OOAD Reference Material

Requirements Overview:

- Input from Business Modeling.
- Actors- are someone or something that is external to the system, but is also interacting with the system that we are developing. Actors can act on the system or can be acted upon.
- Use cases – are sequences of actions that produce an observable result of value to an actor.
- Use cases are not actions but they are collection of actions.
- Use cases should not lead to Functional decomposition (as in SSAD), this is what is referred as Writing C code in C ++.

Analysis & Design:

- Input for Analysis & Design – SRS Package. (Problem Statement, Supplementary Specs, Glossary, Use Case Model, Use case Specification)

1.	Architecture Analysis
2.	Use case Analysis
3.	Design Mechanisms
4.	Design Elements
5.	Run Time Architecture
6.	Deployment
7.	Use case Design
8.	Subsystem Design
9.	Class Design
10	Data base Design

STEP 1 :

Architecture Analysis:
Architect would identify Patterns & Frameworks that are needed to develop the project or product.

Patterns exist at various levels in the SDLC.

Business Patterns

Architectural Patterns

Design Patterns

Idioms.

The choice of Architectural Patterns depends on the problem, Context, Constraints, Solutions & the domain of the problem.

Pattern Classification	Patterns
Mud 2 Structures	Layers, Pipes & Filters, Black board
Interactive	MVC, Document view, PAC
Distribution	Broker, Proxy
Adaptable	Micro kernel

Architecture Mechanisms
AAM- are intermediate solutions b/w requirements to Implementation. It is generic and does not talk of Technology or Implementation. It is done at Analysis level so that Implementation Environment does not bind decisions.

Design & Implementation Mechanism decisions are taken at the Architectural Design phase ie., by the end of Elaboration Phase .

Classes	AAM	DM	IM	DP
	Persistency	OODBMS	Object Store	OO- oodbms Pattern

Key Abstractions are important entity classes necessary to build the system. They are the building blocks of the application that you are developing. They are Nouns. This is done at Architectural Analysis to help Designer identify other classes centered on the key abstract classes. The no of Key abstract classes that you identify may vary from one designer to another.

Finally, Architect with Project Manager will Elaborate risk list and work on Mitigation and Contingency Plans. They have to identify the use cases that have to be realized so that risk is minimized as the project proceeds.

Risk No & Name	Priority	Mitigation Plan	Contingency Plan	Iteration plan
5. Get data with in 10 secs	1	Develop prototypes in early iterations	Look for caching	Iteration 1

STEP 2 : Use Case Analysis:

The technique for finding analysis classes described in this module uses three different perspectives of the system to drive the identification of candidate classes. These three perspectives are:

The boundary between the system and its actors

The information the system uses

The control logic of the system.

The use of stereotypes to represent these perspectives (e.g., boundary, control and entity) results in a more robust model because they isolate those things most likely to change in a system: the interface/environment, the control flow and the key system entities. These stereotypes are conveniences used during analysis that disappear in design.

Identification of classes means just that: they should be identified, named, and described briefly in a few sentences.

The different stereotypes are discussed in more detail throughout this module.

Analysis classes represent an early conceptual model for ‘things in the system which have responsibilities and behavior’. Analysis classes are used to capture a ‘first-draft’, rough-cut of the object model of the system.

Analysis classes handle primarily functional requirements. They model objects from the problem domain. Analysis classes can be used to represent "the objects we want the system to support" without making a decision on how much of them to support with hardware and how much with software.

There are three aspects of the system that are likely to change:

the boundary between the system and its actors,

the information the system uses, and

the control logic of the system.
In an effort to isolate the parts of the system that will change, different types of analysis classes are identified with a “canned” set of responsibilities:

boundary,
entity and
control classes.
Stereotypes may be defined for each type. These distinctions are used during analysis, but disappear in design.
The different types of analysis classes can be represented using different icons or with the name of the stereotype in guillemets (<< >>): <<boundary>>, << control>>, <<entity>>.
Each of these types of analysis classes are discussed on the following slides.

Finding a candidate set of roles is the first step in the transformation of the system from a mere statement of required behavior to a description of how the system will work.

The analysis classes, taken together, represent an early conceptual model of the system. This conceptual model evolves quickly and remains fluid for some time as different representations and their implications are explored. Formal documentation can impede this process, so be careful how much energy you expend on maintaining this ‘model’ in a formal sense; you can waste a lot of time polishing a model which is largely expendable. Analysis classes rarely survive into the design unchanged. Many of them represent whole collaborations of objects, often encapsulated by subsystems.

Analysis classes are 'proto-classes', which are essentially "clumps of behavior". These analysis classes are early conjectures of the composition of the system; they rarely survive intact into implementation. Many of the analysis classes morph into something else later (subsystems, components, split classes, combined classes). They provide us with a way of capturing the required behaviors in a form that we can use to explore the behavior and composition of the system. Analysis classes allow us to "play" with the distribution of responsibilities, re-allocating, as necessary.

A boundary class intermediates between the interface and something outside the system. Boundary class insulate the system from changes in the surroundings (e.g. changes in interfaces to other systems, changes in user requirements, etc.), keeping these changes from affecting the rest of the system.

A system may have several types of boundary classes:
User interface classes - Classes which intermediate communication with human users of the system.
System interface classes - Classes which intermediate communication with other systems. A boundary class, which communicates with an external system, is responsible for managing the dialogue with the external system; it provides the interface to that system for the system being built.
Device interface classes - Classes which provide the interface to devices which detect external events. These boundary classes capture the responsibilities of the device or sensor.
One recommendation for the initial identification of boundary classes is one boundary class per actor/use-case pair.

Entity objects represent the key concepts of the system being developed. Entity classes provide another point of view from which to understand the system because they show the logical data structure. Knowing the data structure can help you understand what the system is supposed to offer its users.

Frequent sources of inspiration for entity classes are the:
Glossary (developed during requirements)
Business-domain model (developed during business modeling, if business modeling has been performed)
Use-case flow of events (developed during requirements)
Key abstractions (identified in Architectural Analysis)
As mentioned earlier, sometimes there is a need to model information about an actor within the system. This is not the same as modeling the actor (actors are external. by definition). In this case, the information about the actor is modeled as an entity class. These classes are sometimes called “surrogates”.

Taking the use case flow of events as input, underline the noun phrases in the flow of events. These form the initial candidate list of analysis classes.
Next, go through a series of filtering steps where some candidate classes are eliminated. This is necessary due to the ambiguity of the English language. The result of the filtering exercise is a refined list of candidate entity classes. While the filtering approach does add some structure to what could be an ad-hoc means of identifying classes, people generally filter as they go rather than blindly accepting all nouns and then filtering.

Control classes provide coordinating behavior in the system. The system can perform some use cases without control classes by using just entity and boundary classes. This is particularly true for use cases that involve only the simple manipulation of stored information. More complex use cases generally require one or more control classes to coordinate the behavior of other objects in the system. Examples of control classes include transaction managers, resource coordinators and error handlers.

Control classes effectively decouple boundary and entity objects from one another, making the system more tolerant of changes in the system boundary. They also decouple the use-case specific behavior from the entity objects, making them more reusable across use cases and systems.

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)

Although complex use cases may need more than one control class it is recommended, for the initial identification of control classes, that only one control class be created per use case.

A control class is a class used to model control behavior specific to one or more use cases. Control objects (instances of control classes) often control other objects, so their behavior is of the coordinating type. Control classes encapsulate use-case-specific behavior.

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, 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.

One recommendation is to identify one control class per use case. Each control class is responsible for orchestrating/controlling the processing that implements the functionality described in the associated use case.
In the above example, the Registration Controller <<control>> class has been defined to orchestrate the Register for Courses processing within the system.

A driving influence on where a responsibility should go is the location of the data needed to perform the operation.
The best case is that there is one class that has all the information needed to perform the responsibility. In that case, the responsibility goes with the data (after all, that's one of the tenets of OO -- data and operations together).
If this isn't the case, the responsibility may need to be allocated to a "third party" class that has access to the information needed to perform the responsibility. Classes and/or relationships may need to be created to make this happen. Be careful when adding relationships -- all relationships should be consistent with the abstractions they connect. Don't just add relationships to support the implementation without considering the overall affect on the model. Class relationships will be discussed later in this module.
When a new behavior is identified, check to see if there is an existing class that has similar responsibilities, reusing classes where possible. You should create new classes only when you are sure that there is no existing object that can perform the behavior.

A sequence diagram describes a pattern of interaction among objects, arranged in a chronological order. It shows the objects participating in the interaction and the messages they send.

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.
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. The arrow may also be labeled with a sequence number.
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. 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 1.1 depends on message 1.
Scripts describe the flow of events textually.

A collaboration diagram describes a pattern of interaction among objects. It shows the objects participating in the interaction by their links to each other and the messages that they send to each other.
An object is represented in one of three ways:
Objectname:Classname
ObjectName
:ClassName
A link is a relationship between objects and can be used to send messages. In collaboration diagrams, a link is shown as a solid line between two objects. An object interacts with, or navigates to, other objects through its links to these objects. A link is defined as an instance of an association.
A message is a communication between objects that conveys information with the expectation that activity will result. In collaboration diagrams, a message is shown as a labeled arrow placed near a link. This means that the link is used to transport, or otherwise implement the delivery of the message to the target object. The arrow points along the link in the direction of the target object (the one that receives the message). The arrow is labeled with the name of the message, and its parameters. The arrow may also be labeled with a sequence number to show the sequence of the message in the overall interaction. Sequence numbers are often used in collaboration diagrams because they are the only way of describing the relative sequencing of messages. A message can be unassigned, meaning that its name is a temporary string that describes the overall meaning of the message. You can later assign the message by specifying the operation of the message's destination object. The specified operation will then replace the name of the message.

Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.
Collaboration diagrams emphasize the structural collaboration of a society of objects and provide a clearer picture of the patterns of relationships and control that exist amongst the objects participating in a use case. Collaboration diagrams show more structural information (i.e., the relationships among objects). Collaboration diagrams are better for understanding all the effects on a given object and for procedural design.
Sequence diagrams show the explicit sequence of messages and are better for real-time specifications and for complex scenarios. A sequence diagram includes chronological

sequences, but does not include object relationships. Sequence numbers are often omitted in sequence diagrams, in which the physical location of the arrow shows the relative sequence. On sequence diagrams, the time dimension is easier to read, operations and parameters are easier to present, and a larger number of objects are easier to manage than in collaboration diagrams.
Both sequence and collaboration diagrams allow you to capture semantics of the use-case flow of events; they help identify objects, classes, interactions, and responsibilities; and they help validate the architecture.

Examine classes to ensure they have consistent responsibilities. When a class's responsibilities are disjoint, split the object into two or more classes. Update the interaction diagrams accordingly.
Examine classes to ensure that there are not two classes with similar responsibilities. When classes have similar responsibilities, combine them and update the interaction diagrams accordingly.
Sometimes a better distribution of behavior will become evident while you were working on another interaction diagram. In this case, go back to the previous interaction diagram and redo it. It's better (and easier) to change things now than later in design. Take the time to set the diagrams right, but don't get hung-up trying to optimize the class interactions.
A class with only one responsibility is not a problem, per se, but it should raise questions on why it is needed. Be prepared to challenge and justify the existence of all classes.

Attributes are used to store information. Attributes are atomic things with no responsibilities.
The attribute name should be a noun that clearly states what information the attribute holds. The description of the attribute should describe what information is to be stored in the attribute; this can be optional when the information stored is obvious from the attribute name.
During analysis, the attribute types should be from the domain, and not adapted to the programming language in use. For example, in the above diagram, enum will need to be replaced with a true enumeration that describes the days the CourseOffering is offered (e.g., MWF, TR, etc.).

Sources of possible attributes:
domain knowledge
requirements
glossary
domain model
business model
Attributes are used instead of classes where :
Only the value of the information, not it's location, is important

The information is uniquely "owned" by the object to which it belongs; no other objects refer to the information

The information is accessed by operations which only get, set or perform simple transformations on the information; the information has no "real" behavior other than providing its value

If, on the other hand, the information has complex behavior, or is shared by two or more objects the information should be modeled as a separate class.

Attributes are domain dependent (an object model for a system, includes those characteristics that are relevant for the problem domain being modeled).

Remember, the process is use-case-driven. Thus, all discovered attributes should support at least one use case. For this reason, the attributes that are discovered are affected by what functionality/domain being modeled.

Associations represent structural relationships between objects of different classes; it connects instances of two or more classes together for some duration.

You can use associations to show that objects know about other objects. Sometimes, objects must hold references to each other to be able to interact, for example send messages to each other; thus, in some cases associations may follow from interaction patterns in sequence diagrams or collaboration diagrams.

Most associations are simple (exist between exactly two classes), and are drawn as solid paths connecting pairs of class symbols. Ternary relationships are also possible. Sometimes, a class has an association to itself. This does not necessarily mean that an instance of that class has an association to itself; more often, it means that one instance of the class has associations to other instances of the same class.

An association may have a name that is placed on, or adjacent to the association path. The name of the association should reflect the purpose of the relationship and be a verb phrase. The name of an association can be omitted, particularly if role names are used.

Avoid names like "has" and "contains", as they add no information about what the relationships are between the classes.

To find relationships, start studying the links in the collaboration diagrams. Links between classes indicate that objects of the two classes need to communicate with one another to perform the use case. Thus, an association or an aggregation is needed between the associated classes.

Reflexive links do not need to be instances of reflexive relationships; an object can send messages to itself. A reflexive relationship is needed when two different objects of the same class need to communicate.

The navigability of the relationship should support the required message direction. In the above example, if navigability was not defined from the Client to the Supplier, then the Perform Responsibility message could not be sent from the Client to the Supplier.
Focus only on associations needed to realize the use cases; don't add association you think "might" exist unless they are required based on the interaction diagrams.
Remember to give the associations role names and multiplicities. You can also specify navigability, though this will be refined in Class Design.
Write a brief description of the association to indicate how the association is used, or what relationships the association represents.

As analysis classes are identified, it is important to identify the analysis mechanisms that apply to the identified classes.
The classes that must be persistent are mapped to the Persistency mechanism.
The classes that are maintained within the legacy Course Catalog system are mapped to the Legacy Interface mechanism.
The classes for which access must be controlled (i.e., control who is allowed to read and modify instances of the class) are mapped to the Security mechanism. Note: The Legacy Interface classes do not require additional security as they are read-only and are considered readable by all.
The classes that are seen to be distributed are mapped to the Distribution mechanism. The distribution identified during analysis is that which is specified/implied by the user in the initial requirements. Distribution will be discussed in detail in the Describe Distribution module. For now, just take it as an architectural given that all control classes are distributed for the OOAD course example and exercise.

We now have a pretty good feeling about our Analysis Model. Now it’s time to review our work for completeness and consistency.
Be sure to:
Verify that the analysis classes meet the functional requirements made on the system
Verify that the analysis classes and their relationships are consistent with the collaborations they support.
It is very important that you evaluate your results at the conclusion of Use-Case Analysis.
The number of reviews, the formality of the reviews, and when they are performed will vary depending on the process defined for the project.

The above checkpoints for the analysis classes might be useful.

Note: All checkpoints should be evaluated with regards to the use cases being developed for the current iteration.
The class should represent a single well-defined abstraction. If not, consider splitting it.
The class should not define any attributes or responsibilities that are not functionally coupled to the other attributes or responsibilities defined by that class.
The classes should offer the behavior the use-case realizations and other classes require.
The class should address all specific requirements on the class from the requirement specification.
Remove any attributes and relationships if they are redundant or are not needed by the use-case realizations.

The differences between analysis and design are ones of focus and emphasis. The above slide lists the things that you focus on in analysis versus design.
The goal in Analysis is to understand the problem and to begin to develop a visual model of what you are trying to build, independent of implementation and technology concerns. Analysis focuses on translating the functional requirements into software concepts. The idea is to get a rough cut at the objects that comprise our system, but focusing on behavior (and therefore encapsulation). We then move very quickly, nearly seamlessly into “design” and the other concerns.
A goal of design is to refine the model with the intention of developing a design model that will allow a seamless transition to the coding phase. In design, we adapt to the implementation and the deployment environment. The implementation environment is the 'developer' environment, which is a software superset and a hardware subset of the deployment environment
In modeling, we start with an object model that closely resembles the real world (analysis), and then find more abstract (but more fundamental) solutions to a more generalized problem (design). The real power of software design is that it can create more powerful metaphors for the real world which change the nature of the problem, making it easier to solve.

STEP 3: Design Mechanisms:
Identify Design Mechanisms is performed by the Architect, once per iteration.
Purpose
To refine the analysis mechanisms into design mechanisms based on the constraints imposed by the implementation environment.

Input Artifacts
Supplementary Specifications

Design Guidelines
Software Architecture Document
Analysis Classes
Design Model

Resulting Artifacts
Design model elements
Classes
Packages
Subsystems
Updated Software Architecture document
Design Guidelines

Identify the clients of each analysis mechanism. Scan all clients of a given analysis mechanism, looking at the characteristics they require for that mechanism. For example, a number of analysis objects may make use of a Persistence mechanism, but their requirements on this may widely vary: a class which will have a thousand persistent instances has significantly different persistence requirements than a class which will have four million persistent instances. Similarly, a class whose instances must provide sub-millisecond response to instance data requires a different approach than a class whose instance data is accessed through batch applications.
Identify characteristic profiles for each analysis mechanism. There may be widely varying characteristics profiles, providing varying degrees of performance, footprint, security, economic cost, etc. Each analysis mechanism is different - different characteristics will apply to each. Many mechanisms will require estimates of the number and size of instances to be managed. The movement of large amounts of data through any system will create tremendous performance issues that must be dealt with.
Group clients according to their use of characteristic profiles. Identify design mechanism for groups of clients that seem to share a need for an analysis mechanism with a similar characteristics profile. These groupings provide an initial cut at the design mechanisms. An example analysis mechanism, "inter-process communication", may map onto a design mechanism "object request broker". Different characteristic profiles will lead to different design mechanisms which emerge from the same analysis mechanism. The simple persistence mechanism in analysis will give rise to a number of persistence mechanisms in design: in-memory persistence, file-based, database-based, distributed, etc.

During Architecture Analysis we identified the key architectural mechanisms that may be required to solve the problem. Now it is time to refine these and incorporate the decisions we have made about our implementation.

A design mechanism assumes some details of the implementation environment, but it is not tied to a specific implementation (as is an implementation mechanism). Examples of design mechanisms include:

for Persistency - RDBMS, OODBMS, flash card, in-memory storage

for inter-process communication (IPC) - shared memory, function-call-like IPC, semaphore-based IPC

An implementation mechanism is used during the implementation process. They are refinements of design mechanisms, and specify the exact implementation of the mechanism. They are are bound to a certain technology, implementation language, vendor, etc. Some examples of implementation mechanisms include: the actual programming language, COTS products, database (Oracle, Sybase), and the inter-process communication/distribution technology in use (COM/DCOM, CORBA).

The above example shows the architectural mechanism decisions that have been made for our example. For RDBMS Persistency (i.e., legacy data access), JDBC was chosen. For OODBMS Persistency, ObjectStore was chosen, and for Distribution, RMI was chosen. The JDBC mechanism is presented in this module. Information on the ObjectStore and RMI mechanisms are provided in the Additional Information Appendix.

These characteristics were first introduced in Architectural Analysis. When we described the persistency architectural mechanism we used the following characteristics;

Persistency: For all classes whose instances may become persistent, we need to identify:

Granularity: Range of size of the persistent objects

Volume: Number of objects to keep persistent

Duration: How long to keep persistent objects

Access mechanism: How is a given object uniquely identified and retrieved?

Access frequency: Are the objects more or less constant; are they permanently updated?

Reliability: Shall the objects survive a crash of the process; the processor; or the whole system?

STEP 4: Design Elements:

Identify Design Elements is performed by the Architect, once per iteration.

Purpose

To analyze interactions of analysis classes to identify design model elements

To analyze interactions of analysis classes to find interfaces, design classes and design subsystems

To refine the architecture, incorporating reuse where possible.

To identify common solutions to commonly encountered design problems

To include architecturally significant design model elements in the Logical View section of the Software Architecture Document.

Input Artifacts

Supplementary Specifications

Design Guidelines

Software Architecture Document

Analysis Classes

Design Model
Resulting Artifacts
Design model elements
Classes
Interfaces
Packages
Subsystems.

Identify Design Elements is where the analysis classes identified during Use-Case Analysis are refined into design elements (e.g., classes or subsystems). Analysis classes handle primarily functional requirements, and model objects from the "problem" domain; design elements handle non-functional requirements, and model objects from the "solution" domain.
It is in Identify Design Elements that we decide which analysis ‘classes’ are really classes, which are subsystems (which must be further decomposed), and which are existing components and don’t need to be “designed” at all.
Once the design classes and subsystems have been created, each must be given a name and a short description. The responsibilities of the original analysis classes should be transferred to the newly-created subsystems. In addition, the identified design mechanisms should be linked to design elements.

If the analysis class is simple and already represents a single logical abstraction, then it can be directly mapped, one-to-one, to a design class. Typically, entity classes survive relatively intact into design.
Throughout the design activities, analysis classes are refined into design elements (e.g., design classes, packages, subsystems, etc.). Some analysis classes may be split, joined, removed, etc. In general, there is a many-to-many mapping between analysis classes and design elements. The possible mappings include the following.
An analysis class can become:
One single class in the design model.
A part of a class in the design model.
An aggregate class in the design model (meaning that the parts in this aggregate may not be explicitly modeled in the analysis model.)
A group of classes that inherits from the same class in the design model.
A group of functionally related classes in the design model (e.g., a package).
A subsystem in the design model .
A relationship in the design model.
A relationship between analysis classes can become a class in the design model.

Part of an analysis class can be realized by hardware, and not modeled in the design model at all.
Any combination of the above.

When identifying classes, they should be grouped into Packages, for organizational and configuration management purposes.
The Design Model can be structured into smaller units to make it easier to understand. By grouping Design Model elements into packages and subsystems, then showing how those groupings relate to one another, it is easier to understand the overall structure of the model.
You may want to partition the Design Model for a number of reasons:
You can use packages and subsystems as order, configuration, or delivery units when a system is finished.
Allocation of resources and the competence of different development teams may require that the project be divided among different groups at different sites.
Subsystems can be used to structure the design model in a way that reflects the user types. Many change requirements originate from users; subsystems ensure that changes from a particular user type will affect only the parts of the system that correspond to that user type.
Subsystems are used to represent the existing products and services that the system uses.

When the boundary classes are distributed to packages there are two different strategies that can be applied. Which one to choose depends on whether or not the system interfaces are likely to change greatly in the future.
If it is likely that the system interface will be replaced, or undergo considerable changes, the interface should be separated from the rest of the design model. When the user interface is changed, only these packages are affected. An example of such a major change is the switch from a line-oriented interface to a window-oriented interface.
If no major interface changes are planned, changes to the system services should be the guiding principle, rather than changes to the interface. The boundary classes should then be placed together with the entity and control classes with which they are functionally related. This way, it will be easy to see what boundary classes are affected if a certain entity or control class is changed.
Mandatory boundary classes that are not functionally related to any entity or control classes, should be placed in separate packages, together with boundary classes that belong to the same interface.
If a boundary class is related to an optional service, group it in a separate subsystem with the classes that collaborate to provide the service. The subsystem will map onto an optional component which will be provided when the optional functionality is ordered.

A package should be identified for each group of classes that are functionally related. There are several practical criteria that can be applied when judging if two classes are functionally related. These are, in order of diminishing importance:
If changes in one class' behavior and/or structure necessitate changes in another class, the two classes are functionally related.
It is possible to find out if one class is functionally related to another by beginning with a class - for example, an entity class - and examining the impact of it being removed from the system. Any classes that become superfluous as a result of a class removal are somehow connected to the removed class. By superfluous, we mean that the class is only used by the removed class, or is itself dependent upon the removed class.
Two objects can be functionally related if they interact with a large number of messages, or have an otherwise complicated intercommunication.
A boundary class can be functionally related to a particular entity class if the function of the boundary class is to present the entity class.
Two classes can be functionally related if they interact with, or are affected by changes in, the same actor. If two classes do not involve the same actor, they should not lie in the same package. The last rule can, of course, be ignored for more important reasons

In Architectural Analysis, we discussed package dependencies. Now let's look at package dependencies in more detail and see how visibility can be defined.
Visibility may be defined for package elements the same way it is defined for class attributes and operations. This visibility allows you to specify how other packages can access the elements that are owned by the package.
The visibility of a package element can be expressed by including a visibility symbol as a prefix to the package element name.
There are three types of visibility defined in the UML:
Public: Public classes can be accessed outside of the owning package. Visibility symbol: +.
Protected: Protected classes can only be accessed by the owning package and any packages that inherit from the owning package. Visibility symbol: #.
Private: Private classes can only be accessed by classes within the owning package. Visibility symbol: -.
The public elements of a package constitute the package's interface. All dependencies on a package should be dependencies on public elements of the package.
Package visibility provides support for the OO principle of encapsulation.

Package coupling is good and bad: good, because coupling represent re-use, and bad, because coupling represents dependencies that make the system harder to change and evolve. Some general principles can be followed:

Packages should not be cross-coupled (i.e. co-dependent); e.g. two packages should not be dependent on one another. In these cases, the packages need to be reorganized to remove the cross-dependencies.

Packages in lower layers should not be dependent upon packages in upper layers. Packages should only be dependent upon packages in the same layer and in the next lower layer. In these cases, the functionality needs to be repartitioned. One solution is to state the dependencies in terms of interfaces, and organize the interfaces in the lower layer.

In general, dependencies should not skip layers, unless the dependent behavior is common across all layers, and the alternative is to simply pass-through operation invocations across layers.

Packages should not depend on subsystems, only on other packages or on interfaces.

A subsystem is a model element which has the semantics of a package, such that it can contain other model elements, and a class, such that it has behavior. A subsystem realizes one or more interfaces, which define the behavior it can perform.

A subsystem may be represented as a UML package (i.e., a tabbed folder) with the «subsystem» stereotype.

An interface is a model element which defines a set of behaviors (a set of operations) offered by a classifier model element (specifically, a class, subsystem or component). The relationship between interfaces and classifiers (subsystems) is not always one-to-one. An interface may be realized by multiple classifiers and a classifier may realize multiple interfaces.

Realization is a semantic relationship between two classifiers. One classifier serves as the contract that the other classifier agrees to carry out.

The realization relationship may be modeled as a dashed line with a hollow arrow head pointing at the contract classifier (canonical form), or when combined with an interface, as a “lollipop” (elided form). Thus, in the above example, the two interface/subsystem pairs with the relation between them are synonymous.

Interfaces are a natural evolution from the public classes of a package (described on the previous slide) to abstractions outside the subsystem. Interfaces are pulled out of the subsystem like a kind of antenna, through which the subsystem can receive signals. All classes inside the subsystem are then private and not accessible from the outside.

A subsystem encapsulates its implementation behind one (or more) interfaces. Interfaces isolate the rest of the architecture from the details of the implementation. Operations defined for the interface are implemented by one or more elements contained within the subsystem.
An interface is a pure specification. Interfaces provide the “family of behavior” that a classifier that implements the interface must support. Interfaces are separate things, that have separate life spans from the elements that realize them. This separation of interface and implementation exemplifies the OO concepts of modularity and encapsulation, as well as polymorphism.
Note: Interfaces are not abstract classes. Abstract classes allow you to provide default behavior for some or all of their methods. Interfaces provide no default behavior.
As mentioned earlier, an interface may be realized by one or more subsystems. Any two subsystems which realize the same interfaces can be substituted for one another. The benefit of this is that, unlike a package, the contents and internal behaviors of a subsystem can change with complete freedom so long as the subsystem's interfaces remain constant.
In the above example, InterfaceK defines the operations X() and Y(). Both SubsystemA and SubsystemB realize InterfaceK, which means that they provide the implementation for operations X() and Y(). Thus, SubsystemA and SubsystemB are completely “plug-and-playable” (i.e., one can be replaced by the other without any impacts on clients of the subsystems).

Subsystems and packages are very alike, but are different in some essential ways. A subsystem provides interfaces by which the behavior it contains can be accessed. Packages provide no behavior; they are simply containers of things which have behavior. Packages help organize and control sets of classes that are needed in common, but which aren’t really subsystems. Packages are just used for model organization and configuration management.
Subsystems completely encapsulate their contents, providing behavior only through their interfaces. Dependencies on a subsystem are on it’s interface(s), not on specific subsystem contents. With packages, dependencies are on specific elements within the package.
With subsystems, the contents and internal behaviors of a subsystem can change with complete freedom as long as the subsystem's interfaces remain constant. With packages, it is impossible to substitute packages for one another unless they have the same public classes. The public classes and their public operations get frozen by the dependencies external classes have on them. Thus, the designer is not free to eliminate these classes or change their behaviors if a better idea presents itself.
Note: Even when using packages, it is important that you hide the implementation from elements external to the package. All dependencies on a package should be on the public classes of the package. Public classes can be considered the interface of the package and should be managed as such (stabilized early).

Look for existing subsystems or components which offer similar interfaces. Compare each identified interface to the interfaces provided by existing subsystems or components. There usually will not be an exact match, but approximate matches can be found. Look first for similar behavior and returned values, then consider parameters.

Modify the newly identified interfaces to improve the fit. There may be opportunities to make minor changes to a candidate interface which will improve its conformance the existing interface. Simple changes include rearranging or adding parameters to the candidate interface. Next, factoring the interface by splitting it into several interfaces, one or more of which match those of the existing component, with the "new" behaviors located in a separate interface.

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. Look at existing components and the set of candidate subsystems. Factor the subsystems so that existing components are used wherever possible to satisfy the required behavior of the system. Where a candidate subsystem can be realized by an existing component, create traceability between the subsystem and the component. Note in the description for this subsystem that the behavior is satisfied by the associated existing component. In mapping subsystems onto components, consider the design mechanisms associated with the subsystem; performance or security requirements may disqualify a component from reuse despite an otherwise perfect match between operation signatures.

Layering provides a logical partitioning of packages into layers with certain rules concerning the relationships between layers. Restricting inter-layer and inter-package dependencies makes the system more loosely coupled and easier to maintain. Failure to restrict dependencies causes architectural degradation, and makes the system brittle and difficult to maintain.

Visibility: Elements should only depend on elements in the same layer and the next lower layer. Exceptions include cases where packages need direct access to lower layer services (e.g., primitive services needed throughout the system, such as printing, sending messages, etc.) There is little value in restricting messages to lower layers if the solution is to effectively implement call pass-throughs in the intermediate layers.

Volatility: In the highest layers, put elements which vary when user requirements change. In the lowest layers, put elements that vary when the implementation platform (hardware, language, operating system, database, etc.) changes. Sandwiched in the middle, put elements which are generally applicable across wide ranges of systems and implementation environments. Add layers when additional partitions within these broad categories help to organize the model.

Generality: Abstract model elements tend to be placed lower in the model, where they can be reused. If not implementation-specific, they tend to gravitate toward to middle layers.
Number of Layers: For a small system, 3 layers are sufficient. For a complex system, 5-7 layers are usually sufficient. For any degree of complexity, more than 10 layers should be viewed with suspicion that increases with the number of layers.

STEP 5: Describe the Run-time Architecture:
is performed by the Architect, once per iteration.

Purpose:

Analyze concurrency requirements
Identify processes
Identify inter-process communication mechanisms
Allocate inter-process coordination resources
Identify process lifecycles
Distribute model elements among processes.
Input Artifacts
Supplementary Specifications
Design Model
Resulting Artifacts
Process View of the Software Architecture Document

Concurrency is the tendency for things to happen at the same time in a system. Concurrency is a natural phenomenon, of course. In the real world, at any given time many things are happening simultaneously. When we design software to monitor and control real-world systems, we must deal with this natural concurrency.
When dealing with concurrency issues in software systems, there are generally two aspects, which are important:
Being able to detect and respond to external events occurring in a random order
Ensuring that these events are responded to in some minimum required interval.
If each concurrent activity evolved independently, in a truly parallel fashion, this would be relatively simple: we could just create separate programs to deal with each activity. The challenges of designing concurrent systems arise mostly because of the interactions, which happen, between concurrent activities. When concurrent activities interact, some sort of coordination is required.

Vehicular traffic provides a useful analogy. Parallel traffic streams on different roadways having little interaction cause few problems. Parallel streams in adjacent lanes require some coordination for safe interaction, but a much more severe type of interaction occurs at an intersection, where careful coordination is required.

Some of the driving forces for concurrency are external. That is, they are imposed by the demands of the environment. In real-world systems many things are happening simultaneously and must be addressed “in real-time” by software. To do so, many real-time software systems must be “reactive.” They must respond to externally generated events which may occur at somewhat random times, in some-what random order, or both.
Designing a conventional procedural program to deal with these situations is extremely complex. It can be much simpler to partition the system into concurrent software elements to deal with each of these events. The key phrase here is “can be”, since complexity is also affected by the degree of interaction between the events.
There can also be internally inspired reasons for concurrency. Performing tasks in parallel can substantially speed up the computational work of a system if multiple CPUs are available. Even within a single processor, multitasking can dramatically speed things up by preventing one activity from blocking another while waiting for I/O. A common situation where this occurs is during the startup of a system. There are often many components, each of which requires time to be made ready for operation. Performing these operations sequentially can be painfully slow.
Controllability of the system can also be enhanced by concurrency. For example, one function can be started, stopped, or otherwise influenced in mid-stream by other concurrent functions—something extremely difficult to accomplish without concurrent components.

Concurrency requirements define the extent to which parallel execution of tasks is required for the system. These requirements help shape the architecture.
A system whose behavior must be distributed across processors or nodes virtually requires a multi-process architecture. A system, which uses some sort of Database Management System or Transaction Manager, also must consider the processes, which those major subsystems introduce.
If dedicated processors are available to handle events, a multi-process architecture is probably best. On the other hand, to ensure events are handled, a uni-process architecture may be needed to circumvent the ‘fairness’ resource sharing algorithm of the operating system: it may be necessary for the application to monopolize resources by creating a single large process, using threads to control execution within the process.
In order to provide good response times, it may be necessary to place computationally intensive activities in a process or thread of their own so that the system is still able to respond to user

inputs while computation takes place, albeit with fewer resources. If the operating system or environment does not support threads (lightweight processes) there is little point in considering their impact on the system architecture.
The above are mutually exclusive and may conflict with one another. Ranking requirements in terms of importance will help resolve the conflict.

The above concurrency requirements were documented in the Course Registration System Supplemental Specification (see the Course Registration Requirements Document).
The first requirement is typical of any system, but the multi-tier aspects of our planned architecture will require some extra thought for this.
The second requirement demonstrates the need for a shared, independent process managing access to the course offerings.
The third issue will lead us to use some sort of mid-tier caching or preemptive retrieval strategy.

Process: A unique address space and execution environment in which instances of classes and subsystems reside and run. The execution environment may be divided into one or more threads of control.
Thread: An independent computation executing within the execution environment and address space defined by an enclosing process.
From “UML Toolkit” by Hans-Erik Eriksson and Magnus Penker :
The difference between process and thread has to do with the memory space in which they execute:
A process executes in its own memory space and encapsulates and protects its internal structure. A process can be viewed as being a system of its own. It is initiated by an executable program. A process may contain multiple threads (i.e., a number of processes may execute within a single process, sharing the same memory space).
A thread executes in a memory space that it may share with other threads.

For each separate flow of control needed by the system, create a process or a thread (lightweight process). A thread should be used in cases where there is a need for nested flow of control (i.e. within a process, there is a need for independent flow of control at the sub-task level).
For example, we can say (not necessarily in order of importance) that separate threads of control may be needed to:
Use of multiple CPUs. There may be multiple CPUs in a node or multiple nodes in a distributed system

Increased CPU utilization. Processes can be used to increase CPU utilization by allocating cycles to other activities when a thread of control is suspended

Fast reaction to external stimuli

Service time-related events. Examples: timeouts, scheduled activities, periodic activities

Prioritize activities. Separate processes allows functionality in different processes to be prioritized individually

Scalability. Load sharing across several processes and processors

Separation of concerns. Separating concerns between different areas of the software, e.g., safety

Availability. Redundant processes. You can achieve a higher system availability by having backup processes

Support major subsystems. Some major subsystems may required separate processes (e.g., the DBMS, Transaction Manager, etc.)

STEP 6: Describe Distribution:

The Describe Distribution activity is where the processes defined in the 'Describe the Run-time Architecture' activity are allocated to actual physical nodes. Just identifying the individual processes is not enough, the relationship between the processes and the hardware must be described.

The focus of the Describe Distribution activity is on developing the Deployment View of the architecture. This activity is really only required for distributed systems.

In this module, we will describe WHAT is performed in Describe Distribution, but will not describe HOW to do it. Such a discussion is the purpose of an architecture course, which this course is not.

The goal of this module is to give the student an understanding of how to model the Deployment Model using the UML.

Understanding the rationale and considerations that support the architectural decisions is needed in order to understand the architecture, which is the framework in which designs must be developed.

The Architect performs the 'Describe Distribution' activity.

Purpose

To describe how the functionality of the system is distributed across physical nodes. Required only for distributed systems.

Input Artifacts

Process View of the Software Architecture Document

Implementation Model

Resulting Artifacts

Deployment View of the Software Architecture Document

Note: The implementation model is a collection of components, and the implementation subsystems that contain them. Components include both deliverable components, such as executables, and components from which the deliverables are produced, such as source code files. The Implementation Model is developed in the Implementation workflow which is out of the scope of the analysis and design course, which focuses on the Analysis and Design workflow.

Before we discuss some common distribution patterns, let's look at some of the reasons why you would distribute an application in the first place. There are many cases where the system workload cannot be handled by one processor. This can be due to special processing requirements, as in the case of digital signal processing, which may require specialized and dedicated processors. The need for distribution may also result from inherent scaling concerns, where the large number of concurrent users are simply too many to support on any single processor, or it may result from economic concerns, where the price performance of smaller, cheaper processors cannot be matched in larger models. For some systems, the ability to access the system from distributed locations may mean that the system itself must be distributed.
The distribution of processes across two or more nodes requires a closer examination of the patterns of inter-process communication in the system. Often, there is a naïve perception that distribution of processing can 'off-load' work from one machine onto a second. In practice, the additional inter-process communication workload can easily negate any gains made from workload distribution if the process and node boundaries are not considered carefully. Achieving real benefits to distribution requires work and careful planning.

There are a number of typical patterns of distribution in systems, depending on the functionality of the system and the type of application. Typically, the distribution patterns are described as 'the architecture' of the system, though the full architecture encompasses this, but also many more things.
In client/server architectures, there are specialized network processor nodes called clients, and nodes called servers. Clients are consumers of services provided by servers.
The distribution of the system functionality to the clients and servers distinguishes the different types or styles of client/server architectures.
3-tier: Functionality is equally divided into three logical partitions: application, business, and data services.
Fat client: More functionality is placed on the client. Examples: database and file servers

Fat server: More functionality is placed on the server. Examples: Groupware, transaction and Web servers

Distributed client/server: The application, business and data services reside on different nodes, potentially with specialization of servers in the business services and data services tiers. A full realization of a 3-tier architecture.

In a Peer-to-Peer architecture, any process or node in the system may be both client and server.

Each of these architectural patterns is described briefly in this module.

Client/server is a conceptual way of breaking up the application into service requestors (clients) and service providers (servers).

A client often services a single user and often handles end-user presentation services (GUI's). A system can consist of several different types of clients, examples of which include user workstations and network computers.

The server usually provides services to several clients simultaneously. The services provided are typically database, security or print services. A system can consist of several different types of servers. For example: database servers, handling database machines such as Sybase, Ingres, Oracle, Informix; print servers, handling the driver logic (queuing etc.) for a specific printer; communication servers (TCP/IP, ISDN, X.25); window manager servers (X); file servers (NFS under UNIX).

The application/business logic is distributed between both the client and the server (application partitioning).

In the above example, Client A is an example of a 2-tier architecture, with most application logic located in the server. Client B is a typical 3-tier architecture, with Business Services implemented in a Business Object Server. Client C is a typical web-based application. We'll look at each of these distribution strategies in more detail.

In the peer-to-peer architecture, any process or node in the system may be both client and server. Distribution of functionality is achieved by grouping inter-related services together to minimize network traffic while maximizing throughput and system utilization. Such systems tend to be complex, and there is a greater need to be aware of issues such as deadlock, starvation between processes, and fault handling.

The topology of the network and the capabilities and characteristics of the processors and devices on the network will determine the nature and degree of distribution possible in the system.
The following information needs to be captured:
The physical layout of the network, including locations
The nodes on the network, their configurations and capabilities. The configuration includes both the hardware and the software installed on the nodes, the number of processors, the amount of disk space, the amount of memory, the amount of swap, etc. Hardware installed on the node can be represented using 'devices'.
The bandwidth of each segment on the network.
The existence of any redundant pathways on the network (This will aid in providing fault tolerance capabilities.)
The primary purpose of the node. This includes:
workstation nodes used by end users
server nodes on which "headless" processing occurs
special configurations used for development and test.
Other specialized processors
IP design and facilities (for example, DNS, VPN), if IP network exists
The role of the Internet in the solution.

The essential elements of a deployment model are nodes and their connections. A node is a run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well. Run-time objects and components may reside on nodes.
Processors and devices are common stereotypes of node. The distinction between the two may seem difficult to assess, as many devices now contain their own CPUs. However, the distinction between processors and devices lies in the type of software that executes on them. Processors execute programs/software that were explicitly written for the system being developed. Devices execute software written that controls the functionality of the device itself. Devices are typically attached to a processor that controls the system's use of the device.
Connections can be drawn between nodes (i.e., between processors and between processors and devices). These connections represent communication mechanisms and can be described by physical medium (e.g., ethernet, fiber optic cable) or software protocol (e.g., TCP/IP). A stereotype may be used to specify the type of connection.

Processes must be assigned to a hardware device for execution to distribute the workload of the system.

If the chosen architecture implies or requires a specific distribution pattern, this should be realized. Distribution patterns were discussed earlier in this module.

Those processes with fast response time requirements should be assigned to the fastest processors.

Processes should be allocated to nodes so as to minimize the amount of cross-network traffic. Network traffic, in most cases, is quite expensive. It is an order of magnitude or two slower than inter-process communication. Processes which interact to a great degree should be co-located on the same node. Processes, which interact less frequently, can reside on different nodes. The crucial decision, and one that sometimes requires iteration, is where to draw the line.

Additional considerations:

Node capacity (in terms of memory and processing power)

Communication medium bandwidth (bus, LANs, WANs)

Availability of hardware and communication links, rerouting requirements for redundancy and fault-tolerance

Deployment diagrams allow you to capture the topology of the system nodes, including the assignment of run-time elements to nodes. This allows you to visually see potential bottlenecks.

As discussed earlier, a Deployment diagram contains nodes connected by associations. The associations indicate a communication path between the nodes.

Nodes may contain run-time component instances and/or objects. This indicates that the component lives on or runs on the node. Only run-time entities reside on Deployment diagrams. Those entities that have been “compiled away” are not shown. An example of a run-time object is a process.

The above diagram once again illustrates the Deployment View for the Course Registration System. It has been updated to include the processes, which execute on the nodes. The processes that are listed are the processes that were defined in the Describe the Run-time Architecture module.

Note: No threads are shown in the above diagram because threads always run in the context of a process.

STEP 7: Use Case Design

Use-Case Design is performed by the Designer, once per use-case.

Purpose

To refine use-case realizations in terms of design element interactions

To refine requirements on the operations of design classes

To refine requirements on the operations of subsystems and/or their interfaces

Input Artifacts

Supplementary Specifications

use-cases

Use-Case Realizations

Design Classes

Design Subsystems

Interfaces

Resulting Artifacts

Use-Case Realizations, described with class diagrams, interaction diagrams, and flow of events refined.

The above are the major steps of the Use-Case Design activity.

Use-case realizations evolve from interactions amongst analysis classes to interactions amongst design elements (e.g., design classes and subsystems).

The purpose of Use-Case Design is to make sure these are consistent and make sense. During Use-Case Design, the focus is on the use-case, and this includes crossing subsystem boundaries. The use-case designer needs to make sure the use-case is implemented completely and consistently across the subsystems.

Anything done from the use-case perspective is part of Use-Case Design.

We will address each of these steps in this module.

Before moving forward in Use-Case Design it is important that we revisit what happened to the analysis classes that were identified in Use-Case Analysis. These analysis classes were mapped to design elements that were identified by the architect during the Identify Design Elements activity. We will be incorporating these design elements into our models in Use-Case Design.

Identify Design Elements is where the analysis classes identified during Use-Case Analysis are refined into design elements (e.g., classes or subsystems). Analysis classes handle primarily functional requirements and model objects from the "problem" domain; design elements handle non-functional requirements, and model objects from the "solution" domain.

It is in Identify Design Elements that we decide what analysis "classes" are really classes, which are subsystems (which must be further decomposed), and which are existing components and don’t need to be “designed” at all.

Once the design classes and subsystems have been created, each must be given a name and a short description. The responsibilities of the original analysis classes should be transferred to the newly created subsystems. In addition, the identified design mechanisms should be linked to design elements.

Each use-case realization should be refined to describe the interactions between participating design objects as follows:
Identify each object that participates in the use-case flow of events. These objects may be instances of design classes and subsystems, or they may be instances of actors that the participating objects interact with.
Represent each participating object in an interaction diagram. Subsystems can be represented by instances of the subsystem's interface(s).
Illustrate the message sending between objects by creating messages (arrows) between the objects. The name of a message should be the name of the operation invoked by the message. For messages going to design classes, the operation is a class operation. For messages going to subsystems, the operation is an interface operation.
Describe what an object does when it receives a message. This is done by attaching a script or note to the corresponding message. When the person responsible for an object's class assigns and defines its operations, these notes/scripts will provide a basis for that work.
For each use-case realization, illustrate the class relationships that support the collaborations modeled in the interaction diagrams by creating one or more class diagrams.

Look at the interaction diagrams.
For each class that has been refined into a subsystem, replace the class with the associated subsystem interface. Any interactions that describe HOW the subsystem should implement the service should be deferred until subsystem design.
Incrementally incorporate any applicable architectural mechanisms, using the patterns of behavior defined for the mechanisms during the architectural activities. This may include the introduction of new design elements and messages.
Any updates need to be reflected in both the static and dynamic parts of the use-case realization (i.e., the interaction diagrams and the VOPC class diagram).

A use-case realization can be described, if necessary, at several levels in the subsystem hierarchy.

In the above example, the lifelines in the middle diagram represent subsystems; the interactions in the circles represent the internal interaction of subsystem members in response to the message.
This approach raises the level of abstraction of the use-case realization flows of events.
The advantages of this approach are described on subsequent slides.

Encapsulate a sub flow within a subsystem when it:
Occurs repeatedly in different use-case realizations: Similar messages are sent to similar objects, providing the same end result. The phrase 'similar' is used because some design work may be needed to make the behavior reusable.
Occurs in only one use-case realization, but it is expected to be performed repeatedly in future iterations, or in similar systems in the future. The behavior may make a good reusable component.
The sub-sequence occurs in only one use-case realization, but is complex and easily encapsulated, needs to be the responsibility of one person or a team, and provides a well-defined result. In these kinds of situations, the complex behavior usually requires special technical knowledge, or special domain knowledge, and as a result is well suited to encapsulation within a subsystem.
Is encapsulated within a component in the Implementation Model. In this case, a subsystem is the appropriate representation for the component within the Design Model.
Make sure that what you are abstracting is worth abstracting.

To achieve true substitutability of subsystems that realize the same interface, only their interfaces should be visible in the interaction diagrams; otherwise all diagrams would need to be changed whenever a subsystem is substituted for another.
On an interaction diagram, sending a message to an interface lifeline means that any subsystem that realizes the interface can be substituted for the interface in the diagram.
In many cases, the interface lifeline does not have messages going out from it, since different subsystems realizing the interface may send different messages. However, if you want to describe what messages should be sent (or are allowed to be sent) from any subsystem realizing the interface, such messages can originate from the interface lifeline.
With this approach, when describing the interactions, the focus remains on the services, not on how the services are implemented within the design elements. This is known as “Design by Contract” and is one of the core tenets of robust software development using abstraction and encapsulation mechanisms.

Describing how the services are implemented is the focus of Subsystem Design (for the design subsystems) and Class Design (for the design classes).

The advantages of encapsulating subsystem interactions over modeling the entire system at once are:
Use-case realizations become less cluttered, especially if the internal design of some subsystems are complex.
Use-case realizations can be created before the internal designs of subsystems are created. This can be used to make sure that use-case functionality has not been “lost” between the allocation of use-case responsibility in Use-Case Analysis and the identification of design elements (i.e., subsystems and design classes) in Identify Design Elements, and before Subsystem Design is performed.
Use-case realizations become more generic and are easier to change, especially if a subsystem needs to be substituted for another subsystem.
Again, encapsulating subsystem interactions raises the level of abstraction of the use-case realization flows of events.

In some cases it can be appropriate to develop a subsystem more or less independently and in parallel with the development of other subsystems. To achieve this, we must first find subsystem dependencies by identifying the interfaces between them.
The work can be done as follows:
Concentrate on the requirements that affect the interfaces between the subsystems.
Make outlines of the required interfaces, showing the messages that are going to pass over the subsystem borders.
Draw interaction diagrams in terms of subsystem interfaces for each use-case.
Refine the interfaces needed to provide messages.
Develop each subsystem in parallel using the interfaces as synchronization instruments between development teams.

You can also choose whether to arrange the interaction diagrams in term of subsystems or in terms of their interfaces only. In some projects, it might even be necessary to implement the classes providing the interfaces before you continue with the rest of the modeling.
The detailed design of the subsystem “internals” is done during Subsystem Design. The interfaces are what ensure compatibility between the Use-Case Design and the Subsystem Design.

In practice, there may be times when the application needs to control various aspects of persistence:
How transactions are managed
When persistent objects are written. There are two cases to be concerned with: initial time the object is written to the persistent object store, and subsequent times when the object is updated.
When persistent objects are read. Retrieval of objects from the persistent object store is necessary before the application can send messages to that object. You need to send a message to an object that knows how to query the database, retrieve the correct objects, and instantiate them.
When persistent objects are deleted. Unlike transient objects, which simply disappear when the process that created them dies, persistent objects exist until they are explicitly deleted. So, it's important to delete the object when it's no longer being used. However, this is hard to determine. Just because one application is done with an object does not mean that all applications, present and future, are done. And, because objects can and do have associations that even they don't know about, it's not always easy to figure out if it's OK to delete an object. The persistence framework may also provide support for this
We will look at each one of these in subsequent slides.

In those cases where the flow of events is not fully clear from just examining the messages sent between participating objects, you may need to add additional descriptions to the interaction diagrams.
These steps are taken in cases where timing annotations, notes on conditional behavior, or clarification of operation behavior is needed to make it easier for external observers to read the diagrams.
Often, the name of the operation is not sufficient to understand why the operation is being performed. Textual notes or scripts in the margin of the diagram may be needed to clarify the interaction diagram. Textual notes and scripts may also be needed to represent control flow such as decision steps, looping, and branching. In addition, textual tags may be needed to correlate extension points in the use-case with specific locations in interaction diagrams.

The Design Model as a whole must be reviewed to detect glaring problems with layering and responsibility partitioning. The purpose of reviewing the model as a whole is to detect large-scale problems that a more detailed review would miss.
We want to ensure that the overall structure for the Design Model is well formed, as well as detect large-scale quality problems that may not visible by looking at lower-level elements.
The above checkpoints are important, as new packages/subsystems may be created when common sub-flows are identified.

5 packages and 1,000 classes is probably a sign of that something is wrong.

Once the structure of the Design Model is reviewed, the behavior of the model needs to be reviewed. First, make sure that there is no missing behavior by checking to see that all scenarios for the current iteration have been completely covered by use-case realizations. All of the behavior in the relevant use-case sub-flows must be described in the completed use-case realizations.

Next, make sure the behavior of the use-case realization is correctly distributed between model elements in the realizations. Make sure the operations are used correctly, that all parameters are passed, and that return values are of the correct type.

We want to ensure that the behavior of the system (as expressed in use-case realizations) matches the required behavior of the system (as expressed in use-cases). Is it complete? We also want to ensure that the behavior is allocated appropriately among model elements, i.e. is it correct?

There needs to be participating objects to perform all the behavior of the corresponding use-case. You need to make sure that it is clear from the flow of events description how the diagrams are related to each other.

STEP 8: Subsystem Design

Subsystem Design is where the detailed collaborations of classes that are needed to implement the responsibilities documented in the subsystem interfaces are fleshed out/designed. In order to support these collaborations, additional relationships may need to be defined between subsystems.

Subsystem Design is performed once per Design Subsystem.

Purpose

To define the behaviors specified in the subsystem's interfaces in terms of collaborations of contained classes

To document the internal structure of the subsystem

To define realizations between the subsystem's interfaces and contained classes

To determine the dependencies upon other subsystems

Input Artifacts

Use-Case Realizations

Design Subsystem with Interface Definitions

Design Guidelines

Contains detailed usage information for the architectural mechanisms

Resulting Artifacts

Use-Case Realizations (updated)

Detailed design of the subsystem may result in changes to the use-case realization.

Design Subsystem with Interface Definitions (updated)

Detailed design of the subsystem may result in changes to the subsystem and interface definitions.

Design Classes

Each subsystem should be as independent as possible from other parts of the system. It should be possible to evolve different parts of the system independently from other parts. This minimizes the impact of changes and eases maintenance efforts.

It should be possible to replace any part of the system with a new part, provided the new part supports the same interfaces. In order to ensure that subsystems are replaceable elements in the model, the following are necessary:

A subsystem should not expose any of its contents. No element contained by a subsystem should have 'public' visibility. No element outside the subsystem should depend on the existence of a particular element inside the subsystem.

A subsystem should only depend on the interfaces of other model elements, so that it is not directly dependent on any specific model elements outside the subsystem. The exceptions are cases where a number of subsystems share a set of class definitions in common, in which case those subsystems 'import' the contents of the packages which contain the common classes. This should only be done with packages in lower layers in the architecture, and only to ensure that common definitions of classes that must pass between subsystems are consistently defined.

All dependencies on a subsystem should be dependencies on the subsystem interfaces. Clients of a subsystem are dependent on the subsystem interface(s), not on elements within the subsystem. In that way, the subsystem can be replaced by any other subsystem that realizes the same interfaces.

The above are the major steps involved in the Subsystem Design activity.

We first must take the responsibilities allocated to the subsystems and further allocate those responsibilities to the subsystem elements.

Once the subsystem elements have been identified, the internal structure of the subsystems (a.k.a. subsystem element relationships) needs to be documented.

Once we know how the subsystem will implement its responsibilities, we need to document the interfaces upon which the subsystem is dependent.

Finally, we'll discuss the kinds of things you should look for when reviewing the results of Subsystem Design.

The external behaviors of the subsystem are defined by the interfaces it realizes. When a subsystem realizes an interface, it makes a commitment to support each and every operation defined by the interface.
The operation may be in turn realized by:
An operation on a class contained by the subsystem; this operation may require collaboration with other classes or subsystems
An operation on an interface realized by a contained subsystem
The collaborations of model elements within the subsystem should be documented using sequence diagrams, which show how the subsystem behavior is realized. Each operation on an interface realized by the subsystem should have one or more documenting sequence diagrams. This diagram is owned by the subsystem, and is used to design the internal behavior of the subsystem.

For each interface operation, identify the classes (or where the required behavior is complex, a contained subsystem) within the current subsystem, which are needed to perform the operation. Create new classes/subsystems where existing classes/subsystems cannot provide the required behavior (but try to reuse first).
Creation of new classes and subsystems should force reconsideration of subsystem content and boundary. Be careful to avoid having effectively the same class in two different subsystems. Existence of such a class implies that the subsystem boundaries may not be well drawn. Periodically revisit Identify Design Elements to re-balance subsystem responsibilities.
The collaborations of model elements within the subsystem should be documented using interaction diagrams, which show how the subsystem behavior is realized. Each operation on an interface realized by the subsystem should have one or more documenting interaction diagrams. This "internal" interaction diagram shows exactly what classes provide the interface, what needs to happen internally to provide the subsystem's functionality, and which classes send messages out from the subsystem. These diagrams are owned by the subsystem, and are used to design the internal behavior of the subsystem. The diagrams are essential for subsystems with complex internal designs. It also enables the subsystem behavior to be easily understood, hopefully rendering it reusable across contexts.
These internal interaction diagrams should incorporate any applicable mechanisms initially identified in Identify Design Mechanisms (e.g., persistence, distribution, etc.)

When a subsystem element uses some behavior of an element contained by another subsystem or package, a dependency on the external element is needed.
If the element on which the subsystem is dependent is within a subsystem, the dependency should be on the subsystem interface, not on the subsystem itself or on any element in the subsystem. This allows the design elements to be substituted for one another as long as they offer the same behavior. It also gives the designer total freedom in designing the internal behavior of the subsystem, as long as it provides the correct external behavior. If a model element directly references a model element in another subsystem, the designer is no longer free to remove that model element or redistribute the behavior of that model element to other elements. As a result, the system is more brittle.
If the element the subsystem element is dependent on is within a package, the dependency should be on the package itself. Ideally, a subsystem should only depend on the interfaces of other model elements for the reasons stated above. The exception is where a number of subsystems share a set of common class definitions, in which case those subsystems 'import' the contents of the packages containing the common classes. This should only be done with packages in lower layers in the architecture to ensure that common class definitions are defined consistently. The disadvantage is that the subsystem cannot be reused independent of the depended-on package.

The above checklist includes the key things to look for when assessing the results of Subsystem Design.
A Designer is responsible for the integrity of the design subsystem, ensuring that:
The subsystem encapsulates its contents; only exposing contained behavior through interfaces it realizes.
The operations of the interfaces the Subsystem realizes are distributed to contained classes or subsystems.
The subsystem properly implements its interfaces.

STEP 9: Class Design

In Class Design, the focus is on fleshing out the details of a particular class (e.g., what operations and classes need to be added to support, and how do they collaborate to support, the responsibilities allocated to the class).

Purpose:

The purpose of Class Design is to ensure that the classes provide the behavior the use-case realizations require, to ensure that it is straightforward to implement the classes, to handle non-functional requirements related to classes, and to incorporate the design mechanisms used by the classes.
Input Artifacts:
Supplementary Specifications
Design Guidelines
Design classes
Use Cases
Use-Case Realizations
Design Model
Resulting Artifacts:
Design Classes
The detailed steps performed during this activity are summarized on the next page, and detailed in the remainder of this module.

When performing Class Design, you need to consider:
How the classes that were identified in analysis as boundary, control, and entity classes will be realized in the implementation
How design patterns can be used to help solve implementation issues
How the architectural mechanisms will be realized in terms of the defined design classes.
There are specific strategies that can be used to design a class, depending on its original analysis stereotype (boundary, control and entity). These stereotypes are most useful during Use-Case Analysis when identifying classes and allocating responsibility. At this point in design, you really no longer need to make the distinction -- the purpose of the distinction was to get you to think about the roles objects play, and making sure that you separate behavior according to the forces, which cause objects to change. Once you have considered these forces and have good class decomposition, the distinction is no longer really useful.
Here the class is refined to incorporate the architectural mechanisms.
When you identify a new class, make an initial pass at its relationships. This is just the initial association that ties the new classes into the existing class structure. These will be refined throughout Class Design.

These are some things you should think about as you define additional classes.
The proper size of a class will depend heavily on the implementation environment. Classes should map directly to some phenomenon in the implementation language in such a way that the mapping results in good code.

With that said, you should design as if you had classes and encapsulation even if your implementation language does not support it. This will help keep the structure easy to understand and modify.

During analysis, high-level boundary classes are identified. During design, the design must be completed. You may need to create additional classes to support actual GUI and external system interactions.

Some mechanisms used to implement the UI can be architecturally significant. These should be identified and described by the architect in the Identify Design Mechanisms activity, and applied by the designer here in the Class Design activity. Design of user interface boundary classes will depend on the user interface development tools being used on the project. You only need design what the development environment will not automatically create for you. All GUI builders are different. Some create objects containing the information from the window. Others create data structures with the information. Any prototyping of the user interface done earlier is a good starting point for this phase of design.

Since interfaces to external systems usually have complex internal behavior, they are typically modeled as subsystems. If the interface is less complex, such as a pass through to an existing API, you may represent it with one or more design classes. In the latter case, use a single design class per protocol, interface, or API.

During analysis, entity classes may have been identified and associated with the analysis mechanism for persistence, representing manipulated units of information. Performance considerations may force some re-factoring of persistent classes, causing changes to the Design Model, which are discussed jointly between the Database Designer and the Designer responsible for the class. The details of a database-based persistence mechanism are designed during Database Design, which is out of the scope of this course.

Here we have a persistent class with five attributes. One attribute is not really persistent; it is used at runtime for bookkeeping. From examining the use-cases, we know that two of the attributes are used frequently. Two other attributes are used less frequently. During design, we decide that we'd like to retrieve the commonly used attributes right away, but only retrieve the rarely used ones if we some client asks for them. We don't want to make a complex design for the client, so, from a data standpoint; we will consider the Fat Class to be a proxy in front of two real persistent data classes. It will retrieve the FatClassDataHelper from the database when it is first retrieved. It will only retrieve the FatClassLazyDataHelper from the database in the rare occasion that a client asks for one of the rarely used attributes.

Such behind-the-scenes implementation is an important part of tuning the system from a data-oriented perspective while retaining a logical object-oriented view for clients to use.

If control classes seem to be just “pass throughs” from the boundary classes to the entity classes, they may be eliminated.
Control classes may become true design classes for any of the following reasons:
They encapsulate significant control flow behavior
The behavior they encapsulate is likely to change
The behavior must be distributed across multiple processes and/or processors
The behavior they encapsulate requires some transaction management.
We saw in Describe Distribution how what was a single control classes in analysis, became two classes in design (a proxy and a remote). For our example, the control classes were needed in design.

To identify operations on design classes:
Study the responsibilities of each corresponding analysis class, creating an operation for each responsibility. Use the description of the responsibility as the initial description of the operation.
Study the use-case realizations in the class participations to see how the operations are used by the use-case realizations. Extend the operations, one use-case realization at a time, refining the operations, their descriptions, return types and parameters. Each use-case realization's requirements, as regards classes, are textually described in the Flow of Events of the use-case realization.
Study the use-case Special Requirements, to make sure that you do not miss implicit requirements on the operation that might be stated there.
Use-case realizations cannot provide enough information to identify all operations. To find the remaining operations, consider the following:
Is there a way to initialize a new instance of the class, including connecting it to instances of other classes to which it is associated?
Is there a need to test to see if two instances of the class are equivalent?
Is there a need to create a copy of a class instance?
Are any operations required on the class by mechanisms, which they use? (Example: a “garbage collection” mechanism may require that an object be able to drop all of its references to all other objects in order for unused resources to be freed.)

Operations should be named to indicate their outcome. For example, getBalance() vs. calculateBalance(). One approach for naming operations that get and set properties is to simply name the operation the same name as the property. If there is a parameter, it sets the property; if not, it returns the current value.

You should name operations from the perspective of the client asking for a service to be performed by the class. For example, getBalance() vs. receiveBalance(). The same applies to the operation descriptions. Descriptions should always be written from the operation USER's perspective. What service does the operation provide?
It is best to specify the operations and their parameters using implementation language syntax and semantics. This way the interfaces will already be specified in terms of the implementation language when coding starts.

In addition to the short description of the parameter be sure to include things like:
Whether the parameter should be passed by value or by reference, and if by reference, is the parameter changed by the operation? By-value means that the actual objects is passed. By-reference means that a pointer or reference to the object is passed.
The signature of the operation defines the interface to objects of that class, and the parameters should therefore be designed to promote and define what that interface is. For example, if a parameter should never be changed by the operation, then design it so that it is not possible to change it - if the implementation environment supports that type of design.
Whether parameters may be optional and/or have default values when no value is provided.
Whether the parameter has ranges of valid values.
The fewer parameters you have the better. Fewer parameters help to promote understandability, as well as maintainability, i.e. the more parameters clients need to understand the more tightly coupled the objects are likely to be conceptually.
One of the strengths of OO is that you can manipulate very rich data structures, complete with associated behavior. Rather than pass around individual data fields (e.g., StudentID), strive to pass around the actual object (e.g., Student). Then the recipient has access to all the properties and behavior of that object.

Operation visibility is the realization of the key object-orientation principle of encapsulation.
Public members are accessible directly by any client.
Protected members are directly accessible only by instances of subclasses.
Private members are directly accessible only by instances of the class to which they are defined.
How do you decide what visibility to use? Look at the interaction diagrams on which the operation is referenced. If the message is from outside of the object, use public. If it is from a subclass, use protected. If it's from itself, use private. You should define the most restrictive visibility possible that will still accomplish the objectives of the class. Client access should be granted explicitly by the class and not taken forcibly.
Visibility applies to attributes as well as operations. Attributes are discussed later in this module.

A method specifies the implementation of an operation. It describes how the operation works, not just what it does.
The method, if described, should discuss:
How operations are to be implemented.
How attributes are to be implemented and used to implement operations.
How relationships are to be implemented and used to implement operations.
The requirements will naturally vary from case to case. However, the method specifications for a class should always state:
What is to be done according to the requirements.
What other objects and operations are to be used.
More specific requirements may concern:
How parameters are to be implemented.
Any special algorithms to be used.
In many cases, where the behavior required by the operation is sufficiently defined by the operation name, description and parameters, the methods are implemented directly in the programming language. Where the implementation of an operation requires use of a specific algorithm, or requires more information than is presented in the operation's description, a separate method description is required.

The state an object resides in is a computational state, and is defined by the stimuli the object can receive and what operations can be performed as a result. An object that can reside in many computational states is state-controlled.
For each class exhibiting state-controlled behavior, describe the relations between an object's states and an object's operations.

A statechart is a tool for describing:
States the object can assume
Events that cause an object to transition from state to state
Significant activities and actions that occur as a result
A statechart is a diagram used to show the life history of a given class, the events that cause a transition from one state to another, and the actions that result from a state change. Statecharts emphasize the event-ordered behavior of a class instance.
The state space of a given class is the enumeration of all the possible states of an object.
A state is a condition in the life of an object. The state of an object determines its response to different events.

An event is a specific occurrence (in time and space) of a stimulus that can trigger a state transition.
A transition is a change from an originating state to a successor state as a result of some stimulus. The successor state could possibly be the originating state. A transition may take place in response to an event, and can be labeled with an event.
A guard condition is a Boolean expression of attribute values which allows a transition only if the condition is true.
An action is an atomic execution that results in a change in state, or the return of a value.
An activity is a non-atomic execution within a state machine.

An activity is:
A non-atomic execution within a state machine
An operation that takes time to complete.
Is interruptible (i.e., an event may be received during the activity, which may result in a state change).
Activities are associated with a state. An activity starts as soon as the state is entered, and can run to completion or, as noted above, can be interrupted by an outgoing transition.
Sometimes, the sole purpose of a state is to perform an activity. An automatic transition occurs when the activity is complete.
An action is:
An atomic execution that results in a change in state, or the return of a value.
An operation that is associated with a transition.
Actions conceptually take an insignificant amount of time to complete, and are considered non-interruptible. Action names are shown on the transition arrow preceded by a slash.
When an action must occur no matter how a state is entered or exited, the action can be associated with the state. In reality, the action is associated with every transition entering or exiting the state. The action is shown inside the state icon preceded by the keyword entry or exit.

The state an object resides in is a computational state. It is defined by the stimuli the object can receive and what operations can be performed as a result. An object that can reside in many computational states is state-controlled.
The complexity of an object depends on:
The number of different states
The number of different events it reacts to
The number of different actions it performs that depend on its state
The degree of interaction with its environment, i.e. other objects

The complexity of conditional, repetitive transitions

Some state transition event can be associated with an operation. Depending on the object's state, the operation may have a different behavior. The transition events describe how this occurs.
The method description for the associated operation should be updated with the state-specific information, indicating what the operation should do for each relevant state.
Operation calls are not the only source of events. In the UML, you can model four different kinds of events:
Signals
Calls
Passing of time
Change in state.
States are often represented using attributes. The state charts serve as input into the attribute identification step.

Attributes the class needs to carry out its operations and the states of the class are identified during the definition of methods. Attributes provide information storage for the class instance and are often used to represent the state of the class instance. Any information the class itself maintains is done through its attributes.
You may need to add additional attributes to support the implementation and establish any new relationships that the attributes require.
Check to make sure all attributes are needed. Attributes should be justified - it is easy for attributes to be added early in the process and survive long after they are no longer needed due to shortsightedness. Extra attributes, multiplied by thousands or millions of instances, can have a large effect on the performance and storage requirements of the system.

In analysis, it was sufficient to specify the attribute name only, unless the representation was a requirement that was to constrain the designer.
During design, for each attribute, define the following:
its name, which should follow the naming conventions of both the implementation language and the project;
its type, which will be an elementary data type supported by the implementation language;
its default or initial value, to which it is initialized when new instances of the class are created
its visibility, which will take one of the following values:
Public: the attribute is visible both inside and outside the package containing the class.
Protected: the attribute is visible only to the class itself, to its subclasses, or to friends of the class (language dependent)

Private: the attribute is only visible to the class itself and to friends of the class.
For persistent classes, also include whether the attribute is persistent (the default) or transient. Even though the class itself may be persistent, not all attributes of the class need to be persistent.
A complete operation signature includes the operation name, operation parameters, and operation return value. This may drive the identification of new classes and relationships.
Operation visibility may be public, private or protected.
A complete attribute signature includes the attribute name, attribute type, and attribute default value (optional). This may drive the identification of new classes and relationships.
Attribute visibility may be public, private or protected.
When adding relationships and/or classes, the package/subsystem interrelationships may need to be refined. Such a refinement can only be approved by the Architecture Team.
The produced statechart should include states, transitions, events, and guard conditions.

After completing a model, it is important to step back and review your work. Some helpful questions are as following:
Is the name of each operation descriptive and understandable? The operation should be named from the perspective of the user, therefore the name of the operation should indicate its outcome.
Does each attribute represent a single conceptual thing? Is the name of each attribute descriptive and correctly convey the information it stores?
Is the state machine understandable? Do the state names and transitions accurately reflect the domain of the system? The state machine should accurately reflect the lifetime of an object.
Does the state machine contain any superfluous states or transitions? States and transitions should reflect the object's attributes, operations, and relationships. Don't read anything extra into the lifetime of the object that cannot be supported by the class structure.

At this point, we have identified which class relationships should be dependencies. Now it's time to design the details of the remaining associations.
Also, additional associations may need to be defined to support the method descriptions defined earlier. Remember, to communicate/send messages between their instances, classes must have relationships with each other.
We will discuss each of the "things to look for" topics on subsequent slides, except for "Association vs. Aggregation" which was discussed in Use-Case Analysis.

Composition is a form of aggregation with strong ownership and coincident lifetimes of the part with the aggregate. The whole "owns" the part and is responsible for the creation and destruction of the part. The part is removed when the whole is removed. The part may be removed (by the whole) before the whole is removed.

A solid filled diamond is attached to the end of an association path (on the “whole side”) to indicate composition.
In some cases, composition can be identified as early as analysis, but more often it is not until design that such decisions can be made confidently. That is why composition is introduced here rather than in Use-Case Analysis.

For composition, the multiplicity of the aggregate end may not exceed one (it is unshared). The aggregation is also unchangeable, that is, once established its links cannot be changed. Parts with multiplicity having a lower bound of 0 can be created after the aggregate itself, but once created, they live and die with it. Such parts can also be explicitly removed before the death of the aggregate.
Non-shared aggregation does not necessarily imply composition.
An aggregation relationship that has a multiplicity greater than one established for the aggregate is called shared, and destroying the aggregate does not necessarily destroy the parts. By implication, a shared aggregation forms a graph, or a tree with many roots.
An example of shared aggregation may be between a University class and a Student class (the University being the “whole” and the Students being the “parts”). With regards to registration, a Student does not make sense outside the context of a University, but a Student may be enrolled in classes in multiple Universities.

The use of association versus aggregation was discussed in the Use-Case Analysis module. Here we discuss the use of “vanilla” aggregation versus composition.
Composition should be used over "plain" aggregation when there is a strong inter-dependency relationship between the aggregate and the parts; where the definition of the aggregate is incomplete without the parts. For example, it does not make sense to have an Order if there is nothing being ordered (i.e., Line Items).
Composition should be used when the whole and part must have coincident lifetimes. Selection of aggregation or composition will determine how object creation and deletion are designed.

A property of a class is something that the class knows about. You can model a class property as a class (with a composition relationship), or as a set of attributes of the class. In other words, you can use composition to model an attribute.
The decision whether to use a class and composition, or a set of attributes, depends on the following:
Do the 'properties' need to have independent identity, such that they can be referenced from a number of objects? If so, use a class and composition.

Do a number of classes need to have the same 'properties'? If so, use a class and composition.
Do the 'properties' have a complex structure, properties, and behavior of their own? If so, use a class (or classes) and composition.
Otherwise, use attributes.
The decision of whether to use attributes or a composition association to a separate class may also be determined based the degree of coupling between the concepts being represented: when the concepts being modeled are tightly connected, use attributes. When the concepts are likely to change independently, use composition.

In the above example, semester is not a property that requires independent identity, nor does it have a complex structure. Thus, it was modeled as an attribute of Schedule.
On the other hand, the relationship from Student to Schedule is modeled as a composition rather than an attribute because Schedule has a complex structure and properties of its own.

The navigability property on a role indicates that it is possible to navigate from a associating class to the target class using the association. This may be implemented in a number of ways: by direct object references, by associative arrays, hash-tables, or any other implementation technique that allows one object to reference another.
Navigability is indicated by an open arrow placed on the target end of the association line next to the target class (the one being navigated to). The default value of the navigability property is true (associations are bi-directional by default).
In the course registration example, the association between the Schedule and the Course Offering is navigable in both directions. That is, a Schedule must know the Course Offering assigned to the Schedule and the Course Offering must know the Schedules it has been placed in.
When no arrowheads are shown, the association is assumed to be navigable in both directions.
In the case of the associations between Schedule and Registration Controller, the Registration Controller must know its Schedules, but the Schedules have no knowledge of the Registration Controllers (or other classes, since many things have addresses) associated with the address. As a result, the navigability property of the Registration Controller end of the association is turned off.

During Use-Case Analysis, associations (and aggregations) may have been assumed to be bi-directional (i.e., communication may occur in both directions).
During Class Design, the association's navigability needs to be refined so that only the required communication gets implemented.
Navigation is readdressed in Class Design because we now understand the responsibilities and collaborations of the classes better than we did in Use-Case Analysis. We also want to refine the

relationships between classes. Two-way relationships are more difficult and expensive to implement than one-way relationships. Thus, one of the goals in Class Design is the reduction of two-way (bi-directional) relationships into one-way (uni-directional) relationships.
The need for navigation is revealed by the use-cases and scenarios. The navigability defined in the class model must support the message structure designed in the interaction diagrams.
In some circumstances even if two-way navigation is required, a one-way association may suffice. For example, you can use one-way association if navigation in one of the directions is very infrequent and does not have stringent performance requirements, or the number of instances of one of the classes is very small.

An association class is a class that is connected to an association. It is a full-fledged class and can contain attributes, operations and other associations.
Association classes allow you to store information about the relationship itself. The association class includes information that is not appropriate for, or does not belong in the classes at either end of the relationship.
There is an instance of the association class for every instance of the relationship (i.e., for every link).

In many cases, association classes are used to resolve many-to-many relationships, as shown in the example above. In this case, a Schedule includes multiple primary CourseOfferings and a CourseOffering can appear on multiple schedules as a primary. Where would a Student's grade for a primary CourseOffering "live"? It cannot be stored in Schedule because a Schedule contains multiple primary CourseOfferings. It cannot be stored in CourseOffering because the same CourseOffering can appear on multiple Schedules as primary. Grade is really an attribute of the relationship between a Schedule and a primary CourseOffering.
The same is true of the status of a CourseOffering, primary or alternate, on a particular Schedule.
Thus, association classes were created to contain such information. Two classes related by generalization were created to leverage the similarities between what must be maintained for primary and alternate CourseOfferings. Remember, Students can only enroll in and receive a grade in a primary CourseOffering, not an alternate.

If there are attributes on the association itself (represented by "association classes"), create a design class to represent the 'association class', with the appropriate attributes. Interpose this class between the other two classes, and by establishing associations with appropriate multiplicity between the association class and the other two classes
The above example demonstrates how an association class can be further designed.

When defining the navigability between the resulting classes, it was decided not to provide navigation directly to CourseOffering from Schedule (must go through PrimaryScheduleOfferingInfo).
Note: Association class design only needs to be done if the implementation does not directly support association classes and an exact visual model of the implementation is required. The above example is hypothetical. It is not included in the Course Registration Model provided with the course since the additional design details only complicated the design model.

The design of a relationship with a multiplicity greater than one can be modeled in multiple ways. You can explicitly model a container class, or you can just indicate what kind of container class will be used. The latter approach keeps the diagrams from getting too cluttered with very detailed implementation decisions. However, the former approach may be useful if you want to do code generation from your model.
Another option that is a more refined version of the first approach is to use a parameterized class (template) as the container class. This is discussed in more detail on the next few slides.

A class that exists only for other classes to inherit from it is an abstract class. Classes that are to be used to instantiate objects are concrete classes.
An operation can also be tagged as abstract. This means that no implementation exists for the operation in the class where it is specified. A class that contains at least one abstract operation must be an abstract class. Classes that inherit from an abstract class must provide implementations for the abstract operations. Otherwise, the operations are considered abstract within the subclass, and the subclass is considered abstract, as well. Concrete classes have implementations for all operations.
In the UML, you designate a class as abstract by putting the tagged value “{abstract}” within the name compartment of the class, under the class name. For abstract operations, “{abstract}” is placed after the operation signature. The name of the abstract item can also be shown in italics.
A discriminator can be used to indicate on what basis the generalization/specialization occurred. A discriminator describes a characteristic that differs in each of the subclasses. In the above example, we generalized/specialized in the area of communication.

In practice, multiple inheritance is a complex design problem and may lead to implementation difficulties.
In general, multiple inheritance causes problems if any of the multiple parents has structure or behavior that overlaps. If the class inherits from several classes, you must check how the relationships, operations, and attributes are named in the ancestors.
Specifically, there are two issues associated with multiple inheritance:

1) Name collisions: Both ancestors have attributes and/or operations with the same name.
If the same name appears in several ancestors, you must describe what this means to the specific inheriting class, for example, by qualifying the name to indicate its source of declaration.
2) Repeated inheritance: The same ancestor is being inherited by a descendant more than once. When this occurs, the inheritance hierarchy will have a "diamond shape" as shown above. The descendents end up with multiple copies of an ancestor.
If you are using repeated inheritance, you must have a clear definition of its semantics; in most cases this is defined by the programming language supporting the multiple inheritance.
In general, the programming language rules governing multiple inheritance are complex, and often difficult to use correctly. Therefore, it is recommended that you use multiple inheritance only when needed and always with caution.
Note: You can use delegation as a workaround to multiple inheritance problems. Delegation is described later in this module.

A subtype is a type of relationship expressed with inheritance. A subtype specifies that the descendent is a type of the ancestor and must follow the rules of the Is-A style of programming.
The Is-A style of programming states that the descendent is-a type of the ancestor and can fill in for all its ancestors in any situation.
The Is-A style of programming passes the Liskov Substitution Principle which states: “If for each object O1 of type S there is an object O2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O1 is substituted for O2 then S is a subtype of T”.

The classes on the left-hand side of the diagram do follow the is-a style of programming: a Lion is-an Animal and a Tiger is-an animal.
The classes on the right-hand side of the diagram do NOT follow the Is-A style of programming: a Stack is not a List. Stack needs some of the behavior of a List but not all of the behavior. If a method expects a List, then the operation insert(position) should be successful. If the method is passed a Stack, then the insert(position) will fail.

Factoring is useful if there are some services provided by a class that you want to leverage in the implementation of another class
When you factor, you extract the functionality you want to reuse and inherit from the new base class.

Inheritance provides a way to implement polymorphism in cases where polymorphism is implemented the same way for a set of classes. This means that abstract base classes that

simply declare inherited operations, but which have no implementations of the operations, can be replaced with interfaces. Inheritance can now be restricted to inheritance of implementations only, if desired.
Polymorphism is not generalization; generalization is one way to implement polymorphism. Polymorphism via generalization is the ability to define alternate methods for ancestor class operations in the descendent classes. This can reduce the amount of code to be written, as well as help abstract the interface to descendent classes.
Polymorphism is an advantage of inheritance realized during implementation and at run-time.
Programming environments which support polymorphism use dynamic binding, meaning that the actual code to execute is determined at run-time vs. compile-time.

The Design Classes should be refined to handle general non-functional requirements as stated in the Design Guidelines specific to the project. An important input to this step are the non-functional requirements on a class that may already be stated in its special requirements and responsibilities. Such requirements are often specified in terms of what architectural (analysis) mechanisms are needed to realize the class. In this step the class is refined to incorporate the design mechanisms corresponding to these analysis mechanisms.
If you remember, the available design mechanisms were identified and characterized by the architect during Identify Design Mechanisms and were documented in the Design Guidelines.
In this step, the Designer should, for each design mechanism needed, qualify as many characteristics as possible, giving ranges where appropriate.
There can be several general design guidelines and mechanisms that need to be taken into consideration when classes are designed:
How to use existing products and components
How to adapt to the programming language
How to distribute objects
How to achieve acceptable performance
How to achieve certain security levels
How to handle errors
Etc.

These are the questions you should be asking at this stage:

Does the name of each class clearly reflect the role it plays?
Does the class represent a single well-defined abstraction?
If the class does not represent a single well-defined abstraction, you should consider splitting it.
Are all attributes and responsibilities functionally coupled?

The class should only define attributes, responsibilities, or operations that are functionally coupled to the other attributes, responsibilities, or operations defined by that class.
Are there any class attributes, operations or relationships that should be generalized, that is, moved to an ancestor?
Are all specific requirements on the class addressed?
Are the demands on the class consistent with any statecharts which model the behavior of the class and its instances?
The demands on the class (as reflected in the class description and by the objects in sequence diagrams) should be consistent with any state diagram that models the behavior of the class and its objects.
Is the complete life cycle of an instance of the class described?
The complete lifecycle of an instance of the class should be described. Each object must be created, used, and removed by one or more use-case realizations.
Does the class offer the required behavior?
The classes should offer the behavior the use-case realizations and other classes require.

These are the questions you should ask about the operations of each class:
Are the operations understandable?
The names of the operations should be descriptive and the operations should be understandable to those who want to use them.
Is the state description of the class and its objects' behavior correct?
Does the class offer the behavior required of it?
Have you defined the parameters correctly?
Make sure that the parameters have been defined for each operation, and that there are not too many parameters for an operation.
Have you assigned operations for the messages of each object completely?
Are the implementation specifications (if any) for an operation correct?
Do the operation signatures conform to the standards of the target programming language?
Are all the operations needed by the use-case realizations?
Remove any operations that are redundant and not needed by the use-case realizations.

Thinking about attributes:
Does each attribute represent a single conceptual thing?
Are the names of the attributes descriptive?
Are all the attributes needed by the use-case realizations?
Remove any attributes that are redundant and not needed by the use-case realizations.
Be sure to identify and define any applicable default attribute values.

STEP 10 : Database Design.

Persistent classes need to be related to tables in the data model. This step will ensure that there is a defined relationship between the design and data model.

This activity assumes the use of a Relational Database (RDBMS).

Database Design is performed for each persistent design class.

Purpose

To ensure that persistent data is stored consistently and efficiently.

To define behavior that must be implemented in the database.

Input Artifacts

Supplementary Specifications

Use-Case Realization

Design Guidelines

Persistent Design Classes

Resulting Artifacts

Data Model

The above are the major steps of the Database Design activity.

The purpose of Database Design is to ensure that persistent data is stored consistently and efficiently, and to define behavior that must be implemented in the database.

There are additional steps in the Database Design activity. However, these two steps are the ones that apply to transitioning persistent design elements to the data model. The other activities are outside the scope of this course.

Relational databases and object orientation are not entirely compatible. They represent two different views of the world: in an RDBMS, all you see is data; in an Object-Oriented system, all you see is behavior. It is not that one perspective is better than the other, but they are different. The Object-Oriented model tends to work well for systems with complex behavior and state-specific behavior in which data is secondary, or systems in which data is accessed navigationally in a natural hierarchy (e.g. bills of materials). The RDBMS model is well-suited to reporting applications and systems in which the relationships are dynamic or ad-hoc.

The real fact of the matter is that a lot of information is stored in relational databases, and if Object-Oriented applications want access to that data, they need to be able to read and write to an RDBMS. In addition, Object-Oriented systems often need to share data with non-Object-Oriented systems. It is natural, therefore, to use an RDBMS as the sharing mechanism.

While object-oriented and relational design share some common characteristics (an object's attributes are conceptually similar to an entity's columns), fundamental differences make seamless integration a challenge. The fundamental difference is that data models expose data (through column values) while object models hide data (encapsulating it behind its public interfaces).

Sl No.	Activity	Modeling Steps
0.	Use case Diagram	Select Usecase View Main and create Actors & Use cases. Select Usecase View, Right click and select new Usecase diagram and rename it as Package diagram. Dbl click on the same and create packages in the diagrams. Drag Use cases & Actors respectively.
1.	Architecture Analysis ✓ **Identify Patterns** ✓ **Define analysis mechanisms.** ✓ **Define key abstractions**	Select Logical View, Right click and select new class diagram and rename the same as Architecture diagram. Dbl click on Architecture diagram and create Layers (Select Package , right click, open specification and select Stereotype Layer) Select Notes and document Architecture Mechanisms & Key Abstractions. Select Logical View, Right click and select new class diagram and rename the same as Realization diagram. Select the usecase to be realized from Usecase view, also create Usecase Realization by selecting Usecase Realization notation in the tool bar.
2	Use Case Analysis ✓ **Supplement use case descriptions** ✓ **Find classes and distribute behavior** ✓ **Describe responsibilities, attributes and associations, and qualify analysis**	Select Usecase to be realized, Right click and create new class diagram. Create Boundary, control & Entity classes. Select Usecase to be realized, Right click and create new Sequence diagram and rename it as basic flow and repeat the same for alternative flows. Right clk on the messages, select new operation option and add // responsibility.

	mechanisms ✓ **Unify analysis classes** ✓ **Insummary, develop analysis use-case realizations**	Select Sequence diagram and press Function key F5 and you would get collaboration diagram. Links in collaboration diagram results in relationship between classes. (This is manual step)
3	Design Mechanisms	Expand Arch Analysis Mechanism chart created in Arch Analysis phase by adding details in Design & Implementation Mechanism Select Design patterns. Load them if it is already there in your repositories.
4	**Design Model** ✓ **Packages and their relationships, and their contents (now includes lower-level layers and packages)** ✓ **Subsystems, Interfaces, and their relationships** ✓ **Update Packages and subsystems to Layers.**	Select Logical view, Right click and create new class diagram and rename it as Package diagram Design. Dbl click and create Packages and group classes accordingly. Select Logical view, Right click and create new class diagram and rename it as Subsystem diagram. Dbl click and create Packages, right clk and stereotype it as subsystem. Create interface class and connect with Subsystem using Realization relationships. Group packages & Subsystems into Layers.
5	Run time Architecture ✓ **Identify concurrency requirements** ✓ **Model processes** ✓ **Map the processes to the implementation environment** **Map model elements to processes**	Select Component view Main, Select Component from Tool bar and stereotype as Process. Right clk on the component and select realizes tab and select the classes necessary to build that component. Create other processes accordingly and build relationships.
6	Describe Distribution ✓ **Analyze distribution patterns and network configuration** ✓ **Allocate processes to nodes**	Select Deployment view, Dbl click and select Processor and Nodes. Rename it as Clients, Servers. Right clk on the Processor and select the process that runs on the same. If there is more than one process running, make sure you select scheduling algorithm. Make network connections.
7	Use case design ✓ **Define interactions between design objects, incorporating any architectural mechanisms** ✓ **Encapsulate Common Sub flows** ✓ **Refine Flow of Events description** ✓ **Unify classes and subsystems**	Load Design patterns Update VOPC with the necessary design patterns Update Sequence diagrams by linking Design patterns Update VOPC, Sequence, and collaboration with the interfaces that you have identified in the Design Elements Module. Look for Consistency in Sequence diagram, if needed Simplify them.

8	Subsystem design. ✓ **Distribute subsystem behavior to subsystem elements** ✓ **Document subsystem elements** ✓ **Describe subsystem dependencies**	Select Subsystem diagram, Dbl clk on Subsystem and complete the design.
9	Class design ✓ **Create initial design classes** ✓ **Define operations** ✓ **Define methods** ✓ **Define states** ✓ **Define attributes** ✓ **Define dependencies** ✓ **Define associations** **Define generalizations**	Replace responsibilities by Operations. Complete by adding Signature, arguments and return types. Select attribute data types and specify signature. Specify Association, Aggregation, Composition relationships with multiplicities and roles defined. Specify Inheritance and dependency relationships.
10	Database Design	Select Entity classes that have to be made persistent and group them in a package. Select Data Modeler option and create Data Model diagram. You can create Scripts (ddl or SQL from this model) and can also take the entities to Relational databases. Reverse Engineer an existing database and generate object Model.