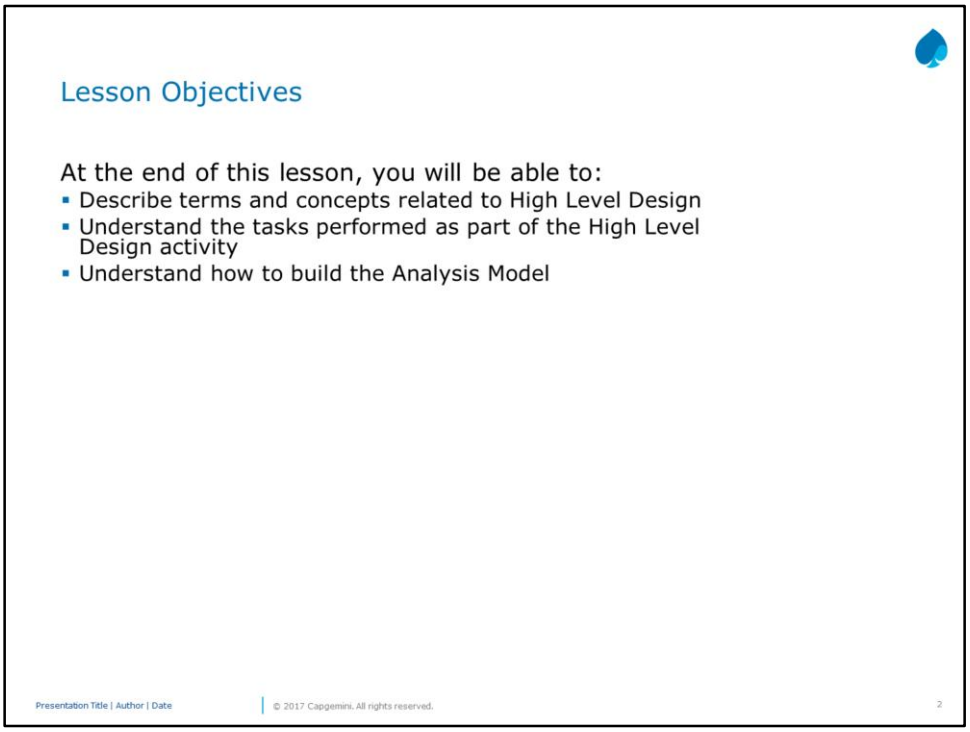
Instructor Notes:

Add instructor notes here.

Instructor Notes:

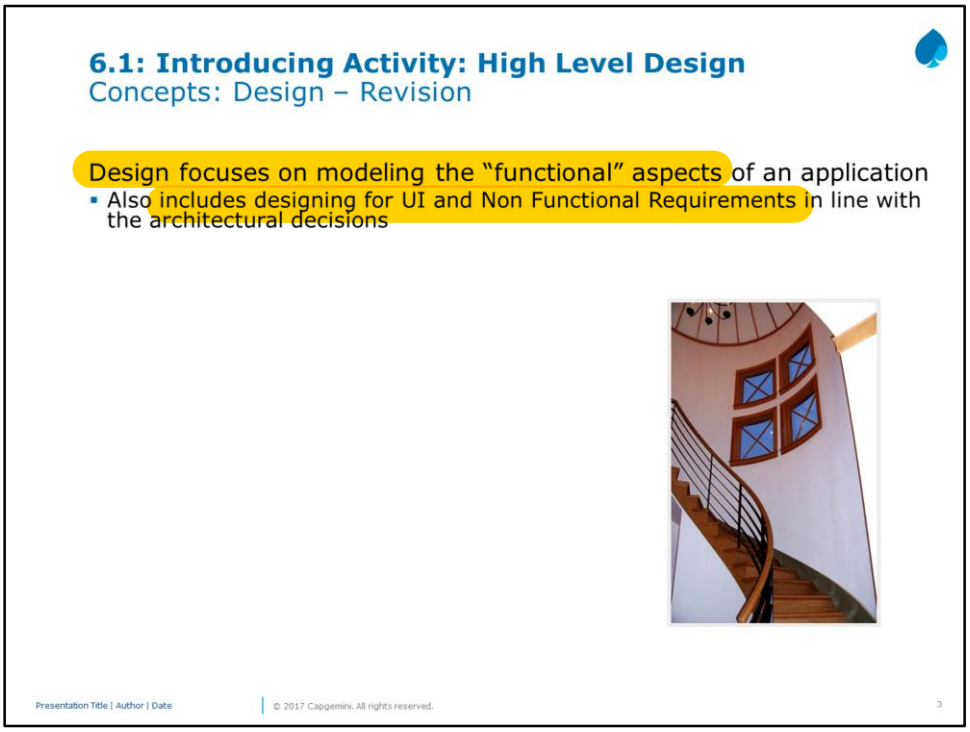
We now look at the tasks relating to High Level Design. Our focus will be on creation of the Analysis Model.



Lesson Objectives:
In this lesson, we shall look at terms, concepts, and tasks related to High Level Design. The focus of High Level Design will be to baseline the Analysis Model.

Instructor Notes:

Quickly review the concept of Design which was introduced earlier.

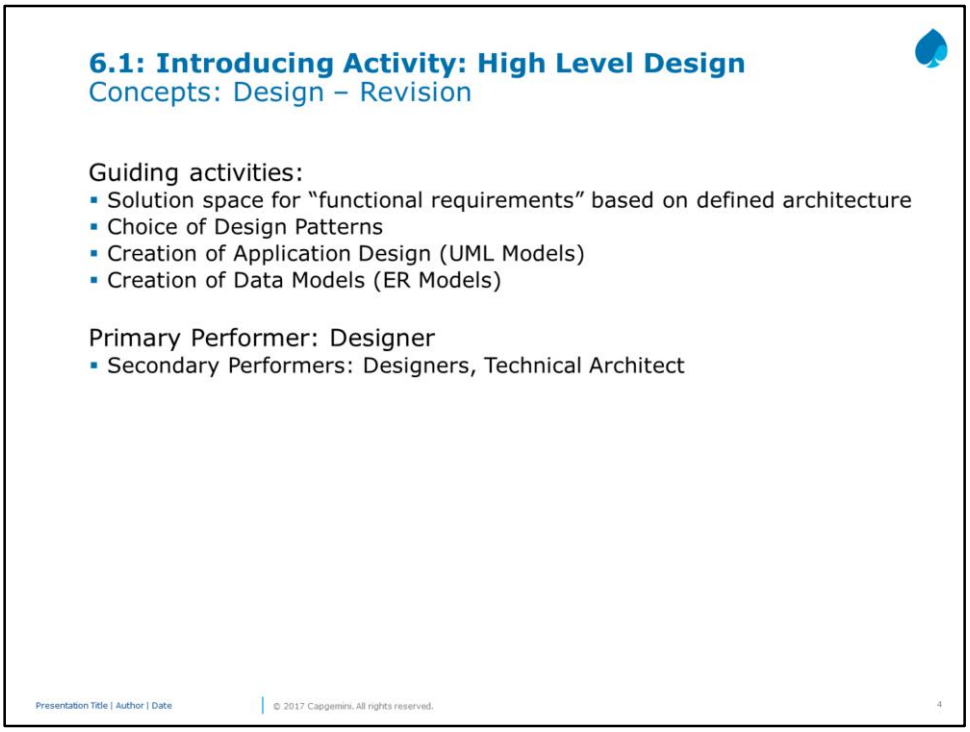


Concepts: Design – Revision:

- This is a review slide.
- While **architecture** focuses on non-functional aspects, **design** focuses on the functional aspects of the application. Design activities and decisions related to application design are taken. Design activity leads towards implementation. Hence focus is on creating detailed UML models and ER models to capture the designs.
- **Design Lead** is the primary owner of the design activities, with support from the **Technical Architect** for reviewing and **Designers** to help with detailing the designs.

Instructor Notes:

Quickly review the concept of Design which was introduced earlier.



Concepts: Design – Revision:

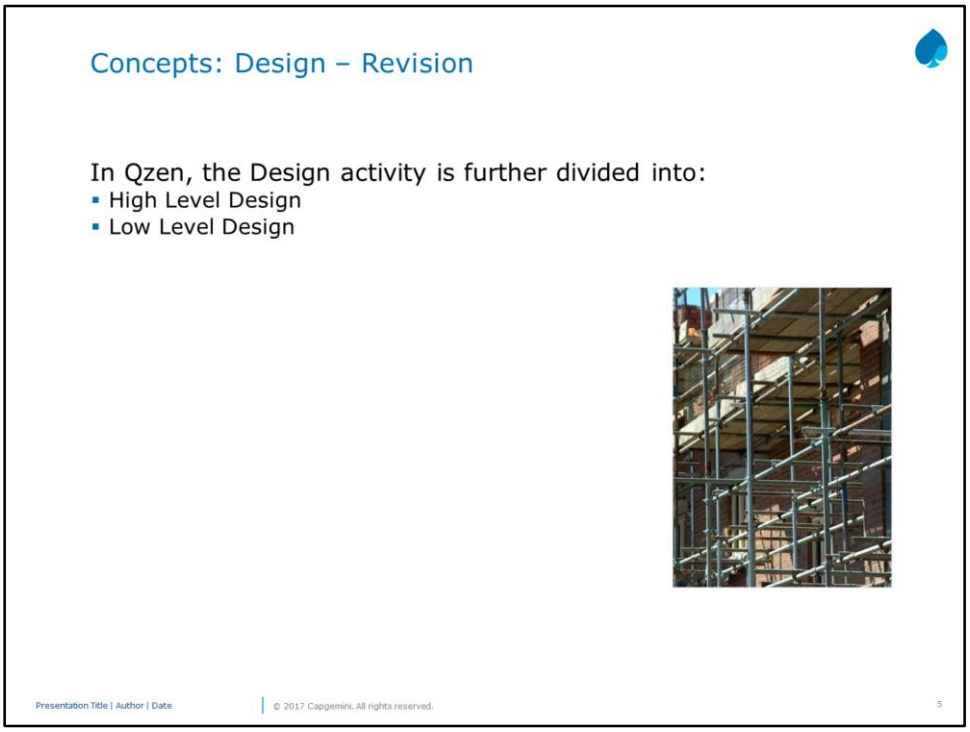
- This is a review slide.
- While **architecture** focuses on non-functional aspects, **design** focuses on the functional aspects of the application. Design activities and decisions related to application design are taken. Design activity leads towards implementation. Hence focus is on creating detailed UML models and ER models to capture the designs.
- **Design Lead** is the primary owner of the design activities, with support from the **Technical Architect** for reviewing and **Designers** to help with detailing the designs.

Instructor Notes:

Design is broadly categorized as HLD and LLD. While HLD focuses on the breadth of application to arrive at an Analysis Model, the LLD focuses on refining the Analysis Model to arrive at the Design Model.

Fig.: The analogy here is with reference to Architecture – while Architecture focused on beams and pillars, design focuses on building the walls and interiors.

Methodologies such as RUP do not distinguish activities for HLD and LLD, but they do have steps leading to creation of Analysis Model and Design Model

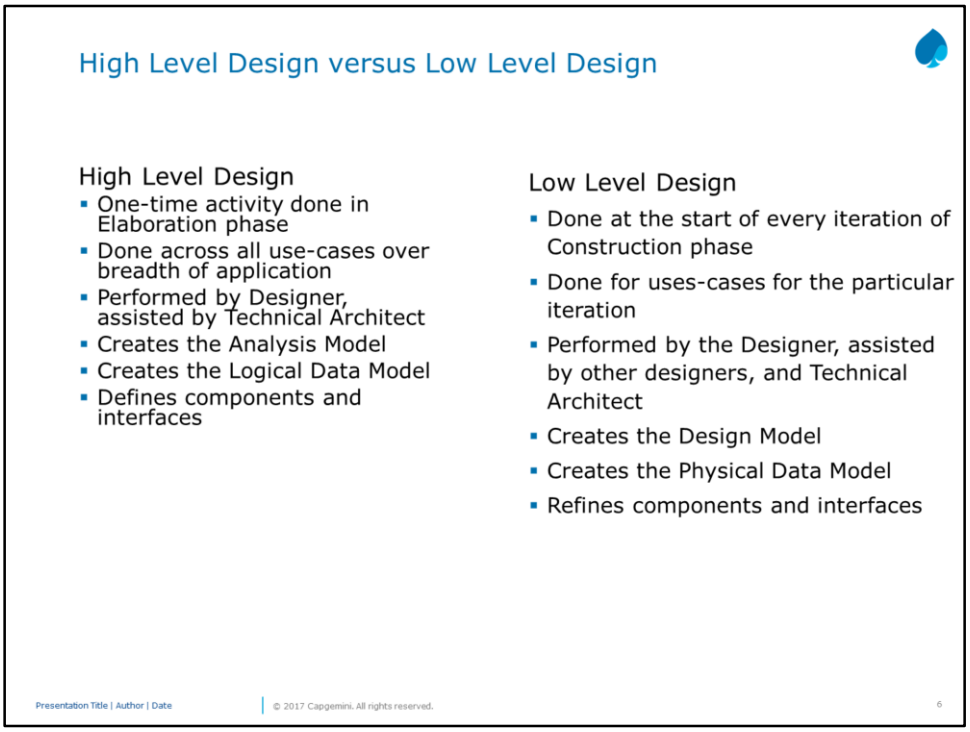


Concepts: Design – Revision:

- The concepts of **High Level Design** (HLD) and **Low Level Design** (LLD) are mapped in this lesson keeping in mind the organization's Vendor scenario. It is advisable to have a "breadth" of requirements and create an analysis model by elaboration, when you provide a fixed price bid.
- The other need for the breaking down the **design** activity was felt by organization teams in view of de-skilling. To optimize the work, senior designers can lead the activities with assistance from other junior designer.

Instructor Notes:

Essentially the HLD focuses on arriving at a conceptual model for the system and the LLD then refines this conceptual model to include technology specific details to arrive at the detailed design for the system.



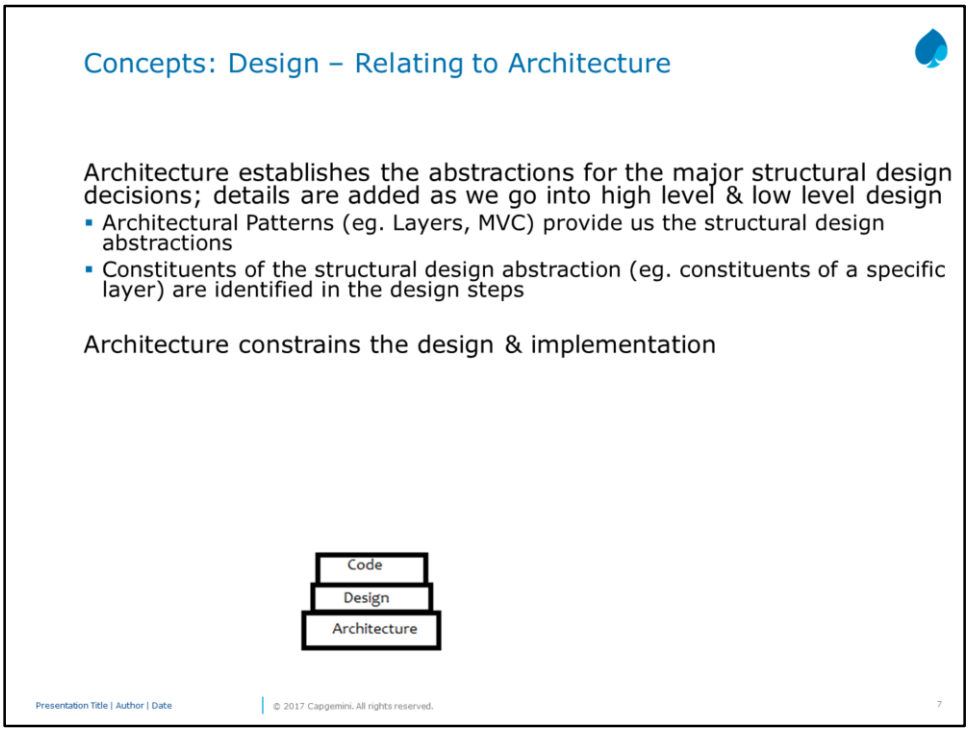
<u>High Level Design versus Low Level Design</u>:

The above slide outlines the following w.r.t. HLD and LLD:

1. When does the activity get done?
2. What is included in the scope?
3. Who is responsible for the activity?
4. What are the outputs?

Instructor Notes:

Explain how design is within the foundation laid by Architecture

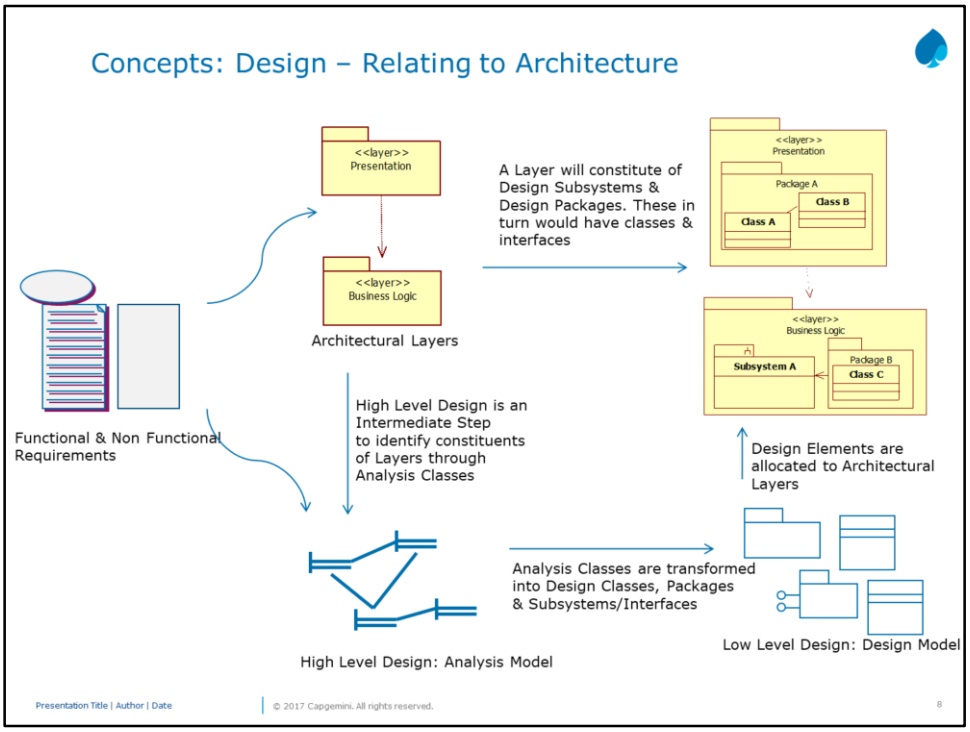


Concepts: Relating Design to Architecture:

- The design activities are centered around the notion of architecture. Architecture, established in early iterations, represents the major structural design decisions. Architecture provides the abstraction of the entire design, details of which are added as we go into design.
- Architecture can be viewed as a set of key design decisions and these constitute the fundamental decisions about the software design. Architecture puts a framework around the design.

Instructor Notes:

Idea here is to give the big picture of how High Level Design & Low Level Design with align with the Architecture

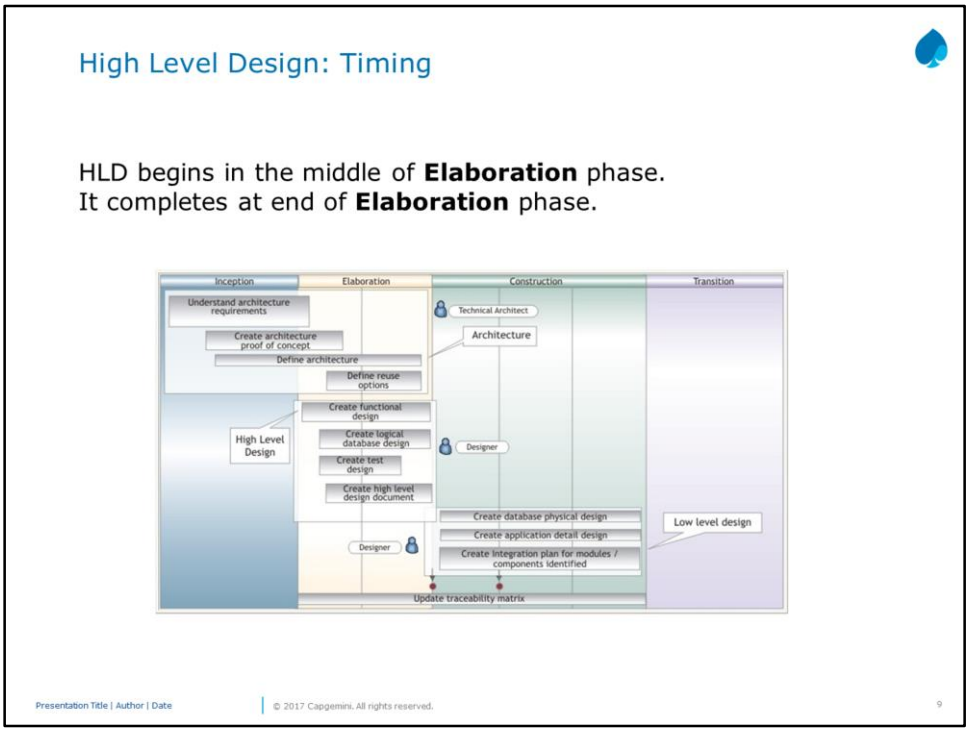


<u>Concepts: Relating Design to Architecture</u>:

- The approach we are adopting is not strictly top down or bottom up; it is more of top-middle, middle-up, middle-down and bottom-middle!
- The inter-relationship and big picture of architecture & design is illustrated in this slide.
    - The Requirements are used to arrive at architectural decisions. In terms of layered architecture, constituents of each layer are to be identified.
    - Constituents of a layer are subsystems and packages, which in turn are constituted of classes & interfaces. To identify these design elements in terms of subsystems/packages/classes/interfaces, High Level Design is an intermediate step which in turn is transformed into Low Level Design.
    - Arriving at Analysis Classes using Boundary/Control/Entity Stereotypes helps in focusing on domain and functional aspects without getting into technology specific implementation details which is the key focus of design elements.

Instructor Notes:

Note that Architectural activities would be in sufficiently advanced stage by the time HLD activities are initiated.

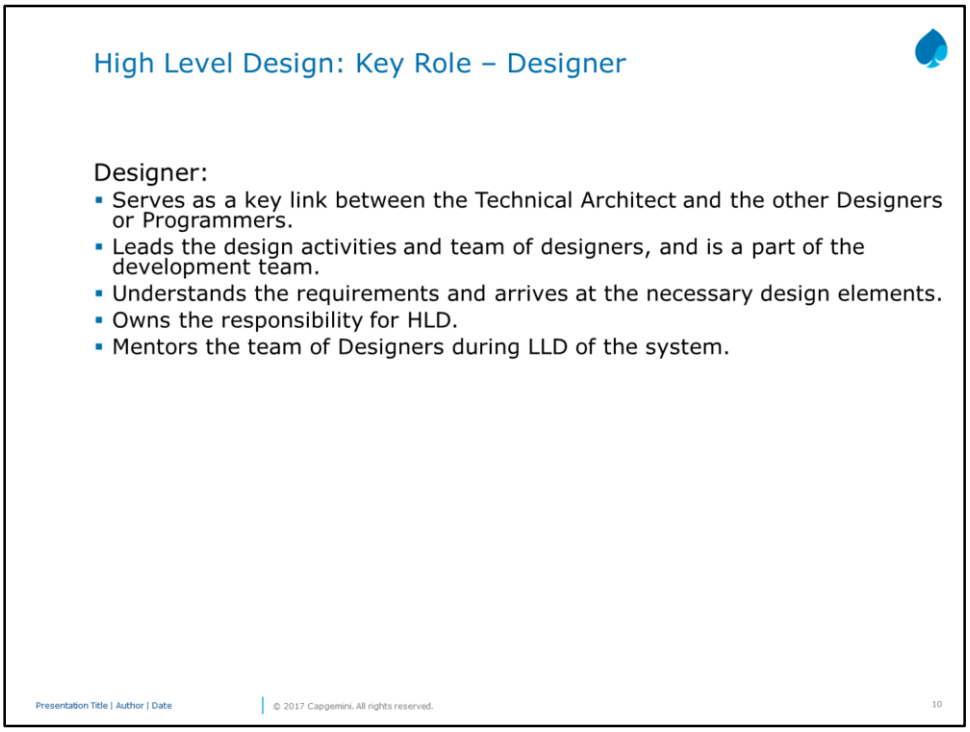


<u>High Level Design: Timing</u>:

- Activities related to High Level Design begin towards middle of Elaboration phase.
- It is baselined by end of Elaboration phase.

Instructor Notes:

Designer working on HLD too needs to have good communication skills and good depth and breadth of technology.

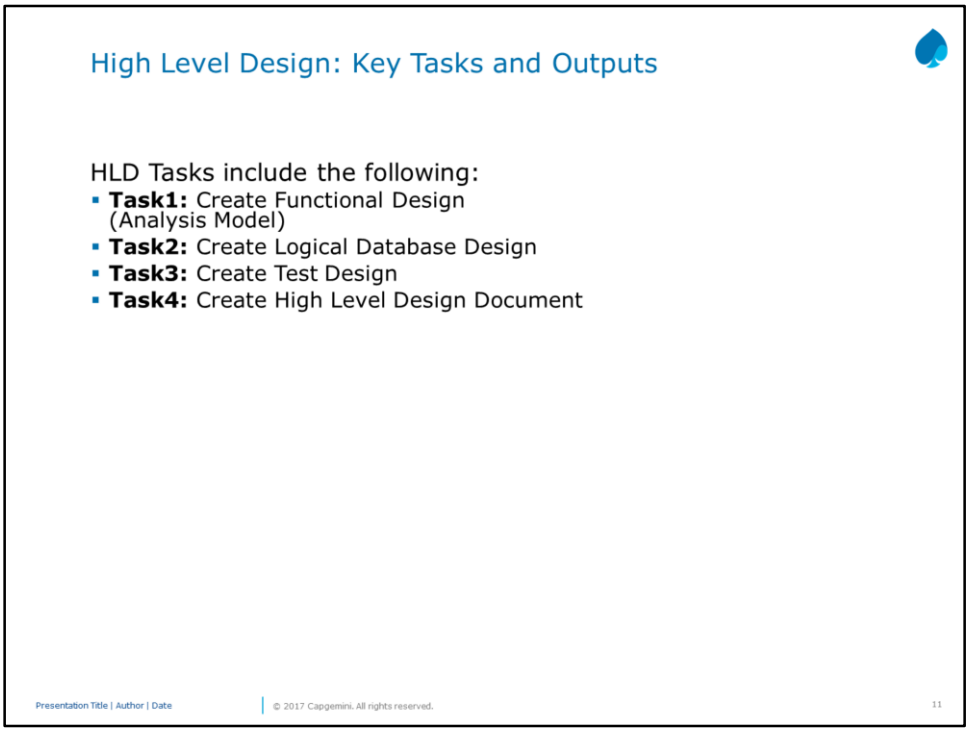


<u>High Level Design: Key Role – Designer</u>:

- The competencies and expectations from the Designer working on HLD are outlined in the above slide.
- The Designer acts as a bridge between the Technical Architect and team of Designers working on LLD. Designer working on HLD is a part of the Design team and also the Development team.

Instructor Notes:

Here are the key tasks which will get discussed in detail in the later sections.

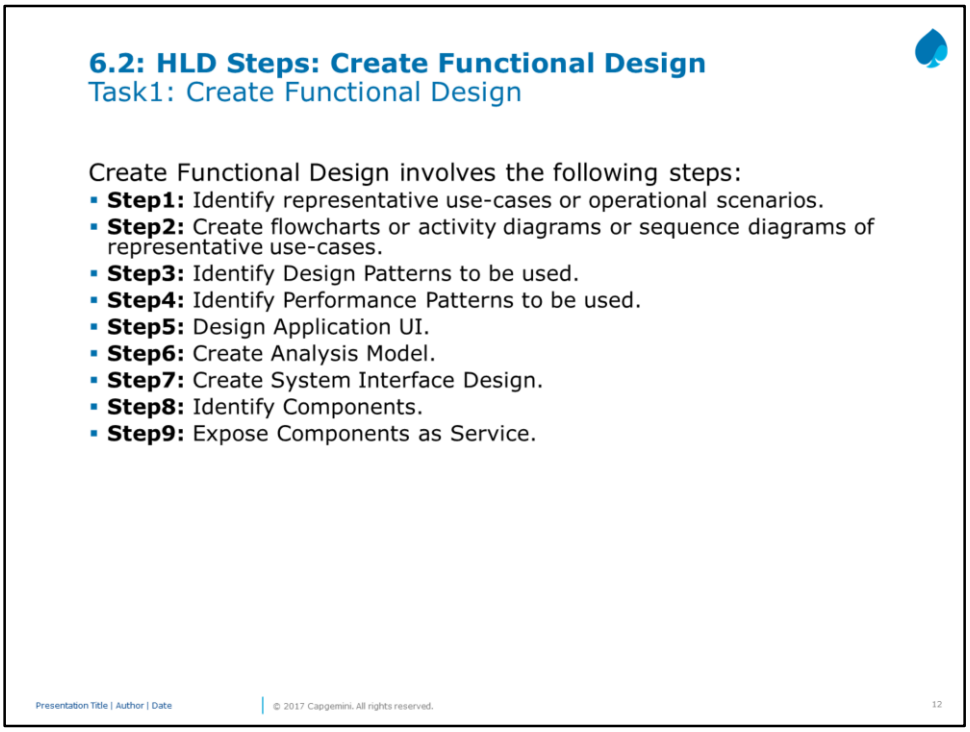


<u>High Level Design: Key Tasks and Outputs</u>:

- The various tasks of High Level Design are listed in the above slide.
- Now, we shall look at the key steps that get done in each of these tasks.

Instructor Notes:

Let us begin with the steps of the first task. The key focus will be on creating the Analysis Model.

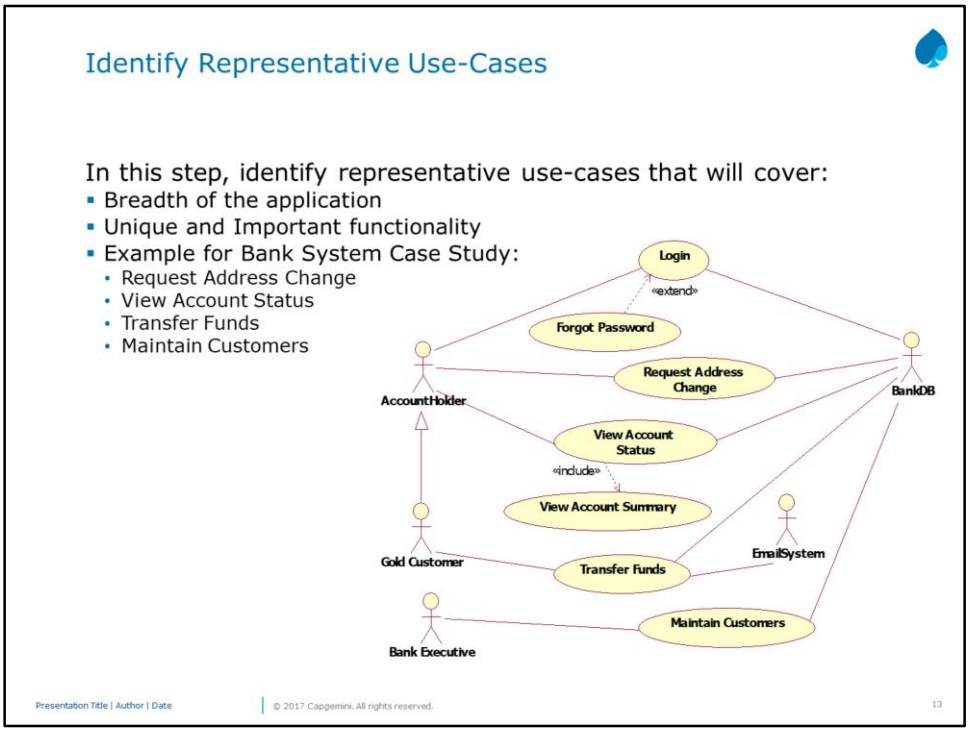


<u>Task1: Create Functional Design</u>:

- The above slide lists the steps for Create Functional Design.
- We shall go through these steps in detail in the slides that follow.

Instructor Notes:

As many use-cases as possible can be considered at this stage. Creating an Analysis Model is a well defined process and with a good analysis model in place, the design activities will be more effective.



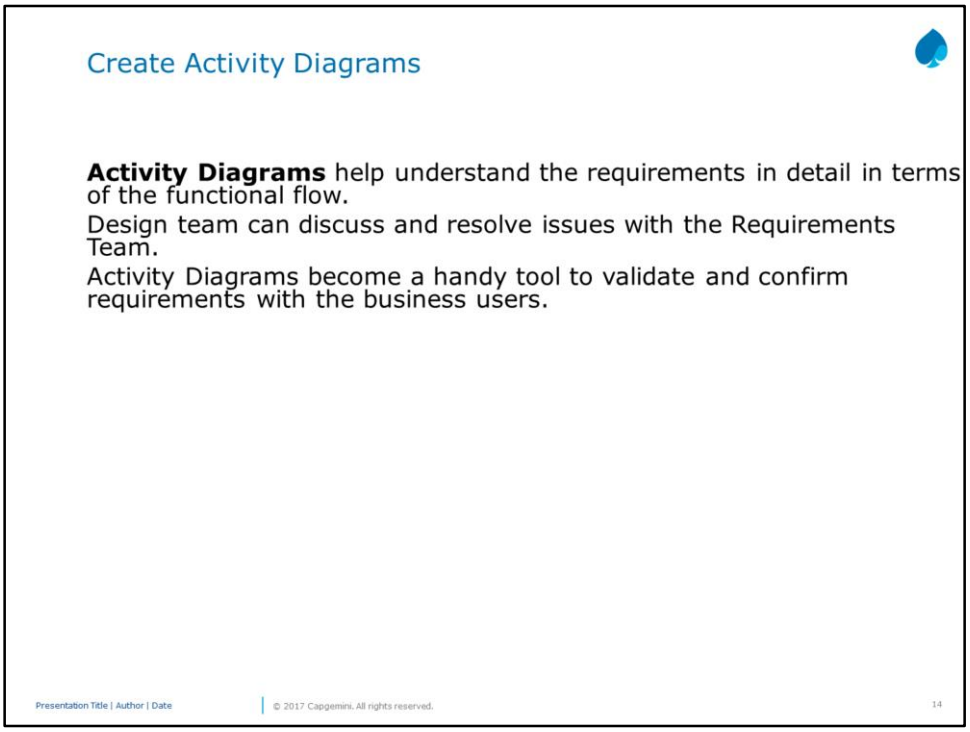
<u>Task1: Create Functional Design</u>:

<u>Step1: Identify Representative Use-Cases</u>:

- The concept of using “representative” use-cases is for reducing repeatability. For example, view page sequences are same in almost all cases. So it will be advisable to model only one and use that as reference for others.
- While choosing representative use-cases, select those that cover the breadth of the system and take care of the unique functionality.
- Note that in the example in the above slide, we have chosen use-cases where all actors (“breadth of application”) and key functionalities get covered.

Instructor Notes:

Ideally the Requirements Analyst Team should already have created use-case descriptions complemented with Activity Diagrams. If not, it is a good idea for the Design Team to perform these activities as it makes the design steps easier to take forward!



Task1: Create Functional Design:

Step2: Create Activity Diagrams or Flow Charts of Representative Use-Cases:

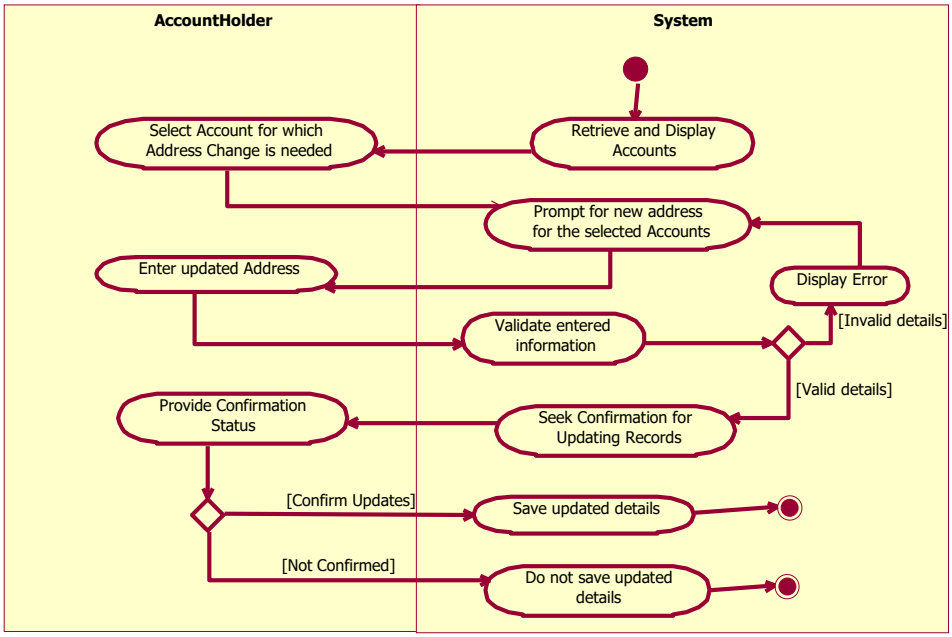
- Earlier, we have discussed about **activity diagrams**. They are a good mechanism to ensure that the Design team is in sync with the Requirements team as far as the functional flow of the requirements is concerned. Furthermore, the notations in activity diagrams are intuitive to understand. Hence the activity diagrams can be used to validate the requirements with business users as well..
- We have also seen earlier that it is ideal to create the activity diagrams along with the requirements artifacts.
  - If the activity diagrams are already created, then the Design team can use them to understand and further refine requirements prior to starting the design activity.
  - If the activity diagrams are not already created, it is recommended that the detailed use-case description along with activity diagrams be created by the Design team. This step is essential because it makes the subsequent steps of design activity easier to be carried out.

Instructor Notes:

Some other notations one might encounter on an activity diagram are:

1. Synchronization Bar, Fork: Indicates concurrent threads of activities after Fork.

2. Synchronization Bar, Join: Indicates that after all concurrent activities are completed, next activity can begin.



Task1: Create Functional Design:

Step2: Create Activity Diagrams or Flow Charts of Representative Use-Cases:

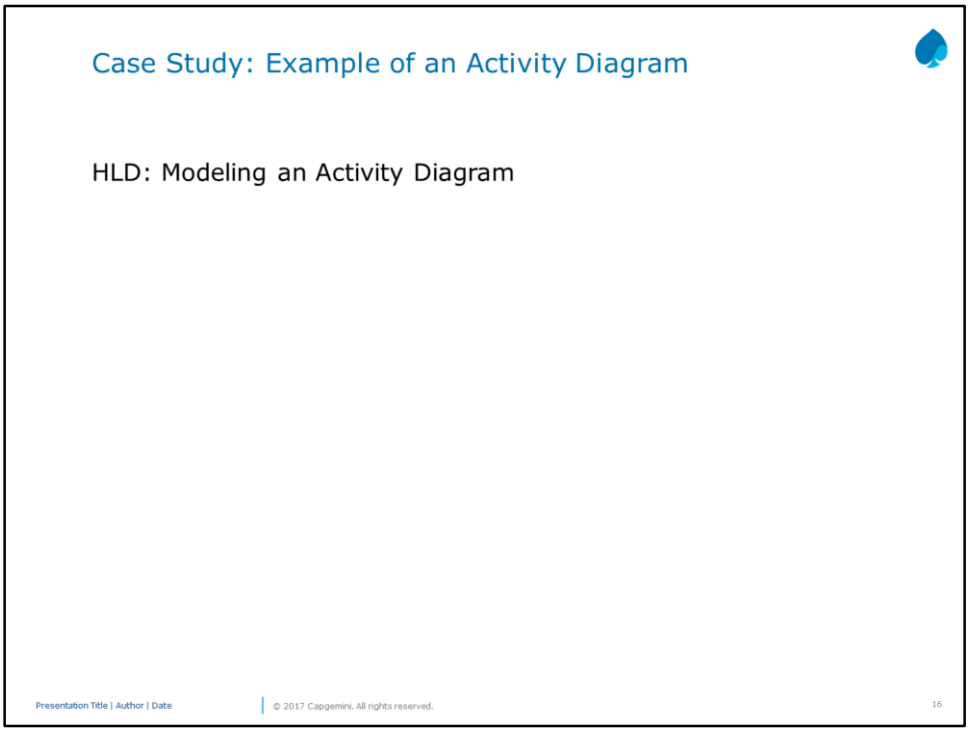
Case Study: Example of an Activity Diagram:

Explanation:

- We have seen this example earlier.
- An activity diagram may include the following elements:
  - **Activity states** represent the performance of an activity or step within the workflow.
  - **Transitions** show an activity state that follows after another.
  - **Decisions** evaluate conditions defined by guard conditions. These guard conditions determine which of the alternative transitions will be made, and thus the activities that are performed. You may also use the **decision** icon to show where the threads merge again. **Decisions** and **guard conditions** allow you to show alternative threads in the workflow of a use-case.
  - **Synchronization bars** show parallel sub-flows. They allow you to show concurrent threads in the workflow of a use-case.
  - **Swimlanes** indicate who is responsible for an activity.

Instructor Notes:

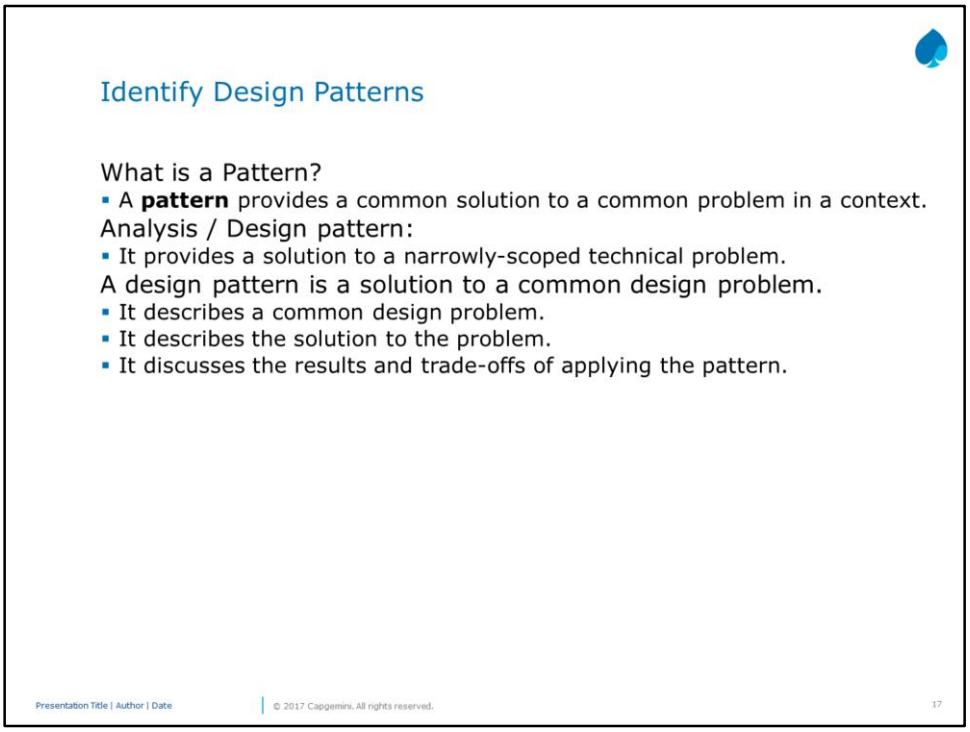
Add an activity diagram for a use case of the case study.



Instructor Notes:

OOAD is not intended to be a patterns course, but will use/introduce patterns to solve particular problems.

As part of Designer Certification, we will separately discuss the Gang Of Four Patterns.

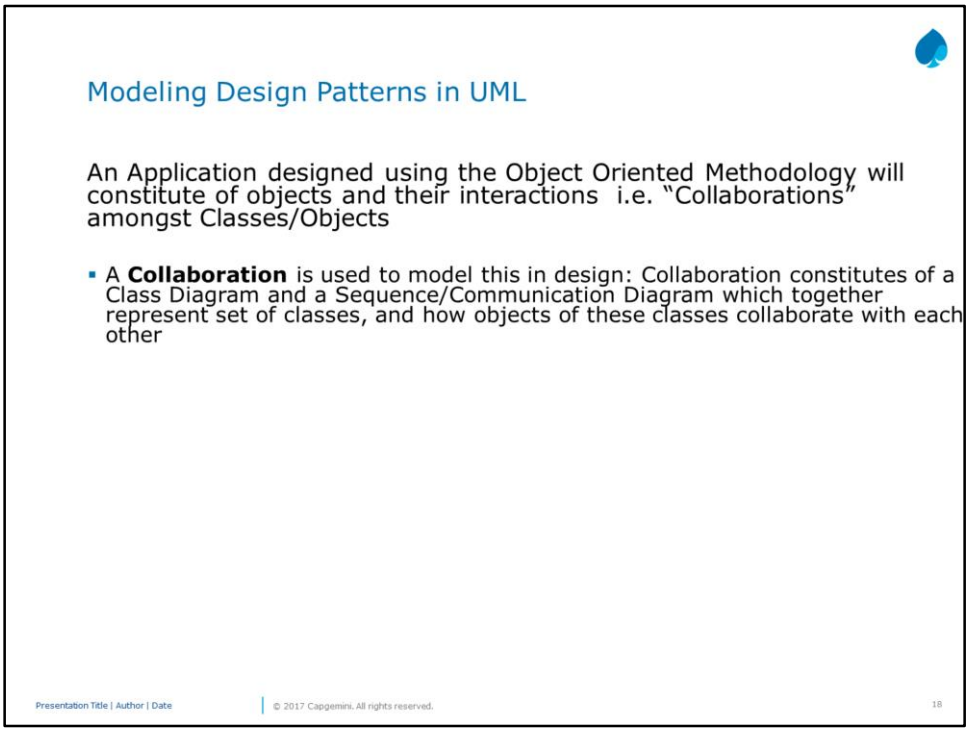


Task1: Create Functional Design:
Step3: Identify Design Patterns:

- Patterns capture best practices that can be reused in a given context.
- Earlier, we have seen some examples of **architectural patterns**. Here we are looking at **design patterns**.
- You can use design patterns to solve issues in your design without "reinventing the wheel". You can also use design patterns to validate and verify your current approaches.
- Using design patterns can lead to more maintainable systems and increased productivity. Design patterns provide excellent examples of good **design heuristics** and **design vocabulary**. In order to use design patterns effectively, you should become familiar with some common design patterns and the issues that they mitigate.

Instructor Notes:

A Pattern in the structural view is modeled using a class diagram; and in the behavioral view as a sequence or collaboration diagram.

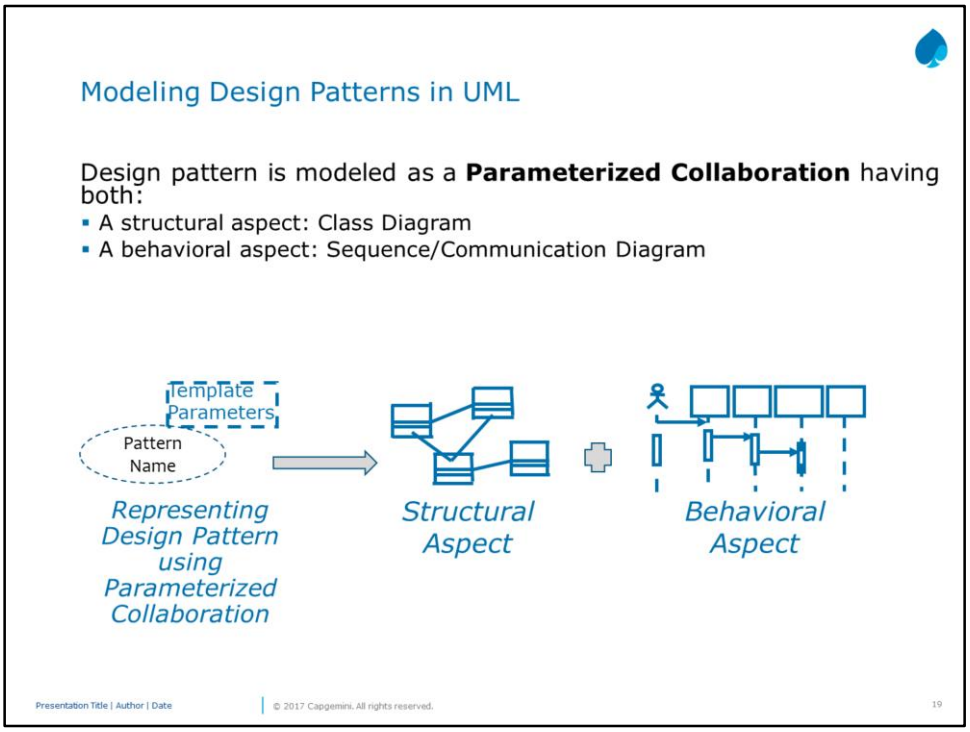


Task1: Create Functional Design:
Step3: Identify Design Patterns:
Modeling Design Patterns in UML:

- A design pattern is modeled in UML as a **parameterized collaboration**. A parameterized collaboration is a template for a collaboration. The **template parameters** are used to adapt the collaboration for a specific usage. These parameters may be bound to different sets of abstractions, depending on how they are applied in the design.
- A **collaboration** has a structural aspect and a behavioral aspect.
  - The structural part is the classes, whose instances implement the pattern, and their relationships (the static view).
  - The behavioral aspect describes how the instances collaborate — usually by sending messages to each other — to implement the pattern (the dynamic view).

Instructor Notes:

A Pattern in the structural view is modeled using a class diagram; and in the behavioral view as a sequence or collaboration diagram.



Task1: Create Functional Design:
Step3: Identify Design Patterns:
Modeling Design Patterns in UML:

- A design pattern is modeled in UML as a **parameterized collaboration**. A parameterized collaboration is a template for a collaboration. The **template parameters** are used to adapt the collaboration for a specific usage. These parameters may be bound to different sets of abstractions, depending on how they are applied in the design.
- A **collaboration** has a structural aspect and a behavioral aspect.
  - The structural part is the classes, whose instances implement the pattern, and their relationships (the static view).
  - The behavioral aspect describes how the instances collaborate — usually by sending messages to each other — to implement the pattern (the dynamic view).

Instructor Notes:

For more details, refer to Qzen guidelines on Performance Principles and Patterns. Some Examples:

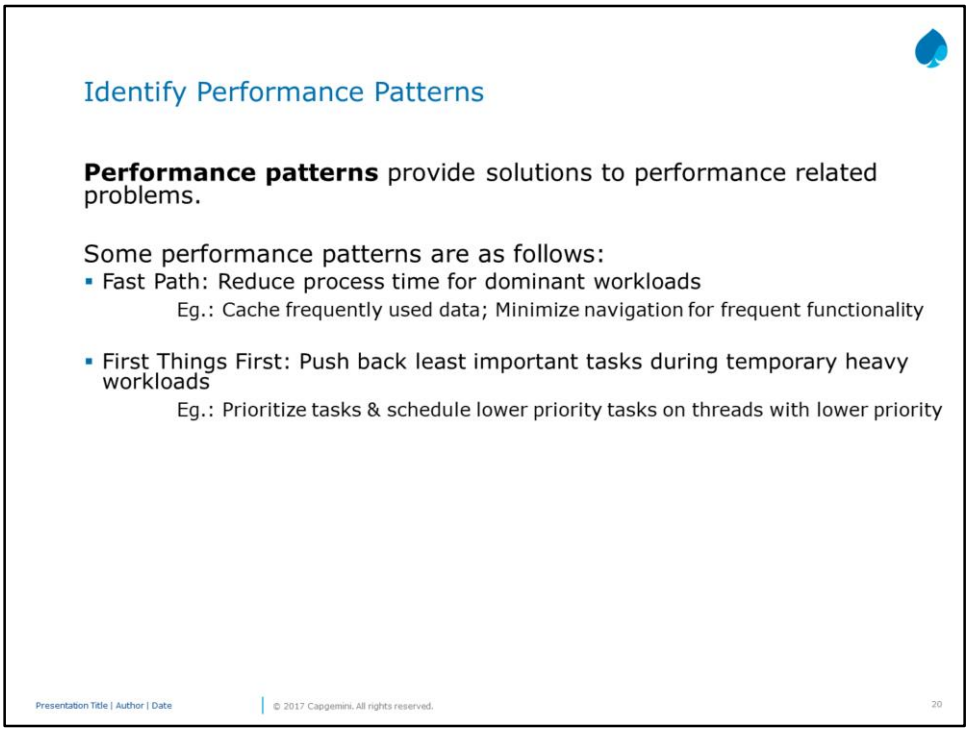
1. Fast Path: HDFC advertises 40% faster withdrawal using "MyFavourite" option. Withdrawal and that too for specific amounts is a dominant workload and system gets coded for it.

2. First Things First: Usually for temporary system overloads; assign priorities for tasks.

3. Coupling: In a web application, distributed requests required to display something can be reduced.

4. Batching: Instead of MQ passing very small messages, MQ messages to a single process can be combined into a single message. This reduces overheads of initialization, processing & termination.

5. Alternate routes: Multithreaded code!



Task1: Create Functional Design:
Step4: Identify Performance Patterns:

- At the Implementation level, **performance patterns** use the underlying **architecture pattern** and **design patterns**.
- Examples:
  1. **Fast Path:** Improving response time by reducing process time required for dominant workloads.
  2. **First Things First:** Omit the least important tasks if it is not possible to execute all tasks in given time.
  3. **Coupling:** Aggregate frequently used data / functions to reduce number of interactions.
  4. **Batching:** Combine requests into a batch for saving on overhead processing time.
  5. **Alternate routes:** Spread high usage objects spatially to reduce contention time.
- Refer QZen for more details on Performance Patterns. Technical Architect will need to be closely involved here since these are activities pertaining to performance (Non Functional Requirement).

Instructor Notes:

For more details, refer to Qzen guidelines on Performance Principles and Patterns. Some Examples:

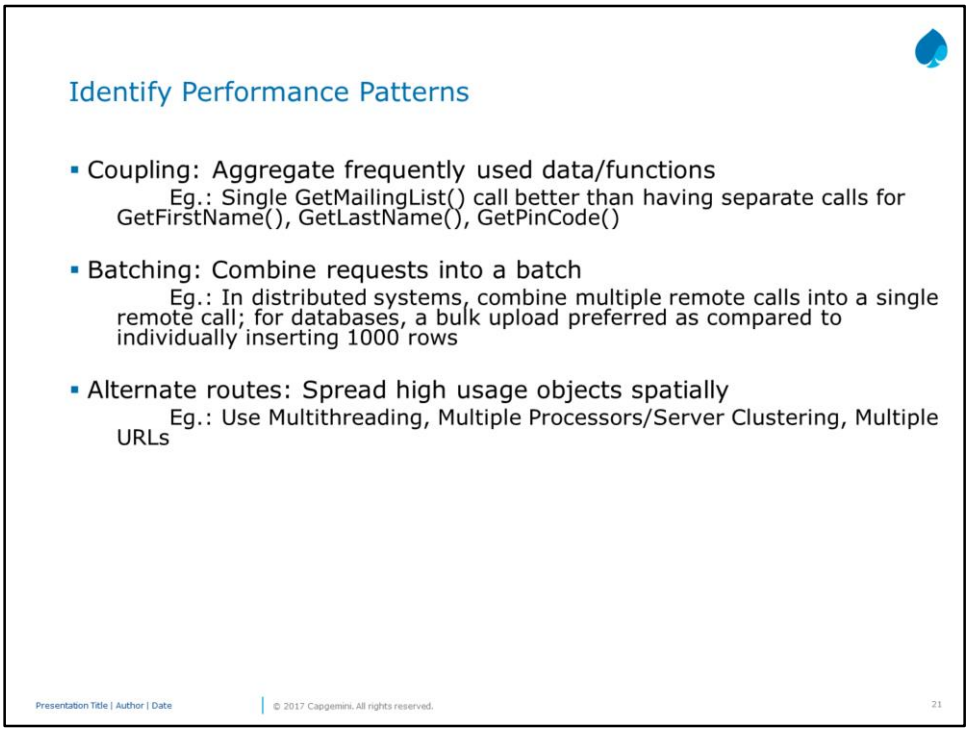
1. Fast Path: HDFC advertises 40% faster withdrawal using "MyFavourite" option. Withdrawal and that too for specific amounts is a dominant workload and system gets coded for it.

2. First Things First: Usually for temporary system overloads; assign priorities for tasks.

3. Coupling: In a web application, distributed requests required to display something can be reduced.

4. Batching: Instead of MQ passing very small messages, MQ messages to a single process can be combined into a single message. This reduces overheads of initialization, processing & termination.

5. Alternate routes: Multithreaded code!



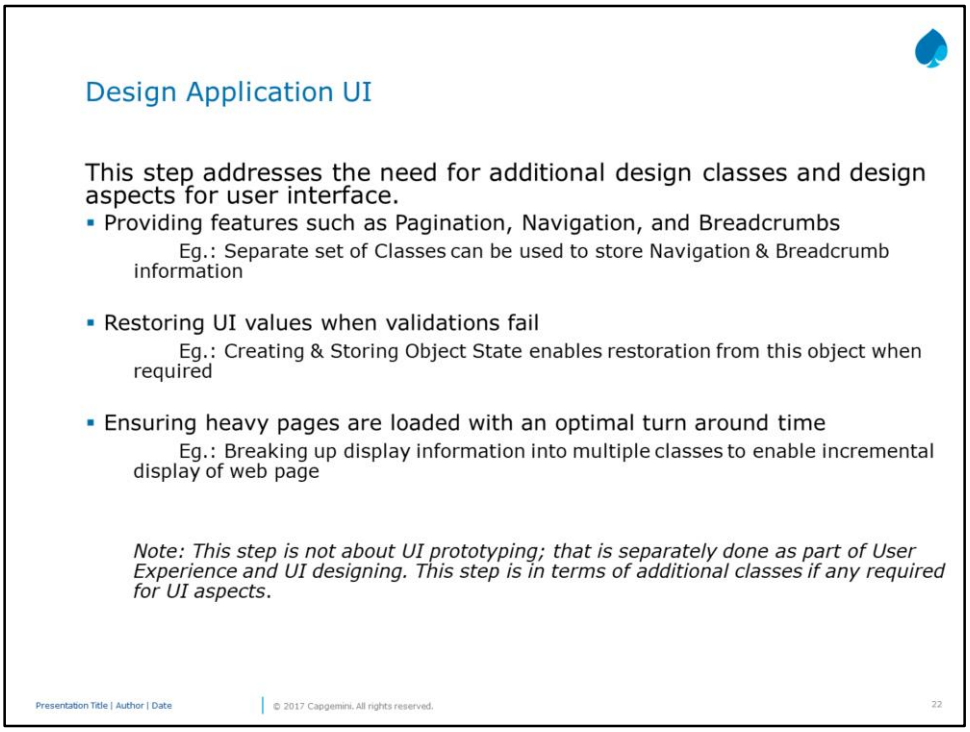
Task1: Create Functional Design:

Step4: Identify Performance Patterns:

- At the Implementation level, **performance patterns** use the underlying **architecture pattern** and **design patterns**.
- Examples:
  1. **Fast Path:** Improving response time by reducing process time required for dominant workloads.
  2. **First Things First:** Omit the least important tasks if it is not possible to execute all tasks in given time.
  3. **Coupling:** Aggregate frequently used data / functions to reduce number of interactions.
  4. **Batching:** Combine requests into a batch for saving on overhead processing time.
  5. **Alternate routes:** Spread high usage objects spatially to reduce contention time.
- Refer QZen for more details on Performance Patterns. Technical Architect will need to be closely involved here since these are activities pertaining to performance (Non Functional Requirement).

Instructor Notes:

UI Prototyping could possibly be done in the Requirements phase itself.



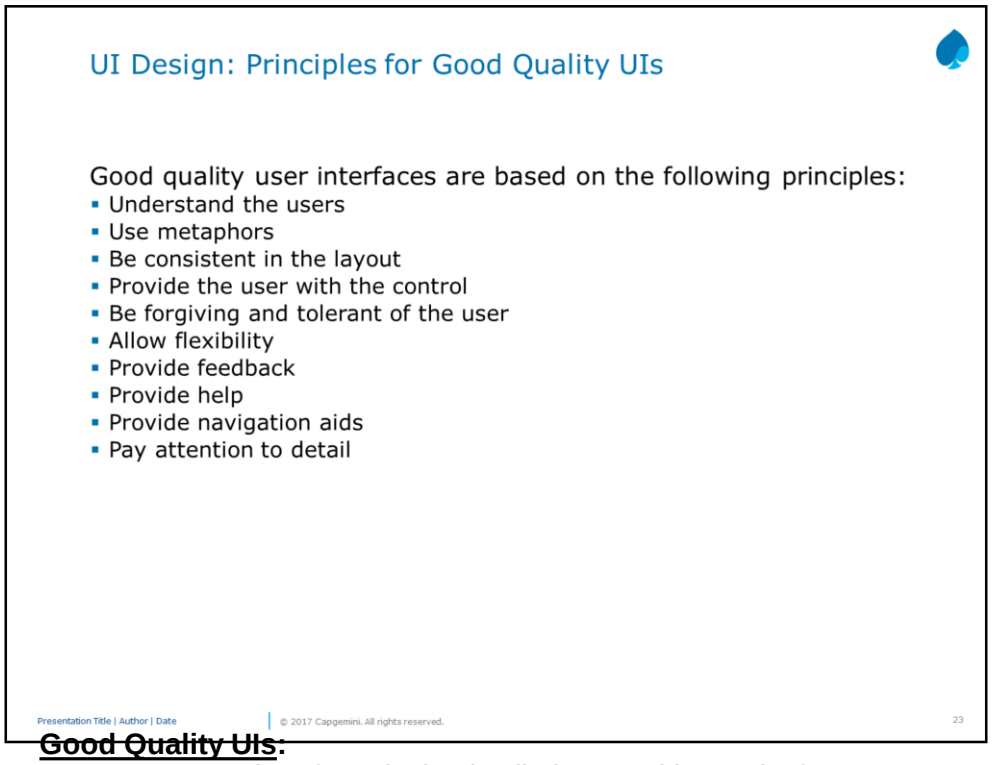
Task1: Create Functional Design:
Step5: Design Application UI:

- This step is not about UI prototyping. However, this step is more about classes that will be needed in the system to take care of UI related aspects such as those mentioned on slide.
- Additional classes will be needed to help in the UI aspects and these are to be taken into account at this step.

Instructor Notes:

Taking a detour to delve into UI design related aspects. In this section we briefly talk about guiding principles for UI design, some GUI components/ and some tips for UI design .

Notes are quite detailed….explain the same with examples.

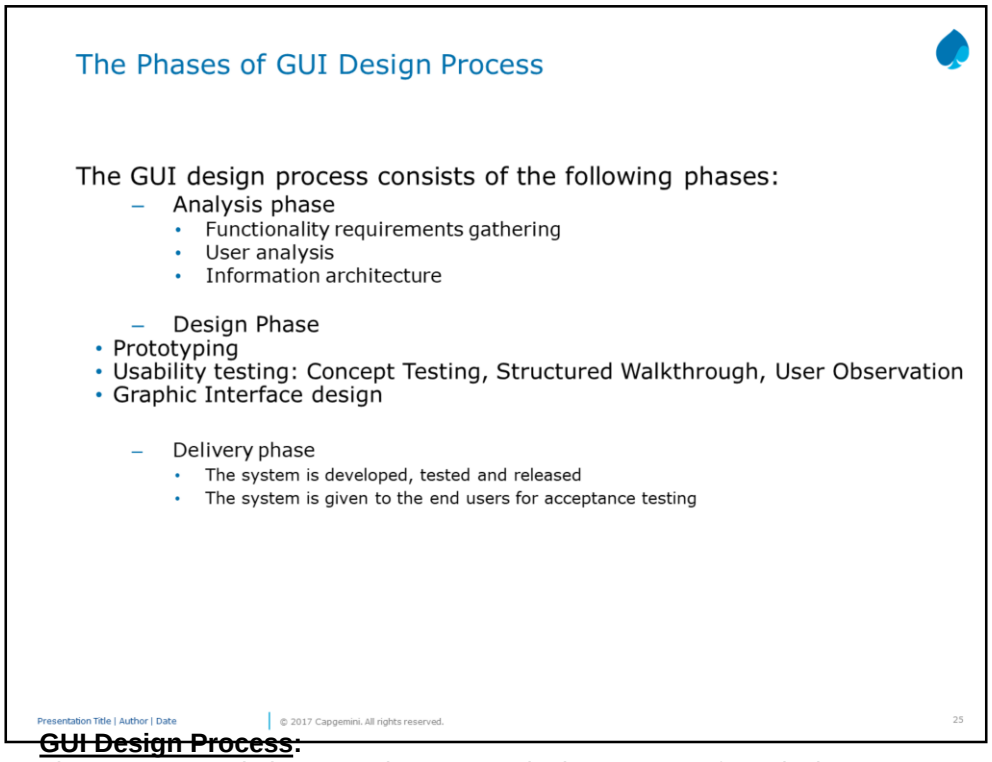


Good Quality UIs:

- A great user interface design is all about making tasks fast, easy, and pleasant to perform. The challenge of creating a helpful interface, however, increases with the complexity of the users' goals and tasks. Interfaces that go beyond good design and actively support goal definition and task performance deserve special recognition and attention.
- Some of the principles that are followed while designing a good UI are as follows:
  - **Understand the Users:** Software developers often make two mistakes. They assume that all users are similar, and that all users are like software developers. Users have different backgrounds, experience, work environments, preferences, job requirements, skills, and learning styles. For example, users of an accounting system in a University may have different needs to those of users in a stock exchange.
  - **Use metaphors:** Metaphors are conceptual analogies that help the user to transfer existing knowledge to the application interface. Metaphors can facilitate the construction of users' mental models of the application, reduce learning time, and make new environments less threatening. The prototypical example of a metaphor is the computer desktop with folders, an inbox, and documents.

Instructor Notes:

Explain briefly the overall approach. Focus is on analysis and design phases.



GUI Design Process:

There are several phases and processes in the User Interface design.

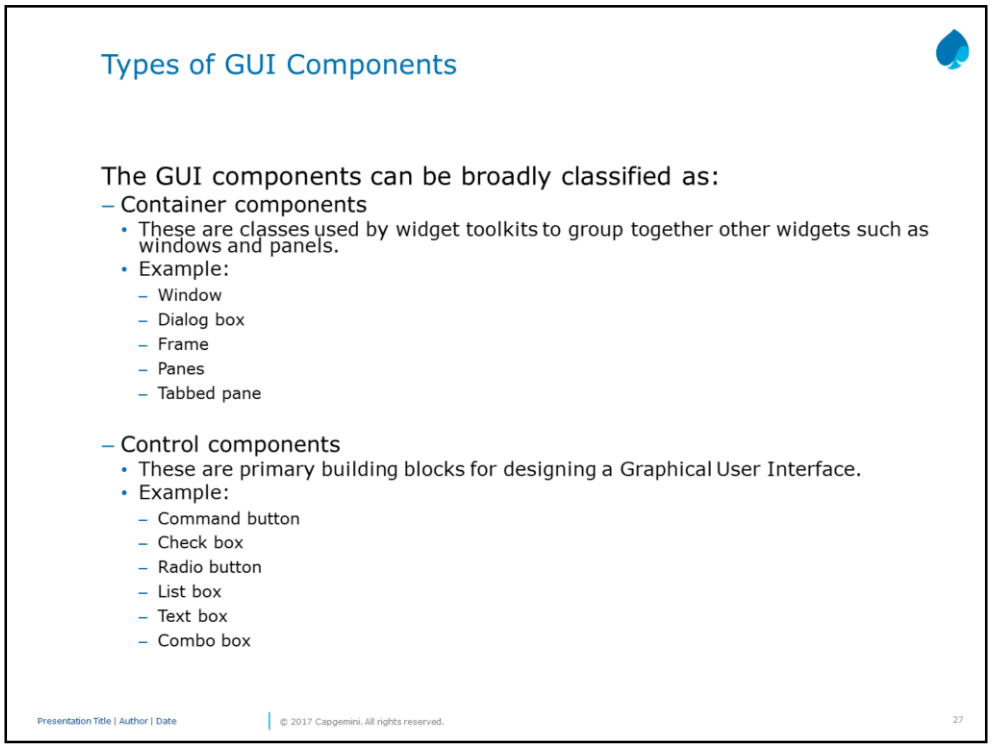
1. **Analysis Phase**

- **Functionality requirements gathering:** It involves assembling a list of the functionality required of the system to accomplish the goals of the project and the potential needs of the users. Users need to understand the tasks they perform with an application. By observing how users approach their work, how they meet their goals, and how they form relationships between objects and concepts, one can design an interface that will most closely mirror users' current conceptual models.
- **User analysis:** It involves analysis of the potential users of the system either through discussion with people who work with the users and/or the potential users themselves. Details of users' backgrounds, experience, job and task characteristics, work preferences, and learning styles are all important in the design of the interface.
  Typical questions asked are:
  - What would the user want the system to do?
  - How would the system fit in with the user's normal workflow or daily activities?
  - How technically savvy is the user and which similar systems does the user already use?
  - What interface look and feel styles appeal to the user?
- **Information architecture:** It involves development of the process and/or information flow of the system.
  For example:
  a) For phone tree systems, the architecture will be an option tree flowchart.
  b) For web sites, the architecture will be a site flow that shows the hierarchy of the pages.

Instructor Notes:

There could be other types of UI too like command line based and for console/batch applications etc. However, considering the wide spread of GUI based applications for OO systems we are discussing that.

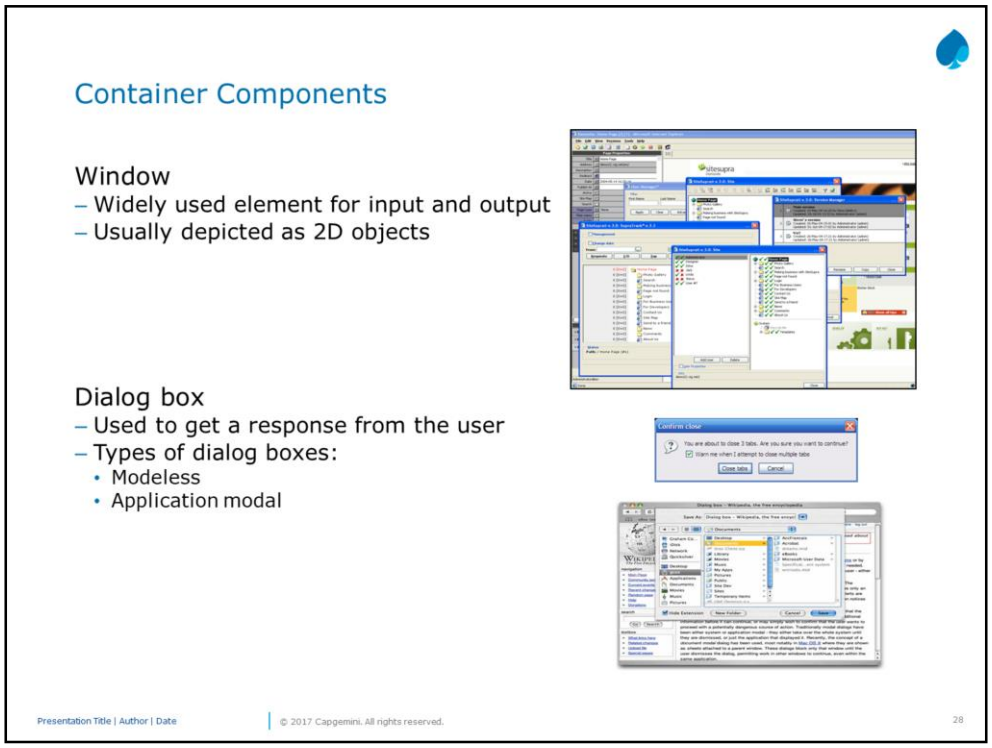
Types & examples of GUI components are explained on this and subsequent slides.



<u>Container Components</u>:
Apart from their graphical properties, the **container components** have the same type of behavior as **container classes**, as they keep a list of their child widgets, and allow to add, remove, or retrieve widgets amongst their children.
Control components are the basic building blocks of GUI.

Instructor Notes:

Bring into picture concept of events and event handlers. GUI components will help raise events and in our design, appropriate event handling functionality would be required.



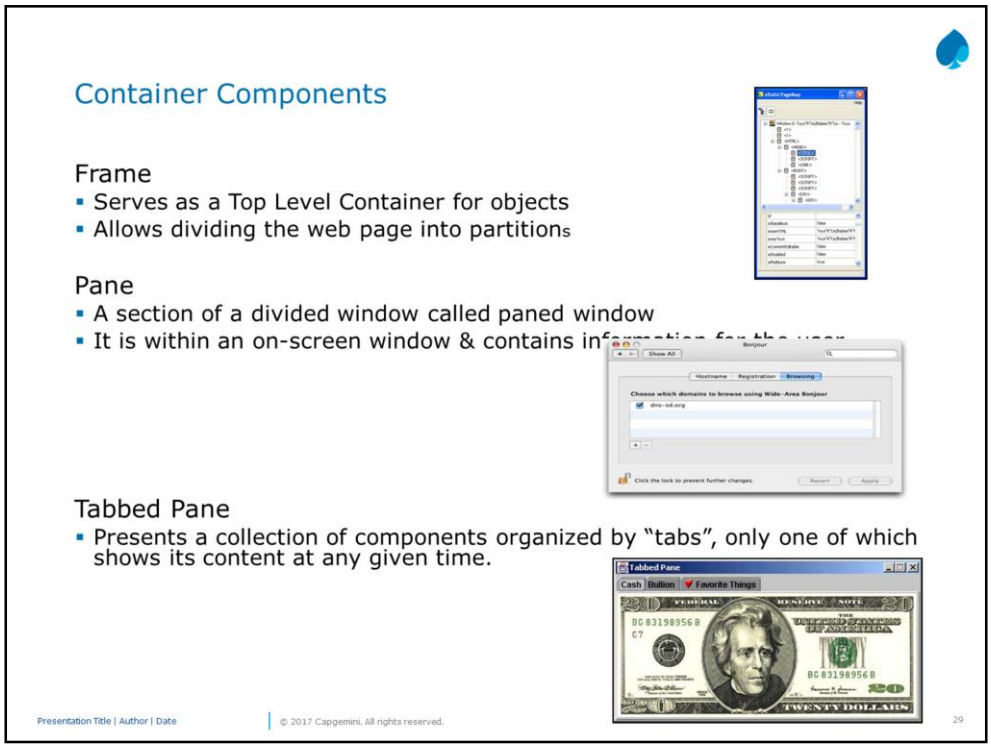
Window:

- Windows are almost always depicted as **two-dimensional objects** (such as papers or books) arranged on a desktop. Most windows can be resized, moved, hidden, restored, and closed at will. When two windows overlap, one is on top of the other, with the covered part of the lower window not being visible.
- A **Graphical User Interface (GUI)** that uses **windows** as one of its main metaphors is called a **windowing system**. Many programs with text user interfaces, for example Emacs, allow their display to be divided into **areas** which may also be referred to as "**windows**".
- The part of a **windowing system** which manages these operations is called a **Window Manager**.

Dialog Box

- There are two types of dialog boxes:
  - **Modeless:** Non-modal or modeless dialog boxes are used when the requested information is not essential to continue, and so the window can be left open while work continues elsewhere. A type of modeless dialog box is a **toolbar** which is either separate from the main application, or may be detached from the main application, and items in the toolbar can be used to select certain features or functions of the application. In general, good software design calls for dialogs to be of modeless type, where possible. This is because they do not force the user into a particular mode of operation.
    An example might be a dialog of settings for the current document, for example, the background and text colors. The user can continue adding text to the main window, whatever color it is, but can change it at any time using the dialog. (This is not meant to be an example of the best possible option for modeless dialog box. Often the same functionality may be accomplished by toolbar buttons on the application's main window).
  - **Application modal:** Modal dialog boxes are those which temporarily halt the program in the sense that the user cannot continue until the dialog has been closed. The program may require some additional information before it can continue, or may simply wish to confirm that the user wants to proceed with a potentially dangerous course of action.
    Traditionally, modal dialogs have been either **system** or **application modal** — they either take over the whole system until they are dismissed, or just the application that displayed it. Recently, the concept of a **document modal dialog** has been used, most notably in Mac OS X, where they are shown as sheets attached to a parent window.
    These dialogs block only that window until the user dismisses the dialog, permitting work in other windows to continue, even within the same application.

Instructor Notes:



Frame:

- A **frame** is a rectangular segment within a browser's window that can be scrolled independently of other such segments. A frame is a top-level window with a title and a border.
- Frames are used on web sites similar to the way in which applications display multiple windows.
  - It enables static data to be visible all the time while other data are scrolled. For example, a menu can be located at the top of the page with links to articles below.
  - The articles can be scrolled without changing the position of the menu on the page.
  - The frame may contain content from a different site, just like links on Web sites can retrieve Web pages from any server.
- Frames automatically provide **scroll bars** if the content is larger than the frame window.
- The **HTML frames** feature is used to partition the page into **windows**. The **content** within a window can be **scrollable** or **static**.

Tabbed Pane:

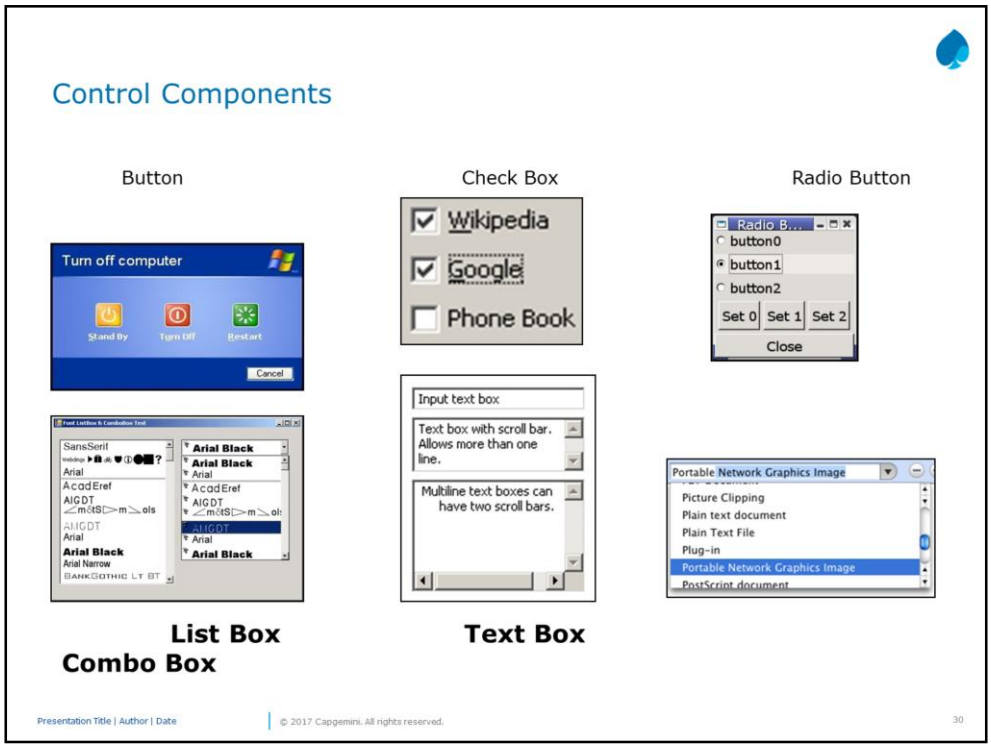
To see the content of another tab, you simply need to click on the tab's title.

Example:

- An example of a **tabbed pane** can be a function in a **web browser** that hides the current web page behind a tab and presents a new blank window for continued browsing.
- All browsers keep track of pages visited, and clicking **Back** and **Forward** moves a user through them. However, after a long period of web surfing, the pages of most interest are often interspersed between a slew of others. Tabbed browsing creates multiple browsing sessions within the same browser window.

Instructor Notes:

Co-relate that these could be made available through class attribute values for eg.



Command Button (sometimes known as a **command button** or **push button**) provides the user a simple way to trigger an event, such as searching for a query at a search engine, or interacting with dialog boxes, like confirming his/her actions. Eg. OK, Cancel, Exit.

Check Box (checkbox, tickbox, or tick box) permits the user to make multiple selections from a number of options.

Radio Button or **option button** allows the user to select one of a predefined set of options.

List Box allows the user to select one or more items from a list contained within a static, multiple line text box.

Text Box (also referred as **text field** or **text entry box**) allows the user to input text information that has to be used by the program. User-interface guidelines recommend:

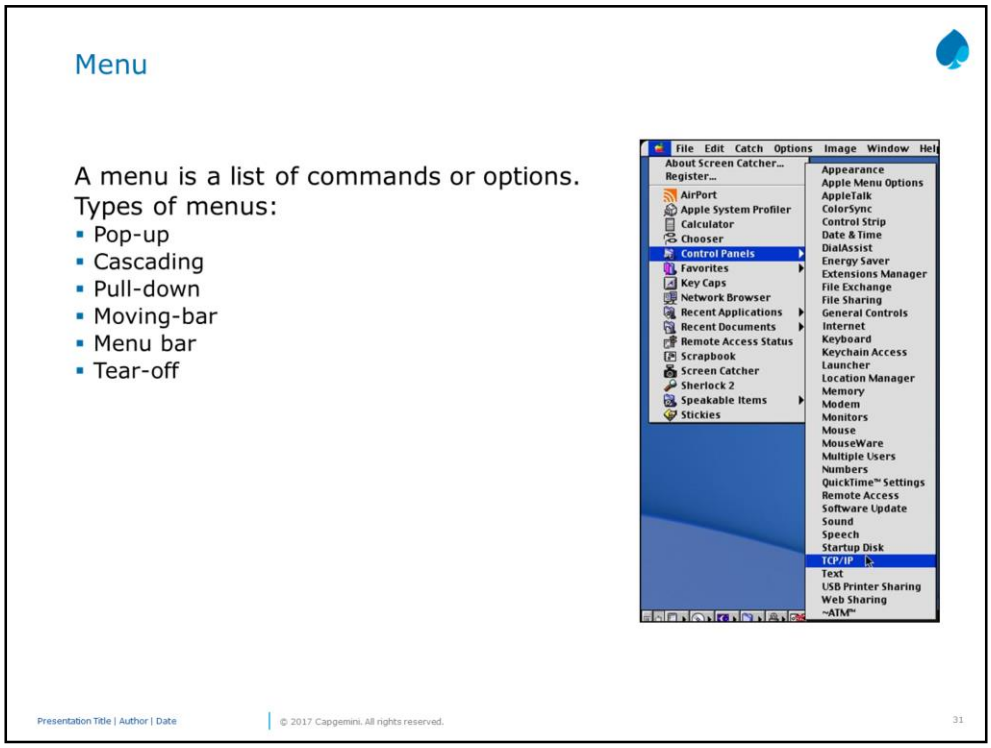
- A **single-line text box** when only one line of input is required
- A **multi-line textbox** only if more than one line of input may be required

Non-editable text boxes can serve the purpose of simply displaying text.

Combo Box is a combination of a drop-down list or list box and a single-line textbox, allowing the user either to type a value directly into the control or select from the list of existing options. Eg. Address bar of graphical web browsers.

Instructor Notes:

Menus are extensively used. Again co-relate this with corresponding objects and event handlers that may be needed.



Menu:

- A **menu** is a list of commands presented to an operator by a computer or communications system. They may be thought of as **shortcuts** to frequently used **commands** that spare the operator from having a detailed knowledge or recall of syntax.
- A **menu** is used in contrast to a command line interface where instructions to the computer are given in the form of **commands** (or **verbs**).

Types of Menus:

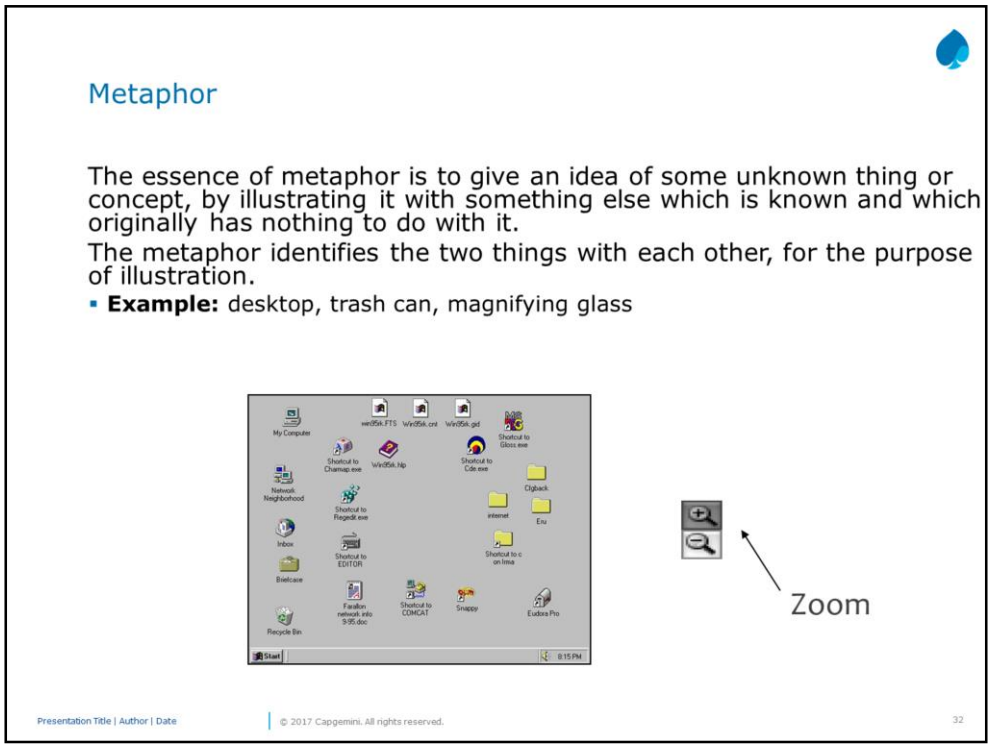
- **Pop-up menu:** It is a menu that appears temporarily when you click the mouse button on a selection. Once you make a selection from a pop-up menu, the menu usually disappears.
- **Cascading menu:** It is a submenu that opens when you select a choice from another menu.
- **Pull-down menu:** It is a special type of pop-up menu that appears directly beneath the command that you selected.
- **Moving-bar menu:** It is a menu in which options are highlighted by a bar that you can move from one item to another. Most menus are moving-bar menus.
- **Menu bar:** It is a menu that is arranged horizontally. Each menu option is generally associated with another pull-down menu that appears when you make a selection.
- **Tear-off menu:** It is a pop-up menu that you can move around the screen like a window.

Choices given from a menu may be selected by the operator by a number of methods (called interfaces):

1. Depressing one or more keys on the keyboard or mouse
2. Positioning a cursor or reverse video bar by using a keyboard or mouse
3. Using an electromechanical pointer, such as a light pen
4. Touching the display screen with a finger
5. Speaking to a voice-recognition system

Instructor Notes:

One metaphor may in turn lead to set of classes in design!



Metaphor:

- The metaphors' role in the user interface is to facilitate learning, orientation, and the formation and maintenance of the concept about the program, that is a **mental model**.
- The most widely used metaphor is the **desktop**. A significant part of office systems is based on it. The program, as it appears on the screen, is identified with the top of an office desk, on which documents and tools can be placed. Documents can usually be seen in individual, partially overlapping "windows", as if they were partially on top of each other. Documents can be arranged into "folders", they can be modified, appended to each other, destroyed, and so on.
- It is an accepted practice to use composite metaphors. A program may have an overall, global metaphor (like the desktop mentioned above), and besides or within that it can have other sub-metaphors assigned to subtasks or operations (for example: the concept of carbon copy in an e-mail system).
- Metaphors are used to assist expression or understanding. Metaphors are very much part of our everyday lives. The use of metaphors can have a significant impact on end users. In an end-user interface, metaphors can provide cues for the recognition of symbolism.

Instructor Notes:

Designing reports is another critical area and one has to think of classes that will be required for the same.

Designing Reports

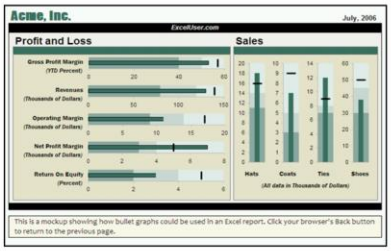
Choose the right reporting style depending on information to be presented and the target audience for whom the report is intended

- The **Tabular Report** output is organized in a multicolumn, multirow format, with each column typically corresponding to a column selected from the database.
- A **Graphical Report** depicts information in a visual manner using charts, bar graphs, and so on.


YOUR Inc. COMPANY

Company Departments

Department Id	Department Name	Manager Id	Location Id
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700

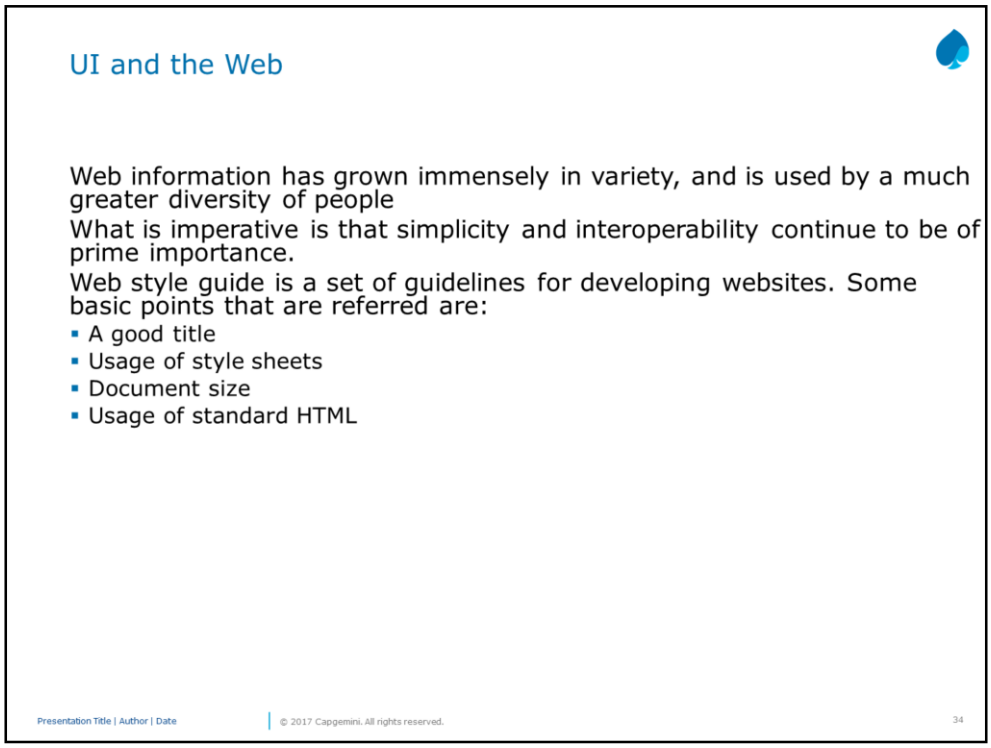


Presentation Title | Author | Date | © 2017 Capgemini. All rights reserved. 33

Reports will tend to be a mix of Tabular information and Graphical representation of data.

Instructor Notes:

Focus on importance of Web Style Guide.

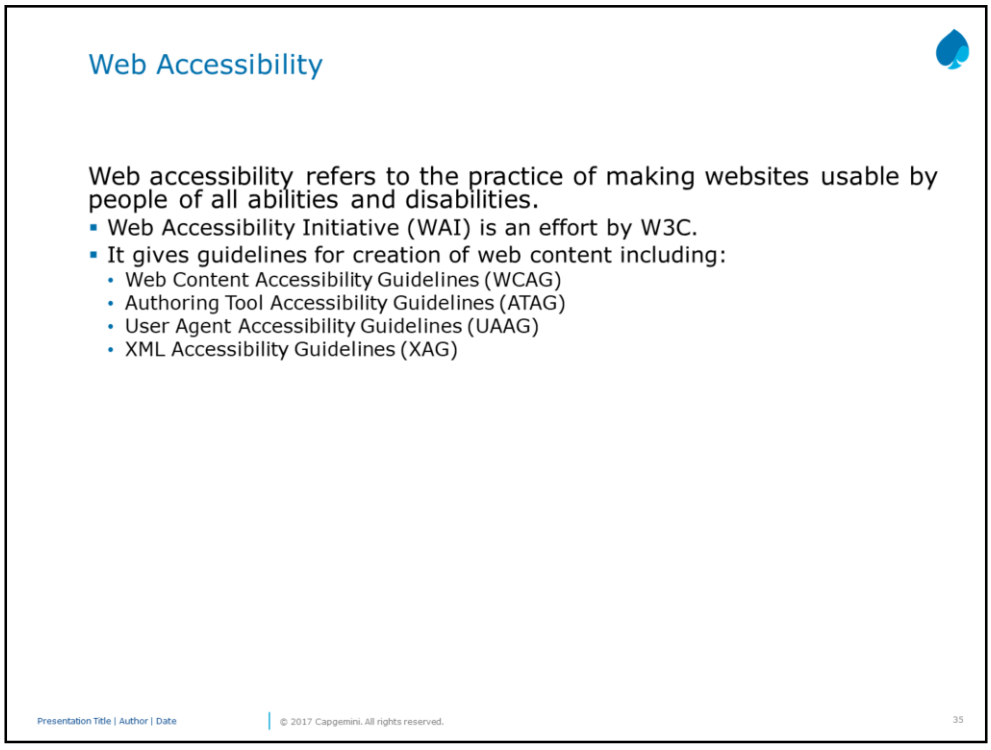


Style Guides:

- Style guides focus on a publication's visual and technical aspects, prose style, best usage, grammar, punctuation, spelling, and fairness.
- Let us discuss some of the aspects that are of prime importance to a style sheet:
  - **Title:** It should identify the content of the document in a fairly wide context.
  - **Usage of Style Sheets:** A style sheet tells a browser how to render a page. Once you have a style sheet, you can use it for many documents. When you do that the browser caches the style sheet, which saves download time. You can change the look and feel of all your documents just by changing the style sheet. Since it keeps your HTML clean, putting the style in a separate sheet helps you with device independence and provides easy access for those with disabilities. You can have separate style sheets for the printed version of a document.
  - **Document Size:** There are two upper limits on a document's size. One is that long documents will take longer to transfer, and so a reader will not be able to simply jump to it and back as fast as he or she can think. The other limit is the difficulty for a reader to scroll through large documents. Readers with character based terminals don't generally read more than a few screens. A rough guide, then, for the size of a document is:
    - For online help, menus giving access to other things: small enough to fit on 24 lines. Check this by using a terminal browser.
    - For textual documents, of the order of half a letter-sized (A4) page to 5 pages.
  - **Usage of standard HTML:** There are two reasons for using standard HTML.
    - It is understood in the same way by software across the world, it helps making your document "device independent".
    - Data you are creating will remain accessible in the future. You want it to be independent of future evolutions in software.

Instructor Notes:

Web accessibility is becoming more and more important and today, there are government backed compliance norms for web accessibility especially in certain geographies.



Web Accessibility:

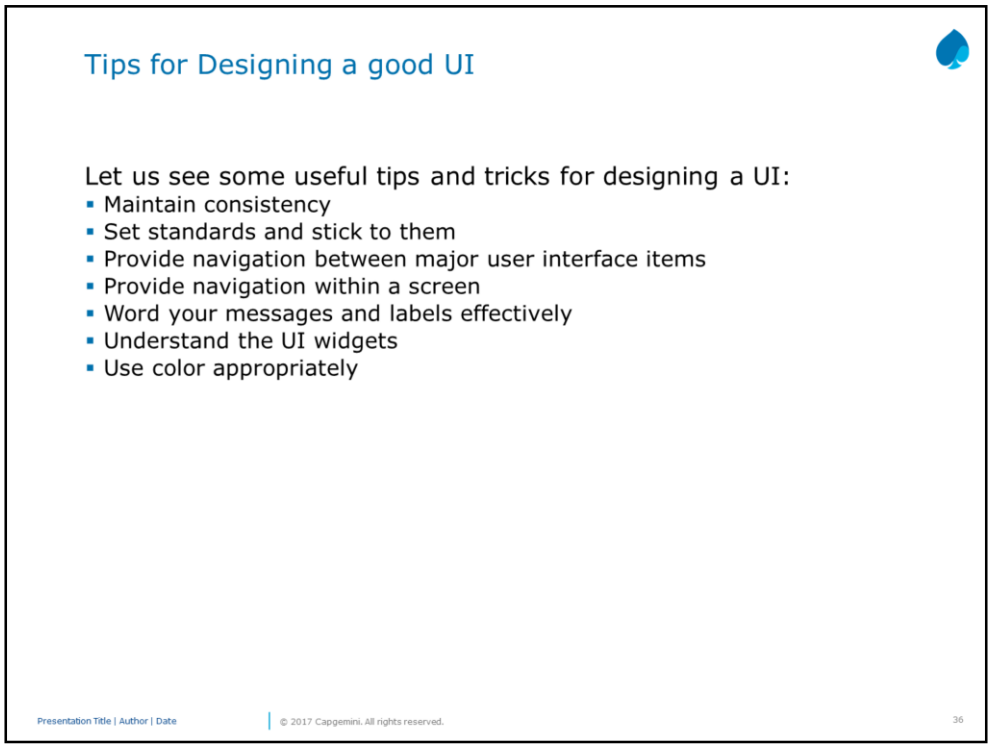
- When sites are correctly designed, developed and edited, all users can have equal access to information and functionality. For example, when a site is coded with semantically meaningful HTML, with textual equivalents provided for images and with links named meaningfully, this helps blind users using text-to-speech software and/or text-to-Braille hardware.
- When text and images are large and/or enlargeable, it is easier for users with poor sight to read and understand the content. When links are underlined (or otherwise differentiated) as well as colored, this ensures that color blind users will be able to notice them. When clickable links and areas are large, this helps users who cannot control a mouse with precision.
- When pages are coded so that users can navigate by means of the keyboard alone, or a single switch access device alone, this helps users who cannot use a mouse or even a standard keyboard.
- When videos are closed captioned or a sign language version is available, deaf and hard of hearing users can understand video.
- When flashing effects are avoided or made optional, users prone to seizures caused by these effects are not put at risk.
- When content is written in plain language and illustrated with instructional diagrams and animations, users with dyslexia and learning difficulties are better able to understand the content.

The needs that Web accessibility aims to address include:

- **Visual:** These include visual impairments such as blindness, various common types of low vision and poor eyesight, various types of color blindness.
- **Motor/Mobility:** These include difficulty or inability to use the hands, including tremors, muscle slowness, loss of fine muscle control, and so on, due to conditions such as Parkinson's Disease, muscular dystrophy, cerebral palsy, and stroke.
- **Auditory:** These include deafness or hearing impairments, including individuals who are hard of hearing.
- **Seizures:** This includes Photoepileptic seizures caused by visual strobe or flashing effects.
- **Cognitive/Intellectual:** These include developmental disabilities, learning disabilities (dyslexia, dyscalculia, and so on), and cognitive disabilities of various origins, affecting memory, attention, developmental "maturity", problem-solving, logic skills, and so on.

Instructor Notes:

Summarize the section



Useful Tips and Tricks:

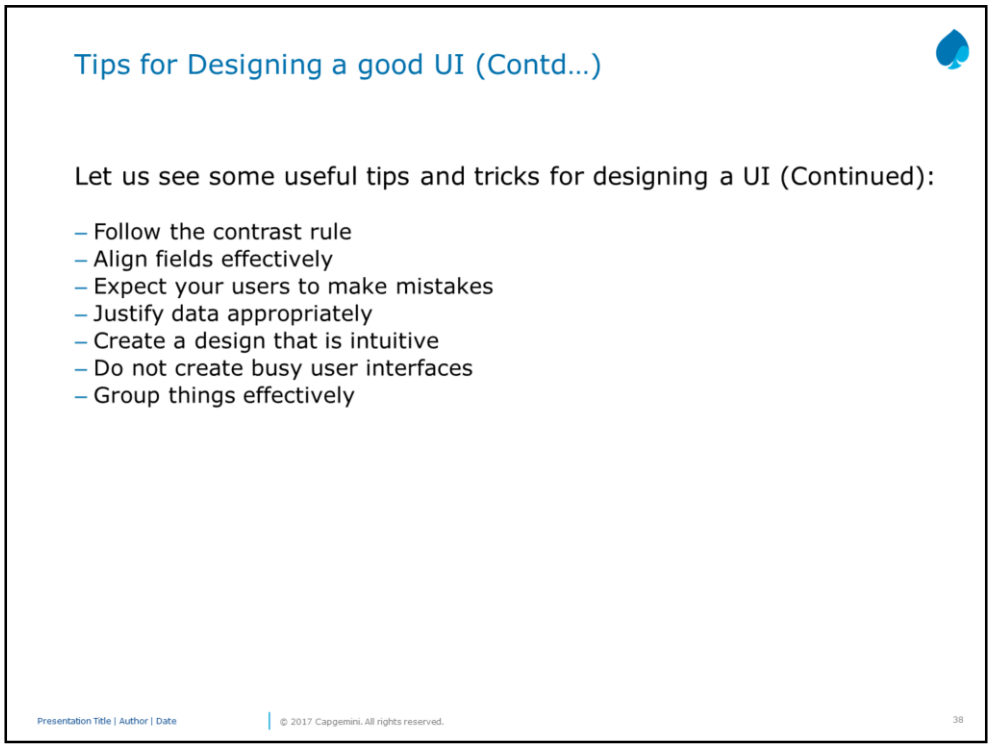
Following are some tips and techniques that should prove valuable for designing a UI:

- **Consistency:** The most important thing you can possibly do is ensure your user that the interface works consistently. If you can double-click on items in one list and have something happen, then you should be able to double-click on items in any other lists and have the same sort of thing happen. Put your buttons in consistent places on all your windows, use the same wording in labels and messages, and use a consistent color scheme throughout.
- **Set standards and stick to them:** The only way you can ensure consistency within your application is to set user interface design standards, and then stick to them.
- **Provide navigation between major user interface items:** If it is difficult to go from one screen to another, then your users will quickly become frustrated and give up. When the flow between screens matches the flow of the work that the user is trying to accomplish, then your application will make sense to your users.

Instructor Notes:

Summarize the section… though participants will not immediately apply the UI design concepts in this course, as a designer, they need to be familiar with UI design concepts.

Also mention about UX team and increasing focus on User Experience in today's applications.

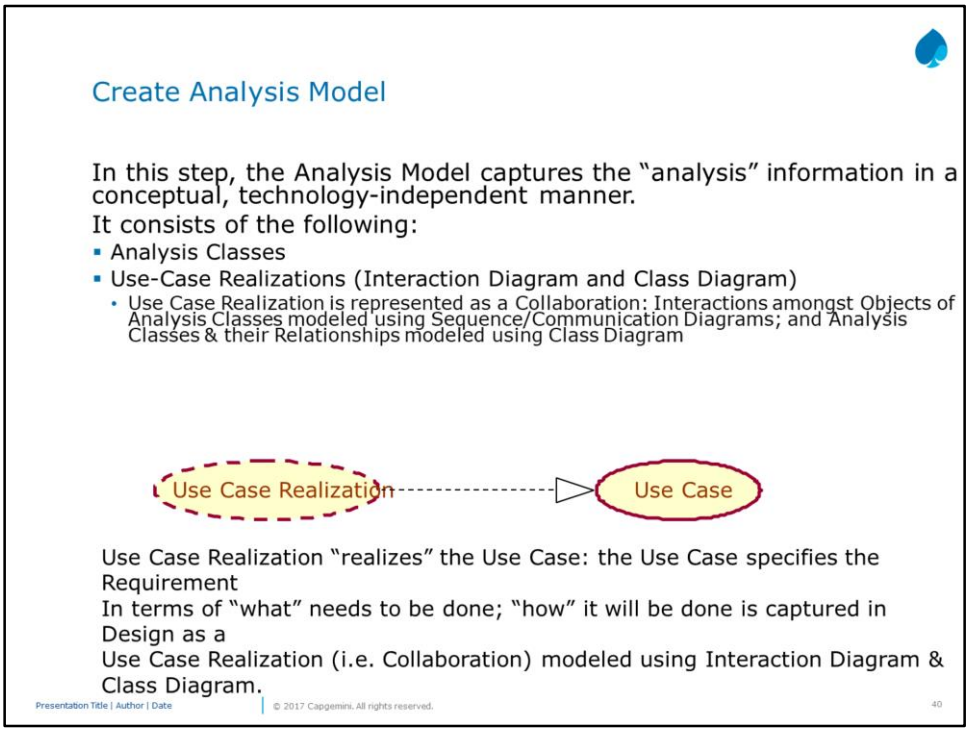


Useful Tips and Tricks (contd.):

- **Follow the contrast rule:** If you are going to use color in your application, you need to ensure that your screens are still readable. The best way to do this is to follow the **contrast rule**.
  - Use dark text on light backgrounds and light text on dark backgrounds.
  - Reading blue text on a white background is easy, but reading blue text on a red background is difficult. The problem is that there is not enough contrast between blue and red to make it easy to read, whereas there is a lot of contrast between blue and white.
- **Align fields effectively:** When a screen has more than one editing field, you want to organize the fields in a way that is both visually appealing and efficient. The best way to do so is to **left-justify** the **edit fields**. In other words, make the left-hand side of each edit field line up in a straight line, one over the other. You can **right-justify** the **corresponding labels** and place them immediately beside the field. This is a clean and efficient way to organize the fields on a screen.
- **Expect your users to make mistakes:** How many times have you accidentally deleted some text in one of your files or in the file itself? Were you able to recover from these mistakes or were you forced to redo hours, or even days, of work? The reality is that to err is human, so you should design your user interface to recover from mistakes made by your users.

Instructor Notes:

We will be creating the analysis model for a sample use-case.

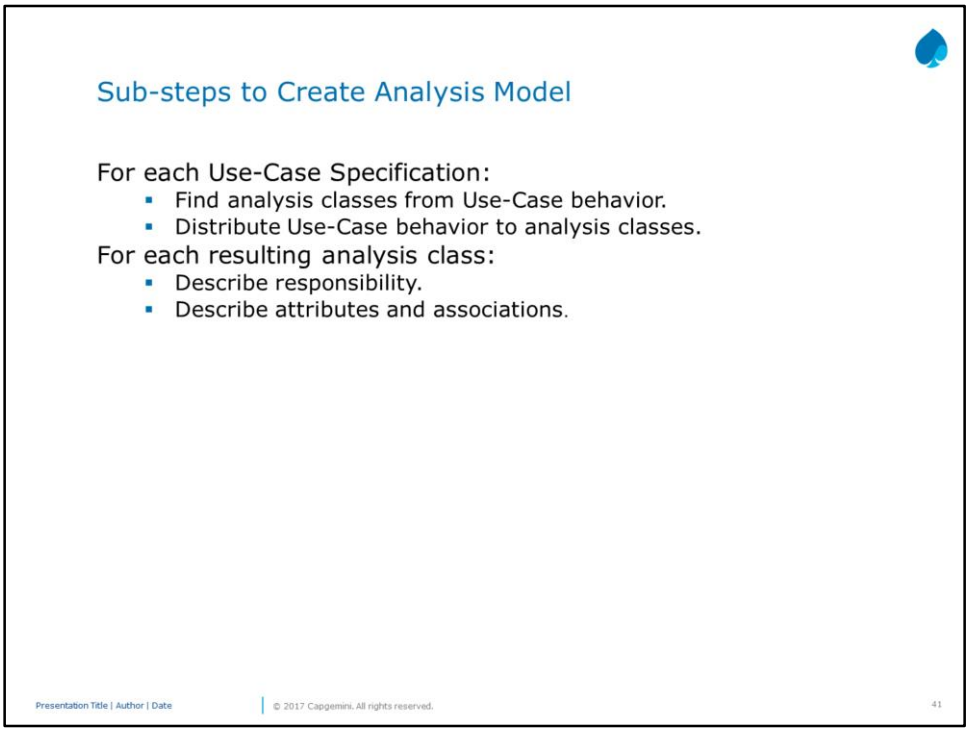


Task1: Create Functional Design:
Step6: Create Analysis Model:

- In this step, we now focus on creating the **UML Analysis Model**. As seen earlier, the Analysis Model emphasizes the conceptual details of the functional requirements (use-cases). Technology specific details do not come into picture in this step.
- The Analysis Model consists of the **Analysis Classes** and **collaborations** between them, which are modeled as use-case realizations.
- The Analysis Classes will be used as a basis to move to the design for the use-cases.

Instructor Notes:

Analysis Classes and Use-Case Realizations are done for each use-case being designed.



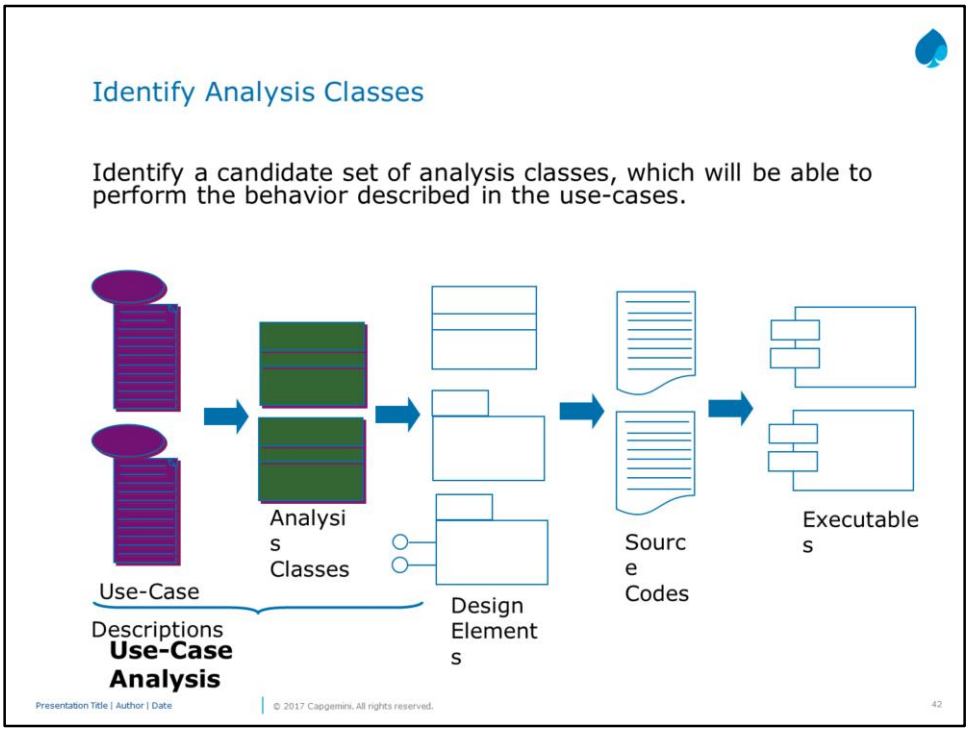
Task1: Create Functional Design:
Step6: Create Analysis Model:

- For each use-case, the Analysis Classes capable of performing the behavior described in use-cases will be identified.
- The behavior of the use-case will be distributed across the identified Analysis Classes.
- Based on this distribution, details for a class and the manner in which the objects of a class will interact with each other will be identified.

Instructor Notes:

Detailed use-case description helps in identification of analysis classes.

We can identify the analysis classes even if there are no detailed descriptions, but in that case more effort is likely to go into that activity.

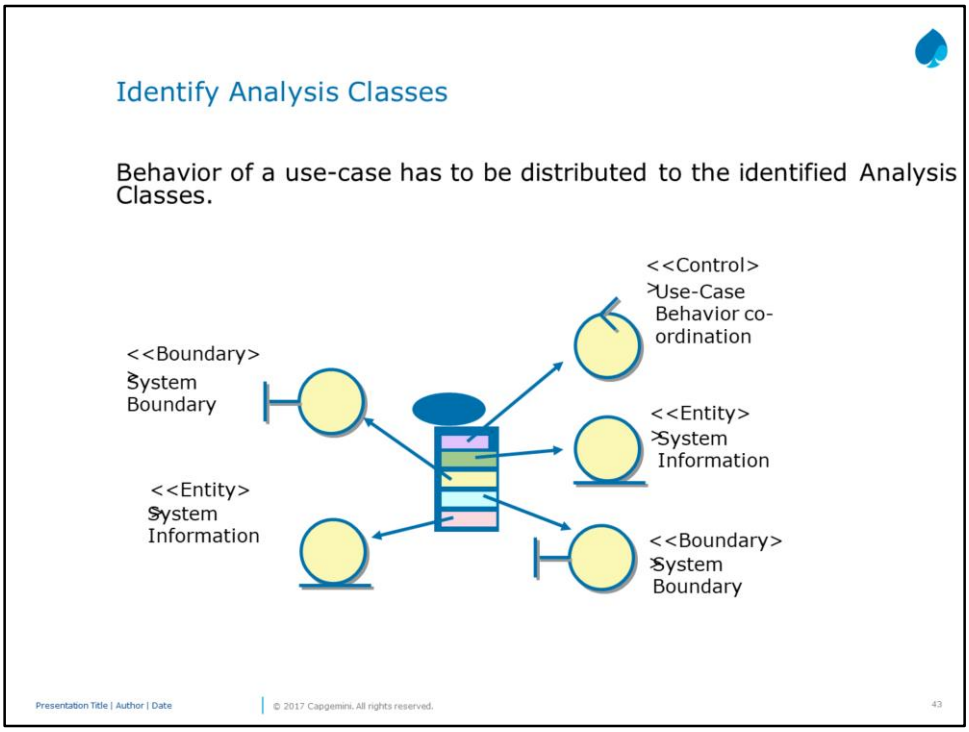


<u>Task1: Create Functional Design</u>:
<u>Step6: Create Analysis Model</u>:
Identify Analysis Classes:

- The use-case descriptions form the inputs for coming up with Analysis Classes. As we move into detailed design, these Analysis Classes will be refined to come up with design elements.
- Analyzing the use-cases with an objective of coming up with the **Analysis Model** is called **Use-Case Analysis**.

Instructor Notes:

Entire behavior would fit into one of the three categories we see here.

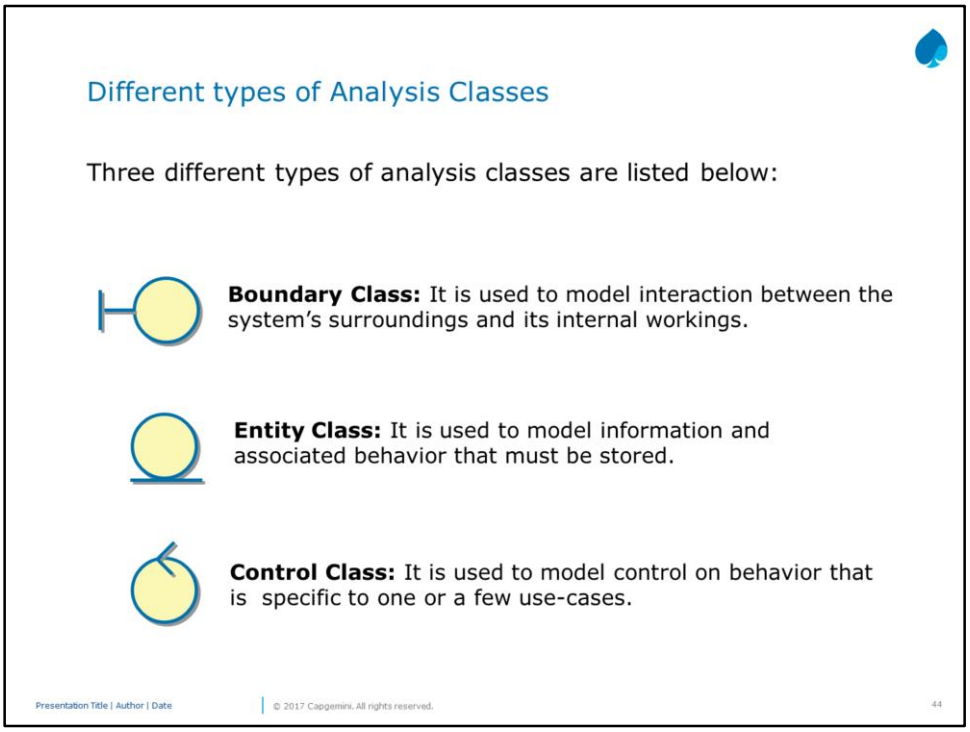


Task1: Create Functional Design:
Step6: Create Analysis Model:
Identify Analysis Classes:

- Analysis classes are identified based on the flow of events of the use-case. We shall identify three kinds of analysis classes. (They are explained in subsequent slides.)
- After the classes have been identified, it is important to distribute the use-case behavior to those classes, using **Analysis Use-Case Realization**.

Instructor Notes:

As mentioned before, these classes will get further refined as we get into design. Important to understand that at this stage, they form a kind of “Place Holder” to ensure that the complete use-case behavior is captured.



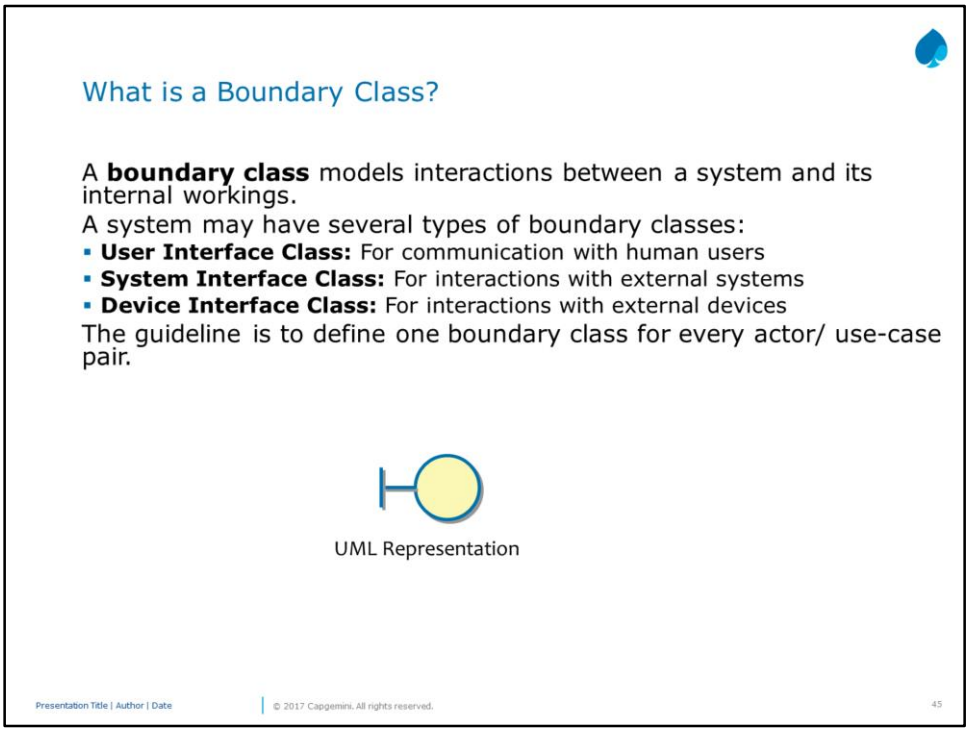
<u>Task1: Create Functional Design</u>:

Different types of Analysis Classes:

- There are three perspectives:
  - The boundary between the system and its actors.
  - The information that the system uses.
  - The control logic of the system.
- Based on these three perspectives, Analysis Classes are stereotyped as:
  - Boundary
  - Entity
  - Control
- Use of stereotypes helps in developing a robust object model. Changes made to the model only affect a specific area. For example:
  - Changes made to the user interface will affect only the boundary classes.
  - Changes in the control flow, will affect only control classes.
  - Changes in long-term information will affect only entity classes.
- These stereotypes help in identifying classes only during analysis and early design phase.

Instructor Notes:

A good use-case diagram along with the guideline defined makes it easy to identify all boundary classes.



What is a Boundary Class?

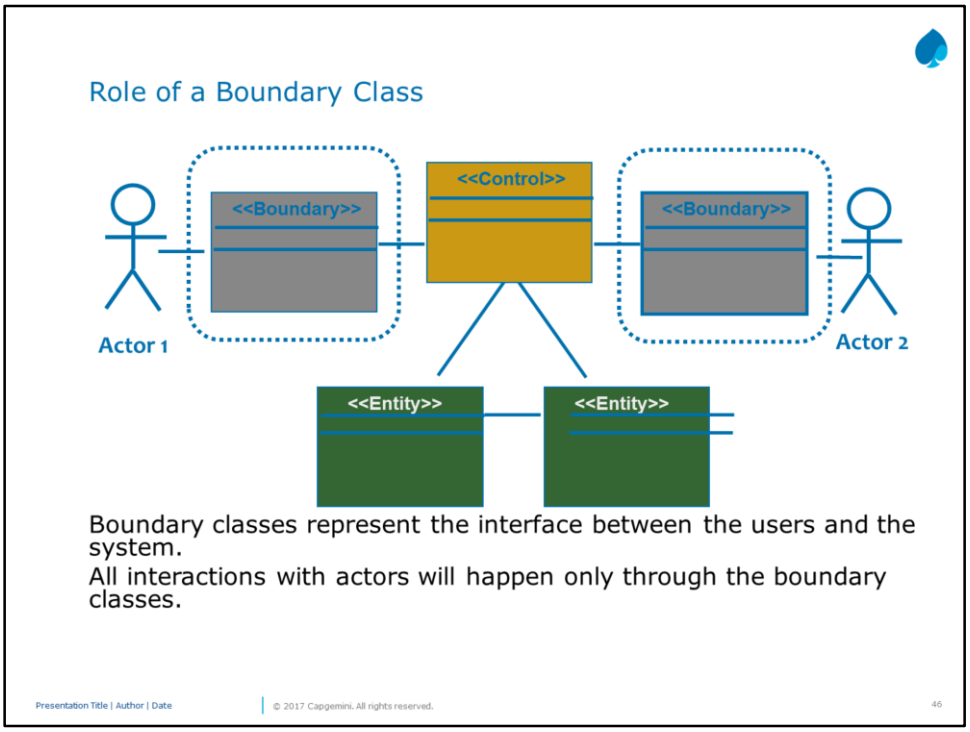
- A **Boundary Class** models those objects that are interfaces between the system and its surroundings. As a result, any changes made in the GUI or the Communication Protocol requires a corresponding change only in the boundary classes.
- Some common examples of boundary classes are given below:
  - Windows, printer interfaces, communication protocols, sensors, and terminals
- Normally, several boundary classes are present in a system, such as:
  - **User-Interfaces Classes:** They represent intermediate communication between the system and the human users.
  - **System-Interface Classes:** They represent intermediate communication with other systems.
  - **Device-Interface Classes:** They represent the interface to devices that detect external events.

Modeling Boundary Classes:

- Boundary classes should be modeled according to the boundary they represent. For example:
  - Interface between two systems
  - Interface between a system and a human actor
- Objectives of the boundary classes will vary accordingly.
  - In the first example, the communication protocol is of prime importance.
  - In the second example, the User Interface is important.

Instructor Notes:

Important to emphasize that all interactions with the actors has to be routed through boundary classes.

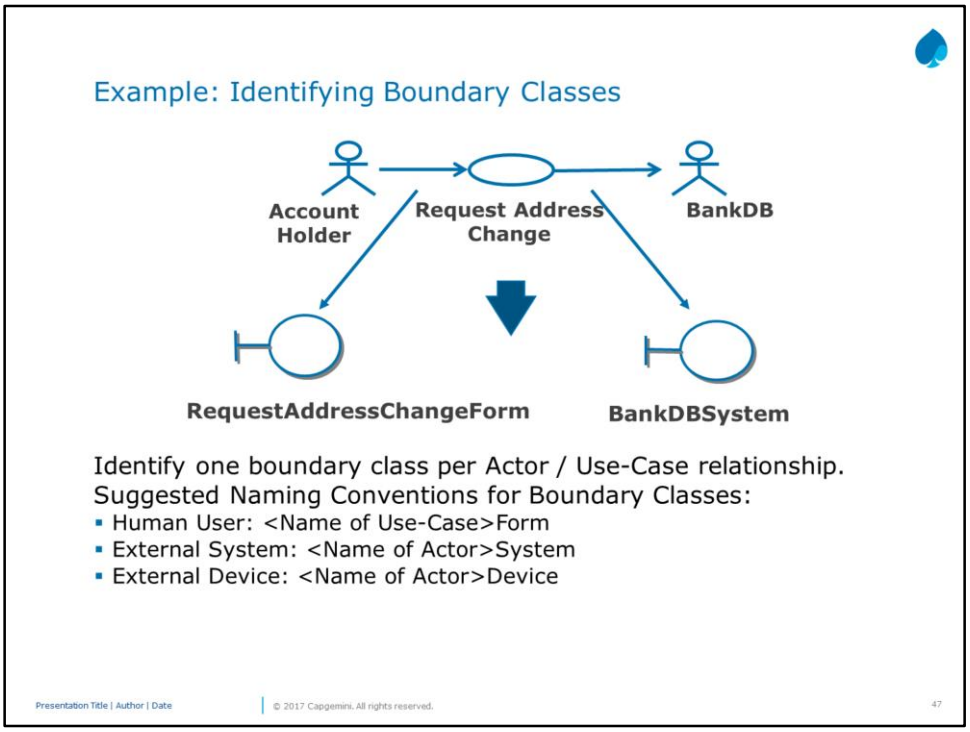


Role of a Boundary Class:

- **Boundary classes** serve as an intermediary or an interface to something outside the system. **Boundary classes** model those parts of the system that depend on its surroundings. **Entity classes** model those parts of the system that are independent of its surroundings. Therefore changes in the **GUI** or **Communication Protocol** only requires a change in the Boundary classes, whereas Entity and Control classes remain the same.
- Boundary classes help in identifying the system's boundary. This allows the Designer to get an overall idea of the system. Designers are able to identify related services during early stages of the development life-cycle.

Instructor Notes:

Following some predefined naming conventions help when working in large teams as there would be consistency across the Analysis elements identified.

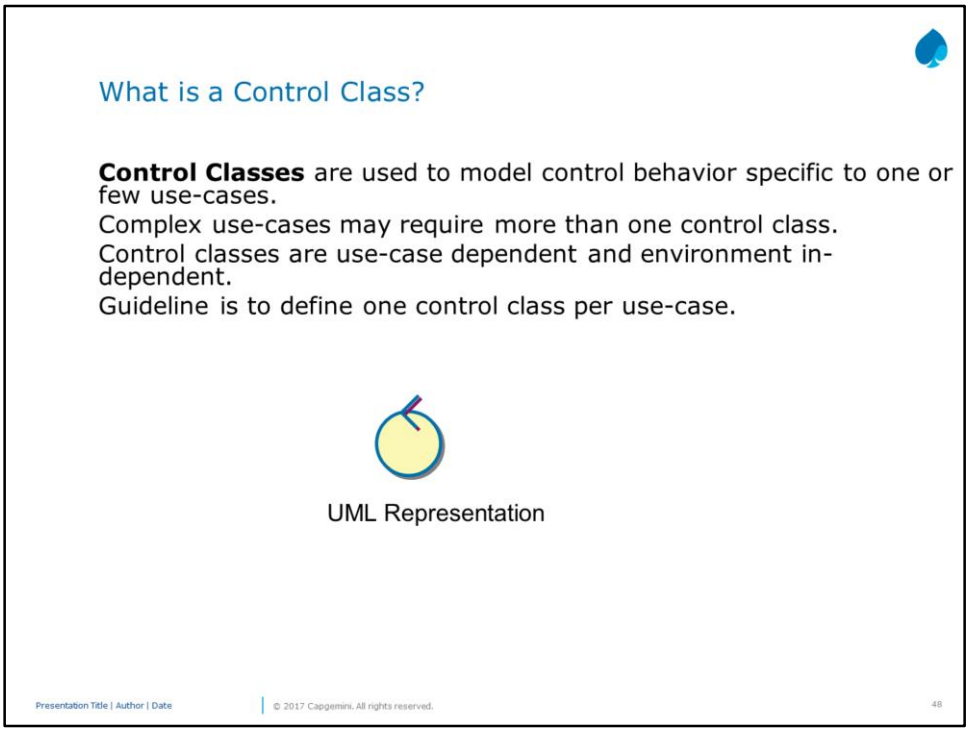


Example: Identifying Boundary Classes:

- The slide shows an example of identifying boundary classes.
- You should always identify one boundary class per actor/use-case pair.
- Note the naming conventions:
  - **Human User:** <Name of Use-Case>Form
  - **External System:** <Name of Actor>System
  - **External Device:** <Name of Actor>Device
- You should concentrate on identifying boundary classes and defining their responsibilities without focusing on details. The primary aim, at this stage, is to get a proper understanding of the problem at hand. These boundary classes would be just place holders which will get refined as we move into designs.

Instructor Notes:

Control Class is like the Project Manager who would run the show for the project, ensuring that the right actions are taken by the right members for the overall closure of the project.

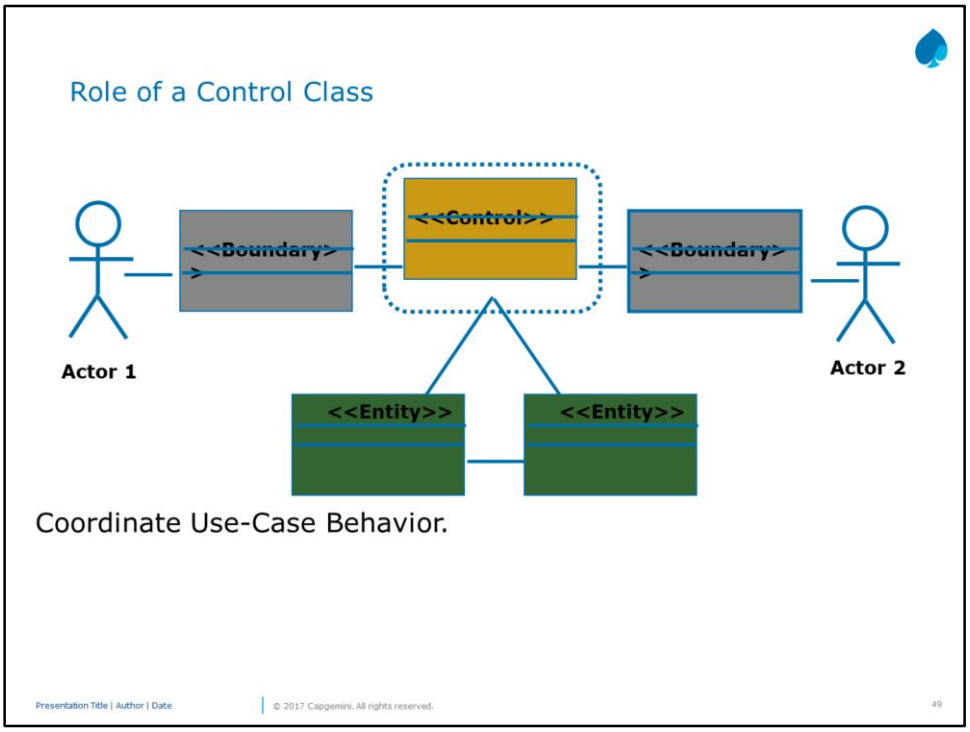


What is a Control Class?

- **Control classes** consist of objects that often control other objects. Therefore these classes have coordinating type of behavior. However, you need to remember that not all use-cases need control classes to be executed. Use-Cases that are associated with general manipulation of information, are executed by **entity** and **boundary classes** only.
- Control Classes help to:
  - Define responsibilities of complex use-cases.
  - Effectively insulate boundary classes from the entity classes. (As a result you are able to design a system more capable of adjusting to changes.)
  - Insulate use-case specific behavior from entity objects. (The resultant use-cases become re-usable.)
- Control classes provide behavior that:
  - Is surroundings-independent (does not change when the surroundings change).
  - Defines control logic (order between events) and transactions within a use-case.
  - Changes little if the internal structure or behavior of the entity classes changes.
  - Uses or sets the contents of several entity classes, and therefore needs to coordinate the behavior of these entity classes.
  - Is not performed in the same way every time it is activated (flow of events features several states).

Instructor Notes:

Control Class effectively decouples boundary and entity classes. Note that all communication between boundary and entity necessarily happens through the control class.

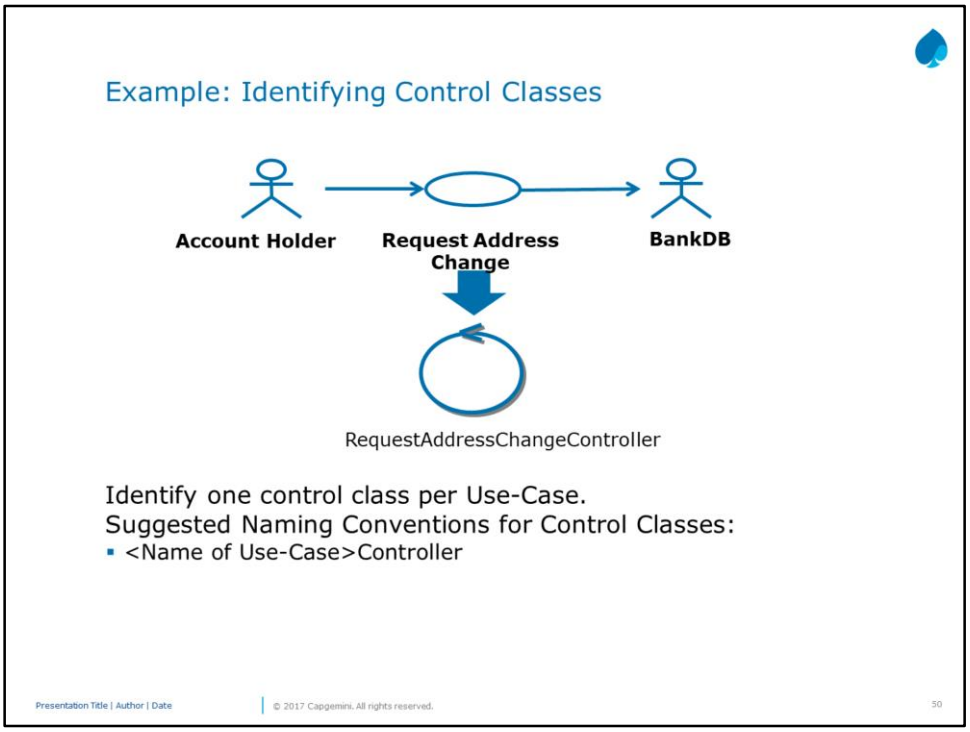


Role of a Control Class:

- **Control classes** encapsulate use-case-specific behavior. It effectively decouples **boundary** from **entity classes**.
- The behavior of a **control object** is closely related to the realization of a specific use-case. In many scenarios, you might even say that the control objects “run” the Use-Case Realizations.
- However, some control objects can participate in more than one Use-Case Realization if the use-case tasks are strongly related. Furthermore, several control objects of different control classes can participate in one use-case.
- Not all use-cases require a **control object**. For example, if the flow of events in a use-case is related to one **entity object**, then a **boundary object** may realize the use-case in cooperation with the **entity object**.
- You can start by identifying one control class per Use-Case Realization, and then refine this as more Use-Case Realizations are identified, and commonality is discovered.
- Control classes can contribute to understanding the system, because they represent the dynamics of the system, handling the main tasks, and control flows.
- When the system performs the use-case, a **control object** is created. Control objects usually die when their corresponding use-case has been performed.

Instructor Notes:

It is possible that a use-case may not need a controller or may need multiple controllers; or same controller can be shared by multiple use-cases. These refinements would be done as we move into designs.

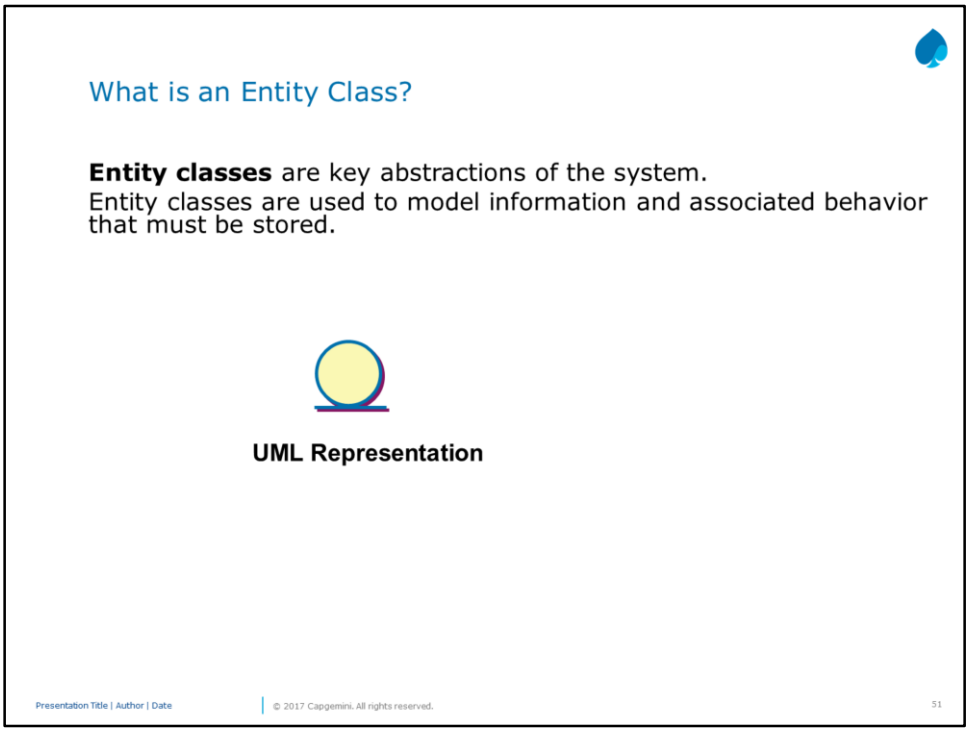


Example: Identifying Control Classes:

- The above slide shows an example of identifying control classes.
- Always identify one control class for each use-case.
- Note the naming convention:
  - <Name of Use-Case>Controller
- Once again, it is important to remember that the control classes are placeholders for the co-ordinating behavior and get refined as we go into designs.

Instructor Notes:

No guideline for number of entities! But we have the Underlining of the Noun technique.

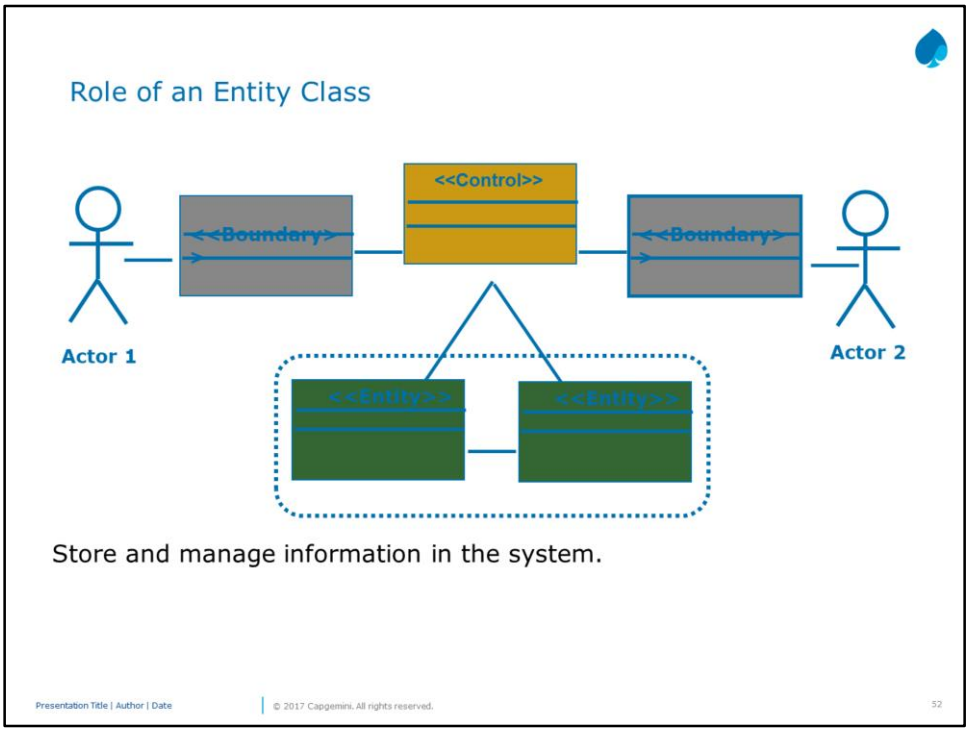


What is an Entity Class?

- **Entity object** (that is, an instance of an entity class) defines main features of the system-to-be. The **entity classes** show the logical data structure, and are therefore very helpful in understanding the system. A designer needs the data structure to know what the system will be offering to the user.
- **Entity objects** represent key concepts of the system being developed.
  - Typical entity classes of a banking system are Account and Customer.
  - In a network-handling system, examples of entity classes would be Node and Link.
- Entity class responsibilities can be sourced from the following business models and use-case descriptions.

Instructor Notes:

Note that entities talk to other entities. Any interaction to the actors have to be routed via the controller and corresponding boundary class.

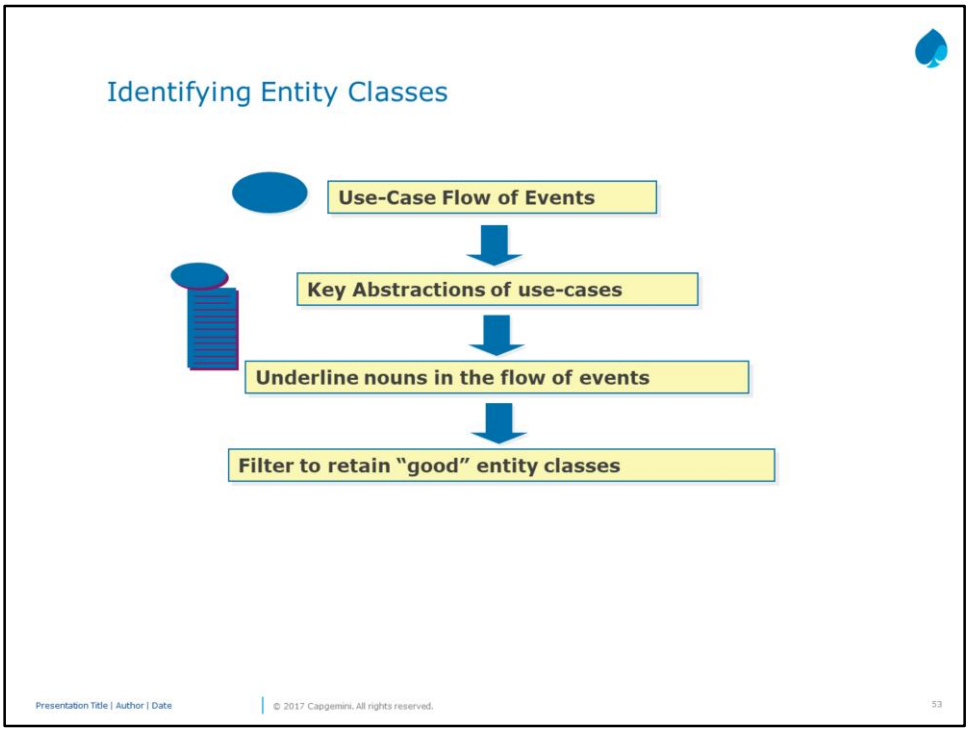


Role of an Entity Class:

- **Entity classes** are stores of information in a system, and represent all key concepts that the system manages.
- **Entity objects** (that is, instances of entity classes) are used to hold and update information about some phenomenon, such as an event, a person, or a real-life object. They are usually persistent, having attributes and relationships needed for a long period, sometimes for the lifetime of the system.
- An **entity object** is usually not specific to one **Use-Case Realization** and sometimes it is not even specific to the **system** itself. The values of its attributes and relationships are often given by an actor.
- An **entity object** may also be needed to help perform internal system tasks. Entity objects can have behavior as complicated as that of other object stereotypes. However, unlike other objects, this behavior is strongly related to the phenomenon that the entity object represents.
- **Entity objects** are independent of the environment (the actors).
- The main responsibilities of entity classes are to store and manage information in the system.
- **Entity classes** provide another view point from which to understand the system because they show the logical data structure, which can help you to understand what the system is supposed to offer its users.

Instructor Notes:

When in doubt on whether or not to "retain" a noun as entity class, a good idea would be to retain it. It would automatically get combined/eliminated as we get into design.

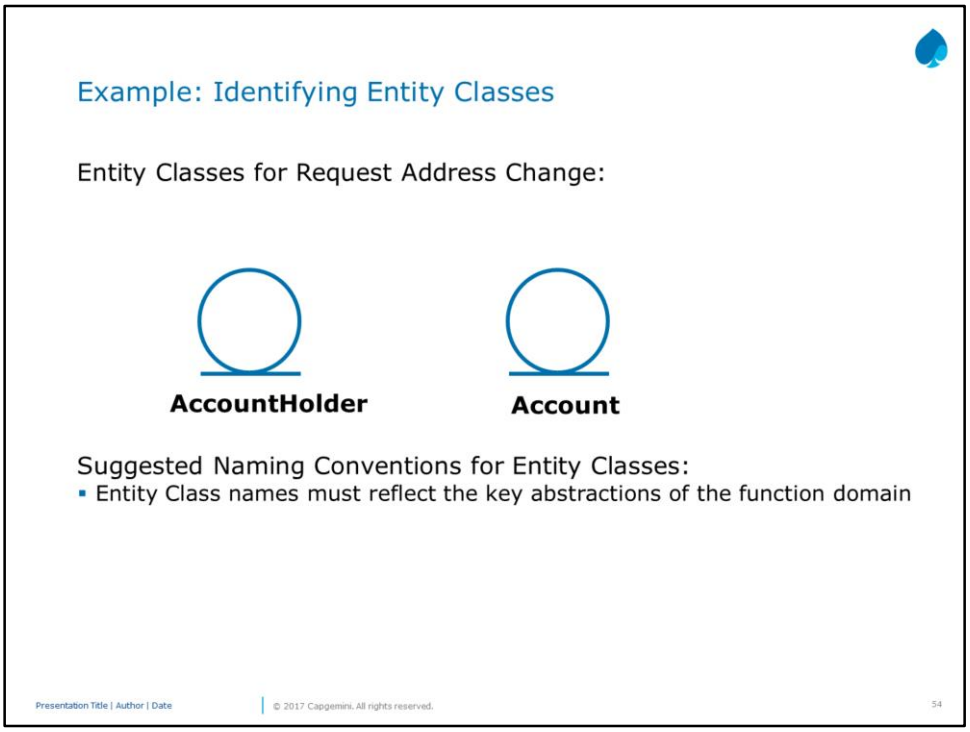


Identifying Entity Classes:

- We can use "Underlining the Noun Technique" for identifying **entity classes**.
- Entity classes can be identified from use-case flow of events.
- You can begin by making a list of **noun phrases** in the use-case flow of events. This list is the initial candidate list of analysis classes.
- The next step is to eliminate classes that do not fit the requirements. The following points need to be kept in mind for filtering nouns:
  - Retain only those abstractions that are clearly defined.
  - Remove actors that are out of scope for system.
  - Remove implementation constructs.
  - Remove attributes.
  - Remove operations.
- What you now have is a "good" set of entity classes.

Instructor Notes:

Entities too would get combined/broken as we move into design.



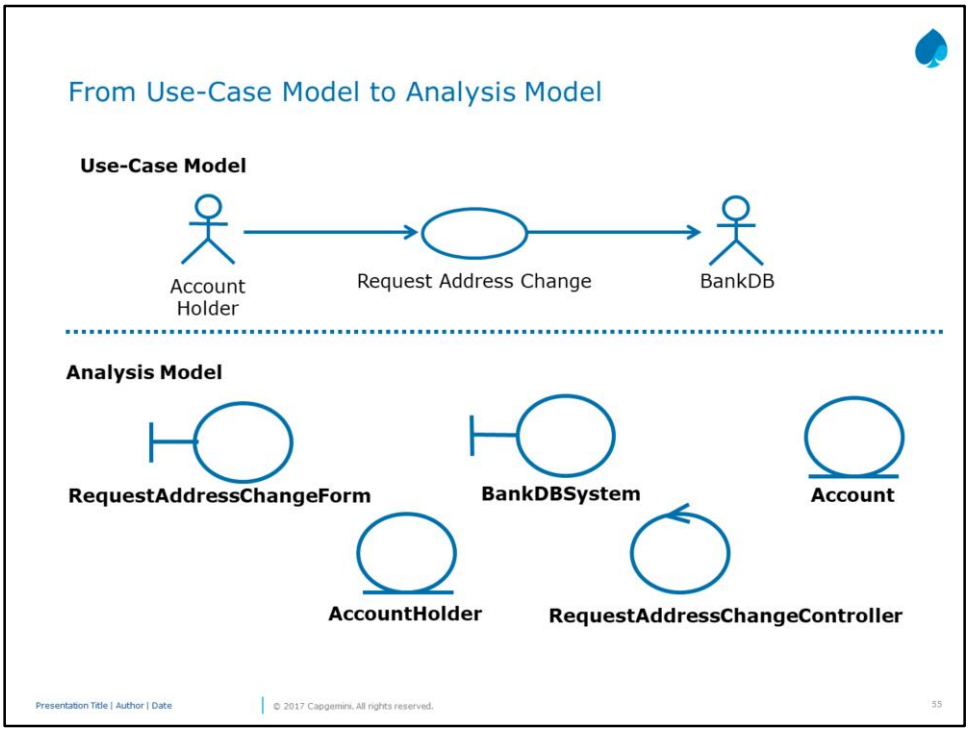
Example: Identifying Entity Classes:

- The above slide shows some entity classes identified for our example.
- There is no specific naming convention for the entity classes. However, just ensure that the name appropriately conveys what the class is all about.

Instructor Notes:

Guidelines make it straightforward to identify the set of analysis classes.

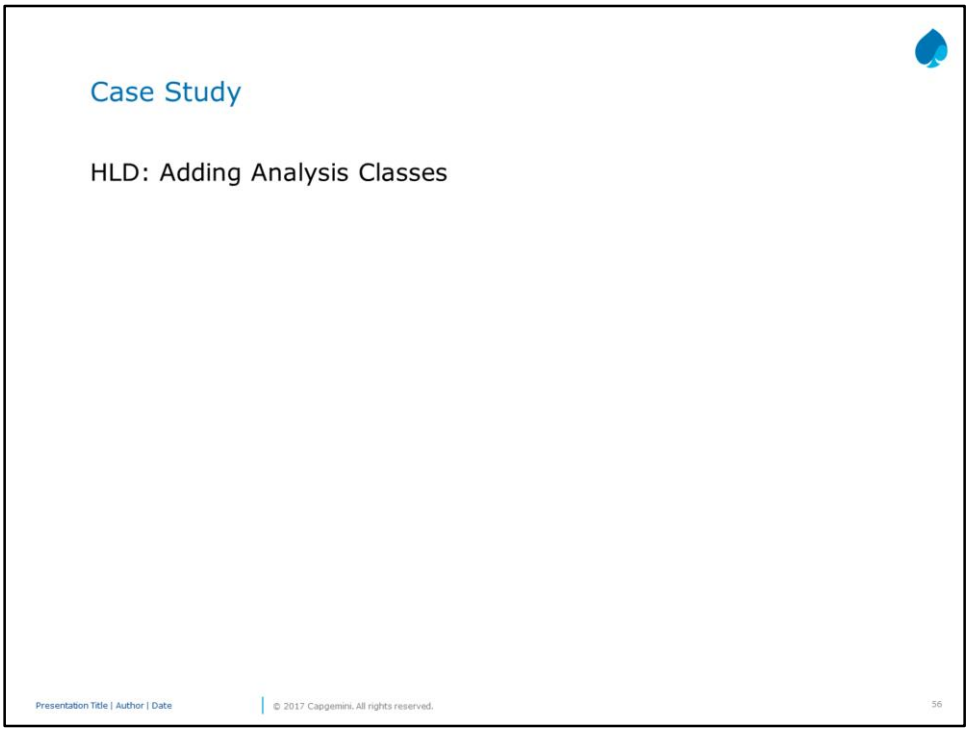
Encourage participants to consider other use-cases and identify analysis classes.



From Use-Case Model to Analysis Model:
The above slide gives a summary of the analysis classes that have been identified for our example.

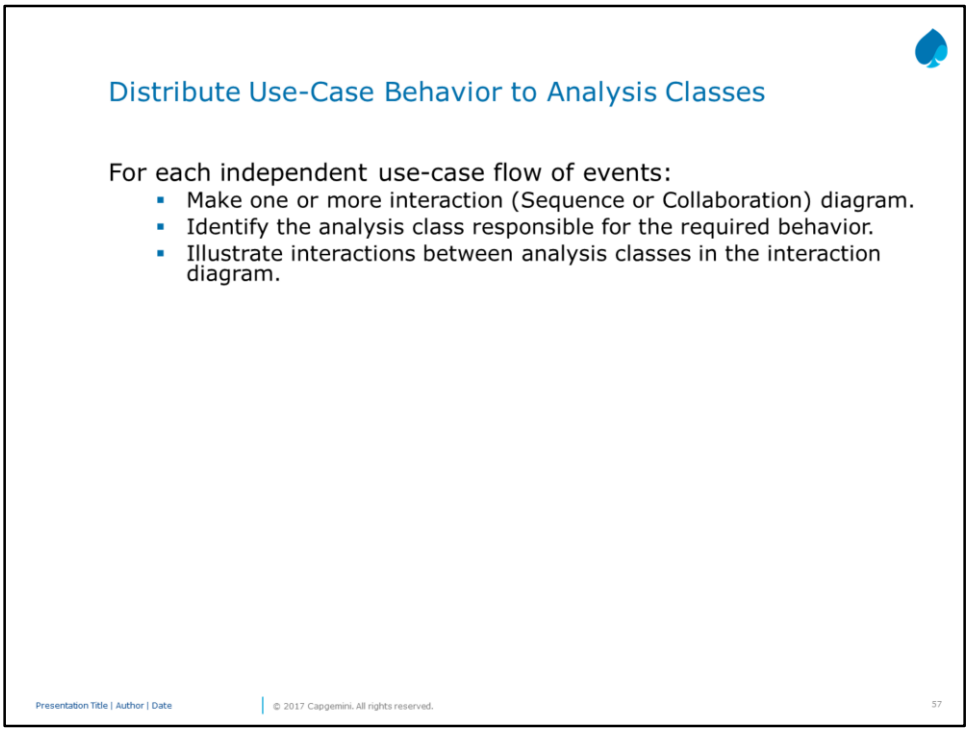
Instructor Notes:

Add the analysis classes for a use case of the case study



Instructor Notes:

Once Analysis Classes have been identified, next step is to determine their responsibilities. This is done with the help of sequence diagram, using the use-case flow of events as an input.

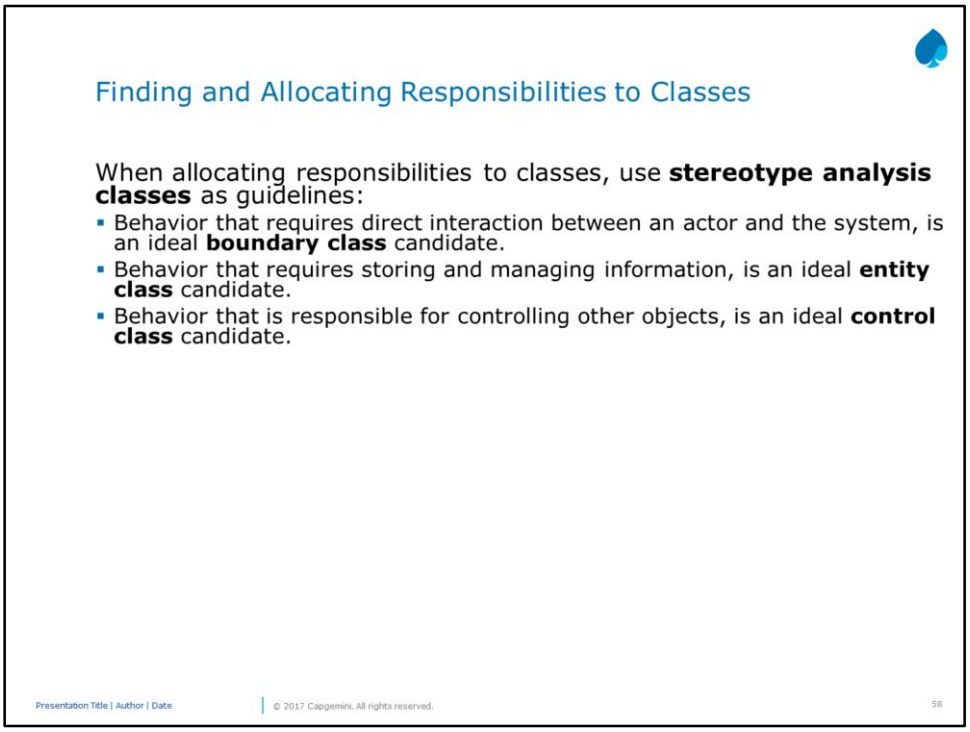


<u>Distribute Use-Case Behavior to Analysis Classes</u>:

- The Use-Case flow of events define the activities that are carried out and the order in which they are performed.
- For each independent flow of events, you need to:
    1. Create one or more Interaction Diagrams. First, model for the main flow of events, which is central to the activities within the use-case. Second, draw at least one diagram for every alternate / exceptional flow of events. Then draw a separate diagram for sub-flows with complex timing and decision points.
    2. Step through the flow of events to identify the analysis classes responsible for a required behavior. Ensure that the analysis use-case realization provides for every behavior defined in the use-case.
    3. Show interactions between **analysis classes**, in the interaction diagrams. The interaction diagram should also show interaction of the **system** with each **actor**. Always show interactions starting from an **actor** because it is an actor that invokes a use-case.
    4. Include classes that represent the control classes of used-cases.

Instructor Notes:

Emphasize the diagrams that we saw earlier which illustrates how the classes need to communicate with each other.

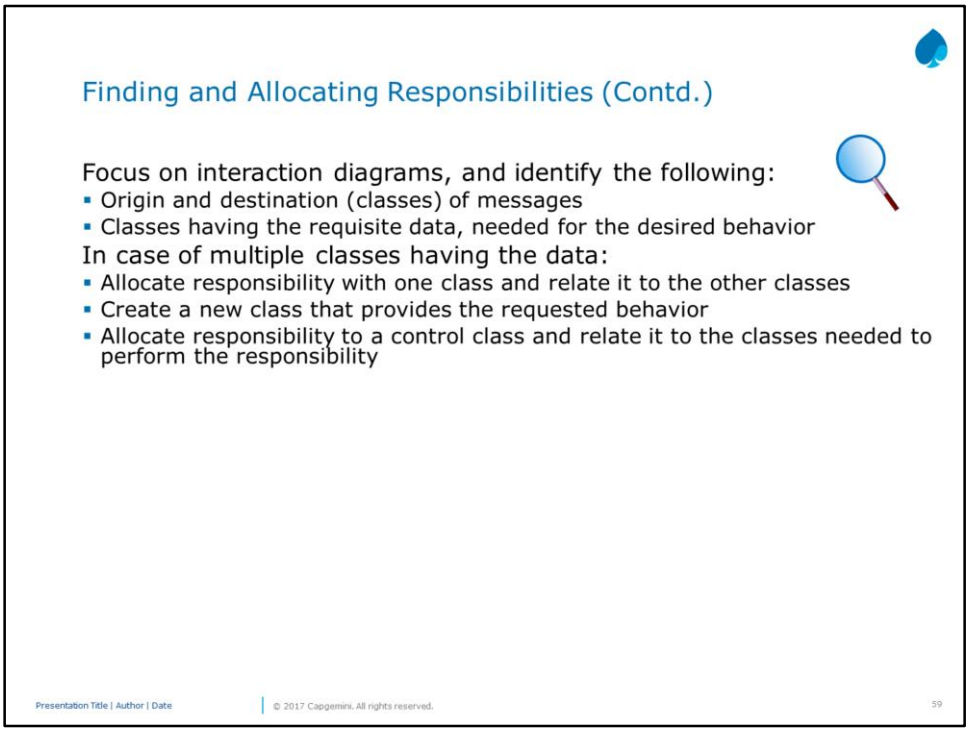


Finding and Allocating Responsibilities to Classes:

- Allocating responsibilities is an important activity. Hence utmost care needs to taken while doing it.
- You should use stereotypes as a guideline for allocating responsibilities.

Instructor Notes:

These are guidelines on how the interaction diagram are to be drawn.

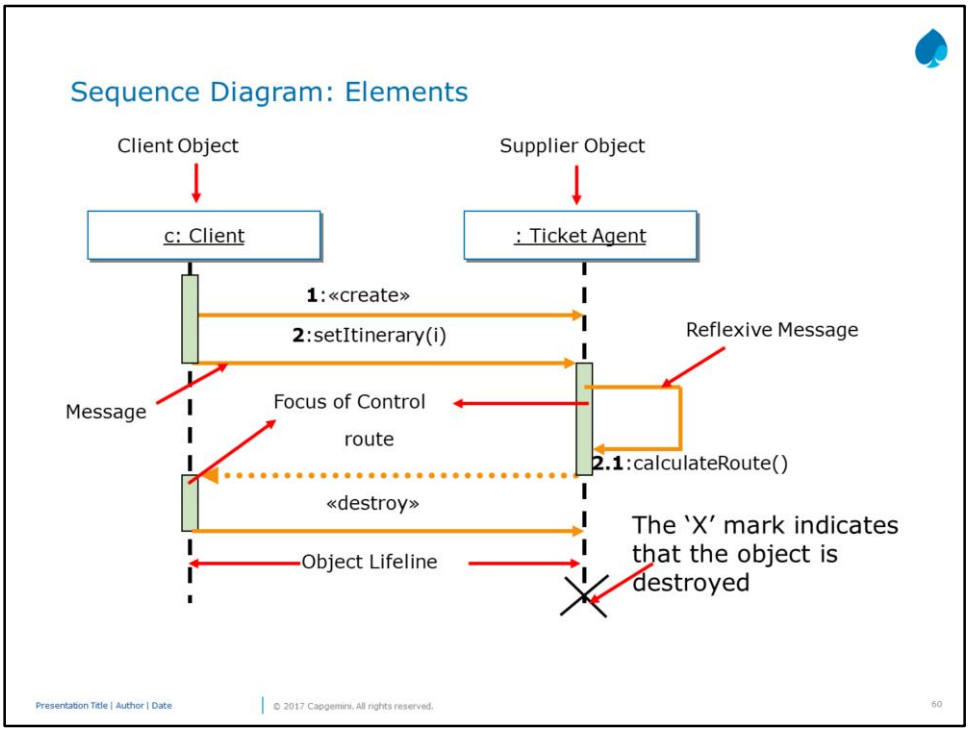


Finding and Allocating Responsibilities (contd.):

- **Responsibilities** are derived from messages in interaction diagrams. For each message, examine the class of the object to which the message is sent. If the responsibility does not yet exist, create a new responsibility that provides the requested behavior.
- You need to ensure that the **responsibility** for a behavior should always lie with the class having the requisite information. It implies that data and responsibility go hand-in-hand.
- In situations where the required data is contained in more than one class, you need to consider the situation thoroughly, before allocating responsibility. In such a situation, you may have to create a **class** or **relationship**, on which the responsibility can be reposed. The newly created element should have access to the data. New classes should be created only after confirming that no object exists that can perform the behavior.

Instructor Notes:

This diagram is just to understand the notations for a sequence diagram.



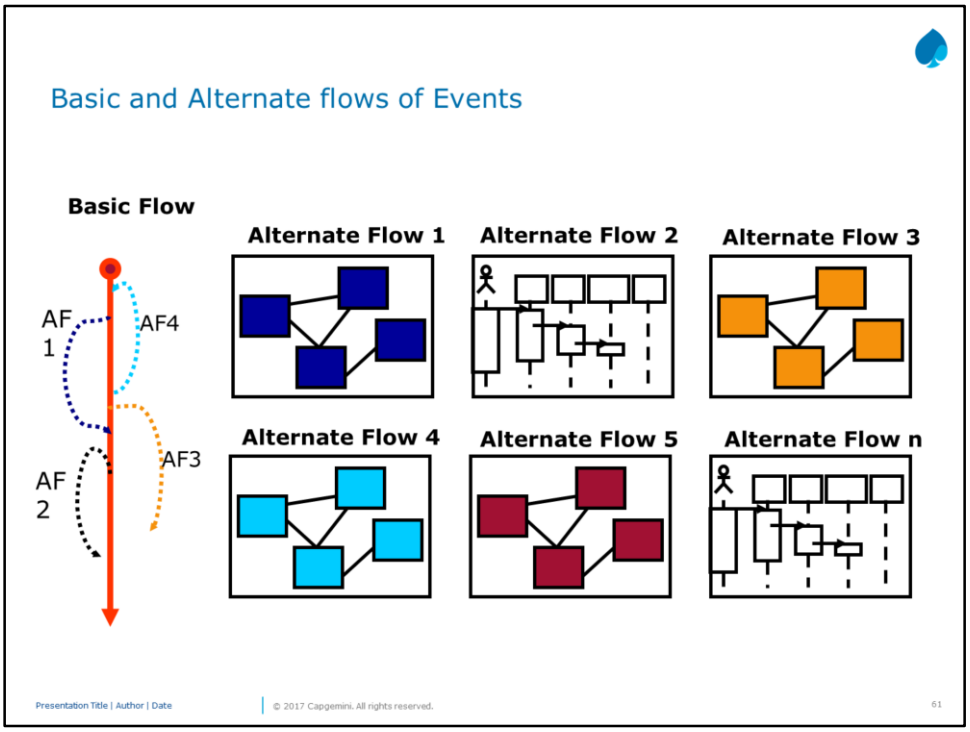
Sequence Diagram: Elements:

- Sequence diagrams, show objects and their interactions arranged in time sequence
- An **object** is shown as a vertical dashed line called the "lifeline". The lifeline represents the existence of the object at a particular time. An object symbol is drawn at the head of the lifeline, and shows the name of the object and its class separated by a colon and underlined.
- In the example in the above slide, there are two objects:
  1. Object Name is 'c' and Class is 'Client' (added)
  2. Unnamed object and class is 'Ticket Agent'
- A **message** is a communication between objects that conveys information with the expectation that activity will result. A message is shown as a horizontal solid arrow from the lifeline of one object to the lifeline of another object. For a **reflexive message**, the arrow starts and finishes on the same lifeline. The arrow is labeled with the name of the message and its parameters. In the example in the above slide, create, setItenarary(i), and destroy are messages going from object 'c' to object ':Ticket Agent'.
  - 'calculateRoute()' is a reflexive message.
- **Focus of control** represents the relative time that the flow of control is focused in an object, thereby representing the time an object is directing messages. Focus of control is shown as narrow rectangles on object lifelines. You can see the focus of control shifting from the "Client" to the "Ticket Agent" and then back to the "Client".
- **Hierarchical numbering** bases all messages on a dependent message. The dependent message is the message whose focus of control the other messages originate in. For example, message **2.1** depends on message **2.**
- **Scripts** textually describe the flow of events.

Instructor Notes:

For each use-case, at least 1 sequence diagram is to be modeled; one for the basic flow and more if needed for the alternate flows.

Do not un-necessarily add too many sequence diagrams, it will only complicate the understanding of the model. Wherever possible, notes can be used for alternate flows. Sequence diagrams can be modeled only for complex scenarios.



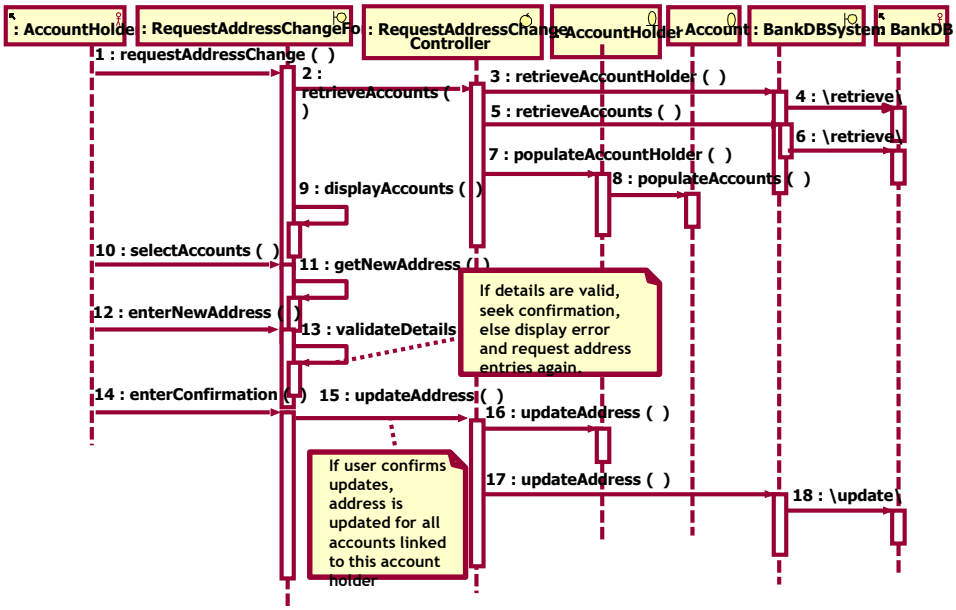
Basic and Alternate flows of Events:

- Apart from the **basic flow**, draw one or more interaction diagrams for the **alternate flows**. It is not required to have an interaction diagram modeled for every alternate flow in the system. You have to just pick up the complex ones and model them. The rest can be handled by using notes.
- So you can have additional sequence diagrams modeled for exceptional flows or optional flows.
- Examples of exceptional flows include the following:
  - In case of error: Situations when the system encounters an error.
  - In case the user does not respond within the stipulated period of time.
  - In case of in-correct input by the user.
- Examples of optional flows include the following:
  - In case the user has to select from a list of options.
  - In cases where the next step depends on the value entered by the user.
  - In cases where the next step depends on the type of data entered by the user.

Instructor Notes:

Discuss this sequence diagram example emphasizing on the notes mentioned for the diagram.

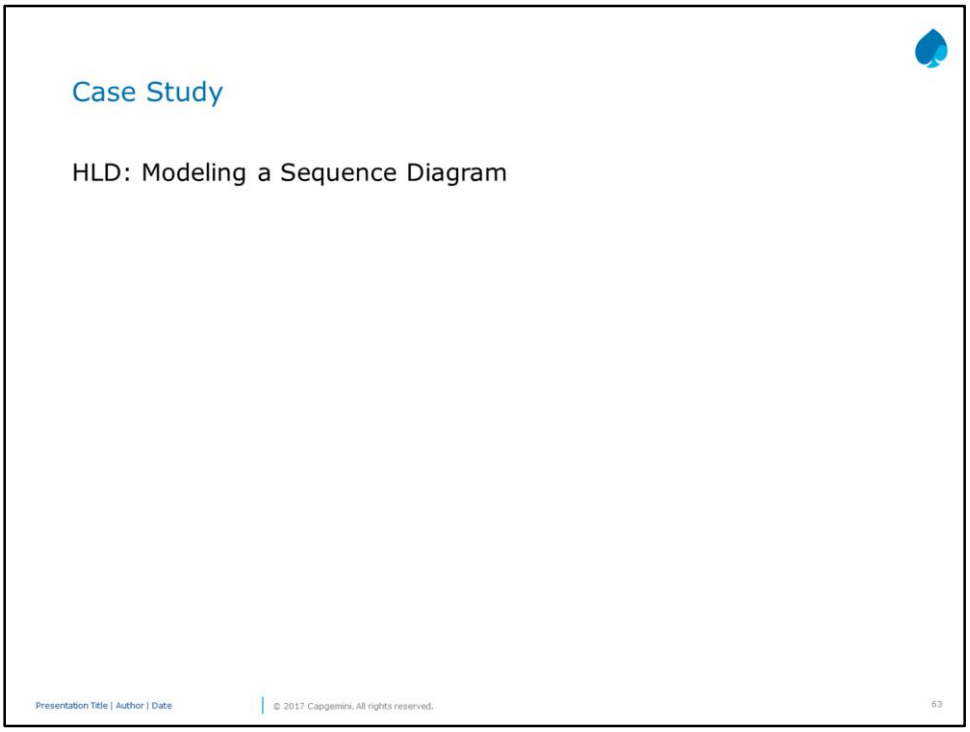
Example: Distributing Use-Case Behavior:



- The above slide shows an example of distributing use-case behavior.
- Note the following:
    1. Actor initiates the flow.
    2. All communications with actors are through boundary classes.
    3. Controller does the co-ordinating behavior.
    4. Entities process information as needed.

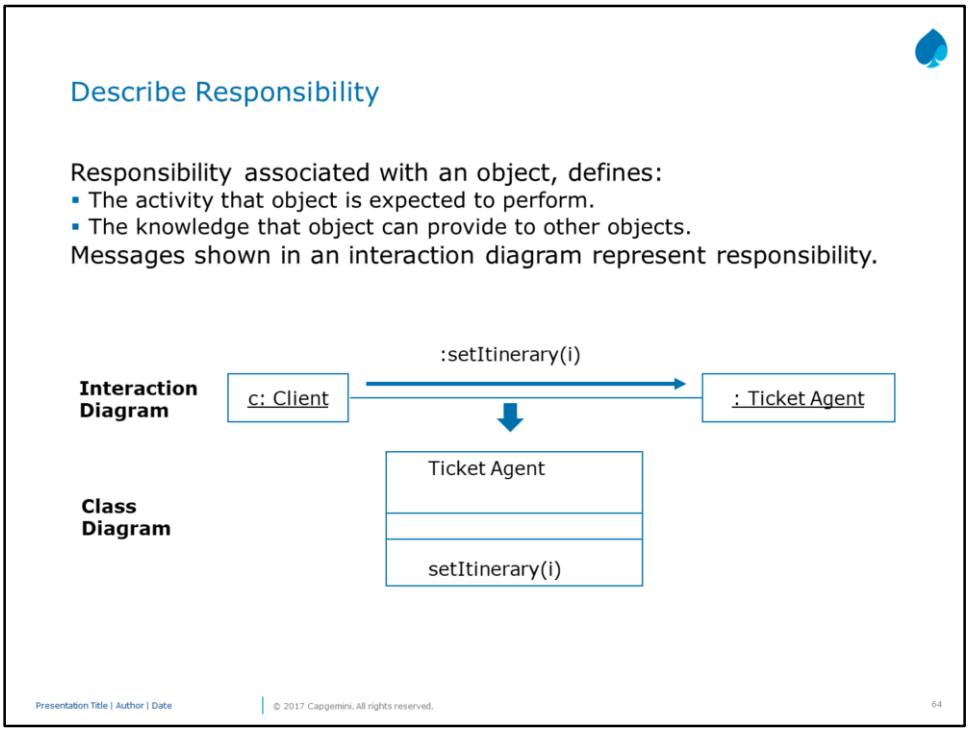
Instructor Notes:

Model a sequence diagram for a use case of the case study



Instructor Notes:

If the object receives a specific message, it means that the corresponding class must have that responsibility defined

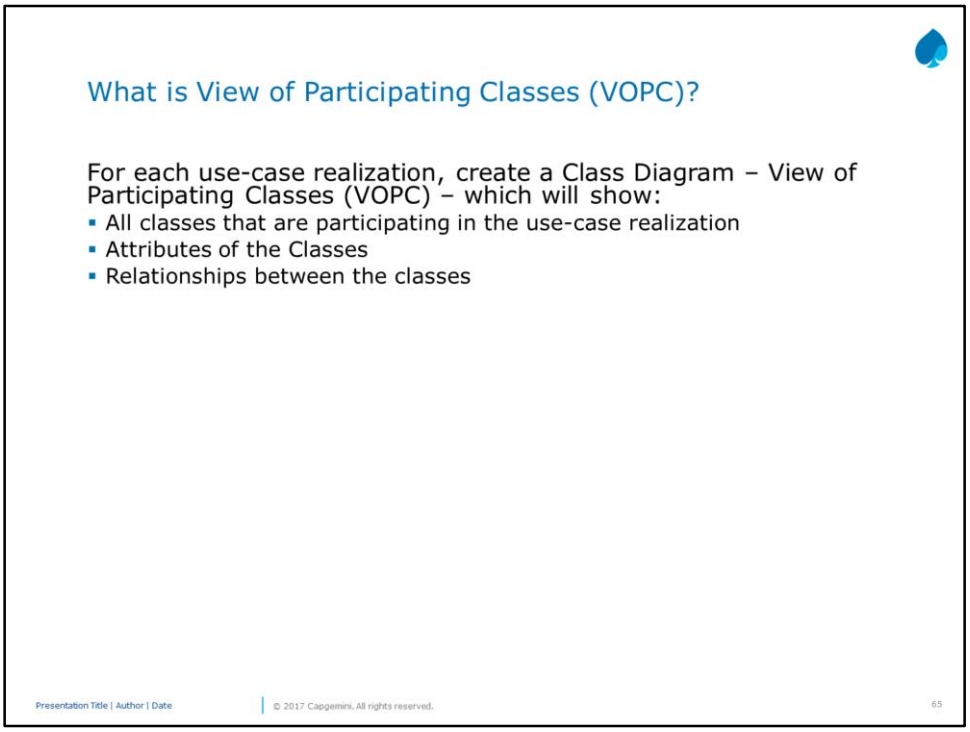


Describe Responsibility:

- A **responsibility** is a statement of something an object can be asked to provide. Responsibilities evolve into one (or more) operations on classes in design. They can be characterized as:
  - The actions that the object can perform.
  - The knowledge that the object maintains and provides to other objects.
- Responsibilities are derived from messages in Interaction diagrams. For each message, examine the class of the object to which the message is sent. If the responsibility does not yet exist, create a new responsibility that provides the requested behavior.

Instructor Notes:

Class Diagram can be derived from the Sequence Diagram by looking at the participating objects and their interactions.



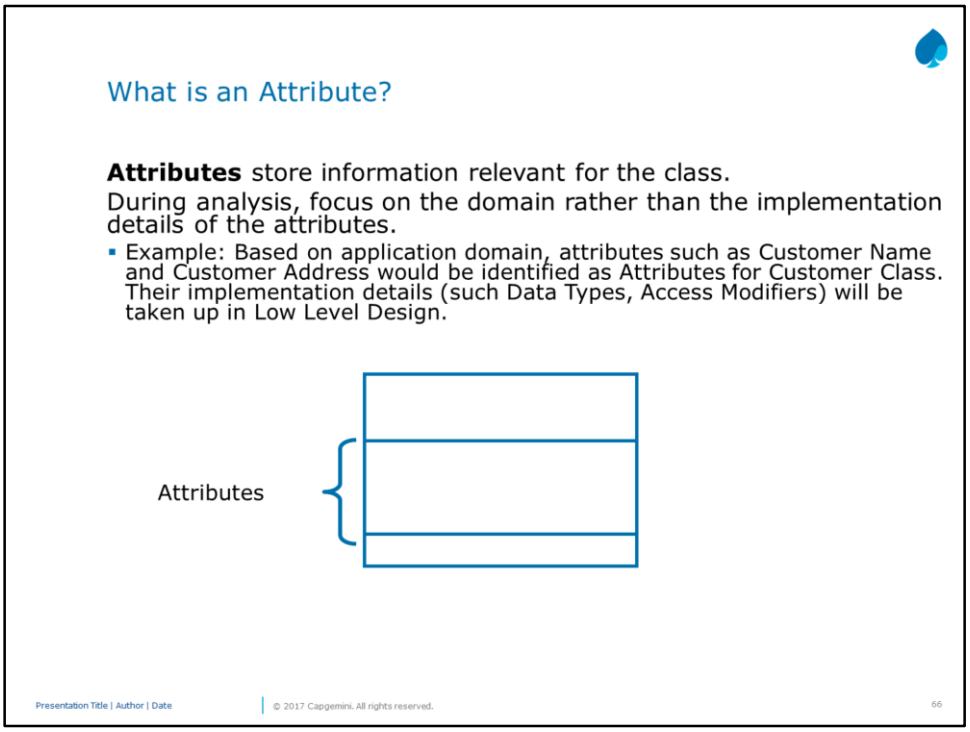
What is View of Participating Classes (VOPC)?

View of Participating Classes (VOPC):

- Is a class diagram showing all classes, which will:
  - Participate in a collaboration, or
  - Implement a use-case

Instructor Notes:

We do not get into details such as datatypes and default values of attributes at this point.



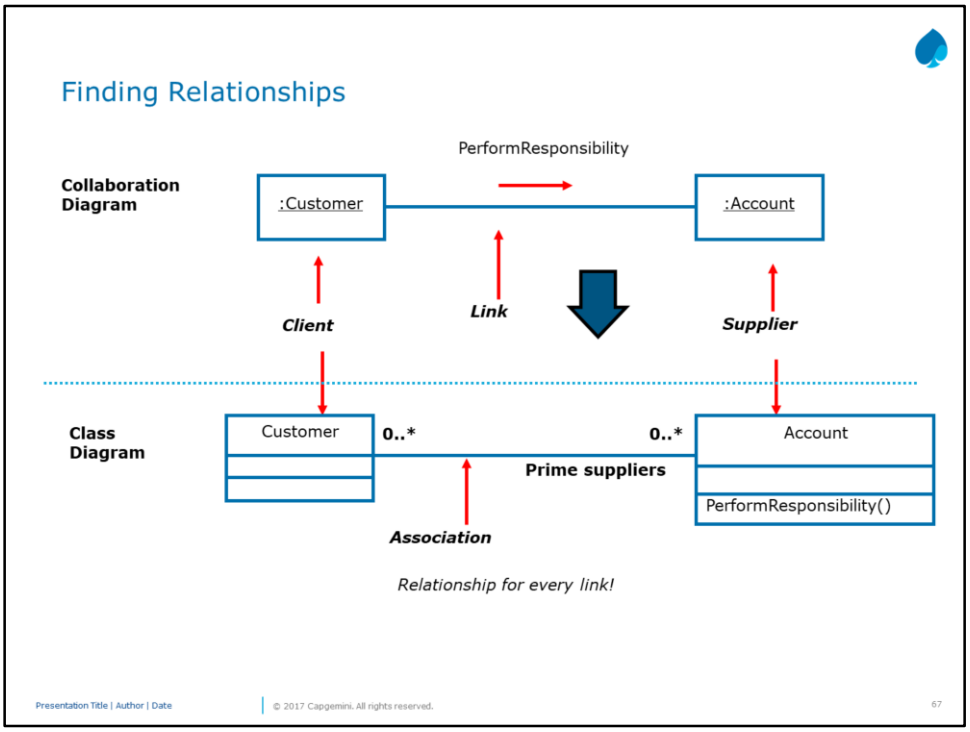
What is an Attribute?

- An **attribute** nearly always denotes **object** properties. An attribute can also denote properties of a **class**, in which case it is a **class attribute**.
- 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 can look at the nouns of the use-case descriptions – that were filtered out during the stage of identifying entity classes – to check if they become attributes for a class. Other than that, attributes are identified depending on the responsibilities of a class.

Instructor Notes:

Ensure that if there is an interaction between two objects, it gets captured as a relationship between the corresponding classes.

For now, all relationships are captured as Associations; they are later refined in the Design Model.



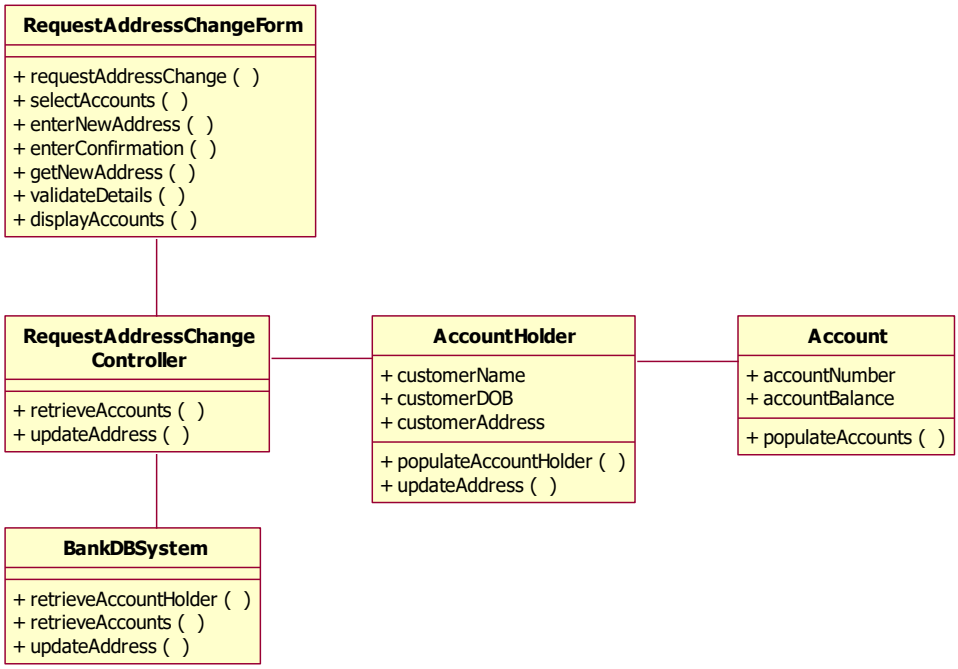
<u>Finding Relationships</u>:

- Carefully look at the interaction diagrams, and study the **messages** or the **links**.
  - **Links** represent the communication lines between two objects, when performing a use-case. These links may be **associations** or **aggregations**.
- **Reflexive links** need not be related to reflexive relationships. An object can send messages to itself. Two more objects of the same class may need to exchange messages.
- The **navigability** of the relationship should support the required message direction.
  - In the above example, if navigability was not defined from the Customer to the Account, then the **PerformResponsibility** message could not be sent from the Customer to the Account.
- When defining associations, take care to define only those that exist. Give appropriate names and multiplicites. Also define navigability.

Instructor Notes:

There would be one VOPC for each use-case.

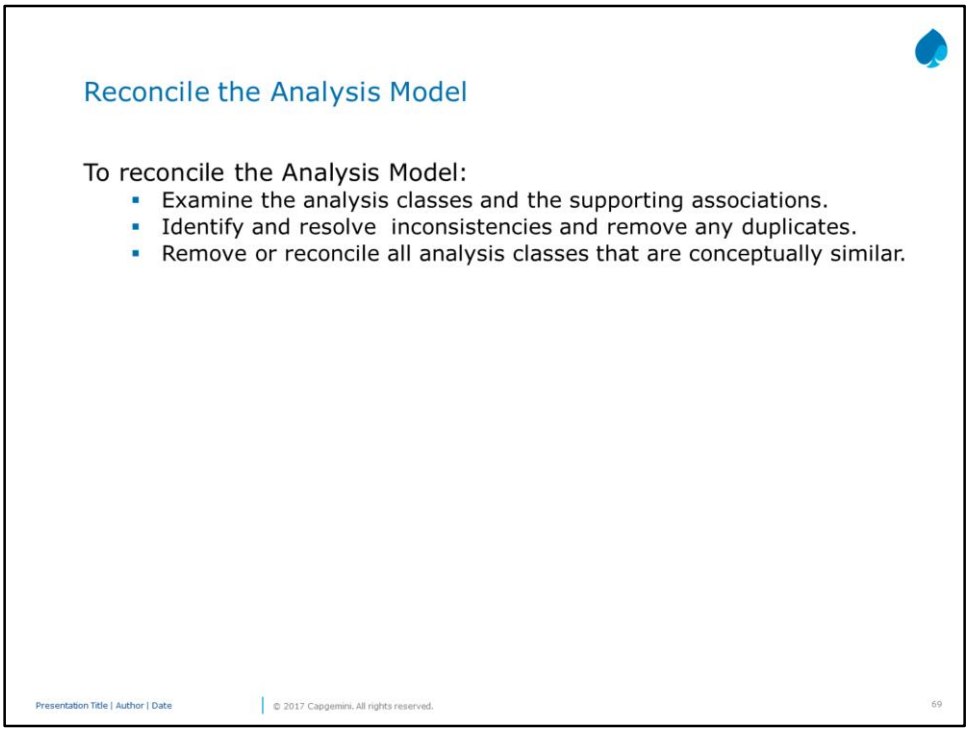
Example: View of Participating Classes:



- The above example shows a Class Diagram for Request Address Change.
- Notice that all incoming operations to an object on the sequence diagram become an operation in the corresponding class.
- This is the conceptual model. Hence details such as data types, parameters, visibility, class relationships etc. are not refined.

Instructor Notes:

Since use-case realization is done on a use-case by use-case basis, it is important to consolidate and reconcile the Analysis Model before moving to design.

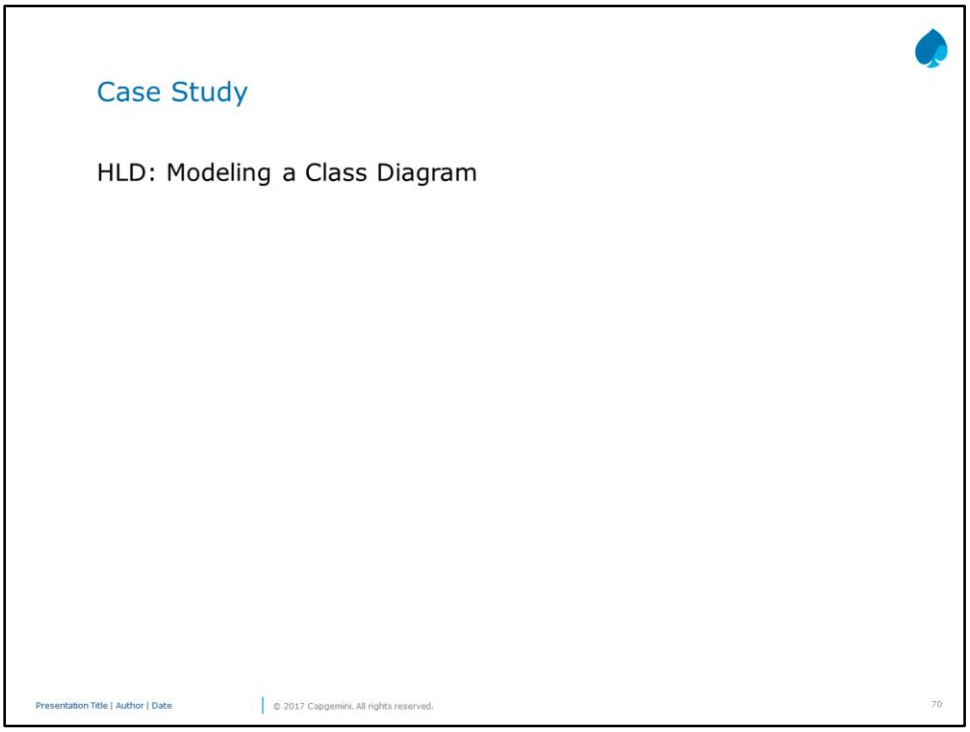


Reconcile the Analysis Model:

- The **analysis use-case realizations** are developed as a result of analyzing a particular use-case. Now, the individual **analysis use-case realizations** need to be reconciled.
  - Examine the analysis classes and the supporting associations defined for each of the analysis use-case realizations.
  - Identify and resolve inconsistencies and remove any duplicates.
- For example, two different analysis use-case realizations might include an analysis class that is conceptually the same. However, the analysis classes may be identified by different designers. Hence a different name is used.

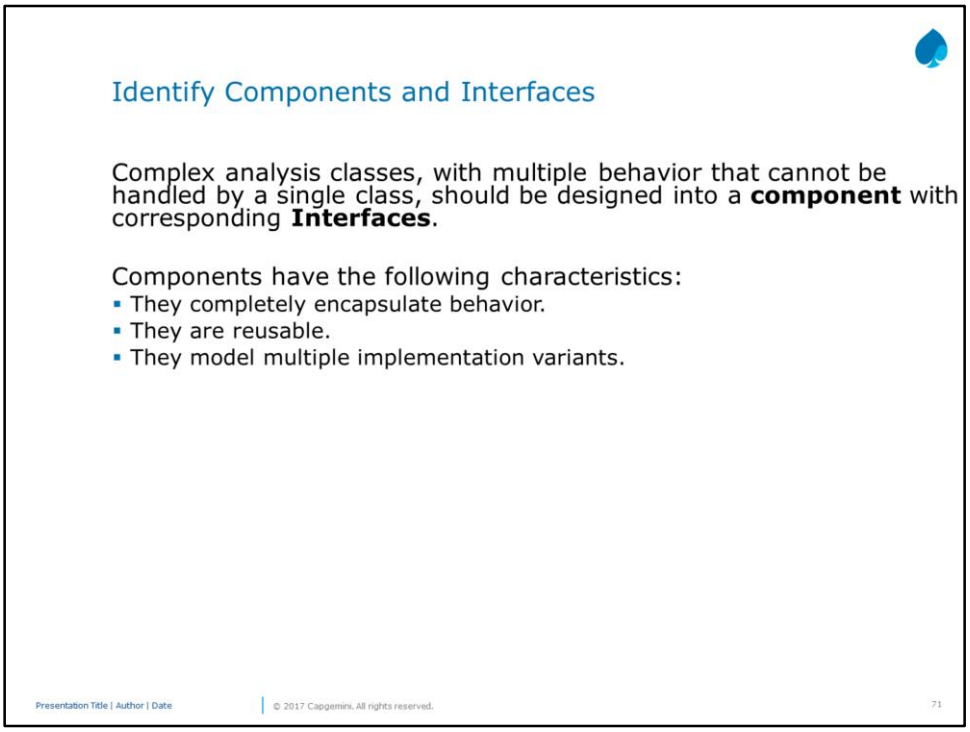
Instructor Notes:

Model a class diagram for the use case where sequence diagram was modeled earlier



Instructor Notes:

Once the Analysis Model in terms of Analysis Classes and Use-Case Realization is done, we now check whether any of the analysis classes should actually be components, which implement one or more interfaces.



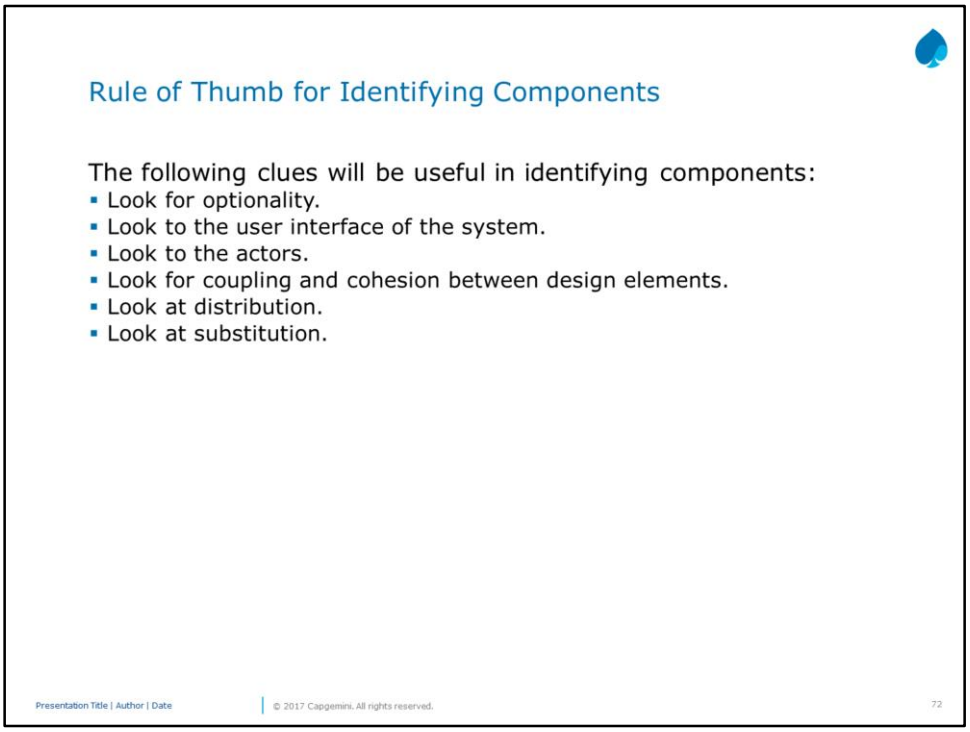
Task1: Create Functional Design:

Step7 & Step 8: Create System Interface Design, Identify Components:

- Now that analysis classes and their responsibilities are known, we can check if any of the **analysis class** has a "complex behavior". If yes, we can identify the class as a potential candidate for a **component**.
- A component in design stage is called a **subsystem**.
- Each **subsystem** is linked up with one or more **interfaces**. These interfaces provide **encapsulation**. As a result, the internal design of the subsystem remains hidden from other **model elements**.
- The **contents** and **collaborations** within a subsystem are completely isolated behind one or more **interfaces**.
  - The client of the subsystem is only dependent upon the interface.
  - When two or more subsystems realize the same interface, they can be substituted for one another. This is possible because the internal behaviors and elements of a subsystem can change freely as long as the interface remains the same.

Instructor Notes:

These guidelines will help in identifying Components.

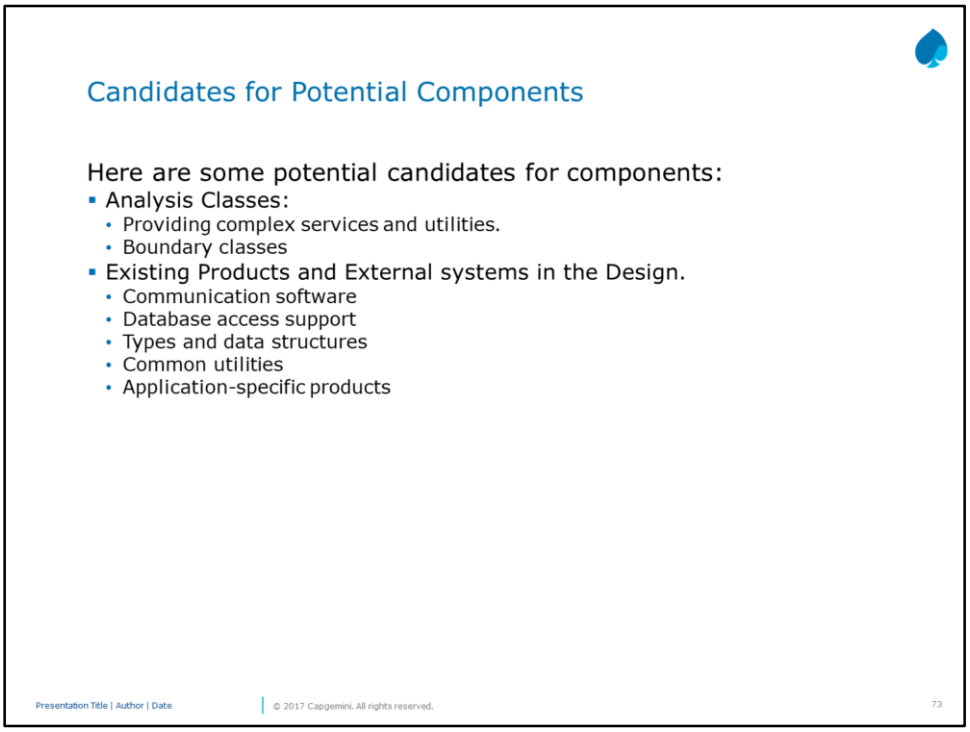


Rule of Thumb for Identifying Components:

- To begin with, complex analysis classes with behavior that cannot be the responsibility of a single design class, should be mapped into subsystems. Sometimes, a complex design class is also mapped into a subsystem, if it is likely to be implemented as a set of collaborating classes.
- Subsystems also identify those parts of a system that need to be developed independently by a separate team.
  - **Optionality**: If a particular collaboration (or sub-collaboration) represents optional behavior, enclose it in a subsystem.
  - **User interface**: If the user interface is relatively independent of the entity classes in the system (that is, the two can and will change independently), create subsystems which are horizontally integrated.
    If the user interface and the entity classes it displays are tightly coupled (that is, a change in one triggers a change in the other), then create subsystems which are vertically integrated.
  - **Actors**: Separate functionality is used by two different actors.
  - **Class coupling and cohesion**: Organize highly coupled classes into subsystems, separating along the lines of weak coupling.
  - **Substitution**: Represent different service levels for a particular capability (for example, high, medium, and low availability) as a separate subsystem, that realizes the same interfaces. If there are several levels of service specified for a particular capability (example: high, medium, and low availability), represent each service level as a separate subsystem, each of which will realize the same set of interfaces.
  - **Distribution**: In cases where subsystem behavior must be split across nodes, it is recommended that you decompose the subsystem into smaller subsystems (each representing a single component) with more restricted functionality.

Instructor Notes:

Note that existing products would get represented in designs as components / interfaces.

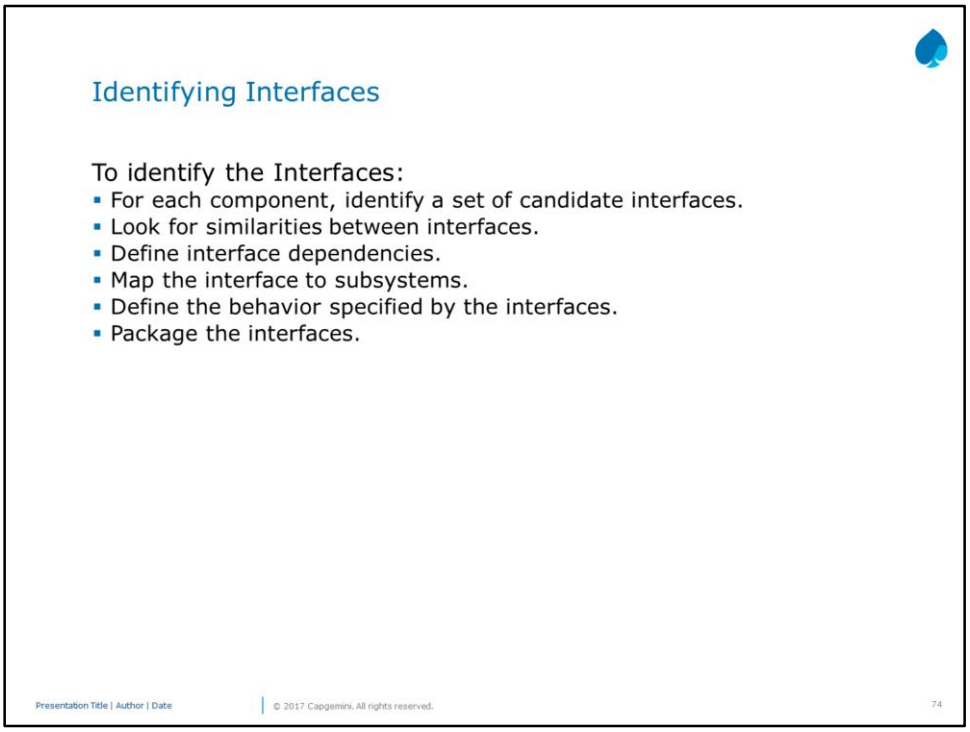


Candidates for Potential Components:

- Examples of **complex analysis classes** that should be modeled into **subsystems**:
  - Classes providing complex services and /or utilities. For example:
    - Credit or risk evaluation engines in financial applications
    - Rule-based evaluation engines in commercial applications
    - Security authorization services in most applications
  - Boundary classes, both for user interfaces and external system interfaces. If the interface(s) are simple and well-defined, a single class may be sufficient. However, these interfaces are often too complex to be represented using a single class. They often require complex collaborations of many classes. Moreover, these interfaces may be reusable across applications. As a result, a subsystem more appropriately models these interfaces in many cases.
- Examples of products, which the system uses, that you can represent by a subsystem include the following:
  - Communication software (middle-ware, COM/CORBA support)
  - Database access support (RDBMS mapping support)
  - Types and data structures (stacks, lists, queues)
  - Common utilities (math libraries)
  - Application-specific products (billing system, scheduler)

Instructor Notes:

Having identified the components, next step is to identify the interfaces for these components.



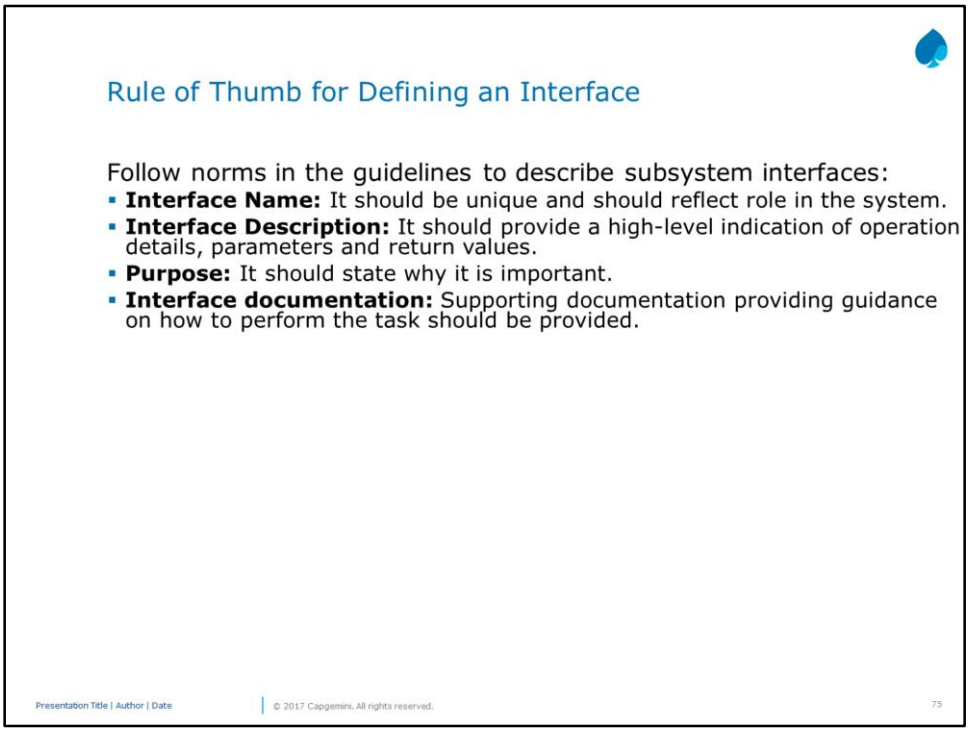
Identifying Interfaces:

After identifying the subsystems, you need to identify the interfaces. The sequential steps that need to be followed are as follows:

1. **Identify a set of candidate interfaces:** Look for the responsibilities that are activated when a subsystem is initiated. Then identify an operation for each responsibility. Identify the input and output parameters and return value, for each responsibility, which the subsystem will realize. This process is repeated till all responsibilities realized by the subsystem are defined.
2. **Look for similarities between interfaces:** Look for similar names, responsibilities, and operations among the identified set of candidate interfaces. Interfaces having common operations should be re-factored to define new interface. While re-factoring, look at the existing interfaces as well and re-use them wherever possible.
3. **Define interface dependencies:** The parameters and return value of each interface operation each have a particular type: they must realize a particular interface, or they must be instances of a simple data type. In cases where the parameters are objects that realize a particular interface, define dependency relationships between the interface and the interfaces on which it depends.
4. **Map the interfaces to subsystems:** Once the interfaces have been identified, create **realization** associations between the subsystem and the interfaces it realizes.
5. **Define the behavior specified by the interfaces:** Suppose the operations on the interface must be invoked in a particular order (for example, the database connection must be opened before it can be used). Then a state machine that illustrates the publicly visible (or inferred) states, which any design element realizes the interface must support, should be defined.
6. **Package the interfaces:** To allow the interfaces to be managed and controlled independently of the subsystems, group interfaces into one or more packages owned by the software architect.

Instructor Notes:

Interfaces will need to be completely identified, including the name, complete list of parameters and returns of all the operations defined on the interface.



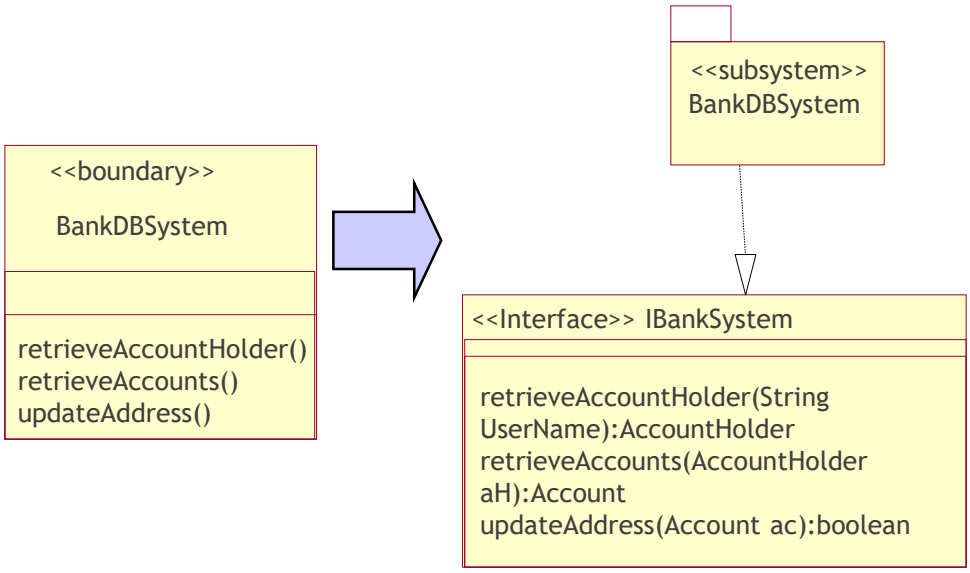
Rule of thumb for Defining an Interface:

- Follow the norms in the project specific guidelines artifact to describe the subsystem interfaces, such as:
  - **Name:** A unique name used to identify the interface
  - **Description:** A short description of the contents of the Interface, typically giving some high-level indication of complexity, scope, and intended audience
  - **Purpose:** An explanation of what this Interface represents and why it is important

Instructor Notes:

Participants generally take more time understanding this concept. So discuss this in detail.

Example: Components and their Interfaces:



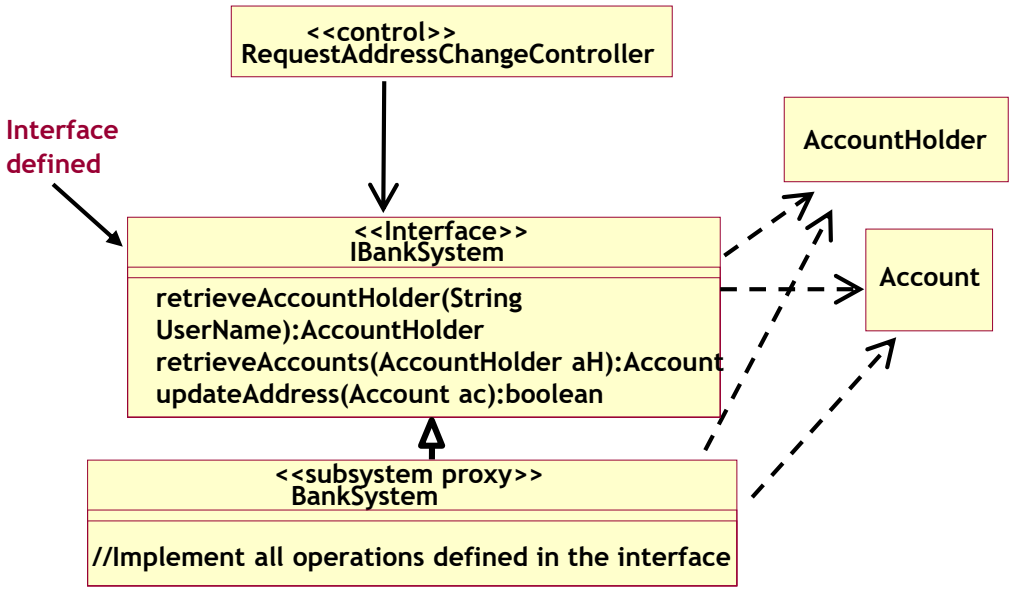
- The above slide shows an example from the Bank System case study. The boundary class BankDBSystem has to interface with the Bank Database, which is an external system. This behavior can be encapsulated in a component that implements interfaces to retrieve and update the database.
- The figure in the above slide shows introduction of the BankDBSystem Component along with interface IBankSystem. Note that a component in design is represented as a subsystem, as indicated in UML by the stereotype <<subsystem>> used on package.
- Also note that all operations of the boundary class have now become operations on the interface identified for the subsystem.

Instructor Notes:

Participants generally take more time understanding this concept. So discuss this in detail.

Concept of Subsystem proxy class also needs to be discussed in detail..the fact that it represents the classes of the subsystem.

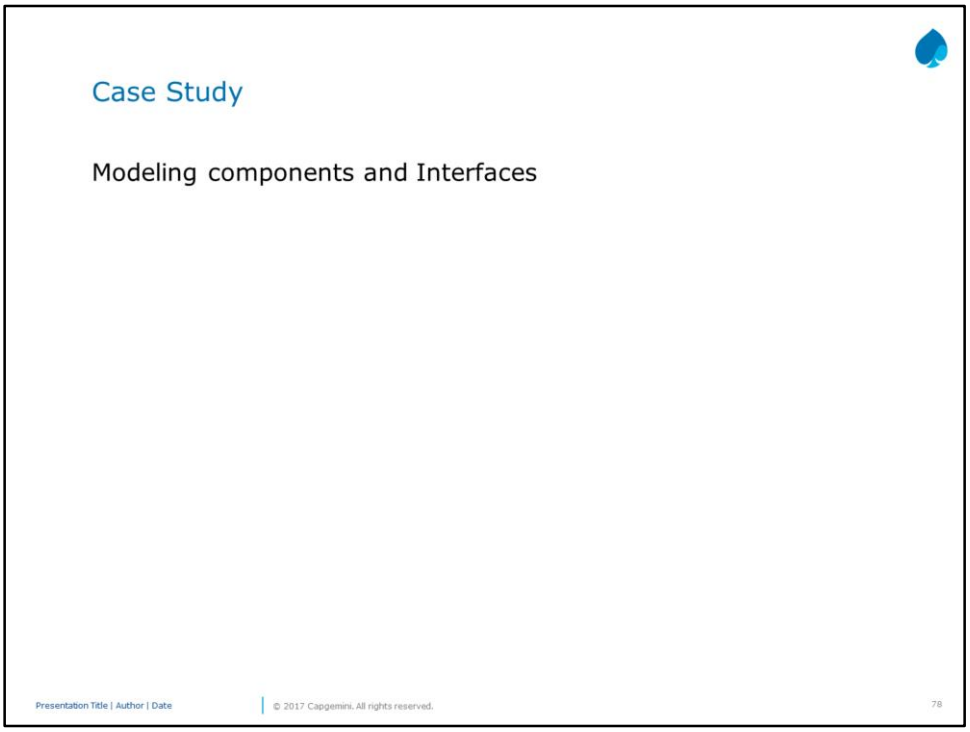
Example: Component Context Diagram:



- A subsystem context diagram indicates who accesses the component using the interface, and who provides the implementation for the interface.
    1. The RequestAddressChangeController class accesses the component using the interface IBankSystem.
    2. The interface operations are implemented by the component BankDBSystem. Assume that the component does not already exist and it needs to be designed. So we consider the component at the moment is a black box. A component within it will have a set of classes with an encapsulated behavior. We will design these set of classes a little later. To represent these set of classes, we will consider only one class at present, called as a Subsystem Proxy class BankSystem. So we just indicate the subsystem proxy class as the class that will implement the interface.
    3. Note the dependency relationship with the class AccountHolder and Account. As we have seen before, since objects of these classes are returned by the interface operations, this becomes parameterized visibility and is represented using dependency.
    4. Also note the **uni-directional association** shown from the controller to the interface. The scenario will always be a class accessing an interface and can never be an interface accessing a class operation. Hence we show a **uni-directional association**.

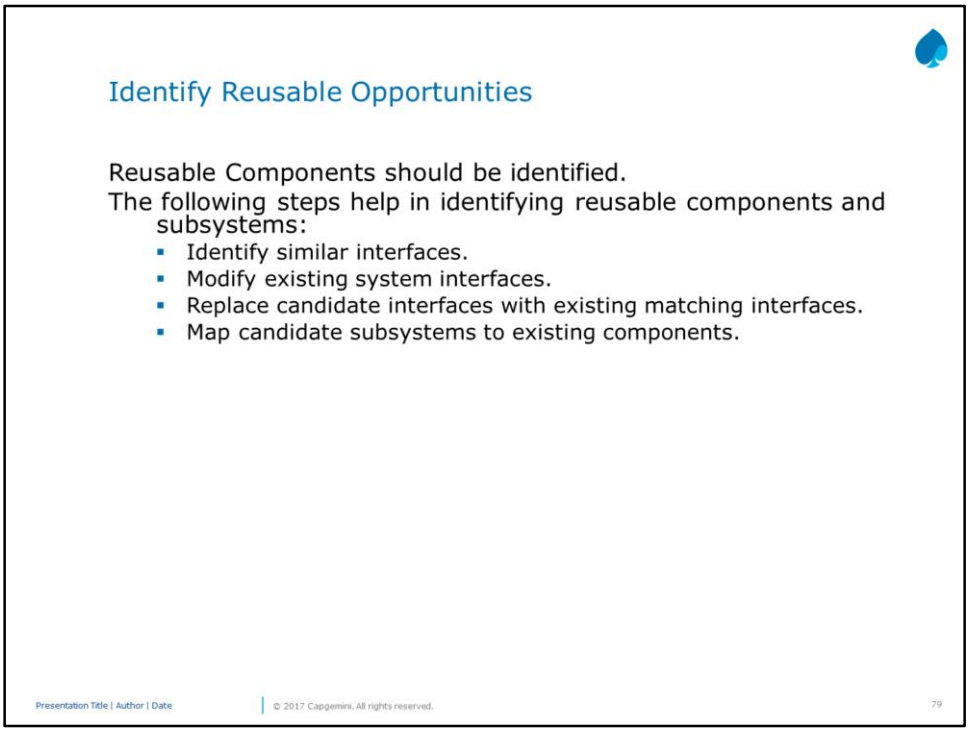
Instructor Notes:

Model a subsystem context diagram for a component



Instructor Notes:

Components are excellent units for reusability. Before jumping into the bandwagon of building a new component, a check needs to be made is something already exists and can be reused. Sometimes reuse as it is may not be possible, in which case we can check if reuse is possible with some tweaking.

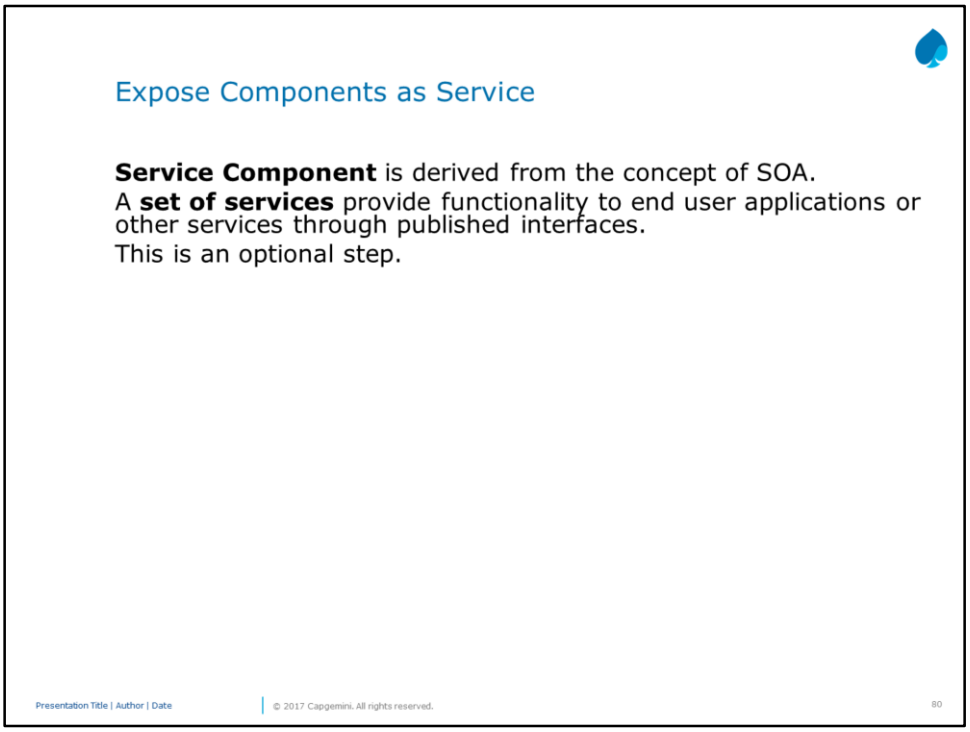


<u>Identify Reusable Opportunities</u>:

- For the identified interfaces and components, make a check whether something can be reused or the components will have to be designed from scratch.
  - **Look for existing subsystems and components which offer similar interfaces:** Examine subsystem and component interfaces for similarities. Look for similar behavior, returned value, and parameters. Two interfaces are said to similar even if they approximate matches.
  - **Modify the newly identified interfaces to improve the fit:** Incorporate minor changes to a candidate interface in order to improve the fit. These minor changes include modifying or adding parameters. At times, an interface may be split into several interfaces, such that a match with an existing component is obtained.
  - **Replace candidate interfaces with existing interfaces where exact matches occur:** After simplification and factoring, if there is an exact match to an existing interface, eliminate the candidate interface and simply use the existing interface.
  - **Map the candidate subsystem to existing components:** Factor the subsystems so that existing components are used wherever possible to satisfy the required behavior of the system.

Instructor Notes:

Here is the last step of Create Functional Design. It is an optional step, to be used when we would like to have a service based architecture. We are looking at a brief overview of this concept.



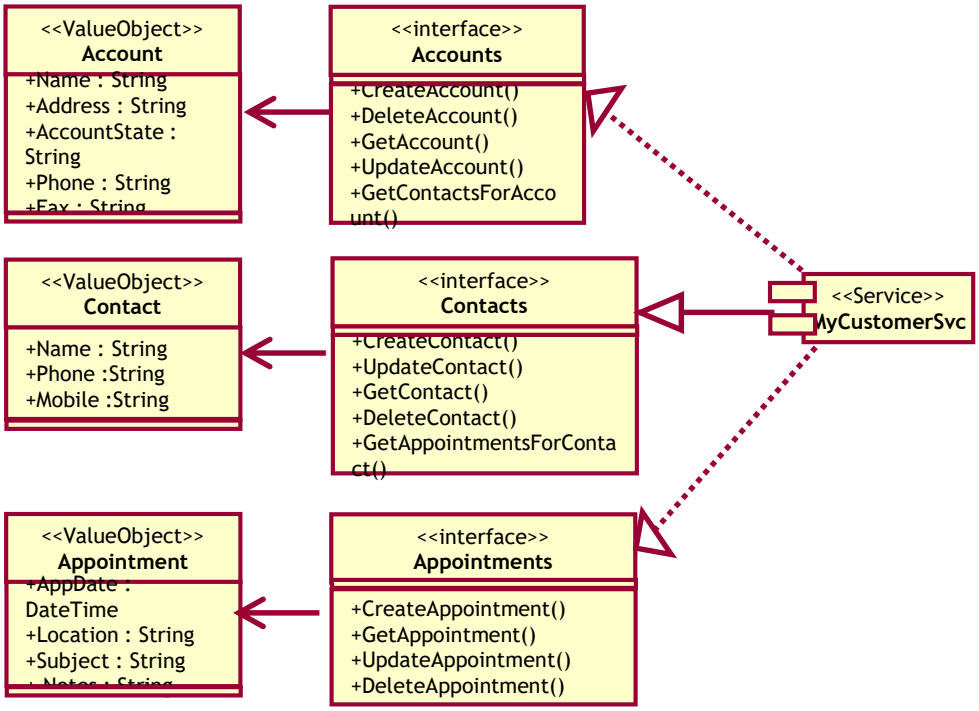
Task1: Create Functional Design:
Step9: Expose Components as Service:

- A **service** is a logical entity. It is a contract defined by one or more published interfaces. This specification is implemented by software entities, called **service provider**. A registry called as a **service locator** allows look up of service interfaces and the locations of the service.
- Service components enhance flexibility and maintainability of the systems.
- Services are much more loosely coupled as compared to components. They have published and discoverable interfaces.

Instructor Notes:

Refer to Qzen for more details on service components.

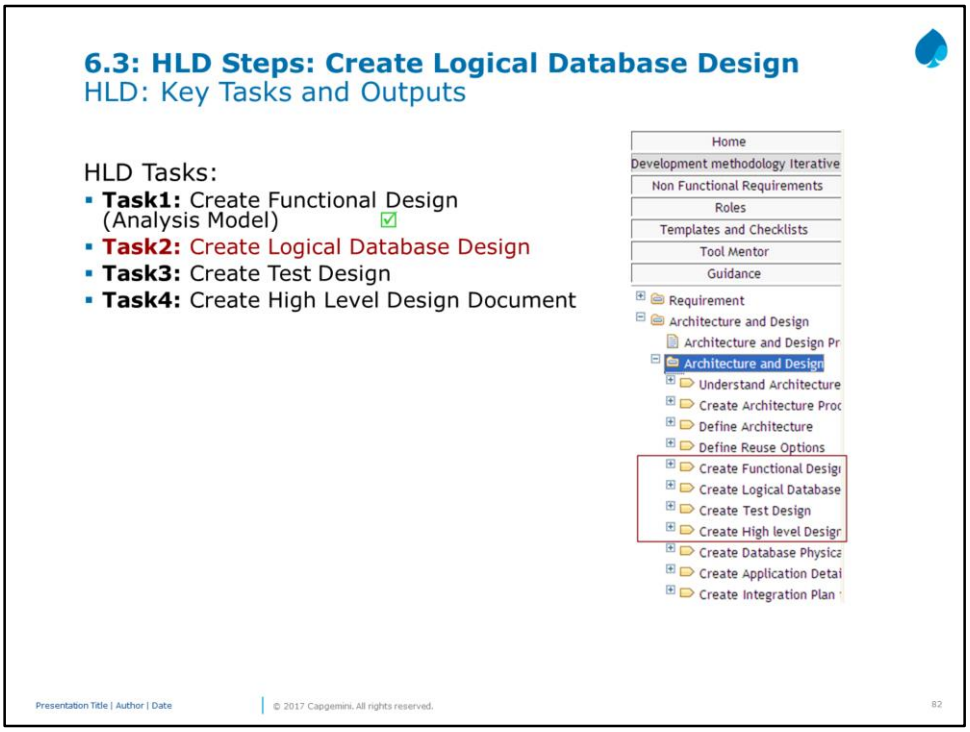
Example: Expose Components as Service:



- The slide shows an example of **service oriented design**. Value object represents the instance state of the object. **MyCustomerSvc** is the Service Provider. Note that a service is more coarsely grained. In a simple component based scenario, we would have separate components for Accounts, Contacts, and so on.
- When working with service based design, it is common to include a Service Layer in the architecture above the layer where the component resides.

Instructor Notes:

Each of these subsequent tasks will be briefly walked through as there will be other key members involved eg. Database designers, Test Designers etc.

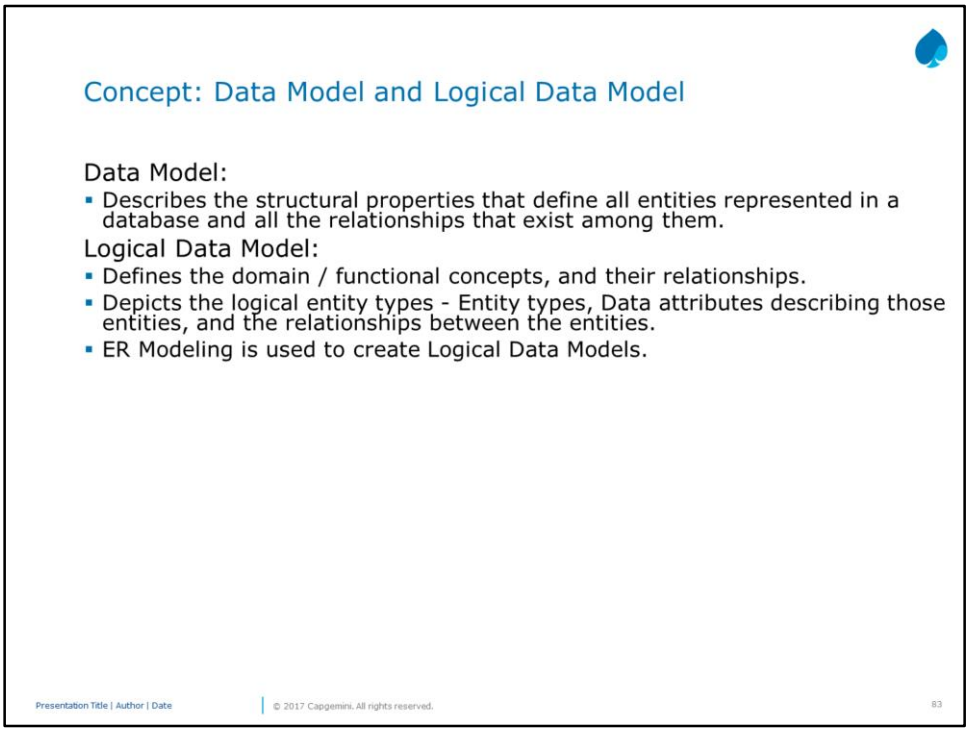


<u>Key Tasks and Outputs</u>:

- Creating Functional design was our key focus as far as OOAD was concerned.
- Now, let us briefly look at the other steps of High Level Design.

Instructor Notes:

Most participants would have some idea about this, so keep it discussion based.

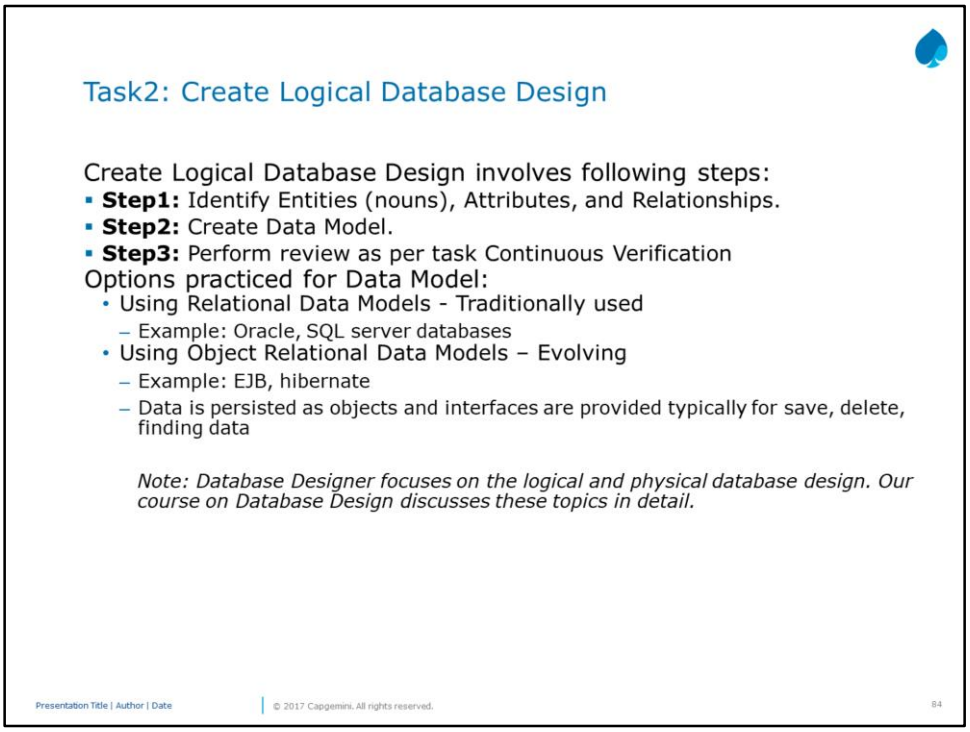


<u>Concept: Data Model and Logical Data Model</u>:

- In many applications, we would have data that is to be persisted using databases. To understand how to store and manage this data, we develop **data models**.
- Data Models can be **logical** and **physical**.
- HLD focuses on the **logical data model**, which is later refined into the **physical data model**.

Instructor Notes:

Explain transition from Analysis Classes to Relational Data Model and Object Relational Model. A Simple example like Account Holder and Account Class can be considered.

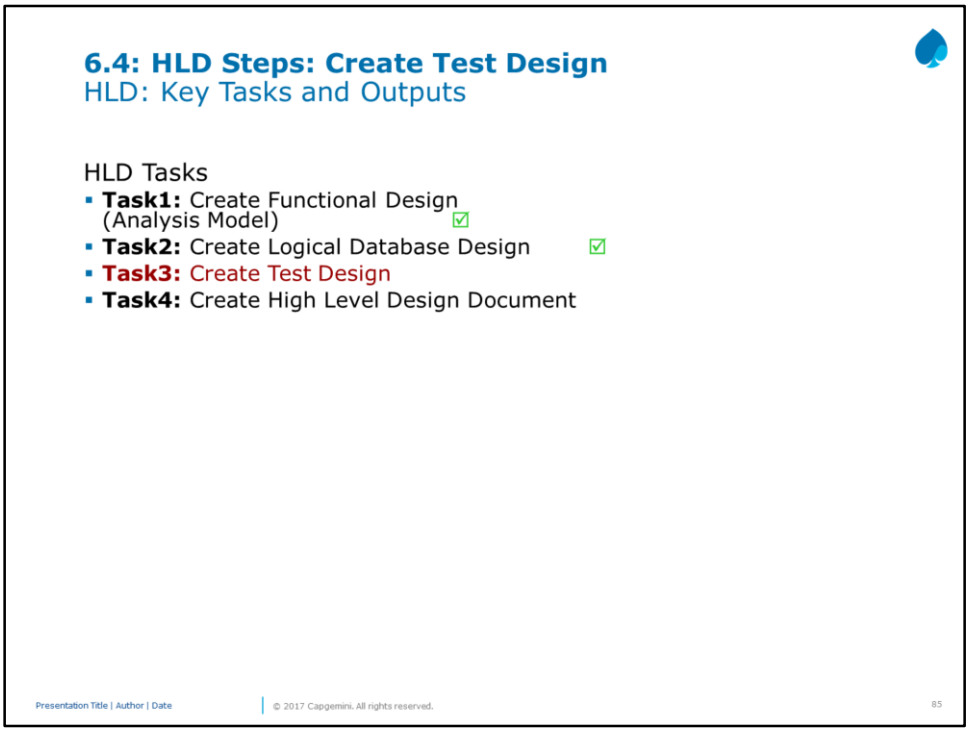


Task2: Create Logical Database Design:

- An important aspect of creating the **logical database design** is to study the **Analysis Model** and identify the data that needs to be persisted in the database. Usually, **persistence** is needed for many of the **entity classes**.
  - If we go with the approach of using **Relational Data Models**, then ER Modeling will be done. In the ER Model, classes that need persistence become the Entities, Attributes of the Class become attributes of the Entity, and relationships between classes become relationship between entities.
  - If we go with the **Object Relational frameworks**, then typically the framework itself would generate most of the required structures and decide how to take care of the required persistence.

Instructor Notes:

Each of these subsequent tasks will be briefly walked through as there will be other key members involved eg. Database designers, Test Designers etc.



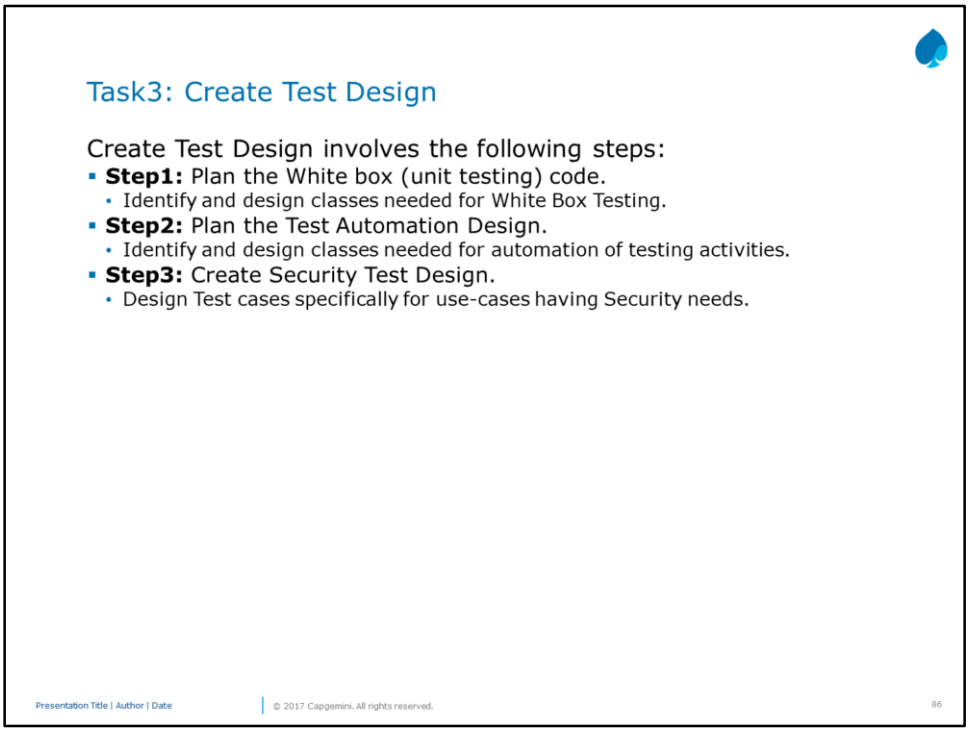
HLD: Key Tasks and Outputs:

- In the following slides, let us have a brief look at what is involved in the “Create Test Design” task.

Instructor Notes:

Test Discipline will have more details.

Security testing is a vast area in itself. For more details, refer to Qzen.

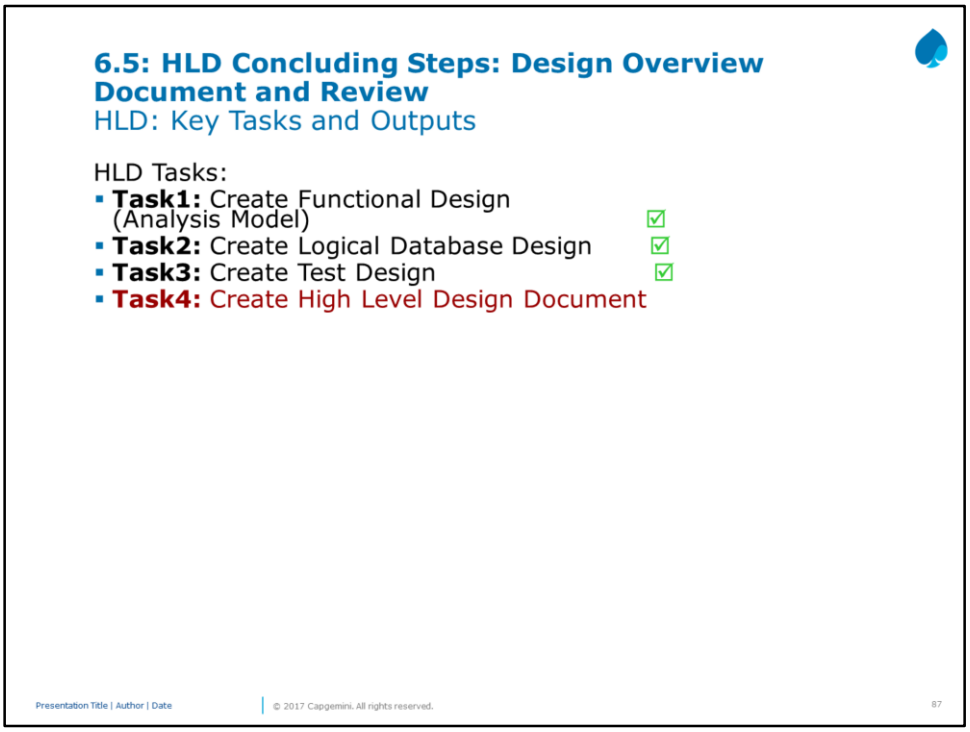


Task3: Create Test Design:

- White box testing deals with testing the internal logic and structure of the code. This may require additional classes, which need to be designed.
- Many a times, unit testing is automated with the help of various tools. Such an automation demands test class classes. These too need to be identified and designed.
- Based on threats, vulnerabilities, and the **Threat Model** that is created, you have to focus on coming up with test cases for validating the security. Many a times, tools are used for security testing based on the criticality of the key threats to the application.

Instructor Notes:

These are now the concluding activities of HLD.

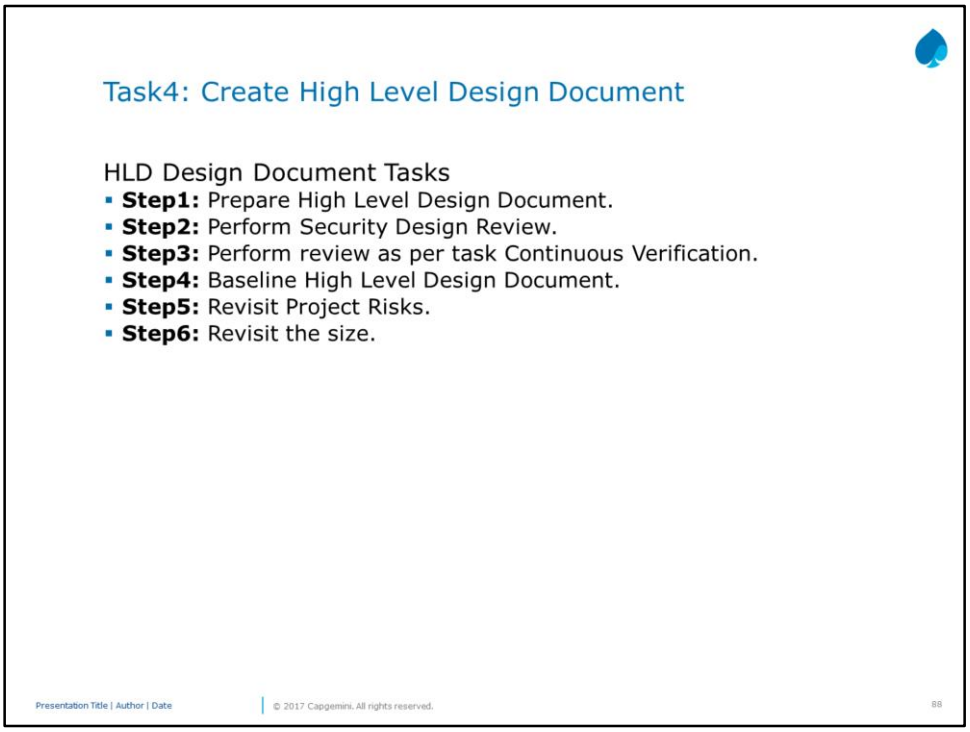


HLD Concluding Steps: High Level Design Document and Review:

- Now, let us look at the concluding tasks of the HLD activity.

Instructor Notes:

Walkthrough the Design document template available on Qzen.

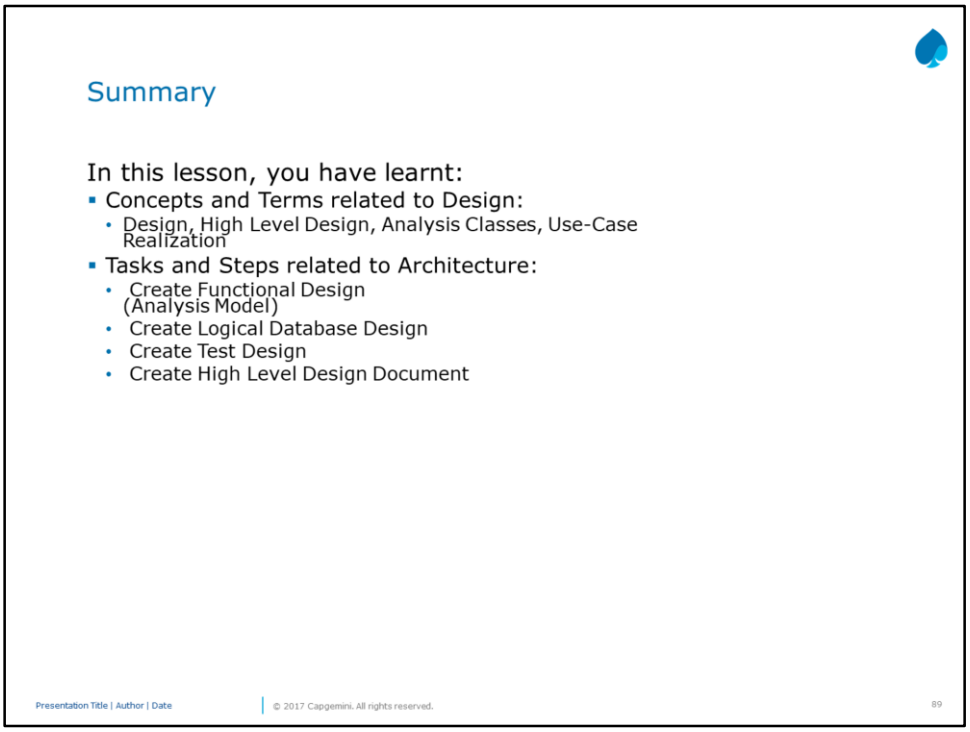


Task4: Create High Level Design Document:

- Having completed the HLD activities, you now have to document the key design decisions in a Design Document. Qzen provides a template for the Design Document.
- The UML Analysis Model in itself provides details of the Analysis Classes and the Use-Case Realizations.
- The design has to be formally reviewed using the design review checklist and then baselined after incorporating the review comments. Review is
  - To verify that the analysis results meet the functional requirements made on the system
  - To verify that the analysis objects and interactions are consistent
- The remaining are project management steps in view of the analysis model evolved at this point in time

Instructor Notes:

Subsequent lessons discuss about moving into Design Model in the Low Level Design; also the component designing.



Summary:
We have now completed the activities of High Level Design.

Instructor Notes:

Answers for Review Questions:

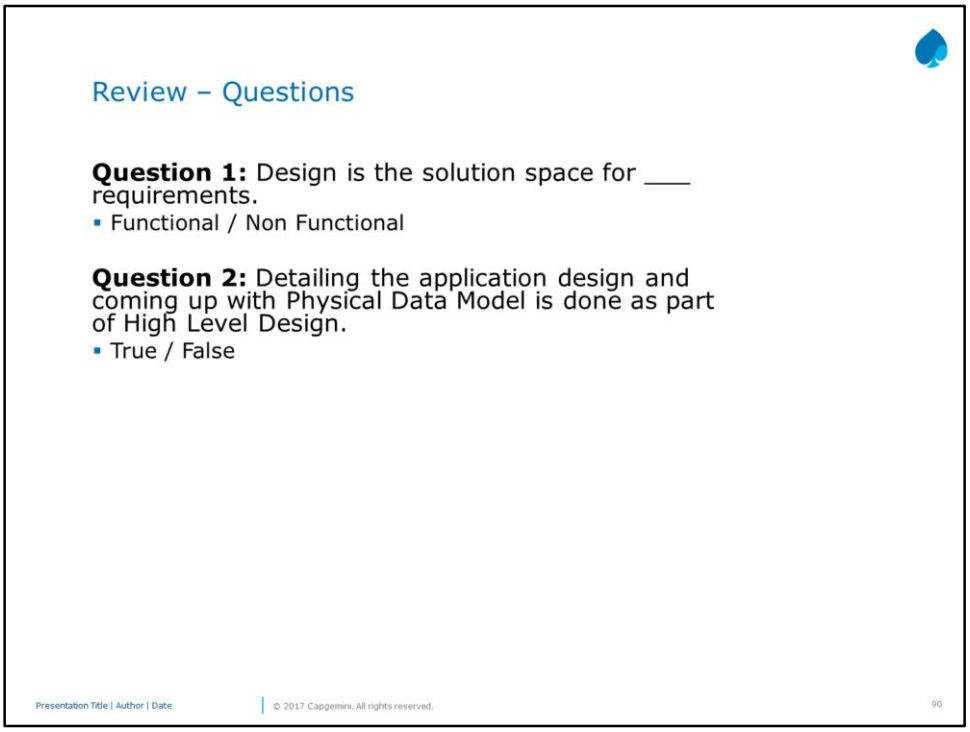
Answer 1: Functional

Answer 2: False. It is LLD

Answer 3: Design Pattern

Answer 4: Boundary, Control, and Entity

Answer 5: True



Instructor Notes:

Answers for Review Questions:

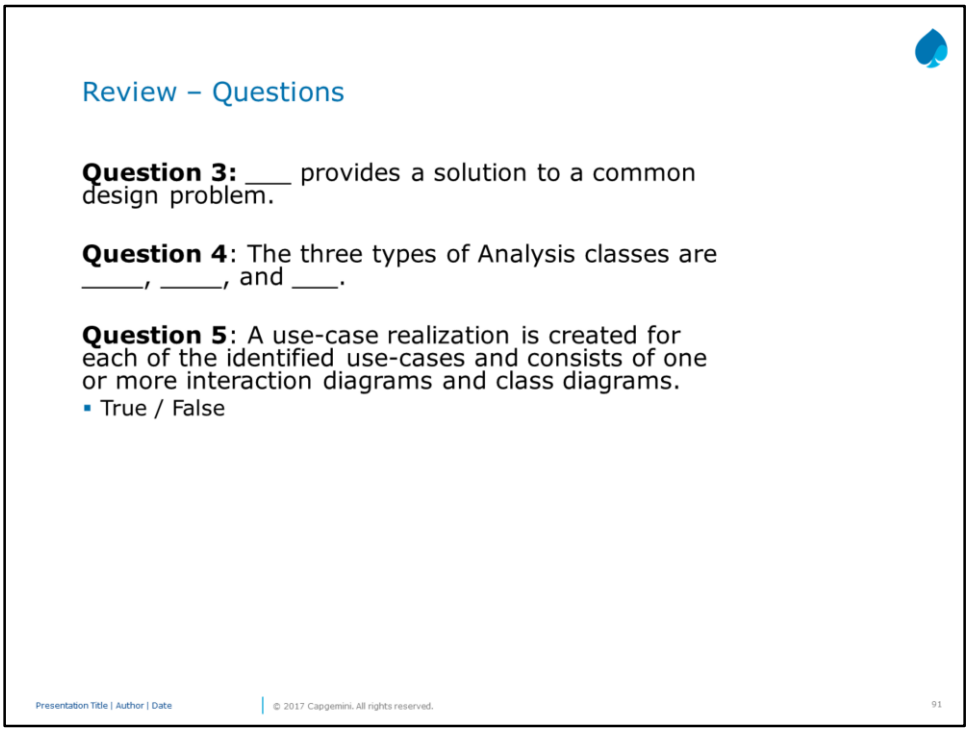
Answer 1: Functional

Answer 2: False. It is LLD

Answer 3: Design Pattern

Answer 4: Boundary, Control, and Entity

Answer 5: True



Instructor Notes:

Answers for Review Questions:

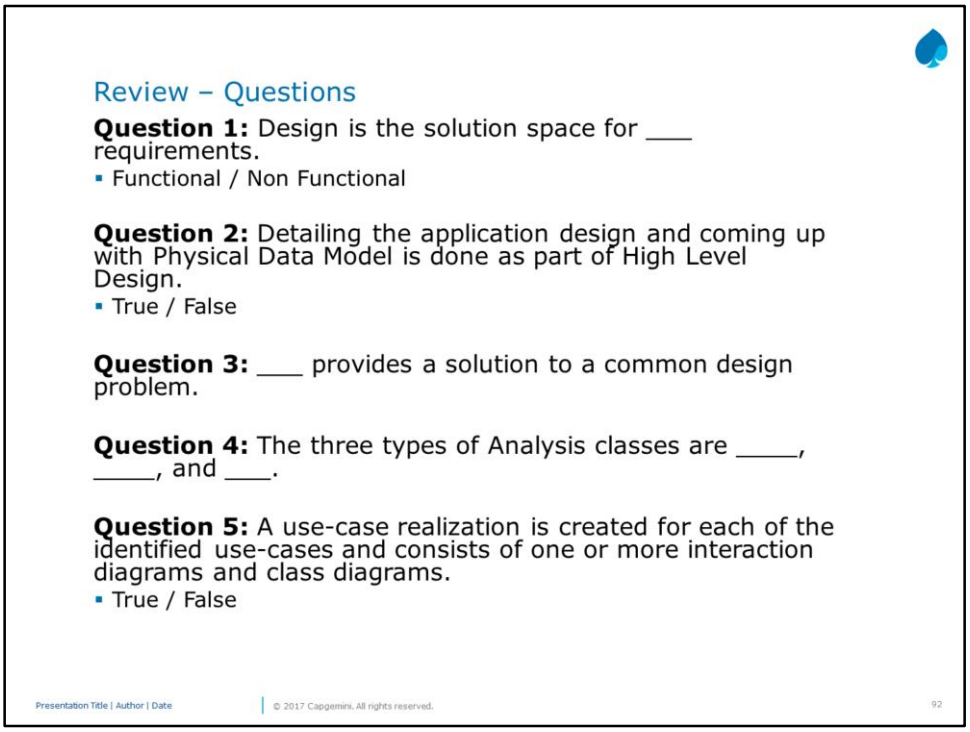
Answer 1: Functional

Answer 2: False. It is LLD

Answer 3: Design Pattern

Answer 4: Boundary, Control, and Entity

Answer 5: True



Instructor Notes:

Answers for Review Questions:

Answer 6: False. It is incoming messages.

Answer 7: Subsystem / Component Context Diagram