Software Engineering - Lecture 6
Software Design

Objectives

- To present the *basic activities* of an *object oriented design process.*
- To present the different *models* that may be used to *document an object-oriented design*;
- To explain the *benefits* of software reuse and some reuse *problems.*
- To explain how *reusable concepts* can be represented as *patterns.*
- To explain the concept of *application framework* as a set of reusable objects and how frameworks can be used in application development.
- To introduce *software product lines*, which are made of a common architecture and configurable, reusable components.
- To discuss *COTS*(commercial off-the-shelf) systems reuse by configuring and composing them.

Design

- **Design** – creative activity to identify *software components* and their *relationships*, based on customer's requirements. The result is the *design model* of the software to be developed.

- Implementation – process of *realizing* the design as a *program.*

Design

UML diagrams: - proper documentation for OO design

OBS: When developing a design, implementation issues are taken into account.

Special approaches :

In agile methods - informal sketches of the design; leave many design decisions to programmers.

In COTS based applications - the design process becomes concerned with how to use the configuration features of COTS system to deliver the requirements of the system to be developed.

Object-oriented development

Object-oriented analysis, design and programming are *related but distinct*.

- **OOA (OO analysis)** is concerned with developing an object model of the application domain. **(problem space)**.
- **OOD (OO design)** is concerned with developing an object-oriented system model to realize the requirements. **(solution space)**.
- **OOP (OO programming)** is concerned with implementing an OOD using an OO programming language such as Java or C++.

An **OO software design** is represented as a set of *interacting objects* that *manage their own state and operations.*

- Design is developed following an *object-oriented design process*.
- Various *models* can be used to describe an object-oriented design.
- *UML* may be used to represent these models.

Object oriented design (OOD)

Characteristics of OOD

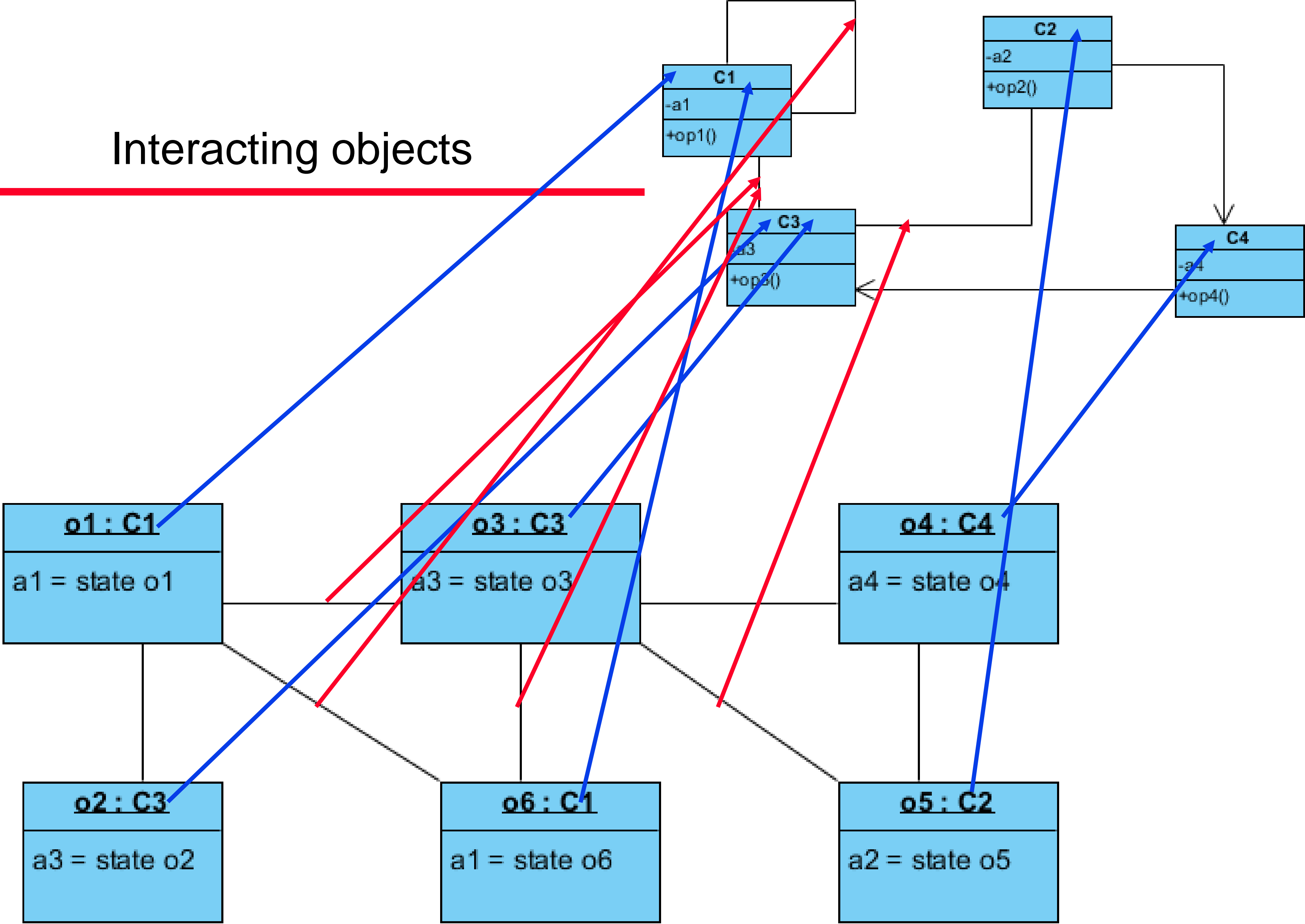
- Objects are *abstractions* of real-world or of system entities and *manage themselves*.
- Objects are *independent and encapsulate* state and representation information.
- System functionality is expressed in terms of object *services*.
- Objects communicate by *message passing*.
- Objects may be distributed and may execute *sequentially or in parallel.*

Advantages of OOD

Encapsulation (*hiding information* inside objects) means that changes made to an object do not affect other objects in an unpredictable way.

- Easier *maintenance*: objects may be understood as stand-alone entities.
- Objects are potentially *reusable* components.
- strong support for software *evolution*.

Interacting objects



Topics covered

Objects and object classes

An object-oriented design process

Design with reuse

The reuse landscape
Design patterns
Application frameworks
Software product lines
COTS product reuse

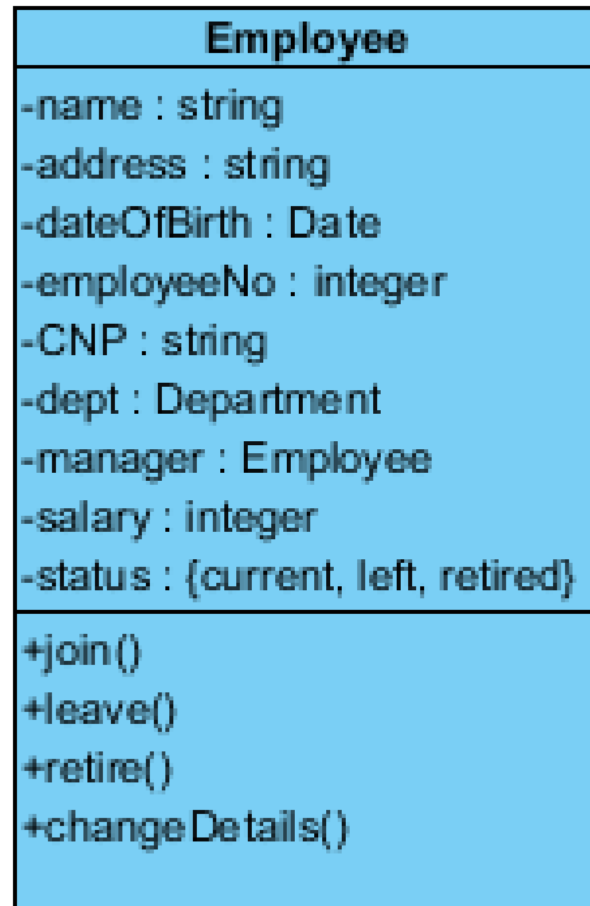
Objects and object classes

- Objects are entities in a software system which represent instances of real-world and of system entities.
- Object classes are templates for objects. They may be used to create objects.
- Object classes may inherit attributes and services from other object classes.

An object is an entity that has a **state** and a defined set of **operations** which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) which request these services when some computation is required.

Objects are created according to some object class definition. An object class definition serves as a template for objects. It includes **declarations** of all the **attributes** and **services** which should be associated with an object of that class, and also the implementations of these services.

Example: Employee object class (UML)



Object communication

Conceptually, objects communicate by message passing.

- Messages content:
  - The *name* of the service requested by the calling object;
  - Copies of the *information* required to execute the service
  - The name of a holder for the *result* of the service.
- In practice, messages are often implemented by operation calls
  - *name* = operation name;
  - *information* = parameter/arguments list;
  - *holder* = return variable

Examples:

Call a method associated with a buffer object that returns the next value in the buffer

```
v = circularBuffer.get () ;
```

Call the method associated with thermostat object that sets the temperature to be maintained

```
thermostat.setTemp (20) ;
```

Generalization and inheritance

Objects are *instances (members)* of classes.

- Classes define *attribute types and operations*.
- Classes may be arranged in a class *hierarchy* where one class (a super-class) is a *generalization* of one or more other classes (sub-classes).
- A sub-class *inherits* the attributes and operations from its super class and may add new operations and attributes of its own.

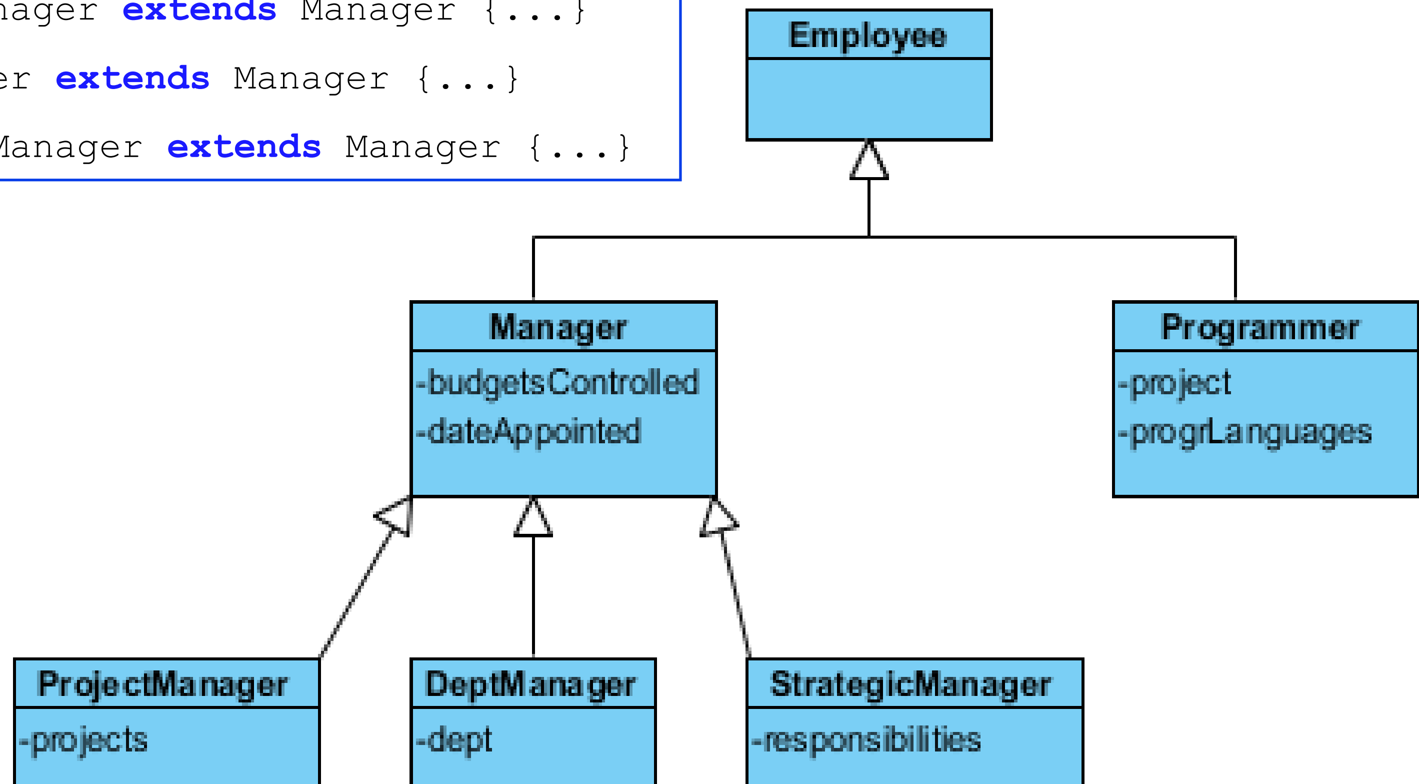
- *Generalization* relationship in the UML coresponds, in OO programming languages, with the relation between a superclass and its subclasses, having *inheritance* as property.
  - Generalization is an abstraction mechanism which may be used to classify entities.
  - Inheritance is a reuse mechanism at both
    - the design level
    - the programming level.

The inheritance graph is a source of organizational knowledge about domains and systems.

Example: A generalization hierarchy

```
class Manager extends Employee {...}
class Programmer extends Employee {...}
```

```
class ProjectManager extends Manager {...}
class DeptManager extends Manager {...}
class StrategicManager extends Manager {...}
```



Realization and implementation

Interface = specification of a set of operations, containing their signatures and semantics.

- Interfaces specify the signatures of the operations: operation name, ordered parameter list including parameter types, return type.
- Classes may implement one or more interfaces defining the implementation of the operations specified in these interfaces.

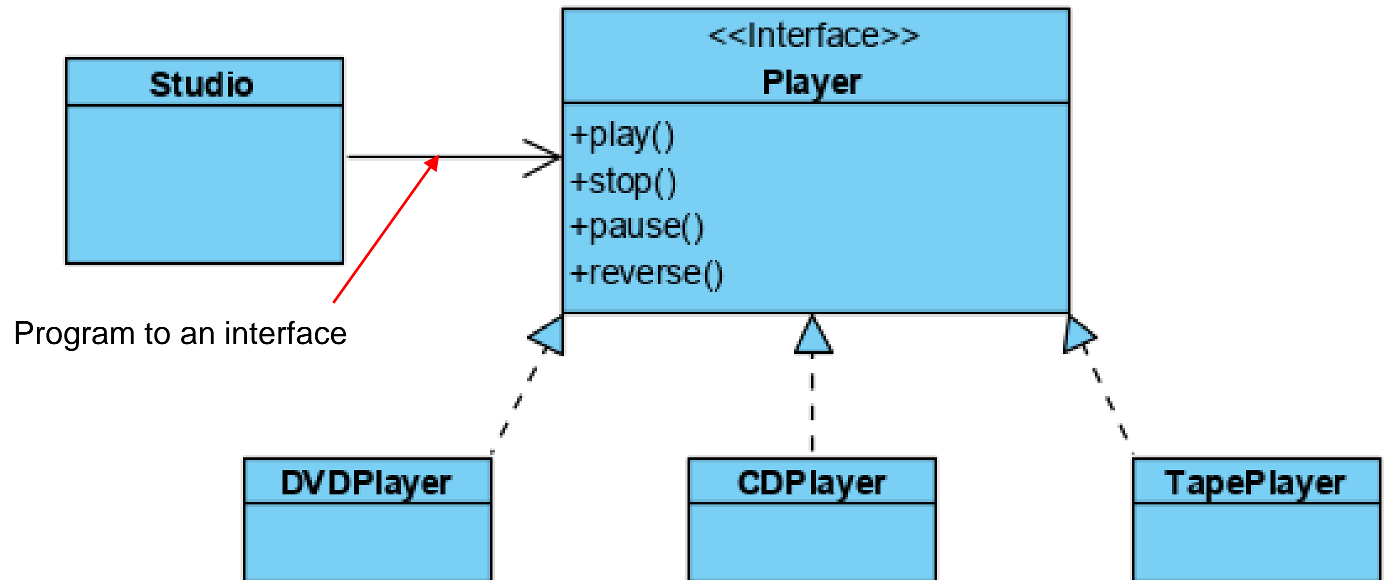
- Realization relationship in UML coresponds to the interface implementation in OO languages.

Hiding implementation behind interfaces and *program to interfaces* means that changes made to an object do not affect other objects in an unpredictable way.

Example : Realization and implementation

```
interface Player {
    public void play();
    public void stop();
    public void pause();
    public void reverse();
}
```

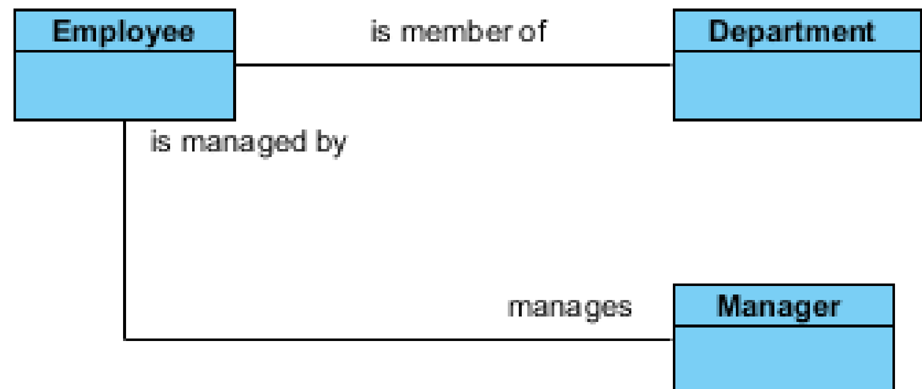
```
class DVDPlayer implements Player {
        public void play(){...}
        ...
}
class CDPlayer implements Player {...}
class TapePlayer implements Player {...}
```



UML associations

- Objects and object classes participate in relationships with other objects and object classes.
- In the UML, a general relationship is indicated by an association.
- Associations may be annotated with information that describes the association.
- Associations indicate that an object has attributes which are references to associated objects.

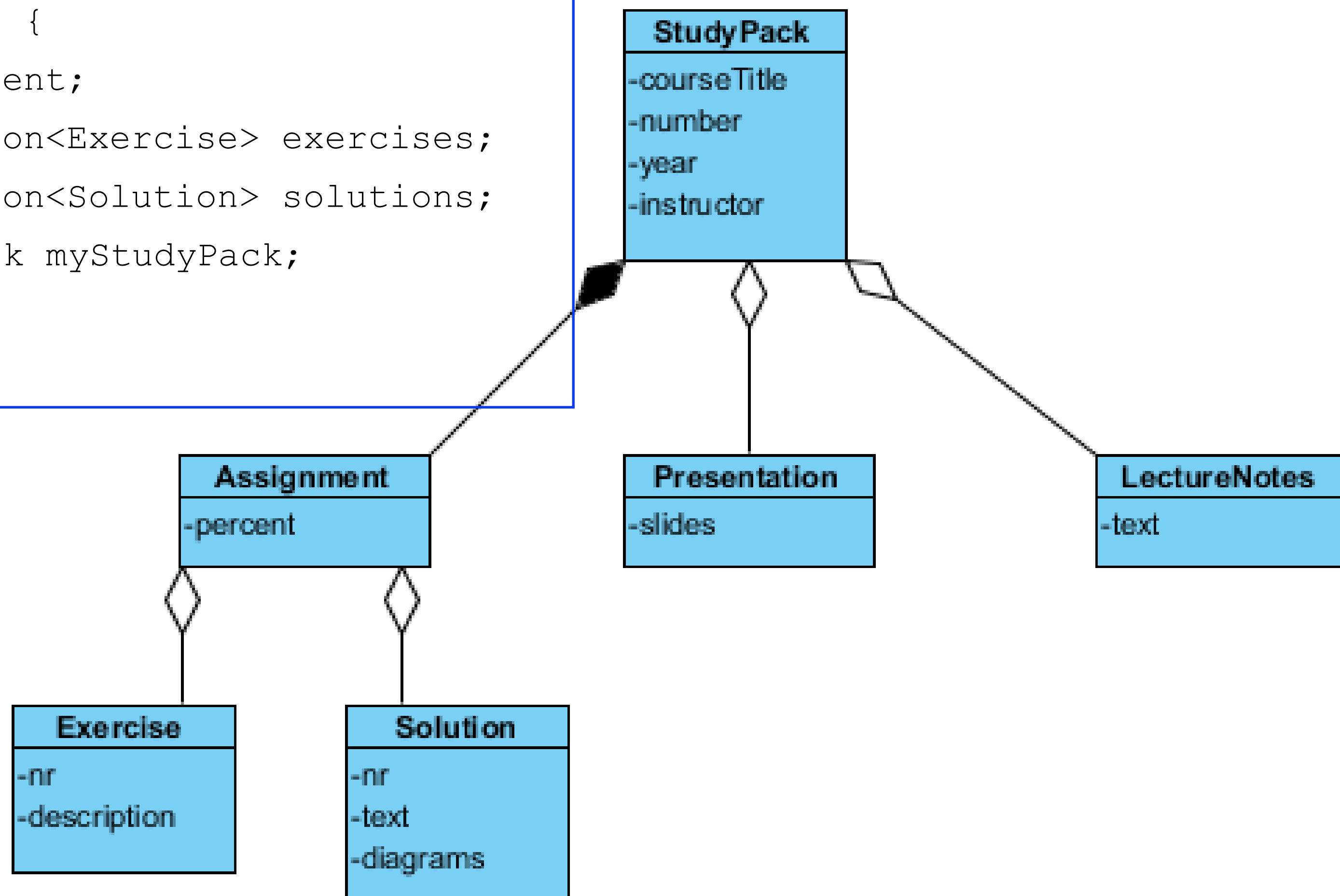
```
class Employee {
  private Department dep;
  private Manager boss;
  ...
}
```



Example: object aggregation

A particular association is **aggregation**, with its strong version, **composition**.

```
class Assignment {
private int percent;
private Collection<Exercise> exercises;
private Collection<Solution> solutions;
private StudyPack myStudyPack;
...
}
```



Concurrent objects
Server and active objects

- The nature of objects as **self-contained entities** make them suitable for **implementation of concurrecy**.
- The **message-passing** model of object communication can be implemented directly if objects are running on **separate processors** in a distributed system.

Server object

- The object is implemented as a parallel process (server) with entry points corresponding to object operations. If no calls are made to it, the object suspends itself and waits for further requests for service.

Active object

- The object is implemented as parallel process and the internal object state may be changed either by internal operations executing within the object itself or by external calls. The process representing the object continually executes these operations, so it never suspends itself.

Example of active object : Transponder

Active objects may have their attributes modified by external calls to operations but may also update them autonomously using internal operations.

Example: A Transponder object broadcasts an aircraft's position. The position may be updated using a satellite positioning system. The object periodically updates the position by triangulation from satellites.

```
class Transponder extends Thread {
        Position currentPosition ;
        Coords c1, c2 ;
        Satellite sat1, sat2 ;
        Navigator theNavigator ;

        public Position givePosition () {return currentPosition ;}

        public void run () {
                while (true) {
                        c1 = sat1.position () ;
                        c2 = sat2.position () ;
                        currentPosition = theNavigator.compute (c1, c2) ;
                }
        }
} //Transponder
```

Threads in Java are a simple construct for implementing concurrent objects.

Threads must include a method called `run()` and this is started up by the Java run-time environment.

Active objects typically include an infinite loop so that they are always carrying out the computation.

Topics covered

Objects and object classes

An object-oriented design process

Design with reuse

The reuse landscape
Design patterns
Application frameworks
Software product lines
COTS product reuse

An object-oriented design process

- Knowing the UML is not enough –a **process** is needed to realize OO design.
- The UML provides/defines many different types of diagrams, but really does not provide any guidance on where in a process they should be used.

Just learning the UML to learn object-oriented analysis and design would be like learning the English dictionary to learn the English language.

- Structured design processes involve developing a number of different system models.
  - A lot of effort is required for development and maintenance of these models and, for small systems, this may not be cost-effective.
  - However, for large systems developed by different groups, design models are an essential communication mechanism.

System context and models of use

Develop an understanding of the relationships between the software being designed and its external environment.

- System context (structure)
  - A *static* model that describes other systems in the environment.
- Model of system use (interactions)
  - A *dynamic* model that describes how the system interacts with its environment.

Architectural design

- Define the context and modes of use of the system;
- **Design the system architecture;**
- Identify the principal system objects;
- Develop design models;
- Specify object interfaces.

Once interactions between the system and its environment have been understood, this information is used for designing the system architecture.

Activities implied:

- Identify major components that make up the system and their interactions.
- Organize the components using architectural styles (ex. layered, client-server,…).

Object identification

- Identifying objects (or object classes) is the most difficult part of object oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.
- Domain knowledge is used to identify:
  - Objects
  - Attributes
  - Services

Approaches to objects identification

- Define the context and modes of use of the system;
- Design the system architecture;
- **Identify the principal system objects;**
- Develop design models;
- Specify object interfaces.

- Use a *grammatical* approach based on a natural language description of the system (used in Hood OOD method).
- Base the identification on *tangible things* in the application domain.
- Use a *behavioural* approach and identify objects based on what participates in what behaviour.
- Use a *scenario*-based analysis. The objects, attributes and methods in each scenario are identified.
- Use *domain knowledge* to identify more objects and operations.

Design models

Design models show the objects and object classes and relationships between these entities.

- **Static** models describe the static structure of the system in terms of object classes and relationships (class models).
- **Dynamic** models describe the dynamic interactions between objects, and their response to events.

Examples of design models:

- *System - subsystem* models that show logical groupings of objects into coherent subsystems.
- *Sequence* models that show the sequence of object interactions.
- *State machine* models that show how individual objects change their state in response to events.
- Other models include *use-case* models, *aggregation* models, *generalization* models, etc.

Object interface specification

- Define the context and modes of use of the system;
- Design the system architecture;
- Identify the principal system objects;
- Develop design models;
- **Specify object interfaces.**

Object interface specification = definition for the services provided through the interface (their signatures and semantics).

- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- Objects may have several interfaces which are viewpoints on the methods provided.
- A group of objects may be accessed through the same interface.

UML uses class diagrams to represent interfaces.

- Classes are stereotyped with `<<interface >>` and do not have attributes compartment.

An alternative approach is to use a programming language to describe interfaces.

- The advantage is given by syntax-checking facilities in the compiler.

Semantics may be defined using OCL (Object Constraint Language).

Formative evaluation

1. Explain why specifying object interfaces allows objects and other components to be designed in parallel.

2. Which are the main results obtained during an object-oriented design process ?

https://forms.gle/P1zpKNw2YPnbN1VT8

Topics covered

Objects and object classes

An object-oriented design process

Design with reuse

The reuse landscape
Design patterns
Application frameworks
Software product lines
COTS product reuse

Software reuse

- In most engineering disciplines, systems are designed by *composing existing components* that have been used in other systems.

- Software engineering has been more focused on original development but it is now recognised that to achieve *better software*, *more quickly* and at *lower cost*, we need to adopt a design process that is based on *systematic software reuse.*

Obs. There has been a major switch to reuse-based development.

Software reuse

Reuse is possible at a range of ***granularity levels***, from simple functions to complete application systems.

- **Application system** reuse

A whole application system reused either by *incorporating* it without change into other systems (COTS reuse) or by *developing application families* (SPL).

- **Component** reuse

Components - collections of objects and object classes that operate together to provide related functions and services, often found in frameworks - are reused.

- **Object and function** reuse

Software components that implement a single well-defined object or function, available in libraries, may be reused.

Reuse is possible at a range of ***abstraction levels***, from concrete entities (functions, classes, components, applications) to concepts.

- **Concept (model)** reuse

Patterns: knowledge of successful abstractions in the design of the software

  - design patterns,
  - architectural patterns.

Reuse benefits

Benefit	Explanation
Lower development costs	Development costs are proportional to the size of the software being developed. Reusing software means that fewer lines of code have to be written.
Increased dependability	Reused software, which has been *tried and tested in working systems*, should be more dependable than new software. Its design and implementation faults should have been found and fixed.
Reduced process risk	The cost of existing software is already known, whereas the costs of development are always a matter of judgment. This is an important factor for project management because it *reduces the margin of error in project cost estimation*. This is particularly true when relatively large software components such as subsystems are reused.
Effective use of specialists	Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.
Standards compliance	Some standards, such as user interface standards, can be implemented as a *set of standard reusable components*. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.
Accelerated development	Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.

Reuse problems

Problem	Explanation
Increased maintenance costs	If the source code of a reused software system or component is not available then maintenance costs may be higher because the reused elements of the system may become increasingly incompatible with system changes.
Lack of tool support	Some software tools do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account. This is particularly true for tools that support embedded systems engineering, less so for object-oriented development tools.
Not-invented-here syndrome	Some software engineers prefer to rewrite components because they believe they can improve on them. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.
Creating, maintaining, and using a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
Finding, understanding, and adapting reusable components	Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include a component search as part of their normal development process.

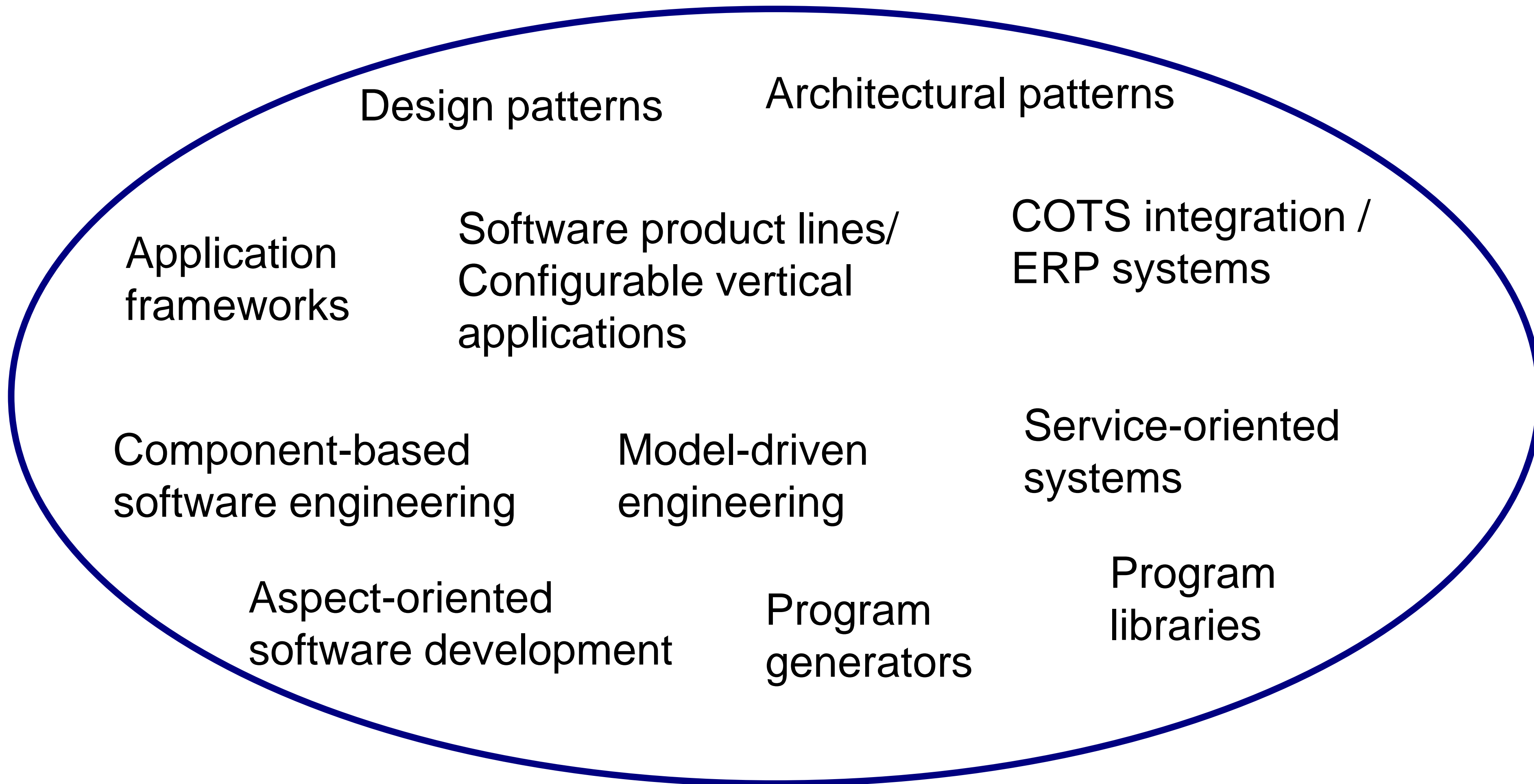
The reuse costs

Reusing software imply costs for:

- *looking for* reusable software and *testing* it in the current environment
- if applicable, *buying* reusable software
- *adapting* and *configuring* for the current requirements
- *integrating*

The reuse landscape

A range of possible reuse techniques.



Reuse approaches (1)

Approach	Description
Architectural patterns	*Standard software architectures* that support *common types of application systems* are used as the basis for application design.
Design patterns	*Generic abstractions* that occur across applications are represented as design patterns showing *abstract and concrete objects and interactions*.
Component-based development	Systems are developed by *integrating components* (collections of objects) that generally conform to component-model standards.
Application frameworks	Collections of *abstract and concrete classes are adapted and extended* to create application systems.
Legacy system wrapping	Legacy systems are 'wrapped' by *defining a set of interfaces* and providing access to these legacy systems through these interfaces.

Reuse approaches (2)

Approach	Description
Service-oriented systems	Systems are developed by *linking shared services*, which may be externally provided.
Software product lines	An *application type* is generalized around a *common architecture* so that it can be *adapted for different customers*.
COTS product reuse	Systems are developed by *configuring and integrating existing application systems*.
ERP systems	Large-scale systems that encapsulate *generic business functionality and rules* are *configured* for an organization.
Configurable vertical applications	*Generic systems* are designed so that they can be configured to the *needs of specific system customers*.

Reuse approaches (3)

Approach	Description
Program libraries	*Class and function libraries* that *implement* commonly used abstractions are available for reuse.
Model-driven engineering	Software is represented as *domain models* and *implementation independent models* and then *code is generated* from these models.
Program generators	A generator system embeds *knowledge of a type of application* and is used to *generate systems in that domain* from a *user-supplied system model*.
Aspect-oriented software development	*Shared components are woven* into an application at different places when the program is *compiled*.

Reuse planning factors

- The development schedule for the software.

Granularity of reuse depends on the time available for development.

- The expected software lifetime.

Focus on maintenability for a long-lifetime systems ⇒ avoid external suppliers.

- The background, skills and experience of the development team.

Reuse technologies are complex ⇒ need time to understand and use them effectively. ⇒ Focus on area where team members have skills.

- The criticality of the software and its non-functional requirements.

Access to source code is needed in critical software. Also performance requirements impose efficient code (which can not be produced by program generators).

- The application domain.

Generic products exist for some application domains (ex. manufacturing, medical, accounting).

- The execution platform for the software.

Component models or generic products must be used on the platform(s) they are developped for (ex. ActiveX components on Microsoft platform).

Topics covered

Objects and object classes

An object-oriented design process

Design with reuse

The reuse landscape
Design patterns
Application frameworks
Software product lines
COTS product reuse

Seminal book :

Design Patterns: Elements of Reusable Object-Oriented Software

by Erich Gamma Richard Helm Ralph Johnson and John Vlissides

often referred to as the ***Gang of Four***, or ***GoF*** .

Design patterns

- *Design patterns* and *architectural patterns (styles)* are examples of *concept reuse*.
- A design pattern is a way of reusing *abstract knowledge* about a *problem* and its *solution.*
- A pattern is a description of the *problem* and the *essence of its solution*.
- It should be sufficiently *abstract* to be *reused* in different settings.
- Patterns often rely on object characteristics such as *inheritance* and *polymorphism.*

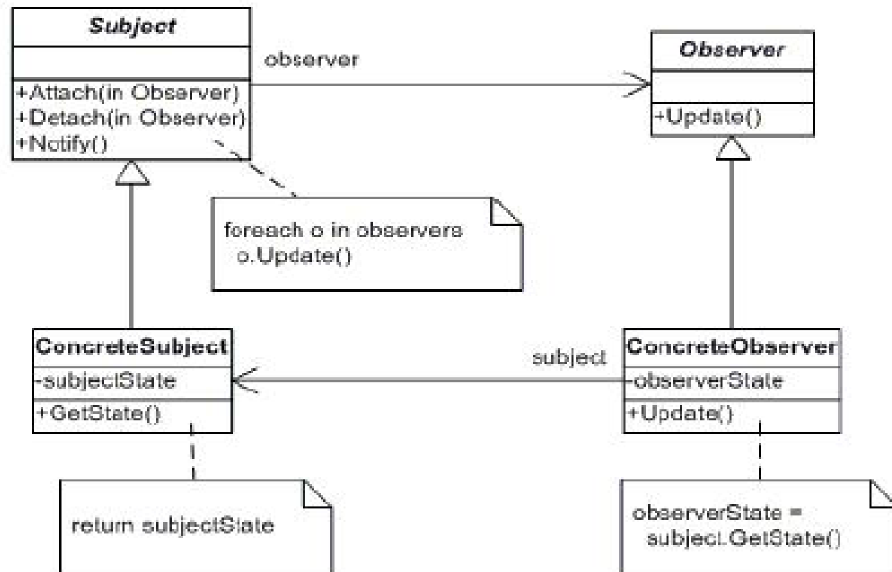
Elements of the pattern:

- Name : A meaningful pattern identifier.
- Problem description.
- Solution description.
  - Not a concrete design, but a template for a design solution that can be instantiated in different ways.
- Consequences : The results and trade-offs of applying the pattern.

Observer pattern

Excerpt from the design patterns catalogue.

Name:
OBSERVER
Intent:
Defines a one-to-many dependency between objects so that when one object changes its state, all of its dependants are notified and updated automatically.
Structure:

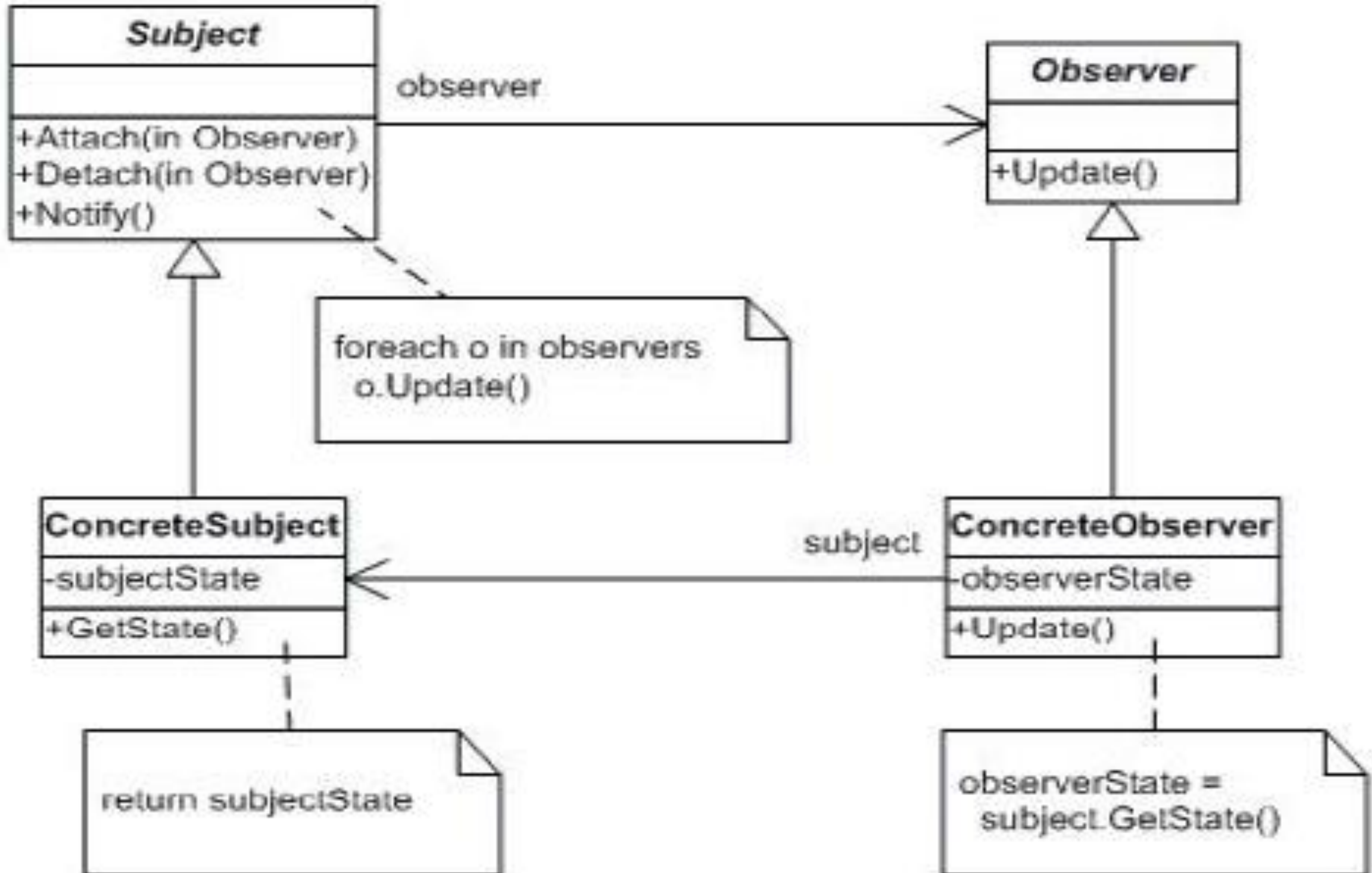


Participants:
`Subject` (the publisher from the metaphor) – publishes its state.
`Observer` (the subscriber from the metaphor) – dependant, automatically notified and updated when the state of the Subject changes.

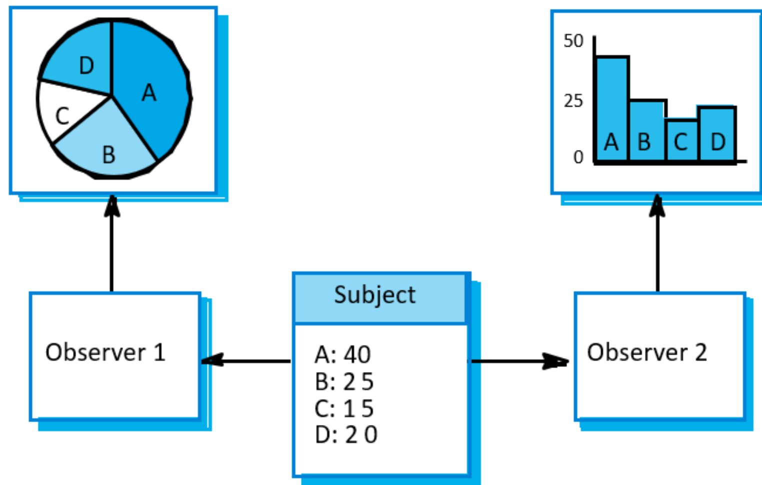
When data in `Subject` changes, each `Observer` is notified.
`Observer` has registered (subscribed to) with the `Subject` to receive updates when the data in the `Subject` changes.

Aplicability when:
- an abstraction has two aspects, one dependent on the other; encapsulating these aspects in separate objects lets you vary and reuse them independently.
- a change to one object requires changing others, and you do not know how many objects need to be changed.
- An object should be able to notify other objects without making assumptions about who these objects are; in other words, you do not want these objects tightly coupled.

Observer pattern

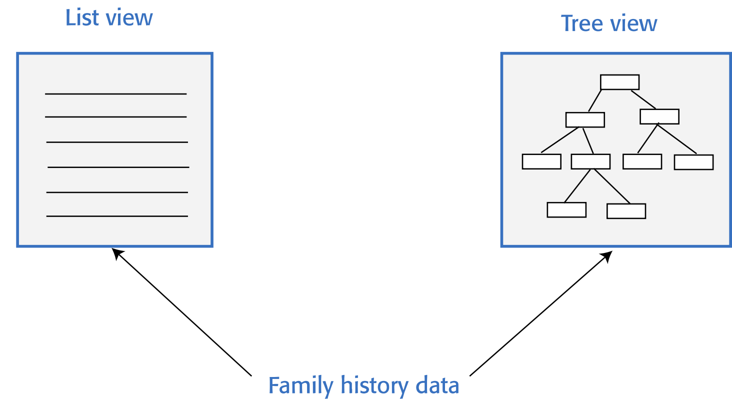


Examples : Multiple displays



Pie graph and bar graph

Figure 8.7 List view and tree view
Ian Sommerville – Engineering Software Products



Topics covered

Objects and object classes

An object-oriented design process

Design with reuse

The reuse landscape
Design patterns
Application frameworks
Software product lines
COTS product reuse

Application frameworks

- Application framework is a software structure made up of:
  - a collection of *abstract and concrete classes*
  - the *interfaces between them*.
- Any framework has a default behaviour that can not be changed
- An application is implemented by:
  - *adding components* to fill in parts of the design
  - *extending the abstract classes* in the framework.
  - *adding handlers for events* that are recognised by the framework.
- Frameworks are moderately large entities that can be reused.

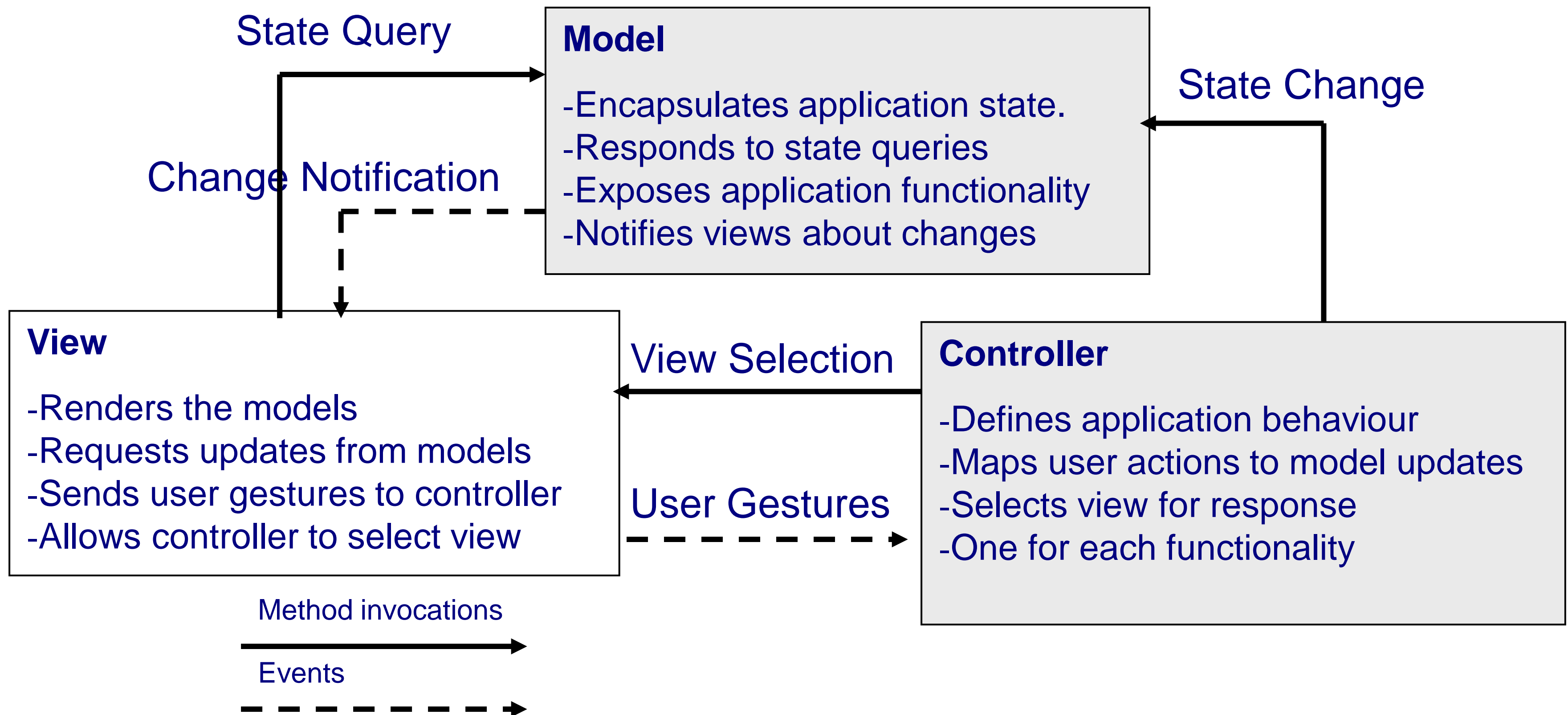
Web application frameworks (WAF)

- Support construction of dynamic web sites as a front-end for web applications.
- WAF – available for all of the commonly used web programming languages (Java, Python, Ruby, etc.)
- Interaction model is based on the Model-View-Controller composed pattern.

MVC - System infrastructure framework for GUI–based applications.

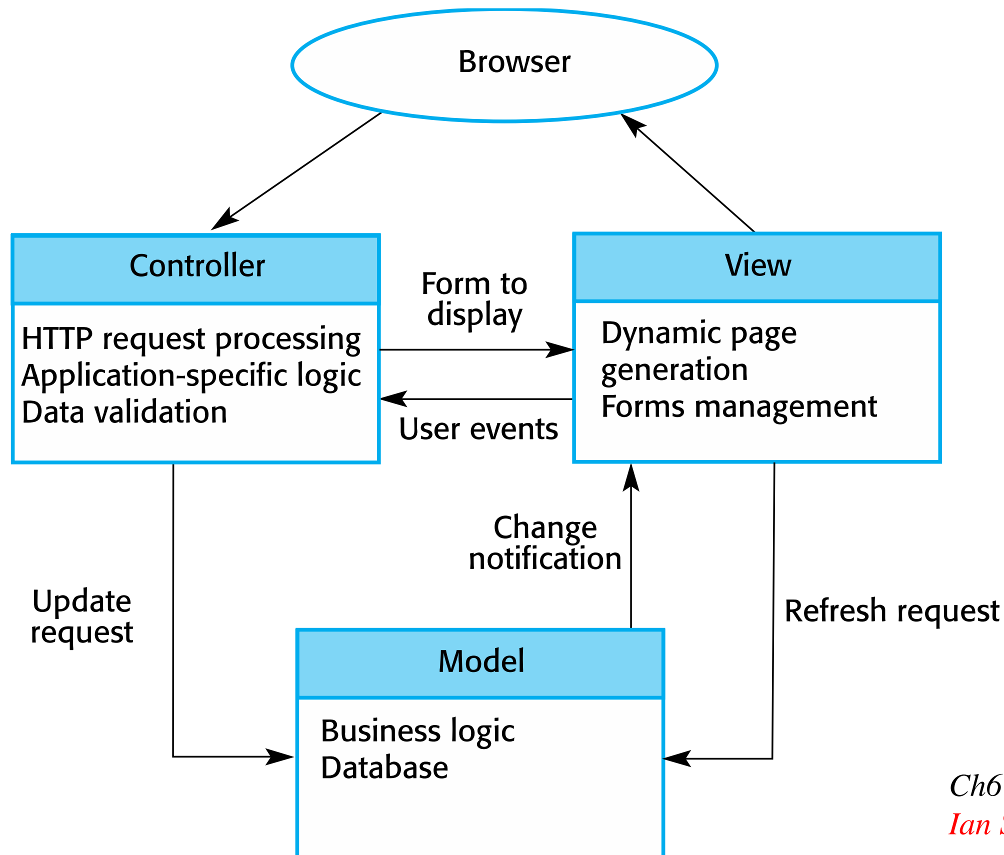
- Allows for :
  - multiple presentations of an object
  - separate interactions with these presentations.
- MVC framework involves the instantiation of a number of design patterns: Observer, Strategy, Composite.

Model-view-controller



- Decouples views and model ⇒ allows multiple presentations of an object.
- Establishes a subscribe / notify protocol between them ⇒ keeps the presentations synchronized with the model.

Web application architecture using the MVC pattern



Ch6 Architectural design.ppt
Ian Sommerville – Software Engineering, ed.10

WAF features

- Security

WAF may include classes to help implement user *authentication* (login) and access.

- Dynamic web pages

Classes are provided to help in defining web page *templates* and to populate these *dynamically* from the system database or other data sources.

- Database support

The framework may provide classes to provide an *abstract interface* to different databases.

- Session management

Classes to *create and manage sessions* (a number of interactions with the system by a user) are usually part of WAF.

- User interaction

Most web frameworks now provide AJAX support (Holdener, 2008), which allows *efficient interactive* web pages to be created.

Extending frameworks

- Frameworks are *generic* and must be *extended* to create a more specific application or sub-system. They provide a *skeleton architecture and behaviour* for the system.

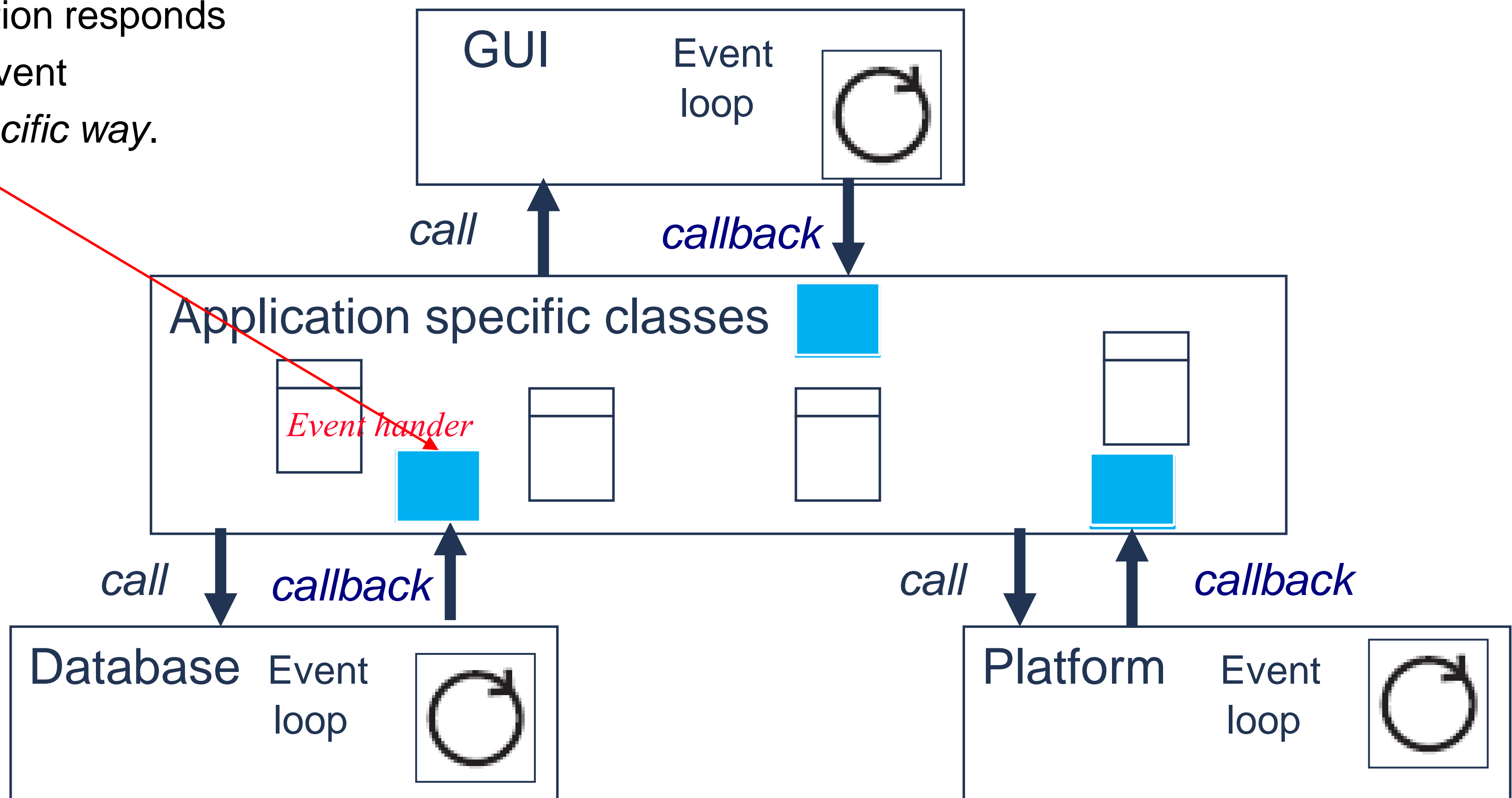
- Extending the framework involves
  - *Adding concrete classes* that *inherit* from *abstract classes* in the framework and *implement* the *abstract methods* and *interfaces*;
  - *Adding methods* that are called in *response to events* that are recognised by the framework.
  - *Adding* new *components*, in predifined locations of the framework, to fill in parts of the design.

Problem with frameworks is their complexity, which means that it takes a long time to use them effectively.

Inversion of control (IoC) in frameworks

Callback – method called in response to events recognized by the framework.

Application responds
to the event
in a *specific way*.



Formative evaluation

1. Consider an application to be developed using an application framework. What will be reused by the application and what the application needs to add ?

https://forms.gle/Gh4Z2mGfyQKu6CET8

Topics covered

Objects and object classes

An object-oriented design process

Design with reuse

The reuse landscape
Design patterns
Application frameworks
Software product lines
COTS product reuse

Software product lines

Def.1 **Software product line** = a set of applications with a *common architecture* and *shared components*, with each application *specialized* to fulfill *different requirements* in *a specific domain*.

Def.2 **Software product line** = a set of software-intensive systems that share a *common, managed set of features* satisfying the *specific needs* of a particular *market segment* or *mission* and that are developed from a *common set of core assets* in a *prescribed way*.

Software product lines (or application families) are *applications with generic functionality* that can be *adapted* and *configured* for use in a specific context.

- Adaptation may involve:
  - Component and system *configuration*;
  - *Adding* new components to the system;
  - *Selecting* from a library of existing components;
  - *Modifying* components to meet new requirements.

Component categories for a software product line

Specialized application components

Configurable components

Core components

- *Provide infrastructure support.*
- *Usually not modified when developing a new instance of the product line.*

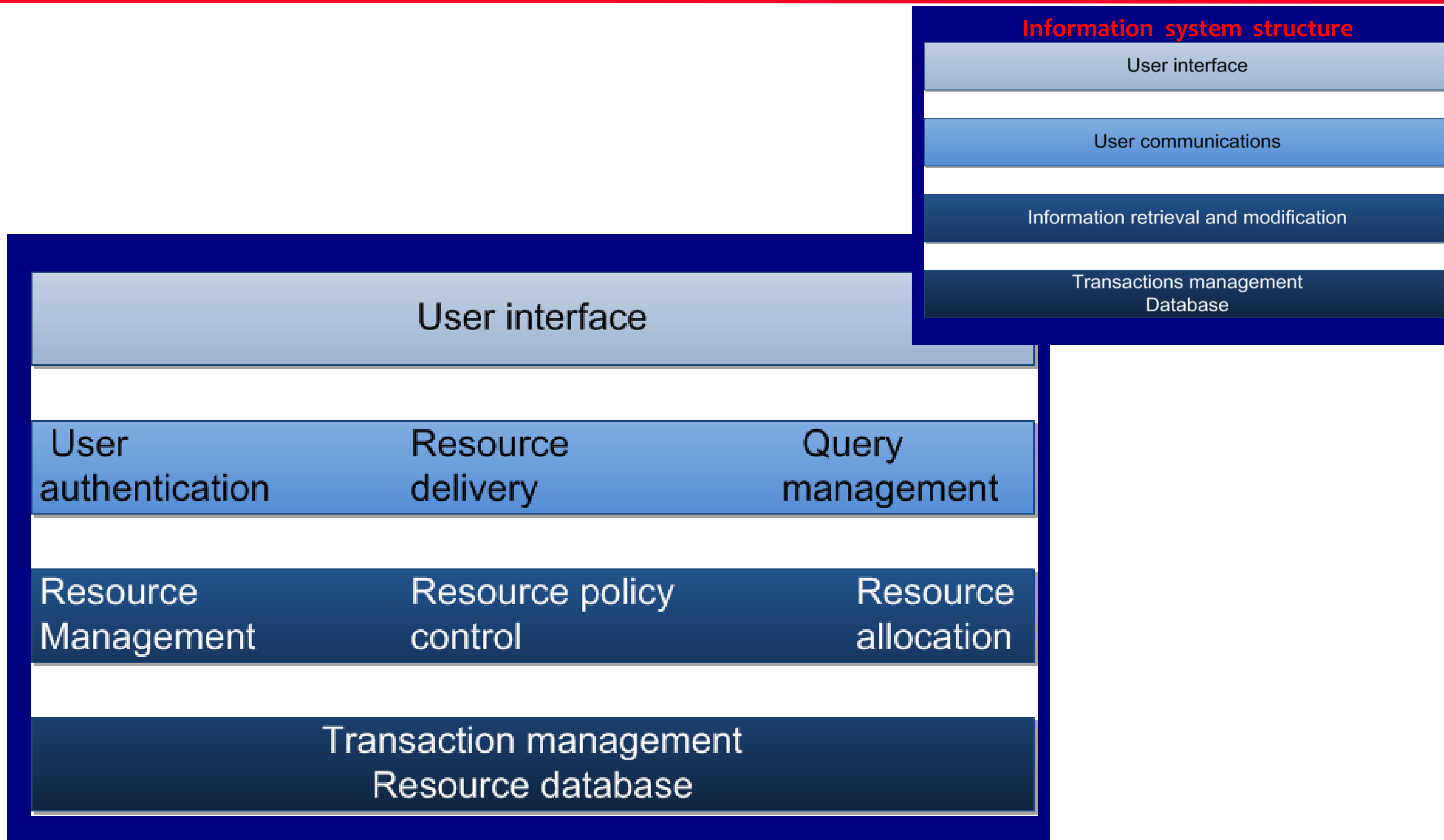
- *Modifyed and configured to specialize them to a new application.*

- *Specialized, domain-specific components some or all of which may be replaced when a new instance of a product line is created.*

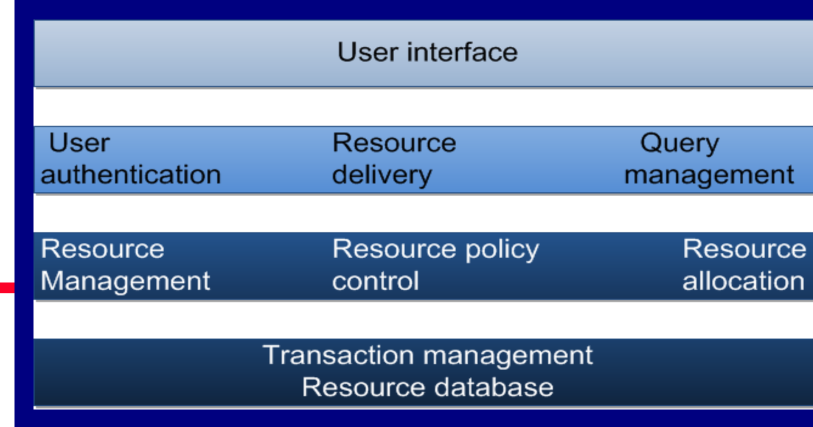
Application frameworks (AF) and product lines (SPL)

Application frameworks	Software product lines
Rely on object-oriented features such as polymorphism to implement extensions.	Need not be object-oriented (ex. embedded software for a mobile phone).
Focus on providing technical support rather than domain-specific support.	Embed domain and platform information
Different types of applications.	(often) Control applications for equipment.
Different applications, developed by different companies.	Made up of a family of applications, usually owned by the same organization.

Example:
a ***generic*** resource management ***system***



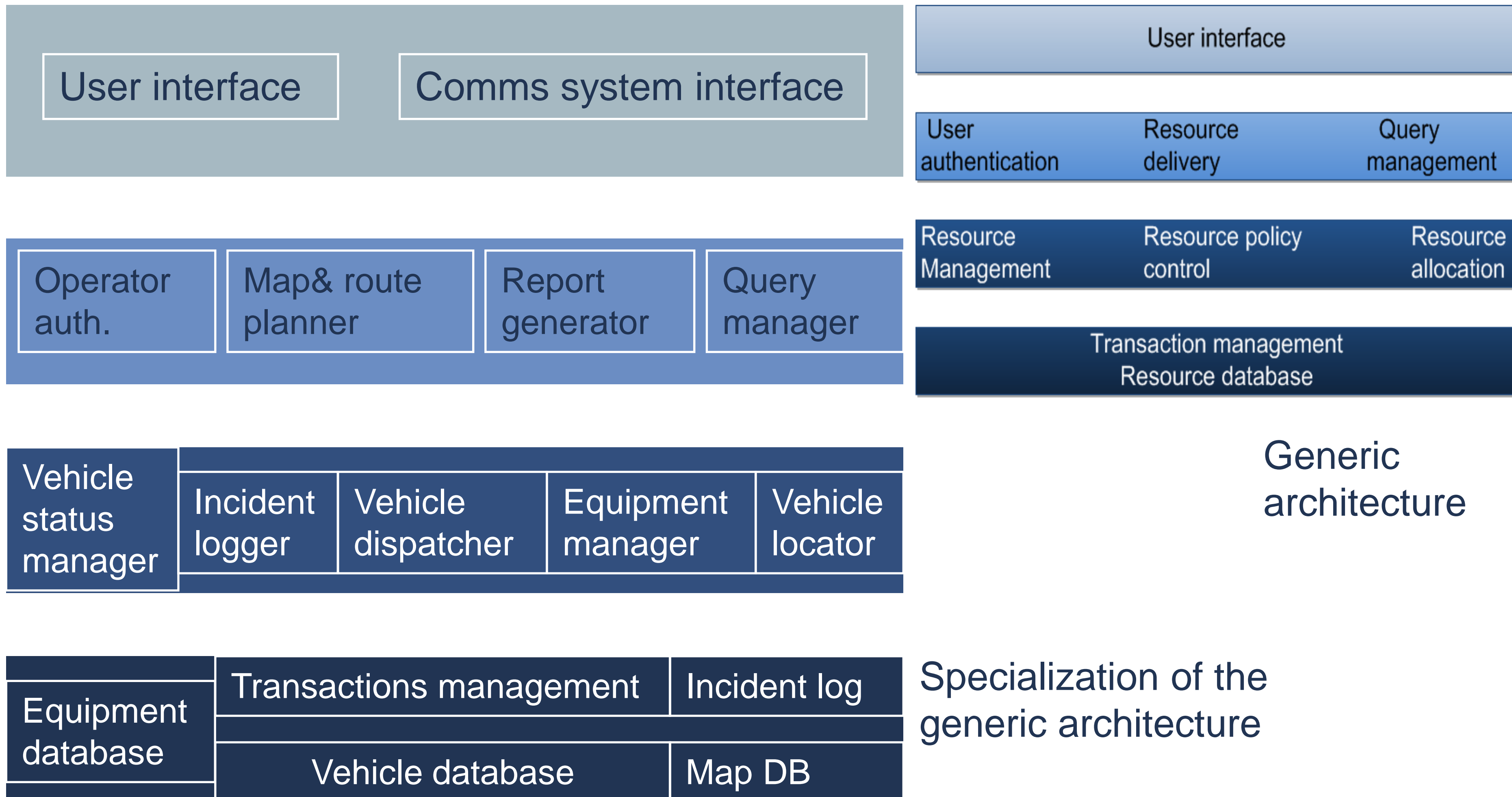
Example:
A ***product line*** for vehicle dispatching



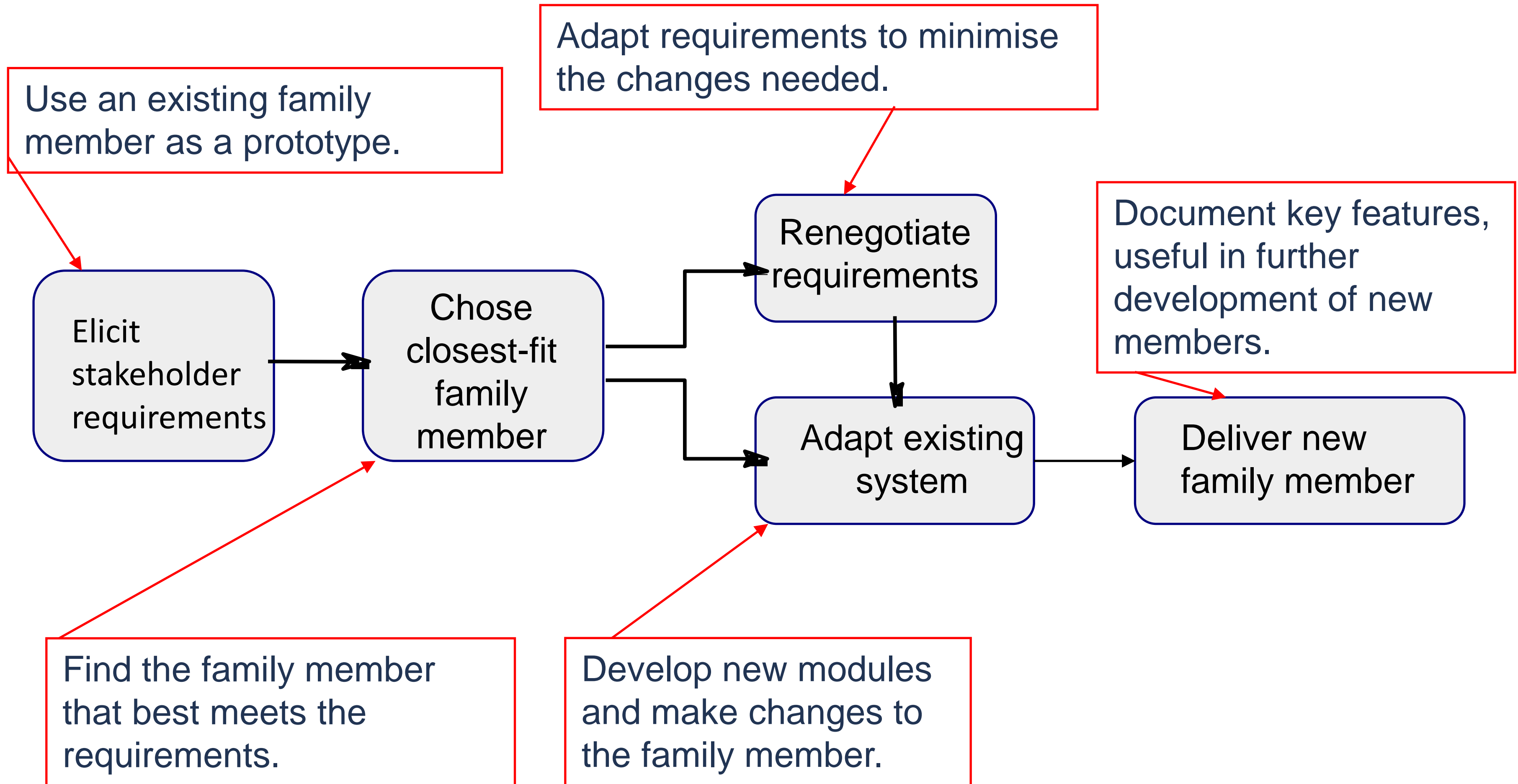
The product line for vehicle dispatching : *specialised* resource management system where the aim is to allocate resources (vehicles) to handle incidents.

- Specialization includes:
  - At the ***UI*** level, there are components for *operator display* and *communications*;
  - At the ***I/O management*** level, there are components that handle *authentication, reporting* and *route planning*;
  - At the ***resource management*** level, there are components for vehicle *location* and *dispatch, managing vehicle status* and *incident logging*;
  - The ***database*** includes *equipment, vehicle* and *map* databases.

Example:
A vehicle dispatching system family architecture



Product instance development
(ex. for police, fire or ambulance service)



Software product configuration

- *Design time* configuration
  - A common product line core is *adapted* and *changed* according to the requirements of particular customers.
  - Involves *changing and extending the source code* ⇒ flexibility.

- *Deployment time* configuration
  - A generic system is configured by *embedding knowledge* of the customer's requirements and business processes.
  - A configuration tool is used to record the configuration in a configuration database or in a set of configuration files.
  - The *code is not changed*.

Design time configuration

- Generic products usually emerge after experience with specific products.

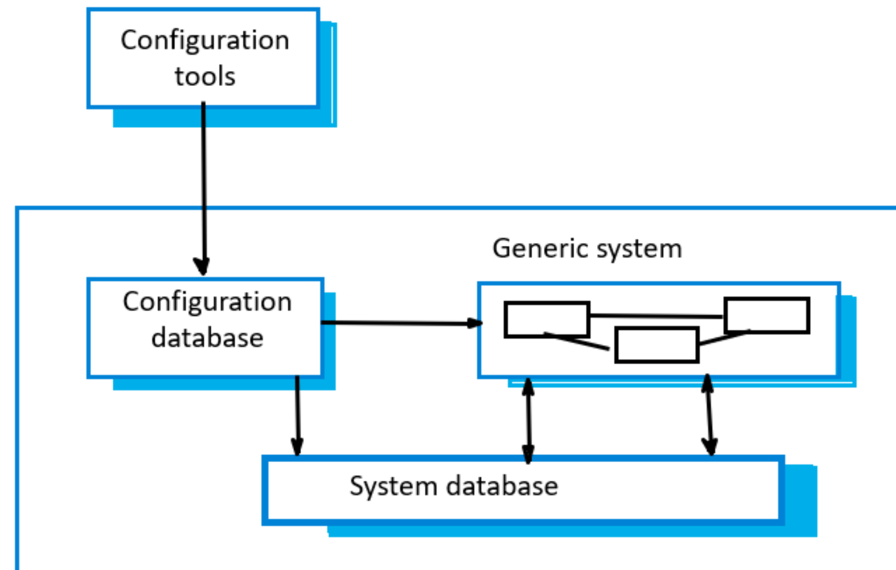
- Design-time configuration may start with an existing product instance or with a generic system.

- Software product lines that are configured at design time are *instantiations and specializations of generic application architectures.*

- Architectures must be structured in such a way:
  - to separate different sub-systems
  - to allow them to be modified.

Deployment time configuration

- *Component selection* - select the modules in a system that provide the required functionality.
- *Workflow and rule definition* - define workflows (how information is processed, stage-by-stage) and validation rules that should apply to information entered by users or generated by the system.
- *Parameter definition* - specify the values of specific system parameters that reflect the instance of the application to be created.



Topics covered

Objects and object classes

An object-oriented design process

Design with reuse

The reuse landscape
Design patterns
Application frameworks
Software product lines
COTS product reuse

Application system reuse

Involves the reuse of entire application systems either by

- *configuring* a system for an environment

or by

- *integrating* two or more systems to create a new application.

COTS - Commercial Off-The-Shelf systems.

- COTS systems are usually *complete application systems* that offer an *API* (Application Programming Interface).
- COTS systems have *generic features* and can be *adapted* for different customers by *deployment time configuration* (without changing the source code).
- Usually contain *built-in configuration mechanisms*.

Examples of COTS products

- Database managers
- Image processors
- Graphics and charting components
- Internet communication components
- Security and encryption components
- Spreadsheet tools
- Text processing tools
- GUI controls

Benefits of COTS reuse

- More *rapid* deployment of a *reliable* system may be possible.
- *Visible* provided functionality makes it easier to judge whether or not they are likely to be suitable.
- Some development risks are avoided by using existing software. However, this approach has its own risks.
- Businesses can focus on their core activity without having to devote a lot of resources to IT systems development.
- As operating platforms evolve, technology updates may be simplified as these are the responsibility of the COTS product vendor rather than the customer.

Problems of COTS reuse

- Requirements usually have to be adapted to reflect the functionality and mode of operation of the COTS product.
- The COTS product may be based on assumptions that are practically impossible to change.
- Choosing the right COTS system for an enterprise can be a difficult process, especially as many COTS products are not well documented.
- There may be a lack of local expertise to support systems development.
- The COTS product vendor controls system support and evolution.

COTS-solution and COTS-integrated systems

COTS-solution systems	COTS-integrated systems
Single product that provides the functionality required by a customer	Several heterogeneous system products are integrated to provide customized functionality
Based around a generic solution and standardized processes	Flexible solutions may be developed for customer processes
Development focus is on system configuration	Development focus is on system integration
System vendor is responsible for maintenance	System owner is responsible for maintenance
System vendor provides the platform for the system	System owner provides the platform for the system

COTS-solution systems

- COTS-solution systems are generic application systems that may be designed to support a particular business type, business activity or, sometimes, a complete business enterprise.
- Domain-specific COTS-solution systems, such as systems to support a business function (e.g. document management) provide functionality that is likely to be required by a range of potential users.
- Example:

Enterprise Resource Planning (ERP) system = generic (large scale, integrated) system that supports common business processes (such as ordering and invoicing, manufacturing, etc).

  - very widely used in large companies
  - Examples of ERP systems : SAP ERP (SAP SA) or BEA (Oracle).

ERP architecture

The *generic core*: infrastructure framework plus modules to be composed.

- A number of modules to support different business functions.
- A defined set of business processes, associated with each module, which relate to activities in that module.
- A common database that maintains information about all related business functions.
- A set of business rules that apply to all data in the database.

The generic core is *adapted by configuration,* specifying:

- included modules
- incorporated knowledge of business processes and rules.

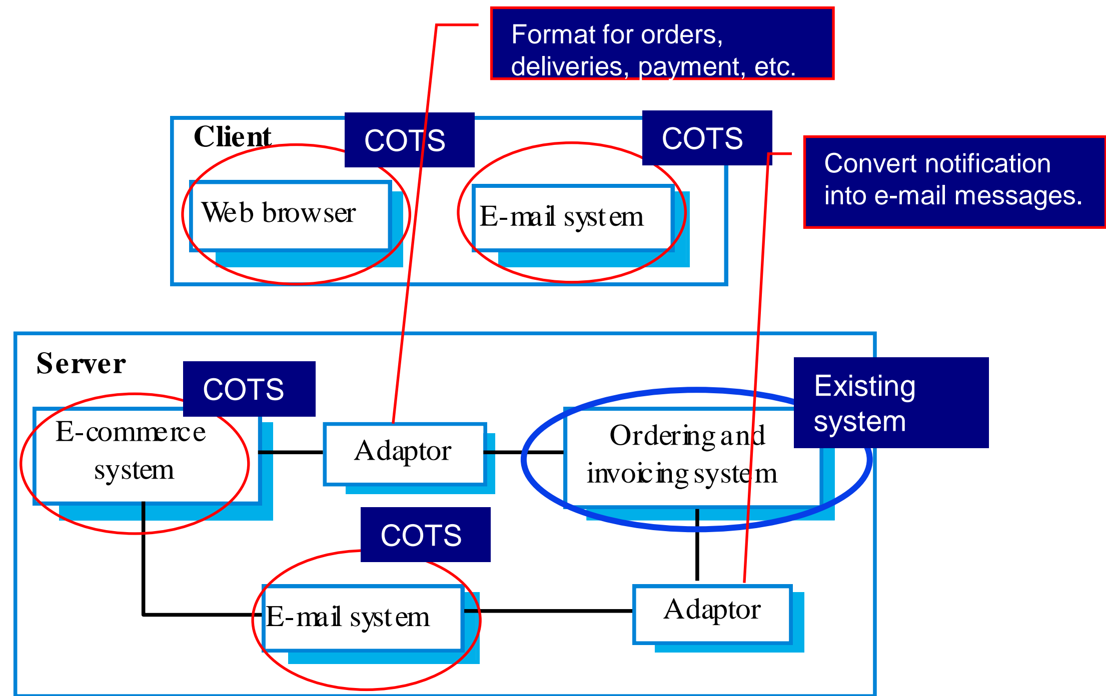
ERP configuration

- *Selecting* the required *functionality* for the system (modules).
- Establishing a *data model* that defines how the organization's data will be structured in the system database.
- Defining *business rules* that apply to that data.
- Defining the expected *interactions with external systems*.
- Designing the *input forms* and the *output reports* generated by the system.
- Designing *new business processes* that conform to the underlying process model supported by the system.
- Setting *parameters* that define how the system is *deployed* on its underlying platform.

COTS-integrated systems

- COTS-integrated systems are applications that include two or more COTS products and/or legacy application systems.
- Applicable when there is no single COTS system that meets all application needs or when a new COTS product will be integrated with systems already in use and/or with subsystems to be developed.

Example: E-procurement system

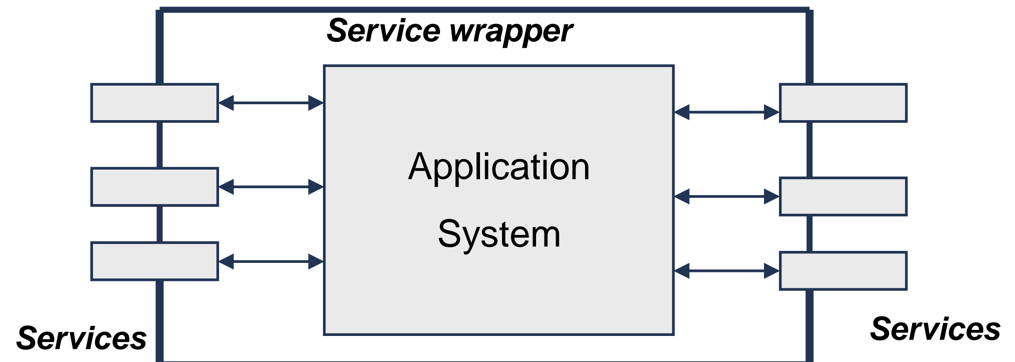


Service-oriented COTS interfaces

- COTS integration can be simplified if a service-oriented approach is used.
- A service-oriented approach means allowing access to the application system’s functionality through a *standard service interface*, with *a service for each discrete unit of functionality*.
- Some applications may offer a service interface.
- For other applications this service interface has to be implemented by the system integrator.

Service wrapper

- hides the application and provides externally visible services.
- allows access to the application system’s functionality through a *standard service interface*, with *a service for each discrete unit of functionality*.



COTS system integration problems

- Lack of control over functionality and performance
  - COTS systems may be less effective than they appear.
- Problems with COTS system inter-operability
  - Different COTS systems may make different assumptions, that means integration is difficult.
- No control over system evolution
  - COTS vendors, not system users, control evolution.
- Support from COTS vendors
  - COTS vendors may not offer support over the lifetime of the product.

Formative evaluation

1. One benefit of reusing COTS systems is avoding some development risks by using existing software. What risks are nevertheless introduced by this approach ?

https://forms.gle/fhzSVBrFDDuc3Ch47

Key points

- OOD is an approach to design so that design components have their own private state and operations.
- Objects provide services to other objects.
- Objects may be implemented sequentially or concurrently.
- UML provides different notations for defining different object models.
- A range of different models may be produced during an object-oriented design process. These include static and dynamic system models.
- The basic activities in the OO design process are:
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.
- Object-oriented design potentially simplifies system evolution.

Key points

- Advantages of reuse are lower costs, faster software development and lower risks.
- Design patterns are high-level abstractions that document successful design solutions.
- Application frameworks are collections of concrete and abstract classes that are designed for reuse through specialisation.
- Software product lines are related applications developed around a common core of shared functionality.
- COTS product reuse - reuse of large, off-the-shelf systems.
- Problems with COTS reuse include lack of control over functionality, performance, evolution and problems with inter-operation.
- ERP systems are created by configuring a generic system with information about a customer's business.