
8. Objekt-Orienteret Design

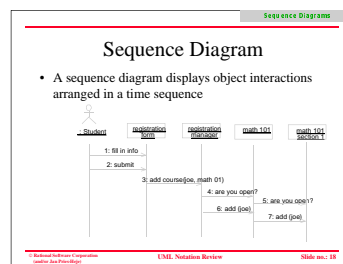
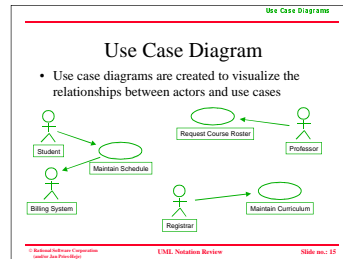
Dynamic View at OO Design
Static View at OO Design
Sequence Diagrams
Collaboration Diagrams
Package Diagrams
Design Patterns

Efter denne lektion (og øvelserne) skal du:

- Kunne analysere en given problemstilling ved hjælp af teknikker til Objekt-Orienteret Design
- For en given (mindre) problemstilling kunne tegne et komplet klassesdiagram incl. metoder
- Kunne udlede metoder til et klassesdiagram fra Use Cases, via et Sequence Diagram
- Kunne dokumentere et arkitekturdesign ved hjælp af Package Diagrams
- Kende til Design Patterns

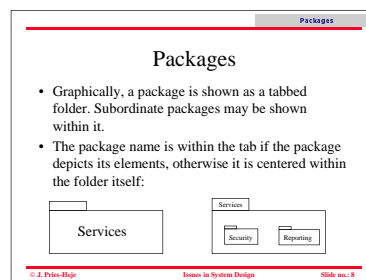
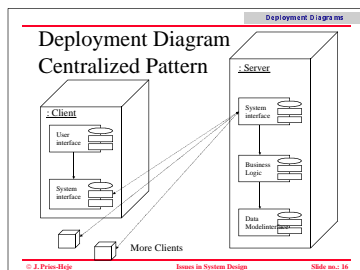
Dynamic View on Design

- User-level use-cases
- Developer-level use-cases
- Package-level use-case database
- Sequence diagrams
- Eventually State Transition Diagrams



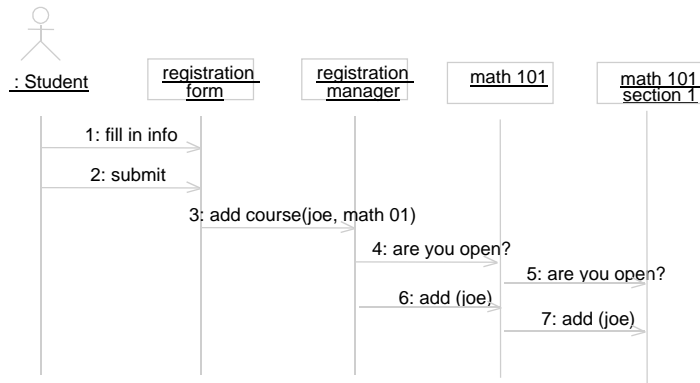
Static View on Design

- Package Diagram(s)
- Detailed Design Class Diagrams
- Deployment Diagrams



Sequence Diagram

- A sequence diagram displays object interactions arranged in a time sequence



Sequence Diagram Terminology

- System operation
 - an operation of the system that executes in response to a system event
- System event
 - an stimulus generated by an actor to a system
- Placement of operations
 - a stimulus event points toward an object
 - the object responds by executing an operation (of a similar or same name as the event)
 - an object receiving an event must contain corresponding operation

Building a Sequence Diagram

- From a Use Case textual description
 - Draw an object (line) representing a system (component, subsystem) as a black box
 - Draw an actor (line) for each actor that operates on the system
 - Draw the (external) events each actor generates, from the actor to the system
 - Draw any responses the system generates, from the system to an actor

Use Case: Buy Items

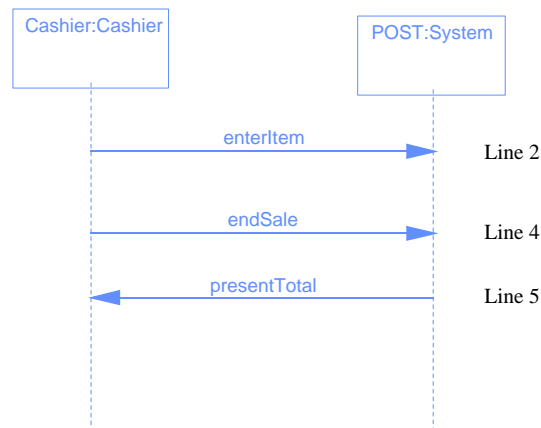
Actor Action

- 1 This use case begins when the **Customer** arrives at the **POST** checkout with items to purchase
- 2 The **Cashier records each item**.
If there is more than one of an item, the Cashier can enter the quantity as well
- 4 On completion of the item entry, the **Cashier** indicates to the **POST** that item **entry is complete**
- 6 The Cashier tell the Customer the total

System Response

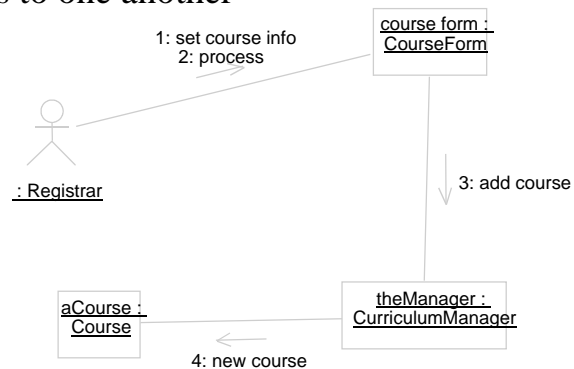
- 3 Determines the item price and adds the item information to the running sales transaction
The description and price of the current item are presented
- 5 Calculates and **presents the sale total**

A Buy Item Sequence Diagram



Collaboration Diagram

- A collaboration diagram displays object interactions organized around objects and their links to one another

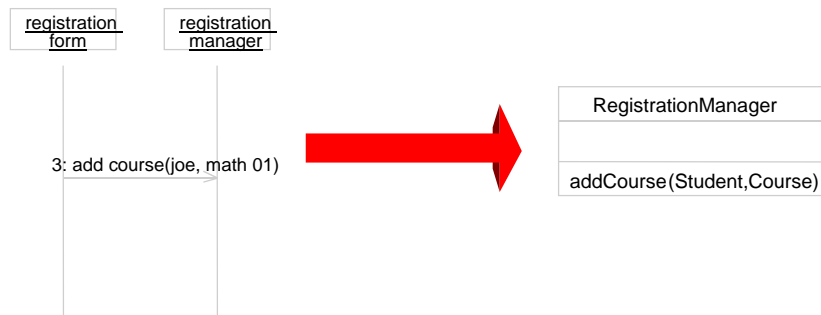


Class Diagrams

- A class diagram shows the existence of classes and their relationships in the logical view of a system
- UML modeling elements in class diagrams
 - Classes and their structure and behavior
 - Association, aggregation, dependency, and inheritance relationships
 - Multiplicity and navigation indicators
 - Role names

Operations

- The behavior of a class is represented by its operations
- Operations may be found by examining interaction diagrams



Relationships

- Relationships provide a pathway for communication between objects
- Sequence and/or collaboration diagrams are examined to determine what links between objects need to exist to accomplish the behavior -- if two objects need to “talk” there must be a link between them
- Three types of relationships are:
 - Association
 - Aggregation
 - Dependency

Relationships

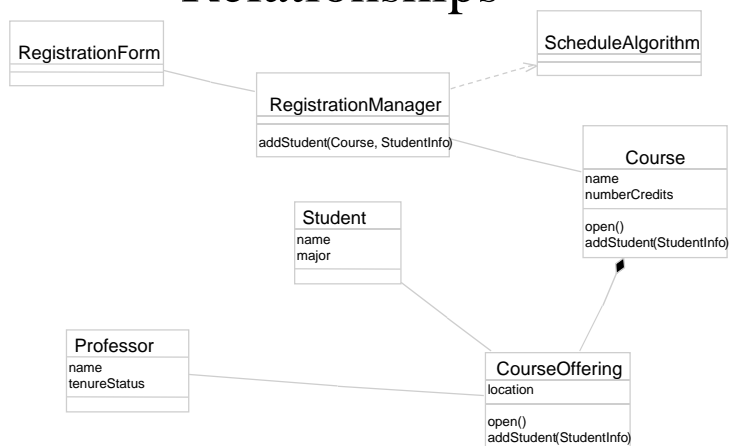
- An association is a bi-directional connection between classes
 - An association is shown as a line connecting the related classes
- An aggregation is a stronger form of relationship where the relationship is between a whole and its parts
 - An aggregation is shown as a line connecting the related classes with a diamond next to the class representing the whole
- A dependency relationship is a weaker form of relationship showing a relationship between a client and a supplier where the client does not have semantic knowledge of the supplier
 - A dependency is shown as a dashed line pointing from the client to the supplier

Finding Relationships

- Relationships are discovered by examining interaction diagrams
 - If two objects must “talk” there must be a pathway for communication



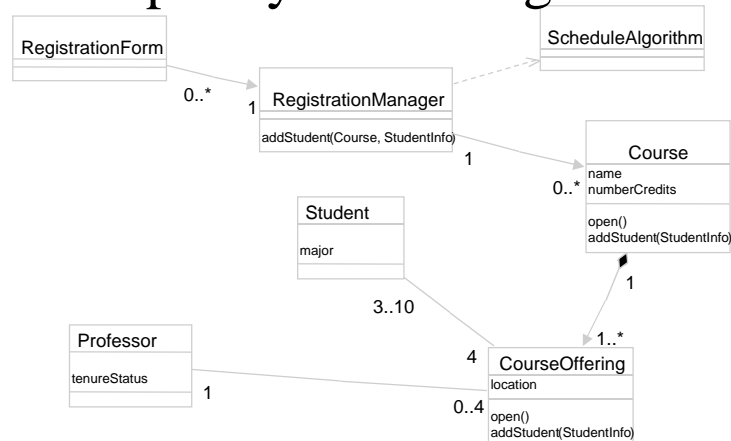
Relationships



Multiplicity and Navigation

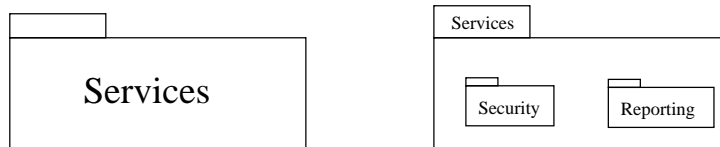
- Multiplicity defines how many objects participate in a relationships
 - Multiplicity is the number of instances of one class related to ONE instance of the other class
 - For each association and aggregation, there are two multiplicity decisions to make: one for each end of the relationship
- Although associations and aggregations are bi-directional by default, it is often desirable to restrict navigation to one direction
- If navigation is restricted, an arrowhead is added to indicate the direction of the navigation

Multiplicity and Navigation



Packages

- Graphically, a package is shown as a tabbed folder. Subordinate packages may be shown within it.
- The package name is within the tab if the package depicts its elements, otherwise it is centered within the folder itself:



Package Architecture

- A Package Architecture is a system structure composed of interconnected packages
- A package is a collection of classes whose objects collaborate to provide a service

Follow three principles:

- Reduce complexity by separating concerns
- Reflect stable context structures
- Reuse existing components

Identifying Packages

(Larman page 278)

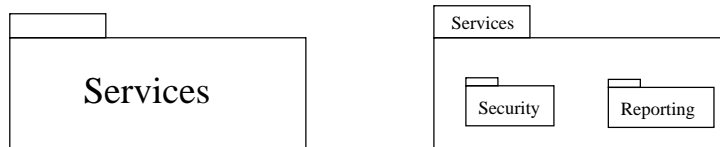
- Group elements that provide a common service (or family of related services), with relatively high coupling and collaboration, into a package
- At some level of abstraction the package will be viewed as highly cohesive – it has strongly related responsibilities
- In contrast, the coupling and collaboration between elements of different packages will be relatively low

Packages

- A package is a collection of classes whose objects collaborate to provide a service
- A class can only belong to one package
- Packages may be combined to general packages; See for example Cantor (1998) page 65
- The packages and their association make up the architecture, the high-level design, of the system

Packages

- Graphically, a package is shown as a tabbed folder. Subordinate packages may be shown within it.
- The package name is within the tab if the package depicts its elements, otherwise it is centered within the folder itself:



The Physical Architecture

- The class and package design is called the *logical architecture*
- The hierarchy of modules/object files, subsystems, and systems is called the *physical architecture*
- Source files are written implementing the design. Then compiled to modules or object files
- Modules are linked to form subsystems which are then linked to form a system
- Subsystems can be delivered as separate files linked at runtime – Dynamic linked libraries (DLLs)

The Physical Architecture

- Component diagrams illustrate the organizations and dependencies among software components
- A component may be
 - A source code component
 - A run time components or
 - An executable component

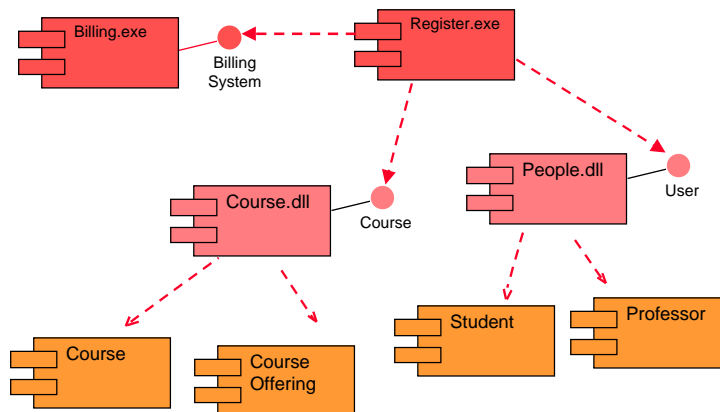
The Physical Architecture

The default mapping of design artifacts into components is that:

- Each class maps to a module
- Each package maps to a subsystem
- The top-level package diagram maps to the system

However, the granularity of the logical and the physical architecture need not be the same – That is a design decision

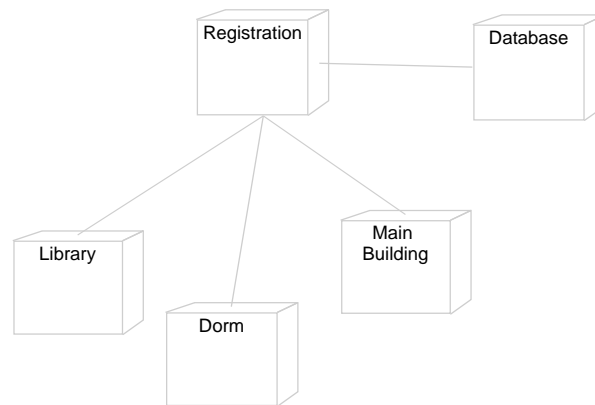
Component Diagram



Deploying the System

- The deployment diagram shows the configuration of run-time processing elements and the software processes living on them
- The deployment diagram visualizes the distribution of components across the enterprise

Deployment Diagram



Design Patterns

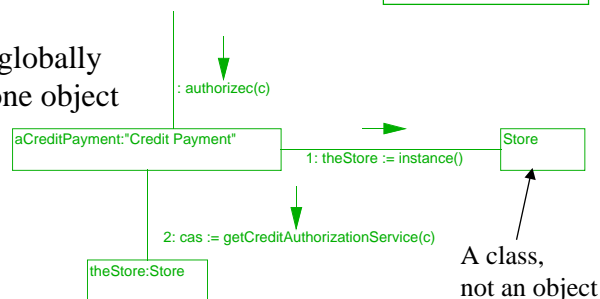
- How to choose (or construct) objects to be responsible to carry out operations?
 - What rational principles can be methodically applied to guide the design of responsibilities and object interactions?
 - “It seems right” or “it’s natural” is not good enough!!! Especially for novices
- Design Pattern
 - Principles or idioms of design codified in a structured format describing the problem it solves and a solution for the problem

Gang of Four

- The GoF book
 - Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns*, Addison-Wesley, 1995.
- A Related Book on Analysis Patterns
 - Fowler, M., 1996, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1996.

Singleton

- (Many) classes need access to a single object
 - Could pass the object as a parameter through (possibly) many messages
 - Alternatively, globally reference the one object

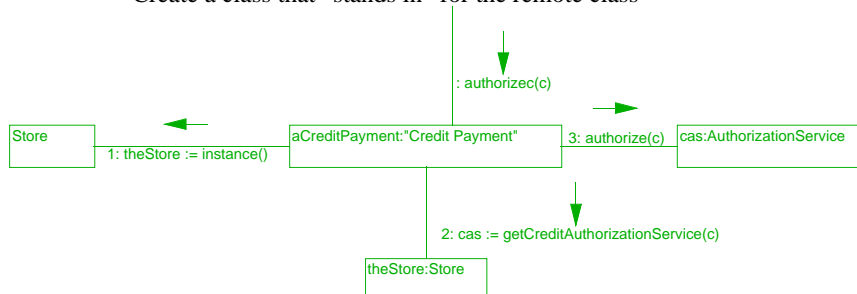


Singleton

- Problem
 - Exactly one instance of a class is allow; it is a “singleton”.
 - Many objects need to access the singleton
- Solution
 - In the singleton class, define a method that returns the one singleton class instance
 - In languages that do not provide for class methods, instead use a global function
 - Other objects can request the singleton from the class
- Motivation
 - Class (type definitions) are generally global in scope for most languages
 - with the exception of import/export
 - Asking the class for its instance does not increase coupling

Remote Proxy

- Problem
 - How to represent non-local components?
 - E.g., Credit Card Authorization Service, Database, etc.
- Solution
 - Create a class that “stands in” for the remote class



Remote Proxy

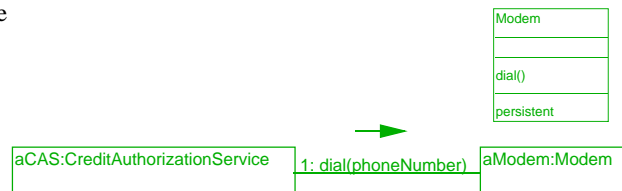
- Problem
 - The system must communicate with a component in another address space (e.g., on another machine). Who should be responsible for contacting the component?
- Solution
 - Make a local class that represents the remote component. (Local objects should think of it as actually being the component.) This local proxy for the remote object handles all interaction with the actual component.
- Motivation
 - Localize remote access behavior (which is likely to change)

Proxy

- Problem
 - Direct access to a component is not desired or possible.
- Solution
 - Make a local class that represents the component. (Local objects should think of it as actually being the component.) This local proxy for the remote object handles all interaction with the actual component.
- Motivation
 - Localize access behavior (which is likely to change)

Facade & Device Proxy

- Problem
 - How can object-oriented system be cleanly connected to non-object-oriented (legacy) systems?
 - How can hardware devices or operating system functions be represented?
- Solution
 - Create a class that encapsulates the behavior of the legacy system or device



Facade & Device Proxy

- Facade Proxy
 - Problem
 - How can a common interface to a variety of functions or interface (such a provided by a legacy system) be provide within an object-oriented system?
 - Solution
 - Define a single class that encapsulates the behavior of all the functions
- Device Proxy
 - Problem
 - How can an interface to a hardware device be provided within an object-oriented system?
 - Solution
 - Define a single class that encapsulates the behavior of the device