UNIVERSITY OF
TECHNOLOGY SYDNEY

CONCEPTUAL ARCHITECTURE

Dr Tom McBride

UTS:
ENGINEERING AND
INFORMATION
TECHNOLOGY

Contents

- Discuss the purpose of a conceptual architecture.
- What are we trying to do and why is it so hard to explain.
- Structuring the architecture.
- Evaluating the conceptual architecture.

Starting from…

- We will start from an assumption that you are familiar with object-oriented modelling and UML.
- If you are not, please pause and learn that first.
- UTS students can take the course “Foundations of Programming: Object-Oriented Design” at Lynda.com, available through the UTS Library.
  - www.lib.uts.edu.au
  - Search the library catalogue for lynda.com
  - Select the search result “Lynda.com”
  - Select electronic resource. You will be asked to log in twice.

UNIVERSITY OF TECHNOLOGY SYDNEY

WHAT IS A CONCEPTUAL ARCHITECTURE

A conceptual architecture models the problem

- A model of the problem or its proposed solution without considering implementation constraints.
  - Resources are infinite
  - Communication is instantaneous
  - Processing speed is instantaneous

- The conceptual architecture is not concerned with execution or implementation constraints, so is a useful starting point on which to base more detailed work.

- But it is a computer system and may be part of a larger socio-technical system

But that doesn't tell me how to design one

- The architecture must describe a system that is, to some extent, autonomous.
- It is not simply a collection of objects.
- Like designing a factory, hospital or office building, the task is to organize the relationships between its elements so that it achieves its objectives under varying conditions.

A system is almost a living entity

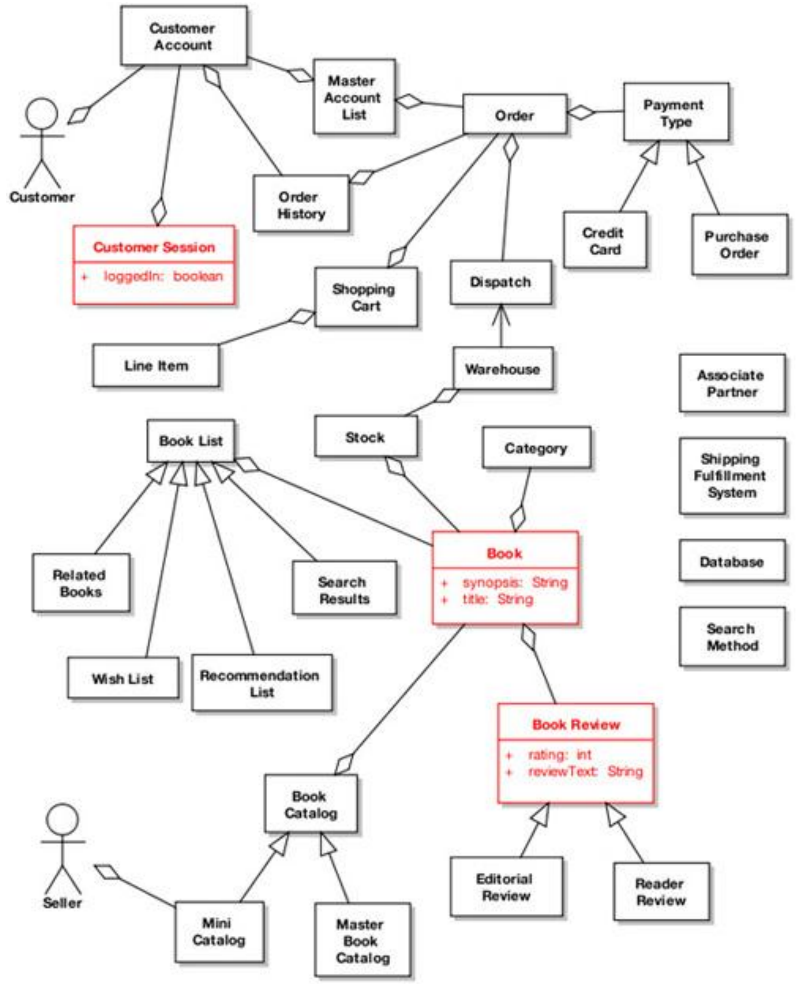
- There must be some way to get input, store and retrieve information, move information about the system, coordinate actions, make decisions, handle exceptions and failures.
- It must be a functioning system, able to defend itself, able to cope with and respond to events.
- It's alive – sort of.

UNIVERSITY OF TECHNOLOGY SYDNEY

STANDARD UML VIEWS

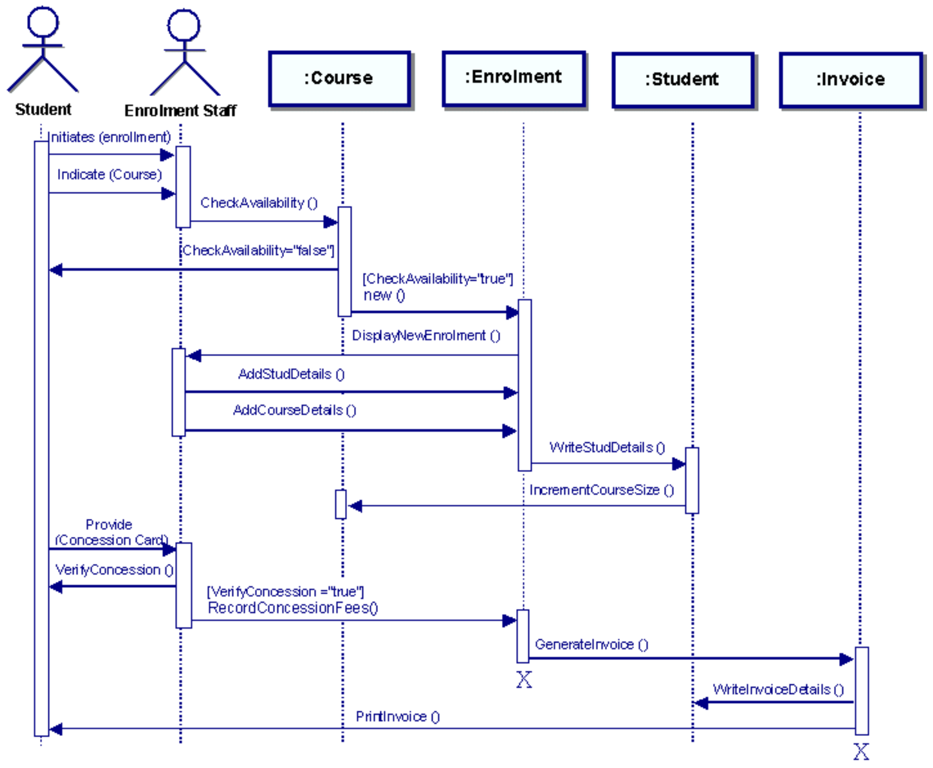
Class diagram: A static model does not vary much with time

- A static model provides a structural view of the problem, which does not vary with time.
- Describes the classes, their attributes, operations and relationships to other classes
- Does not describe movement of control or data through the system



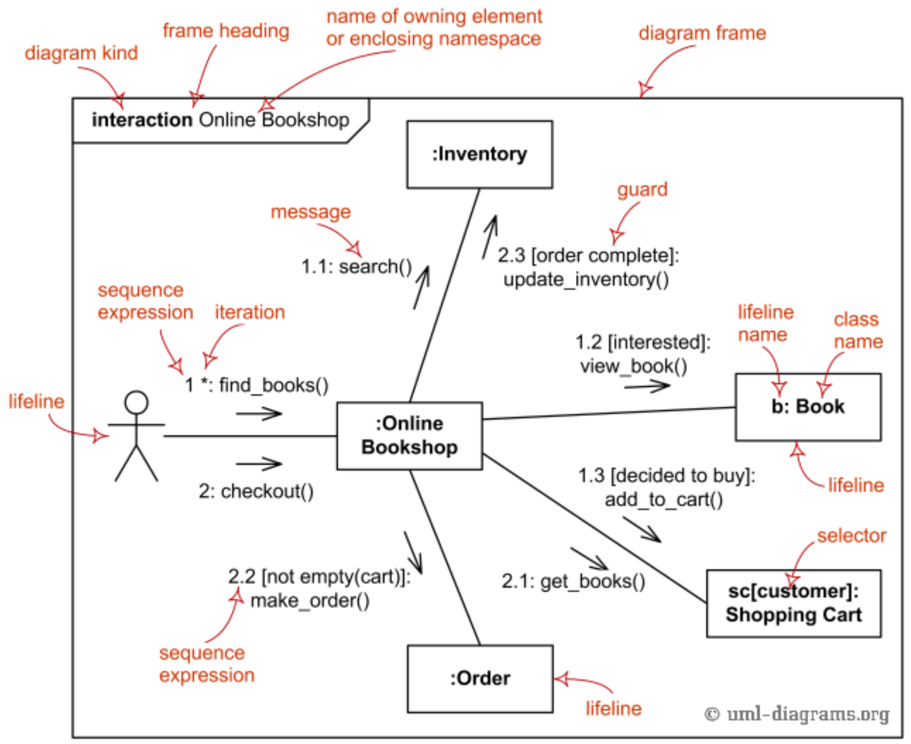
Sequence diagram: Shows interaction among objects

- Records the sequence of messages between objects
- Does not specify object methods or details of the message
- Useful to check the overall sequence of events.



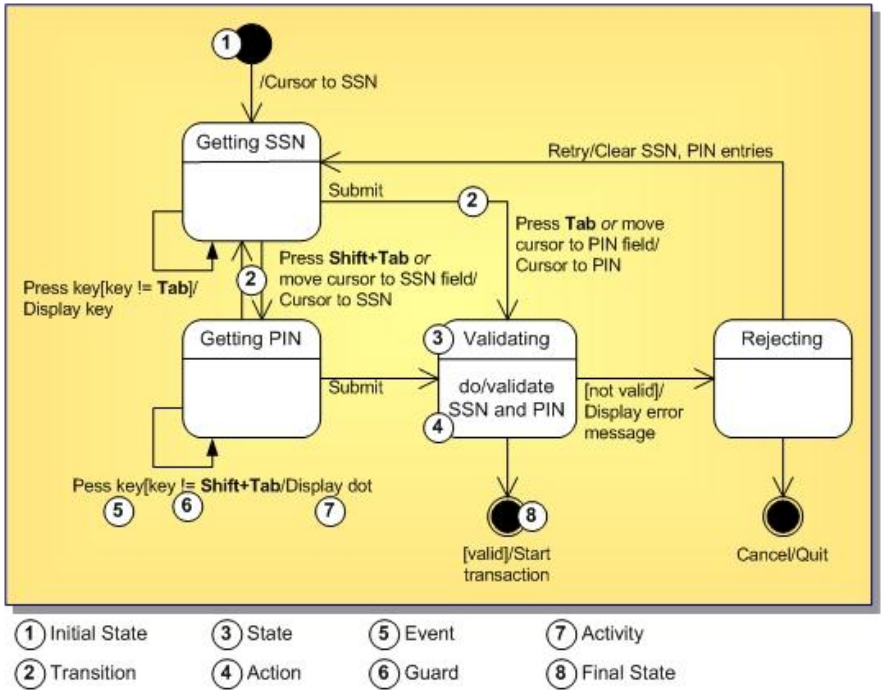
Communication diagram

- Some prefer communication diagrams for showing the dynamics of a system



State transitions with finite state diagrams

- Record the events that initiate a change of state
- Record the state (of the system)



1
/Cursor to SSN
Getting SSN
Retry/Clear SSN, PIN entries
Submit
2
Press Tab or move cursor to PIN field/ Cursor to PIN
Press Shift+Tab or move cursor to SSN field/ Cursor to SSN
2
Press key[key != Tab]/ Display key
Getting PIN
3 Validating
do/validate SSN and PIN
4
Rejecting
Submit
[not valid]/ Display error message
Pess key[key != Shift+Tab/Display dot
5
6
7
8
[valid]/Start transaction
Cancel/Quit
1 Initial State
2 Transition
3 State
4 Action
5 Event
6 Guard
7 Activity
8 Final State

UNIVERSITY OF TECHNOLOGY SYDNEY

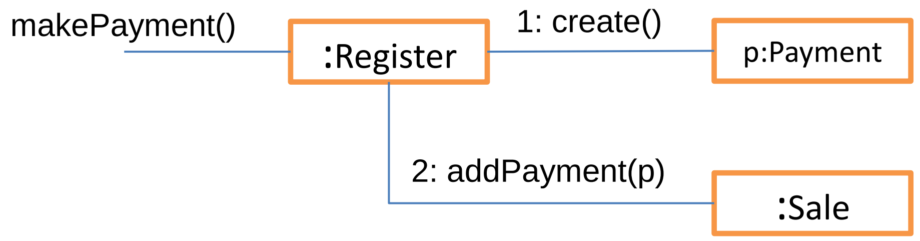
GENERAL RESPONSIBILITY ASSIGNMENT SOFTWARE PATTERNS (GRASP)

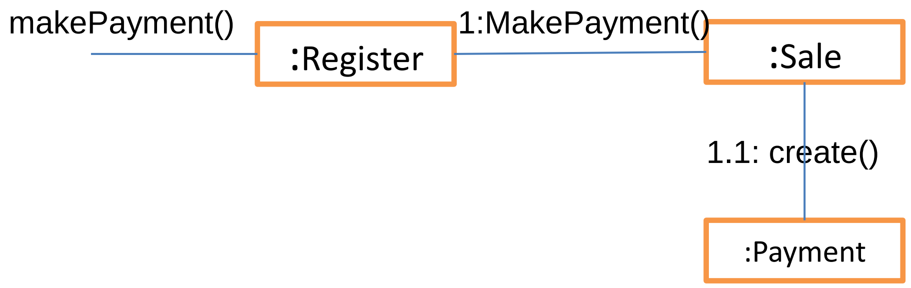
Nine GRASP Principles help to organize the conceptual architecture

- Low Coupling
- High Cohesion
- Information Expert
- Creator
- Controller
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Low Coupling

- Problem
  - How to encourage reuse and reduce the impact of change
- Solution
  - Assign responsibilities to minimize coupling





High coupling is undesirable because..

- Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
- An element with low (or weak) coupling is not dependent on too many other elements (classes, subsystems, …)
  - "too many" is context-dependent
- A class with high (or strong) coupling relies on many other classes.
  - Changes in related classes force local changes.
  - Such classes are harder to understand in isolation.
  - They are harder to reuse because its use requires the additional presence of the classes on which it is dependent.

High Cohesion Principle

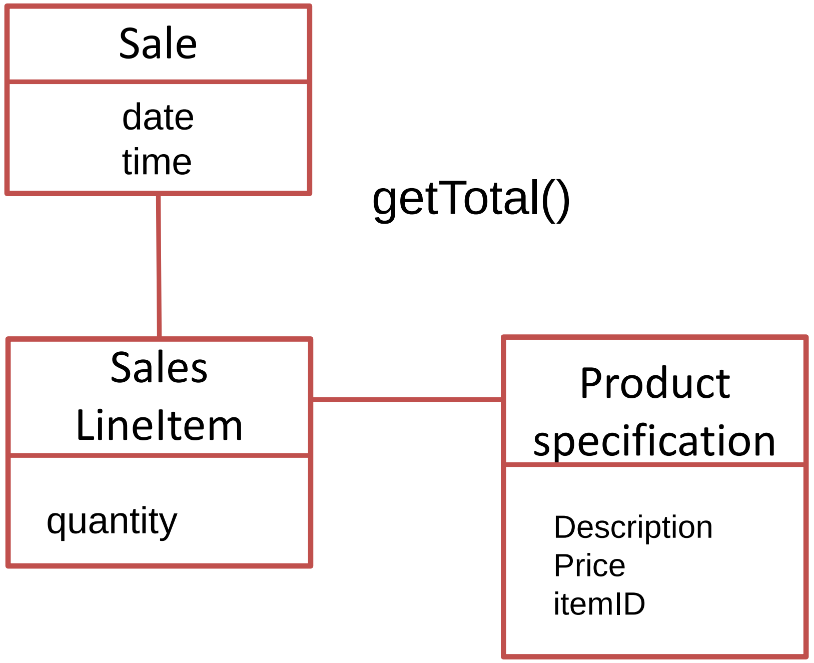
- Problem:
  - How to keep complexity manageable
- Solution
  - Assign responsibilities so that cohesion remains high
  - If elements are used together, keep them together
- Benefits
  - Classes are easier to maintain
  - Easier to understand
  - Often support low coupling
  - Supports reuse because of fine grained responsibility

Information Expert Principle

- Problem: What is a general principle of assigning responsibilities to objects?
- Solution: Assign a responsibility to the information expert, **the class that has the information necessary to fulfill the responsibility**
- Start assigning responsibilities by clearly stating responsibilities!
- Typically follows common intuition
- Design Classes (Software Classes) instead of Conceptual Classes
  - If Design Classes do not yet exist, look in Domain Model for fitting abstractions (-> low representational gap)

Example

- What information is needed to determine the grand total?
  - Line items and the sum of their subtotals
- Sale is the information expert for this



Creator Principle

- Who creates an object?
- Whoever is closest to it.
  – If A aggregates B, A creates B (aggregation)
  – If A contains B, A creates B (composition)
  – If A records instances of B, A creates B
  – If A closely uses B, A creates B
- Promotes low coupling by making class instances responsible for creating objects they need to reference

Controller Principle

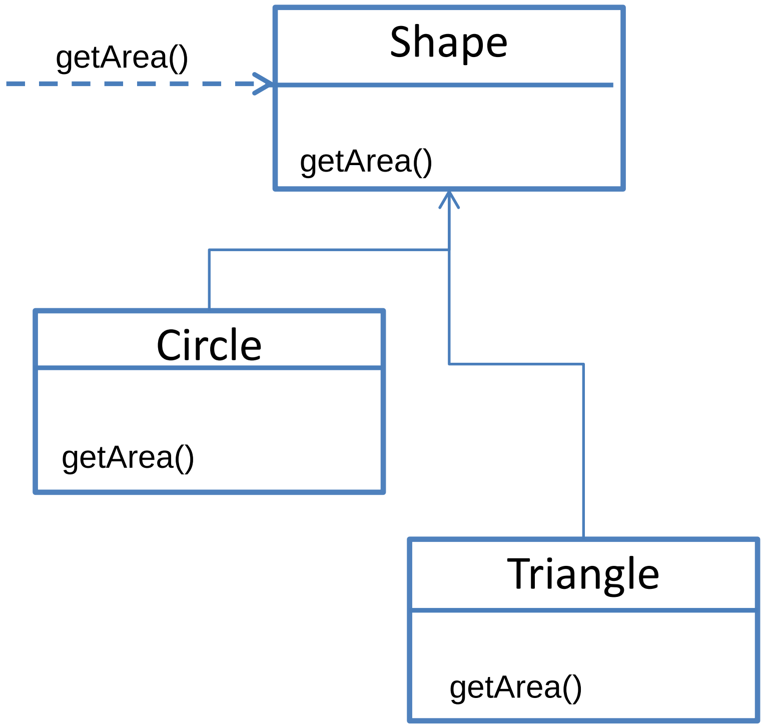
- Problem:
  - Who should be responsible for handling an input system event?
- Solution:
  - Assign the responsibility for receiving or handling a system event message to a class representing the overall system, device, or subsystem (facade controller) or a use case scenario within which the system event occurs (use case controller)
- Reasons
  - Avoids conflicts of priorities
  - Avoids conflicting decisions and resource allocations

Controller - Discussion

- Normally, a controller should delegate to other objects the work that needs to be done;
  - Coordinates or controls the activity.
  - Does not do much work itself.
- Facade controllers are suitable when there are not "too many" system events
- A use case controller is an alternative to consider when placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling
  - typically when the facade controller is becoming "bloated" with excessive responsibilities.

Polymorphism

- Where to assign responsibility to handle related but varying elements based on element type
- Assign it to the subtype



Pure Fabrication

- Create a fabricated class that doesn’t represent any domain object but is required in the system.
- Provides a cohesive set of activities.
- Possibly implements some algorithm or library of methods.
- Examples: Adapter, Strategy

- Example
  - Suppose we have a Shape class, if we must store the shape data in a database.
  - If we put this responsibility in Shape class, there will be many database related operations thus making Shape incoherent.
  - So, create a fabricated class DBStore which is responsible to perform all database operations.
  - Similarly logInterface which is responsible for logging information is also a good example for Pure Fabrication

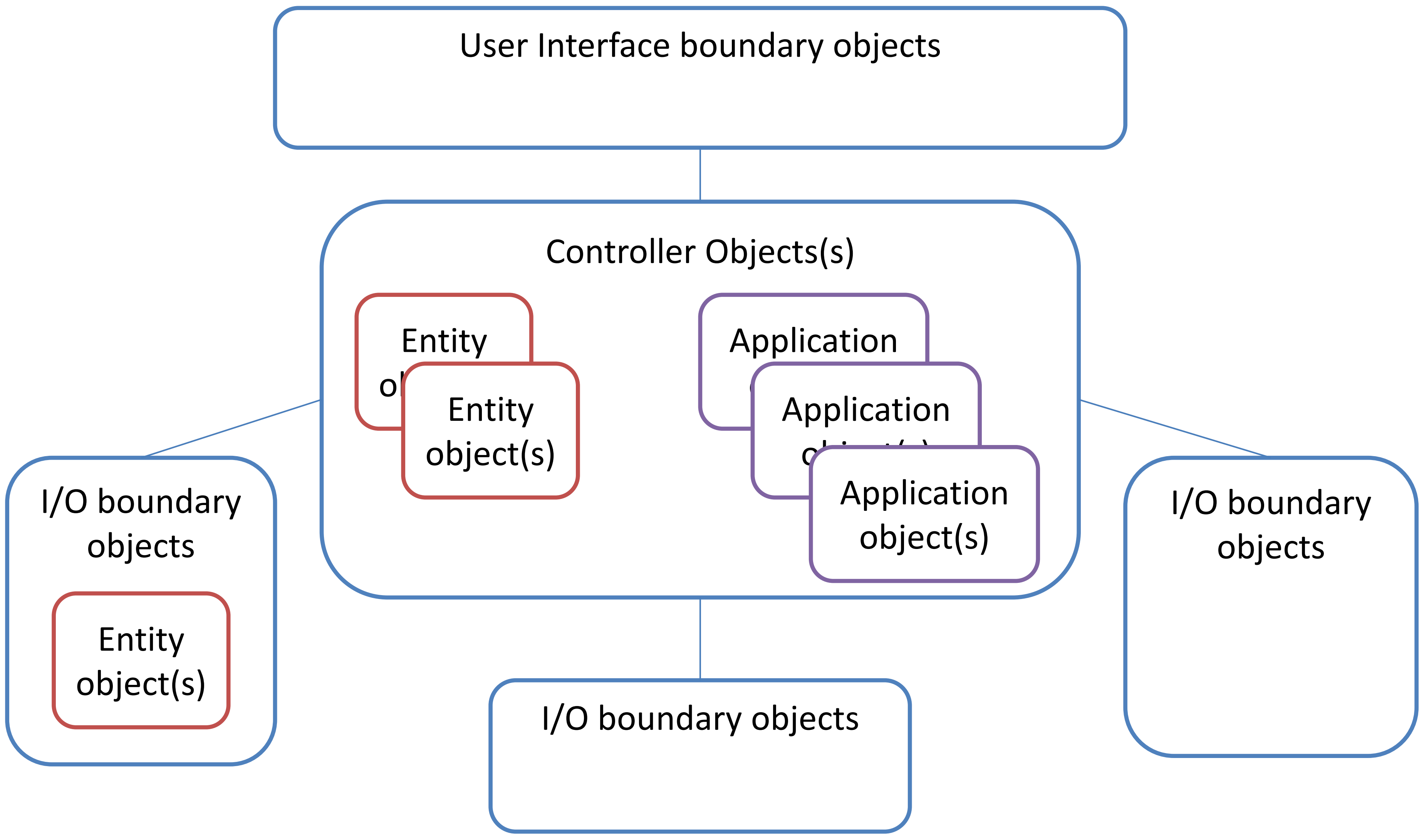
UNIVERSITY OF TECHNOLOGY SYDNEY

FIRST PASS SYSTEM STRUCTURE

From Object-Oriented design to a structured architecture

- A system must function on its own, interacting with the world, responding to events, handling exceptions
- A system has some of the characteristics of a living system, but only some.
- Hassan Gomaa finds it useful to classify objects as
  - Entity
  - Boundary
  - Controller
  - Application logic
- These can help convert an O-O design into an architecture by structuring the objects into useful categories with which to construct a system.

The broadest possible architecture



Indirection Principle

- Problem
  - Direct couplings between classes leads to high maintenance
  - Change one and all connected classes need a matching change
- Solution
  - Insert an intermediate element between classes
  - Façade, Adapter, Observer
  - Can also be an abstract class or interface

Protected variation

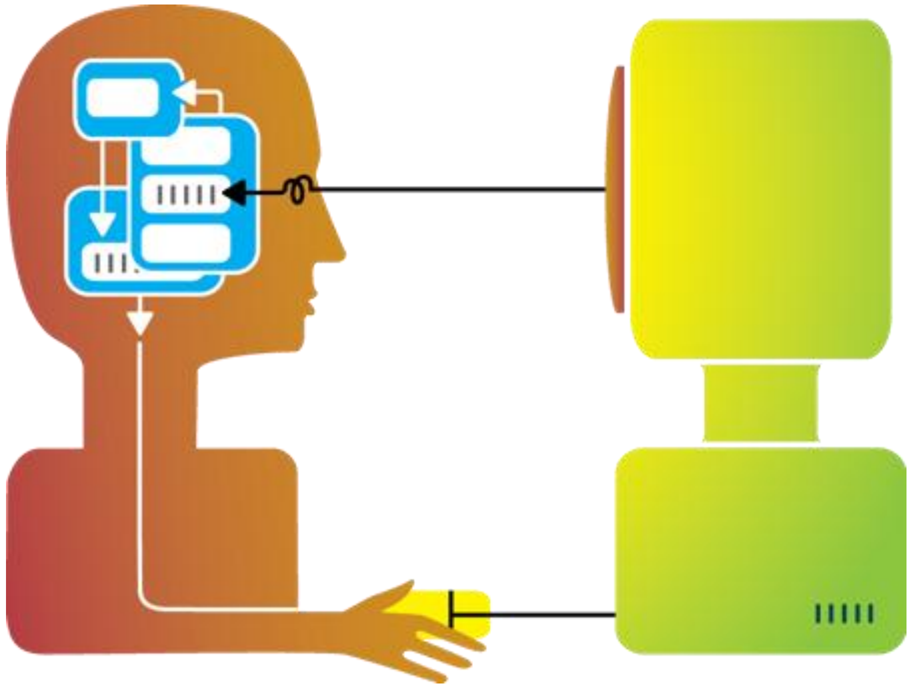
- Problem:
  - How to protect elements from the impact of variations in the elements they are connected to
- Solution
  - Provide a well defined interface
- Provides flexibility and protection from variations
- Provides a more structured design

Hassan Gomaa's Object and Class structuring categories

- Entity
  - A software object, in many cases persistent, which encapsulates information and provides access to the information it stores.
- Boundary object
  - Software that interfaces to and communicates with the external environment
  - User interaction object
  - Proxy object
  - Device I/O object
- Control object
  - Software that provides overall coordination for a collection of objects.
  - Coordinator objects
  - State-dependent objects
  - Timer objects
- Application logic objects
  - A software object that contains the details of the application logic
  - Business logic
  - Algorithm logic
  - Service logic

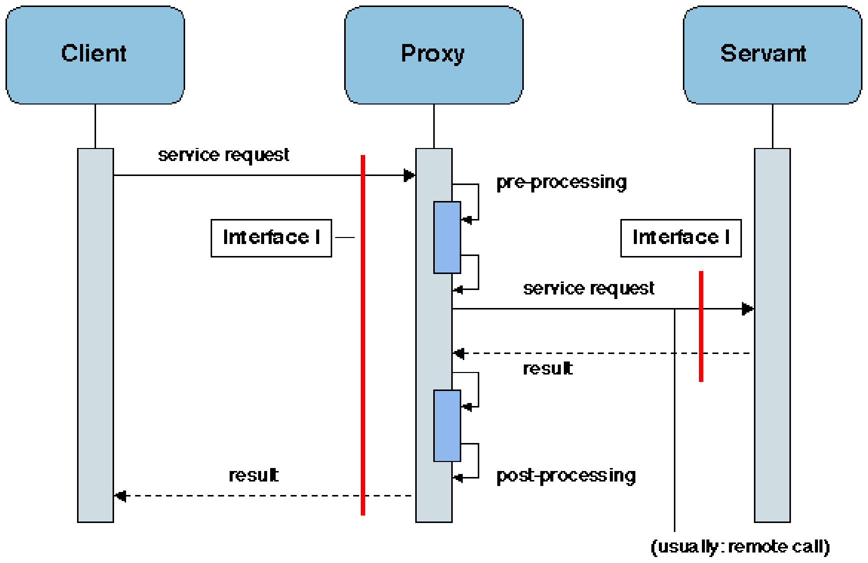
Boundary: User Interaction object

- Communicates directly with a human user.



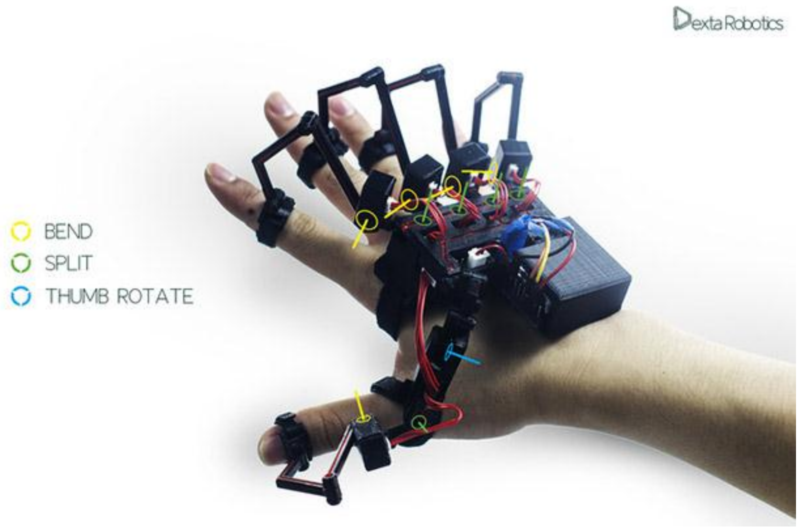
Boundary: Proxy object

- Interfaces to and communicates with external systems



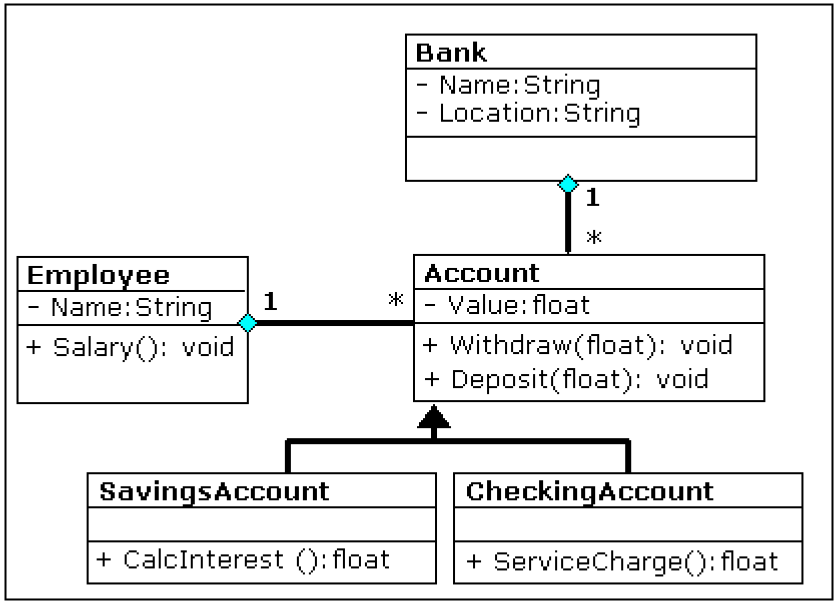
Boundary : Device I/O object

- Provides a software interface to a hardware object.
- Usually needed for non-standard application specific devices, prevalent in real-time systems.



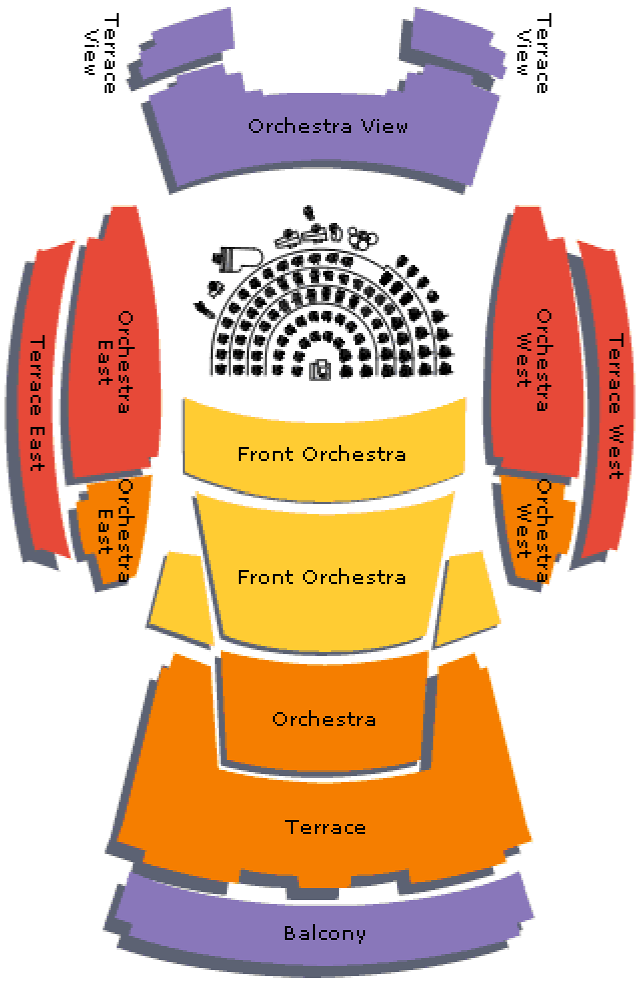
Entity Classes and Objects

- Primary purpose is to represent something that the system manipulates.
- In many systems the information encapsulated by an entity object is stored in a file or database. Persistent information.
- Example is an “Account” in a banking application.



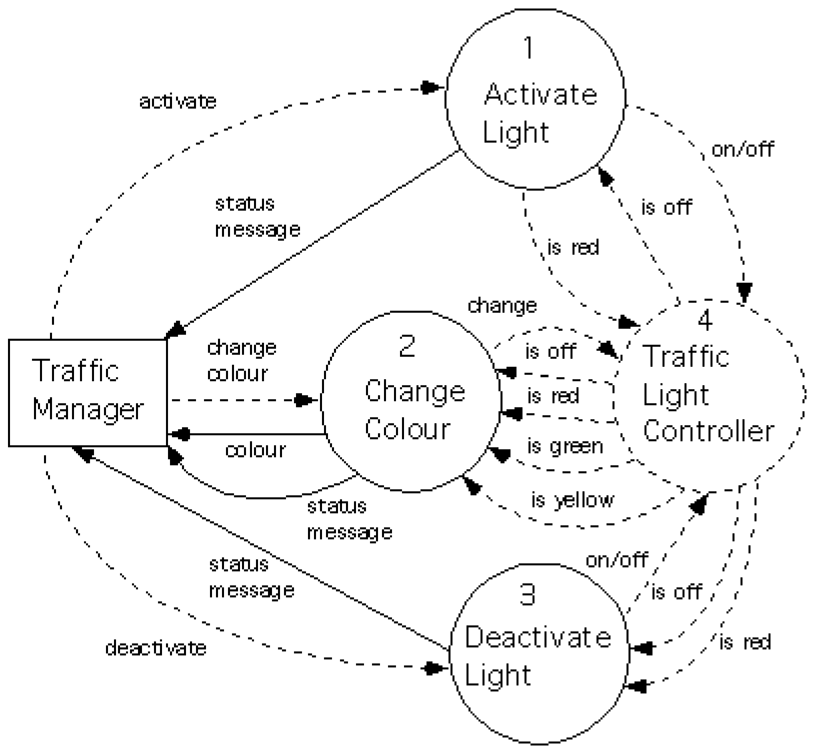
Control: Coordinator classes and objects

- Overall decision-making object that determines sequencing of a collection of related objects and coordination among them.
- Example; Sell a ticket to a concert. Hold ticket until final sale, get payment, send ticket, remove seat from available.
- Responds to events and is not state dependent.



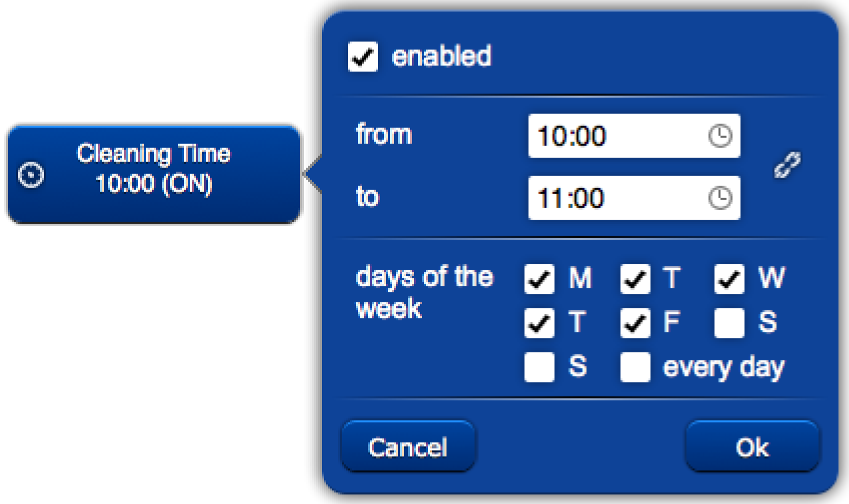
Control: State-Dependent Control Objects

- Behaviour varies in each of its states.
- Receive incoming events, transition to new state, generate outputs that control other objects.



Control: Timer Objects

- Receive an interrupt from an external clock, perform an action or activate another object to perform an action


Cleaning Time
10:00 (ON)

✓ enabled

from 10:00
to 11:00

days of the week: ✓ M ✓ T ✓ W ✓ T ✓ F ☐ S ☐ S ☐ every day

Application logic: Business Logic Objects

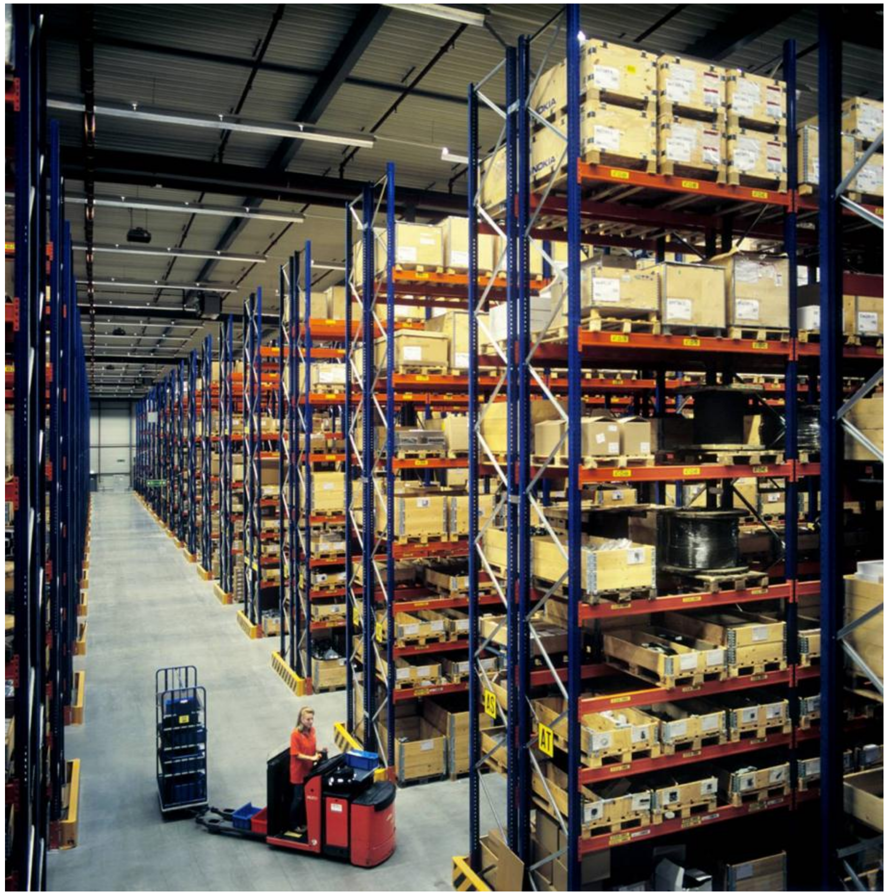
- Defines the business-specific application logic for processing a client request.
- Separate the business rules from entities because they change independently.

Application logic: Algorithm Objects

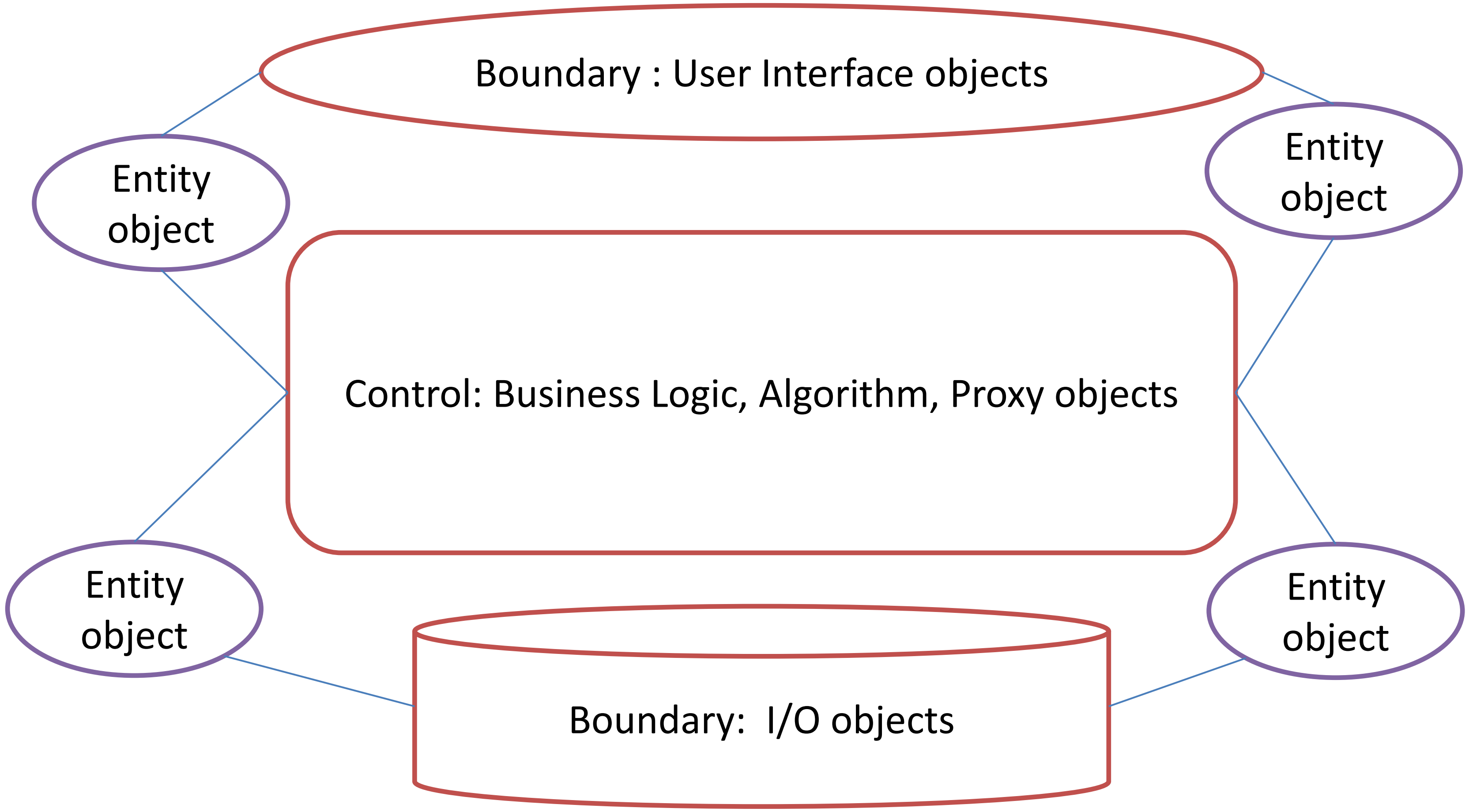
- Encapsulates and algorithm used in the problem domain.
- Example – encryption algorithm

Application logic: Service objects

- Provide a service on request.
- Does not initiate a request but may seek assistance from other service objects. (seems to be a matter of granularity)



Now the system can be structured in terms of object types


Boundary : User Interface objects
Entity object
Entity object
Control: Business Logic, Algorithm, Proxy objects
Entity object
Entity object
Boundary: I/O objects

UNIVERSITY OF TECHNOLOGY SYDNEY

EVALUATING A CONCEPTUAL ARCHITECTURE

What can be evaluated?

- Strong ability to evaluate the functional characteristics

- Limited ability to evaluate achievement of the non-functional goals

- No ability to evaluate any implementation dependent goals

Is it SOLID

- Have the SOLID principles been applied
  - Single responsibility
  - Open – Closed
  - Liskov substitution
  - Interface segregation
  - Dependency inversion

Have responsibilities been assigned where they belong

- Low coupling
- High cohesion
- Information expert
- Creator
- Controller
- Polymorphism
- Protected variations
- Pure fabrication

Will this model satisfy the business needs?

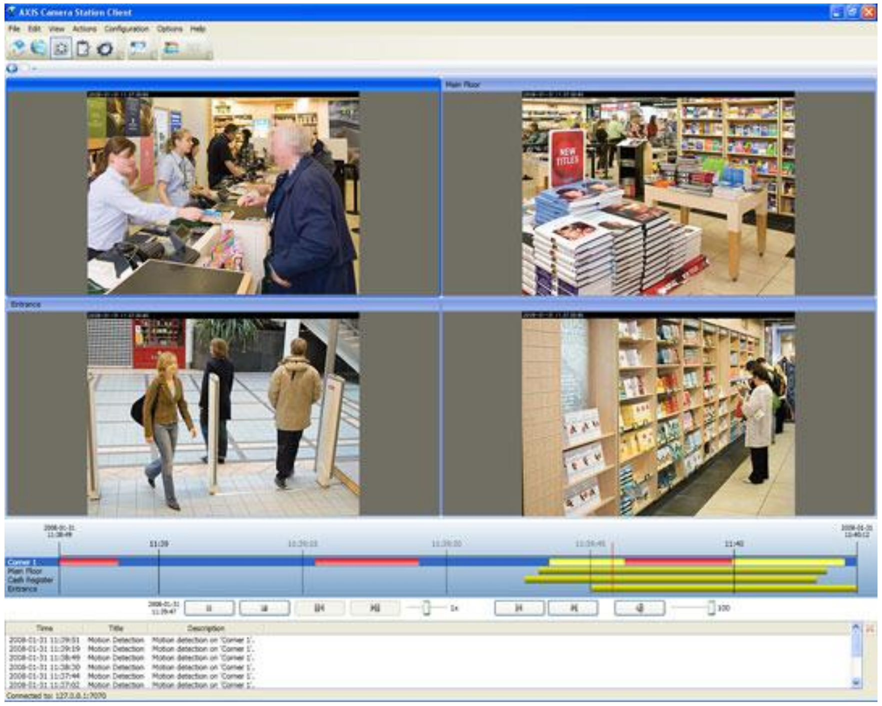
- Trace a representative sample of transactions through the system
- As with all reviews, a more rigorous review is better even though it takes more effort.

Will the architecture support the expected exceptions

- Most of the effort is spent dealing with exception conditions.
- We tend not to think of the exceptions, but deal with them on a case by case basis.
- A computer system can't invent a response so they will have to be anticipated.
- Workflow exceptions
- State transition exceptions
- Failure conditions

Will the architecture deal with foreseeable stresses

- Web storm
- Hacker attack
- Multiple simultaneous events
- Exceptionally large data

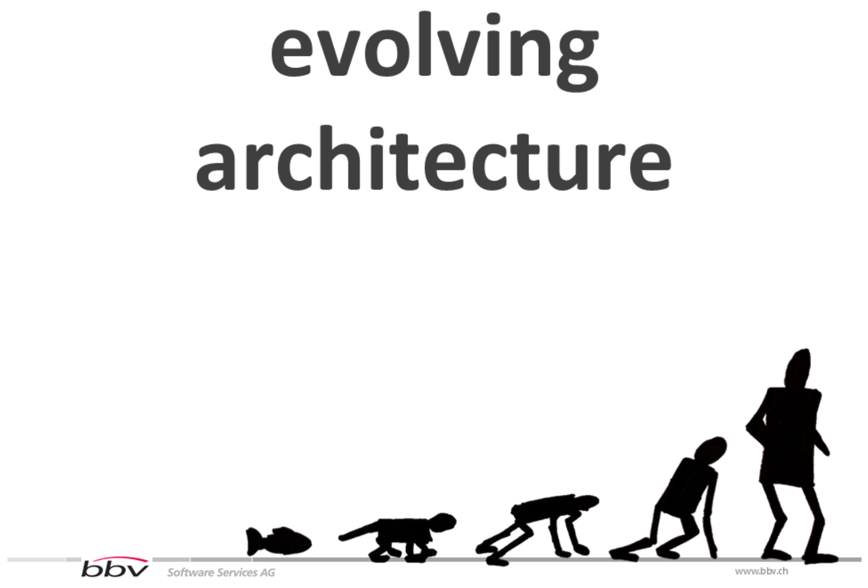

AXIS Camera Station Client
File Edit View Actions Configuration Options Help
Main Floor
Entrance
NEW TITLES
Corner 1
Main Floor
Cash Register
Entrance

Time	Title	Description
2008-01-31 11:39:51	Motion Detection	Motion detection on 'Corner 1'.
2008-01-31 11:39:19	Motion Detection	Motion detection on 'Corner 1'.
2008-01-31 11:38:49	Motion Detection	Motion detection on 'Corner 1'.
2008-01-31 11:38:30	Motion Detection	Motion detection on 'Corner 1'.
2008-01-31 11:37:44	Motion Detection	Motion detection on 'Corner 1'.
2008-01-31 11:37:02	Motion Detection	Motion detection on 'Corner 1'.

Connected to: 127.0.0.1:7070

Will the architecture cope with foreseeable evolution

- Additional transaction types, customer types, etc.
- Vastly increased transaction rate
- Vastly increased customer base
- Adaptation to a different business
- Significant modification



Summary

- Have the SOLID and GRASP principles been observed.
- Does the architecture support the expected business functions.
- Does the architecture support the expected exception conditions.
- Can the architecture support expected business evolution.
- Can the architecture support expected system evolution.

UNIVERSITY OF TECHNOLOGY SYDNEY

Single responsibility
Open Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

SOLID PRINCIPLES OF OBJECT ORIENTED CLASS DESIGN

Single responsibility

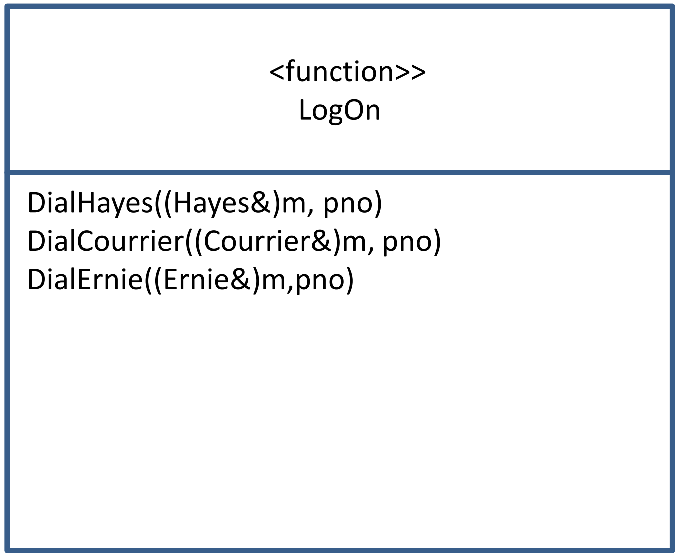
- Every class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.
- Example – a module compiles and prints a report
  - Content could change
  - Format could change
  - These are two separate responsibilities and should be handled by different classes

Open Closed Principle

- **A module should be open for extension but closed for modification.**
- We should write our modules so that they can be extended, without requiring them to be modified.
- In other words, we want to be able to change what the modules do, without changing the source code of the modules.

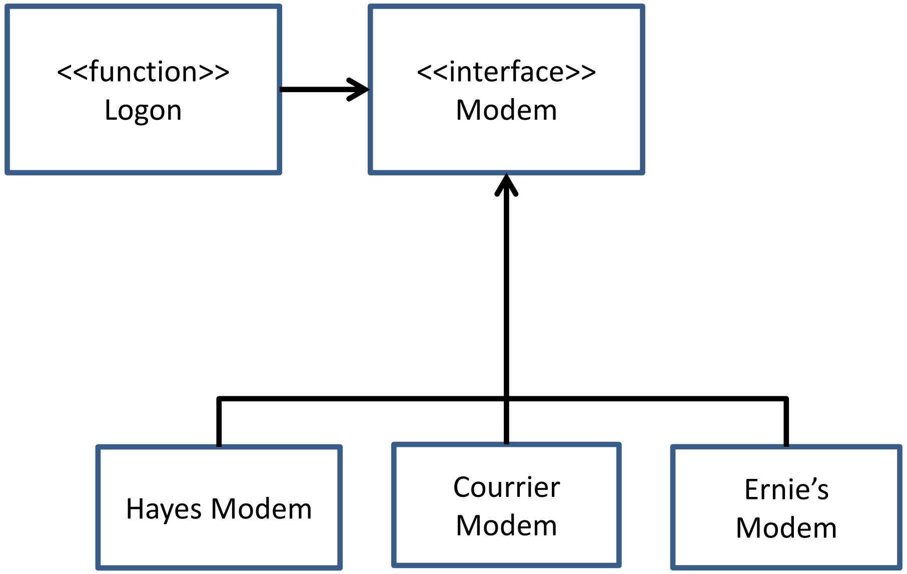
Open Closed Principle example

- The function must be changed every time a new modem is added
- Develop a logon to a modem that can be extended by adding new modems as they come available.



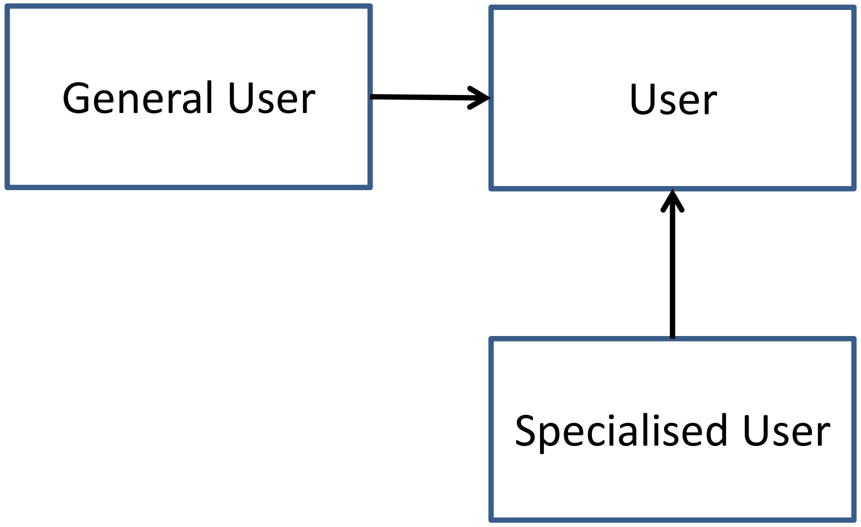
A better modem logon

```
class modem
{
public:
virtual void dial (const string& pno) = 0;
virtual void Send(char) = 0;
virtual char Recv() = 0;
virtual void Hangup() = 0;
};
void LogOn (Modem& m, string& pno, string& user,
string& pw)
{
m.Dial(pno);
// you get the idea
}
```



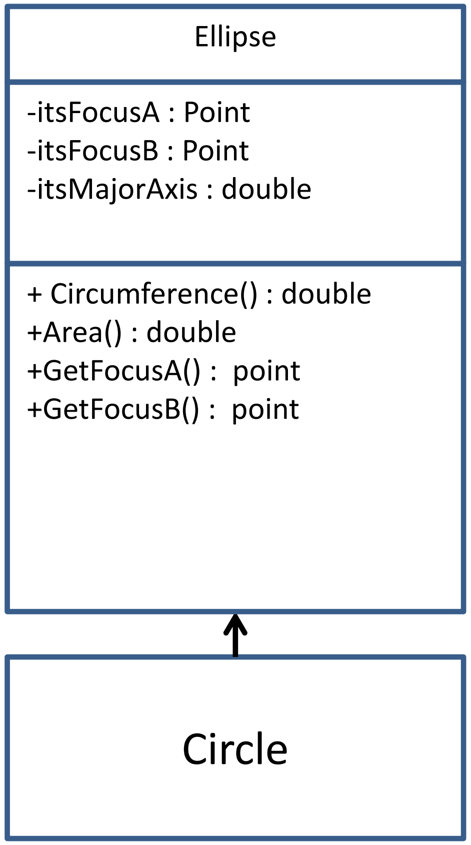
Liskov Substitution Principle

- **Subclasses should be substitutable for their base classes.**
- Derived classes should be substitutable for their base classes.
- That is, a user of a base class should continue to function properly if a derivative of that base class is passed to it.



Circle/Ellipse Dilemma

- We are all taught that a circle is a degenerate ellipse – one with coincident foci
- You can make this work by setting both foci to the same value within circle
- But it's a hack and it can fail



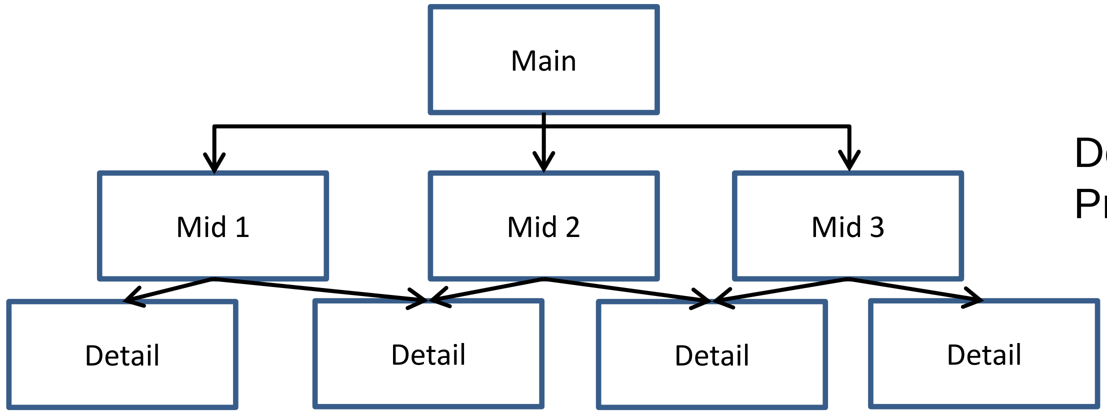
Interface Segregation Principle

- No client should be forced to depend on methods it does not use.
- Interface Segregation Principle splits interfaces which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.
- Such shrunken interfaces are also called *role interface*s.
- Interface Segregation Principle is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

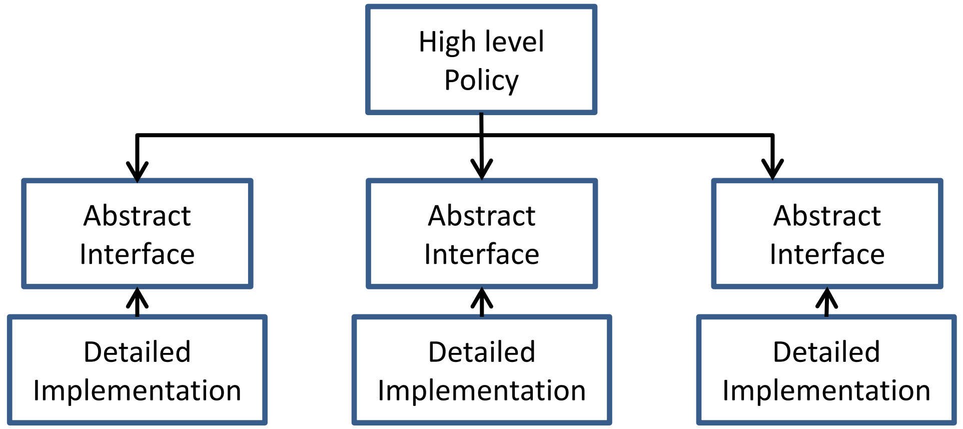
Dependency Inversion Principle

- **Depend upon Abstractions. Do not depend upon concretions**.
- Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes.
- This principle is the enabling force behind component design, COM, CORBA, EJB, etc.
- The implication of this principle is quite simple.
  - Every dependency in the design should target an interface, or an abstract class.
  - No dependency should target a concrete class.

Dependency structures



Dependency structure of a Procedural Dependency



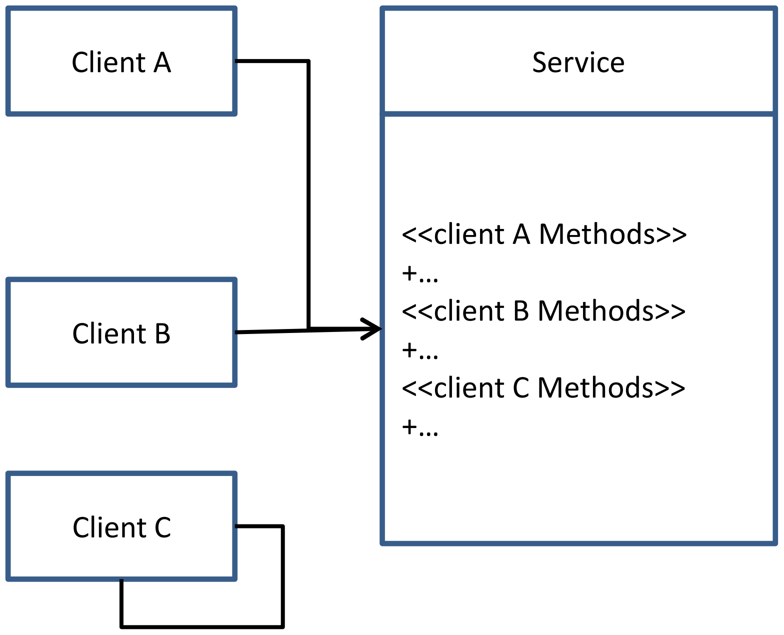
Dependency structure of an Object Oriented Architecture

Interface Segregation Principle

- **Many client specific interfaces are better than one general purpose interface**

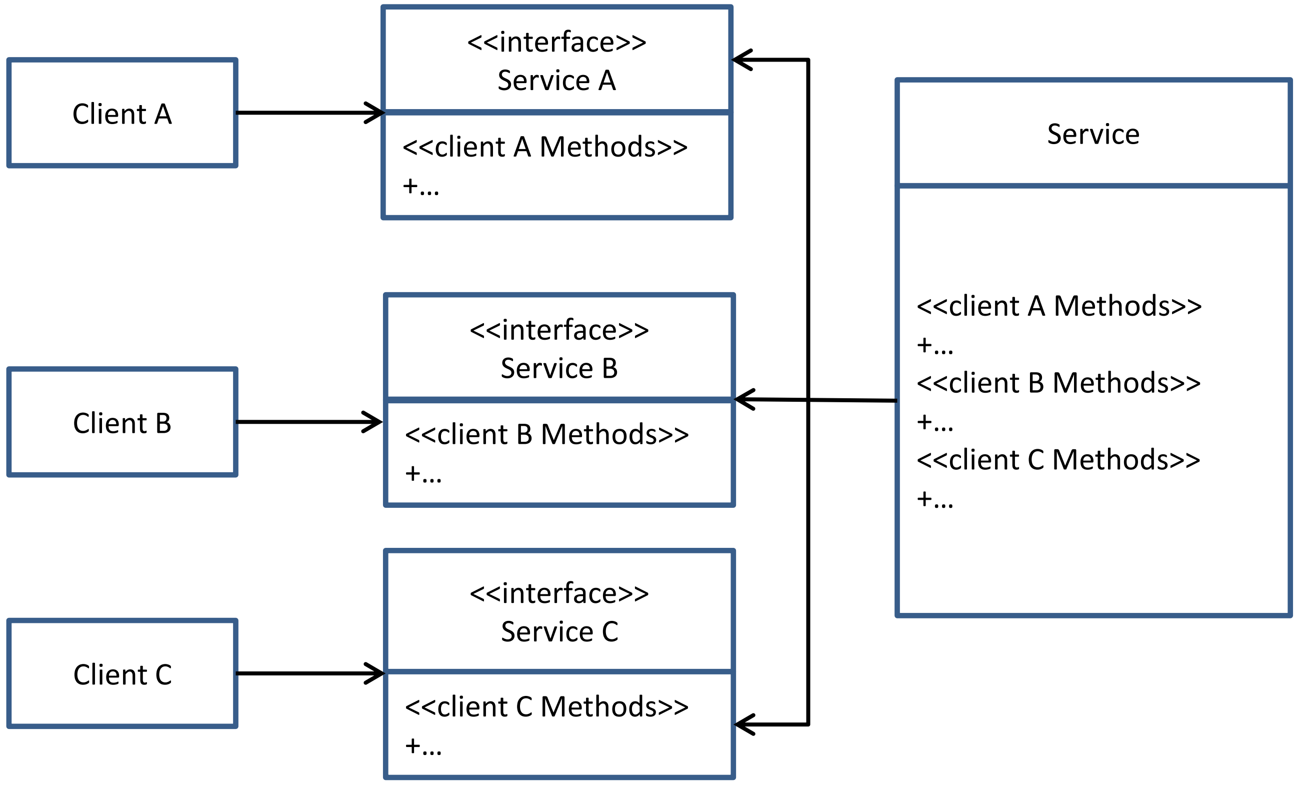
- If you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each client and multiply inherit them into the class.

Fat Service with Integrated Interfaces



- a class with many clients, and one large interface to serve them all.
- Note that whenever a change is made to one of the methods that ClientA calls, ClientB and ClientC may be affected. It may be necessary to recompile and redeploy them.
- This is unfortunate.

With Segregated Interfaces



Cautions on client specific interfaces

- The ISP does not recommend that every class that uses a service have its own special interface class that the service must inherit from.
- When object oriented applications are maintained, the interfaces to existing classes and components often change.
- There are times when these changes have a huge impact and force the recompilation and redeployment of a very large part of the design.
- This impact can be mitigated by adding new interfaces to existing objects, rather than changing the existing interface.
- This impact can be mitigated by adding new interfaces to existing objects, rather than changing the existing interface.
- As with all principles, care must be taken not to overdo it. The specter of a class with hundreds of different interfaces, some segregated by client and other segregated by version, would be frightening indeed.

Summary

- Open Closed Principle
  - *A module should be open for extension but closed for modification.*
- Liskov Substitution Principle
  - *Subclasses should be substitutable for their base classes.*
- Dependency Inversion Principle
  - *Depend upon Abstractions. Do not depend upon concretions.*
- Interface Segregation Principle
  - *Many client specific interfaces are better than one general purpose interface*
- Design by Contract