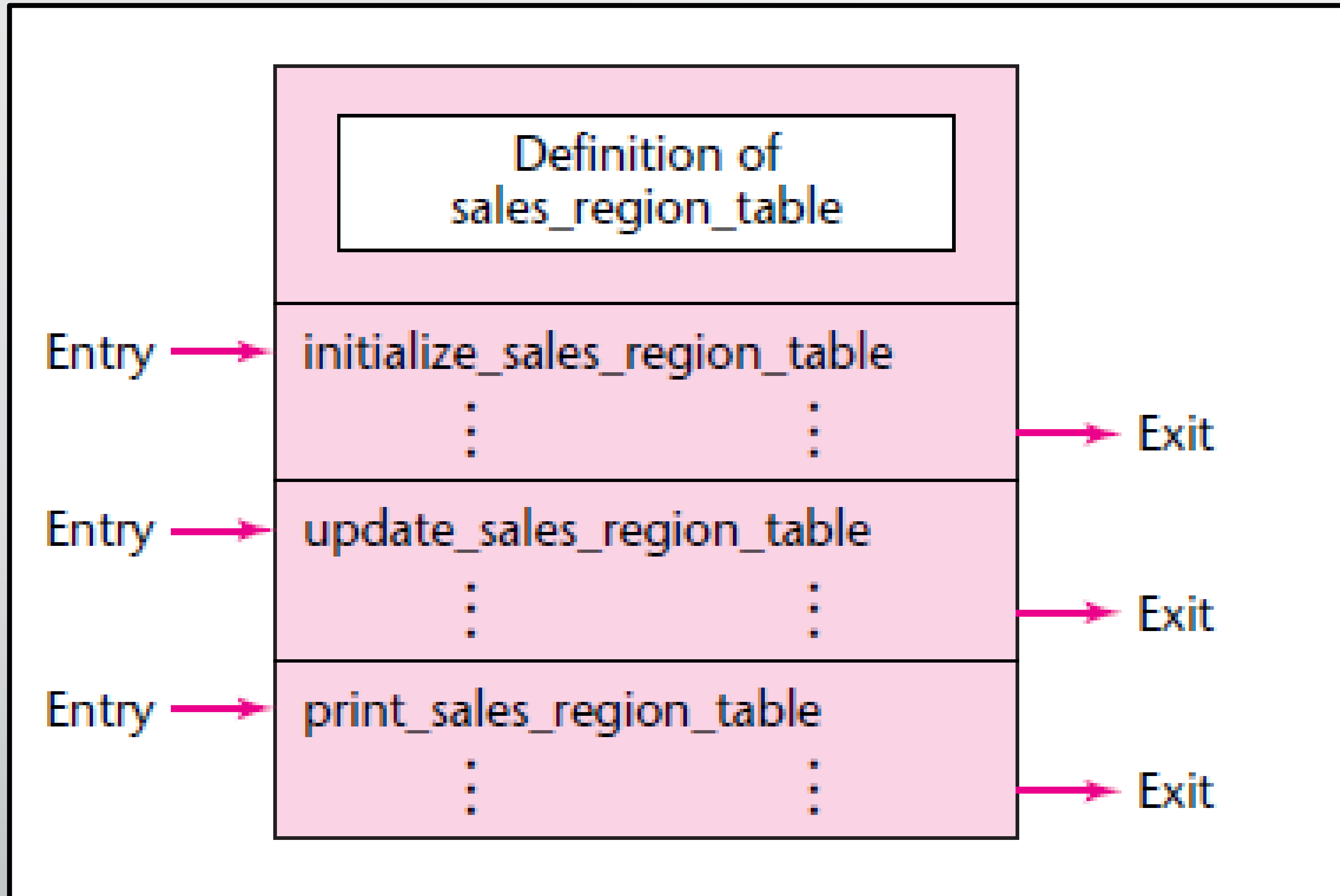


Fig 1. Informational cohesion (Schach, 2017)

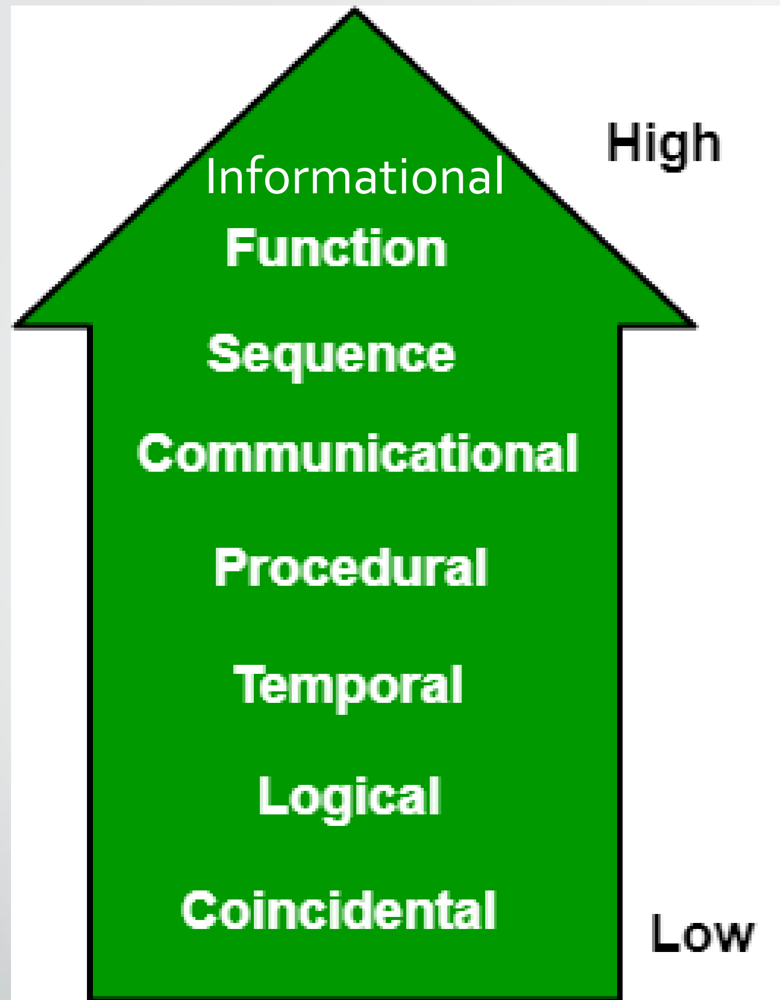


Fig 2. Method cohesion levels (geeksforgeeks.org)

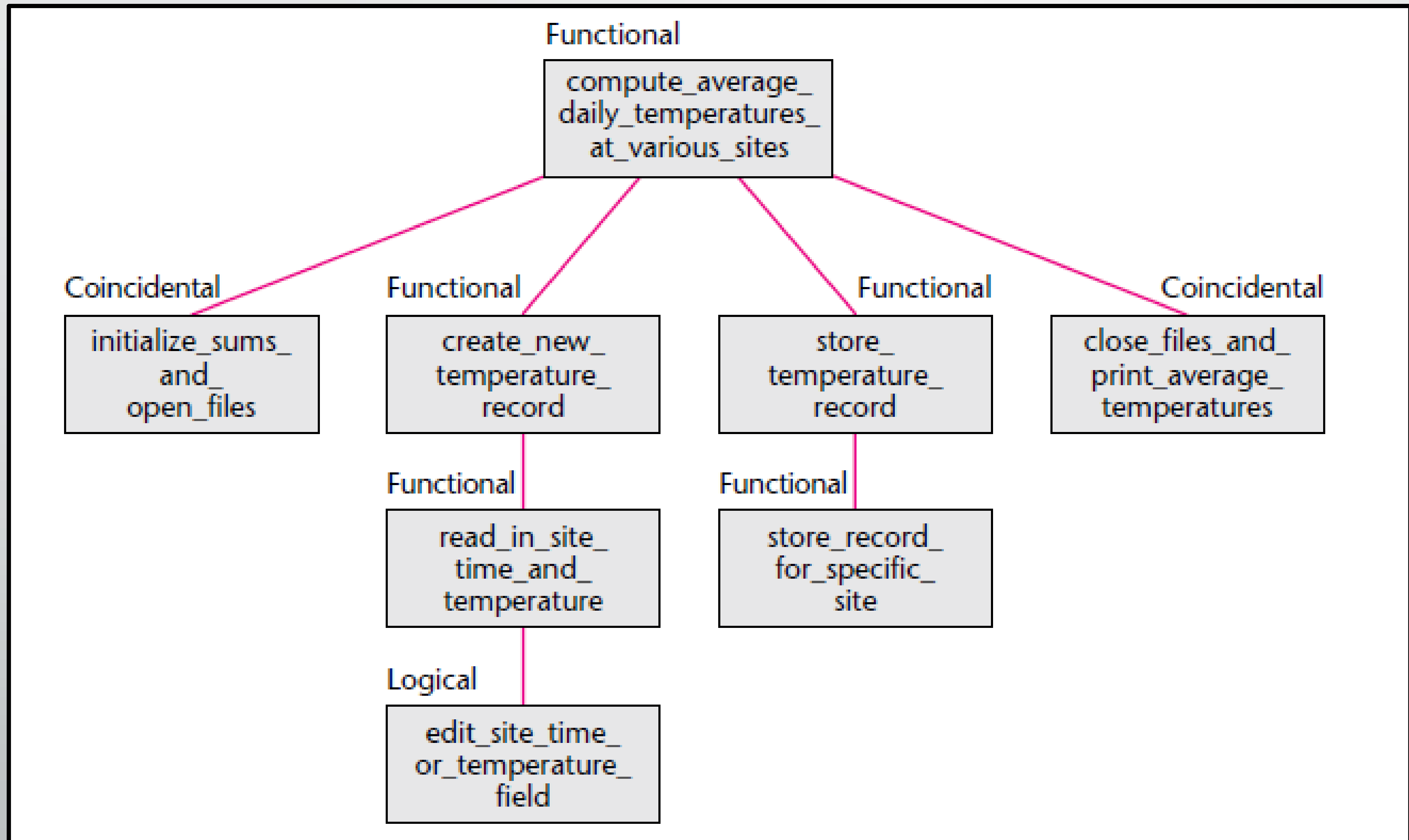


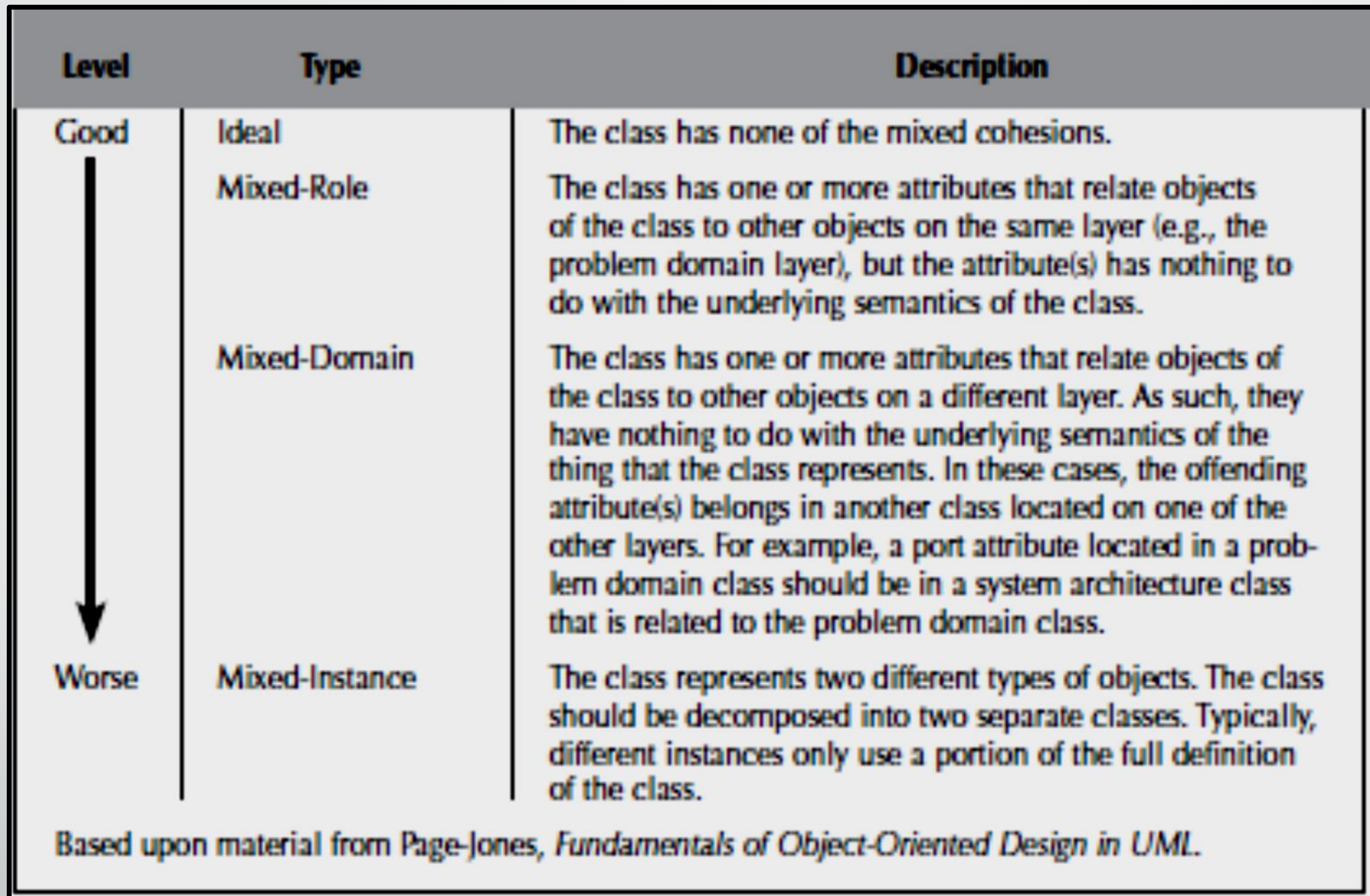
Fig 3. Cohesion example (Schach, 2017)

1.2.2 Class cohesion

- This refers to the level of cohesion between attributes and methods of the class.
- The class should only represent one distinct thing; further, its methods and attributes should only be the ones that are required for it to do what is required in the problem domain.
- This means that the class should only have the requisite methods, and only attributes that are relevant for its instances to be used effectively; further, all attributes and methods of the class must be used in the module.
- Consequently if only the attributes and methods needed by the class are defined it means the class has what it needs to instantiate instances relevant to the problem domain; a situation referred to as ideal cohesion.
- If a method or attribute is not used then it makes the class that much less cohesive.

1.2.2 Class cohesion (cont'd)

- Meyers (as cited by Dennis et al., 2015) suggested that a cohesive class should have the following attributes:
- It should contain multiple methods that are visible outside the class (i.e., a
- single-method class rarely makes sense).
- Each visible method performs only a single function (i.e., it has functional cohesion)
- All methods reference only attributes or other methods defined within the class or one of its superclasses (i.e., if a method is going to send a message to another object, the remote object must be the value of one of the local object's attributes).
- It should not have any control couplings between its visible methods.
- Three types of class cohesion were identified by Page-Jones (as cited by Dennis et al., 2015); these are summarized in fig 4.



Level	Type	Description
Good	Ideal	The class has none of the mixed cohesions.
	Mixed-Role	The class has one or more attributes that relate objects of the class to other objects on the same layer (e.g., the problem domain layer), but the attribute(s) has nothing to do with the underlying semantics of the class.
	Mixed-Domain	The class has one or more attributes that relate objects of the class to other objects on a different layer. As such, they have nothing to do with the underlying semantics of the thing that the class represents. In these cases, the offending attribute(s) belongs in another class located on one of the other layers. For example, a port attribute located in a problem domain class should be in a system architecture class that is related to the problem domain class.
Worse	Mixed-Instance	The class represents two different types of objects. The class should be decomposed into two separate classes. Typically, different instances only use a portion of the full definition of the class.

Based upon material from Page-Jones, *Fundamentals of Object-Oriented Design in UML.*

Fig 4. Class cohesion types (Dennis et al., 2015)

1.2.3 Generalization/specialization cohesion

- Inheritance is one of the OO principles where reuse can be implemented very effectively. Recall that child classes inherit all the attributes of the parent classes?
- This means that modules written for parent classes are automatically available to the child classes using the inheritance principle. Further child classes can add their own attributes and create more methods, or even override the methods of their parents. Cool, right?
- *Generalization/specialization cohesion* addresses the sensibility of the inheritance hierarchy. How are the classes in the inheritance hierarchy related? Are the classes related through a generalization/specialization (a-kind-of) semantics? Or, are they related via some association, aggregation, or membership type of relationship that was created for simple reuse purposes? (Dennis et al, 2015)
- Highly cohesive inheritance hierarchies should support only the semantics of generalization and specialization (a-kind-of) and the principle of substitutability. (Dennis et al, 2015)
- The principle of substitutability also known as Liskov substitutability principle states that "objects of a superclass should be replaceable with objects of its subclasses without breaking the application. In other words, what we want is to have the objects of our subclasses behaving the same way as the objects of our superclass."

1.2.3 Generalization/specialization cohesion

- Not all cases are ideal for reuse purposes. Consider fig 5:
- The subclasses ClassRooms and Staff inherit from the superclass Department. Obviously, instances of the ClassRooms and Staff classes are not a-kind-of Department. However, in the early days of object-oriented programming, this use of inheritance was quite common. When a programmer saw that there were some common properties that a set of classes shared, the programmer would create an artificial abstraction that defined the commonalities. This was potentially useful in a reuse sense, but it turned out to cause many maintenance nightmares. In this case, instances of the ClassRooms and Staff classes are associated with or a-part-of an instance of Department. (Dennis et al., 2015). This is not an ideal setup

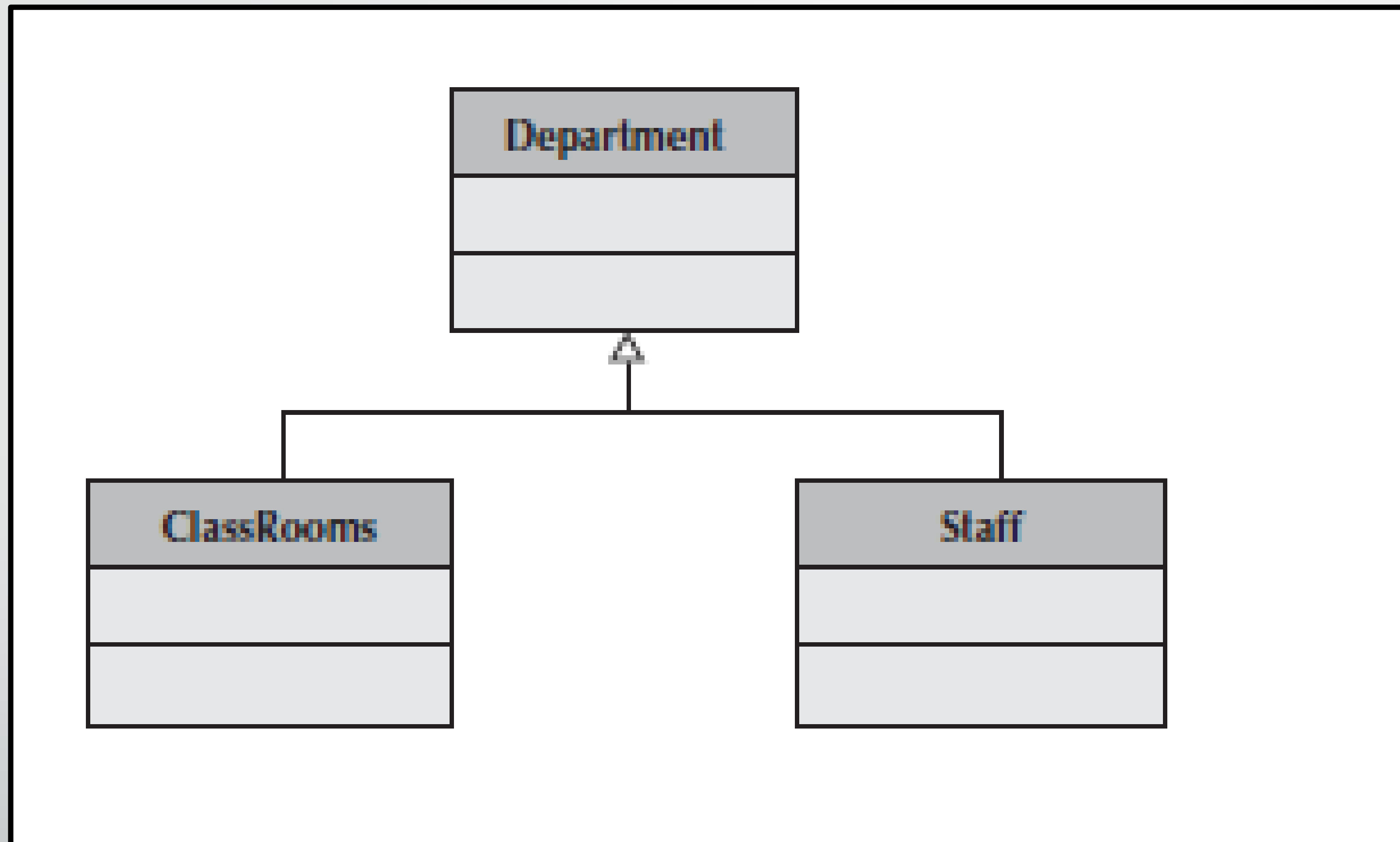


Fig 5. Generalization vs inheritance abuse (Dennis et al, 2015)

1.3 Coupling

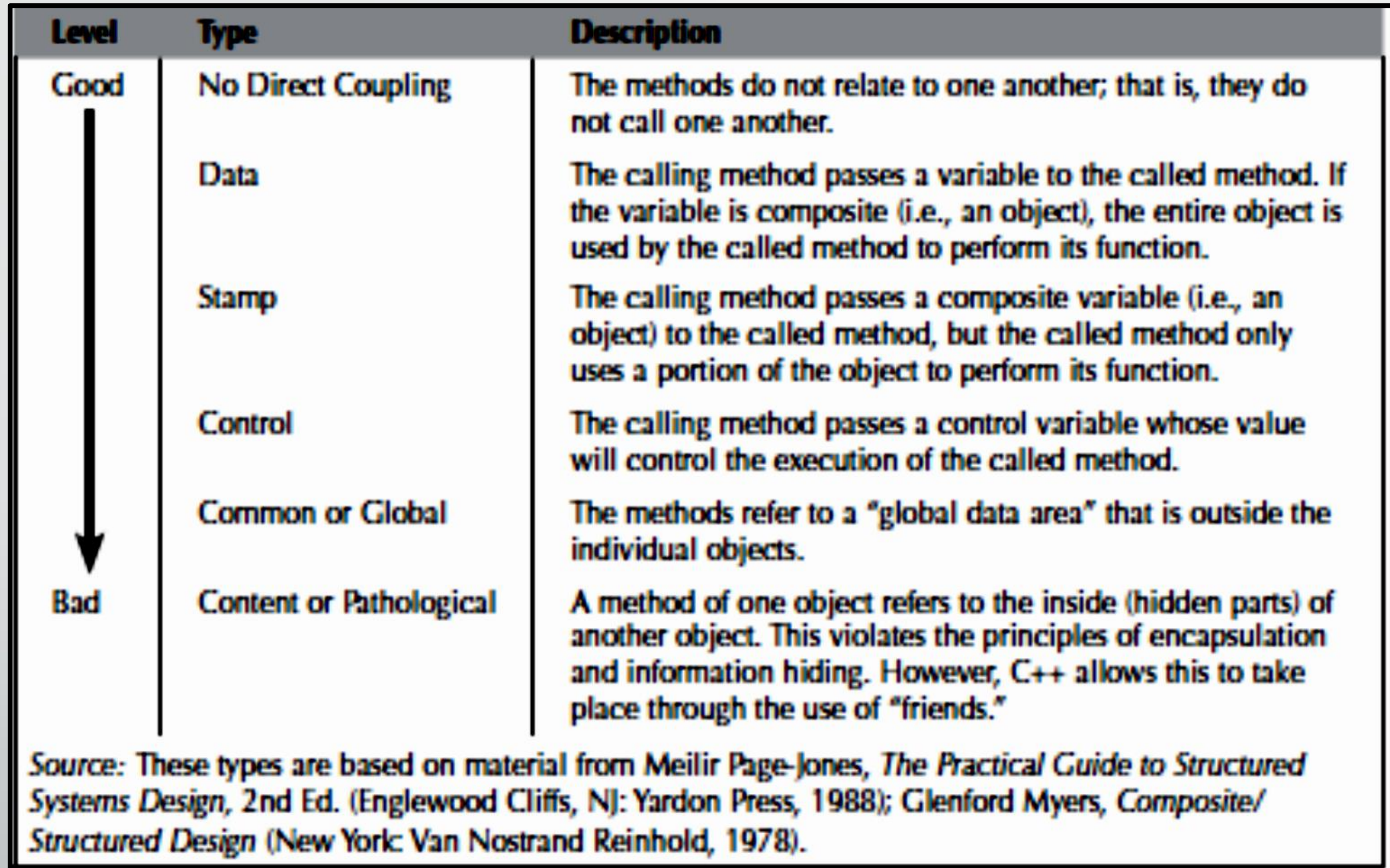
- Coupling simply refers to the degree of interdependence or interrelation of modules in a system.
- If the modules are too interdependent it presents a problem; it means that changes in one part of the system will result in changes in other parts of the system (the highly interdependent ones). Consequently, therefore, little or no coupling is desirable. However, is this always possible?
- Coad and Yourdon (as cited by Dennis et al, 2015) identified two types of coupling: inheritance and interaction coupling.

1.3.1 Interaction coupling

- Interaction coupling deals with coupling among methods and objects through message passing (the way in which it is done in OO).
- The law of Demeter provides guidelines to minimize interaction coupling. It states that an object should only send messages to one of the following:
  - Itself
  - An object that is contained in an attribute of the object or one of its superclass
  - An object that is passed as a parameter to the method
  - An object that is created by the method
  - An object that is stored in a global variable

1.3.1 Interaction coupling (cont'd)

- However, Dennis et al. (2015) still observed shortcomings of the law of Demeter in increasing interaction coupling. They observed as follows:
- "Even though the law of Demeter attempts to minimize interaction coupling among methods and objects, each of the above allowed forms of message sending in fact increases coupling. For example, the coupling increases between the objects if the calling method passes attributes to the called method or if the calling method depends on the value being returned by the called method."
- There are six types of interaction coupling, varying from good to bad. These are summarized in fig 6



Level	Type	Description
Good	No Direct Coupling	The methods do not relate to one another; that is, they do not call one another.
	Data	The calling method passes a variable to the called method. If the variable is composite (i.e., an object), the entire object is used by the called method to perform its function.
	Stamp	The calling method passes a composite variable (i.e., an object) to the called method, but the called method only uses a portion of the object to perform its function.
	Control	The calling method passes a control variable whose value will control the execution of the called method.
	Common or Global	The methods refer to a "global data area" that is outside the individual objects.
Bad	Content or Pathological	A method of one object refers to the inside (hidden parts) of another object. This violates the principles of encapsulation and information hiding. However, C++ allows this to take place through the use of "friends."

Source: These types are based on material from Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, 2nd Ed. (Englewood Cliffs, NJ: Yardon Press, 1988); Glenford Myers, *Composite/Structured Design* (New York: Van Nostrand Reinhold, 1978).

Fig 6. Interaction coupling types (Dennis et al, 2015)

1.3.2 Inheritance coupling

- This type of inheritance deals with how tight coupling is between classes in a hierarchy.
- The literature states that this type of coupling is desirable.
- The coupling is through data members that are inherited from a parent class but **not** re-defined by its subclass. (cs.gmu.edu)
- Dennis et al. (2015) observe the following regarding inheritance coupling:
- "most problems with inheritance involve the ability within the object-oriented programming languages to violate the encapsulation and information-hiding principles. From a design perspective, the developer needs to optimize the trade-off s of violating the encapsulation and information-hiding principles and increasing the desirable coupling between subclasses and its superclasses. The best way to solve this conundrum is to ensure that inheritance is used only to support generalization/specialization (a-kind-of) semantics and the principle of substitutability. All other uses should be avoided."

1.4 Connascence

- Connascence generalizes the ideas of cohesion and coupling, and it combines them with the arguments for encapsulation.
- Connascence literally means to be born together. From an object-oriented design perspective, it really means that two modules (classes or methods) are so intertwined that if you make a change in one, it is likely that a change in the other will be required. On the surface, this is very similar to coupling and, as such, should be minimized. (Dennis et al, 2015)
- Generally speaking cohesion (connascence) should be maximized within an encapsulation boundary while coupling (connascence) should be minimized between encapsulation boundaries.
- There are five types of connascence summarized in fig 7.

Type	Description
Name	If a method refers to an attribute, it is tied to the name of the attribute. If the attribute's name changes, the content of the method will have to change.
Type or Class	If a class has an attribute of type A, it is tied to the type of the attribute. If the type of the attribute changes, the attribute declaration will have to change.
Convention	A class has an attribute in which a range of values has a semantic meaning (e.g., account numbers whose values range from 1000 to 1999 are assets). If the range would change, then every method that used the attribute would have to be modified.
Algorithm	Two different methods of a class are dependent on the same algorithm to execute correctly (e.g., insert an element into an array and find an element in the same array). If the underlying algorithm would change, then the insert and find methods would also have to change.
Position	The order of the code in a method or the order of the arguments to a method is critical for the method to execute correctly. If either is wrong, then the method will, at least, not function correctly.

Based upon material from Meilir Page-Jones, "Comparing Techniques by Means of Encapsulation and Connascence" and Meilir Page-Jones, *Fundamentals of Object-Oriented Design in UML*.

Fig 7. Connascence types (Dennis et al, 2015)

Reuse and reusability

2.1 Concepts

- A software product is said to be portable if it can be used on different hardware/operating systems without having to modify it in any way, whatsoever.
- Reuse on the other hand refers to using components of one product to develop a different product with different functionality.
- There are two types of reuse:
  - Opportunistic reuse – this is also known as accidental reuse. As the name implies this is where developers accidentally realize that a component can be used in a different product. Essentially the component wasn't developed with reuse in mind, but it so happens that it can be reused.
  - Systematic reuse – this is also known as deliberate reuse. This is when a component is intentionally designed for reuse in future products.

2.2 Impediments to reuse

- Schach (2017) describes the following impediments to reuse:
- All too many software professionals would rather rewrite a routine from scratch than reuse a routine implemented by someone else, the implication being that a routine cannot be any good unless they implemented it themselves, otherwise known as the not invented here (NIH) syndrome.
- Many developers would be willing to reuse a routine provided they could be sure that the routine in question would not introduce faults into the product.
- A large organization may have hundreds of thousands of potentially useful components. How should these components be stored for effective later retrieval?

2.2 Impediments to reuse (cont'd)

- Reuse can be expensive. Three costs are involved: the cost of making something reusable, the cost of reusing it, and the cost of defining and implementing a reuse process.
- Legal issues can arise with contract software. In terms of the type of contract usually drawn up between a client and a software development organization, the software product belongs to the client. Therefore, if the software developer reuses a component of one client's product in a new product for a different client, this essentially constitutes a violation of the first client's copyright. This is not a problem, however, for internal software.
- Another impediment arises when commercial off-the-shelf (COTS) components are reused. Rarely are developers given the source code of a COTS component, so software that reuses COTS components has limited extensibility and modifiability.

2.3 Objects and reuse

- Initially it was presumed that the best module has functional cohesion; and consequently, best for reuse.
- The flaw in this presumption is explained by Schach (2017) as follows: "a module with functional cohesion is not self-contained and independent. Instead, it has to operate on data. If such a module is reused, then the data on which it is to operate must be reused, too. If the data in the new product are not identical to those in the original, then either the data have to be changed or the module with functional cohesion has to be changed. Therefore, contrary to what we used to believe, functional cohesion is not ideal for reuse."
- Schach (2017) went further to argue that when OO principles are properly applied and utilized (encapsulation/information hiding), the resulting modules will have informational cohesion, and this is the best for reuse.

2.4 Design reuse

- Different approaches to reuse can be used in the design stage; some of these approaches are even carried into the implementation stage. Some of these approaches are briefly described:
- Design reuse – this happens when a designer realizes that a class used in a different project, can be reused in the current project with or without some minor modifications. This is common with companies that develop software in specific domains, such as higher education, financial institutions, and so on. There are two advantages to this approach:
- The reused designs have already been tested, making the overall design process shorter (saving time)
- If the design of the class can be reused then it is quite likely that the implementation of the same class can also be reused, or at least most of it.

2.4 Design reuse (cont'd)

- Application frameworks – an application framework incorporates the control logic of a design. When a framework is reused, the developers have to design the application-specific operations of the product being built. The places where the application-specific operations are inserted frequently are referred to as hot spots.
- Design patterns - A design pattern is a solution to a general design problem in the form of a set of interacting classes that have to be customized to create a specific design.
- Software architecture - The idea is to develop a software architecture common to a number of software products and instantiate this architecture when developing a new product. Architecture patterns are another way of achieving architectural reuse.
- Component based software engineering – constructing a standard collection of reusable components.

2.5 Categories of design patterns

- Design patterns are divided into three categories:
- Creational design patterns – these solve design problems by creating objects.
- Structural design patterns – these solve design problems by exploring (realizing) relationships between entities in the problem domain.
- Behavioral design patterns – these solve design problems by identifying the common communication problems between objects.
- Fig 8 shows the list of design patterns with their definitions. Relevant sections from the text are provided for more explanation.

Pattern	Definition
Creational patterns	
Abstract factory	Creates an instance of several families of classes (Section 8.6.5)
Builder	Allows the same construction process to create different representations
Factory method	Creates an instance of several possible derived classes
Prototype	A class to be cloned
Singleton	Restricts instantiation of a class to a single instance
Structural patterns	
Adapter	Matches interfaces of different classes (Section 8.6.2)
Bridge	Decouples an abstraction from its implementation (Section 8.6.3)
Composite	A class that is a composition of similar classes
Decorator	Allows additional behavior to be dynamically added to a class
Façade	A single class that provides a simplified interface
Flyweight	Uses sharing to support large numbers of fine-grained classes efficiently
Proxy	A class functioning as an interface
Behavioral patterns	
Chain-of-responsibility	A way of processing a request by a chain of classes
Command	Encapsulates an action within a class
Interpreter	A way to implement specialized language elements
Iterator	Sequentially access the elements of a collection (Section 8.6.4)
Mediator	Provides a unified interface to a set of interfaces
Memento	Captures and restores an object's internal state
Observer	Allows the observation of the state of an object at run time
State	Allows an object to partially change its type at run time
Strategy	Allows an algorithm to be dynamically selected at run time
Template method	Defers implementations of an algorithm to its subclasses
Visitor	Adds new operations to a class without changing it

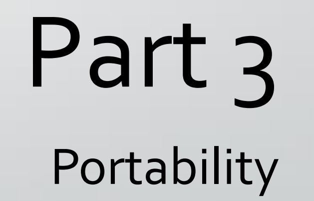
Fig 8. Design pattern definitions (Schach, 2017)

2.6 Strengths of design patterns

- The reusability of a design pattern can be enhanced further by incorporating OO principles such as inheritance.
- They provide high level documentation of the design since they specify design abstrations.
- There are very many implementations of design patterns in existence; therefore, no need to repeat coding or documentation (maybe to customize only)
- In the maintenance phase, implementers who are familiar with design patterns have an easier time since the programs are easier to understand even without specific knowledge of the particular program.
- There is a lot of ongoing research into automation of design patterns and looks promising for the future.

2.7 Weaknesses of design patterns

- The use of 23 standard design patterns may indicate a weakness in the choice of programming language. For example C++ is found to be weaker than Lisp or Dylan in the use of some of the patterns.
- There is still no systematic way to determine when and how to apply design patterns.
- To get maximum benefit from the design patterns there is a need to employ several interacting patterns.
- Design patterns present challenges in the maintenance phase especially when trying to retrofit classes and objects from products built using non – OO methods.


Part 3
Portability

3.1 Portability incompatibilities

- Schach (2017) defines portability as follows: "Suppose a product *P* is compiled by compiler *C* and then runs on the **source computer** , namely, hardware configuration *H* under operating system *O* . A product *P′* is needed that functionally is equivalent to *P* but must be compiled by compiler *C* and run on the **target computer** , namely, hardware configuration *H* under operating system *O* . If the cost of converting *P* into *P′* is significantly less than the cost of coding *P* from scratch, then *P* is said to be *portable* ."
- The key problems involving portability revolve around different compilers, operating systems, and hardware configurations.

3.2 Portability techniques

- Schach (2017) suggests the following techniques:
- Portable system software - Instead of forbidding all implementation-dependent aspects, which would prevent almost all system software from being implemented, a better technique is to isolate any necessary implementation-dependent pieces.
- Portable application software - With regard to application software, rather than system software such as operating systems and compilers, it generally is possible to implement the product in a high-level language. At every stage in the development of a product, decisions can be made that result in a more portable product.
- Portable data - The safest way of porting data is to construct an unstructured (sequential) file, which can then be ported with minimal difficulty to the target machine. From this unstructured file, the desired structured file can be reconstructed.
- Model driven architecture (MDA) - is an emerging technology that achieves portability by entirely decoupling the functionality of a software product from its implementation. MDA solves the problem of moving a software product to a new platform at the analysis level rather than at the design level. There are three models in MDA: computation independent model (CIM), platform independent model (PIM), and platform specific model (PSM).

Summary

- The OO principle of modularity makes code more reusable as functionality is divided across different methods.
- Cohesion is a measure of the degree to which elements of a module are related. In OO context cohesion is examined in the context of classes, objects and methods (these are the individual parts of the module).
- A good software design will have high cohesion
- Coupling simply refers to the degree of interdependence or interrelation of modules in a system; consequently, little or no coupling is desirable.
- Connascence literally means to be born together. From an object-oriented design perspective, it really means that two modules (classes or methods) are so intertwined that if you make a change in one, it is likely that a change in the other will be required. Consequently, connascence should be minimized.
- Reuse refers to using components of one product to develop a different product with different functionality. There are two types of reuse – opportunistic and systematic.
- Design patterns are divided into three categories: creational, structural and behavioral.
- The key problems involving portability revolve around different compilers, operating systems, and hardware configurations.

References

- Ashrafi, Noushin, and Hessam Ashrafi. *Object Oriented Systems Analysis and Design*. 1st ed., Pearson, 2014.
- Dennis, A., Wixom, B. H., Tegarden, D. P., & Seeman, E. (2015). *System analysis & design: An object-oriented approach with Uml*. Wiley.
- GeeksForGeeks. (2018, July 30). *Software Engineering | Coupling and Cohesion - GeeksforGeeks*. GeeksforGeeks. https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/
- *Inheritance Coupling Help*. (2022). Gmu.edu. https://cs.gmu.edu/~offutt/coupdemo/help/inherCoup.html#:~:text=Inheritance%20coupling%20refers%20to%20the
- Schach, S. R. (2017). *Object-oriented and classical software engineering*. Langara College.