CHAPTER 2 - SOFTWARE DESIGN STRATEGIES & METHODS

LECTURE OUTLINE

- General Strategies
- Function Oriented Design (Structured Design)
- Object-Oriented Design
- Component-Based Design (CBD)
- Data Structure-Centered Design
- Other Methods

RECAP…

- Software design consists of two **main** activities that fit between software requirements analysis and software construction:
    - Software architectural design (sometimes called top-level design)
        - Describing software’s top-level structure and organization and identifying the various components.
    - Software detailed design
        - Describing each component sufficiently to allow for its construction.

RECAP…

- Software design plays an important role in developing software:
  - It allows software engineers to produce various models that form a kind of **blueprint of the solution** to be implemented.
  - We can analyze and evaluate these models to determine whether or not they will allow us to **fulfill the various requirements.**
  - We can also examine and evaluate **various alternative solutions and trade-offs.**
  - Finally, we can use the resulting models to plan the subsequent development activities, in addition to using them as input and the **starting point of construction and testing.**

RECAP…

- Software design process revolves around decomposing of the system into smaller and simpler units and then systematically integrates these units to achieve the desired results.
- Software design is a process to conceptualize the software requirements into software implementation
    - Deriving a solution which satisfies software requirements.

SOFTWARE DESIGN STRATEGIES

- While the software is being conceptualized from the software requirements into software implementation, a plan is chalked out to find the **best possible design for implementing** the intended solution.
- There are multiple variants of software design strategies to guide the design process.

GENERAL STRATEGIES

- Some often-cited examples of general strategies useful in the design process are:
  - divide-and-conquer and stepwise refinement
  - top-down vs. bottom-up strategies
  - data abstraction and information hiding
  - use of heuristics (such as “trial-by-error” or “using a rule-of-thumb”)
  - use of patterns and pattern languages
  - use of an iterative and incremental approach.

FUNCTION-ORIENTED (STRUCTURED) DESIGN

FUNCTION-ORIENTED (STRUCTURED) DESIGN

- The system is comprised of many smaller sub-systems known as **functions.** These functions are capable of performing significant task in the system. The system is considered as top view of all functions.
- Function-oriented design mostly based on “divide and conquer” methodology is used.
  - “divide and conquer” strategy where a problem is broken into several small problems and each small problem is individually solved until the whole problem is solved using an iterative top-down approach.

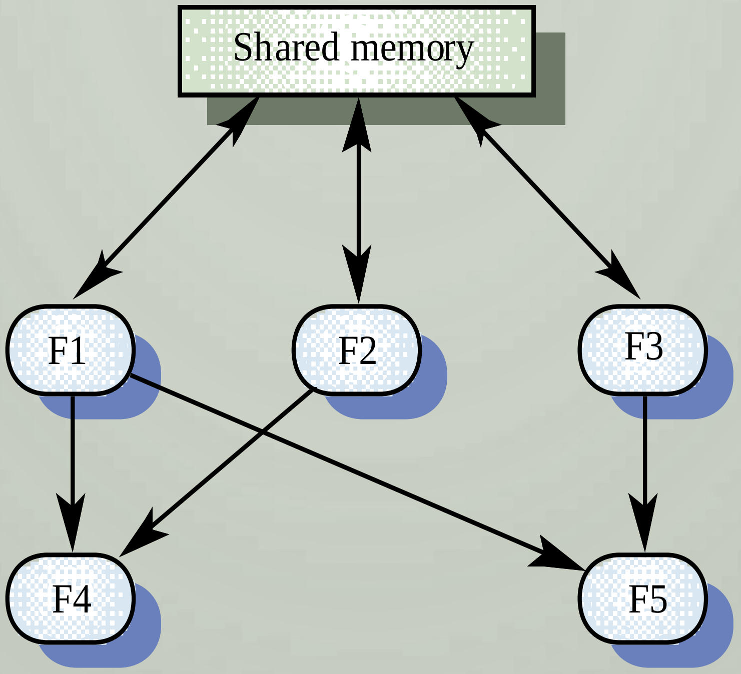
FUNCTION-ORIENTED (STRUCTURED) DESIGN

- This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation.
- These functional modules can share information among themselves by means of information passing and using information available globally.

FUNCTION-ORIENTED (STRUCTURED) DESIGN

- Function-oriented design works well where the system state does not matter and program/functions work on input rather than on a state.
- In this design approach, the system state, that is the data maintained by the system, is centralized and is shared by these functions.
- Conceals the details of an algorithm in a function but system state information is not hidden.

FUNCTION-ORIENTED (STRUCTURED) DESIGN



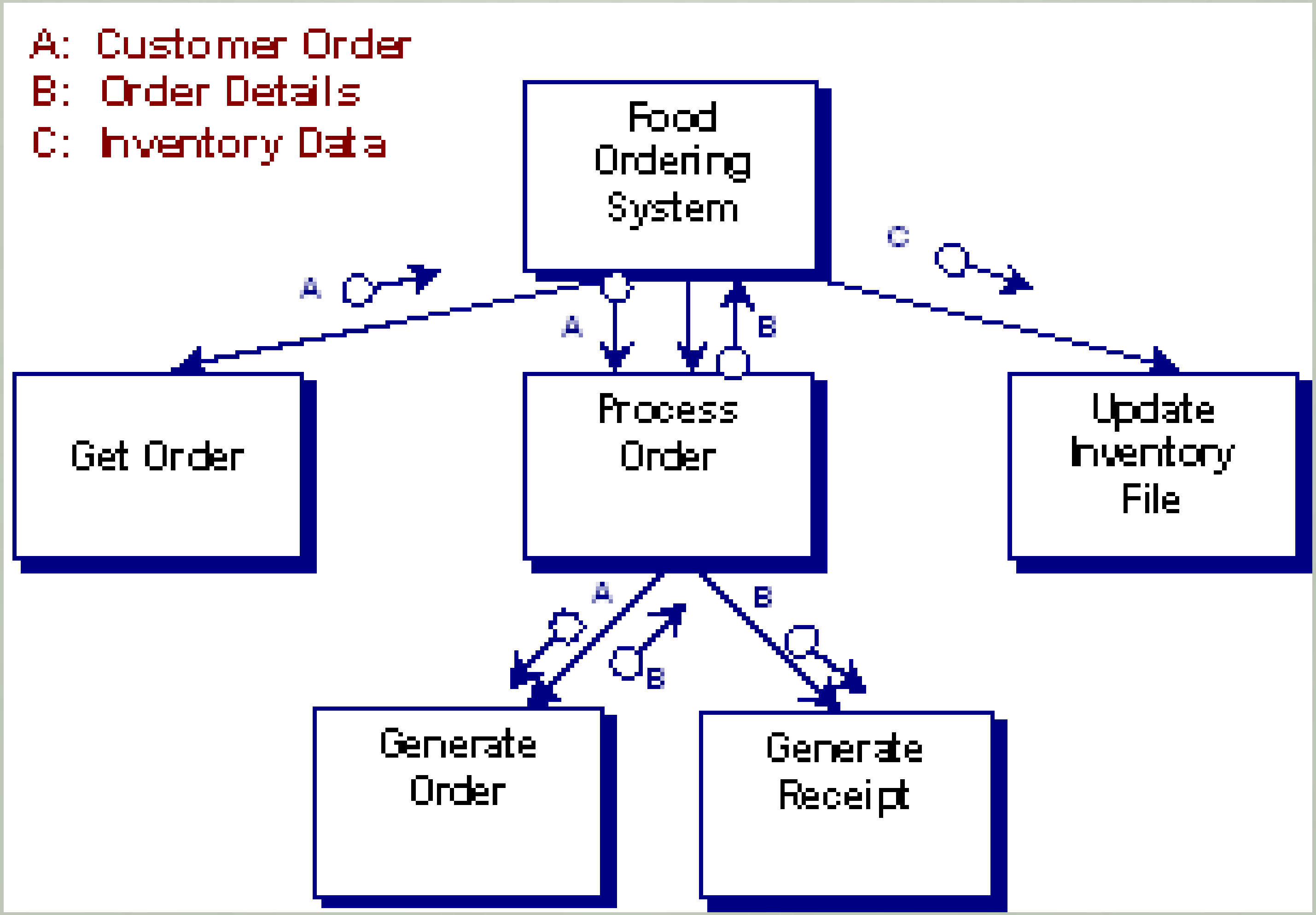
FUNCTION-ORIENTED (STRUCTURED) DESIGN

- Generally used after structured analysis, thus producing, among other things, data flow diagrams and associated process descriptions.
- Typically employed after structured analysis, where the main purpose is to derive a **structure chart** from data flow diagrams (DFDs).

FUNCTION-ORIENTED (STRUCTURED) DESIGN

- Structure Chart
  - Used in Architectural Design
  - Shows decomposition of program into modules
  - Module corresponds to
    - Procedure
    - Function
    - Information Hiding Module
  - Shows module calling structure
  - Defines interfaces between modules
    - Input parameters
    - Output parameters

EXAMPLE STRUCTURE CHART



FUNCTION-ORIENTED (STRUCTURED) DESIGN

- The activities of this strategy:
  - Data-flow design
    - Model the data processing in the system using data-flow diagrams (DFDs)
  - Structural decomposition
    - Model how functions are decomposed to sub-functions using graphical structure charts
  - Detailed design description
    - The entities in the design and their interfaces are described in detail.

FUNCTION-ORIENTED (STRUCTURED) DESIGN

- Pros and Cons
  - modules are highly functional
  - best suited when state information is not pervasive
  - data decisions must be made earlier
  - changes in data ripple through entire structure
  - little chance for reusability

OBJECT-ORIENTED DESIGN

OBJECT-ORIENTED DESIGN

- Object-oriented design works around the entities and their characteristics instead of functions involved in the software system.
- The system is decomposed into a set of objects that cooperate and coordinate with each other to implement the desired functionality.
- The system state is decentralized and each object is held responsible for maintaining its own state. That is, the responsibility of marinating the system state is distributed and this responsibility is delegated to individual objects.

OBJECT-ORIENTED DESIGN

- Based on the idea of information hiding.
- System is viewed as a set of interacting objects, with their own private state.
- Objects communicate by calling on services offered by other objects rather than sharing variables. This reduces the overall system coupling.

OBJECT-ORIENTED DESIGN

- Concepts of Object-oriented Design:
  - Objects - All entities involved in the solution design.
  - Classes - A class is a generalized description of an object
  - Encapsulation – information hiding
  - Inheritance - create generalized classes from specific ones.
  - Polymorphism - methods performing similar tasks but vary in arguments, can be assigned same name.

COMPONENT-BASED DESIGN

COMPONENT-BASED DESIGN (CBD)

- An approach to software development that relies on reuse using reusable software "**components**".
- It emerged from the failure of Object-oriented development to support effective reuse. Single object classes are too detailed and specific.
- Object-oriented design starts with either an inheritance hierarchy or an interface based design.
- Component-based design also uses Object-oriented concepts, but instead of inheritance (or interfaces), utilizes **object composition.**

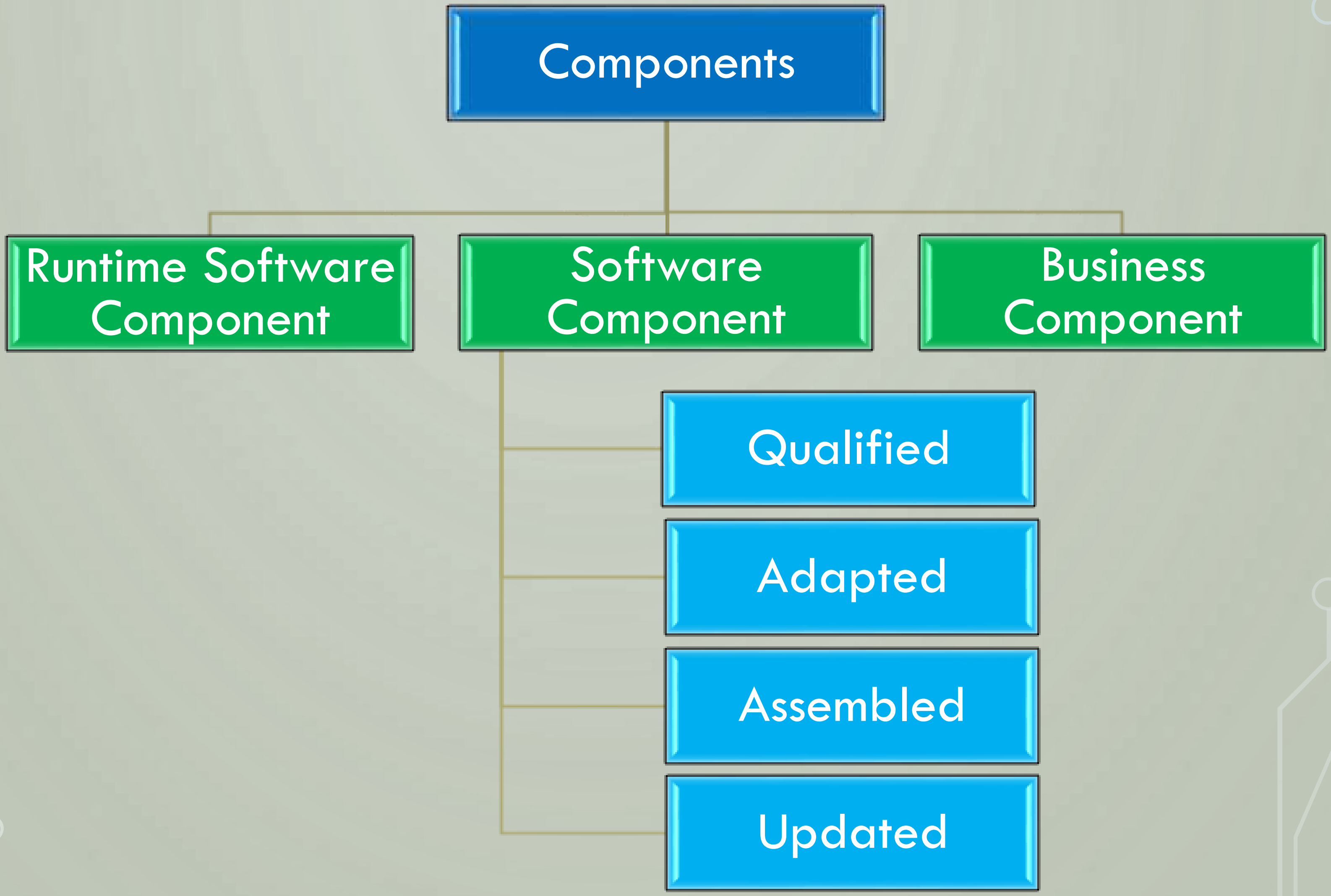
COMPONENTS

- A software component is an **independent unit,** having well-defined interfaces and dependencies that can be composed and deployed independently .
- An **independent executable entity** that can be made up of one or more executable objects such as:
  - A software package
  - A web service
  - A web resource or
  - A module that encapsulates a set of related functions (or data).

COMPONENTS

- A component is a higher-level concept, usually incorporating more than one class. Thus, we identify all components and their interfaces first instead of identifying classes and their relationships.
- Component does not have to be compiled before it is used with other components.
- Components can range in size from simple functions to entire application systems.
- Component based design addresses issues related to providing, developing, and integrating such components in order to improve reuse.

COMPONENTS


Components
Runtime Software Component
Software Component
Business Component
Qualified
Adapted
Assembled
Updated

COMPONENTS

- Possesses well-defined functionality, provides a set of well-defined interfaces and clearly specifies any explicit dependencies.

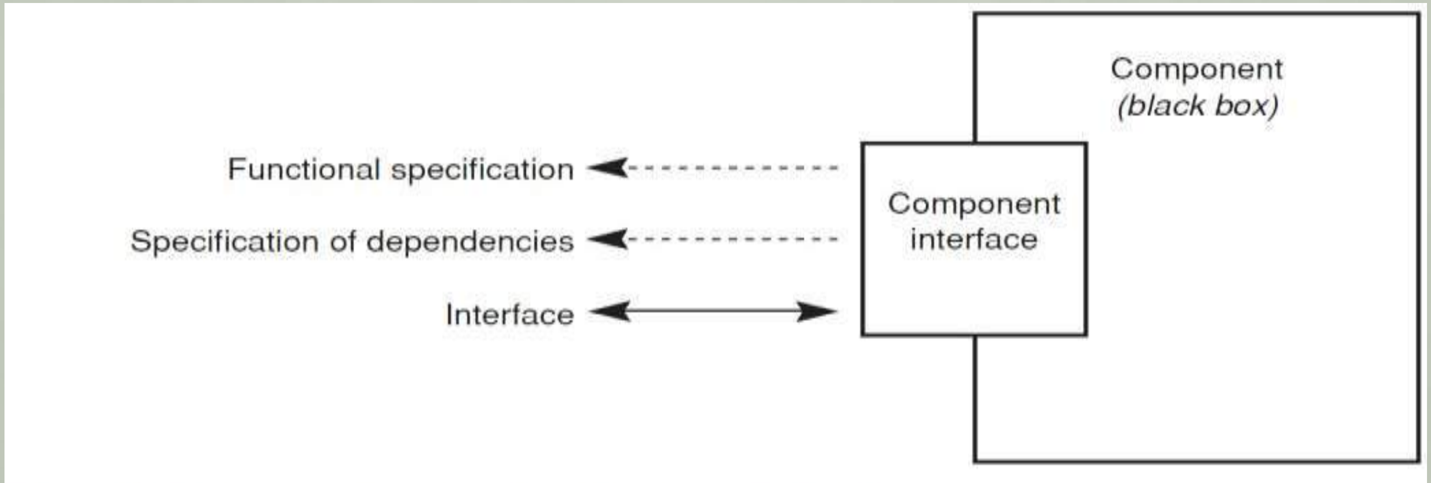


Figure 17.1 Characteristics of a software component.

- It is a reusable block of software.

COMPONENT CHARACTERISTICS

- Encapsulation
  - Its inner workings must be hidden from the outside world that it exists in.
- Reusable
  - In different application contexts, so that developers can reduce the effort in developing new applications
- Replaceable
  - Able to replace, or be replaced by, other existing components.
  - The new component must at least be able to provide the same services as the old component

COMPONENT CHARACTERISTICS

- Interoperable
  - Its services must be accessible on any platform
- Extensible
  - A component can be extended from existing component to provide new behaviour.
- Independent
  - Components are designed to have minimal dependencies on other components.
  - It should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a ‘requires’ interface specification.

COMPONENT CHARACTERISTICS

- Composable
  - For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.
- Standardized
  - Conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.
- Not context specific
  - Components are designed to operate in different environments and contexts.

COMPONENT CHARACTERISTICS

- Deployable
  - To be deployable, a component has to be self-contained.
  - It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model.
  - This usually means that the component is binary and does not have to be compiled before it is deployed. If a component is implemented as a service, it does not have to be deployed by a user of a component. Rather, it is deployed by the service provider.
- Documented
  - Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.

COMPONENT INTERFACES

- Components provide a service without regard to where the component is executing or its programming language
  - The component interface is published and all interactions are through the published interface
- It should be a black-box, with only its interface to define:
  - What services it requires,
  - What services it can provide
  - How the provided services can be accessed by other (compatible) components
  - Where the component can be deployed

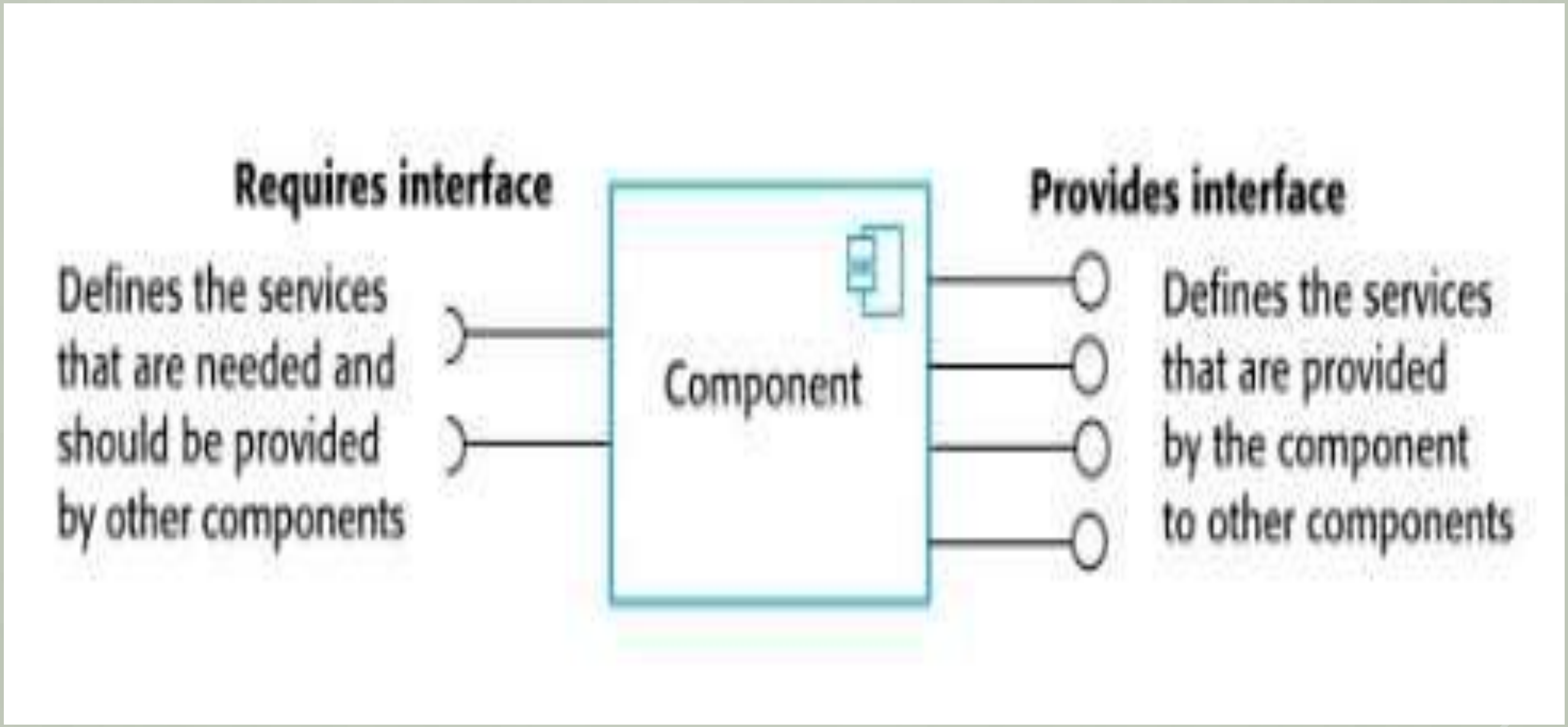
COMPONENT INTERFACES

- A useful interface which the outside world can interpret so that other components know what services it exports.
- The interface should also specify what the component needs to import in order to carry out its operations.
- The interface must conform to the component model (the accepted standard used in the world in which the component works).

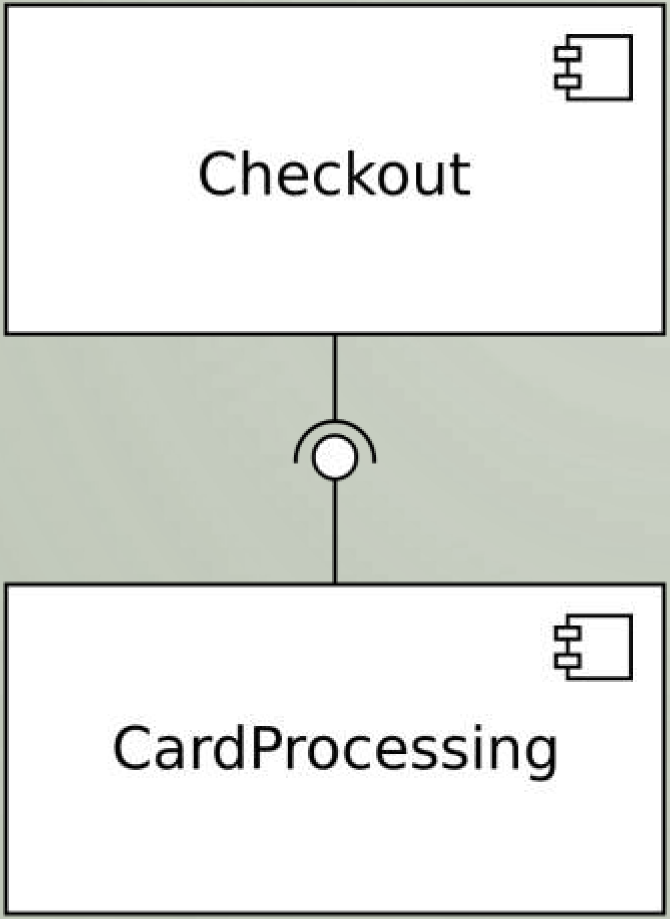
COMPONENT INTERFACES

- Provides interface
  - Defines the services that are provided by the component to other components.
  - This interface, essentially, is the component API. It defines the methods that can be called by a user of the component.
- Requires interface
  - Defines the services that specifies what services must be made available for the component to execute as specified.
  - This does not compromise the independence or deployability of a component because the 'requires' interface does not define how these services should be provided.

COMPONENT INTERFACES

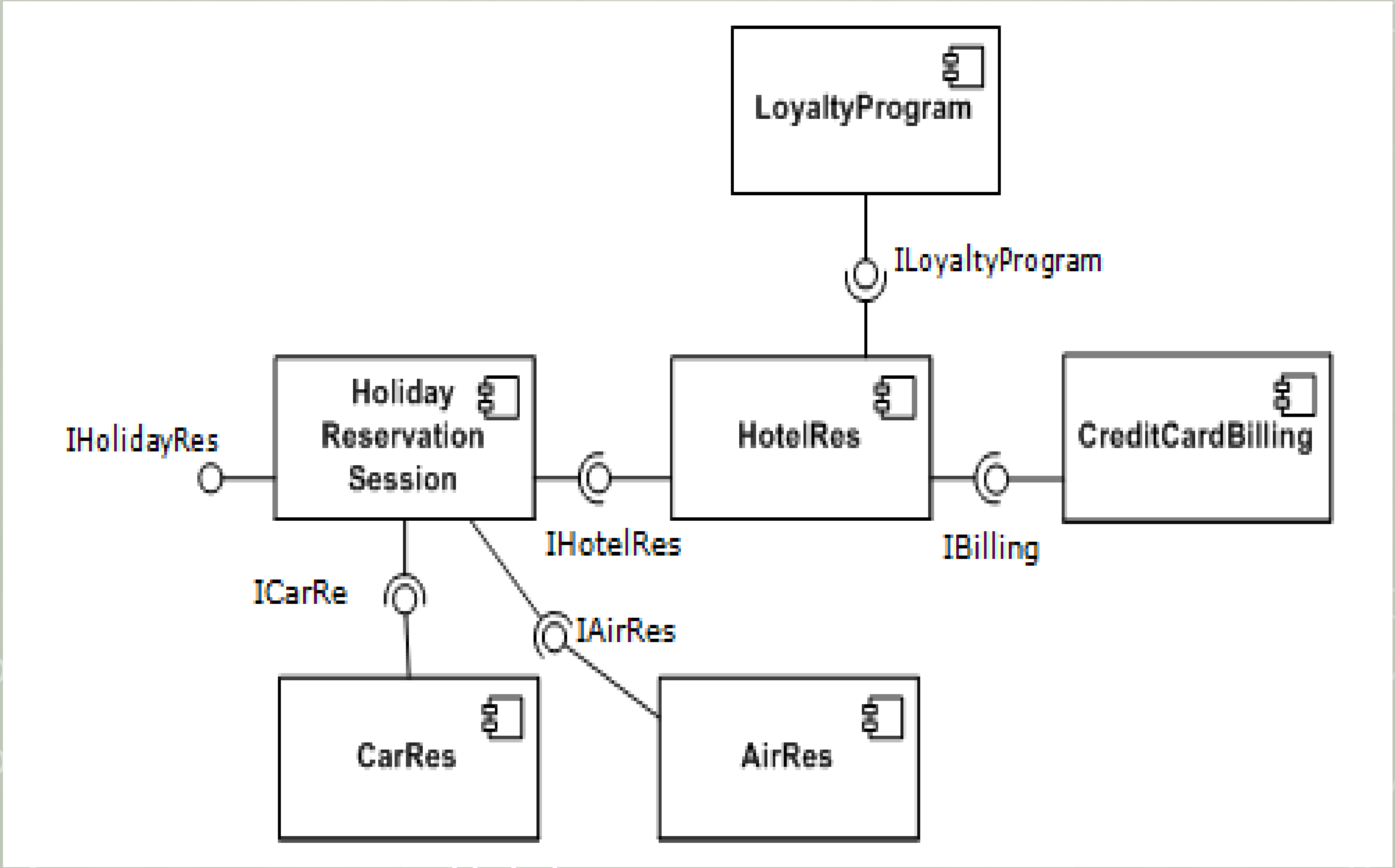

Requires interface
Defines the services that are needed and should be provided by other components
Component
Provides interface
Defines the services that are provided by the component to other components

COMPONENT INTERFACES



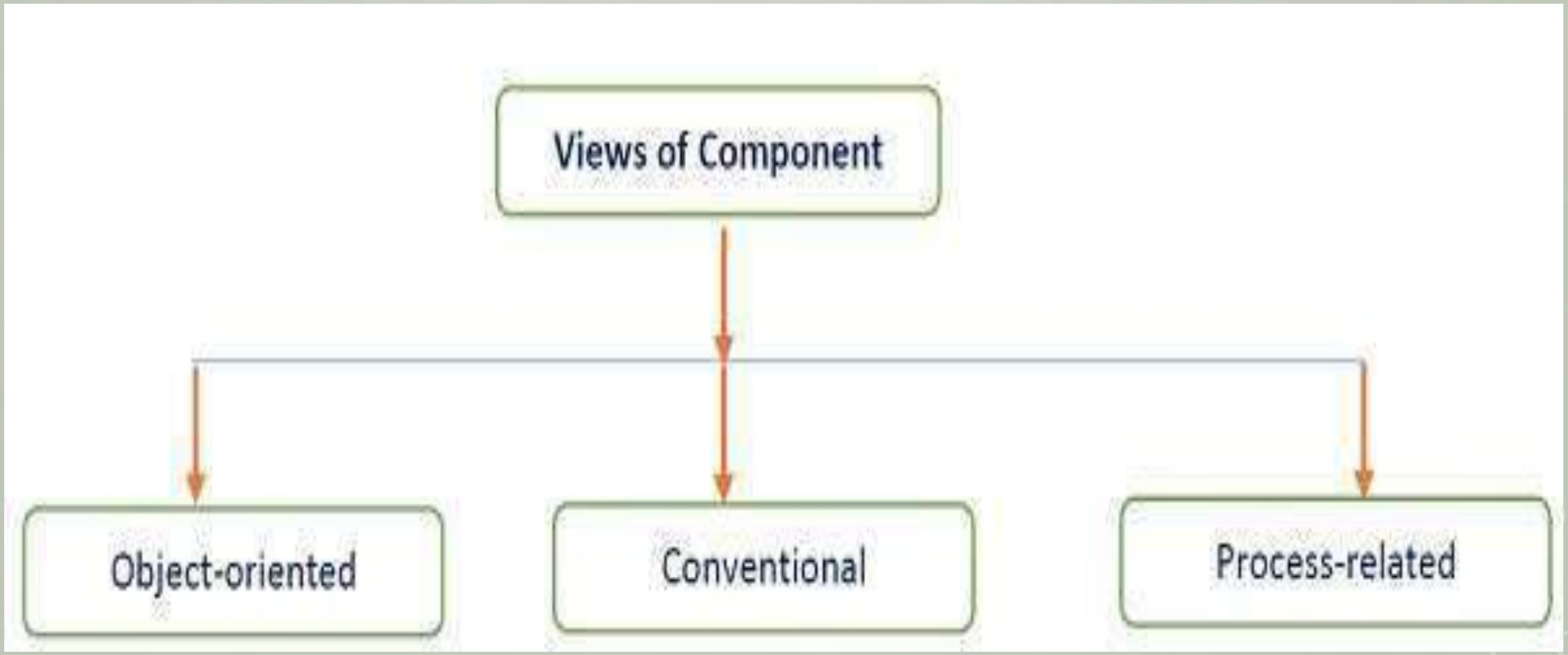
- The **checkout component,** responsible for facilitating the customer's order, *requires* the **card processing component** to charge the customer's credit/debit card (functionality that the latter *provides*).

COMPONENT INTERFACES EXAMPLE



Holiday-reservation system

COMPONENT VIEWS



COMPONENT VIEWS

- **Object-oriented View**
  - A component is viewed as a set of one or more cooperating classes.
  - Each problem domain class (analysis) and infrastructure class (design) are explained to identify all attributes and operations that apply to its implementation.
  - It also involves defining the interfaces that enable classes to communicate and cooperate.

COMPONENT VIEWS

- **Conventional view**
  - It is viewed as a functional element or a module of a program that integrates the processing logic, the internal data structures that are required to implement the processing logic and an interface that enables the component to be invoked and data to be passed to it.

COMPONENT VIEWS

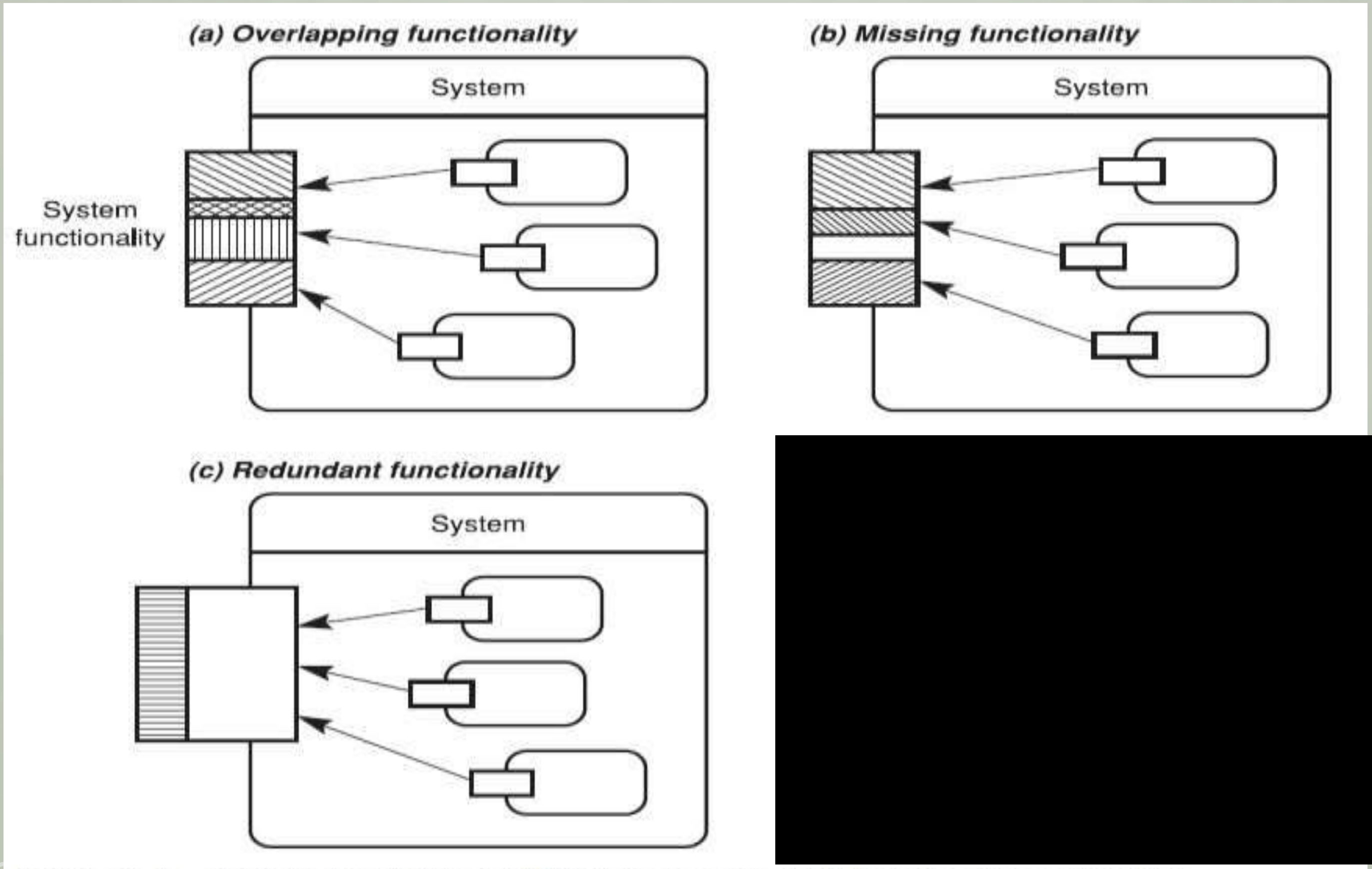
- **Process-related view**
  - In this view, instead of creating each component from scratch, the system is building from existing components maintained in a library. As the software architecture is formulated, components are selected from the library and used to populate the architecture.
  - A user interface (UI) component includes grids, buttons referred as controls, and utility components expose a specific subset of functions used in other components.

COMPONENT VIEWS

- Other common types of components are those that are resource intensive, not frequently accessed, and must be activated using the just-in-time (JIT) approach.
- Many components are invisible which are distributed in enterprise business applications and internet web applications such as Enterprise JavaBean (EJB), .NET components, and CORBA components.

CBD FUNCTIONALITY PROBLEMS

- Arise during component integration.


(a) Overlapping functionality
System
System functionality
(b) Missing functionality
System
(c) Redundant functionality
System

[redacted]

Figure 17.4 Illustrations of the problem that can arise during component integration.

CBD FUNCTIONALITY PROBLEMS

- **Overlapping Functionality**
  - This occurs where two or more components are able to provide a particular system function.
  - The designer’s task is to determine which one should perform the task, and how to ensure that it is only performed by the chosen component.
  - From a design point of view, it is particularly undesirable to have the same function being provided by different elements (possibly with different ranges and limits), and this also creates a problem for system maintenance and upgrading.

CBD FUNCTIONALITY PROBLEMS

- **Missing Functionality**
  - The problem arises when the total functionality available from the system is less than that required.
  - The solution to this is fairly simple (find another component or, possibly, create one).
  - So the problem is chiefly a short-term one of identification.

CBD FUNCTIONALITY PROBLEMS

- **Redundant Functionality**
  - Components may well provide services over and above those that are the basis for choosing those components.
  - The designer’s choice is then either to incorporate this added functionality (with possible adverse consequences should this lead to overlapping functionality at a higher level of integration), **or** to find some way to isolate and disable the unwanted functions.
  - The latter strategy is likely to be the sounder one, but may be quite difficult to achieve.

DATA-STRUCTURE CENTERED DESIGN

DATA-STRUCTURE CENTERED DESIGN

- A structured software design (for example, Jackson, Warnier-Orr) technique wherein the architecture of a system is derived from analysis of the structure of the data sets with which the system must deal rather than from the function it performs.
- The software engineer first describes the input and output data structures (using Jackson's structure diagrams, for instance) and then develops the program's control structure based on these data structure diagrams.
- Various heuristics have been proposed to deal with special cases—for example, when there is a mismatch between the input and output structures

ASPECT-ORIENTED SOFTWARE DEVELOPMENT

ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD)

- An emerging software development technology that seeks new modularizations of software systems in order to isolate secondary or supporting functions from the main program's business logic.
- Allows multiple concerns to be expressed separately and automatically unified into working systems.
- Used in conjunction with other approaches - normally object-oriented software engineering.
  - To improve their expressiveness and utility.

ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD)

- Focuses on the identification, specification and representation of **cross-cutting concerns** and their modularization into separate functional units as well as their automated composition into a working system.
- An approach to software development based around a new type of abstraction - an **aspect**.

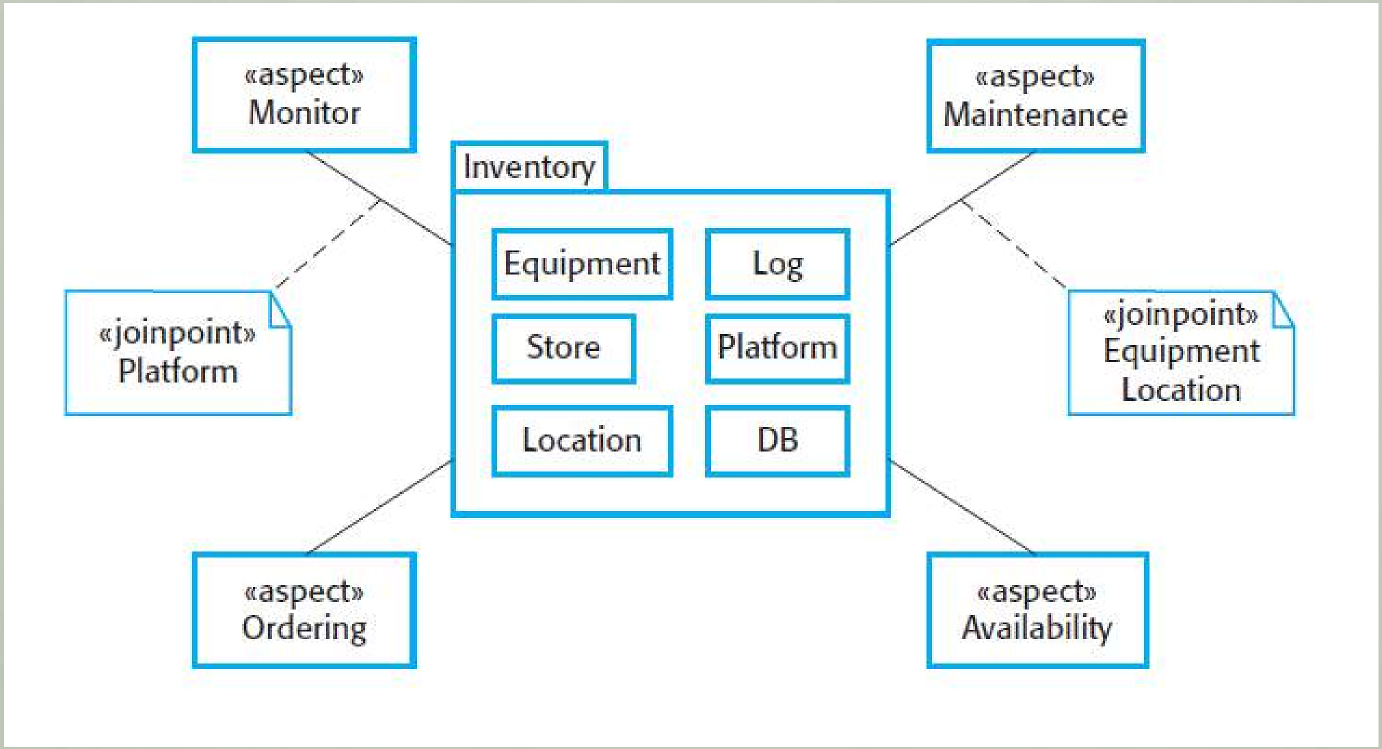
ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD)

- Aspect is a module or class-like structure that encapsulate cross-cut concerns:
  - Can be static or dynamic
  - Can have fields and methods as members
  - Can be abstract or not
  - Can be instantiated
  - Can have pointcuts, advice, and inter-type declarations
  - May be ‘privileged’ to access private members of other types

ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD)

- Aspect:
  - encapsulate functionality that cross-cuts and co-exists with other functionality.
  - include a definition of where they should be included in a program as well as code implementing the cross-cutting concern.
  - Express these concerns and automatically incorporates them into a system.
  - Enhances the ability to express separation of these concerns.
  - Leads a well-designed, maintainable software system.

ASPECT-ORIENTED SOFTWARE DEVELOPMENT (AOSD)



CONCERNS

- An area of interest or focus in a system.
- The primary criteria for decomposing software into smaller, more manageable and comprehensible parts.
- A concern is a particular set of information that has an **affect** on the code of a computer program.
- It can be as general as the details of database interaction or as specific as performing a primitive calculation, depending on the level of conversation between developers and the program being discussed.

CONCERNS

- A typical system may consist of several kind of concerns including:
  - Business logic
  - Performance
  - Data persistence
  - Logging and debugging
  - Authentication
  - Security
  - Multi-threaded safety
  - Error-checking
  - Etc.

CONCERNS

- Quality concerns are the different aspects of software functionality. Examples of concerns are performance, security, specific functionality, etc.

- Concerns are **not** program issues but reflect the system requirements and the priorities of the system stakeholders.
- For instance, the "business logic" of software is a concern, and the interface through which a person uses this logic is another.

CONCERNS

- **Core concerns**
  - Are the functional concerns that relate to the primary purpose of a system.
  - Can be:
    - Core business related concern
    - Domain specific concern
    - Functional requirement
    - Can be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API)
- **Secondary concerns**
  - Are functional concerns that reflect non-functional and Quality of Service (QoS) requirements.

STAKEHOLDER CONCERNS

- Functional concerns which are related to specific functionality to be included in a system.
- Quality of service (QoS) concerns which are related to the non-functional behaviour of a system.
- Policy concerns which are related to the overall policies that govern the use of the system.
- System concerns which are related to attributes of the system as a whole such as its maintainability or its configurability

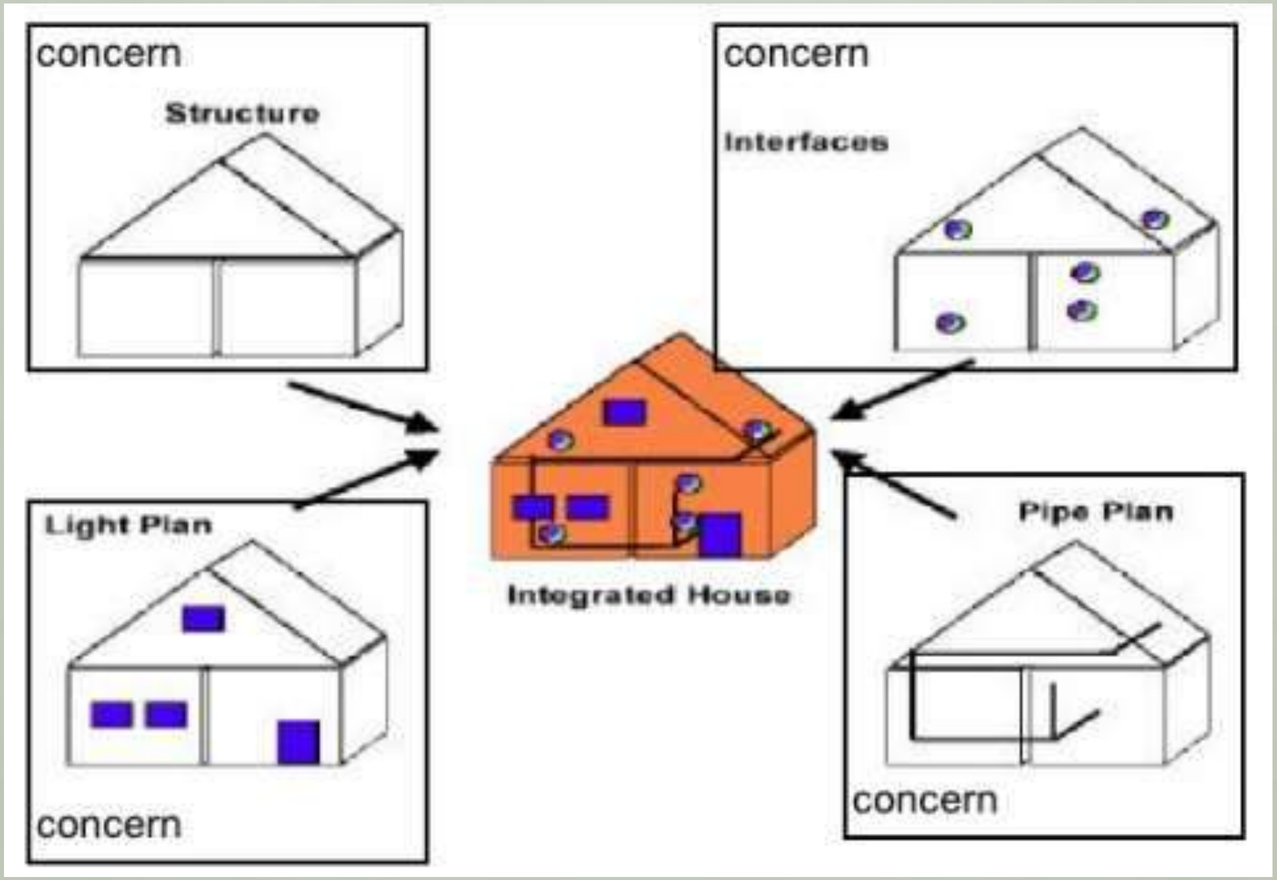
STAKEHOLDER CONCERNS

- Organisational concerns which are related to organisational goals and priorities such as producing a system within budget, making use of existing software assets or maintaining the reputation of an organisation.

SEPARATION OF CONCERNS

- When concerns are well-separated, individual sections can be reused, as well as developed and updated independently.
- The **separation of concerns** is keeping the code for each of these concerns separate. Changing the interface should not require changing the business logic code, and vice versa.

SEPARATION OF CONCERNS



SEPARATION OF CONCERNS

- Focused on breaking down software into distinct parts that overlap in functionality as little as possible.
- The principle of separation of concerns states that software should be organized so that each program element does one thing and one thing only.
- Goal – to make subsystems more self-contained, so that different concerns are addressed by different subsystems.
- Each program element should therefore be understandable without reference to other elements.

SEPARATION OF CONCERNS

- Program abstractions (subroutines, procedures, objects, etc.) support the separation of concerns.
- By reflecting the separation of concerns in a program, there is clear traceability from requirements to implementation.
- Model-View-Controller (MVC) design is an excellent example of separating these concerns for better software maintainability.

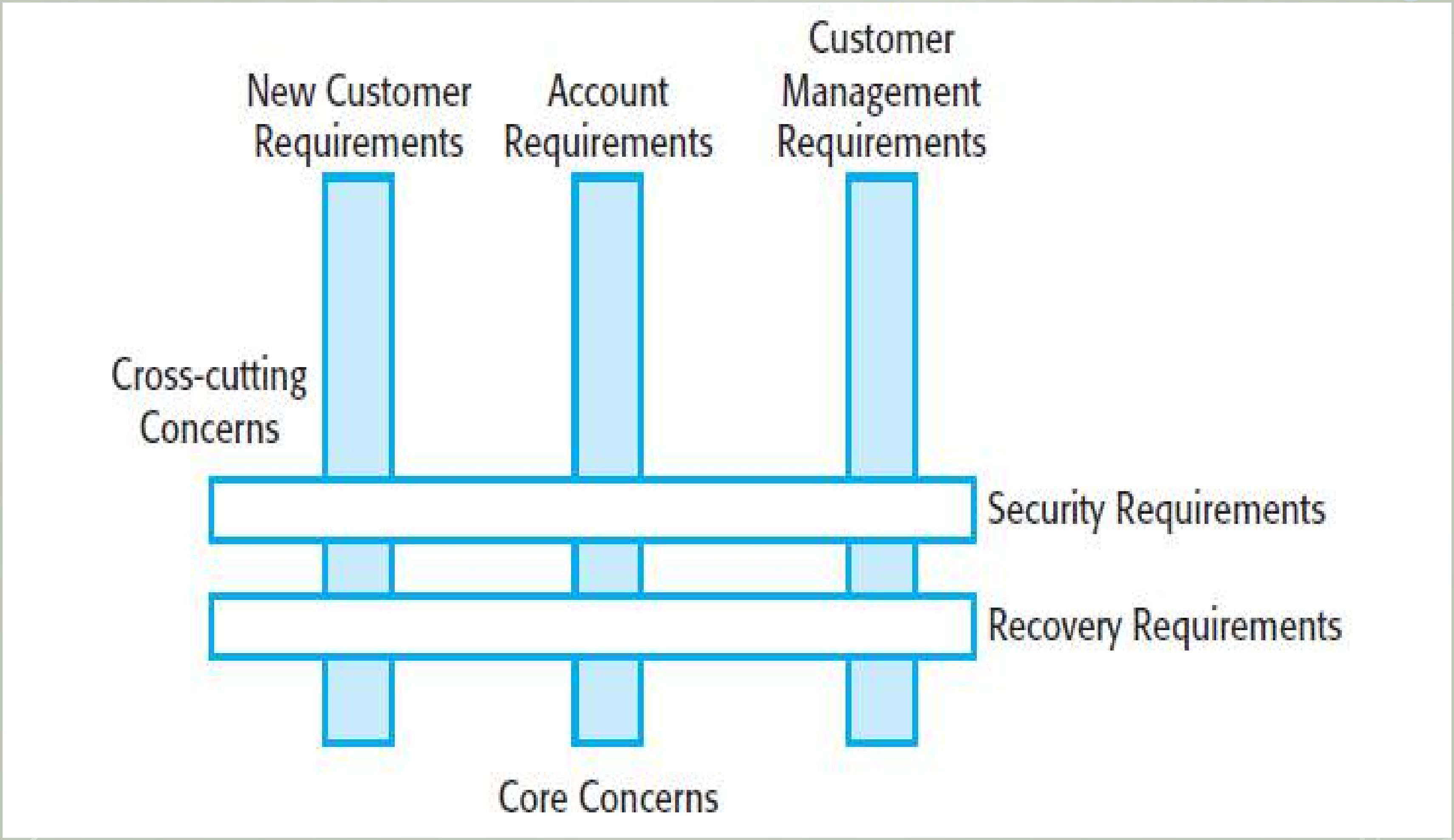
SEPARATION OF CONCERNS

- Allows to:
  - Reduce software complexity and improve comprehensibility
  - Promote traceability within and across artifacts and throughout the lifecycle
  - Limit the impact of change, facilitating evolution and non-invasive adaptation and customization
  - Facilitate reuse
  - Simplify component integration.

CROSS-CUTTING CONCERNS

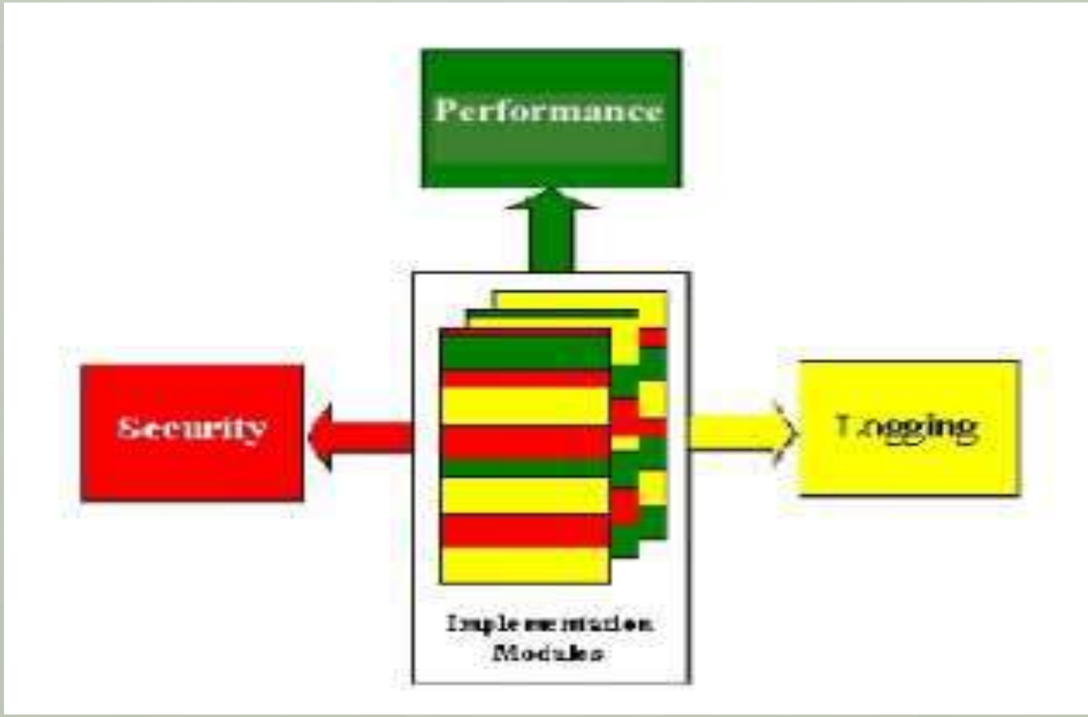
- Not every concerns fits into a component: cross-cutting.
- Cross-cutting concerns are concerns whose implementation cuts across a number of program components.
- This results in problems when changes to the concern have to be made - the code to be changed is not localized but is in different places across the system.
- Cross-cutting concerns lead to complexity.

CROSS-CUTTING CONCERNS



CROSS-CUTTING CONCERNS

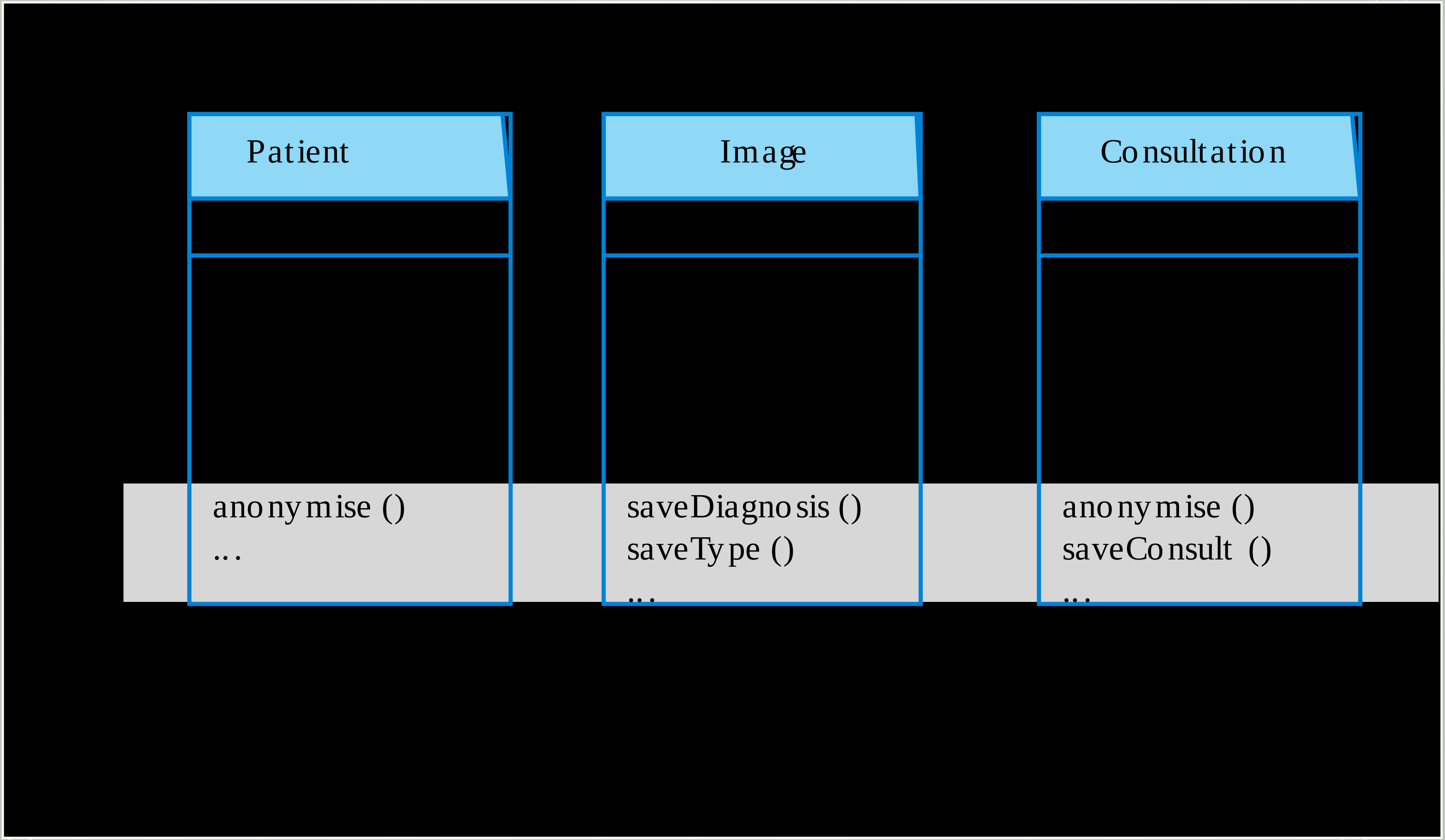
- Cross-cutting concerns lead to **scattering** and **tangling.**
- E.g.: Implementation modules becomes mess when we code these concerns.



SCATTERING

- When similar code is distributed throughout many program modules
  - Related implementations spread over many modules
- This differs from a component being used by many other components since it involves the risk of misuse at each point and of inconsistencies across all points.
- Changes to the implementation may require finding and editing all affected code.
- E.g. Implementation for authentication, contract checking and logging is not localized
  - Code spans over many methods of potentially many classes and packages aka code scattering

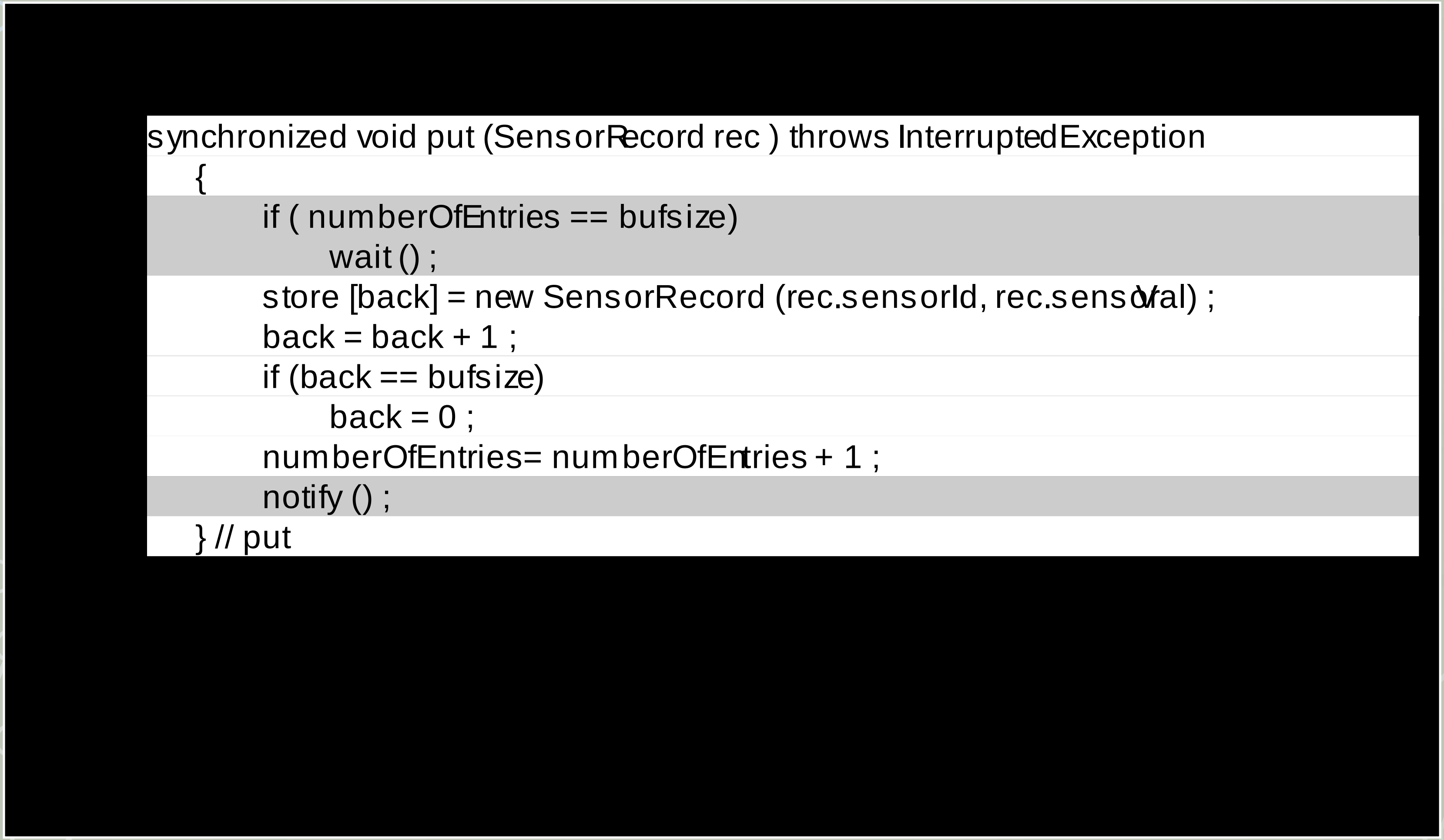
SCATTERING


Patient
anonymise ()
…
Image
saveDiagnosis ()
saveType ()
…
Consultation
anonymise ()
saveConsult ()
…

TANGLING

- Modules in a software system may simultaneously interact with several requirements.
- When two or more concerns are implemented in the same body of code or component, making it more difficult to understand.
- Changes to one implementation may cause unintended changes to other tangled concerns.
- E.g. Implementation of someOperations() does much more than performing some code functionality.
    - Contain code for more that one concerns aka code tangling.

TANGLING



```
synchronized void put (SensorRecord rec ) throws InterruptedException
    {
          if ( numberOfEntries == bufsize)
                wait () ;
          store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
          back = back + 1 ;
          if (back == bufsize)
                back = 0 ;
          numberOfEntries= numberOfEntries + 1 ;
          notify () ;
    } // put
```

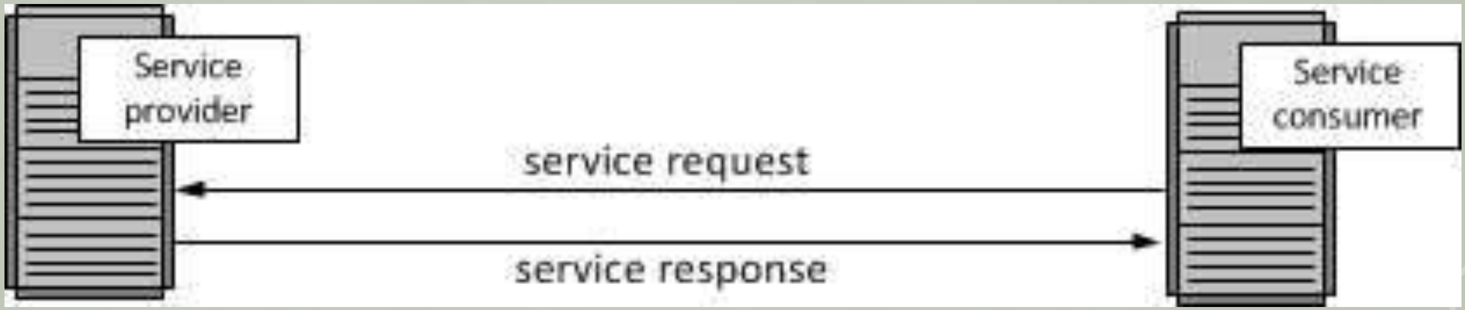
SERVICE-ORIENTED ARCHITECTURE

SERVICE-ORIENTED ARCHITECTURE (SOA)

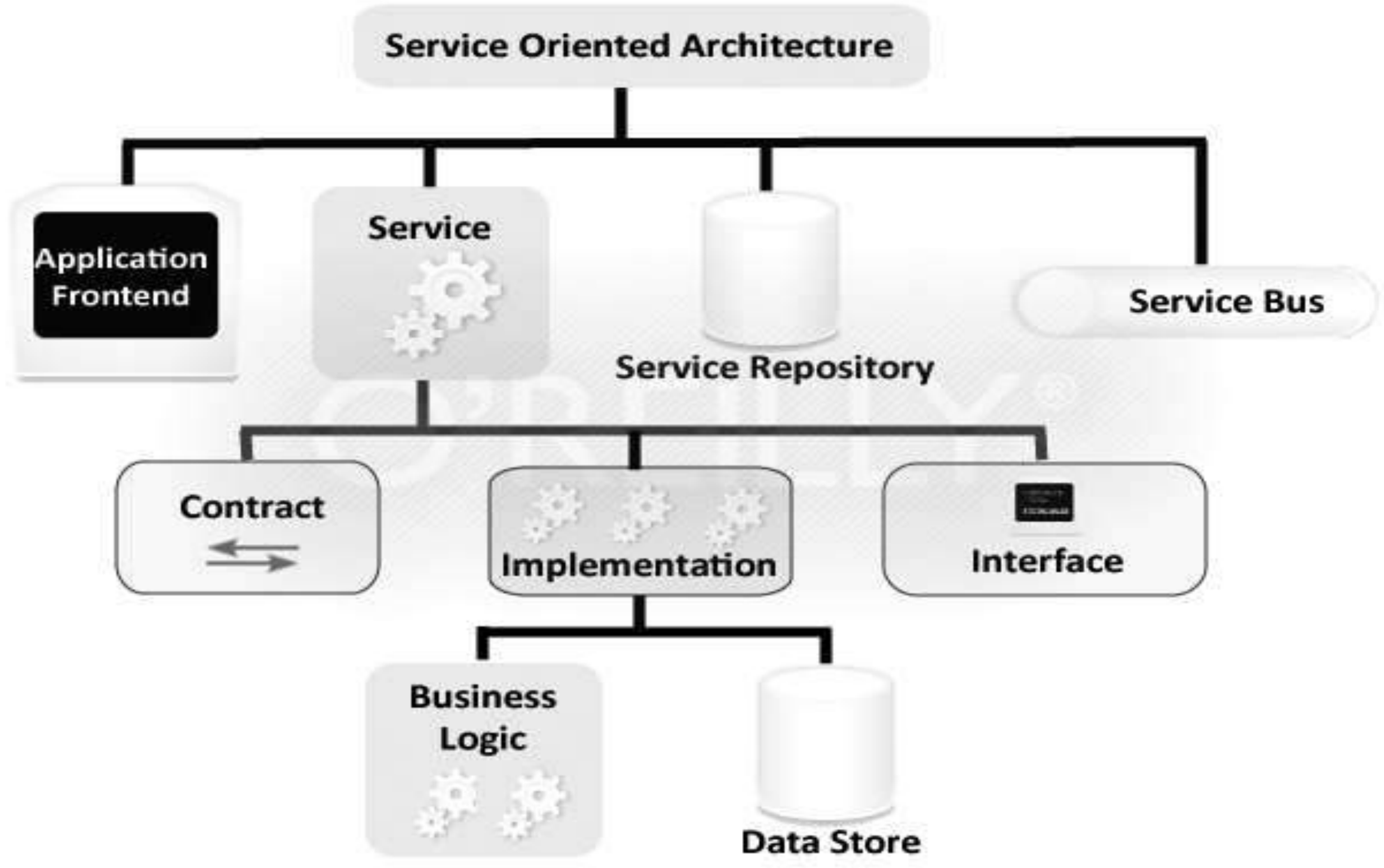
- A way to build distributed software using web services executed on distributed computers.
- Break software into *services* (a collection of services)
- A **service** is a function that is well-defined, self-contained, and does not depend on the context or state of other services.
- A service is the endpoint of a connection. Also, a service has some type of underlying computer system that supports the connection offered.
- Services are very loosely coupled and hide information about how they work.
- An application can string together many services to provide functionality.

SERVICE-ORIENTED ARCHITECTURE (SOA)

- These services communicate with each other - involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed.



SERVICE-ORIENTED ARCHITECTURE (SOA)



Layout of Service Oriented Architecture

SUMMARY

- This chapter presented the software design strategies such as Function-oriented (structured) design, Object-oriented design, Component-based design (CBD), Data-structure centered design, Aspect-oriented software development and Service-oriented architecture.