



System Design



What is System Design?


- ❑ System design is the transformation of an analysis model into a system design model.
- ❑ During system design, developers define the design goals of the project and decompose the system into smaller subsystems that can be realized by individual teams.
- ❑ Developers also select strategies for building the system, such as the hardware/software strategy, the persistent data management strategy, the global control flow, the access control policy, and the handling of boundary conditions.
- ❑ The result of system design is a model that includes a subsystem decomposition and a clear description of each of these strategies.
- ❑ System design is not algorithmic.




Activities of System design




Identify design goals

-  Developers identify and prioritize the qualities of the system that they should optimize.

Design the initial subsystem decomposition

-  Developers decompose the system into smaller parts based on the use case and analysis models

Refine the subsystem decomposition to address the design goals

-  The initial decomposition usually does not satisfy all design goals. Developers refine it until all goals are satisfied.



Overview of System Design



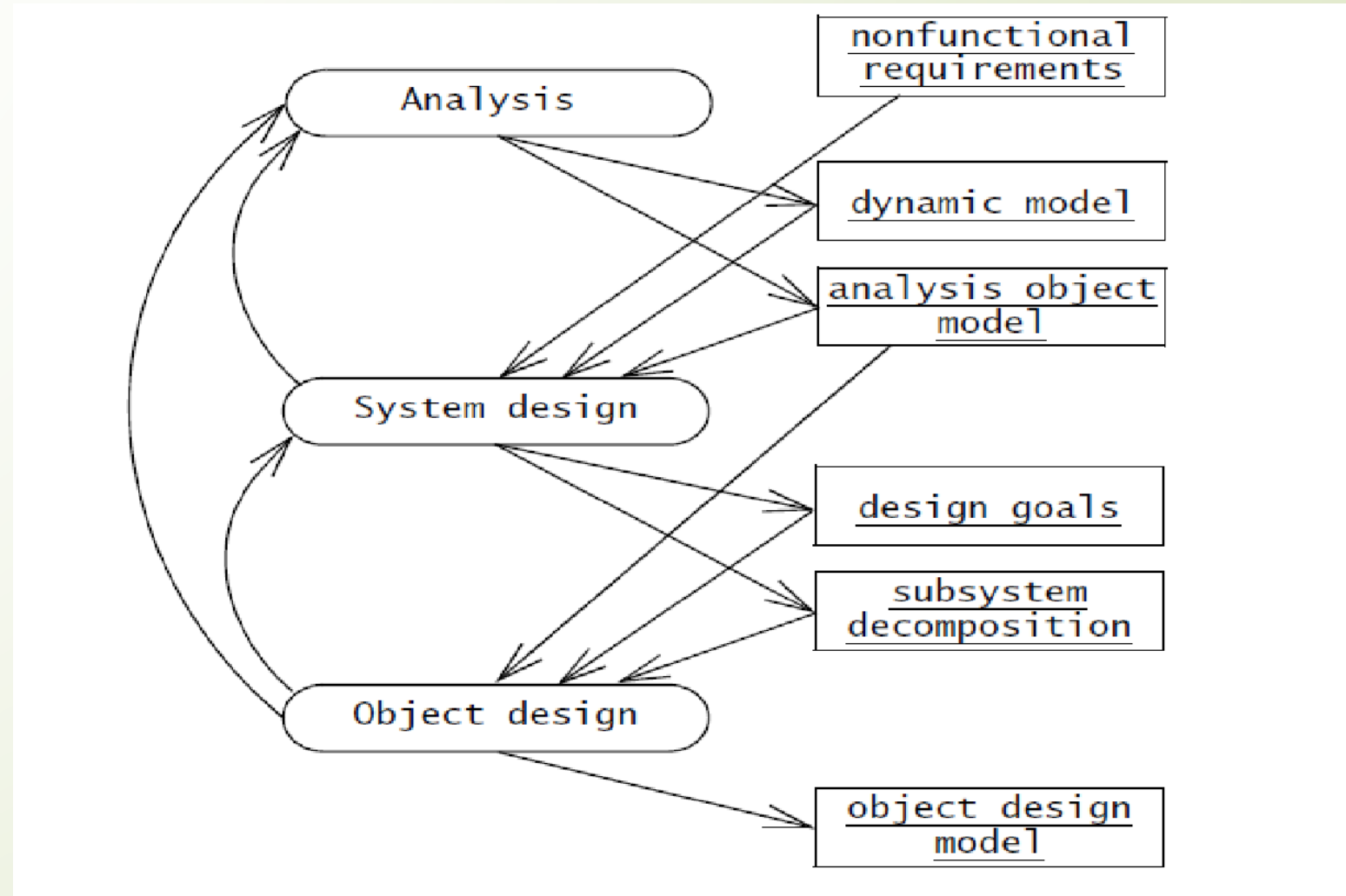
- ❑ Analysis results in the requirements model described by the following products:
 - ❑ a set of *nonfunctional requirements* and *constraints*, such as maximum response time, minimum throughput, reliability, operating system platform, and so on
 - ❑ a *use case model*, describing the system functionality from the actors' point of view
 - ❑ an *object model*, describing the entities manipulated by the system
 - ❑ a *sequence diagram* for each use case, showing the sequence of interactions among objects participating in the use case.



Outcomes of System Design:

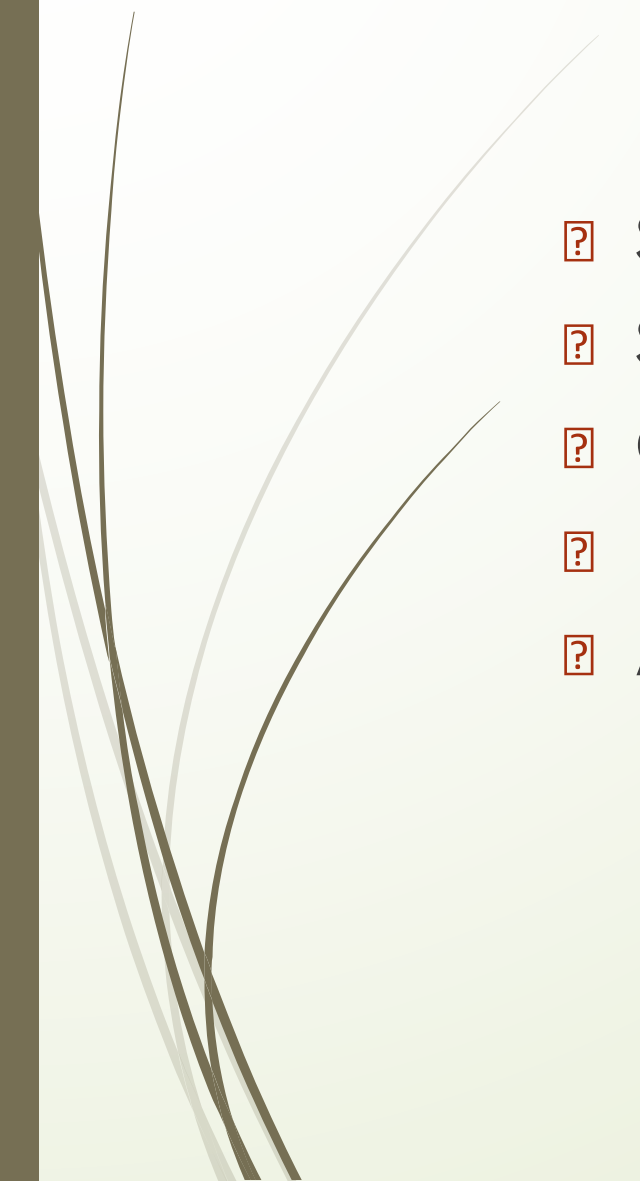
- ❑ System design results in the following products:
 - ❑ *design goals*, describing the qualities of the system that developers should optimize
 - ❑ *software architecture*, describing the subsystem decomposition in terms of subsystem responsibilities, dependencies among subsystems, subsystem mapping to hardware, and major policy decisions such as control flow, access control, and data storage
 - ❑ *boundary use cases*, describing the system configuration, startup, shutdown, and exception handling issues.

Activities of system design





System Design Concepts

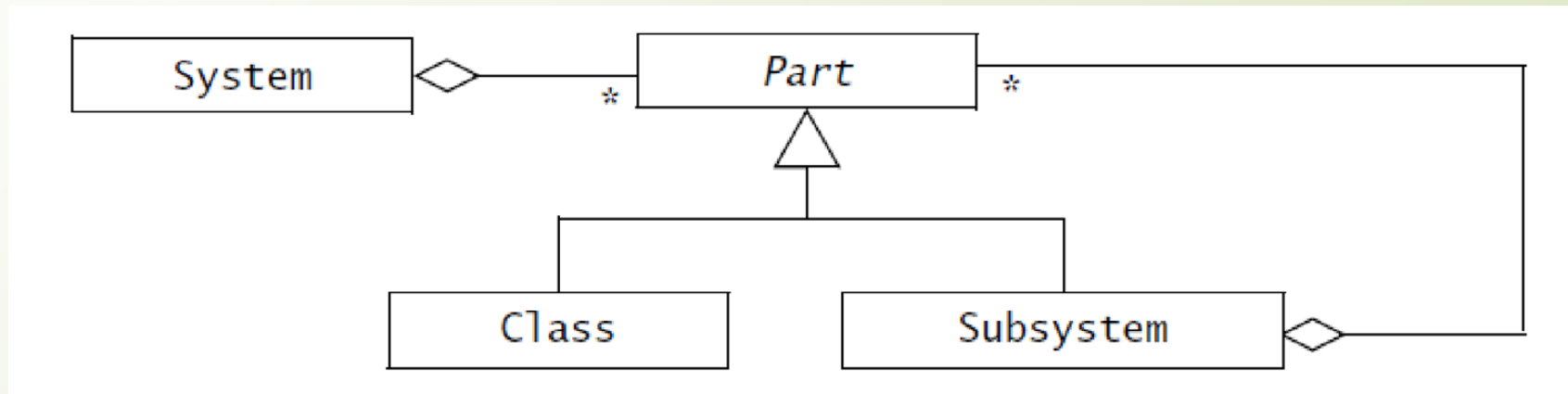
- ❑ Subsystems and Classes
 - ❑ Services and Subsystem Interfaces
 - ❑ Coupling and Cohesion
 - ❑ Layers and Partitions
 - ❑ Architectural Styles
- 



Subsystems and Classes

- ❓ In order to reduce the complexity of the application domain, we identified smaller parts called “classes” and organized them into packages.
- ❓ Similarly, to reduce the complexity of the solution domain, we decompose a system into simpler parts, called “subsystems,” which are made of a number of solution domain classes.
- ❓ A **subsystem** is a replaceable part of the system with well-defined interfaces that encapsulates the state and behavior of its contained classes.
- ❓ A subsystem typically corresponds to the amount of work that a single developer or a single development team can tackle.
- ❓ By decomposing the system into relatively independent subsystems, concurrent teams can work on individual subsystems with minimal communication overhead.
- ❓ In the case of complex subsystems, we recursively apply this principle and decompose a subsystem into simpler subsystems

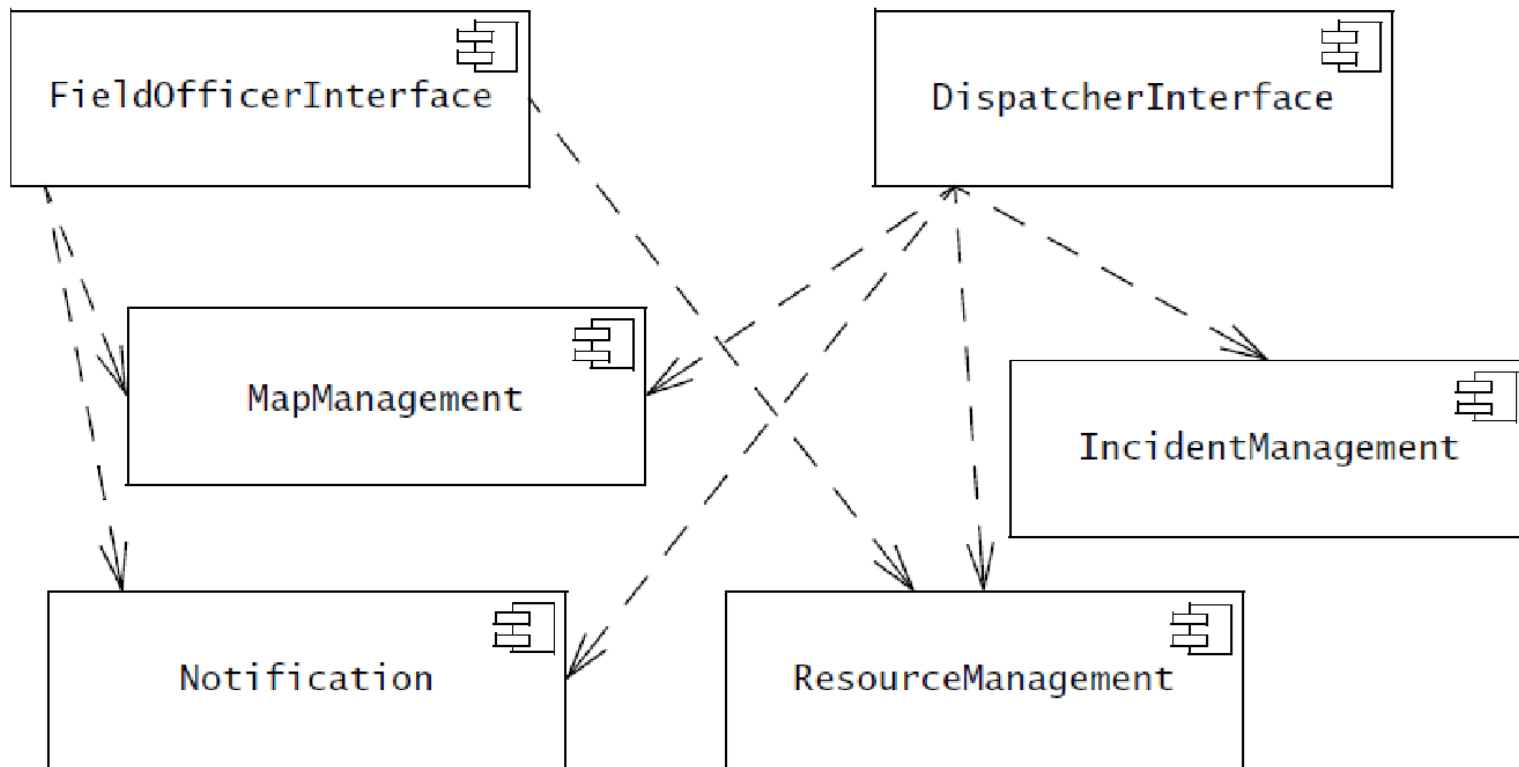
Subsystem decomposition (UML class diagram)


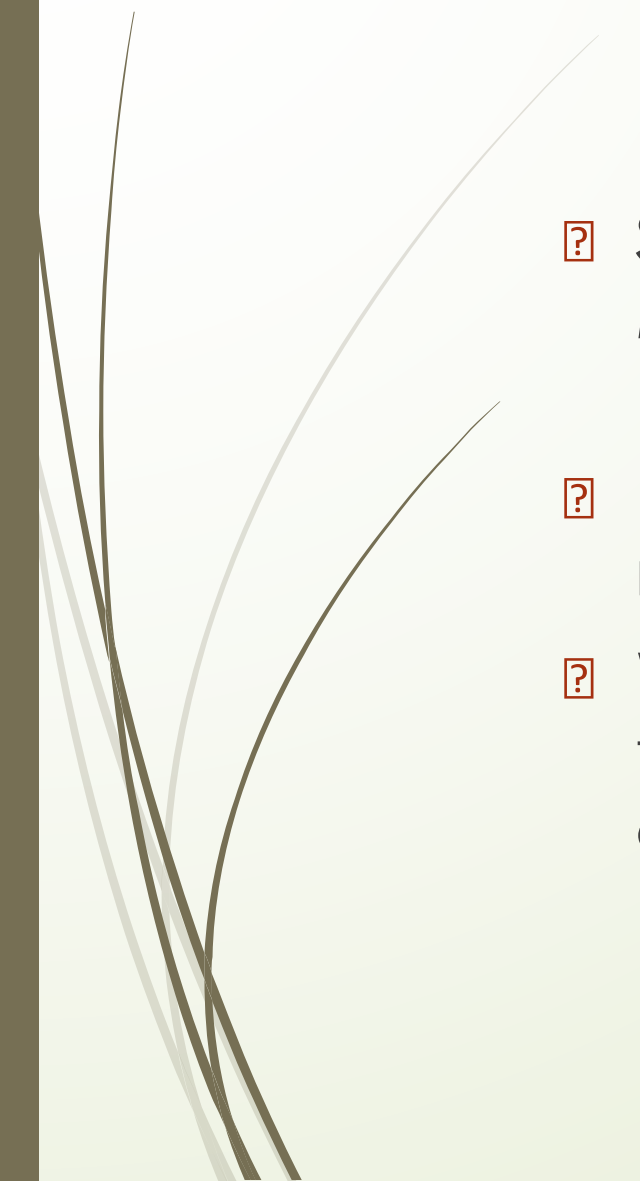




How to depict subsystems in UML?

- ❑ Components are depicted as rectangles with the component icon in the upper right corner.
- ❑ Dependencies among components can be depicted with dashed stick arrows.
- ❑ In UML, components can represent both logical and physical components.
- ❑ A **logical component** corresponds to a subsystem that has no explicit run-time equivalent, for example, individual business components that are composed together into a single run-time application logic layer.
- ❑ A **physical component** corresponds to a subsystem that as an explicit run-time equivalent, for example, a database server.



- 
- 
- ❑ Several programming languages (e.g., Java and Modula-2) provide constructs for modeling subsystems (packages in Java, modules in Modula-2).
 - ❑ In other languages, such as C or C++, subsystems are not explicitly modeled.
 - ❑ Whether or not subsystems are explicitly represented in the programming language, developers use conventions for grouping classes



Services and Subsystem Interfaces

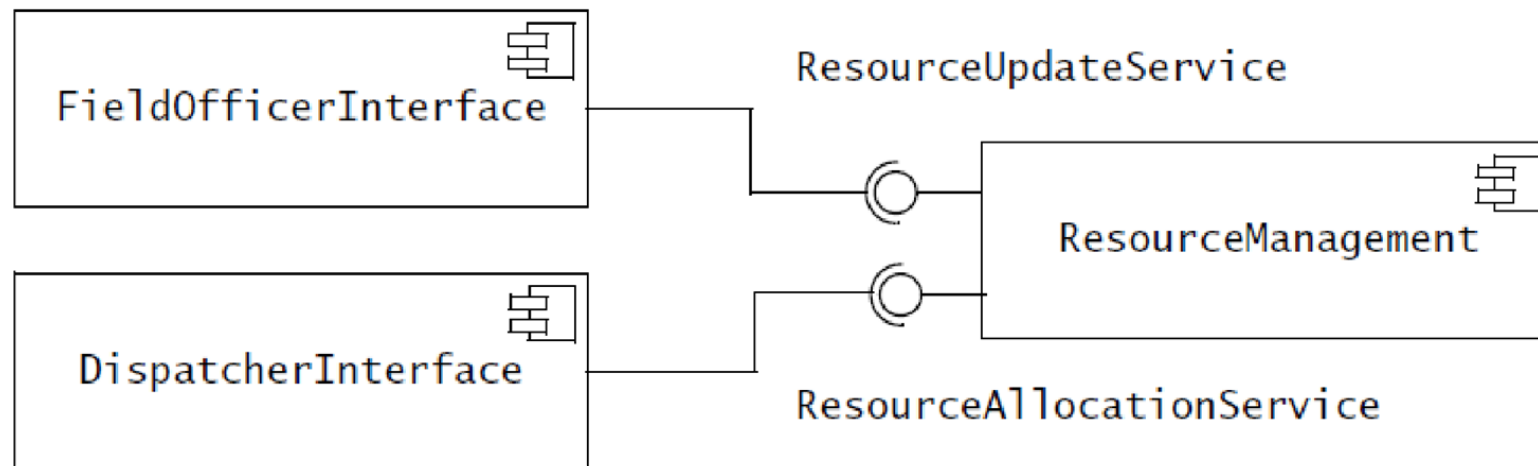
- ❑ A subsystem is characterized by the services it provides to other subsystems.
- ❑ A **service** is a set of related operations that share a common purpose.
- ❑ A subsystem providing a notification service, for example, defines operations to send notices, look up notification channels, and subscribe and unsubscribe to a channel.
- ❑ The set of operations of a subsystem that are available to other subsystems form the **subsystem interface**.
- ❑ The subsystem interface includes the name of the operations, their parameters, their types, and their return values.
- ❑ System design focuses on defining the services provided by each subsystem, that is, enumerating the operations, their parameters, and their high-level behavior.
- ❑ Object design will focus on the **application programmer interface** (API), which refines and extends the subsystem interfaces



Representation



- ❑ Provided and required interfaces can be depicted in UML with **assembly connectors**, also called **ball-and-socket connectors**.
- ❑ The provided interface is shown as a ball icon (also called lollipop) with its name next to it.
- ❑ A required interface is shown as a socket icon.
- ❑ The dependency between two subsystems is shown by connecting the corresponding ball and socket in the component diagram.


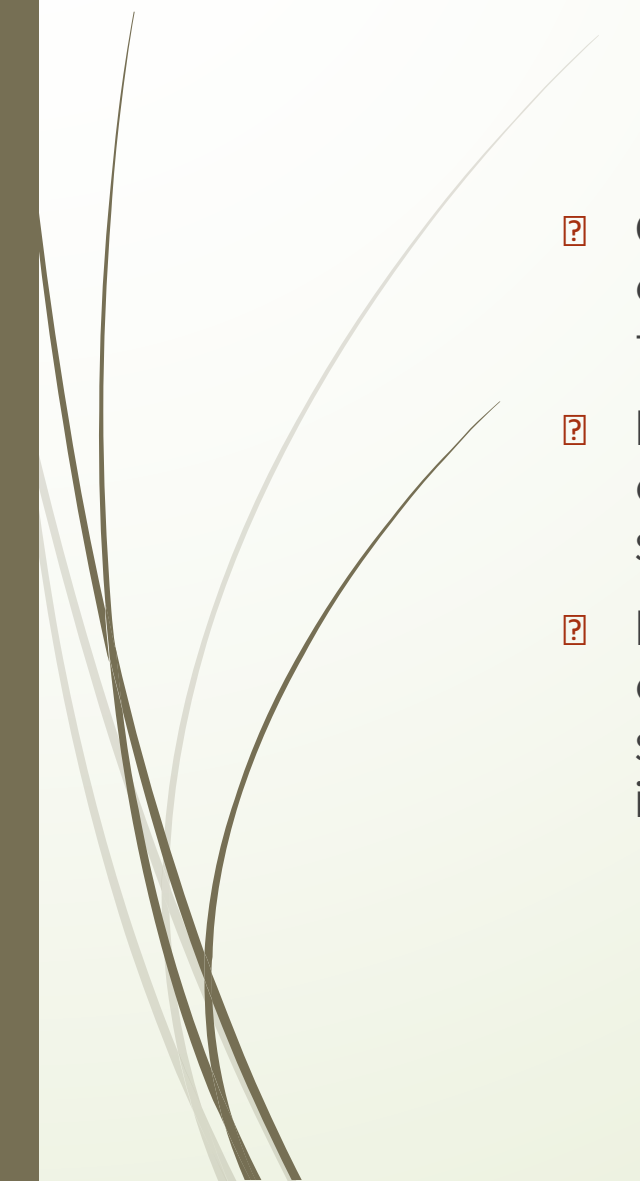




Coupling and Cohesion



- ❑ **Coupling** is the number of dependencies between two subsystems.
- ❑ If two subsystems are loosely coupled, they are relatively independent, so modifications to one of the subsystems will have little impact on the other. If two subsystems are strongly coupled, modifications to one subsystem is likely to have impact on the other.
- ❑ A desirable property of a subsystem decomposition is that subsystems are as loosely coupled as reasonable.
- ❑ By reducing coupling, developers can introduce many unnecessary layers of abstraction that consume development time and processing time.

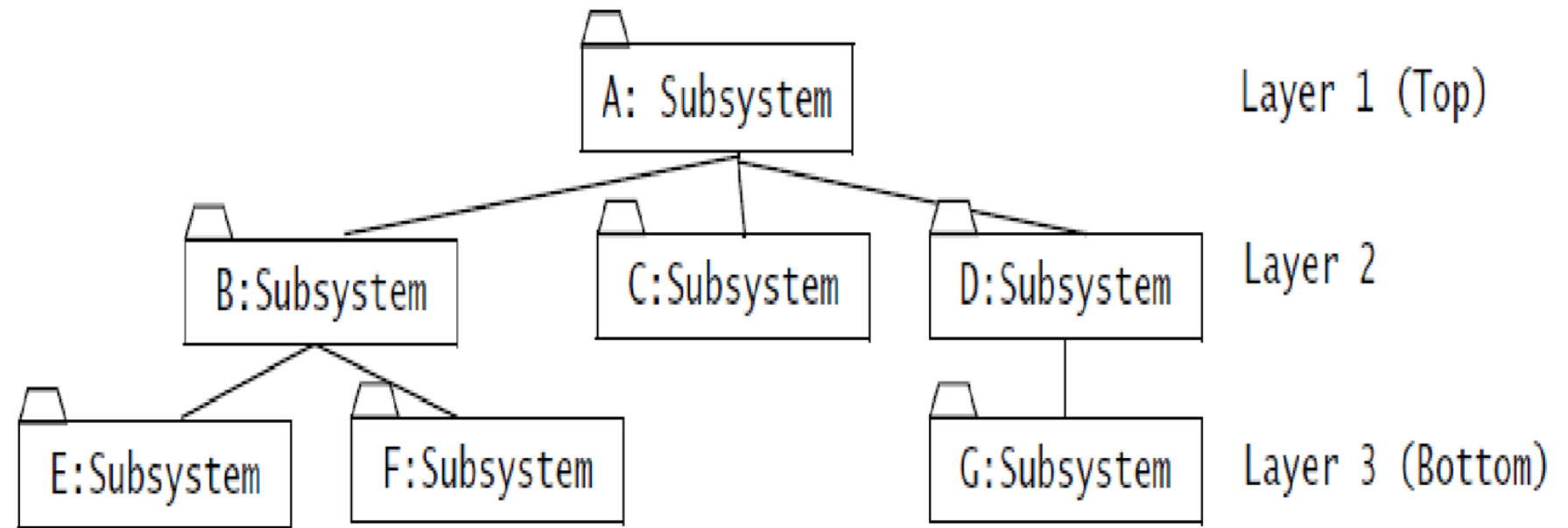
- 
- 
- ❓ **Cohesion** is the number of dependencies within a subsystem. If a subsystem contains many objects that are related to each other and perform similar tasks, its cohesion is high
 - ❓ If a subsystem contains a number of unrelated objects, its cohesion is low. A desirable property of a subsystem decomposition is that it leads to subsystems with high cohesion.
 - ❓ In general, there is a trade-off between cohesion and coupling. We can often increase cohesion by decomposing the system into smaller subsystems. However, this also increases coupling as the number of interfaces increases.



Layers and Partitions

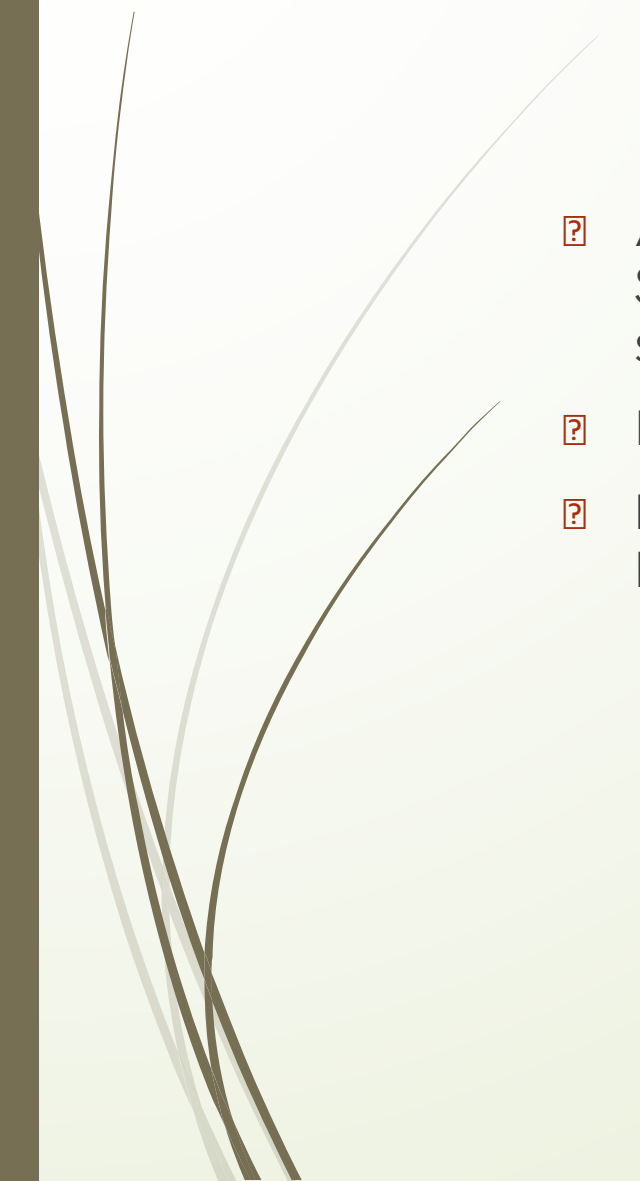


- ❑ A **hierarchical decomposition** of a system yields an ordered set of layers.
- ❑ A **layer** is a grouping of subsystems providing related services, possibly realized using services from another layer.
- ❑ Layers are ordered in that each layer can depend only on lower level layers and has no knowledge of the layers above it.
- ❑ The layer that does not depend on any other layer is called the bottom layer, and the layer that is not used by any other is called the top layer.
- ❑ In a **closed architecture**, each layer can access only the layer immediately below it.
- ❑ In an **open architecture**, a layer can also access layers at deeper levels.





Closed architecture – ISO OSI Model

- ❑ An example of a closed architecture is the Reference Model of Open Systems Interconnection (in short, the OSI model), which is composed of seven layers.
 - ❑ Each layer is responsible for performing a well-defined function.
 - ❑ In addition, each layer provides its services by using services of the layer below.
- 



Open architecture – Swing UI toolkit

- ❑ An example of an open architecture is the Swing user interface toolkit for Java
- ❑ The lowest layer is provided by the operating system or by a windowing system
- ❑ AWT is an abstract window interface provided by Java to shield applications from specific window platforms.
- ❑ Swing is a library of user interface objects that provides a wide range of facilities, from buttons to geometry management.
- ❑ An Application usually accesses only the Swing interface
- ❑ However, the Application layer may bypass the Swing layer and directly access AWT.
- ❑ In general, the openness of the architecture allows developers to bypass the higher layers to address performance bottlenecks



Layers and Partitions



- ❑ Another approach to dealing with complexity is to **partition** the system into peer subsystems, each responsible for a different class of services
- ❑ Each subsystem depends loosely on the others, but can often operate in isolation
- ❑ In general, a subsystem decomposition is the result of both partitioning and layering.



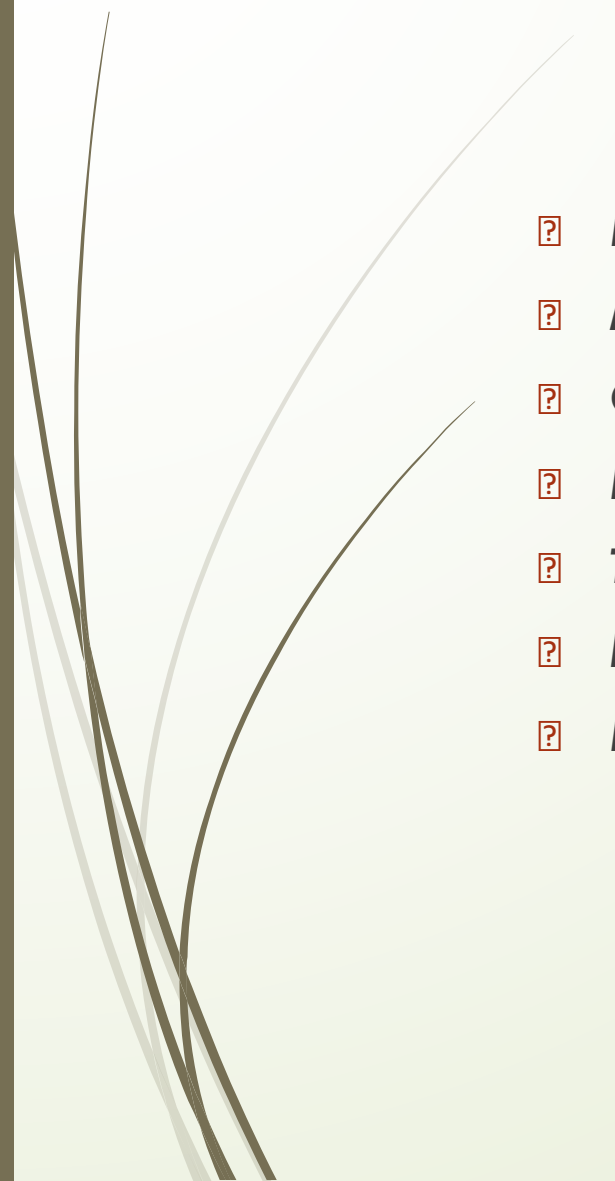
Architectural Styles



- ❑ As the complexity of systems increases, the specification of system decomposition is critical.
- ❑ It is difficult to modify or correct weak decomposition once development has started, as most subsystem interfaces would have to change.
- ❑ In recognition of the importance of this problem, the concept of **software architecture** has emerged.
- ❑ A software architecture includes system decomposition, global control flow, handling of boundary conditions, and intersubsystem communication protocols

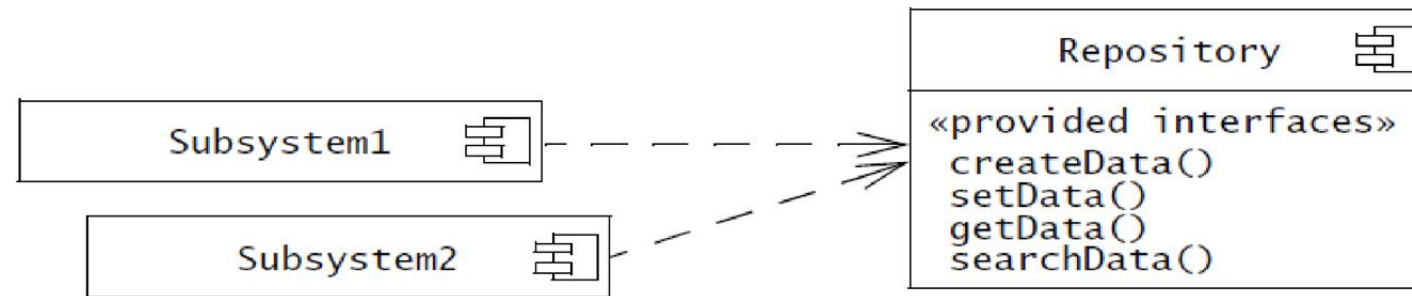


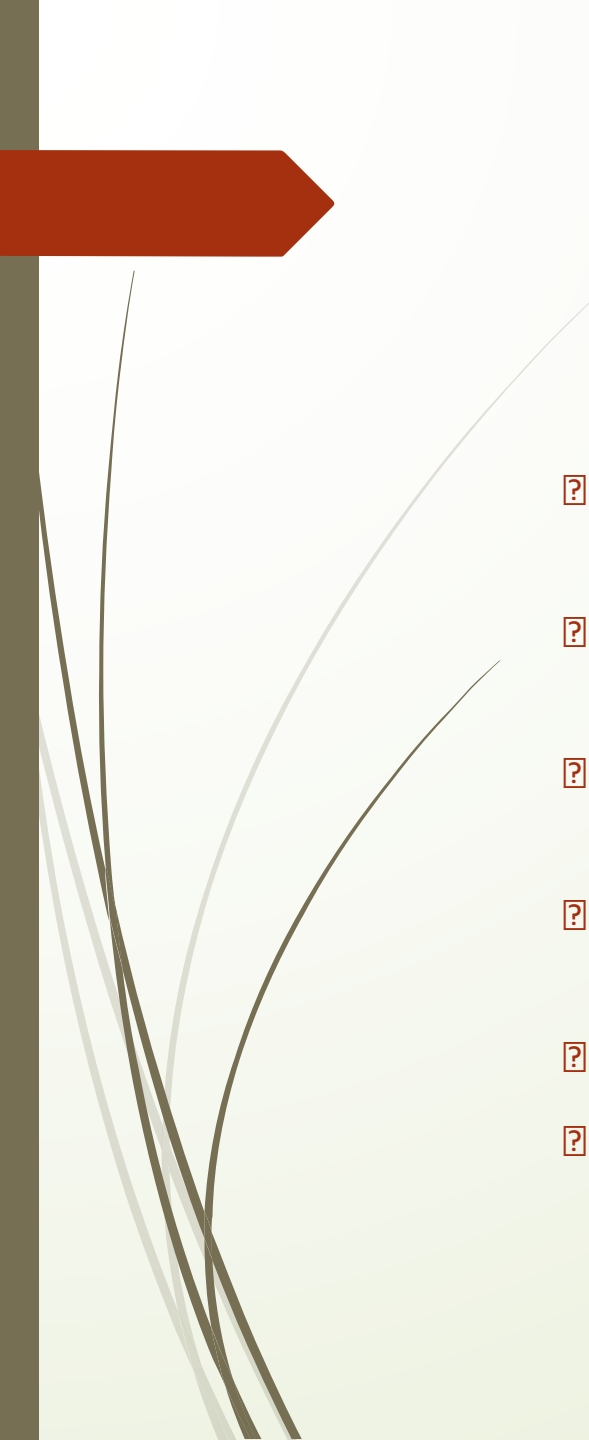
Different architectures discussed


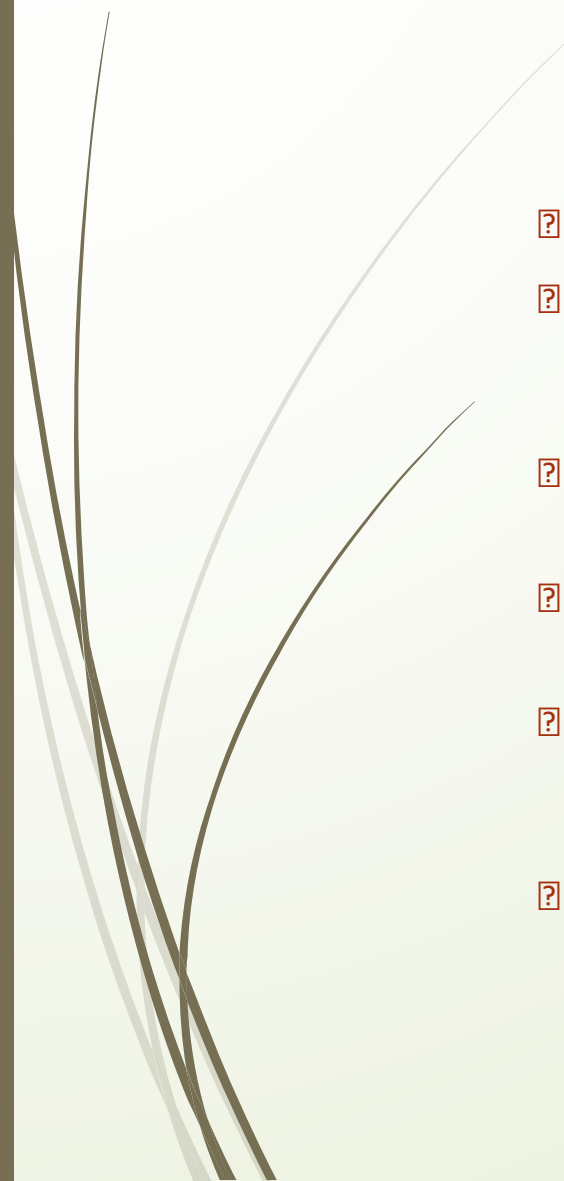
- ❑ ***Repository***
 - ❑ ***Model/View/Controller***
 - ❑ ***Client/server***
 - ❑ ***Peer-to-peer***
 - ❑ ***Three-tier***
 - ❑ ***Four-tier***
 - ❑ ***Pipe and filter***
- 

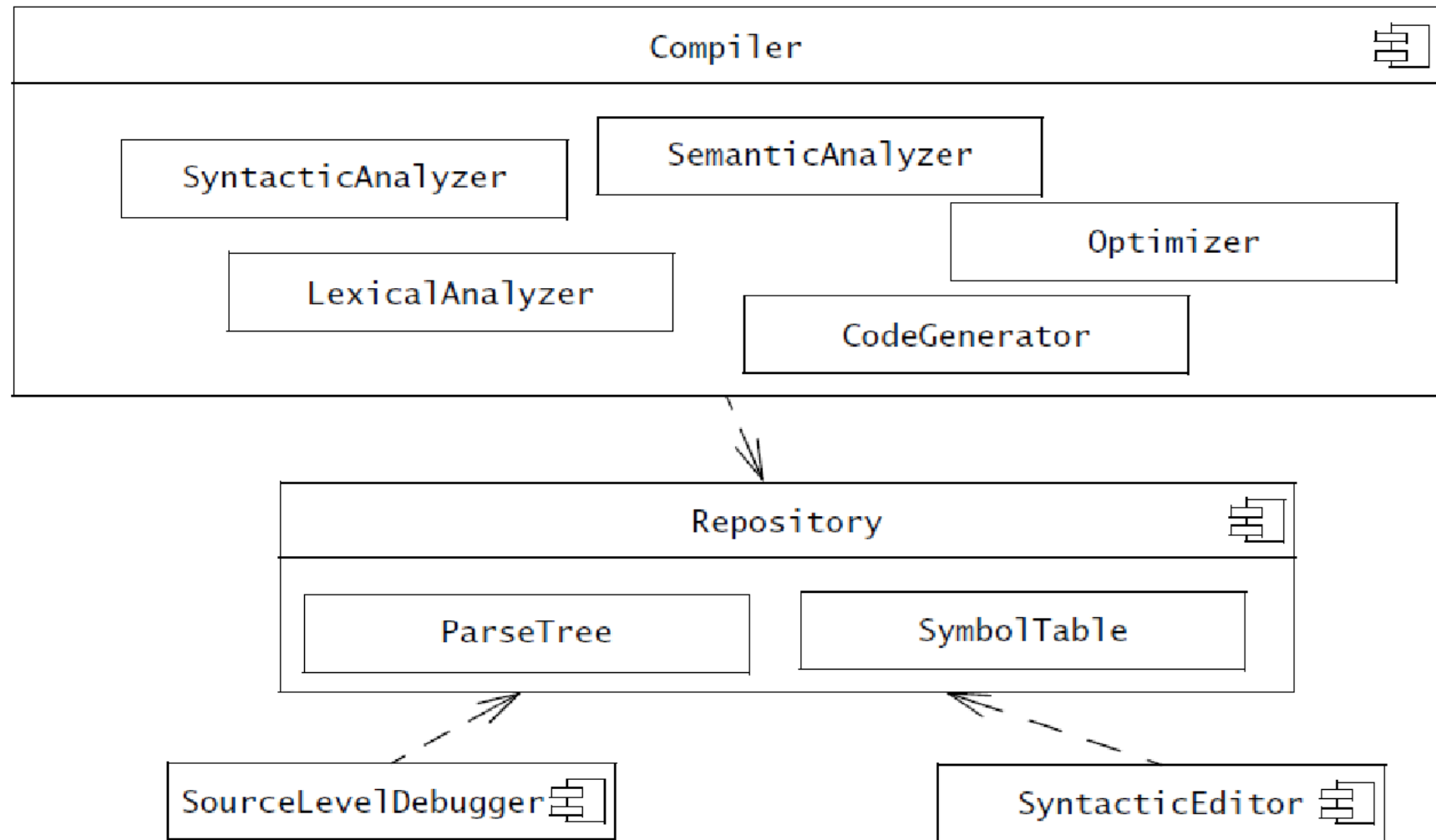
Repository

- ❓ In the **repository architectural style** subsystems access and modify a single data structure called the central **repository**.
- ❓ Subsystems are relatively independent and interact only through the repository.
- ❓ Control flow can be dictated either by the central repository
- ❓ e.g., triggers on the data invoke peripheral systems) or by the subsystems e.g., independent flow of control and synchronization through locks in the repository).



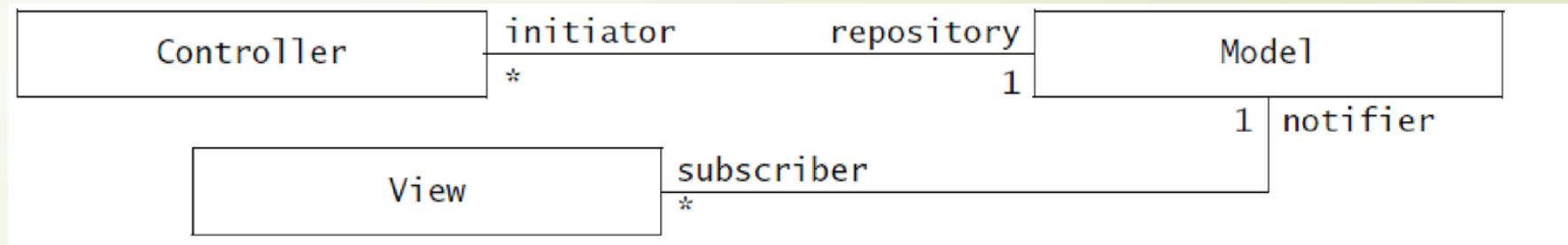
- 
- ❑ Repositories are typically used for database management systems, such as a payroll system or a bank system.
 - ❑ The central location of the data makes it easier to deal with concurrency and integrity issues between subsystems.
 - ❑ Compilers and software development environments also follow a repository architectural style .
 - ❑ The different subsystems of a compiler access and update a central parse tree and a symbol table.
 - ❑ Debuggers and syntax editors access the symbol table as well.
 - ❑ The repository subsystem can also be used for implementing the global control flow.

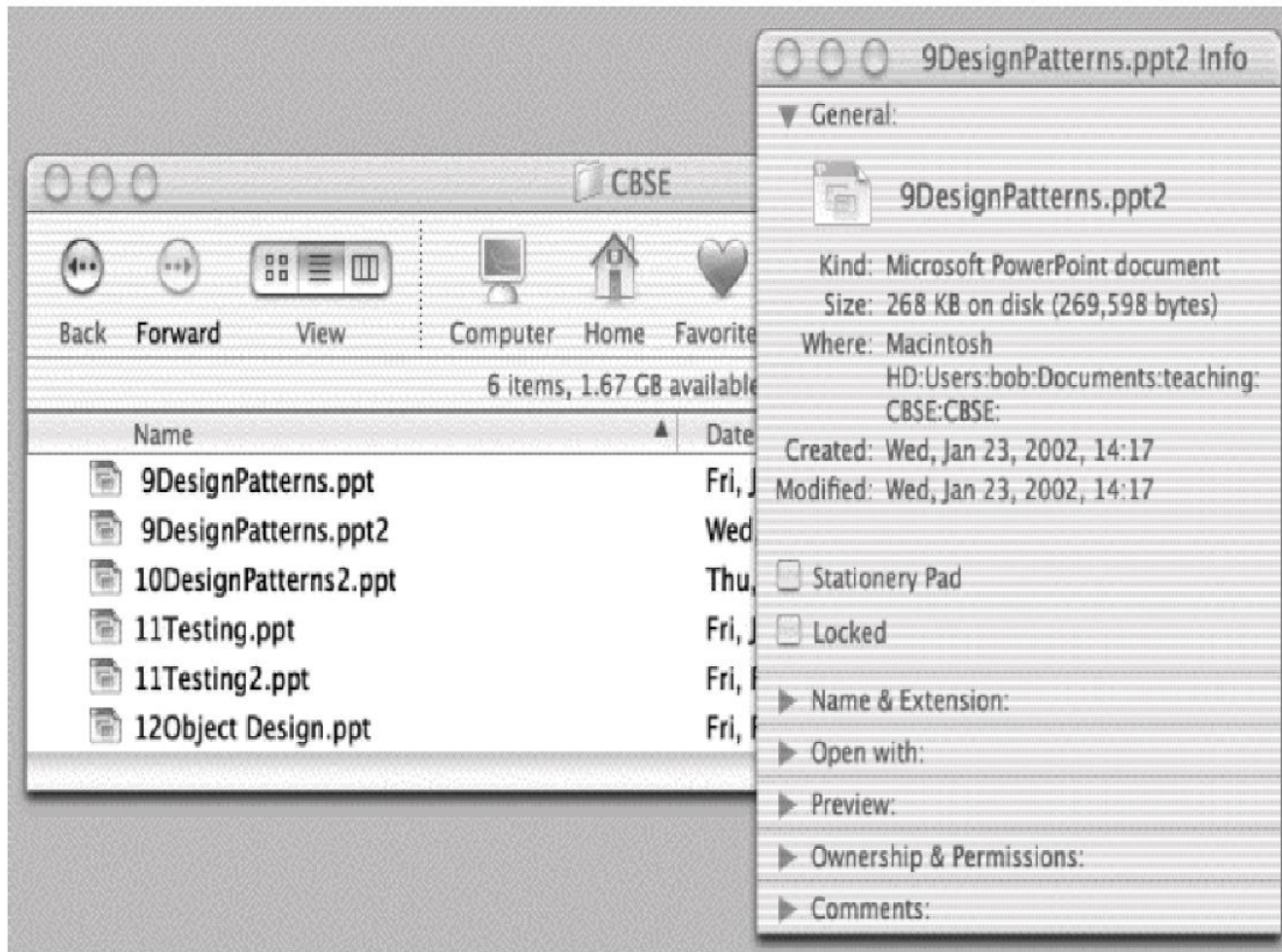
- 
- 
- ❑ The repository only ensures that concurrent accesses are serialized.
 - ❑ Conversely, the repository can be used to invoke the subsystems based on the state of the central data structure. These systems are called “blackboard systems.”
 - ❑ Repositories are well suited for applications with constantly changing, complex dataprocessing tasks.
 - ❑ Once a central repository is well defined, we can easily add new services in the form of additional subsystems.
 - ❑ The main disadvantage of repository systems is that the central repository can quickly become a bottleneck, both from a performance aspect and a modifiability aspect.
 - ❑ The coupling between each subsystem and the repository is high, thus making it difficult to change the repository without having an impact on all subsystems




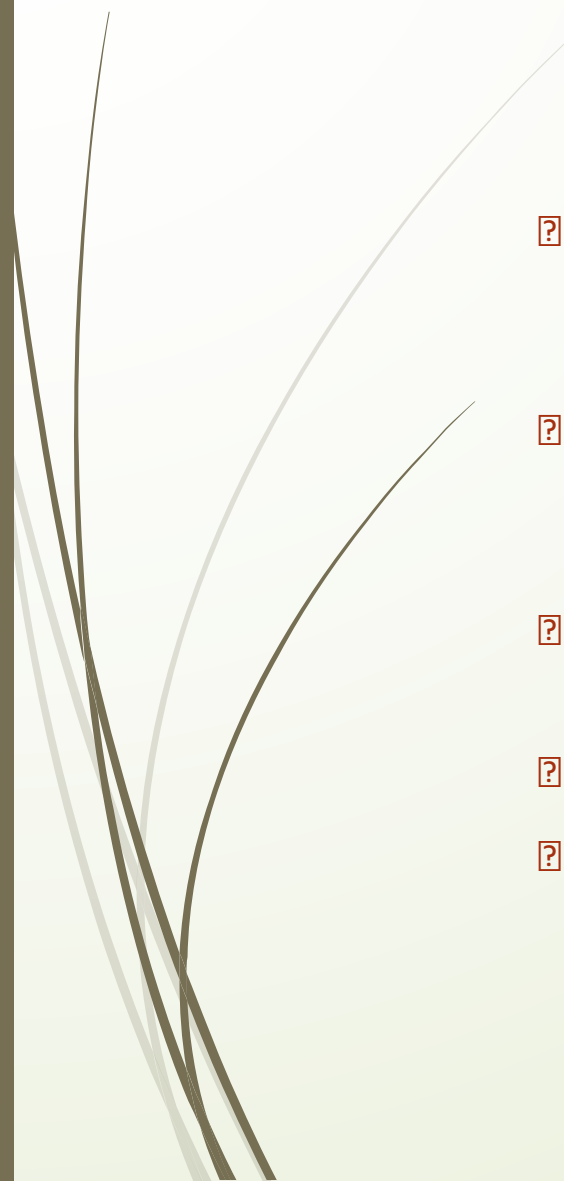
Model/View/Controller

- ❑ In the **Model/View/Controller** (MVC) architectural style subsystems are classified into three different types:
 - ❑ **model** subsystems maintain domain knowledge,
 - ❑ **View** subsystems display it to the user, and
 - ❑ **controller** subsystems manage the sequence of interactions with the user





- ? An example of MVC architectural style.
- ? The “model” is the filename 9DesignPatterns2.ppt.
- ? One “view” is a window titled CBSE, which displays the contents of a folder containing the file 9DesignPatterns2.ppt. The other “view” is window called 9DesignPatterns2.ppt
- ? Info, which displays information related to the file. If the file name is changed, both views are updated by the “controller.”

- 
- 
- ❑ The rationale between the separation of Model, View, and Controller is that user interfaces, i.e., the View and the Controller, are much more often subject to change than is domain knowledge, i.e., the Model.
 - ❑ Moreover, by removing any dependency from the Model on the View with the subscription/notification protocol, changes in the views (user interfaces) do not have any effect on the model subsystems.
 - ❑ MVC is well suited for interactive systems, especially when multiple views of the same model are needed.
 - ❑ MVC can be used for maintaining consistency across distributed data;
 - ❑ however it introduces the same performance bottleneck as for other repository styles.



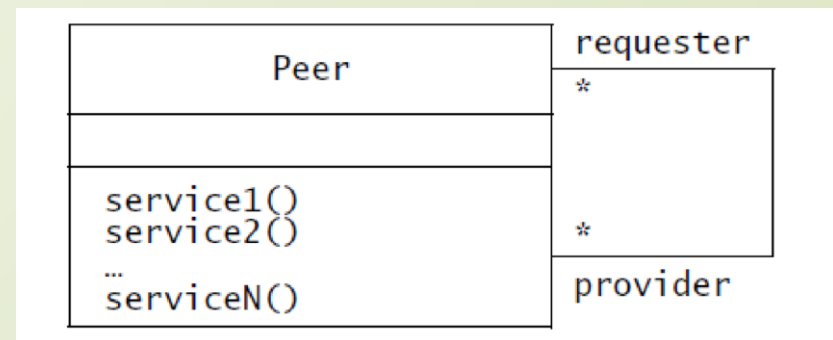
Client/server



- ❑ In the **client/server architectural style**, a subsystem, the **server**, provides services to instances of other subsystems called the **clients**, which are responsible for interacting with the user
- ❑ An information system with a central database is an example of a client/server architectural style.
- ❑ The clients are responsible for receiving inputs from the user, performing range checks, and initiating database transactions when all necessary data are collected.
- ❑ The server is then responsible for performing the transaction and guaranteeing the integrity of the data.
- ❑ In this case, a client/server architectural style is a special case of the repository architectural style in which the central data structure is managed by a process.
- ❑ Client/server systems, however, are not restricted to a single server. On the World Wide Web, a single client can easily access data from thousands of different servers
- ❑ Client/server architectural styles are well suited for distributed systems that manage large amounts of data

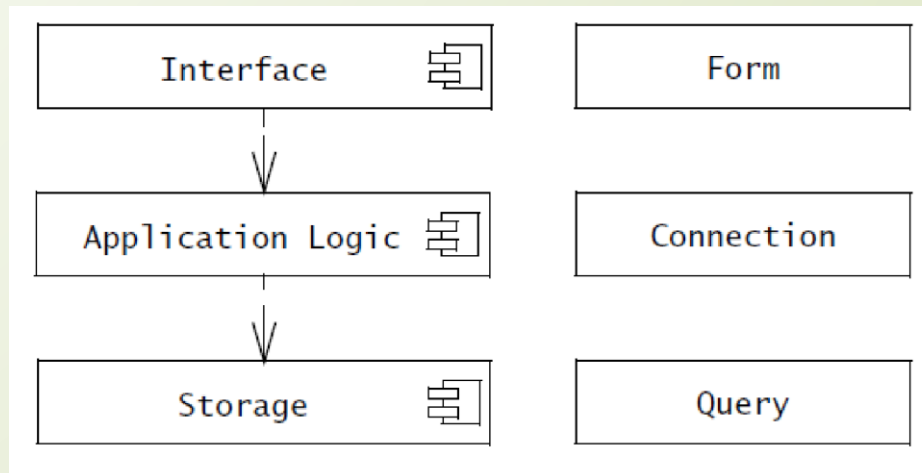
Peer-to-peer

- ❑ A **peer-to-peer architectural style** is a generalization of the client/server architectural style in which subsystems can act both as client or as servers, the sense that each subsystem can request and provide services.
- ❑ The control flow within each subsystem is independent from the others except for synchronizations on requests
- ❑ Peer-to-peer systems are more difficult to design than client/server systems because they introduce the possibility of deadlocks and complicate the control flow.



Three-tier

- ❑ The **three-tier architectural style** organizes subsystems into three layers:
 - ❑ The *interface layer* includes all boundary objects that deal with the user, including windows, forms, web pages, and so on.
 - ❑ The *application logic layer* includes all control and entity objects, realizing the processing, rule checking, and notification required by the application.
 - ❑ The *storage layer* realizes the storage, retrieval, and query of persistent objects.







Four-tier



- ❑ The **four-tier architectural style** is a three-tier architecture in which the Interface layer is decomposed into a Presentation Client layer and a Presentation Server layer
- ❑ The Presentation Client layer is located on the user machines, whereas the Presentation Server layer can be located on one or more servers.
- ❑ The four-tier architecture enables a wide range of different presentation clients in the application, while reusing some of the presentation objects across clients



Presentation Client

WebBrowser

Presentation Server


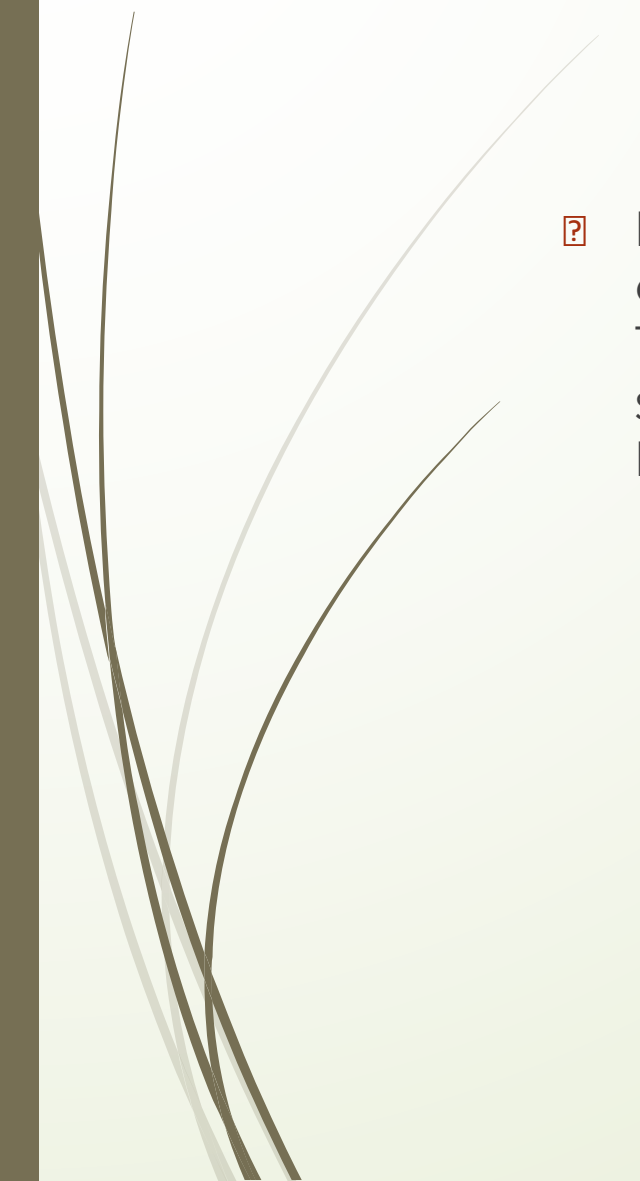
Form

Application Logic

Connection

Storage

Query

- 
- 
- ❓ For example, a banking information system can include a host of different clients, such as a Web browser interface for home users, an Automated Teller Machine, and an application client for bank employees. Forms shared by all three clients can then be defined and processed in the Presentation Server layer, thus removing redundancy across clients.



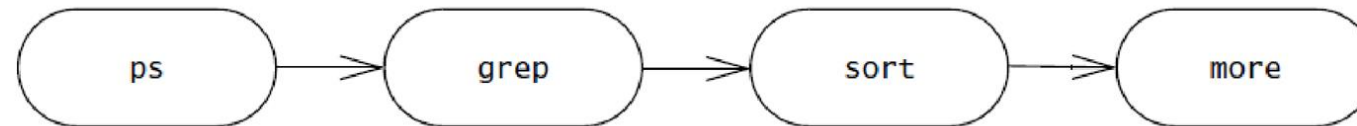
Pipe and filter

- ❑ In the **pipe and filter architectural style** , subsystems process data received from a set of inputs and send results to other subsystems via a set of outputs.
- ❑ The subsystems are called “filters,” and the associations between the subsystems are called “pipes.”
- ❑ Each filter knows only the content and the format of the data received on the input pipes, not the filters that produced them.
- ❑ Each filter is executed concurrently, and synchronization is accomplished via the pipes.
- ❑ The pipe and filter architectural style is modifiable: filters can be substituted for others or reconfigured to achieve a different purpose.

Unix command line


```
% ps auxwww | grep dutoit | sort | more
```

```
dutoit 19737 0.2 1.6 1908 1500 pts/6 0 15:24:36 0:00 -tcsh  
dutoit 19858 0.2 0.7 816 580 pts/6 S 15:38:46 0:00 grep dutoit  
dutoit 19859 0.2 0.6 812 540 pts/6 0 15:38:47 0:00 sort
```



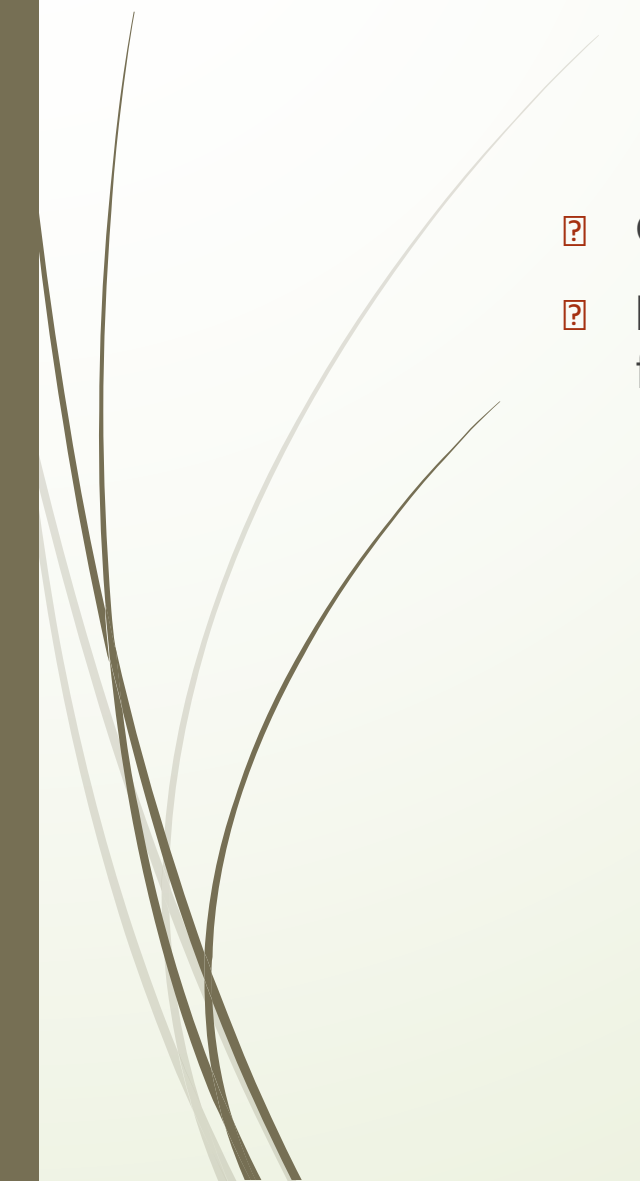


System Design Activities: From Objects to Subsystems

- ❑ Reviewing the Analysis Model
 - ❑ Identifying Design Goals
 - ❑ Identifying Subsystems
- 



Reviewing the analysis model

- ❑ Complete and detailed study of Analysis model is done.
 - ❑ In addition, during requirements elicitation, there would be a set of non functional requirements specified which has to be identified
- 




Identifying Design Goals



- ❑ The definition of design goals is the first step of system design.
- ❑ It identifies the qualities that our system should focus on.
- ❑ Many design goals can be inferred from the nonfunctional requirements or from the application domain. Others will have to be elicited from the client.
- ❑ It is, however, necessary to state them explicitly such that every important design decision can be made consistently following the same set of criteria.



Desirable qualities to identify design goal

- ❑ *Performance*
 - ❑ *Dependability*
 - ❑ *Cost*
 - ❑ *Maintenance*
 - ❑ *end user criteria.*
- 



Performance Criteria



Design criterion	Definition
Response time	How soon is a user request acknowledged after the request has been issued?
Throughput	How many tasks can the system accomplish in a fixed period of time?
Memory	How much space is required for the system to run?



Dependability criteria



Design criterion	Definition
Robustness	Ability to survive invalid user input
Reliability	Difference between specified and observed behavior
Availability	Percentage of time that system can be used to accomplish normal tasks
Fault tolerance	Ability to operate under erroneous conditions
Security	Ability to withstand malicious attacks
Safety	Ability to avoid endangering human lives, even in the presence of errors and failures



Cost criteria



Design criterion	Definition
Development cost	Cost of developing the initial system
Deployment cost	Cost of installing the system and training the users
Upgrade cost	Cost of translating data from the previous system. This criteria results in backward compatibility requirements
Maintenance cost	Cost required for bug fixes and enhancements to the system
Administration cost	Cost required to administer the system



Cost criteria



Design criterion	Definition
Development cost	Cost of developing the initial system
Deployment cost	Cost of installing the system and training the users
Upgrade cost	Cost of translating data from the previous system. This criteria results in backward compatibility requirements
Maintenance cost	Cost required for bug fixes and enhancements to the system
Administration cost	Cost required to administer the system



Maintenance criteria

Design criterion	Definition
Extensibility	How easy is it to add functionality or new classes to the system?
Modifiability	How easy is it to change the functionality of the system?
Adaptability	How easy is it to port the system to different application domains?
Portability	How easy is it to port the system to different platforms?
Readability	How easy is it to understand the system from reading the code?
Traceability of requirements	How easy is it to map the code to specific requirements?



End user criteria

Design criterion	Definition
Utility	How well does the system support the work of the user?
Usability	How easy is it for the user to use the system?

Design goal trade-offs


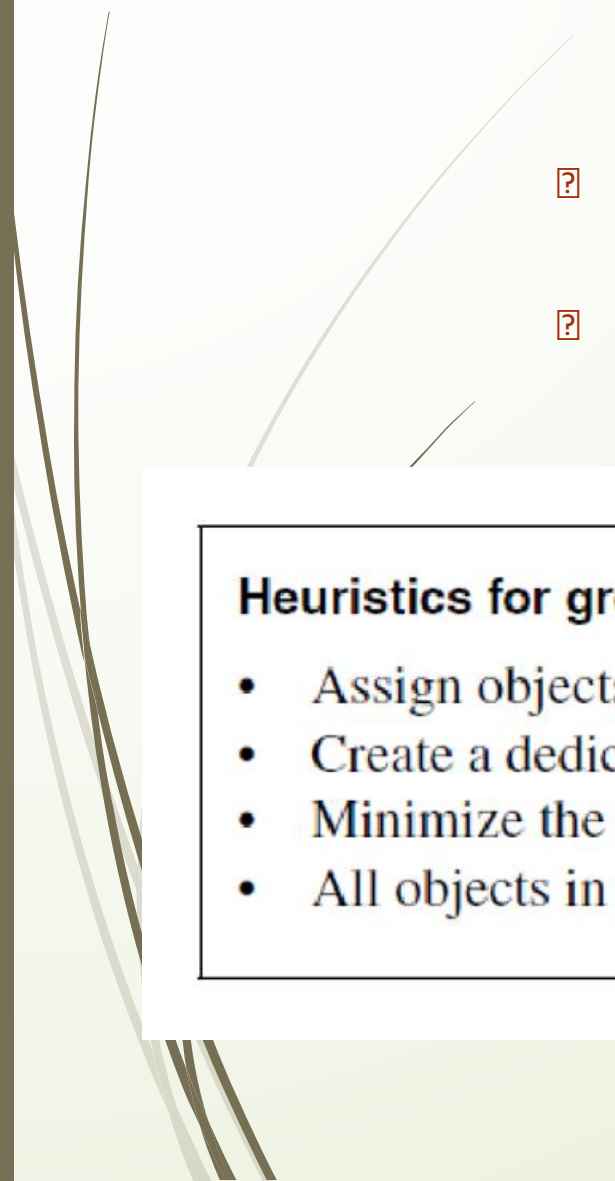
Trade-off	Rationale
Space vs. speed	If the software does not meet response time or throughput requirements, more memory can be expended to speed up the software (e.g., caching, more redundancy). If the software does not meet memory space constraints, data can be compressed at the cost of speed.
Delivery time vs. functionality	If development runs behind schedule, a project manager can deliver less functionality than specified on time, or deliver the full functionality at a later time. Contract software usually puts more emphasis on functionality, whereas off-the-shelf software projects put more emphasis on delivery date.
Delivery time vs. quality	If testing runs behind schedule, a project manager can deliver the software on time with known bugs (and possibly provide a later patch to fix any serious bugs), or deliver the software later with fewer bugs.
Delivery time vs. staffing	If development runs behind schedule, a project manager can add resources to the project to increase productivity. In most cases, this option is only available early in the project: adding resources usually decreases productivity while new personnel are trained or brought up to date. Note that adding resources will also raise the cost of development.



Identifying Subsystems




- ❑ Finding subsystems during system design is similar to finding objects during analysis.
- ❑ Moreover, subsystem decomposition is constantly revised whenever new issues are addressed
- ❑ several subsystems are merged into one subsystem, a complex subsystem is split into parts, and some subsystems are added to address new functionality.
- ❑ The first iterations over subsystem decomposition can introduce drastic changes in the system design model. These are often best handled through brainstorming.

- 
- 
- ❓ The initial subsystem decomposition should be derived from the functional requirements.
 - ❓ Another heuristic for subsystem identification is to keep functionally related objects together.

Heuristics for grouping objects into subsystems

- Assign objects identified in one use case into the same subsystem.
- Create a dedicated subsystem for objects used for moving data among subsystems.
- Minimize the number of associations crossing subsystem boundaries.
- All objects in the same subsystem should be functionally related.



Encapsulating subsystems with the Facade design pattern

- ❑ The **Facade design pattern** allows us to further reduce dependencies between classes by encapsulating a subsystem with a simple, unified interface
- ❑ The façade provides access only to the public services offered by the subsystem and hides all other details, effectively reducing coupling between subsystems.
- ❑ Subsystems identified during the initial subsystem decomposition often result from grouping several functionally related classes. These subsystems are good candidates for the Facade design pattern and should be encapsulated under one class.