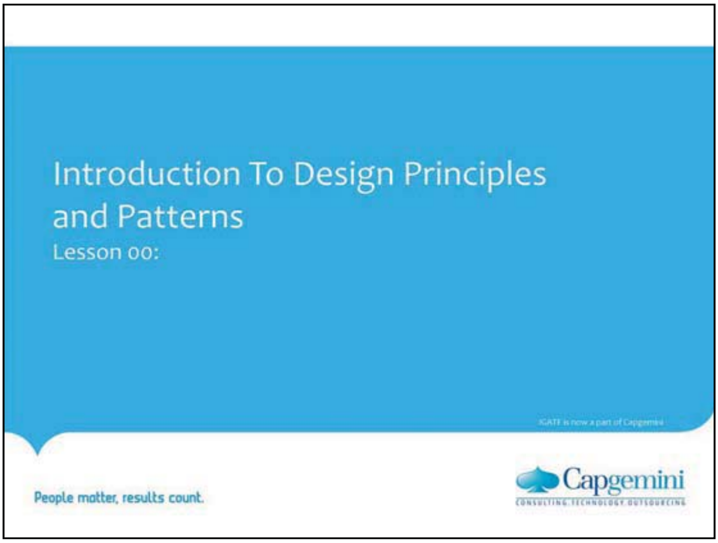

Introduction To Design Principles
and Patterns
Lesson 00:
People matter, results count.
Capgemini

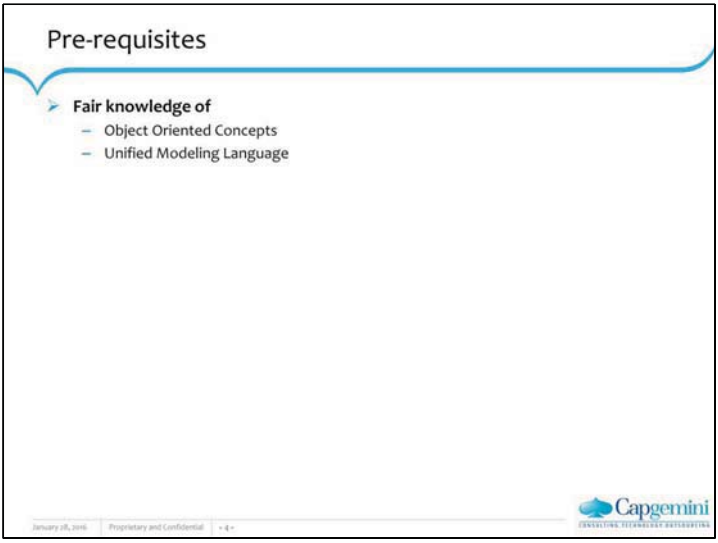
Document History

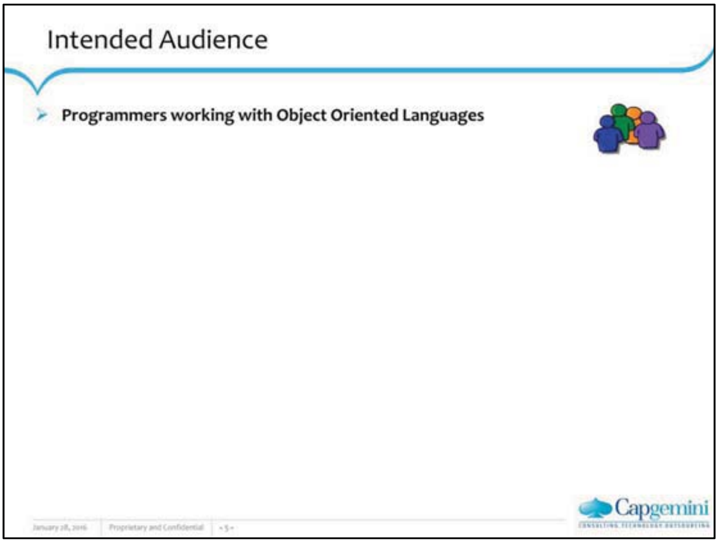
Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
15-Feb-2011	0.01D	NA	Veena Deshpande	Content Compilation
26 Jul 2011	1.0	NA	Veena Deshpande Kishori Khadilkar	Baselining after content and format Review

Course Goals and Non Goals

- **Course Goals**
  - At the end of this course, participants gain an understanding of
    - Key Design Principles
    - Overview of Design Patterns with some examples
- **Course Non Goals**
  - Identification and application of design patterns for the purpose of designing

January 28, 2016 | Proprietary and Confidential | - 3 -


Pre-requisites
Fair knowledge of
Object Oriented Concepts
Unified Modeling Language
Capgemini


Intended Audience
Programmers working with Object Oriented Languages
Capgemini

Day Wise Schedule

- **Day 1**
  - Lesson 1: Introducing Design Principles
  - Lesson 2: Introducing Design Patterns
  - Lesson 3: Examples of Design Patterns

January 28, 2016 | Proprietary and Confidential | - 6 -

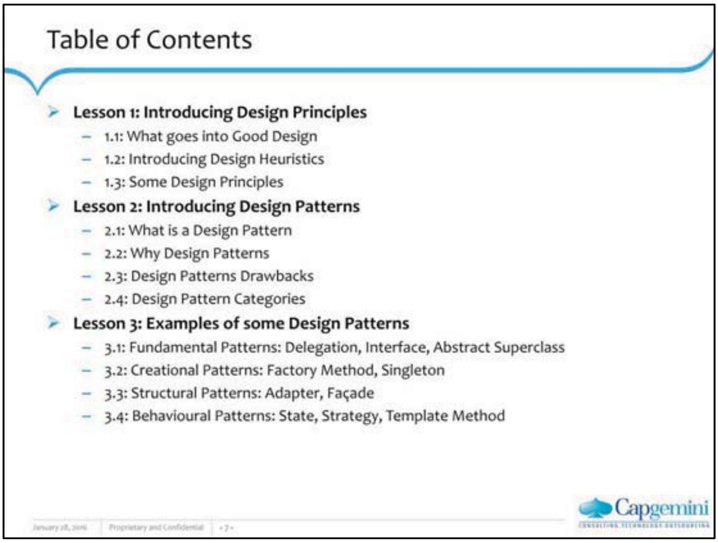


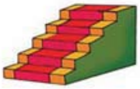
Table of Contents
Lesson 1: Introducing Design Principles
1.1: What goes into Good Design
1.2: Introducing Design Heuristics
1.3: Some Design Principles
Lesson 2: Introducing Design Patterns
2.1: What is a Design Pattern
2.2: Why Design Patterns
2.3: Design Patterns Drawbacks
2.4: Design Pattern Categories
Lesson 3: Examples of some Design Patterns
3.1: Fundamental Patterns: Delegation, Interface, Abstract Superclass
3.2: Creational Patterns: Factory Method, Singleton
3.3: Structural Patterns: Adapter, Façade
3.4: Behavioural Patterns: State, Strategy, Template Method
Capgemini

References

- **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process**
  - By Craig Larman
- **Object Oriented Design Heuristics**
  - By Arthur J. Riel
- **Head First Design Patterns**
  - By Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra

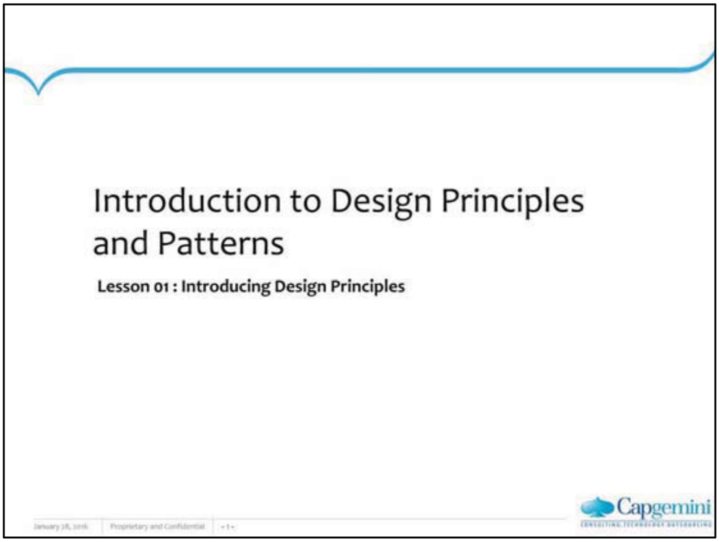
Next Step Coursess

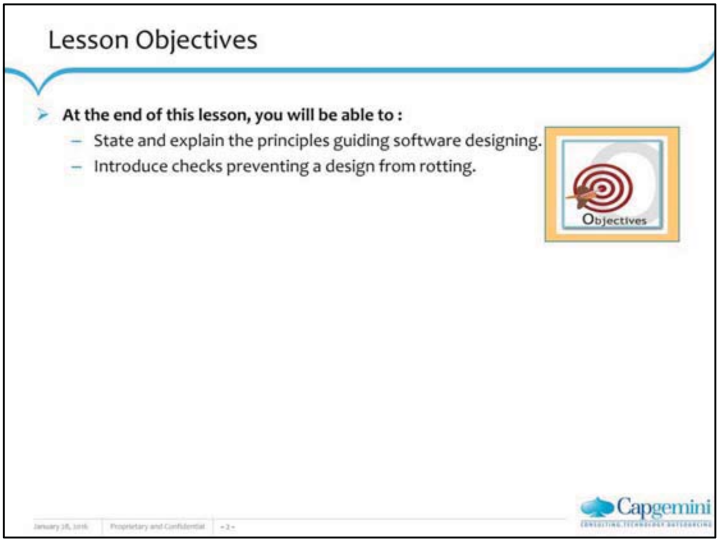
- **Object Oriented Analysis and Design**
- **GoF Design Patterns**
- **Technology Specific Designing and Technology Specific Design Patterns**



January 28, 2016 | Proprietary and Confidential | - 9 -

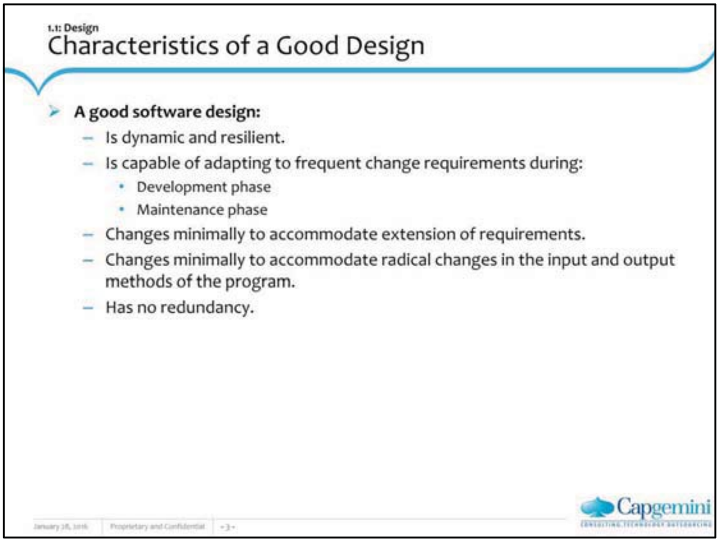
Capgemini


Introduction to Design Principles
and Patterns
Lesson 01 : Introducing Design Principles
Capgemini



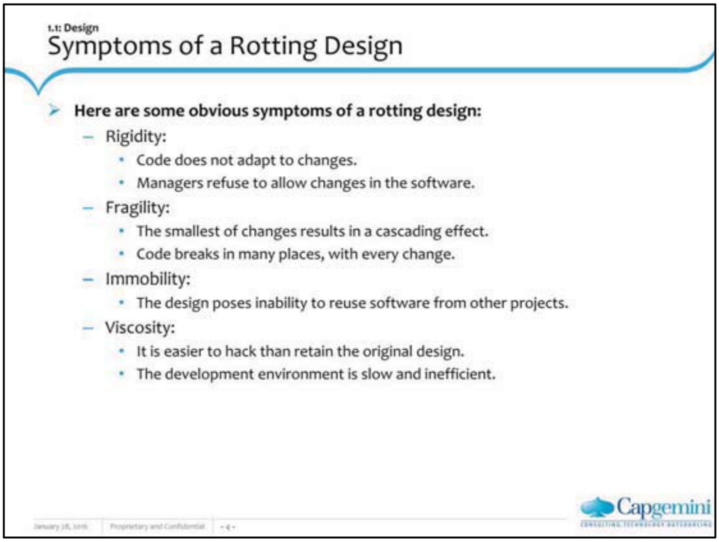
<u>Lesson Objectives</u>:

- We are now familiar with features of an object oriented language
- We write our Object Oriented programs based on the designs provided for the application
- For a given design, is there any method to gauge whether it is good, bad, or somewhere in between?
  - We may get the answer from an "OO Guru". If the design "feels right", the Guru certifies that the design is good.
- How do we know if the design "feels right"?
  - We can get the answer by looking at the **design heuristics** and **principles** introduced in this lesson.



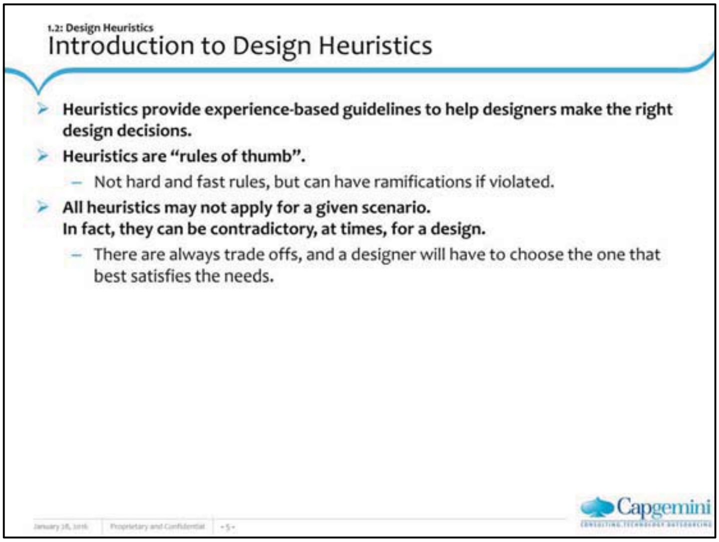
Characteristics of a Good Design:

- Today, we live in a world that is highly dynamic and diverse in nature. As a result, our requirements too are constantly changing. Therefore it is not surprising that "dynamic" and "resilient" software systems are the need of the day.
- The challenge is in developing a software system capable of adapting to the ever changing requirements, complexities of the problem domain, and difficulty of managing software processes.
- A **good design** and **code** adapts easily to the frequent changes that are done during development and maintenance. Very often extensive changes are involved, when a new functionality is added to an application. The design should be able to incorporate the added functionality with minimum change to the existing codes.
- Existing codes are already tested units, and changing them may result in unwanted changes in related functionalities. A software designer should make his or her design **foolproof** against such eventualities.



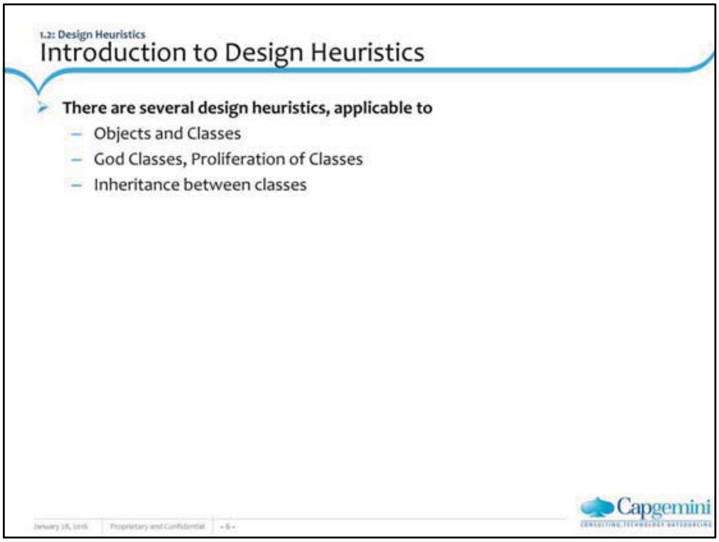
Symptoms of a Rotting Design:

- There are four primary symptoms of a rotting design:
  - **Rigidity:** It is the tendency of a software to change even in the smallest of ways. Every change results in a cascading effect, bringing about subsequent changes in related modules. When software exhibits such characteristics, the managers avoid fixing even the simplest of problems.
  - **Fragility:** It is the tendency of the software to break in many places every time it changes. Often the break occurs at a point remotely connected with the point of change. In fact, the two points may not be related at all. Due to the break, the fragility increases, and soon the situation goes beyond control.
  - **Immobility:** It is the inability to reuse software from other projects or from other parts of the same project. This situation occurs when a module has too much related software that it depends on.
  - **Viscosity:** Viscosity is of two types: viscosity of design and viscosity of environment. When design preserving methods are more difficult to implement than the hacks, then we can say that the **viscosity of design** is very high. When the design environment is slow and inefficient we can say that the **viscosity of environment** is high.



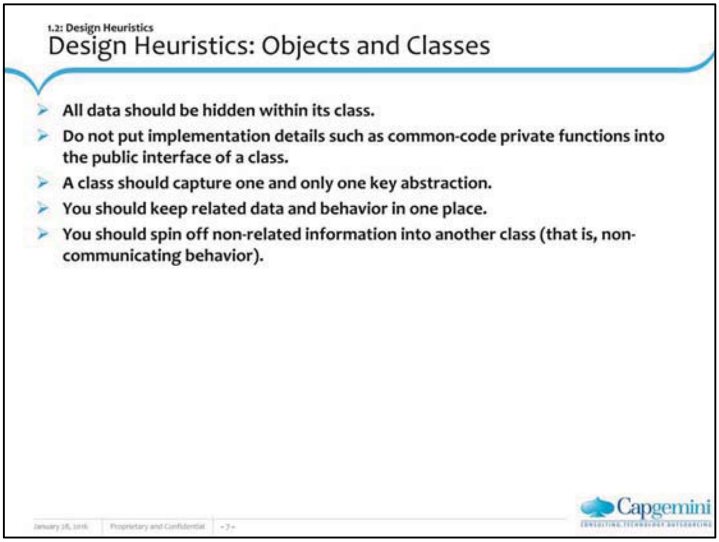
<u>Design Heuristics</u>:

- How do we know whether right decisions are taken with respect to designing? This is where guidelines captured over the years through experience help in taking the right decisions.
- These guidelines, also referred as “rules of thumb”, are not hard and fast rules. However, these can be thought of as “warning bells” if violated. Using these guidelines, appropriate changes can be brought about for removing the heuristic violation, wherever necessary



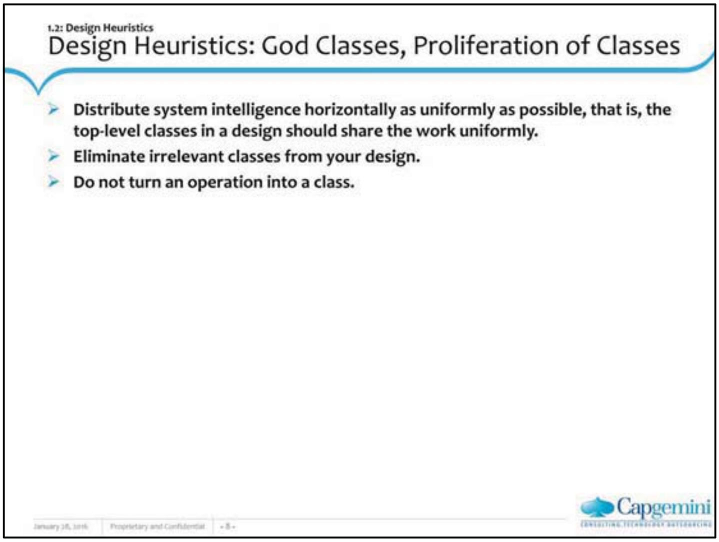
Design Heuristics:

- Arthur J. Riel has put together **60 design heuristics,** and these are applicable for various aspects like Objects and Classes, God Classes and Proliferation of classes, Inheritance between classes etc.
- We will see some of these on the subsequent slides.



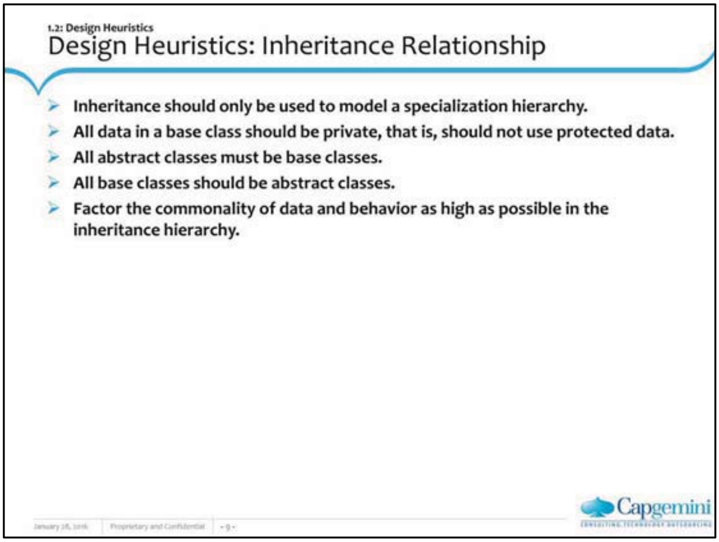
<u>Design Heuristics: Objects and Classes</u>:

- The above slide lists some of the design heuristics related to objects and classes.
  - Data is operated upon by operations, so there is a direct dependency between them. Data is bound to change, so it is useful to isolate consequences of that change to within the same class by enforcing encapsulation. For example, if data type changes, operations too will need modification. By keeping data, and operations that act on the data together, maintenance becomes easier.
  - This heuristic aims to reduce the complexity of class interface. Implementation details are meant to be “service operations”, which merely factors code within class considering reusability and modularity. They are not expected to be directly used by clients of class and hence must remain private.
  - Key abstraction is usually defined as an element of the problem domain. This heuristic implies the need for a class to be cohesive.
  - Violating this would mean that more than one class is affected in case of change in data because data and behavior actually belong to the same key abstraction and should have been captured in the same class.
  - Look out for classes where a subset of methods operate on a proper subset of data. In this case, one must spin off the other subset of related data and operations into another class.



Design Heuristics: God Classes, Proliferation of Classes:

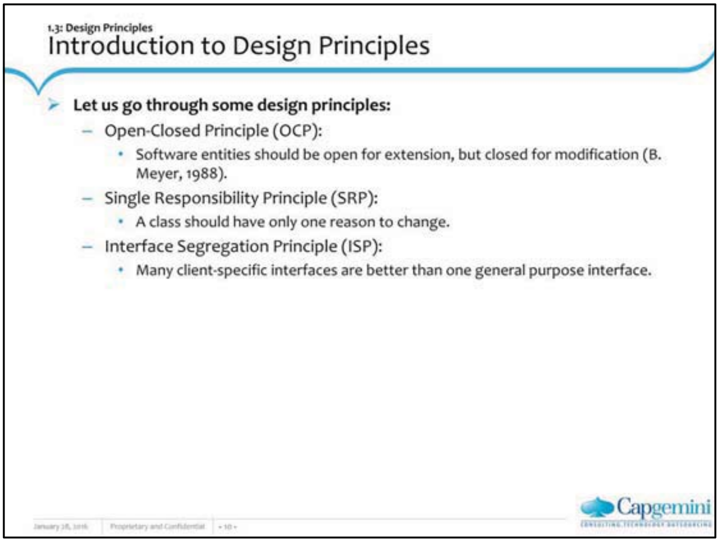
- The above slide lists some of the design heuristics related to God classes and proliferation of classes.
- Object oriented paradigm can go haywire if the design moves in the direction stated above. Poorly distributed system intelligence and creation of too many classes vis-à-vis the problem at hand are respectively referred to as “God Class” and “Proliferation of Classes”.
- Especially when one moves from procedural to object oriented platforms, the tendency is to create God object which will do most of the work and leave smaller details to rest of the classes.
- It is said that one cannot get **spaghetti code** with OO systems, but one can get **ravioli code** instead! If spaghetti is pasta in long thin strings which makes it easy for them to get entangled (“spaghetti code” refers to unstructured code where control can jump from one point to another), ravioli is small pasta pouches containing cheese / vegetables / meat (“ravioli code” is characterized by number of small and loosely coupled software components). If raviolis become too many (“Proliferation” of classes), then there is a different kind of maintenance problem. In which of these multiple classes should changes be incorporated?



Design Heuristics: Inheritance Relationship:

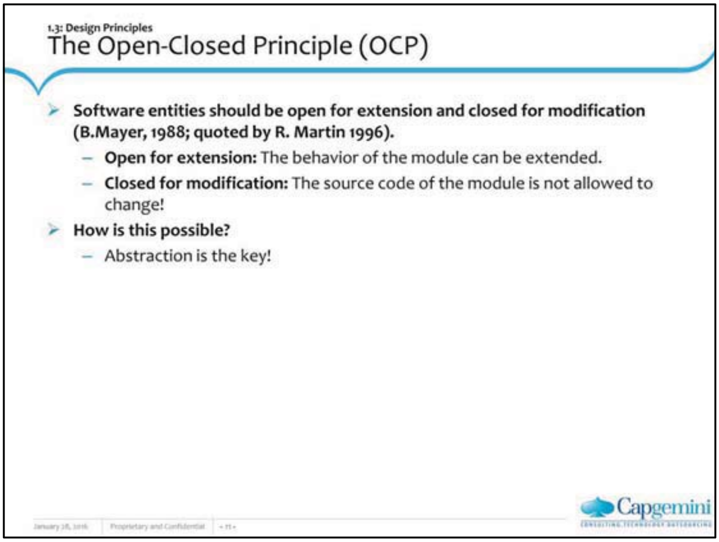
The above slide lists some of the design heuristics related to inheritance relationship.

- The first point actually compares inheritance with containership. It emphasizes that inheritance has to be used only in the cases where there is specialization hierarchy coming into picture. “Favor composition over inheritance” comes from here!
- The 2nd point hints at the fact that **inheritance** potentially violates **encapsulation**! When something is protected, it becomes available in the derived classes, thereby weakening data hiding.
- There are recommendations on how abstract classes and base classes must be considered in design. It is ideal to have an abstract class sitting at the base of the hierarchy.
- By factoring commonality of data and behaviour as high up in the hierarchy as possible, multiple derived classes can leverage the commonality.



<u>Some Design Principles in more details</u>:

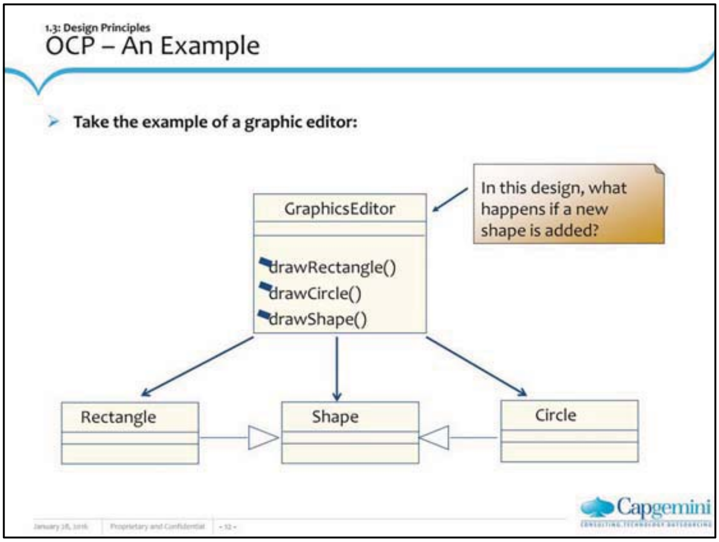
- We have looked at some design heuristics. In the above slide, we now look at some key Object-Oriented design principles.



<u>The Open-Closed Principle (OCP)</u>:

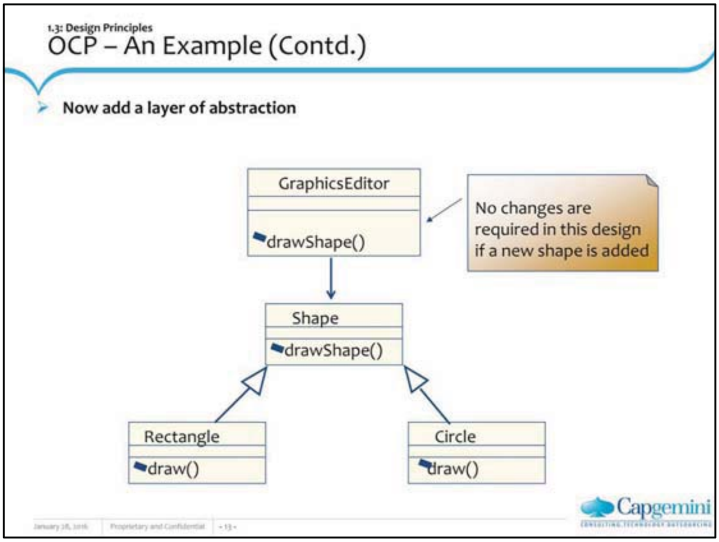
- Software modules that conform to the open-closed principle, exhibit two primary attributes:
    - **They are Open to Extension:** This implies that the behavior of the module can be extended. The system can be made to behave in new and different ways as requirements change, or to meet new applications.
    - **They are Closed to Modifications:** This implies that the source code of such a module remains intact. New codes are added to implement new and changed behavior.

- At first glance, these two attributes appear to be contradicting each other. The normal way to extend behavior of a module is by making changes to that module. A module that does not lend itself to change, is said to have **fixed behavior**. So how does one change existing modules without changing the source code?
- You can use the principle of **abstraction** to develop modules that are open to **extensions** and simultaneously closed to **modifications**. You can create abstractions that are fixed, yet represent an unbounded group of possible behaviors. The abstractions are abstract base classes, and all the possible derivative classes represent the unbounded group of possible behaviors.



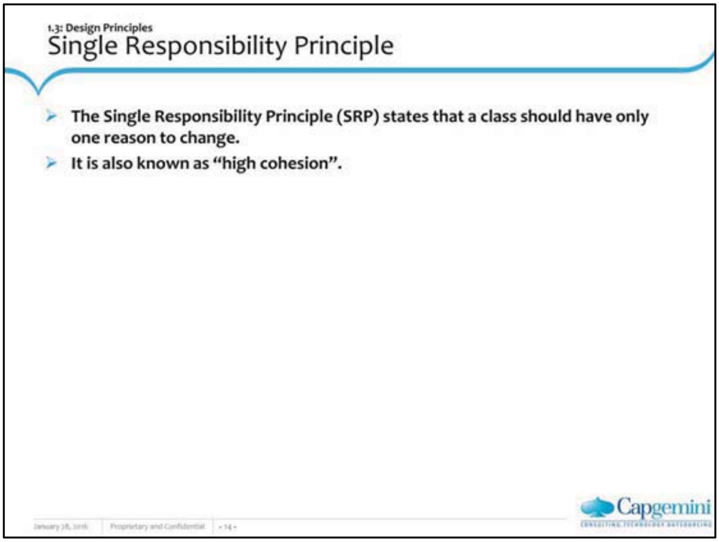
OCP – An Example:

- The above slide shows a graphic editor that can draw shapes, circles, and rectangles. However, when a new shape is added, the **graphic editor** has to be changed. This implies that the **source code** needs changing. Therefore this example does not conform to the OCP principle. The design cannot be closed against new kinds of shape.
- The next slide shows a modified version of the example after incorporating OCP.



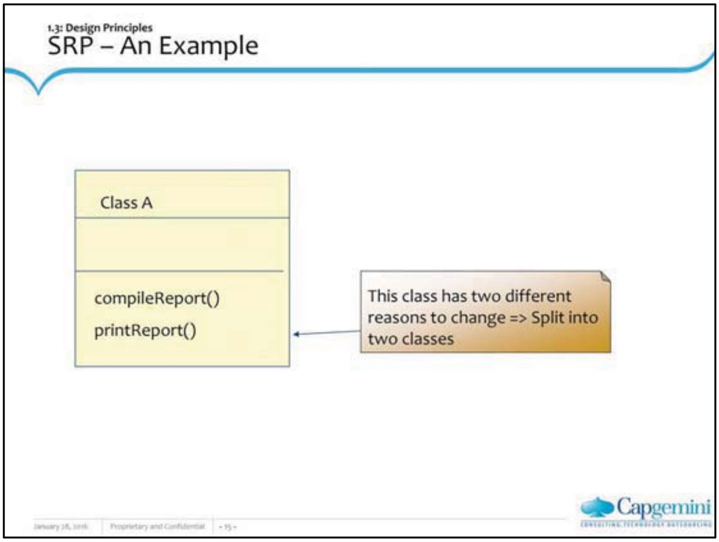
OCP – An Example (contd.):

- Now, consider adding a layer of abstraction as shown in the above slide.
- New shapes can extend the **Shape** class. So behavior can be extended without modification of **GraphicsEditor** class. The source remains intact.
- This was a relatively simple example with a simple solution. In the real world, the **Shape** class would have many more methods. Still adding a new shape to the application is simple and requires the creation of the new derivative and the implementation of all its functions. The designs based on OCP incorporate changes by adding new codes rather than by changing existing codes. Hence one does not encounter the cascading effect seen otherwise.
- It is important to note that no application can be 100% closed. The closure cannot be complete. Hence designers look for **strategic closure**. From a designers angle, this situation requires deciding on the kind of changes against which you want to close your design. This calls for a certain degree of intuition and experience. An experienced designer has his or her finger on the pulse of the industry and the user. He or she can normally foresee the probability of different kinds of changes and design accordingly.



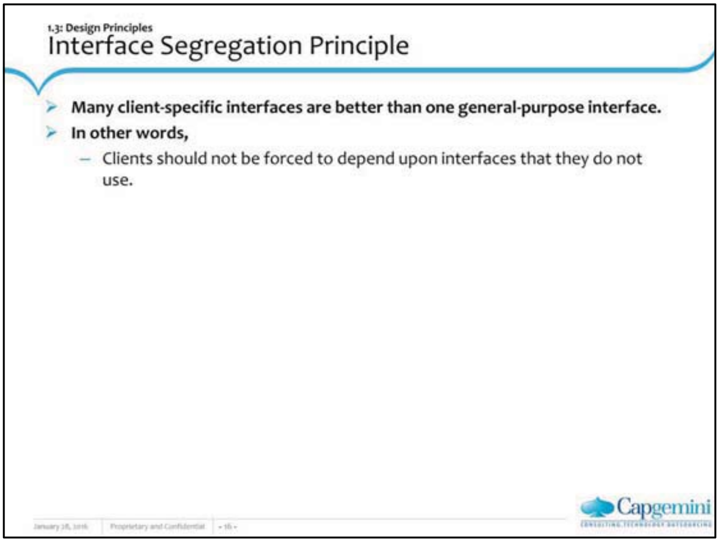
<u>Single Responsibility Principle (SRP)</u>:

- In this context, a responsibility is considered to be one reason to change. This principle states that if we have two reasons for a class to change, then we have to split the functionality into two classes. Each class will handle only one responsibility and in future if we need to make one change, then we are going to make it in the class which handles it. When we need to make a change in a class having more responsibilities, the change might affect the other functionality of the classes.
- **Cohesion** is sticking or working together, that is, the state or condition of joining or working together to form a united whole, or the tendency to do this. A **class** should be **cohesive**, that is, the class should have only a single purpose to live and all its methods should work together to help achieve this goal.
- The **Single Responsibility principle** is a simple and intuitive. However, in practice it is sometimes hard to get it right.



SRP – An Example:

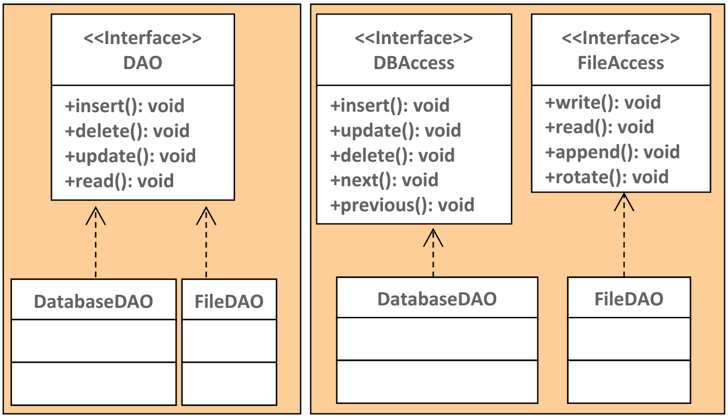
- As an example, consider a class that compiles and prints a report. Such a class can be changed for two reasons.
  - ➢ First, the content of the report can change.
  - ➢ Second, the format of the report can change.
- These two things change for very different causes – one substantive, and one cosmetic. The SRP says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes.
- It is important to keep a class focused on a single concern because it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, then there is a greater danger that the printing code will break if it is part of the same class.
- When a class has more than one responsibility (that is, reason to change), these responsibilities are coupled. This scenario makes the class more difficult to understand, more difficult to change, and more difficult to reuse. **Cohesion** should also be applied at the method level, and for the exact same reasons.
- The challenge with SRP is getting the granularity of a responsibility right. Sometimes, it is easier to see the responsibilities are unrelated. But more often, it needs thorough thinking!
- One last point regarding SRP is that if you cannot separate the responsibilities into separate classes, then at least consider separating them to different interfaces.



<u>Interface Segregation Principle (ISP)</u>:

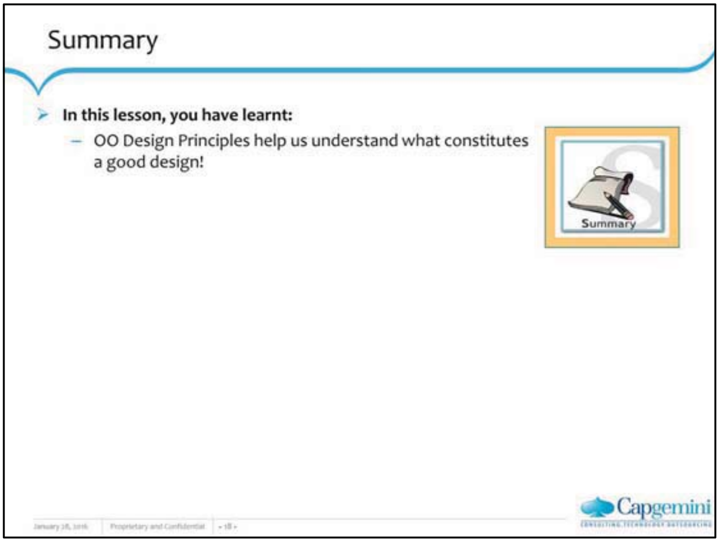
- **Interface Segregation Principle (ISP)** deals with designing "**cohesive**" interfaces and avoiding **"fat"** interfaces. It focuses on the cohesiveness of interfaces with respect to the clients that use them. The idea here is that each client may use a particular object or subsystem in a different way.
- ISP states that clients should not be forced to implement interfaces they do not use. Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one sub-module.
- Why should "fat" interfaces be avoided?
  - ➢ Each client depends on the single class interface. Hence there is an inadvertent coupling between the clients.
- How is the client coupling harmful?
  - ➢ Suppose a client needs that additional functionality be added to the single class interface. When this functionality is added to the interface, every other client must change to support the functionality even though none of them need it. Thus one change of a client forces the change to propagate throughout the system. This situation, in turn, can result in time consuming code maintenance and hard to locate bugs.

ISP – An Example:



Which of these designs follow ISP?

- Imagine that in your application you are required to write some **Data Access Objects (DAOs)**. These data objects should support a variety of data sources. Let us consider that the two main data sources are file and database. You must be careful enough to come up with an **interface-based design**, where the implementation of data access can be varied without affecting the client code using your DAO object.
- What happens if the data source is read-only?
    - The methods for inserting and updating data are not needed. On the other hand, if the *DAO* object should implement the DAO *interface*, then it will have to provide a null implementation for those methods defined in the *interface*. This is still acceptable, but the design is gradually going wrong.
- What if there is a need to rotate the file data source to a different file once a certain amount of data has been written to the file?
    - That will require a separate method to add to the DAO *interface*.
- When a single interface is designed to support different groups of behaviors, then they are, by virtue, inherently poorly designed, and are called Fat interfaces. They are called **Fat** because they grow enormously with each additional function required by clients using that interface. Thus, for the problem with the Data Access Objects, follow the Interface Segregation Principle, and separate the interfaces based on the behaviors. The database access classes and file access classes should subscribe to two separate interfaces.

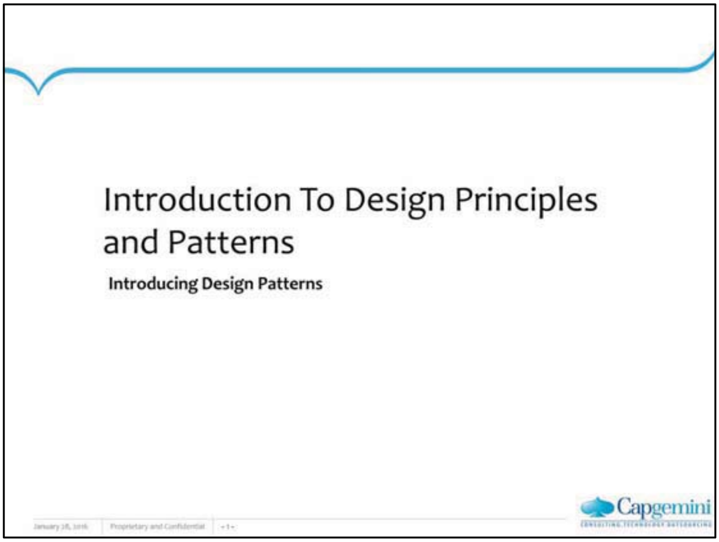

Summary
In this lesson, you have learnt:
– OO Design Principles help us understand what constitutes a good design!
Summary
Capgemini

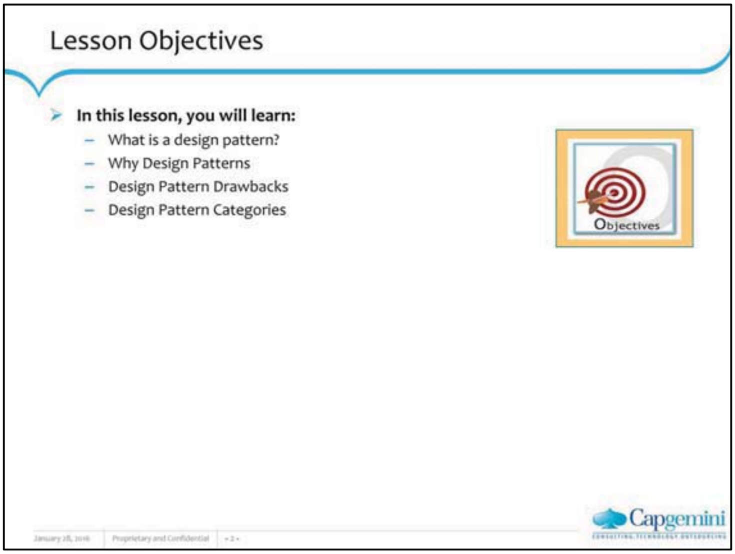
Review Question

- **Question 1: Related data and behavior should be kept in same class. (True/False)**

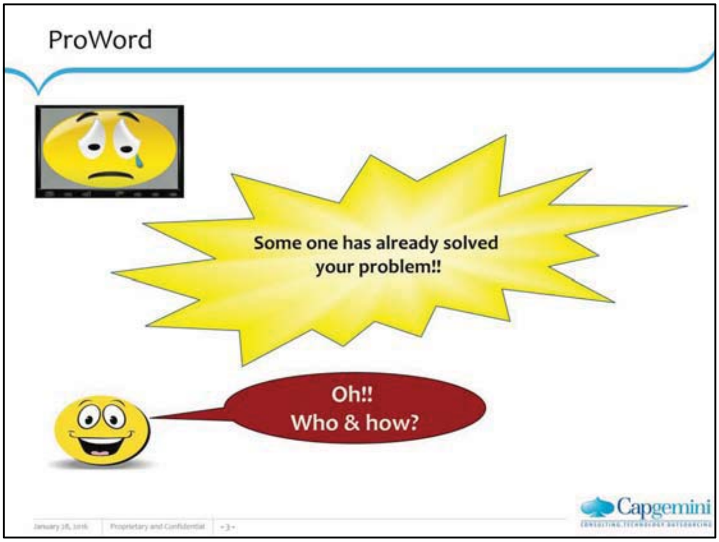
- **Question 2: A God class should centrally control the application. (True/False)**
- **Question 3: Favor ___ over ___.**
- **Question 4: ___ principle says modules should be open for extension but closed for modification.**
- **Question 5: ___ principle discourages use of Fat interfaces.**

Capgemini

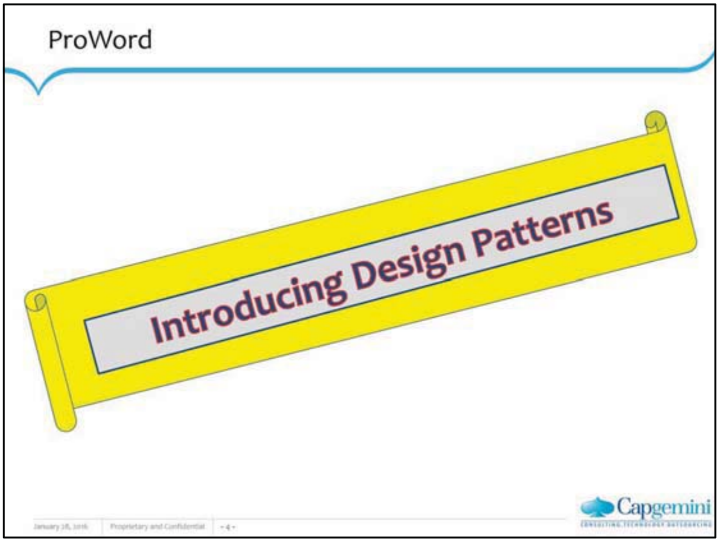

Introduction To Design Principles and Patterns
Introducing Design Patterns
Capgemini



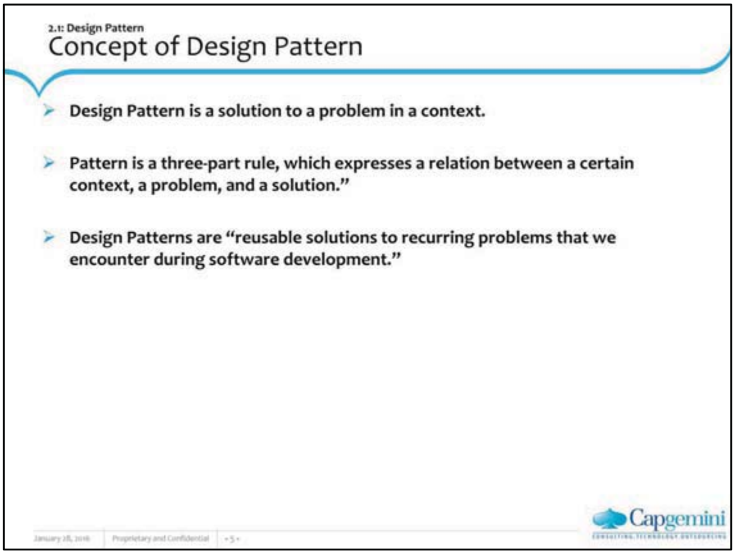
<u>Lesson Objectives</u>:
In this lesson, we will understand the what and why of design patterns, see how design patterns are classified; and have a look at some of these design patterns.



Is it possible that solutions to some of our problems already exist??

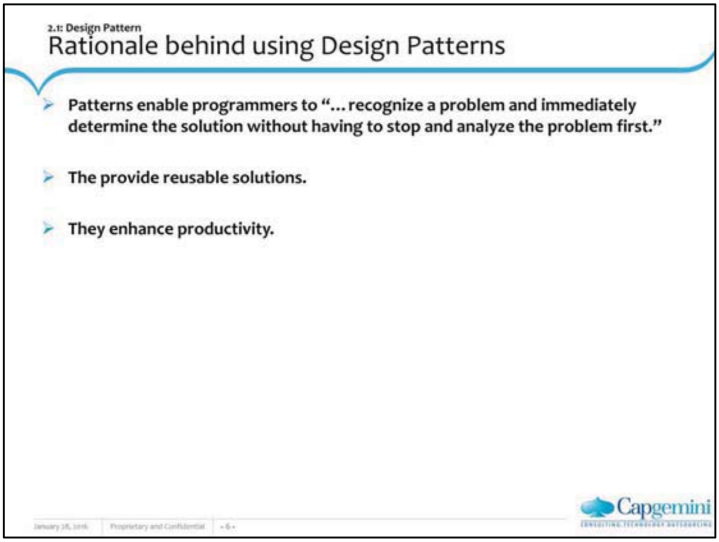


Yes…design patterns provide just that!



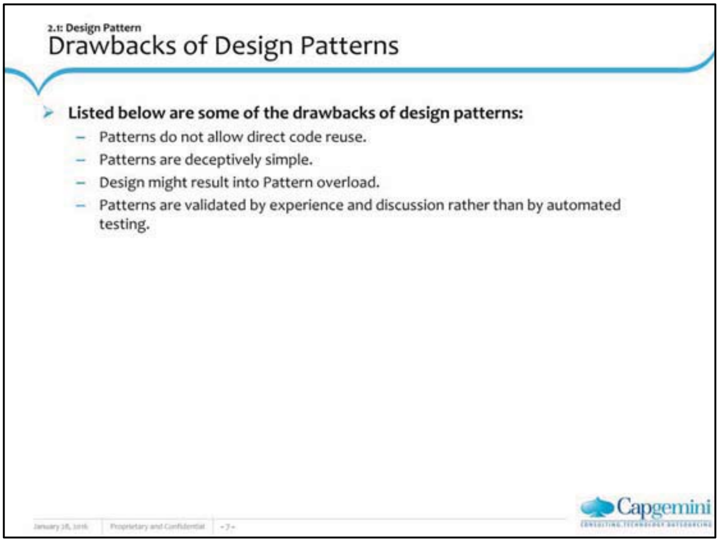
<u>What is a Design Pattern</u>?

- "A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever using it the same way twice."
- Patterns can be applied to many areas of human endeavor, including software development.

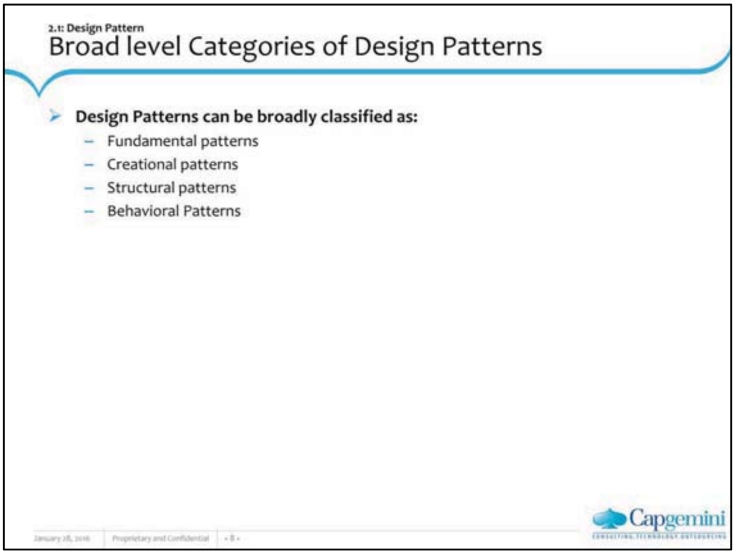


Why Design Patterns?

- Designing object-oriented code is hard, and designing reusable object-oriented software is even harder.
- Patterns enable programmers to "…recognize a problem and immediately determine the solution without having to stop and analyze the problem first."
- Well structured object-oriented systems have **recurring patterns** of classes and objects.
- The patterns provide a framework for communicating complexities of OO design at a high level of abstraction. Bottom line is **productivity**.
- Experienced designers reuse solutions that have worked in the past.
- Knowledge of the patterns that have worked in the past allows a designer to be more productive and the resulting design to be more flexible and reusable.

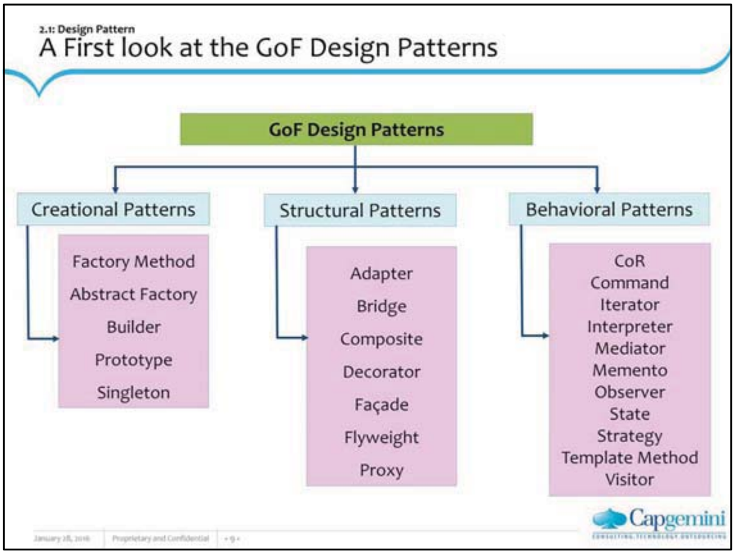


Note:
Design patterns have drawbacks too! Besides the drawbacks mentioned in the slide, Integrating patterns into a software development process is a human-intensive task.



<u>Classification of GOF Design Patterns</u>:

- The Gang of Four (GoF) Design Patterns can be broadly classified as :
  - **Fundamental Patterns:** They are the building blocks for the other three categories of Design Patterns.
  - **Creational Patterns:** They deal with creation, initializing, and configuring classes and objects.
  - **Structural Patterns:** They facilitate decoupling interface and implementation of classes and objects.
  - **Behavioral Patterns:** They take care of dynamic interactions among societies of classes and objects. They also give guidelines on how to distribute responsibilities amongst the classes.



Note:
- There are 23 GOF Design Patterns.
- They have been classified as shown on the slide. We shall see some more details with examples for some of these design patterns in the next lesson.
- Another classification for design patterns is class based or object based. Class based Design Patterns uses “inheritance” as the basic principle whereas the object based patterns use “composition”.
- We have seen, “Favor Composition over Inheritance”.
  Note that in design patterns, “composition” is being favored as most of the Design Patterns are “Object-based”. The class based design patterns are Factory Method, Adaptor, Interpreter and Template Method.

Summary

- **In this lesson, you have learnt:**
  - Concept of Design Pattern
  - Rationale behind using Design Patterns
  - Drawbacks of Design Patterns
  - Classification of Design Patterns

Review Question

- Question 1: Design pattern is a solution to ___ within a particular ___.
- Question 2: Name different types of GOF Design Patterns.