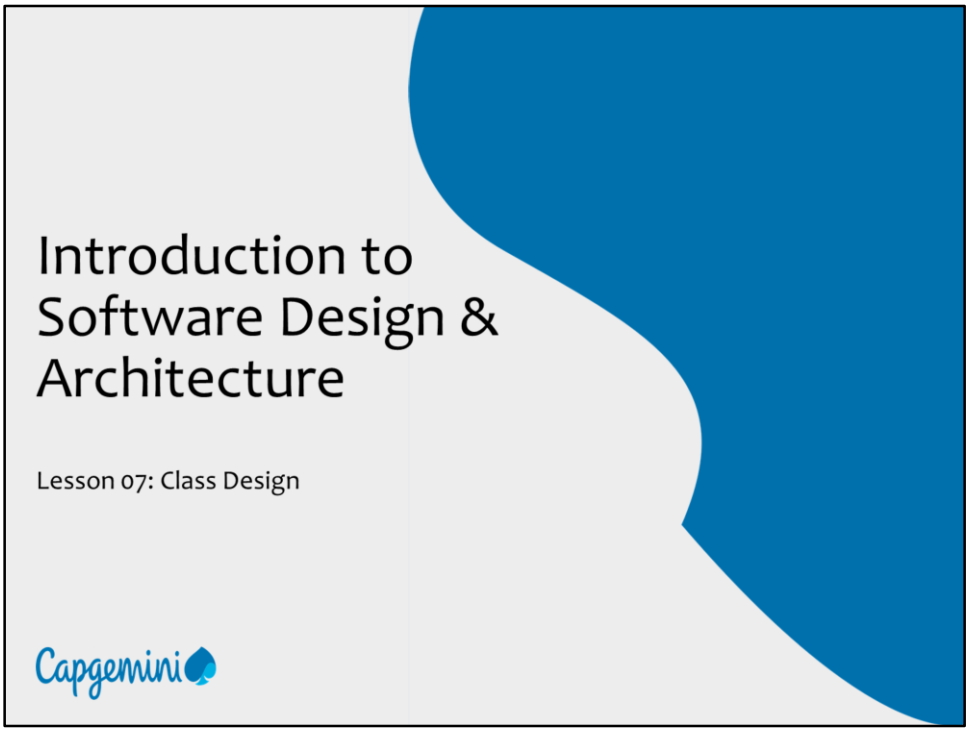
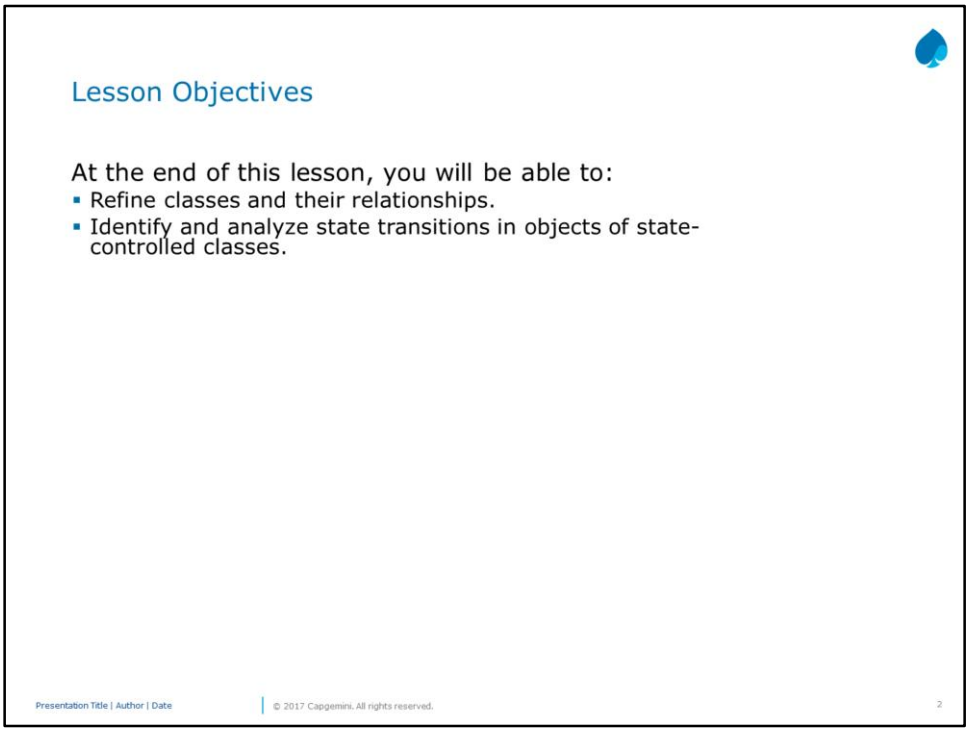
Instructor Notes:

Add instructor notes here.



Instructor Notes:

We continue looking at the Design Model of LLD, where the application design is refined to include refinements for classes and their relationships.

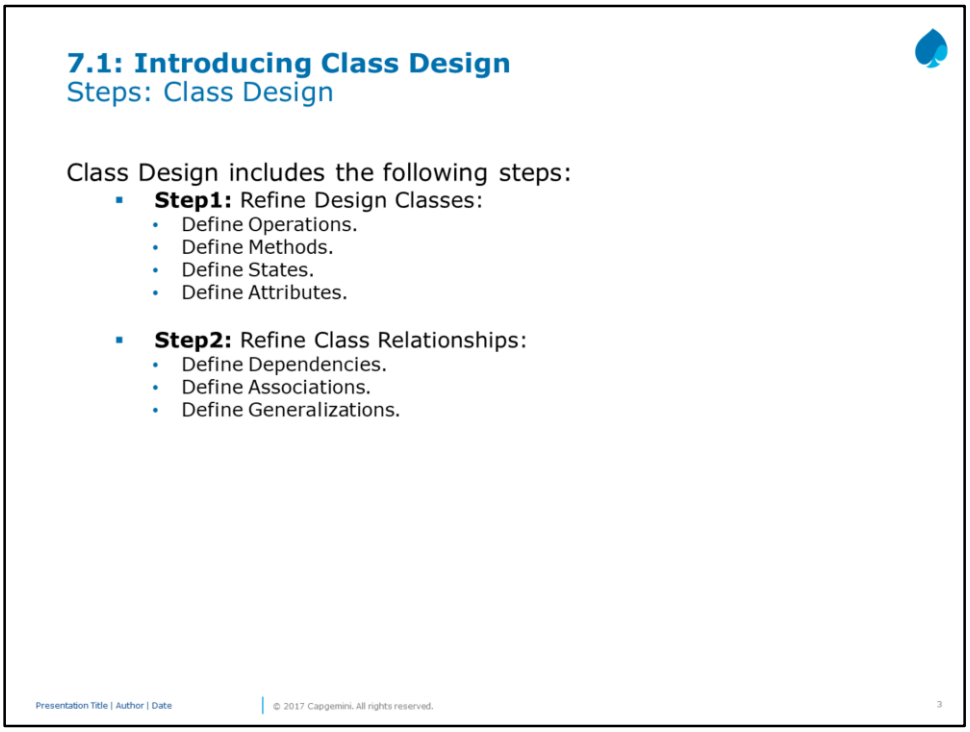


Lesson Objectives:

- Class Design activity involves identifying additional classes and associated relationships. It also involves refining all classes and relationships that are needed to implement the functionality.
- As part of this lesson, we will also have a look at the State Chart Diagram.

Instructor Notes:

We have already arrived at set of design classes. We review them and refine them to include implementation specific aspects; and also refine class relationships.

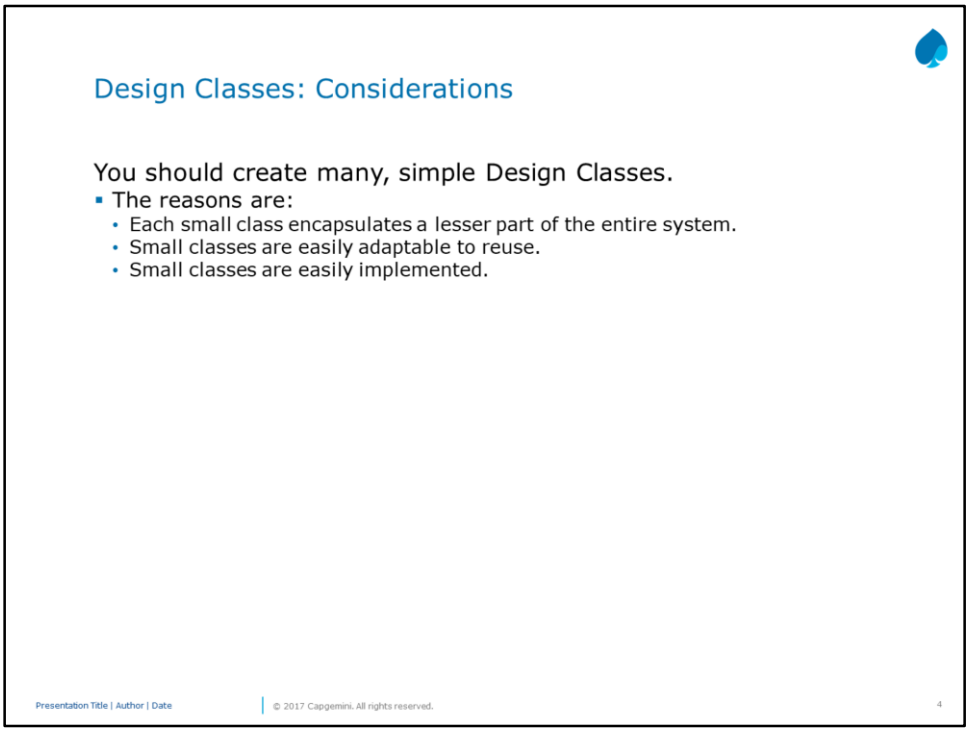


Steps: Class Design:

- As part of LLD, we have now arrived at one or more **design classes** for the **analysis classes** created earlier. Now, these initial design classes go through various modifications and refinements.
- We identify additional **operations** on the design classes. This includes:
  - naming and describing the operations
  - defining operation visibility
  - defining class operations
- After defining the operations, we define **methods** specifying their (operation's) implementation.
- At times, the behavior of an **operation** depends on the **state** of the receiver object. Hence we need to develop the **state machine**, which defines **states** that an object can assume and the **events** that cause the object to move from one state to another.
- Subsequently, we include/refine the **attributes**.
- **Relationships** (between classes) identified in previous phases need to be refined. Hence we now look for **dependencies**. After defining dependencies, you need to focus on the remaining **associations**. Some of these can get modeled as **aggregation** or **composition**. Finally, organize design classes, having common behavior and structure, into a **generalization hierarchy**.
- Each of these steps are described in the subsequent slides.

Instructor Notes:

Some thumb rules for design classes.

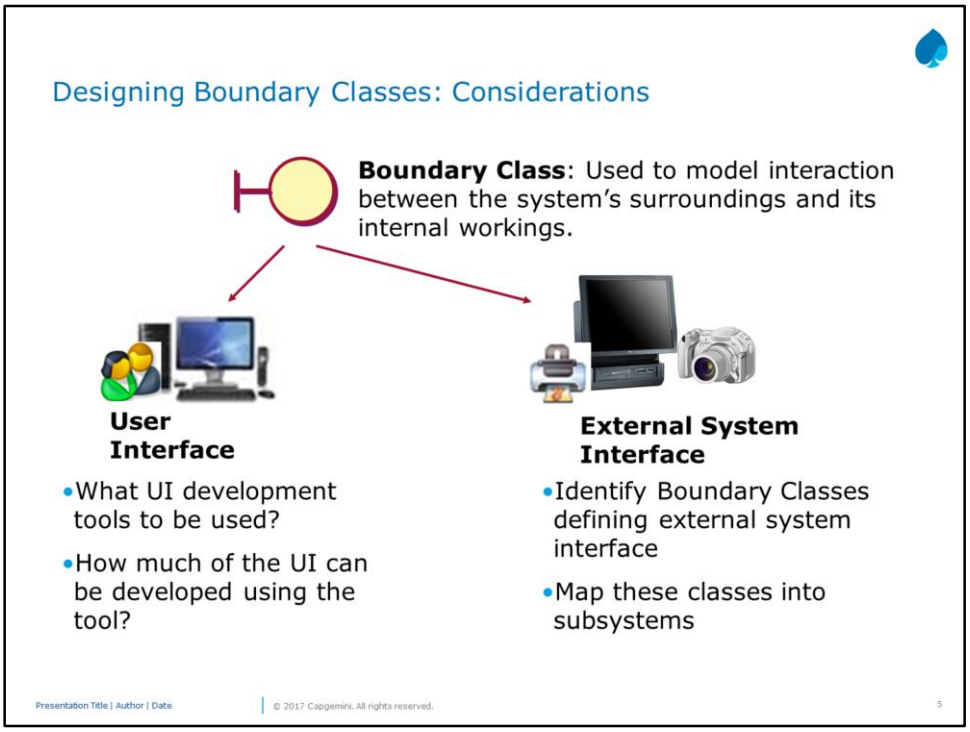


Considerations for Design Classes:

- When defining **design classes**, consider the **size** of each class. This, in turn, depends on the implementation requirements. It is advisable to model small and simple design classes, as they are easy to implement and can be reused.
- Further, each **class** should directly map to a phenomenon in the **implementation language**, such that the code developed is good.

Instructor Notes:

Some thumb rules for designing the boundary classes.

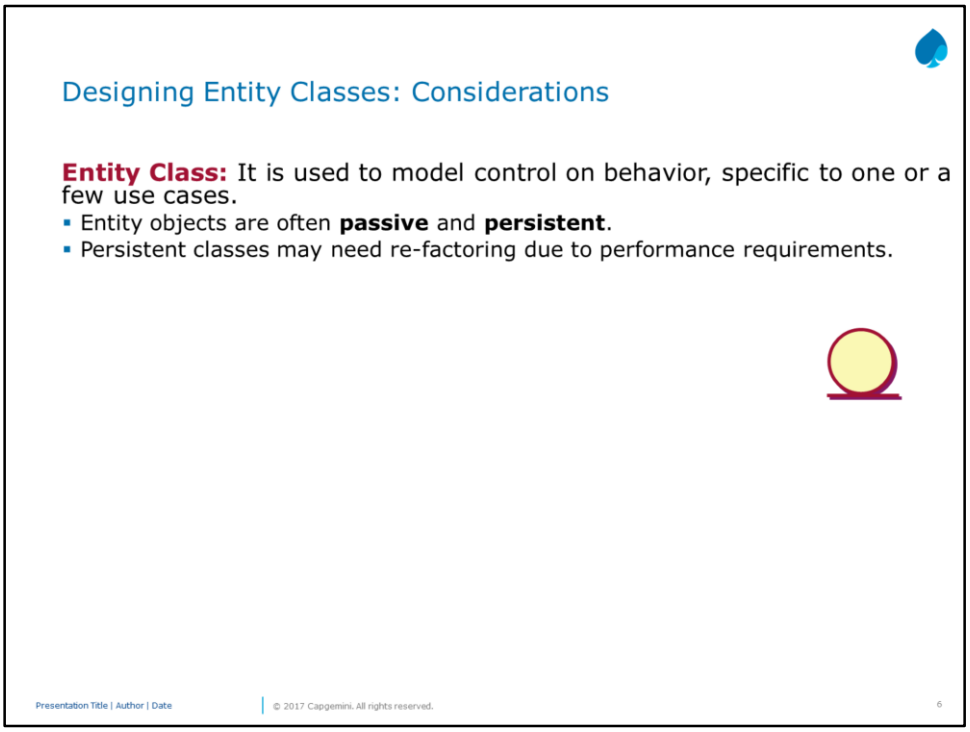


Designing Boundary Classes: Considerations:

- **Boundary classes** either represent interfaces to users or to other systems.
  - **Boundary classes defining user interface:** Try to model one boundary class for every **window**, or one for each **form**, in the user interface. UI boundary classes depend on the user-interface development tools used.
  - **Boundary classes defining interfaces with external systems:** Typically, boundary classes that represent interfaces to other systems are modeled as **subsystems**, because they often have complex internal behavior.
    If the interface behavior is simple (perhaps acting as only a pass-through to an existing API to the external system), you might choose to represent the interface with one or more **design classes**. If you choose this route, then use a single design class per protocol, interface, or API, and note special requirements about standards you used in the special requirements of the class.

Instructor Notes:

Some thumb rules for designing the entity classes.

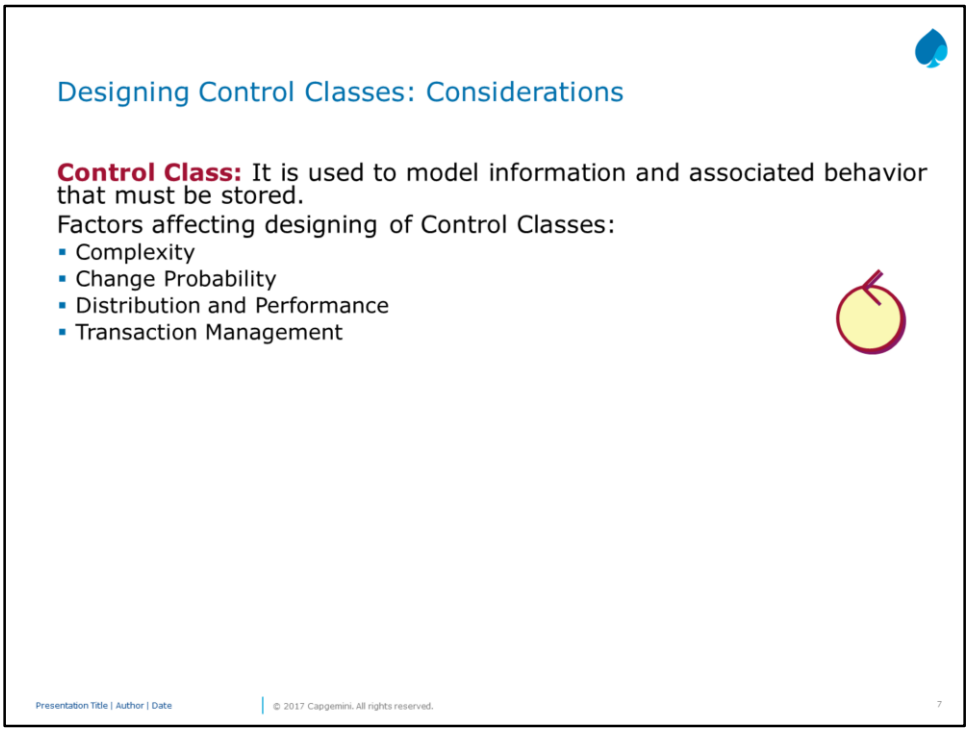


Designing Entity Classes: Considerations:

- During Analysis phase, the **entity classes** may have been identified. While most are good logical abstractions, the performance considerations may force some re-factoring of persistent classes, thus causing changes to the Design Model.
- These are discussed jointly between the Database Designer and the Designer responsible for the class.

Instructor Notes:

Some thumb rules for designing the control classes.

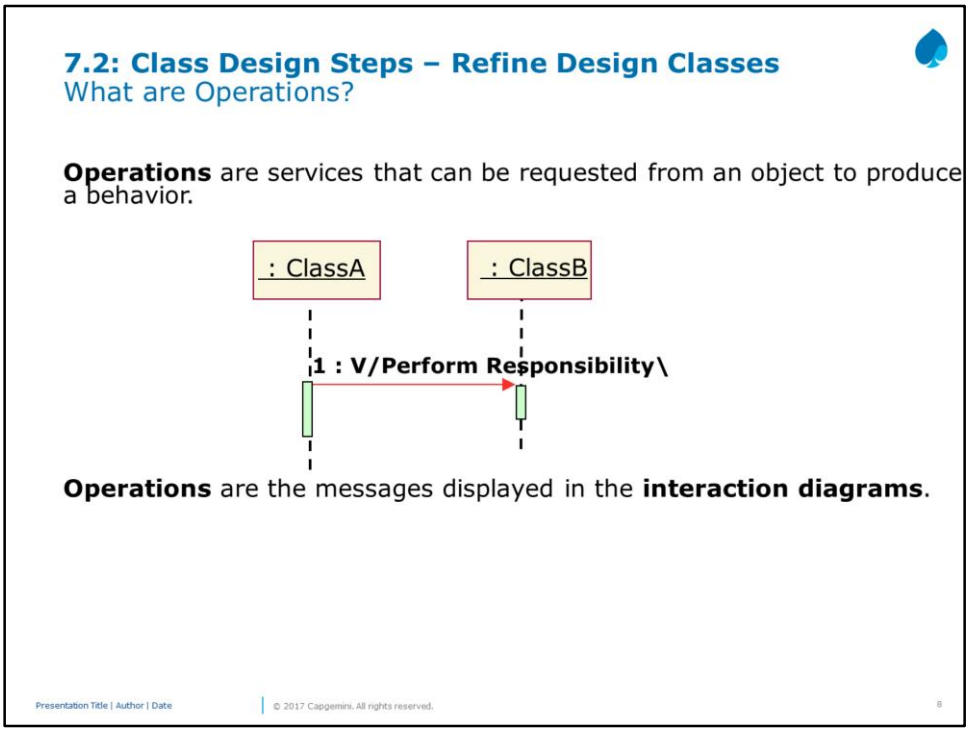


Designing Control Classes: Considerations:

- You can look for the following characteristics, when modeling control classes:
  - **Complexity of behavior:** Complex coordinating and controlling behavior are modeled using control classes.
  - **Change Probability:** If the probability of changing flows of events is low or the cost is negligible, then the extra expense and complexity of additional control classes might not be justified. Control classes are ideal for dynamic systems, with a high degree of probability for changing.
  - **Distribution and Performance:** The need to run parts of the application on different **nodes** or in different **process spaces**, introduces the need to specialize **design model elements**. This specialization is often accomplished by adding **control objects**, and distributing behavior from the **boundary** and **entity classes** onto the **control classes**. In doing this distribution:
    - The boundary classes migrate toward providing purely UI services
    - The entity classes move toward providing purely data services
    - The control classes provide the rest
  - **Transaction Management:** Managing transactions is a classic coordination activity. Without a framework to handle transaction management, one or more transaction manager classes would have to interact to ensure that you maintain the integrity of the transactions.

Instructor Notes:

We have already defined operations for classes based on incoming messages on objects in a sequence diagram.

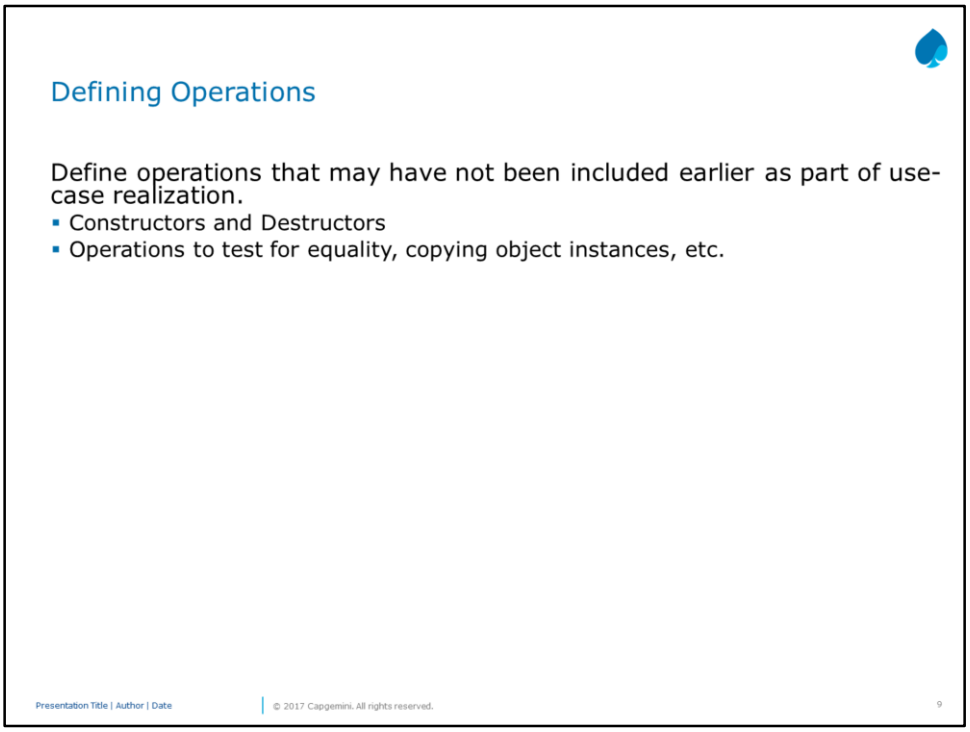


What are Operations?

- To identify **operations** on **design classes**:
  a) Study the **responsibilities** of each corresponding **analysis class**, creating an operation for each responsibility. Use the description of the responsibility as the initial description of the operation.
  b) Study the **Use-Case Realizations** in the **class participations** to see how the operations are used by the Use-Case Realizations. Extend the operations, one Use-Case Realization at a time, thus refining the operations, their descriptions, return types, and parameters. Requirements of each Use-Case Realization, vis-à-vis classes, are textually described in the Flow of Events of the Use-Case Realization.
  c) Study the **Use-Case Special Requirements**, to ensure that you do not miss implicit requirements on the operation that might be stated there.

Instructor Notes:

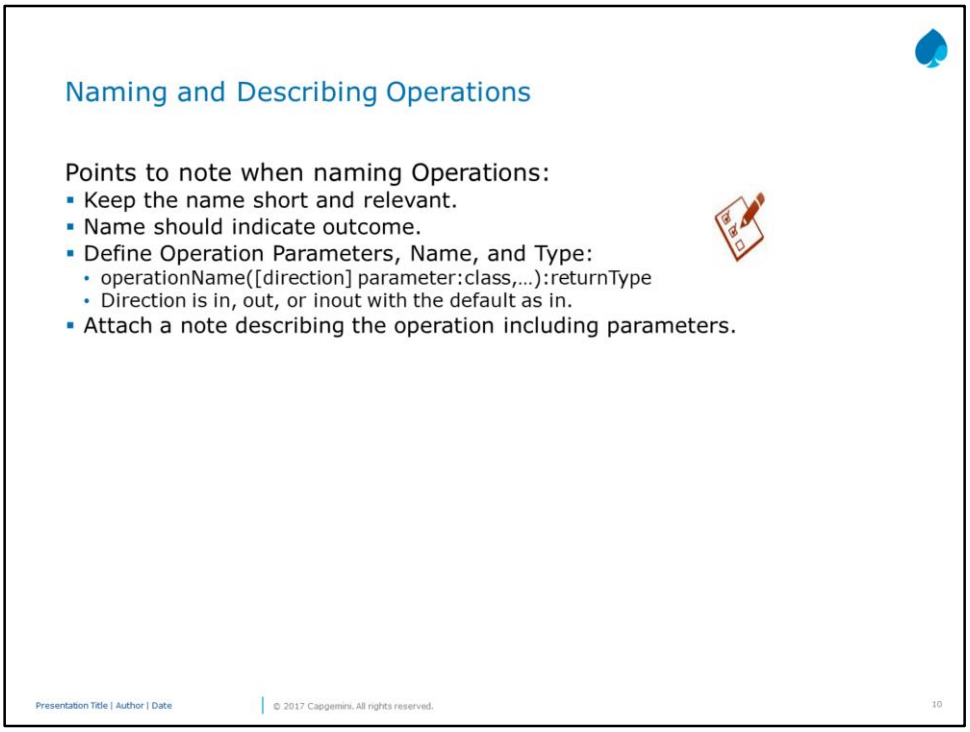
But now add operations that are needed from an implementation perspective.



Defining Operations:
Most of the operations are defined during use-case realizations. You need to add in those that have not been already included. For example: Constructors and Destructors, operations to test for equality, copying object instances…

Instructor Notes:

Operations must be completely defined in terms of parameters and return values.

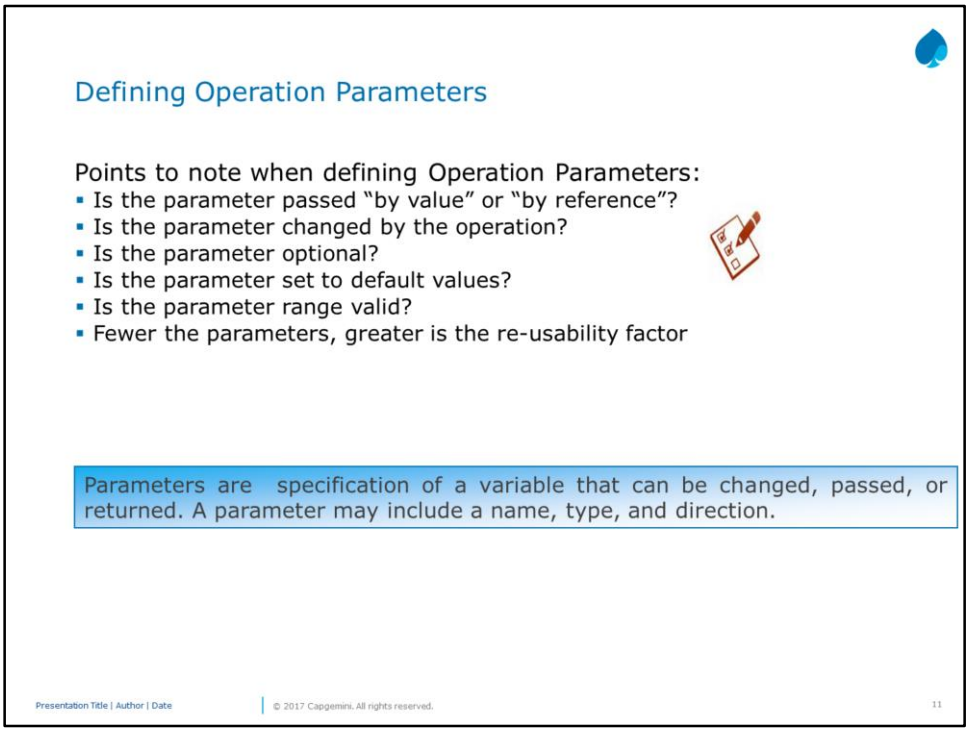


Naming and Describing Operations:

- Here is a little elaboration on the points on the slide:
  - Use naming conventions for the implementation language, when naming operations, return types, parameters and their types.
  - Keep the name short. However, it should give an idea of the result achieved by the operation. That means the name of an operation should clearly show its purpose. Avoid unspecific names, such as getData, that are not descriptive about the result they return. Use a name that shows exactly what is expected, such as getAddress.

Instructor Notes:

Some points to help define operation parameters.

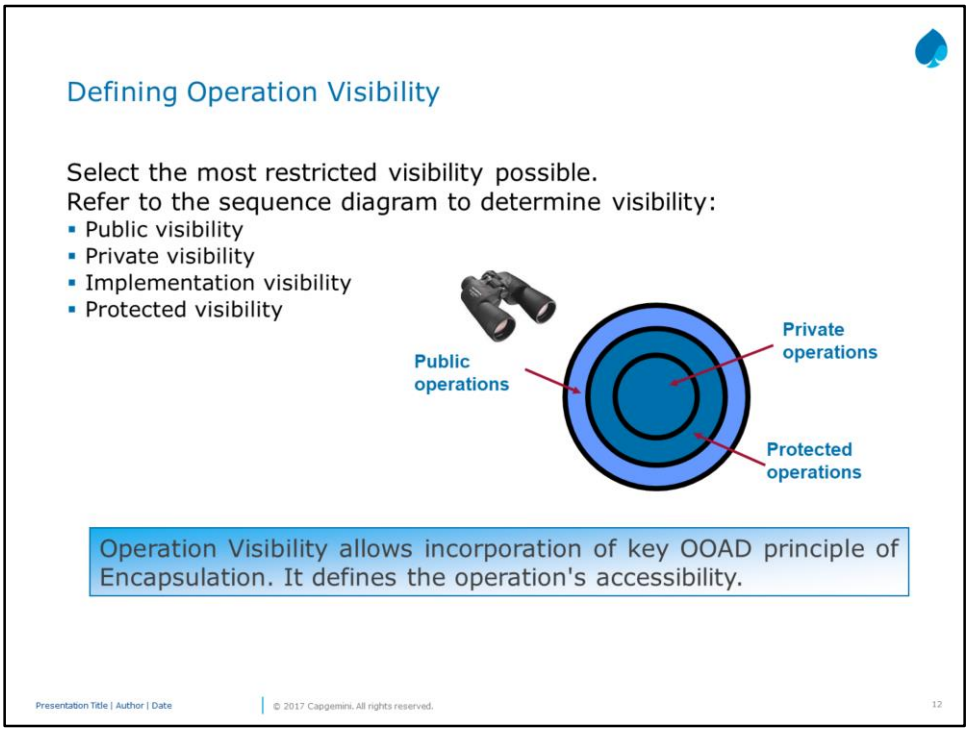


Defining Operation Parameters:

- **The parameters:** For each parameter, create a short descriptive name, decide on its class, and give it a brief description. As you specify parameters, remember that "fewer parameters" mean "better reusability". A small number of parameters makes the operation easier to understand. Therefore a higher likelihood of finding similar operations exists.
  You may need to divide an operation with many parameters into several operations. The operation must be understandable to those who want to use it. The brief description should include:
  - The meaning of the parameters, if not apparent from their names
  - The option whether the parameter is passed **by value** or **by reference**
  - Parameters that must have values supplied
  - Parameters that can be optional and their default values, if no value is provided
  - Valid ranges for parameters, if applicable
  - Steps that are done in the operation
  - The **by reference** parameters that are changed by the operation

Instructor Notes:

To make the most of OO features, decide on the most restrictive visibility as possible.

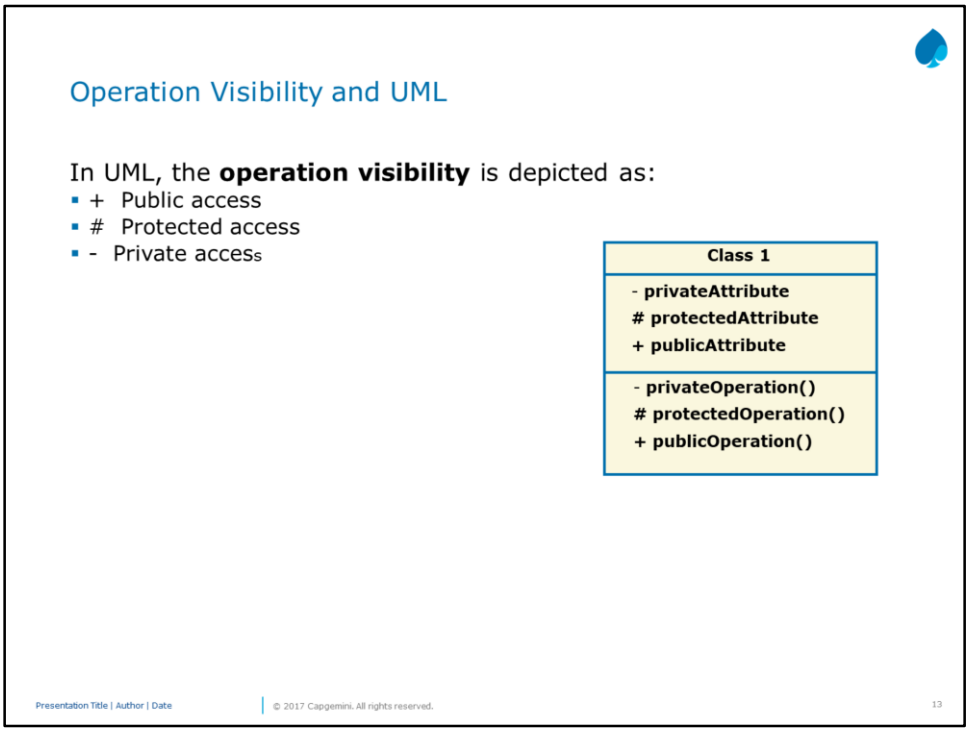


Defining Operation Visibility:

- For each **operation**, identify the **export visibility** of the operation from the following types:
  - **Public:** The operation is visible to model elements other than the class itself.
  - **Private:** The operation is visible only to the class itself and to *friends* of the class.
  - **Implementation:** The operation is visible only within the class itself.
  - **Protected:** The operation is visible only to the class itself, to its subclasses, or to *friends* of the class (language-dependent).
- Select the most restricted visibility possible that can still accomplish the **objectives** of the operation. To do this, look at the sequence diagrams, and for each message, determine whether the message is coming from either of the following:
  - A class outside the receiver's package (requires **public** visibility)
  - Inside of the package (requires **implementation** visibility)
  - A subclass (requires **protected** visibility)
  - From the class itself or a friend (requires **private** visibility)
- Normally, static final members are publicly visible.

Instructor Notes:

These notations may be different in different tools.

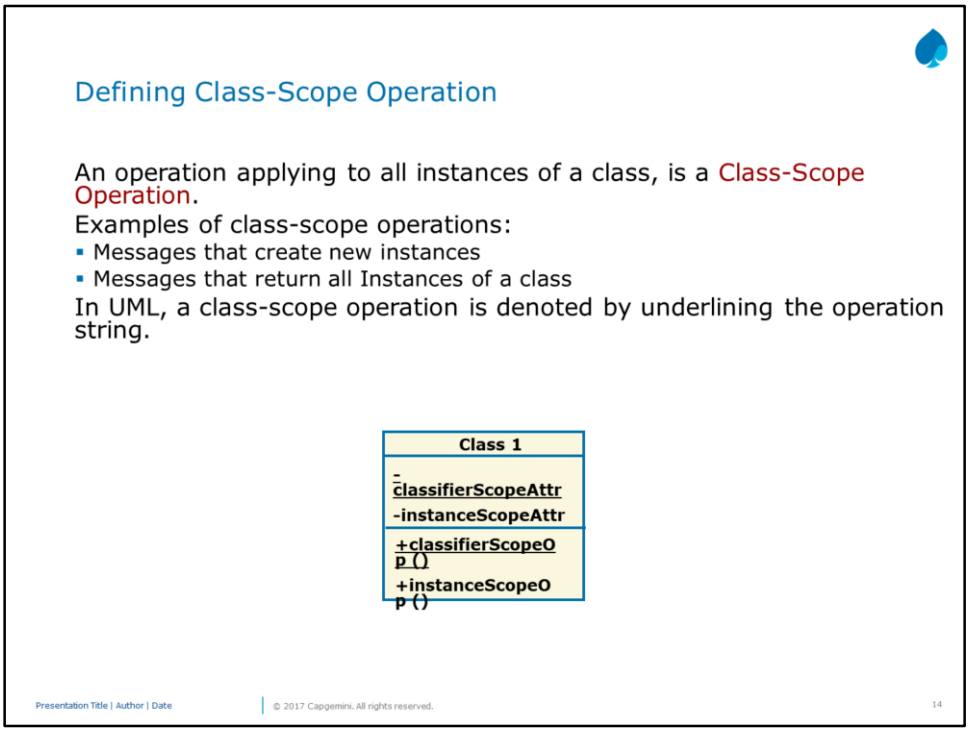


Operation Visibility and UML:

- In UML, you can specify the access that clients have to attributes and operations.
- **Export control** is specified for attributes and operations by preceding the name of the member with the following symbols:
  - \+ Public
  - \# Protected
  - \- Private

Instructor Notes:

Class Scope is what most would know as Static operations.

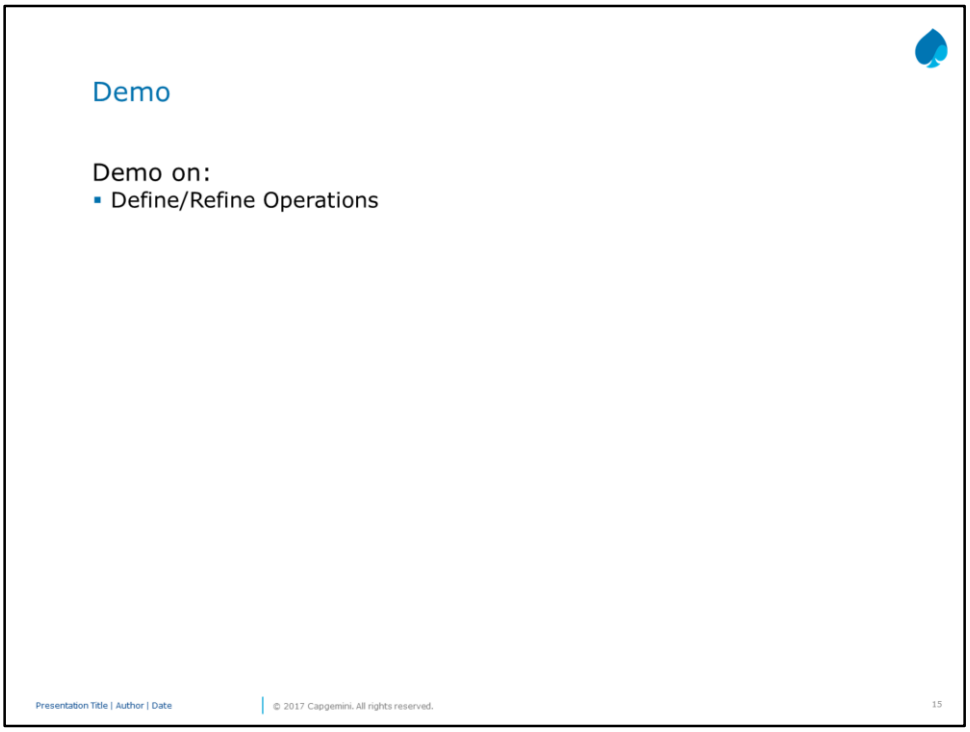


Defining Class-Scope Operation:

- For the most part, **operations** are **instance operations**. That is, operations are performed on instances of the class.
- However, in some cases, an operation applies to all instances of the class. Therefore, it is a **class-scope operation**.
  - The class operation receiver is actually an instance of a metaclass - the description of the class itself - rather than any specific instance of the class.
  - Examples of class operations include messages that create (instantiate) new instances, which return all instances of a class, and so on.
- The operation string is underlined to denote a class-scope operation.

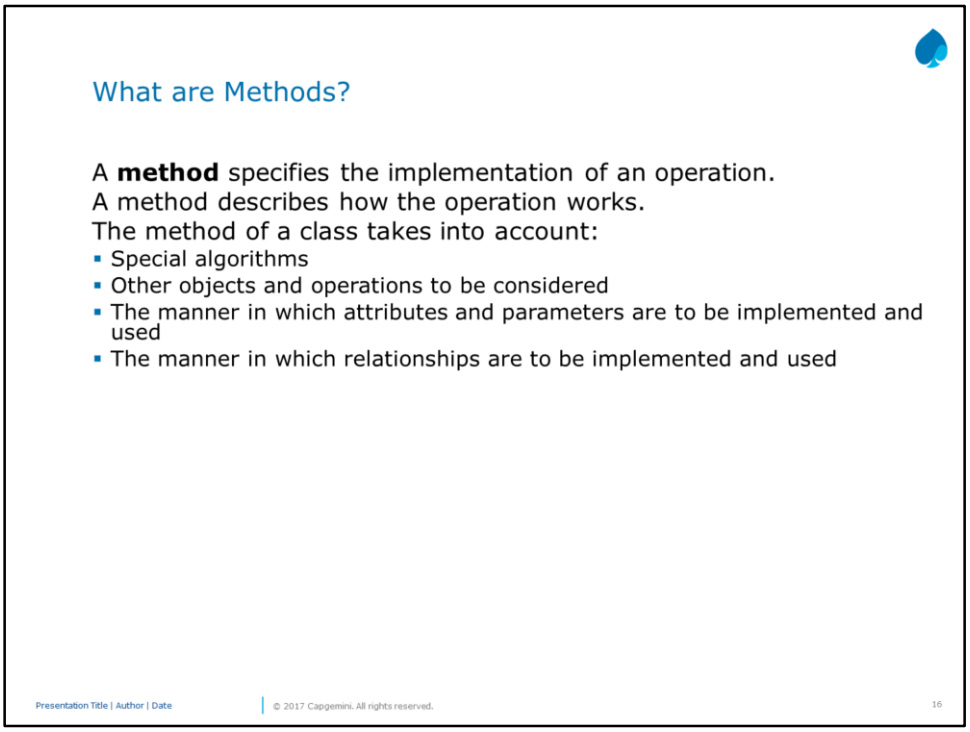
Instructor Notes:

Show how parameter and return details are specified using the tool.



Instructor Notes:

Operations and Methods do tend to get used inter-changeably, but important to note that a method gives the “How” of an operation…specifics of algorithm or any other such details.

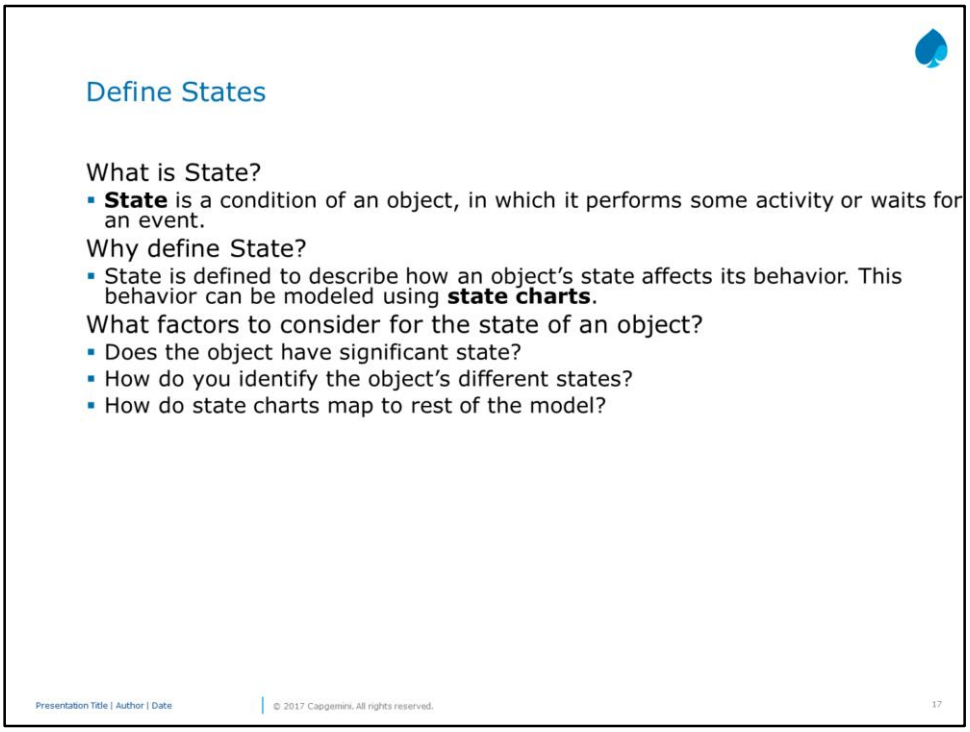


What are Methods?

- A **method** specifies the implementation of an operation.
- Suppose the implementation of an operation requires the use of a specific **algorithm** or more **information** than is presented in the operation’s description. Then a separate **method description** is required.
- The **method** describes how the operation works, and not just what it does.

Instructor Notes:

Introduce State Chart Diagrams through this section.

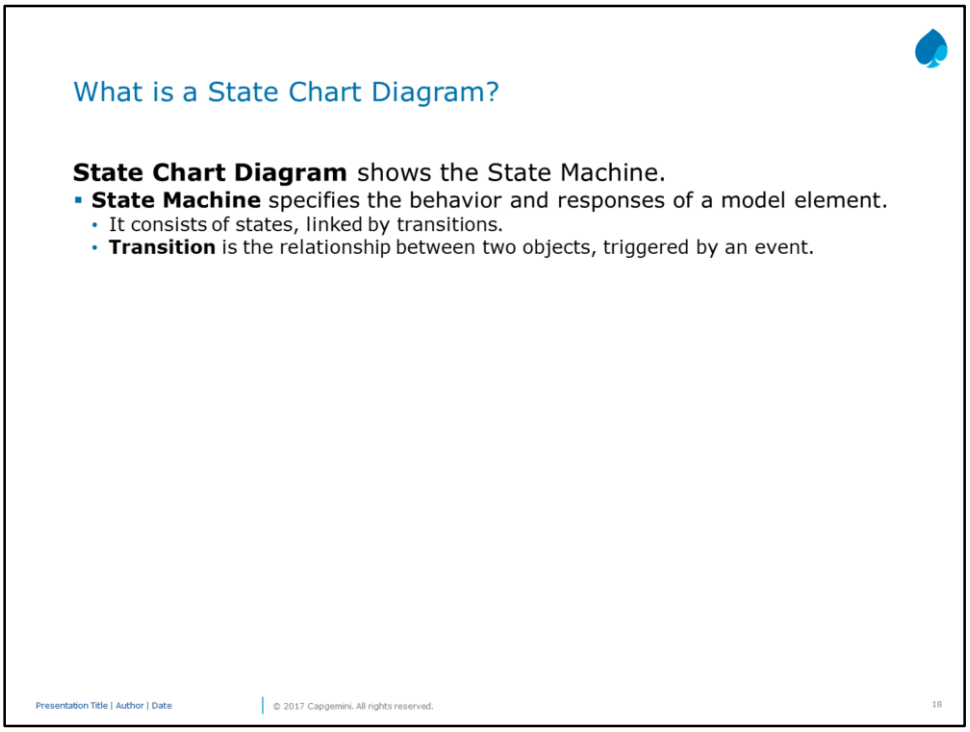


Define States:

- For some **operations**, the behavior of the operation depends upon the **state** that the receiver object is in.
- A **state machine** is a tool that describes:
  - The **states** that an object can assume
  - The **events** that cause the object to move from one state to another
- An object that has more than one computational state is said to be **state-controlled**.

Instructor Notes:

DO we need to model state chart diagrams for all classes? No – only where state will have an impact on the behavior.



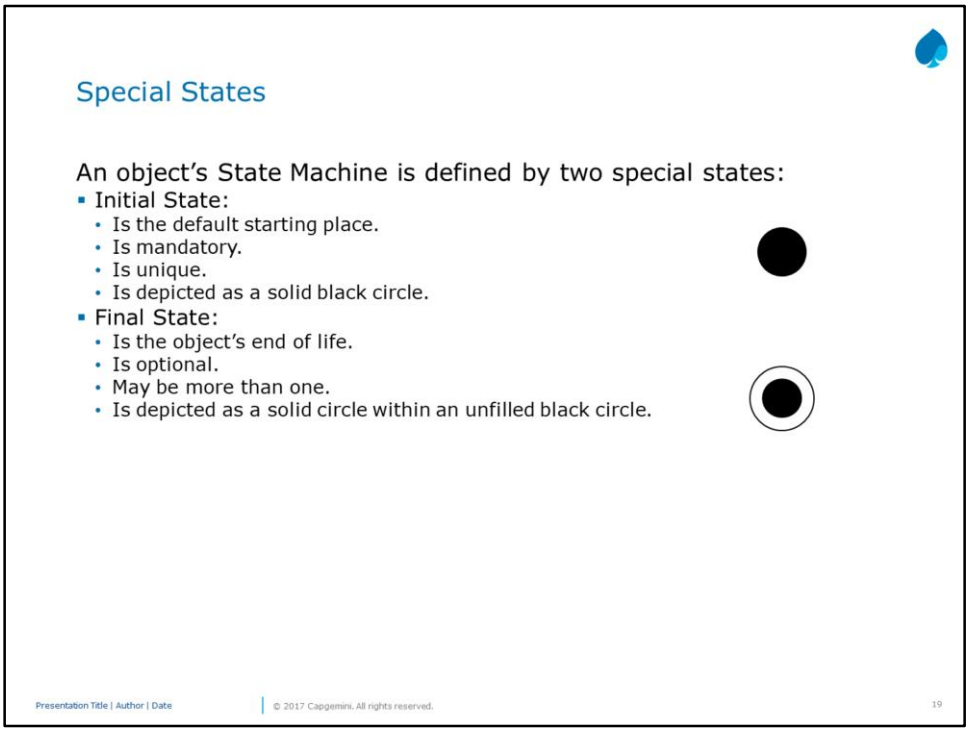
What is a State Chart Diagram?

- A **state chart diagram** describes the **state machine**.
- The **state machine** describes:
  - **States** that an object can exist in (defined by non-overlapping constraints).
  - **Factors** (events or signals, received from other objects) that cause an object to change from one state to another.
  - **Resultant changes** that occur as a consequence of this transition.
  - **Guard condition**, which is a boolean expression of attribute values, that allows a transition only if the condition is true.
  - **Action**, which is an atomic execution that results in a change in state, or the return of a value.
  - **Activity**, which is a non-atomic execution within a state machine.
  - **Protocols**, which are legal sequences of operation calls of a class or interface or of interactions by signals.

- A **state machine** is attached to exactly one **class** and can be inherited by **subclasses**.

Instructor Notes:

We have already seen some of these notations as part of the Activity Diagram. Activity Diagram is a special form of state chart diagram and hence the similarity in notations.

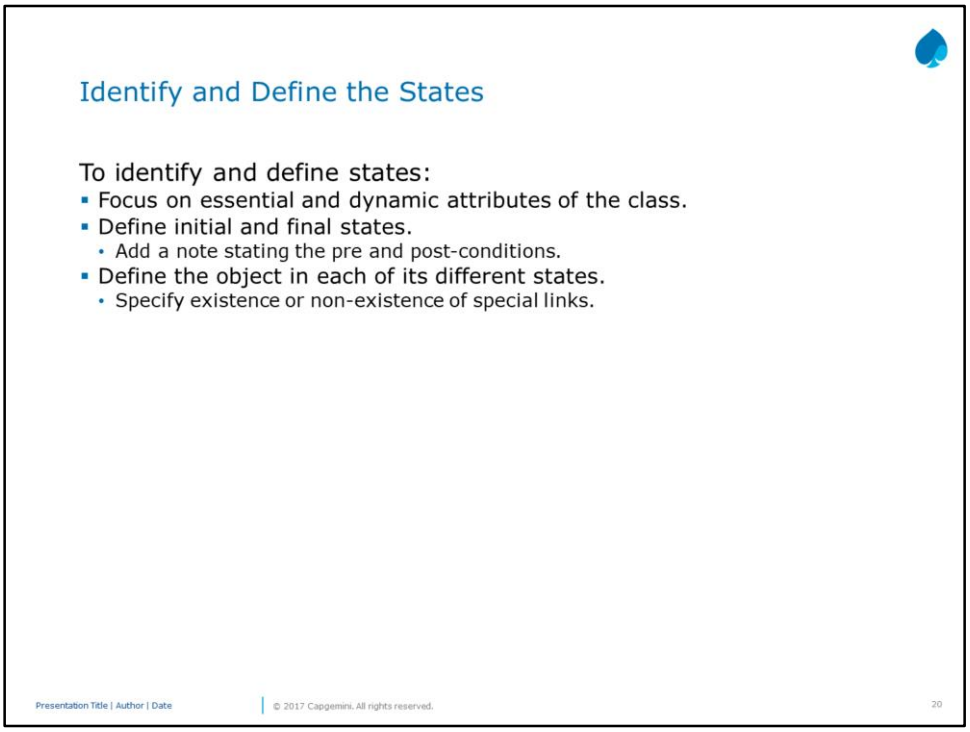


<u>Special States</u>:

- There are two special states that may be defined for an object's state machine. They are initial state and final state.
  - The **initial state** indicates the default starting place for the state machine or substate. It is the point when the object is created. After the initial state the object begins changing states. Conditions based on the activities can determine the next state that the object transitions to.
    An initial state is depicted as a filled black circle.
  - The **final state** indicates the completion of the execution of the state machine or that the enclosing state has been completed.
    A final state is represented as a filled black circle surrounded by an unfilled circle.
- Initial and final states are really **pseudostates**. Neither of them may have the usual parts of a normal state, except for a name. A transition from an initial state to a final state may have the full complement of features, including a **guard condition** and an **action**. However, it may not have a trigger event.
- State diagrams can be nested hierarchically, indicating **sub state machines**.
  - Entering a sub state machine begins at the starting state of the sub machine.
  - Reaching an end state means leaving a sub state.

Instructor Notes:

Here are guidelines for identifying states

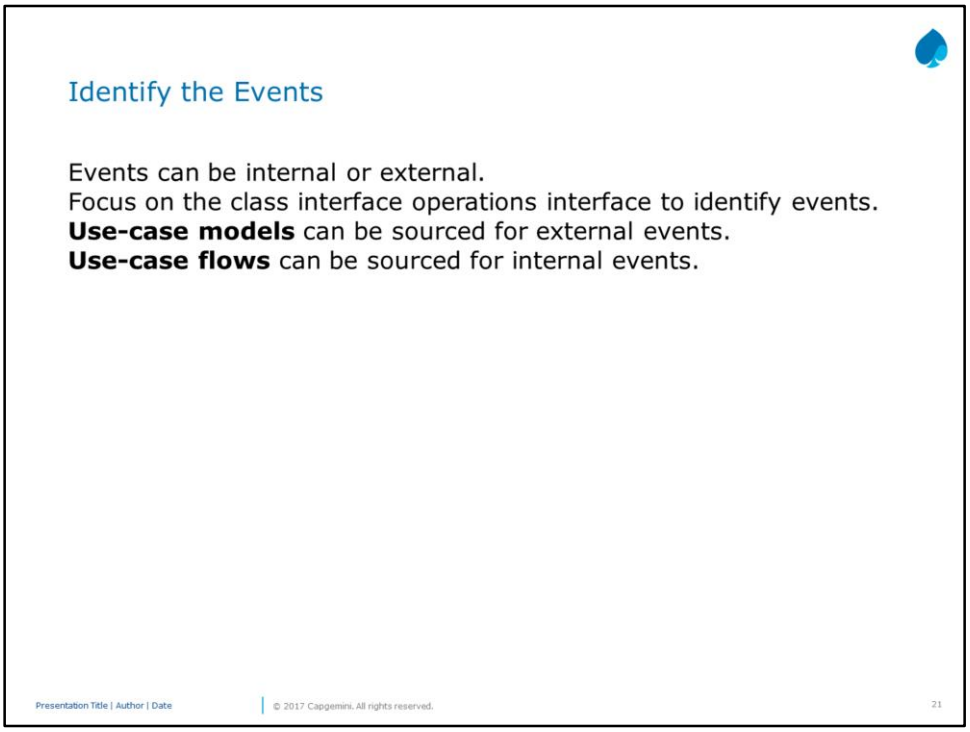


<u>Identify and Define the States</u>:

- **Class attributes** and **relationships** are a good starting point for identifying states. An object may have a **simple** or a **complex state**. More so, an object may exist in more than one state.
- When considering a particular object:
  - Make sure to identify the various states it is likely to exist in.
  - Provide explicit descriptions for every state identified. This removes any kind of ambiguity.
- First, identify the initial and final states of the object under consideration. Second, define all pre-conditions and post-conditions, for the initial and final states.

Instructor Notes:

Having identified states, identify the events which lead to change in states

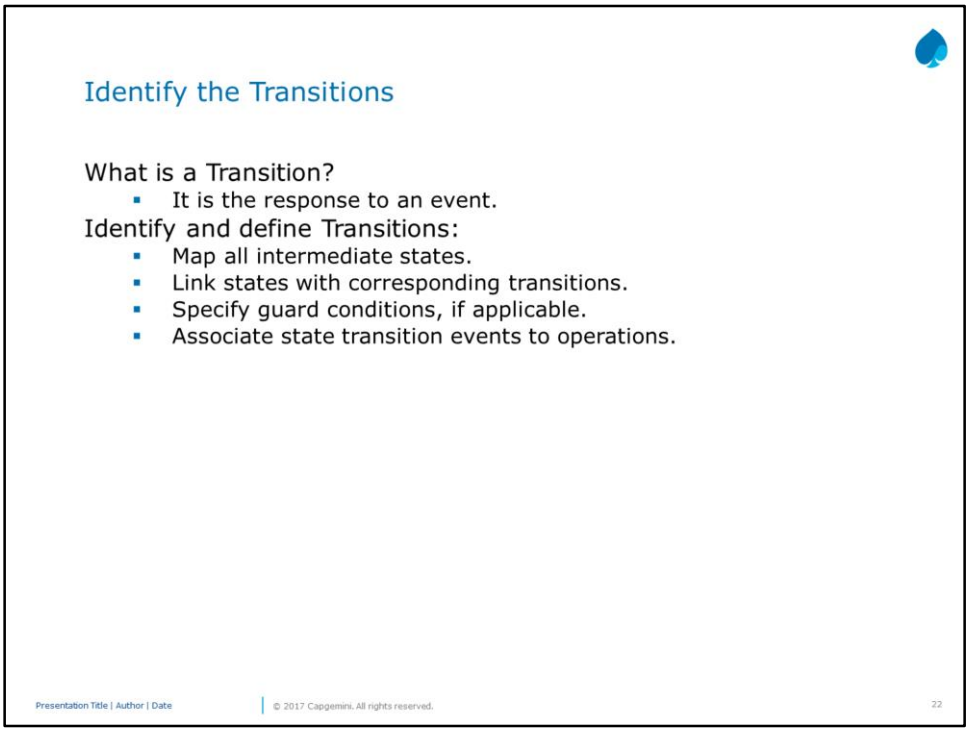


Identify the Events:

- In the context of the **state machines**, the **events** model the occurrence of a stimulus, which may trigger a **state transition**.
  - Events include signals, calls, the passage of time, or a change in state.
  - Events may be synchronous or asynchronous.
- You can identify **events** that trigger a response from the object, by looking at the object's **interfaces** or **protocols**. A class has to respond to all incoming messages for all instances on the various interaction diagrams. These messages normally correspond to **operations** on the associated classes. Therefore the **class interface operation** is a reliable source for identifying events.
- **Transitions** and their **actions** are triggered by events. An event is a phenomenon in space and time significant for the modeled system. An event can trigger state changes.

Instructor Notes:

The events will initiate state transitions.

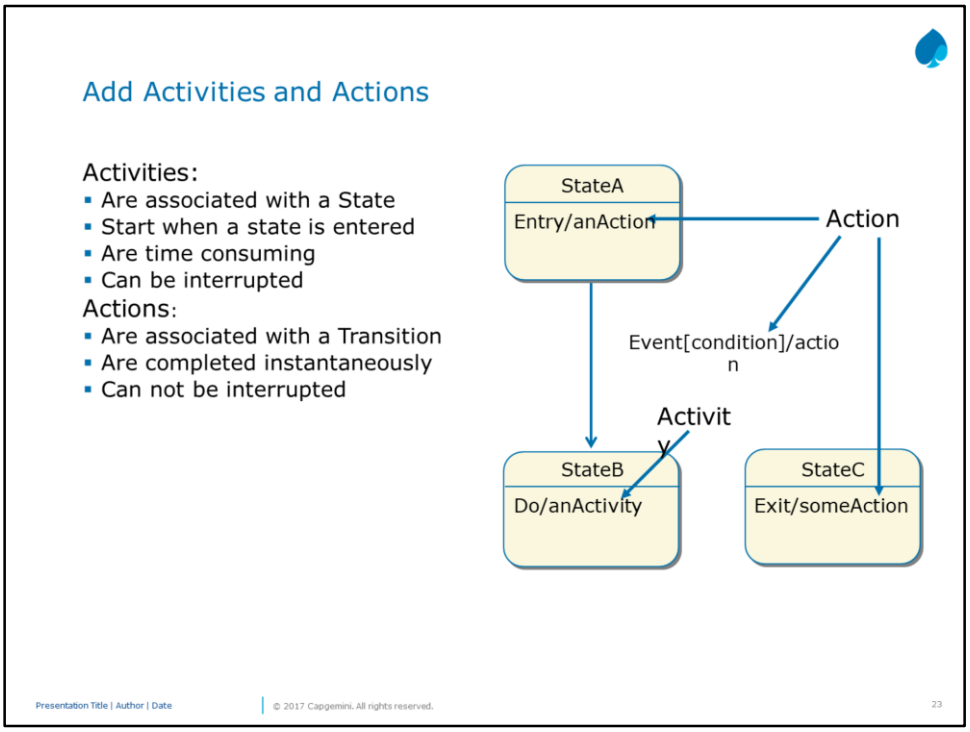


Identify the Transitions:

- A **transition** is a relationship between two states, indicating that an object in the first state will perform certain actions and enter a second state when a specified event occurs and a specified condition is satisfied. On such a change of state, the transition is said to “fire”. Until the transition fires, the object is said to be in the “source” state. After it fires, it is said to be in the “target” state.
- A transition can include **triggering event**, a **guard**, and **actions** to be executed. Transitions without events and guards are executed immediately. When an activity is finished, all the sub states are passed through, respectively.
- Given below is the description of the various elements associated with a state transition:
  - **Event Trigger**: The event that makes the transition eligible to fire, when received by the object in the source state.
  - **Guard Condition** is a boolean expression that is evaluated when the transition is triggered by the reception of the event trigger. If the expression evaluates to be True, then the transition is eligible to fire. However, if the expression evaluates False, then the transition does not fire. Therefore guards may prevent transition from taking place. The guards are modeled by boolean conditions.
  - **Target State**: It is the state that is active after the completion of the transition.

Instructor Notes:

Participants usually have confusions around this, and hence this needs more explanation at times.



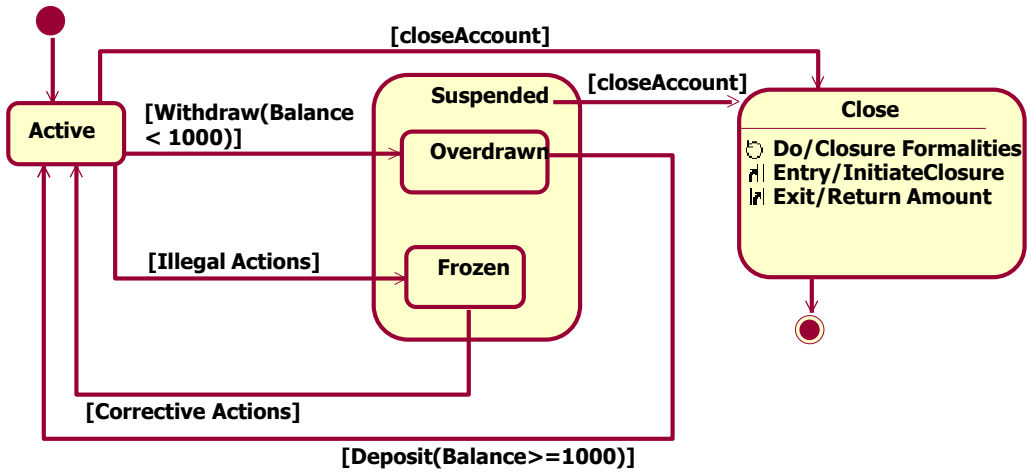
<u>Add Activities and Actions</u>:

- **Activities** are associated with the **state** of an object. The activity begins as soon as the associated state is entered into. An activity can run to completion or can be interrupted before completion. Some states are defined for the sole purpose of executing a particular activity. On completion the state undergoes automatic transition.
- **Actions** are executed within a very short period of time. Once initiated they cannot be interrupted. Actions always run to completion. Actions are shown inside the **state** icon and are preceded by the keyword “entry” or “exit”.

Instructor Notes:

Ideally, we could have "Open" as initial state too!

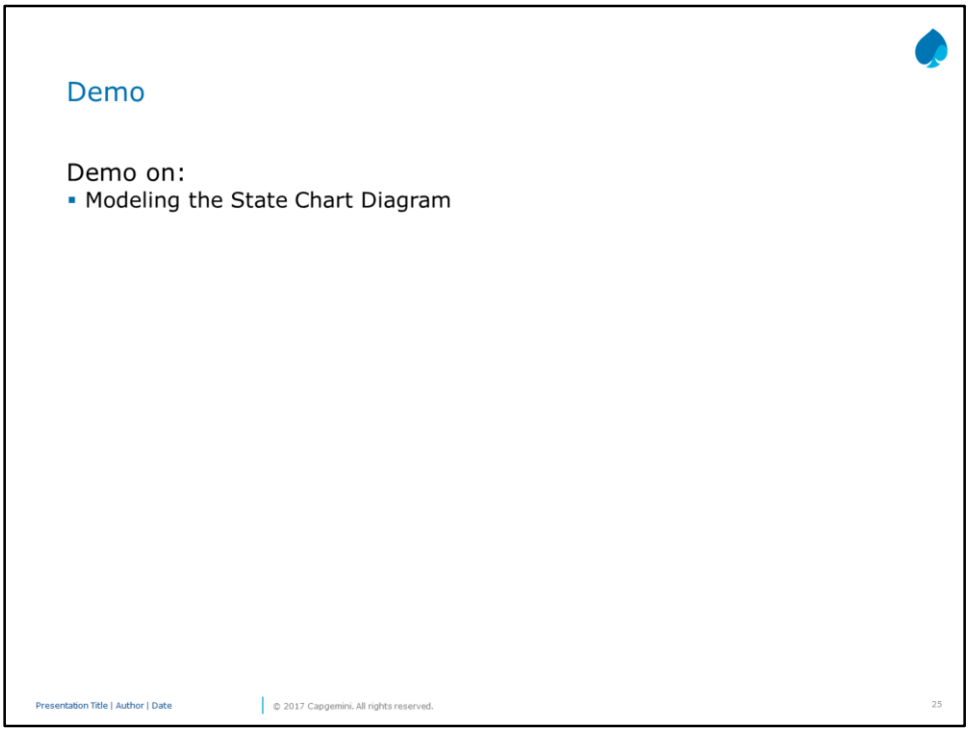
Example: State Chart Diagram:



The above state chart diagram shows some states for an Account Object. Note that Suspended state is a nested state, within which the Account object can be in an overdrawn state or frozen state. Entry, Exit, and Do actions / activities for a state are given for the Close State.

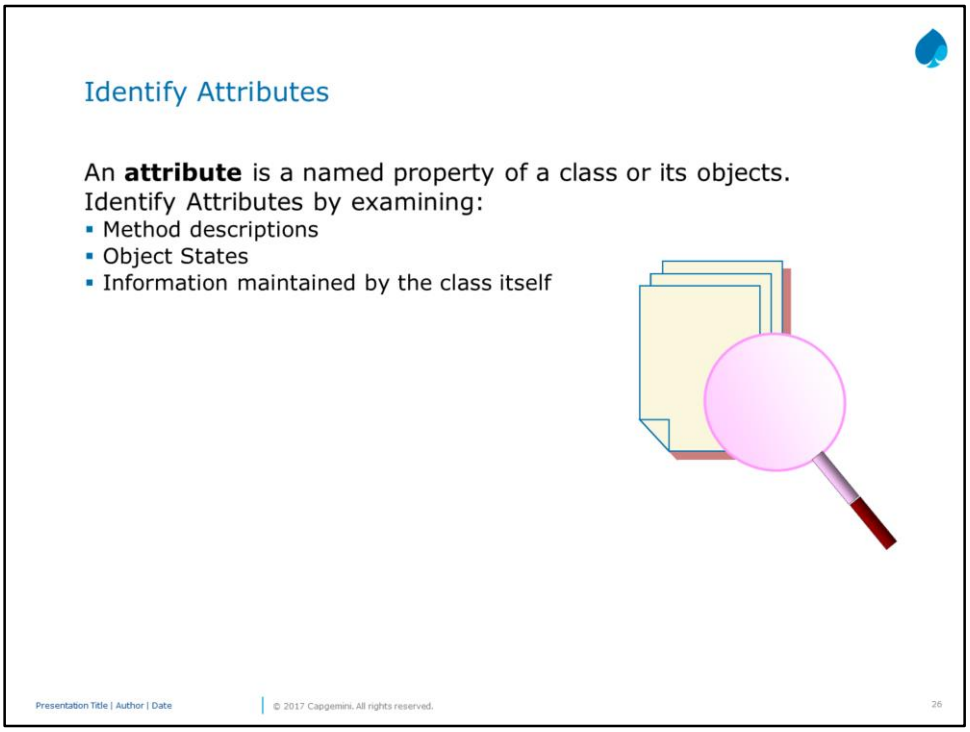
Instructor Notes:

Use the tool to model the state chart diagram.



Instructor Notes:

Guidelines for identifying attributes.

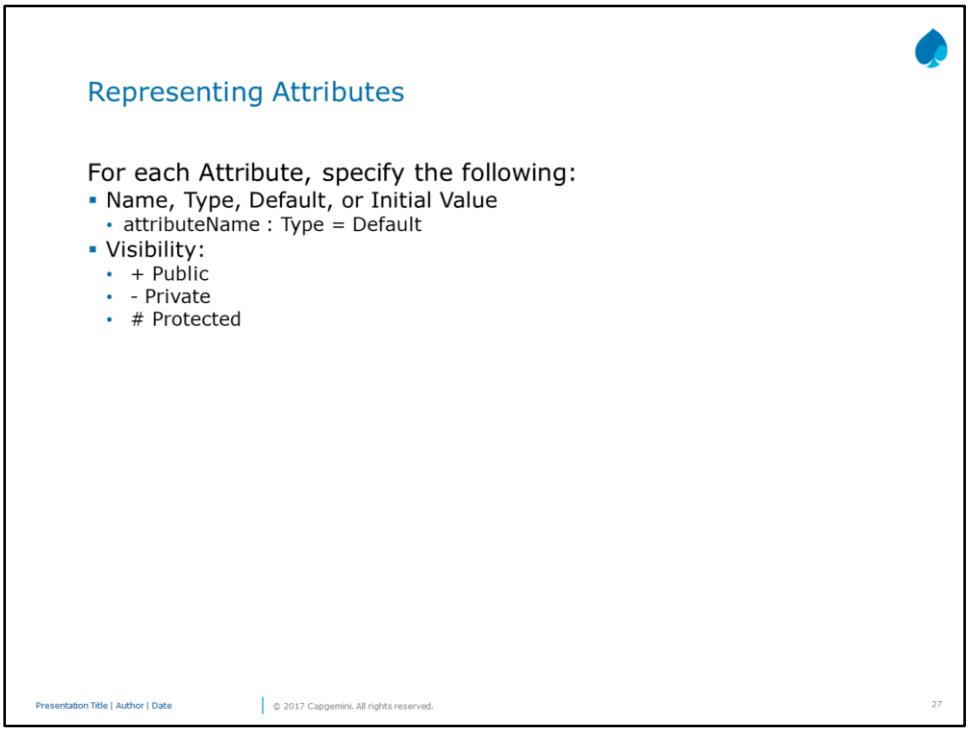


Identify Attributes:

- An **attribute** is a named property of an object. The attribute name is a noun that describes the attribute's role in relation to the object. An attribute can have an initial value when the object is created.
- You should model **attributes** only if doing so makes an object more understandable. You should model the property of an object as an attribute only if it is a property of "that object alone". Otherwise, you should model the property with an association or aggregation relationship to a class whose objects represent the property.
- During the definition of **methods** and the identification of **states**, attributes needed by the class to carry out its operations are identified. Attributes provide information storage for the class instance and are often used to represent the state of the class instance. Any information that the class itself maintains is done through its **attributes**.
- You should check to make sure that all attributes are needed. Extra attributes, multiplied by thousands or millions of instances, can have a detrimental effect on the performance and storage requirements of a system.

Instructor Notes:

Just as operations were completely described earlier, we need to do the same for attributes too.



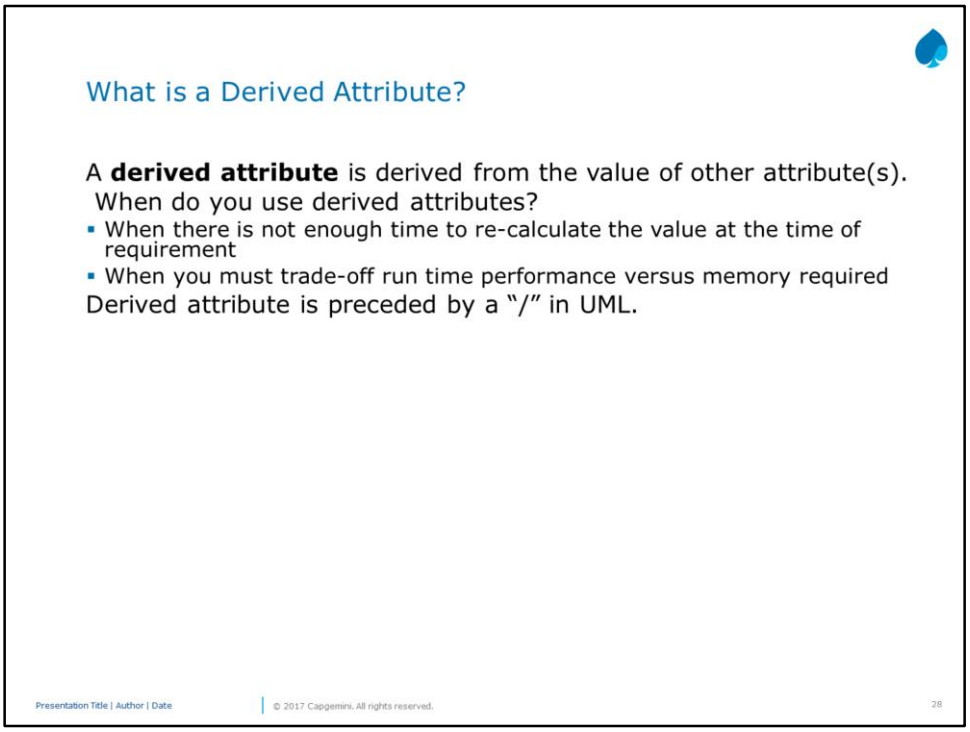
Representing Attributes:

For each attribute, you need to define:

- Its **name**, which should follow the naming conventions of both the implementation language and the project.
- Its **type**, which will be an elementary data type supported by the implementation language.
- Its **default or initial value**, to which it is initialized when new instances of the class are created.
- Its **visibility**, which will take one of the following values:
  - **Public**: The attribute is visible both inside and outside of the package containing the class.
  - **Protected**: The attribute is visible only to the class itself, to its subclasses, or to friends of the class (language-dependent).
  - **Private**: The attribute is only visible to the class itself and to friends of the class.
  - **Implementation**: The attribute is visible only to the class itself.

Instructor Notes:

Derived attributes are generally introduced due to performance reasons.

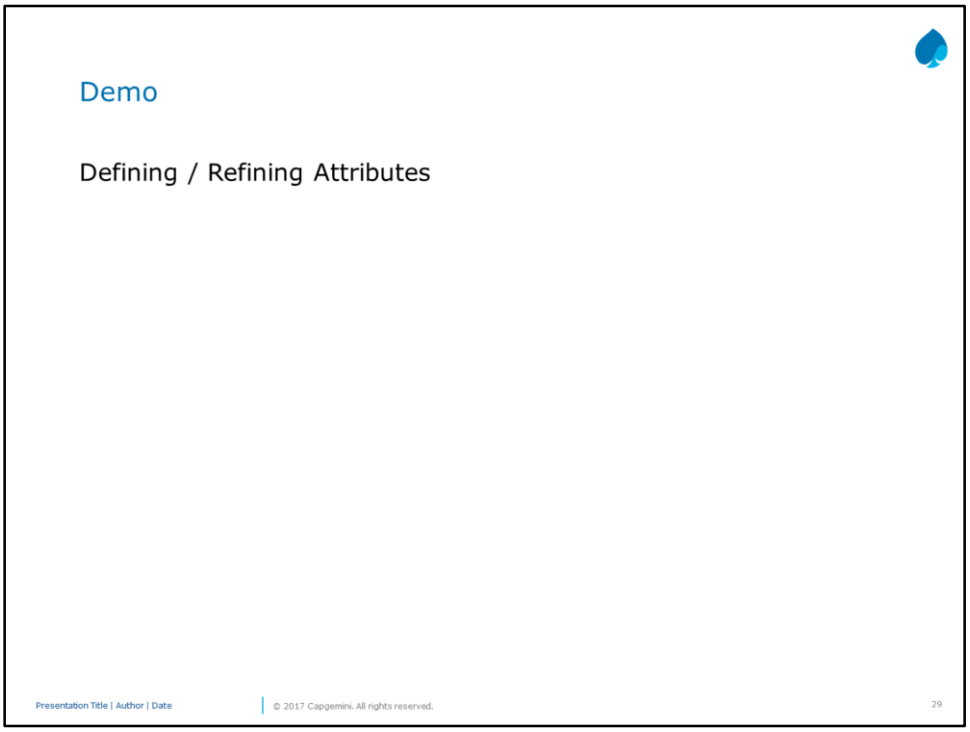


What is a Derived Attribute?

- **Derived attributes** and **operations** are used to describe a dependency between attribute values that must be maintained by the class.
- They do not necessarily mean that the attribute value is always calculated from the other attributes.

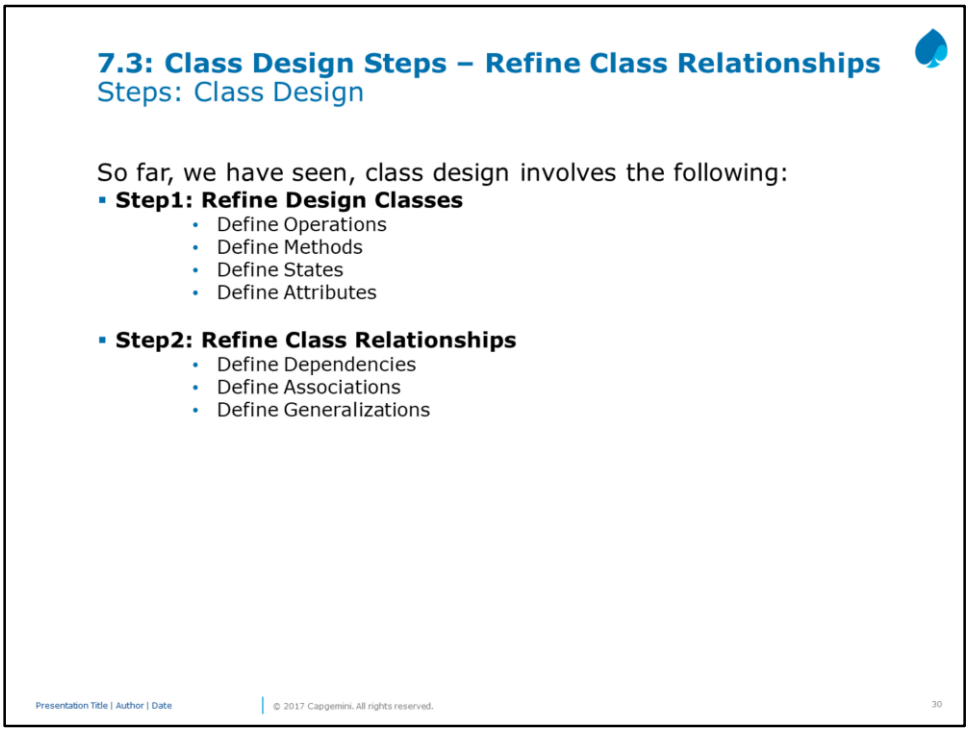
Instructor Notes:

Use the tool to do this.



Instructor Notes:

We have already arrived at set of design classes. We review them and refine them to include implementation specific aspects; and also refine class relationships.



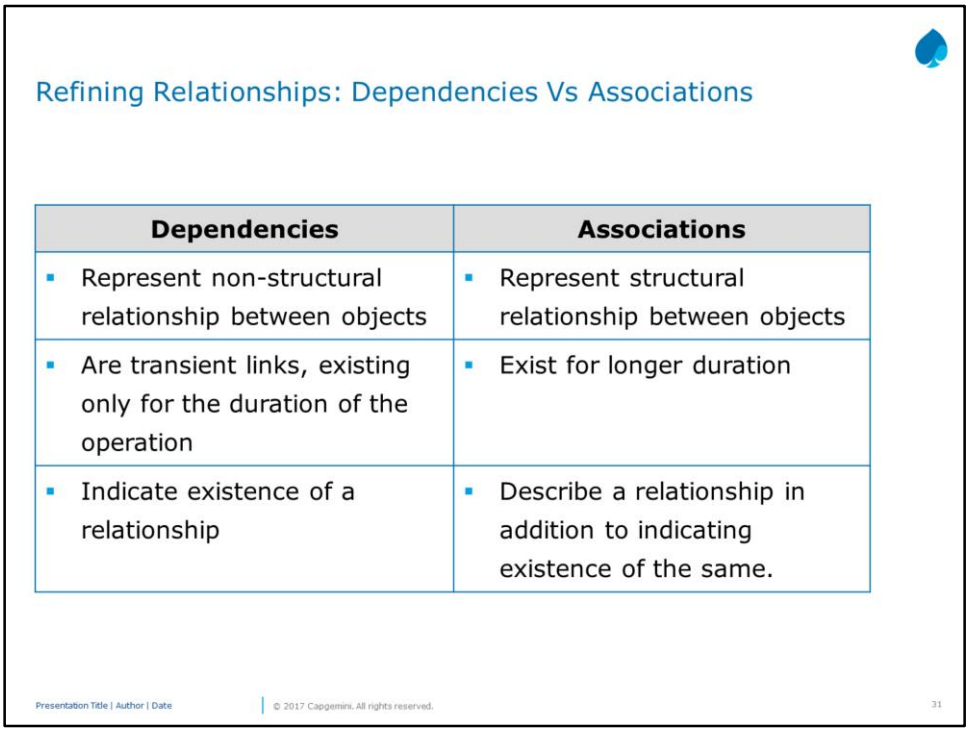
Steps: Class Design:

- To reiterate:
  - As part of LLD, we have arrived at one or more **design classes** for the **analysis classes** created earlier. Now, these initial design classes go through various modifications and refinements.
  - We identify additional **operations** on them. This includes:
    - naming and describing the operations
    - defining operation visibility
    - defining class operations
  - After defining the operations, we define **methods** specifying their (operation's) implementation.
  - At times, the **behavior** of an operation depends on the **state** of the receiver object. We need to develop the **state machine**, which defines states that an object can assume and the events that cause the object to move from one state to another.
  - Subsequently, we include/refine the attributes.
  - **Relationships** (between class) identified in previous phases need to be refined. Hence we look for **dependencies**. After defining dependencies, you need to focus on the remaining **associations**. Some of these can get modeled as **aggregation** or **composition**. Finally, organize design classes, having common behavior and structure, into a **generalization hierarchy**.
  - Each of these steps are described in the subsequent slides.

Instructor Notes:

We have all relationships modeled as Association so far. We now need to refine the class relationships.

Recap of UML relationships. We have discussed this before.

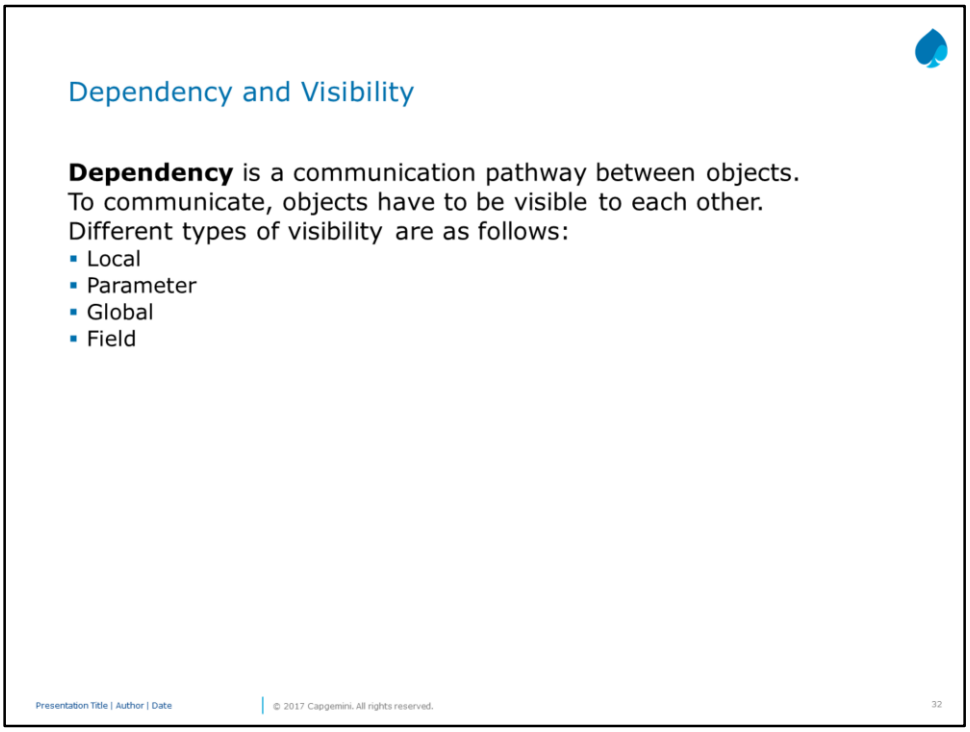


Refining Relationships: Dependencies vs. Associations:

- **Associations** modeled during **analysis** phase have to be refined further.
- Now, in the **design** phase, the actual **communication path** has to be defined. All **dependencies** have to be identified.
  - Dependencies are semantic relationship between elements, in which a change in the supplier may cause a change in the client.
  - Dependency is a kind of pathway between two objects, on which messages are exchanged. It is transient in nature, and occurs when the visibility between the objects is global, parameter, local, or field.
- You have to carefully check each association to identify candidates ideally suited for being modeled into a dependency.

Instructor Notes:

Recap of UML relationships. We have discussed this before.



Dependency and Visibility:

What is Visibility?

- **Visibility** is the ability of one object to see or have reference to another.

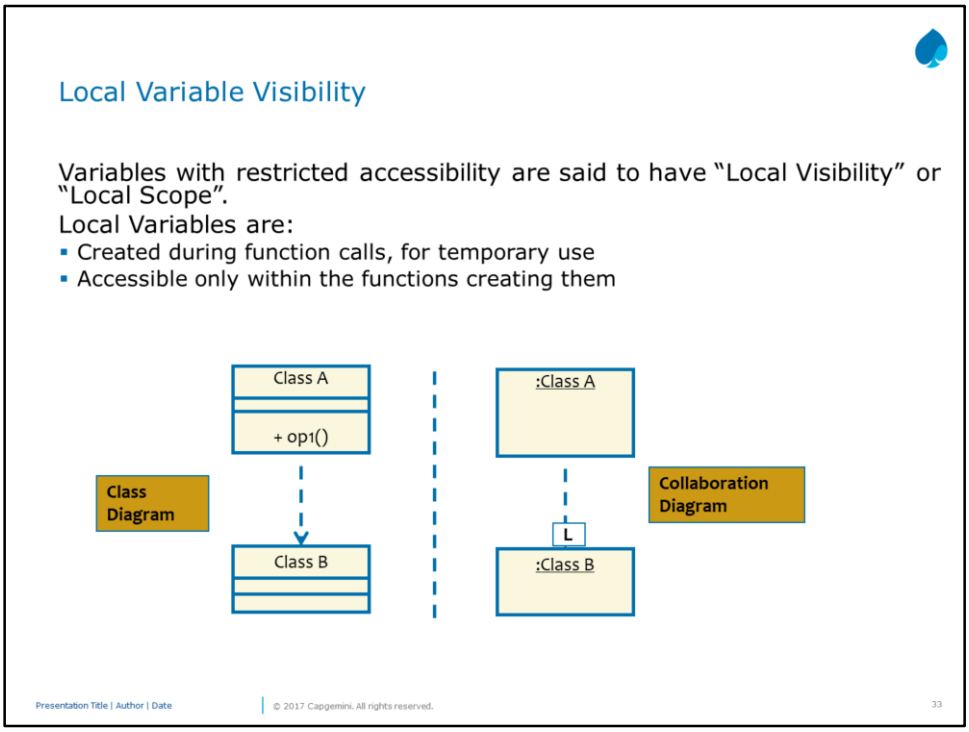
When do you need to identify the visibility of the supplier object?

- When you need to send a message from one object to another, the receiver object has to be visible to the sender. Hence the sender has to have a pointer to the receiver.
- For Example, for an object A to send messages to an object B, the object B must be visible to object A.

- Depending on the visibility of the receiver object, the following four types of communication pathways can be identified:
  - **Global**: The supplier object is a global object. (B is in some way globally visible)
  - **Parameter**: The supplier object is a parameter to, or the return class of, an operation in the client object. (B is a parameter of a method of A)
  - **Local**: The supplier object is declared locally (that is, created temporarily during execution of an operation). (B is a local object in a method of A)
  - **Field**: The supplier object is a data member in the client object. (B is an attribute of A)
- If the **visibility factor** between two associated objects is local, global, or parameter, then the **association relationship** can be modeled as a **dependency**.

Instructor Notes:

Recap of UML relationships. We have discussed this before.

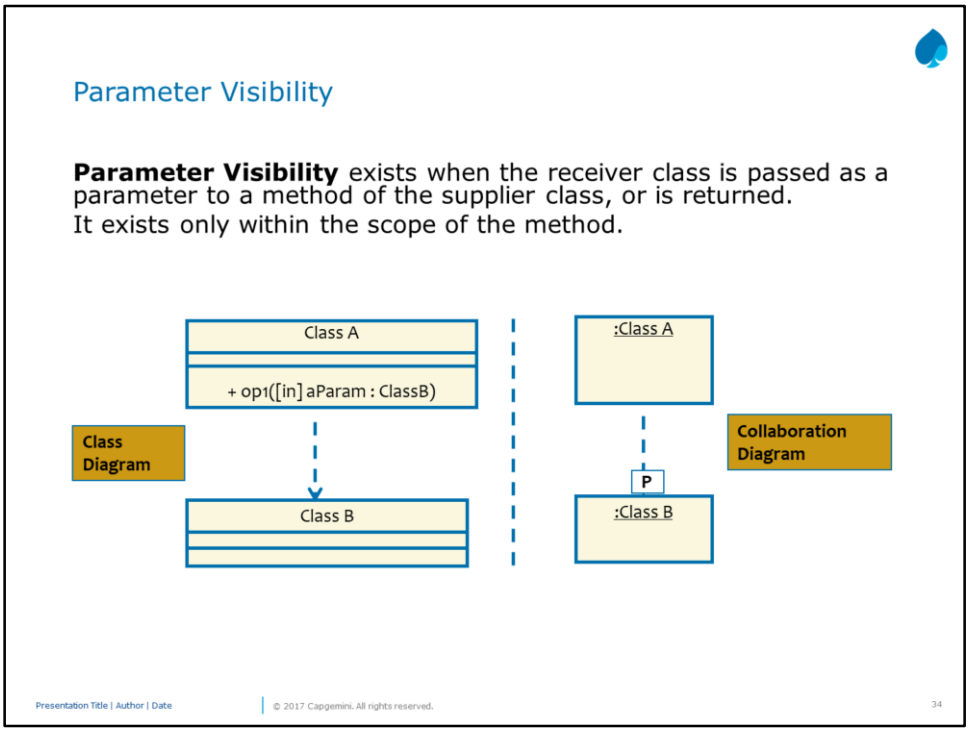


Local Variable Visibility:

- If the supplier object (variable) is declared locally, that is created temporarily during the execution of an operation, then the **class diagram** should show a dependency between the two classes.
- If a **collaboration diagram** is being used, then qualify the transient link as "local" with the letter "L".
- In the example in the above slide, the op1() operation contains a local variable of the type ClassB.

Instructor Notes:

Recap of UML relationships. We have discussed this before.

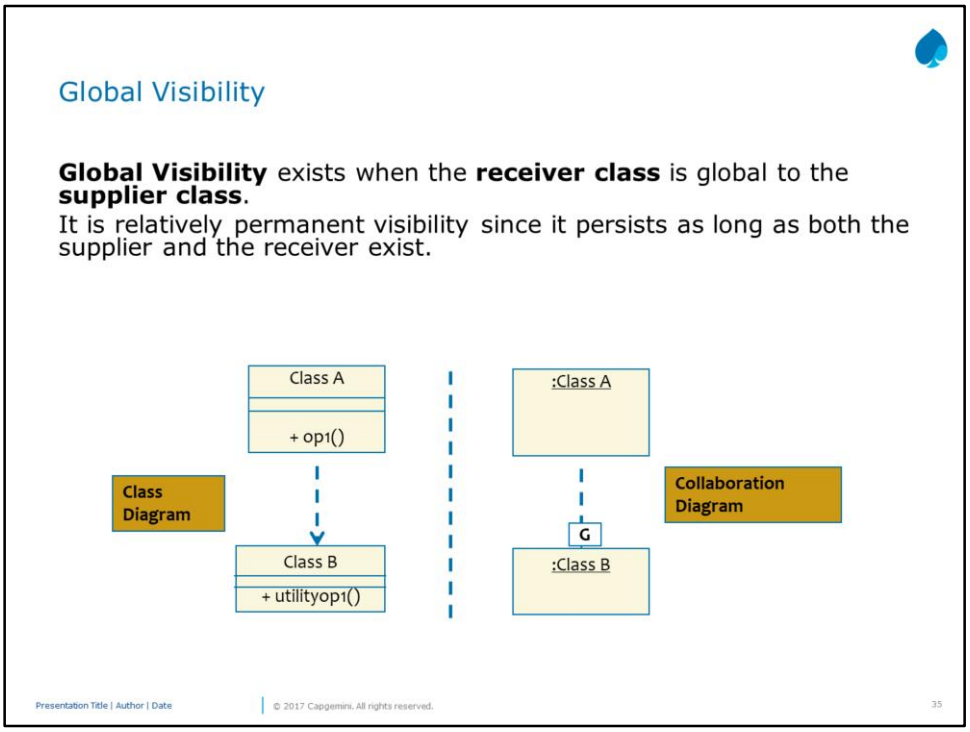


Parameter Visibility:

- If the receiver object is passed as a parameter to a method of the supplier, then a dependency has to established between the supplier and the receiver classes in the **class diagram** containing the two classes. If a **collaboration diagram** is being used, then qualify the transient link as “parameter” with the letter “P”.
- This is a frequently used form of visibility that is used in Object-Oriented programming.
- The “**P**” on the link in the associated collaboration diagram indicates that the supplier object is visible to the client object because it is a **parameter** for one of the **client’s operations**. This type of relationship is modeled as a dependency if this **parameter visibility** link is the only type of link that exists between the two classes.

Instructor Notes:

Global may not be supported by all OO Programming Languages

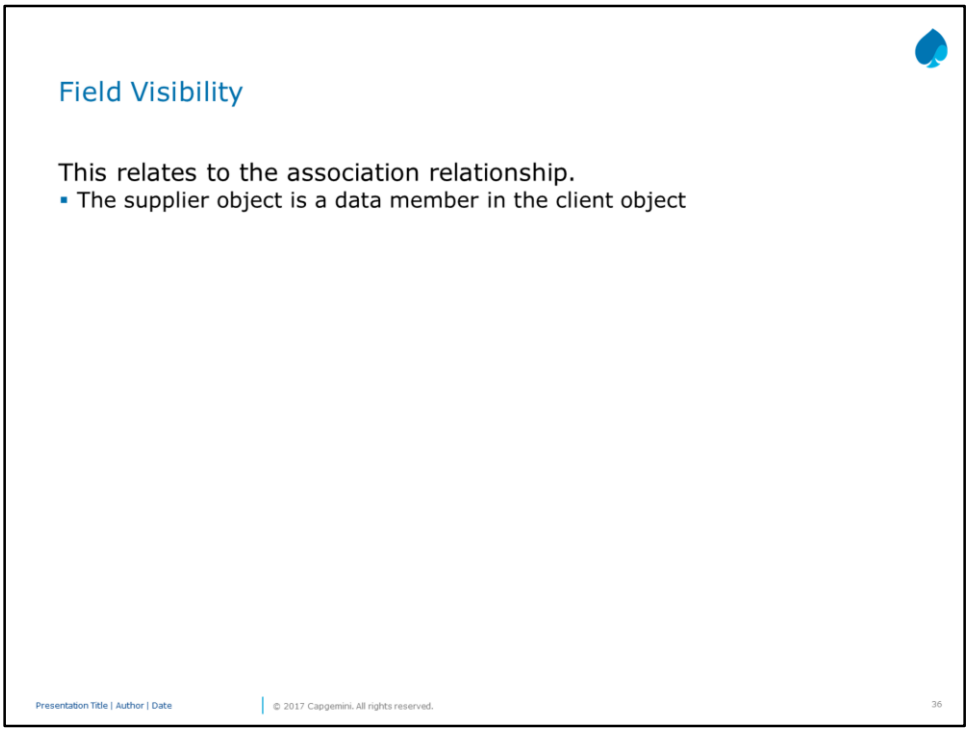


Global Visibility:

- If the receiver is global with respect to the supplier class, then you need to define a dependency in the class diagram containing both the classes.
- If the relationship is being modeled using a collaboration diagram, then you have to qualify the link as **global**, with the letter "**G**".
- **Global visibility** is a relatively permanent visibility since it persists as long as the two classes exist.
- The "**G**" on the link in the associated collaboration diagram indicates that the **supplier object** is **global** to the client object. This type of relationship is modeled as a dependency if this **parameter visibility link** is the only type of link that exists between the two classes.

Instructor Notes:

What does not get into any of the earlier dependency refinements will remain association.

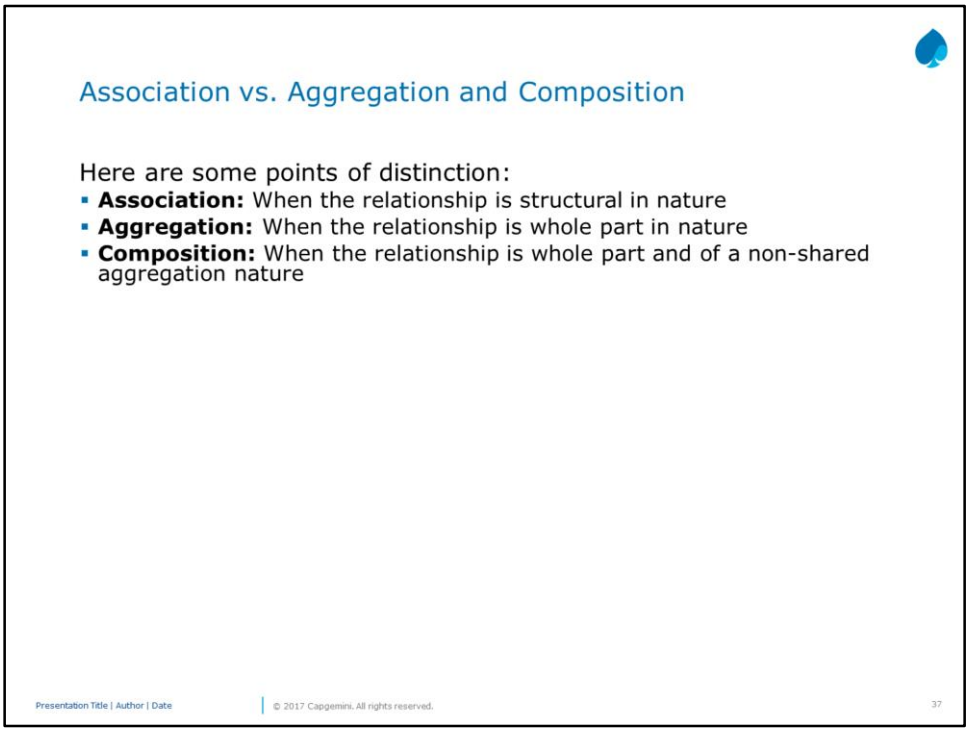


Field Visibility:
In case of field visibility, client “structurally” needs to know about the supplier. This relates to an association relationship.

Instructor Notes:

We now consider these Association relationships and consider whether they need to be containment, and is yes, what form of containment.

Recap of UML relationships. We have discussed this before.

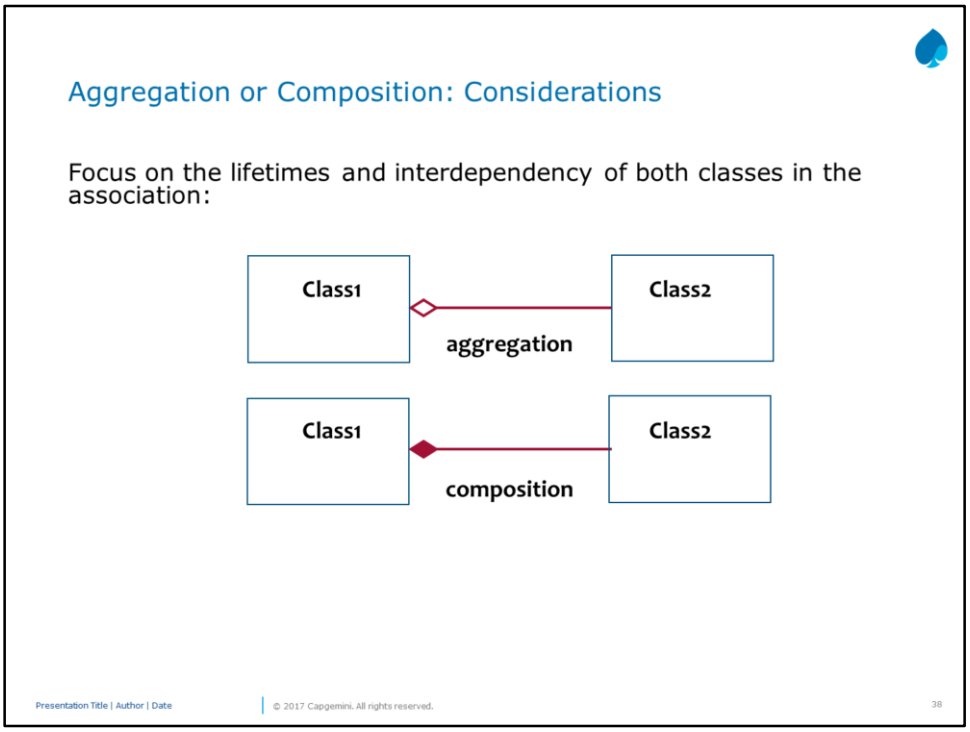


Association vs. Aggregation and Composition:

- **Association** represents the ability of one instance to send a message to another instance. This is typically implemented with a pointer or reference instance variable, although it might also be implemented as a method argument, or the creation of a local variable.
- You have identified class relationships, that can be modeled as dependencies. Now, you need to look at the remaining associations and refine them.
- You may also need to define associations in support of method descriptions, that have been defined earlier.
- **Aggregation** is the typical whole/part relationship. Aggregation is used to model a compositional relationship between model elements. The part can exist without the whole. There are many examples of compositional relationships, for example a **Library** contains **Books.**
- **Composition** is a special kind of aggregation, where the part is physically included in the whole. It is often referred to as a **composed aggregation**. Once created, a part lives and dies with the whole. In fact one can say that the lifetime of the “part” is controlled by the “whole”. This control may be direct or transitive. That is, the “whole” may take direct responsibility for creating or destroying the “part”, or it may accept an already created part, and later pass it on to some other whole that assumes responsibility for it.
- The important thing to note in a composed aggregation is that, the part can only belong to only that whole.

Instructor Notes:

Recap of UML relationships. We have discussed this before.

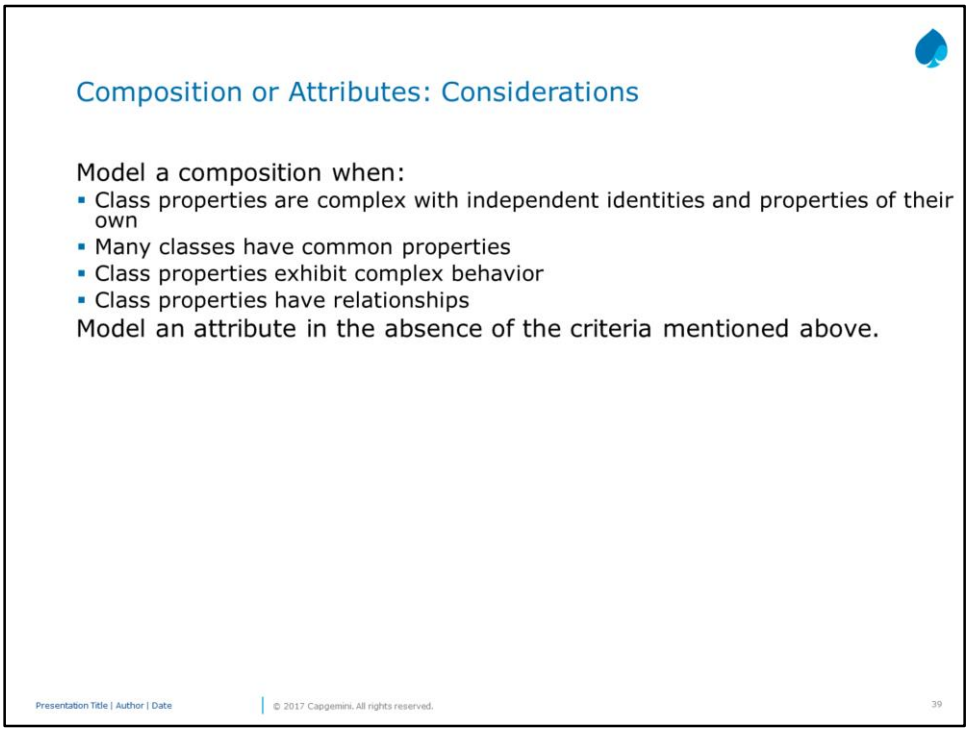


Aggregation or Composition: Considerations:

- **Aggregation** is similar to composition, but is a less rigorous way of grouping things. A department has several employees, but a department may continue to exist even if the employees leave.
- **Composition** indicates that one class belongs to the other. A polygon is made up of several points. If the polygon is destroyed, so are the points.
- You need to carefully consider the nature of the association between the two classes. Identify lifetime of each and their interdependency.
  - If the interdependency is very strong and the aggregate is incomplete without the component parts, then model the association an aggregation.
  - Composition should be used when the whole and part must have coincident lifetimes.
- Selection of aggregation or composition will determine how object creation and deletion are designed.

Instructor Notes:

Address is a good example for Class Composition v/s Attribute discussion

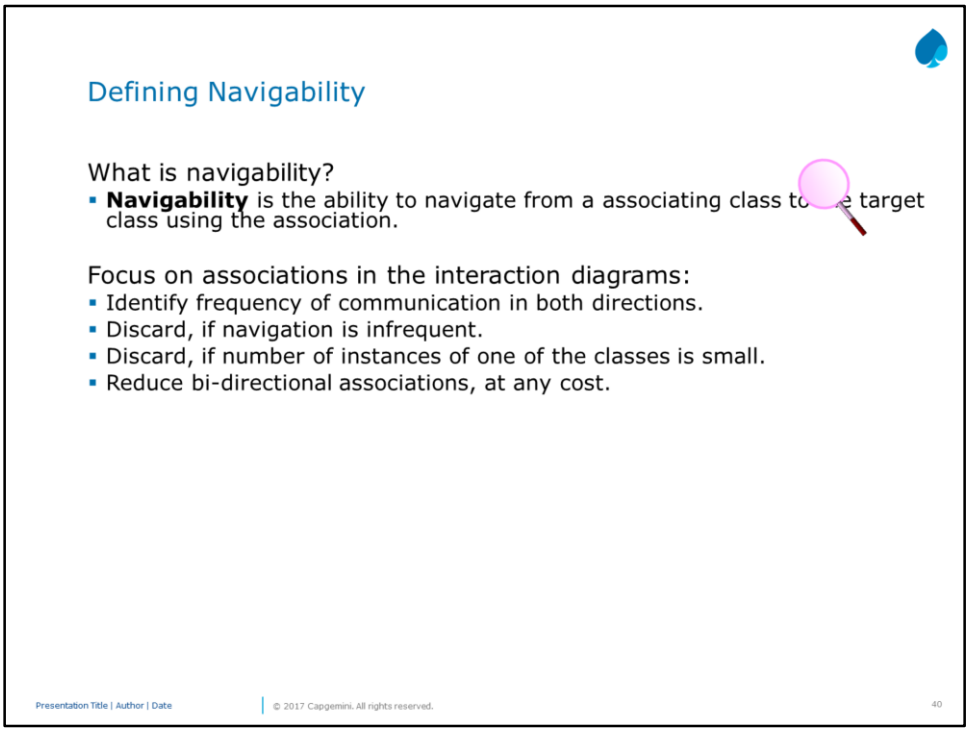


Composition or Attributes: Considerations:

- At times you may have to select either one of the following two methods of modeling relationships:
  - Model a **class property** as a **class** (with a composition relationship), or
  - Model a **class property** as a **set of attributes** of the class. In other words, you can use composition to model an attribute.
- The decision will depend on the nature of the class properties.
  - In case, the properties need to define an independent identity, use a **class** and **composition**.
  - In case, the same properties are shared by more than two classes, use a **class** and **composition**.
  - In case, the properties are **complex** in nature, use a **class** and **composition**.
  - In the absence of the above three criteria, use **attributes**.
  - Also, if the two elements being modeled show a **strong coupling**, use **attributes**.

Instructor Notes:

Bi-directional navigations tend to be expensive, and hence we must optimize the directions of navigation.



Define Navigability:

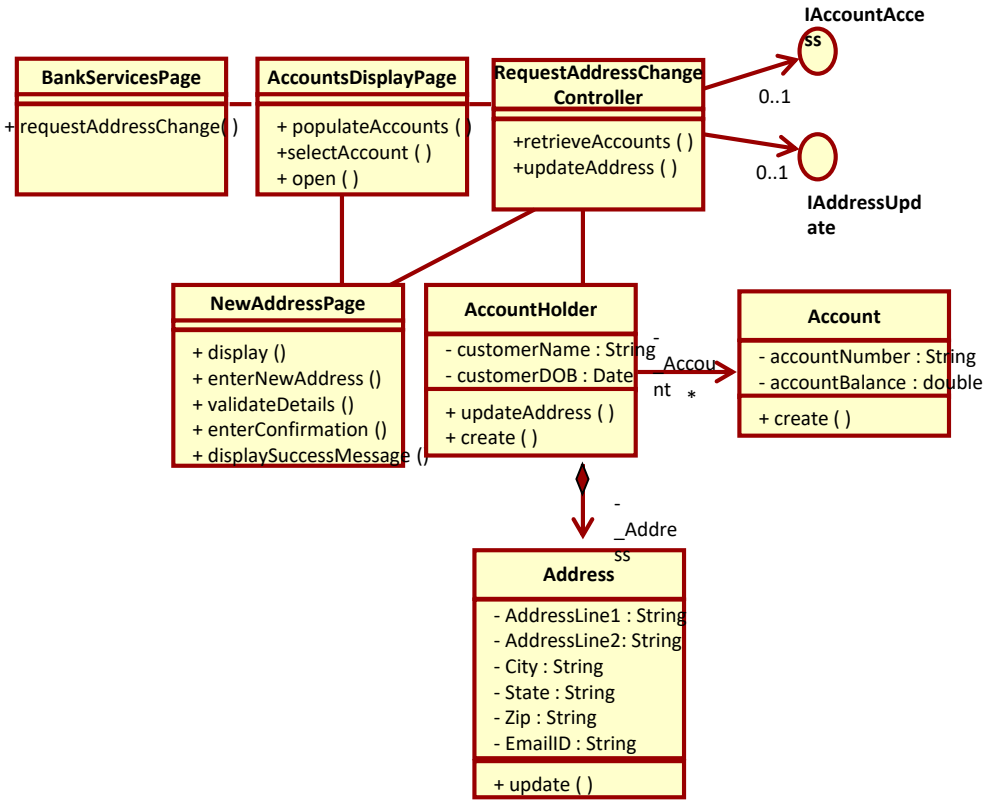
- The **navigability** property on a role indicates that it is possible to navigate from an associating class to the target class using the association. Navigability is indicated by an **open arrow**, which is placed on the target end of the association line next to the target class (the one being navigated to). The default value of the navigability property is **true**.
- You will now have to refine the inter-class association's navigability factor, identified during use-case analysis, and retain only the ones that are most frequently used.
- The issue of navigability is reconsidered in class design. The reason is that we now have a better understanding of class responsibilities and behaviors. **Relationships** too have to be refined. **Bi-directional relations** can be modeled into **unidirectional relations** wherever possible, as bi-directional relationships are more complex and difficult to execute.

Instructor Notes:

A more refined version of class diagram we had seen earlier where relationships have been revisited.

Note that choice between aggregation and composition depends on how closely we want to indicate the relationship.

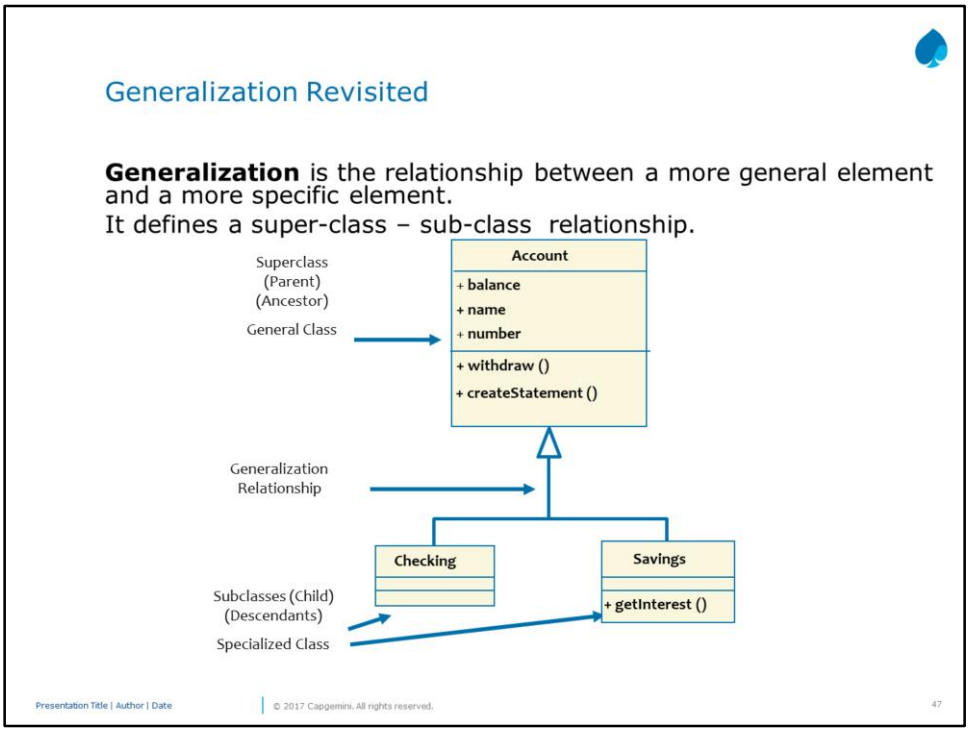
Example: Relationships Refinement:



- In the above example, note that **Aggregation** and **Composition relationships** have replaced the earlier Association relationship we had between Account Holder, and Account/Address.
- Address is assumed to be closely bound to the Account Holder (Here we are assuming only 1 address for a Customer; not separate addresses for each account).

Instructor Notes:

Having checked for possibility of other relationships, we now look at generalization.

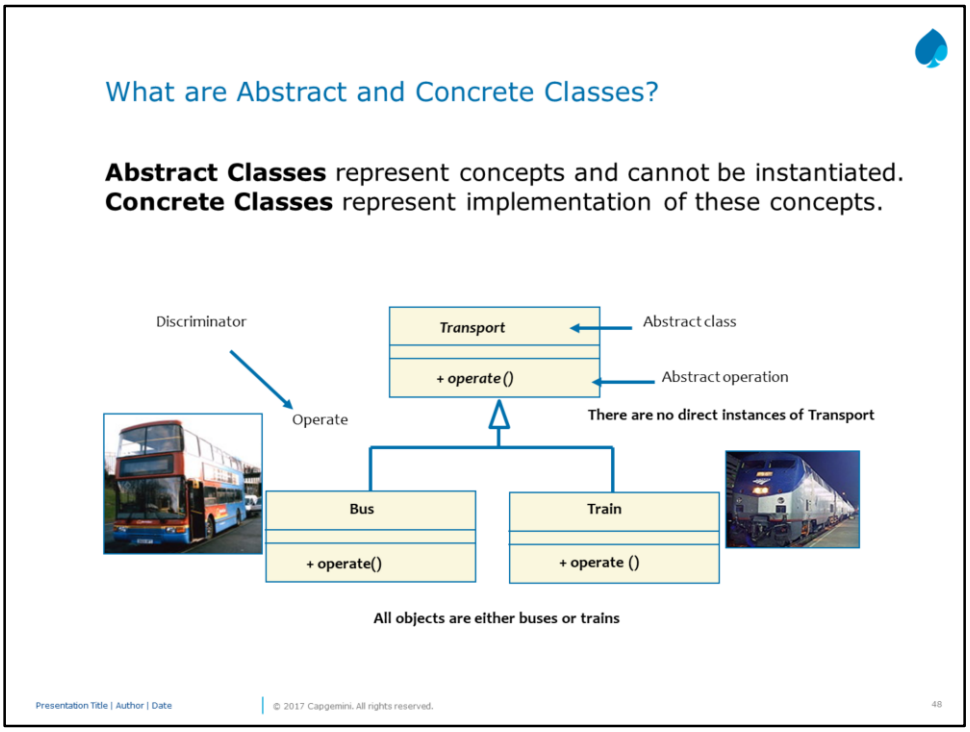


Generalization Revisited:

- You have been introduced to the concepts of **Generalization** and **Inheritance** earlier. Let us recapitulate.
- Generalization represents the "is a" relationship between two or more classes. That means, you should be able to define the child class as being "a kind of" the parent class.
- **Generalization** is always represented by a **hollow triangle**. A vertical line is drawn from the parent class to the vertex of the triangle. Further, lines are drawn from the base of the triangle to the **specialized (child) classes**.
- Various terms are used to define the two classes:
  - **General Class:** Parent, Ancestor, Super-class
  - **Specialized class:** Child, Descendent, Subclass
- After identifying generalization, create a common super-class. This super-class should contain the common attributes, associations, aggregations, and operations of all the subclasses. The common behavior is removed from the classes that are to become sub-classes of the common super-class. A **generalization relationship** is drawn from the **sub-class** to the **super-class**.

Instructor Notes:

A design heuristic says that base of a class hierarchy should ideally be an abstract class.

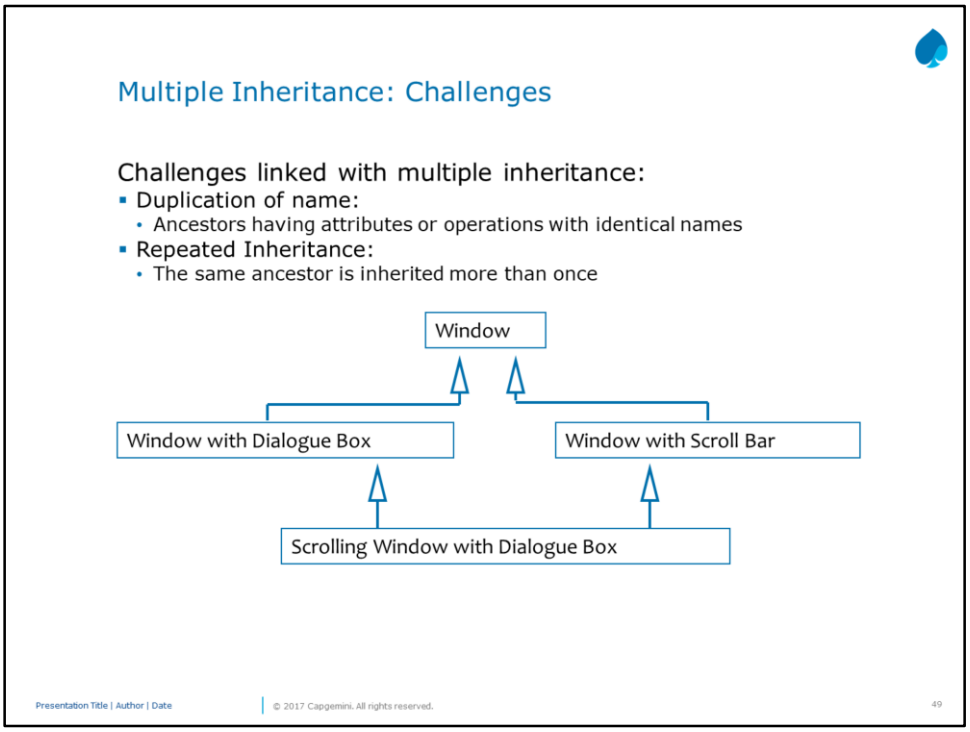


What are Abstract and Concrete Classes?

- A class that is not instantiated and exists only for other classes to inherit it, is an **abstract class**. Classes that are actually instantiated are **concrete classes**. Note that an abstract class must have at least one descendant to be useful.
- On the lines of an **abstract class**, you can have **abstract operations** too. No implementation exists for an **abstract operation** in the class where it is specified. Classes with even one abstract operation are abstract themselves. Implementation (for the operation) is provided by the classes that inherit from such abstract classes. Otherwise, the operations are considered abstract within the subclass, and the subclass is considered abstract, as well.
- **Concrete classes** have implementations for all operations.
- The UML designation of abstract classes and operations is putting the name in italics.
- A **discriminator** can be used to indicate on what basis the generalization / specialization occurred. A discriminator describes a characteristic that differs in each of the subclasses. In the above example, we generalized / specialized in the manner in which they operate.

Instructor Notes:

Use Multiple Inheritance only if absolutely required!

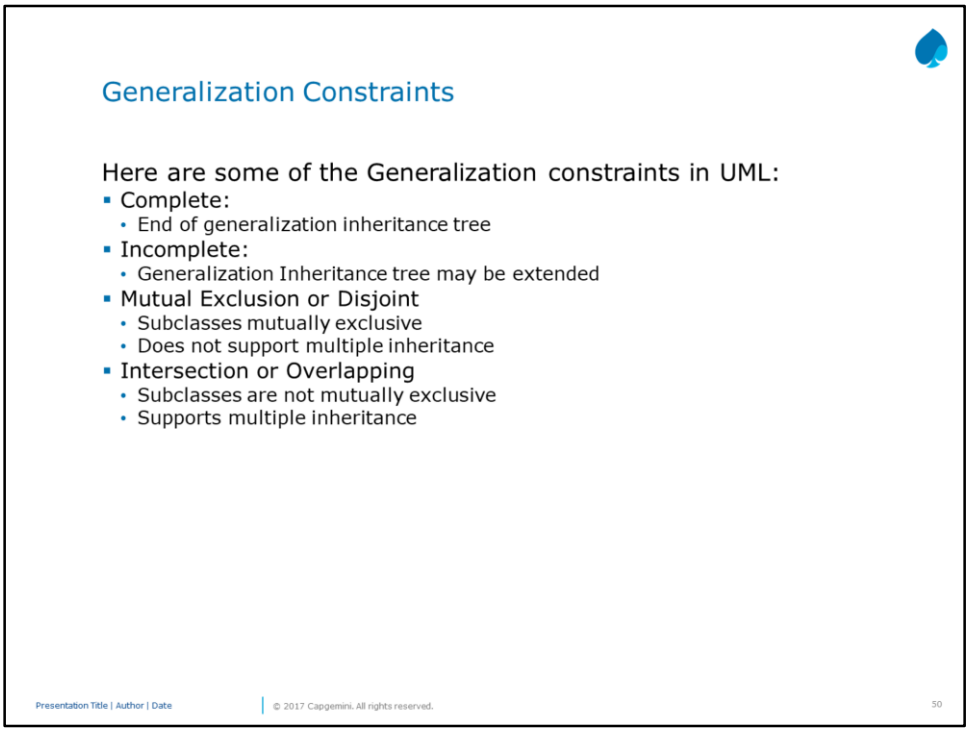


Multiple Inheritance: Challenges:

- A class can inherit from several other classes through **multiple inheritance**, although generally it will inherit from only one.
- There are a couple of potential problems you must be aware of if you use multiple inheritance:
  - If the class inherits from several classes, you must check how the relationships, operations, and attributes are named in the ancestors. If the same name appears in several ancestors, you must describe what this means to the specific inheriting class, for example, by qualifying the name to indicate its source of declaration.
  - If repeated inheritance is used, then in this case, the same ancestor is being inherited by a descendant more than once. When this occurs, the inheritance hierarchy will have a “diamond shape” as shown in the diagram above.
- A question that might arise in this context is “How many copies of the attributes of Window are included in instances of Scrolling Window With Dialog Box?” Thus when you are using repeated inheritance, you must have a clear definition of its **semantics**. In most cases, this is defined by the programming language supporting the multiple inheritance.
- In general, the programming language rules governing multiple inheritance are complex, and often difficult to use correctly.
- Therefore it is recommended that multiple inheritance should be used only when needed and with caution.

Instructor Notes:

To better describe the generalization mechanisms, there are certain predefined constraints that are defined.

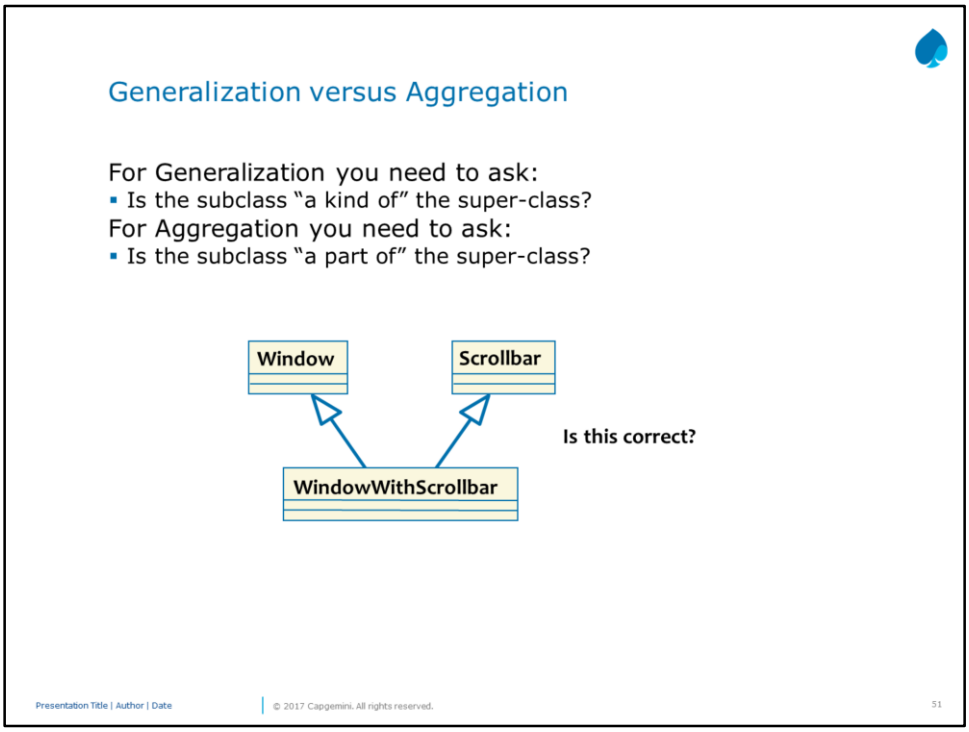


What is a constraint?

- A **constraint** is a semantic condition or restriction. Certain constraints are predefined in the UML, others may be user defined. **Constraints** are one of three **extensibility mechanisms** in UML.
- Four different types of constraints for generalization relationships are defined in the UML.
  - **Complete:** When an inheritance hierarchy cannot be defined any further, the constraint is said to be complete.
  - **Incomplete:** When an inheritance hierarchy has not been completely defined, the constraint is said to be incomplete. In this case more children may be defined.
- The following are types of multiple inheritance:
  - **Disjoint:** When subclasses are mutually exclusive, the constraint is said to be disjoint. In this case, an object of the parent class cannot have more than one of the children as its type.
  - **Overlapping:** This is the very opposite of “Disjoint”. An object of the parent may have more than one of the children as its type.

Instructor Notes:

Discuss the “Is a” versus “Has a” connotations

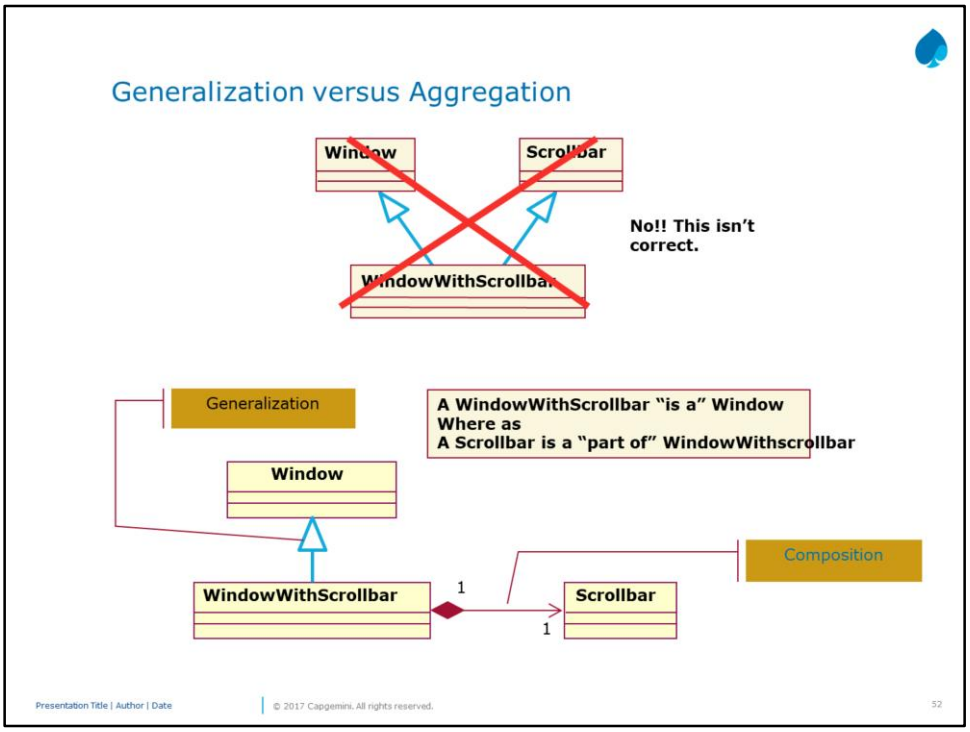


Generalization versus Aggregation:

- In the example in the above slide, **Scrollbar** is “part of” a **WindowWithScrollbar**, and as such, the relationship is an aggregation.
- **WindowWithScrollbar**, on the other hand, “is a” **Window**. Hence its relationship to Window is a generalization.

Instructor Notes:

An example of Is A and Has A.

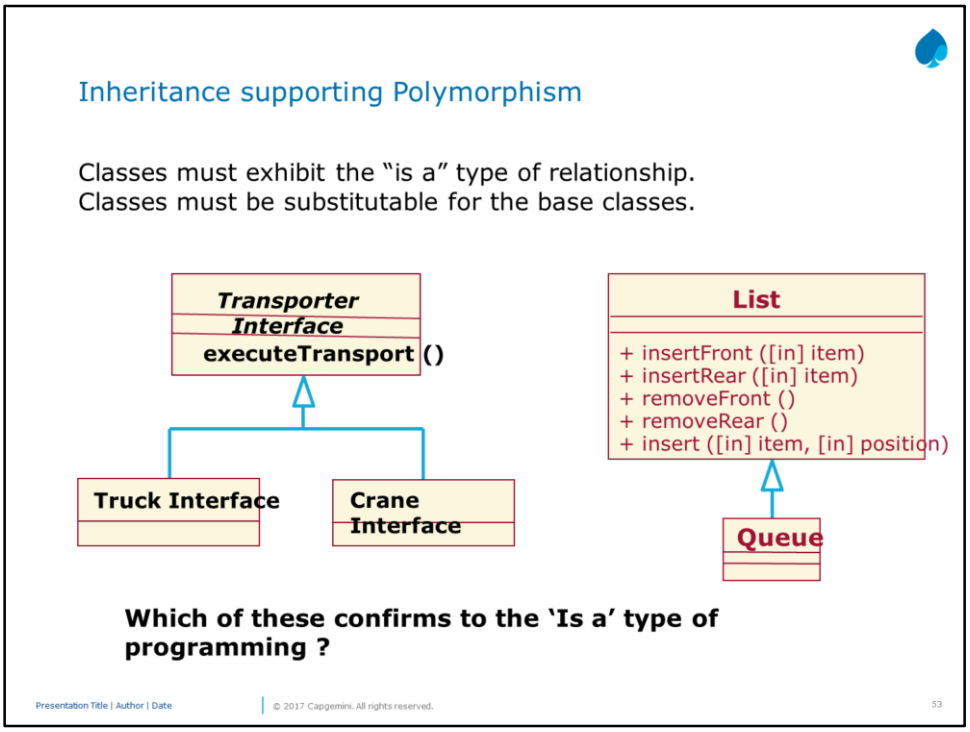


Generalization versus Aggregation:

- The example in the previous slide is part correct and part incorrect. Both **Window** and **Scrollbar** are shown as having a generalization association with **WindowWithScrollbar**.
- Only the relationship between the **WindowWithScrollbar** and **Window** is a generalization, as it returns a "Yes" answer to the question:
  - Is **Window** "a kind of" **WindowWithScrollbar**?
- On the other hand, the **Scrollbar** defines an aggregation with the **WindowWithScrollbar**, as it returns a "Yes" answer to the question:
  - Is **Scrollbar** "a part of" **WindowWithScrollbar**?

Instructor Notes:

An example of Is A with focus on Polymorphism



Inheritance supporting Polymorphism:

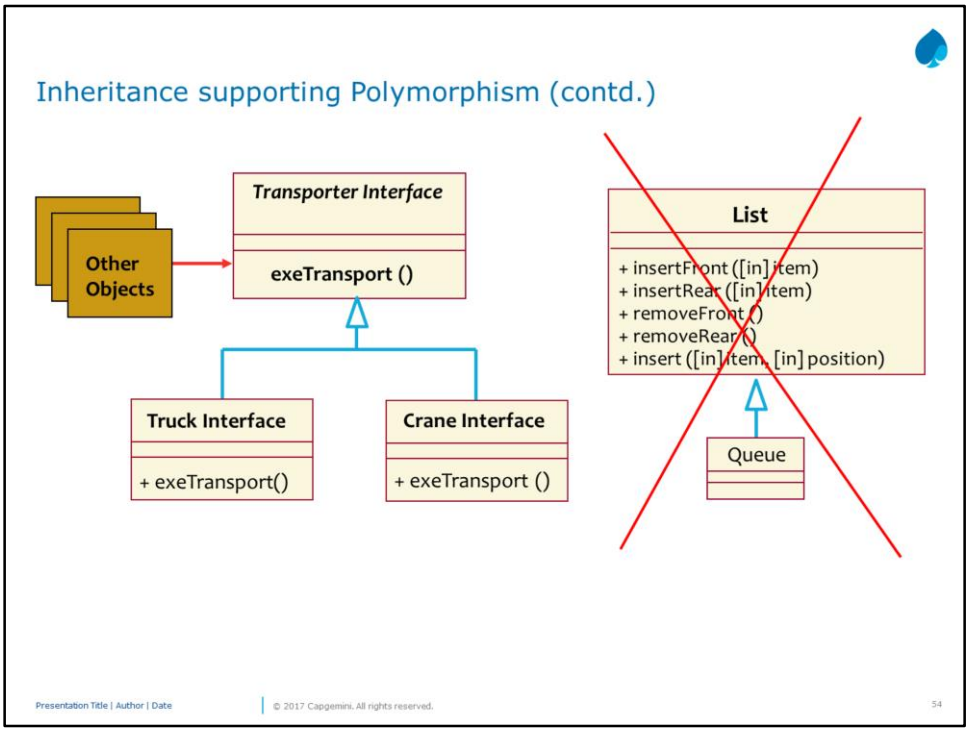
- Normally, a **subtype relationship** is expressed in terms of **inheritance**, which states that the **sub-class** is a type of the **super-class**. It also means that the **sub-class** can be a substituted for the **super-class** in any situation. This is the main principle behind the "is a" style of programming.
- The "is a" style of programming adheres to the **Liskov Substitution Principle**, which we shall discuss in the next lesson.
- In other words, for the **subtype** to be usable in place of the **base type**, the **subtype** must have the same constrains as the **base class**. Let us understand this with examples.

Instructor Notes:

Queue is a FIFO structure and cannot include behavior for inserting at front or any other position; or removing from rear.

How will we model the RHS diagram?

Choose superclass with only generic behavior; both List and Queue can then derive from it and add their own specific behavior.

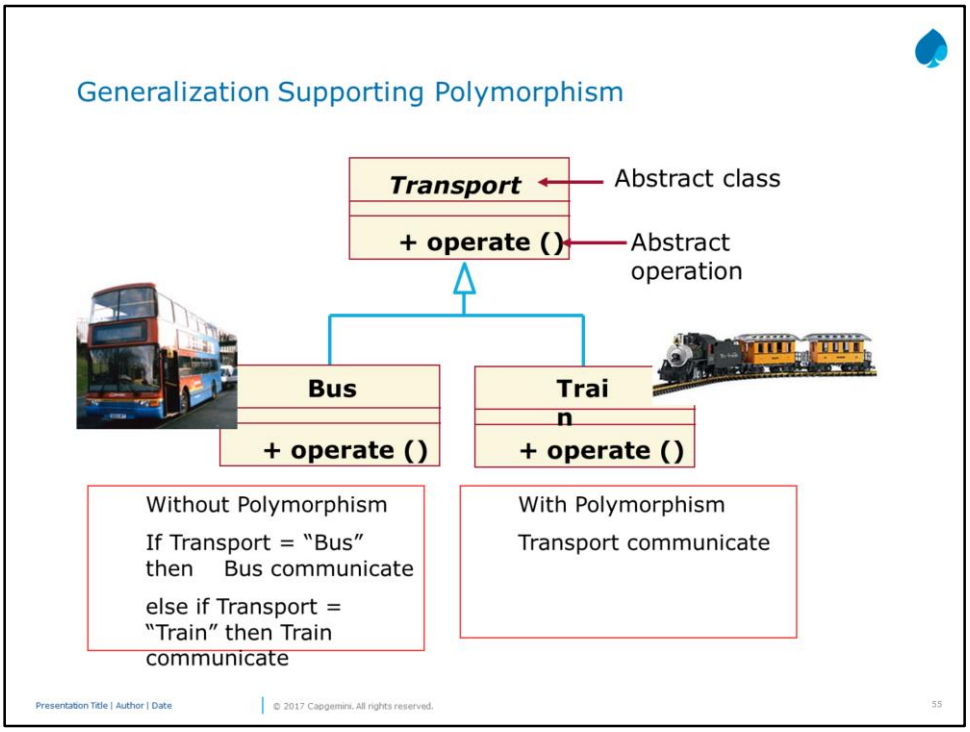


<u>Inheritance Supporting Polymorphism</u>:

- The example in the above slide follows the principles of “is a” type of programming.
- Both the **Truck Interface** and **Crane Interface classes** inherit from the **Transporter Interface**. That is, objects of both classes will respond to the message **executeTransport**. The objects may stand in for **Transporter Interface** at any time and will offer all its behavior. Thus other objects (client objects) can send a message to a **Transporter Interface** object, without knowing if a **Truck Interface** or **Crane Interface** object will respond to the message.
- The **Transporter Interface** class can even be abstract, that is never instantiated on its own. In this case, the **Transporter Interface** might define only the signature of the **executeTransport** operation, whereas the descendant classes implement it.
- On the other hand the example on the right does not adhere to the “is a” type of programming principles.
- A **queue** cannot be substituted for a list. The **queue** inherits some, and not all of the behaviors of a list.

Instructor Notes:

Generalization, which is one approach to implementing polymorphism, allows for compact code



<u>Inheritance and polymorphism</u>:

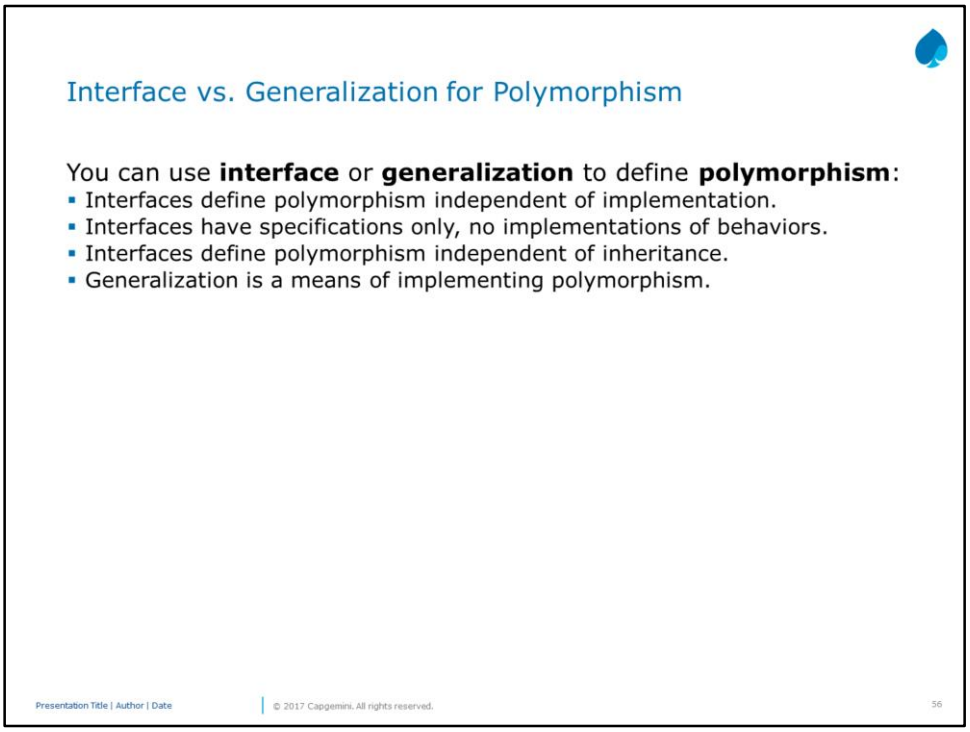
- In case of a set of classes where **polymorphism** is implemented, in the same way you can model the **polymorphism** as an **inheritance**. This means that, the **base classes** having only **inherited operations** and no implementation of operations, can be replaced by **interfaces**.
- The inheritance is restricted to inheritance of implementations only.

Generalization and polymorphism:

- **Generalization** is one of the many ways of implementing **polymorphism**.
- When defining polymorphism through generalization, you are defining alternate methods for **ancestor class operations** in the **descendent classes**. It is useful in reducing the codes to be written and also helps abstract the interface to descendent classes.
- **Polymorphism** is an advantage of **inheritance** realized during implementation and at run time.
- Programming environments that support polymorphism use **dynamic binding**. It implies that the actual code to execute is determined at run time rather than compile time.

Instructor Notes:

Guidelines for choosing Interface v/s Generalization for implementing polymorphism



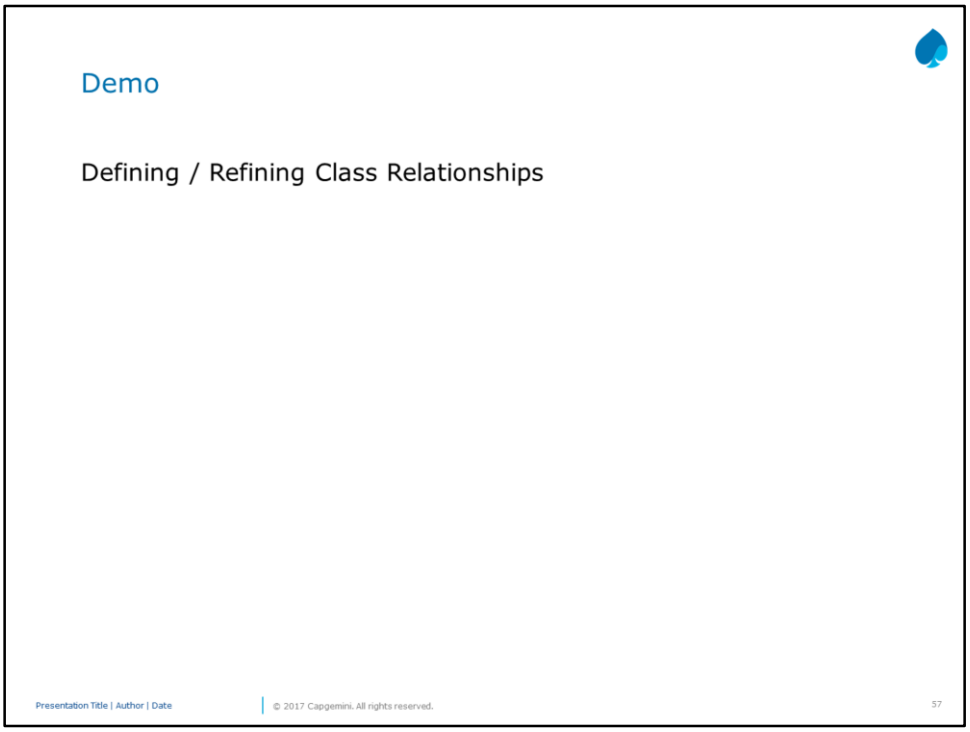
Interface versus Generalization for Polymorphism:

- The million dollar question is, what is better for implementing **polymorphism** – generalization or interfaces?
- You need to understand the basic difference between the two:
  - **Interfaces** define polymorphism in a declarative way, unrelated to implementation. Two elements are said to be polymorphic, with respect to a set of behaviors, if they realize the same interfaces. Interfaces are orthogonal to class inheritance lineage. Two different classifiers may realize the same interface but be unrelated in their class hierarchies.
  - **Interfaces** are purely specifications of behavior (a set of operation signatures). An abstract base class may define attributes and associations as well.
  - **Interfaces** are totally independent of inheritance. **Generalization** is employed to re-use implementations. **Interfaces** are employed to re-use and formalize behavioral specifications.
  - **Generalization** provides a way to implement polymorphism in cases where polymorphism is implemented the same way for a set of classes. This has been discussed earlier in this lesson.

Instructor Notes:

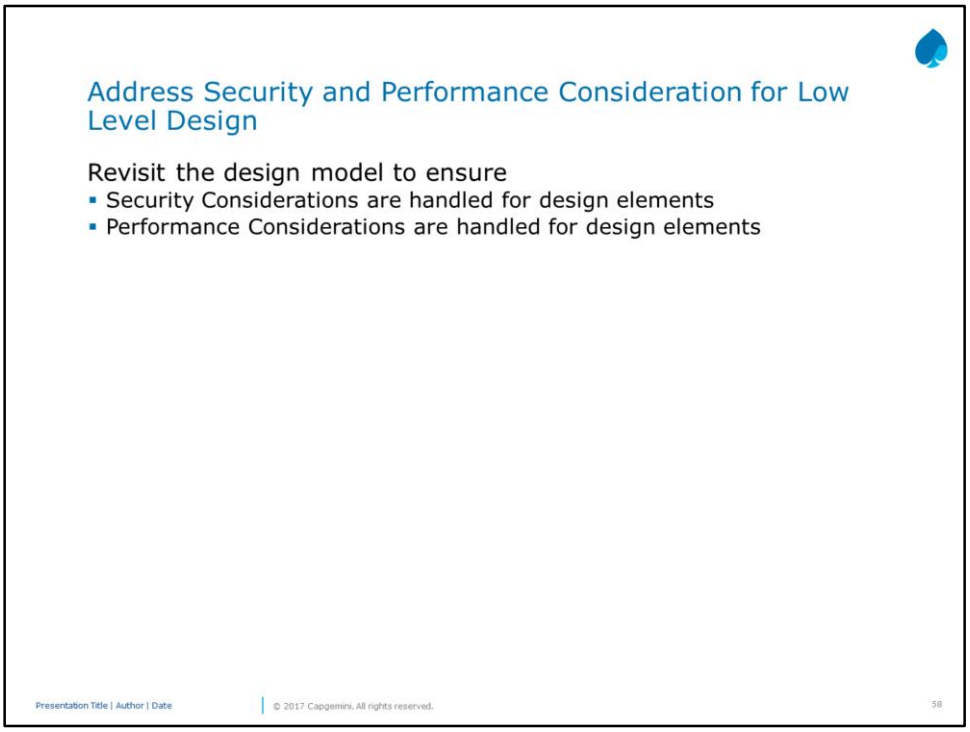
Use the tool to do this.

Other than what is shown in earlier example, we can also model the Account Hierarchy..introduce the savings and current accounts as subclasses of abstract account class



Instructor Notes:

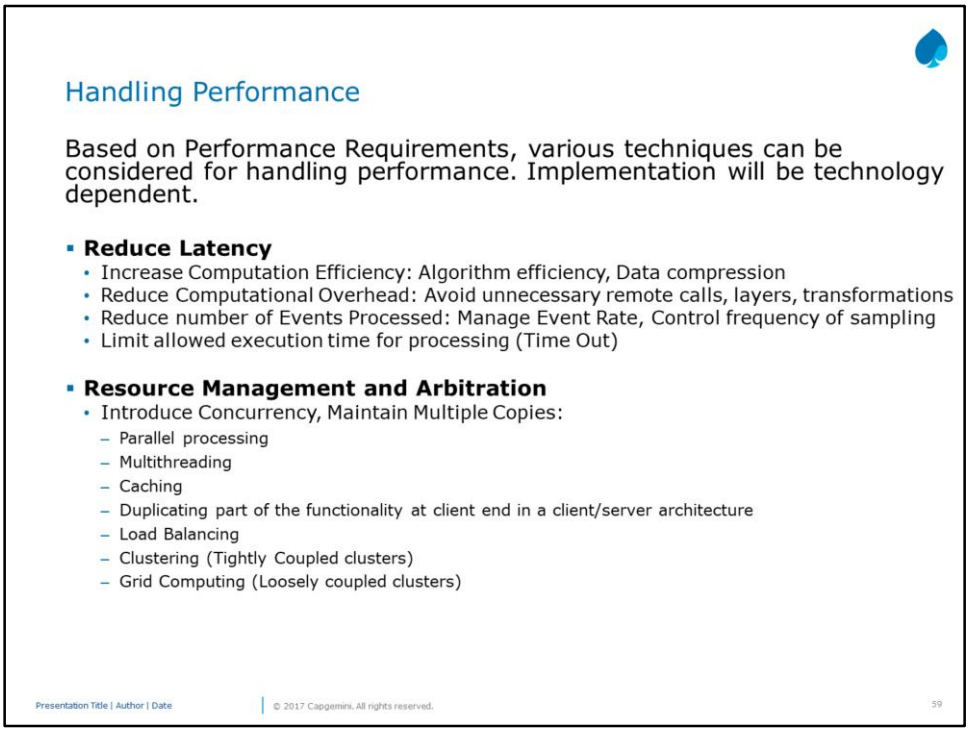
These considerations were earlier outlined in architecture and HLD; need to take that forward for LLD



<u>Task2: Create Application Detail Design</u>:
Step3 & 4: Address Performance and Security Considerations in LLD

Instructor Notes:

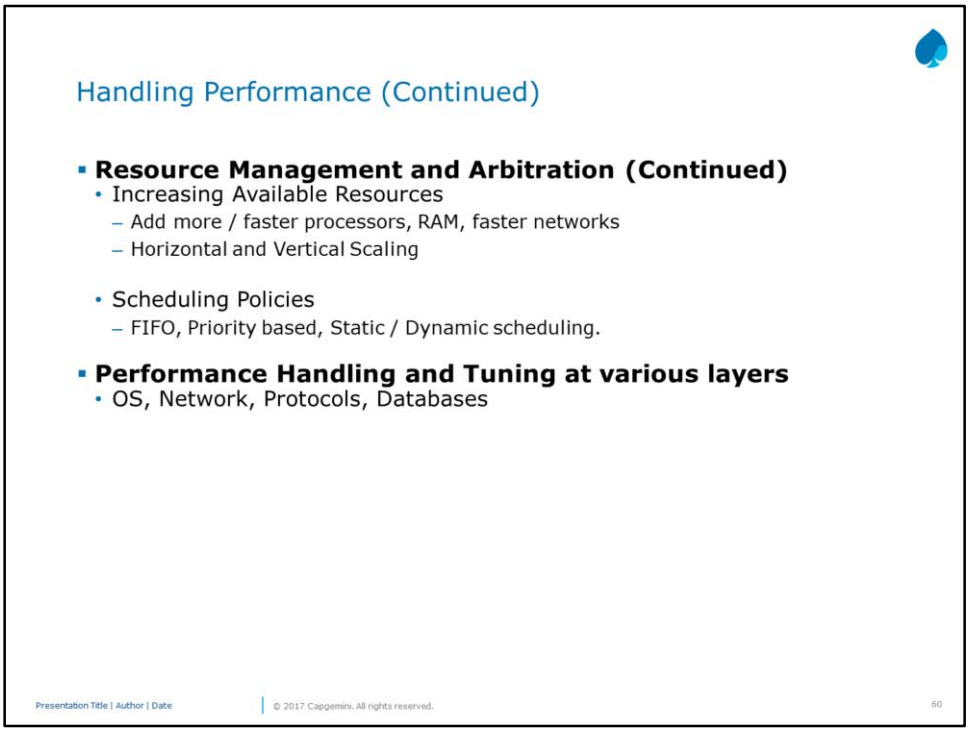
Talk about these techniques at an overview level. Objective here is to touch upon parameters that come into play when we talk about performance.



Listed here are some techniques for handling performance. The LLD implementations will depend on the implementation technology. The implementation specific Level 2 Designer Course will discuss on this topic more in detail.

Instructor Notes:

Talk about these techniques at an overview level. Objective here is to touch upon parameters that come into play when we talk about performance.



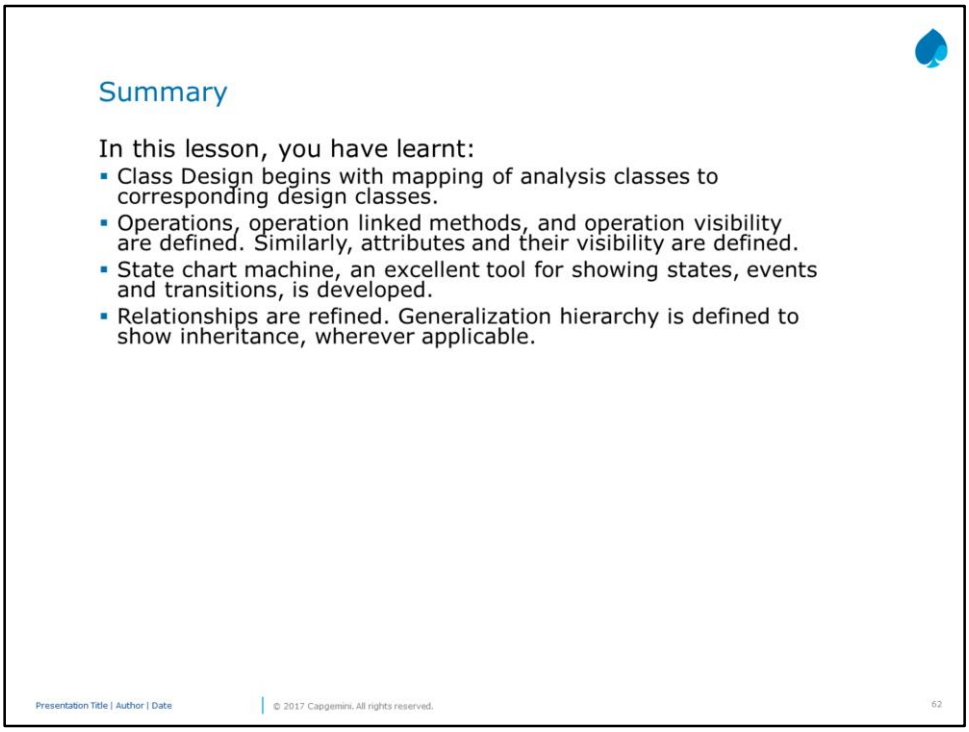
Listed here are some techniques for handling performance. The LLD implementations will depend on the implementation technology. The implementation specific Level 2 Designer Course will discuss on this topic more in detail.

Instructor Notes:

Talk about these techniques at an overview level. Objective here is to touch upon parameters that come into play when we talk about security.

Listed here are some techniques for handling security. The LLD implementations will depend on the implementation technology. The implementation specific Level 2 Designer Course will discuss on this topic more in detail.

Instructor Notes:



Instructor Notes:

Answers for Review Questions:

Answer 1: Operation only gives the specification of behaviour whereas the method also gives the implementation of the behaviour.

Answer 2: Underlining the respective members

Answer 3: Start State and End State

Answer 4: Dependency relationship is nonstructural; Association is structural

Answer 5: Local, Parameter and Global visibility

Answer 6: Association Classes

Answer 7: italics