G64OOS Spring 2014

Lecture 11

Exam Revision

Peer-Olaf Siebers

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

Motivation

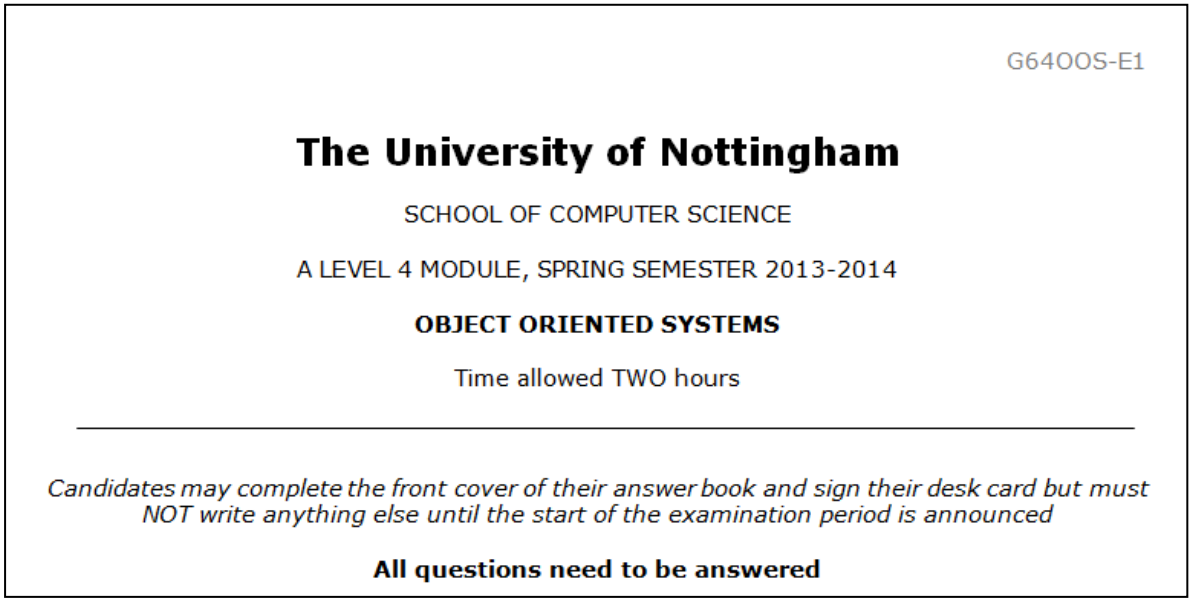
- Refresh your knowledge of OOAD/P
- Prepare you for the exam with some interactive tasks

Overview of Lecture Topics

- Introduction
- Object Oriented System Analysis
- Object Oriented System Modelling
- Object Oriented Programming (Principles)
- Object Oriented Programming in C++ (Part 1)
- Object Oriented Programming in C++ (Part 2)
- Agile Software Development
- Testing + Test Driven Development
- Elements of Reusable Object Oriented Software

Exam Format and Content

- Three compulsory questions (THIS IS NEW)
  - One question requires OOA/D knowledge
  - One question requires OOP knowledge
  - One question requires ???



G64OOS-E1

The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 4 MODULE, SPRING SEMESTER 2013-2014

OBJECT ORIENTED SYSTEMS

Time allowed TWO hours

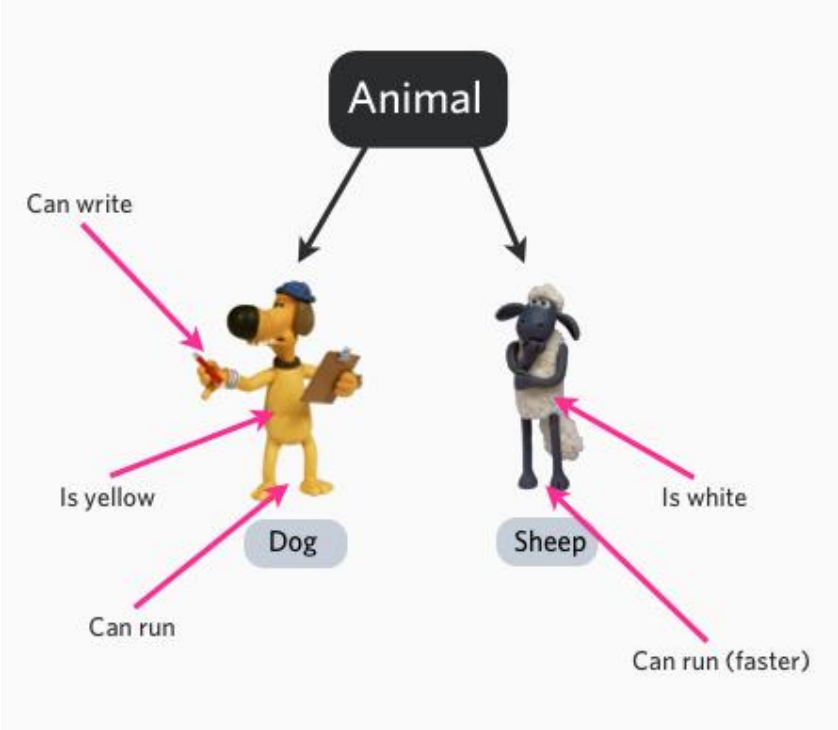
Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced

All questions need to be answered

Object Oriented Basics and Principles

The Object Model [Booch 1994]

- "A sound engineering foundation"
  - Four basic principles:
    - Abstraction
    - Encapsulation
    - Modularity
    - Hierarchy



The Object Model [Booch 1994]

- Abstraction
  - "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."
  - Key concepts:
    - Concentrating only on essential characteristics: Allows complexity to be more easily managed
    - Abstraction is relative to the perspective of the viewer: Many different views of the same object are possible

The Object Model [Booch 1994]

- Encapsulation:
  - "Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation"
  - It associates the code and the data it manipulates into a single unit; it keeps them save from external interference and misuse
  - Key concepts:
    - Packaging structure and behaviour together in one unit: Makes objects more independent
    - Objects exhibit an interface through which others can interact with it: Hides complexity from an object's clients

The Object Model [Booch 1994]

- Modularity:
  - "Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."
  - The unit of modularity in the object oriented world is the class
  - Key concepts:
    - Modules are cohesive (performing a single type of tasks): Makes modules more reusable
    - Modules are loosely coupled (highly independent): Makes modules more robust and maintainable

The Object Model [Booch 1994]

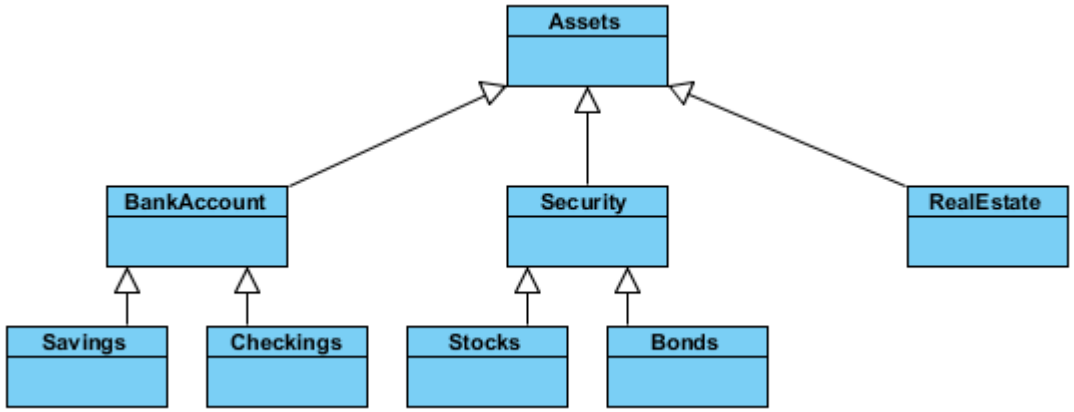
- Hierarchy:
  - "Hierarchy is a ranking or ordering of abstractions."

- Types of hierarchies:
  - Class
  - Aggregation
  - Containment
  - Inheritance
  - Partition
  - Specialization

The Object Model [Booch 1994]

- Hierarchy:
  - "Hierarchy is a ranking or ordering of abstractions."

- Classes at the same level of the hierarchy should be at the same level of abstraction



Other Object Oriented Concepts

- Data Abstraction
  - The technique of creating new data types that are well suited to an application; allows the creation of user defined data types, having the properties of build in data types and a set of permitted operators
  - Abstract data type: A structure that contains both, data and the actions to be performed with that data
  - Class is an implementation of an abstract data type

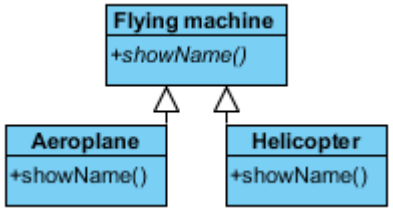
Other Object Oriented Concepts

- Inheritance
  - New data types (classes) can be defined as extensions to previously defined types; parent class = super class; child class = sub class; subclass inherits properties from super class
  - If multiple classes have common attributes and methods, these attributes and methods can be moved to a common class (parent class); this allows reuse since the implementation is not repeated

Other Object Oriented Concepts

- Polymorphism
  - Polymorphism through method overloading (design time p.)
    - Create more than one function with same name but different signatures
  - Polymorphism through method overriding (run time p.)
    - Create a function in the derived class with the same name and signature
  - Polymorphism through sub classing (run time p.)
    - A pointer of a base class is able to reference, instantiate and destroy objects of a derived class
    - A pointer may reference objects of many classes at runtime, but the compiler cannot predict which they will be at compile time

```
FlyingMachine* flyer
flyer = new Aeroplane;
flyer = new Helicopter;
```



SOLID Design Principles

- Software solves real life business problems and real life business processes evolve and change - always.
- A smartly designed software can adjust changes easily; it can be extended, and it is re-usable.

- SOLID Principles (by Uncle Bob) [http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod]
  - S = Single Responsibility Principle
  - O = Open-Closed Principle
  - L = Liscov Substitution Principle
  - I = Interface Segregation Principle
  - D = Dependency Inversion Principle

SOLID Design Principles

- Single Responsibility Principle
  - A class should have one and only one responsibility
- Open-Closed Principle
  - Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification
- Liskov's Substitution Principle
  - Subtypes must be substitutable for their base types

SOLID Design Principles

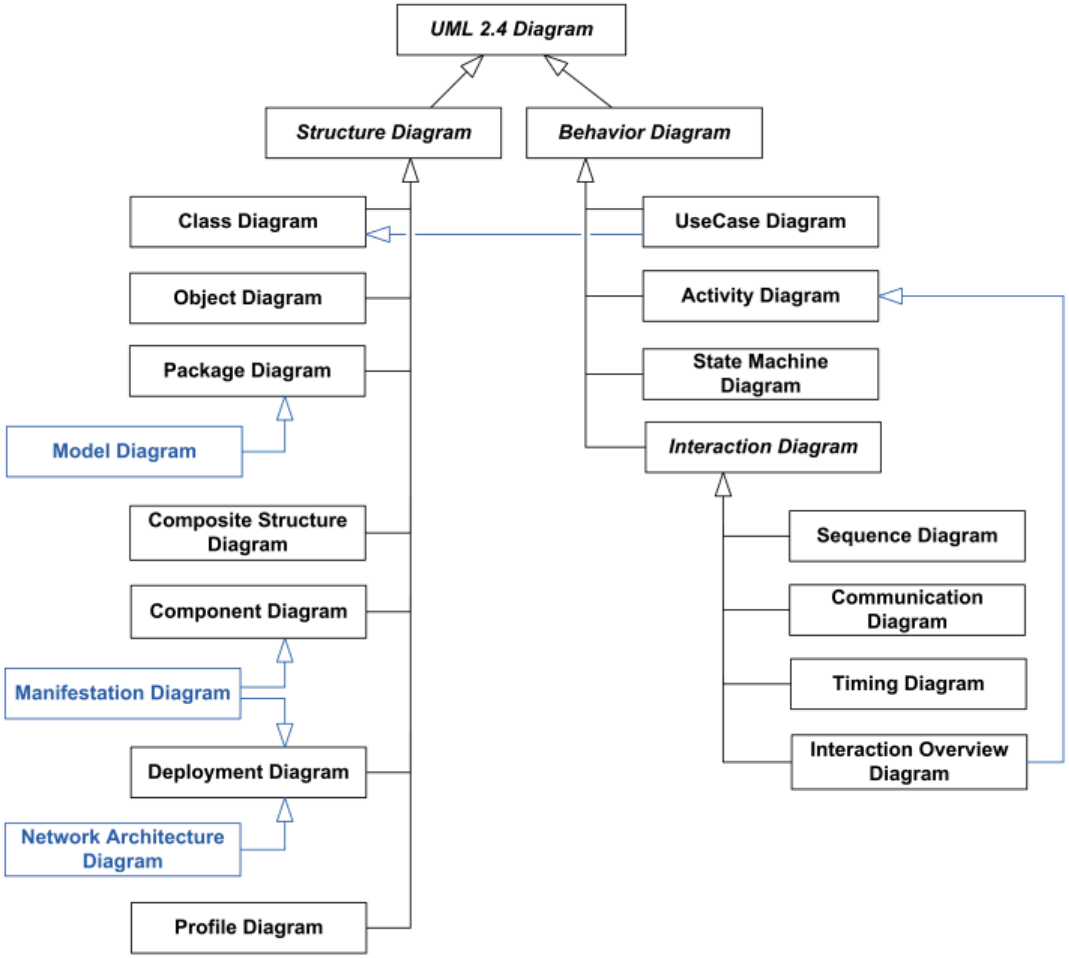
- Interface Segregation Principle
  - Clients should not be forced to depend upon interfaces that they do not use

- Dependency Inversion Principle
  - High level modules should not depend upon low level modules. Rather, both should depend upon abstractions

http://www.codeproject.com/Articles/93369/How-I-explained-OOD-to-my-wife

Object Oriented Analysis and Design

UML


UML 2.4 Diagram
Structure Diagram
Behavior Diagram
Class Diagram
Object Diagram
Package Diagram
Model Diagram
Composite Structure Diagram
Component Diagram
Manifestation Diagram
Deployment Diagram
Network Architecture Diagram
Profile Diagram
UseCase Diagram
Activity Diagram
State Machine Diagram
Interaction Diagram
Sequence Diagram
Communication Diagram
Timing Diagram
Interaction Overview Diagram

Use Case Modelling

- The User Story
  - Stating the need
  - Collecting and prioritising high-level features
  - Should be written by project stakeholders and not the developers
  - Keep it simple!

Case Study: Hotel Rooming System

LODGEGATE

Hotel setup | Reservations | Front desk | Accounting | Night audit | Rooms management | Reporting | Tools | Exit

Search reservation : Reservation search

New search

Room	RT code	Guest name	Spouse name	Conf#	Arrival	Departure	Rate code	NoR	NoG	Status	Guest msg
223	SUP-A	G.K. Bloemsma		46720	Thu 08/11/2007	Sat 10/11/2007	0 RATES ONLINE	1	2 /0/0	1	
109	TW-B	Geert Aslander		45681	Fri 09/11/2007	Sun 11/11/2007	AIRMILES € 59,50	1	2 /0/0	1	
112	TW-B	F. Beltman	Mw. R. Beltman-Koenjer	46445	Fri 09/11/2007	Mon 12/11/2007	KRAS 3N	1	2 /0/0	1	
114	TW-B	M. Bezuijen		45996	Fri 09/11/2007	Sun 11/11/2007	WW € 109	1	2 /0/0	1	
124	TW-A	M. Boonen	Mw. M. Boonen-Hannessen	46468	Fri 09/11/2007	Mon 12/11/2007	KRAS 3N	1	2 /0/0	1	
220	TW-A	M. Broeksteeg	Mw. A. Broeksteeg-Wolf	46648	Fri 09/11/2007	Sat 10/11/2007	KRAS 2N	1	2 /0/0	1	
129	SUP-B	Miranda Bulsing		46657	Fri 09/11/2007	Sun 11/11/2007	0 RATES ONLINE	1	2 /0/0	1	
215	SUP-A	Henoch van der Burgh		46719	Fri 09/11/2007	Sun 11/11/2007	0 RATES ONLINE	1	2 /0/0	1	
116	TW-B	A. Crucq	Mw. A. Rentmeester	46509	Fri 09/11/2007	Mon 12/11/2007	KRAS 3N	1	2 /0/0	1	
118	TW-B	Willemijn Dres		46375	Fri 09/11/2007	Sun 11/11/2007	AIRMILES € 59,50	1	2 /0/0	1	
202	TW-A	Frisse Wind 09 nov. 07		46178	Fri 09/11/2007	Sat 10/11/2007	DI 109	1	3 /0/0	1	
213	DB-A	Frisse Wind 09 nov. 07		46179	Fri 09/11/2007	Sat 10/11/2007	DI 99	1	1 /0/0	1	
910	PM	Frisse Wind 09 nov. 07		46180	Fri 09/11/2007	Sat 10/11/2007	EMP	1	1 /0/0	1	
123	TW-A	Frisse Wind 09 nov. 07		46689	Fri 09/11/2007	Sat 10/11/2007	DI 109	1	2 /0/0	1	
125	TW-A	Frisse Wind 09 nov. 07		46690	Fri 09/11/2007	Sat 10/11/2007	DI 109	1	2 /0/0	1	
127	TW-A	Frisse Wind 09 nov. 07		46691	Fri 09/11/2007	Sat 10/11/2007	DI 109	1	2 /0/0	1	
219	DB-A	Frisse Wind 09 nov. 07		46693	Fri 09/11/2007	Sat 10/11/2007	DI 99	1	1 /0/0	1	

© Hotels Online B.V., 2001-2007 | Logoff | 24:00 | Culture | EN | Calendar date | 08/11/2007 20:10 | Business date | 08/11/2007 | Error | Enhance | Help

Case Study: Hotel Rooming System

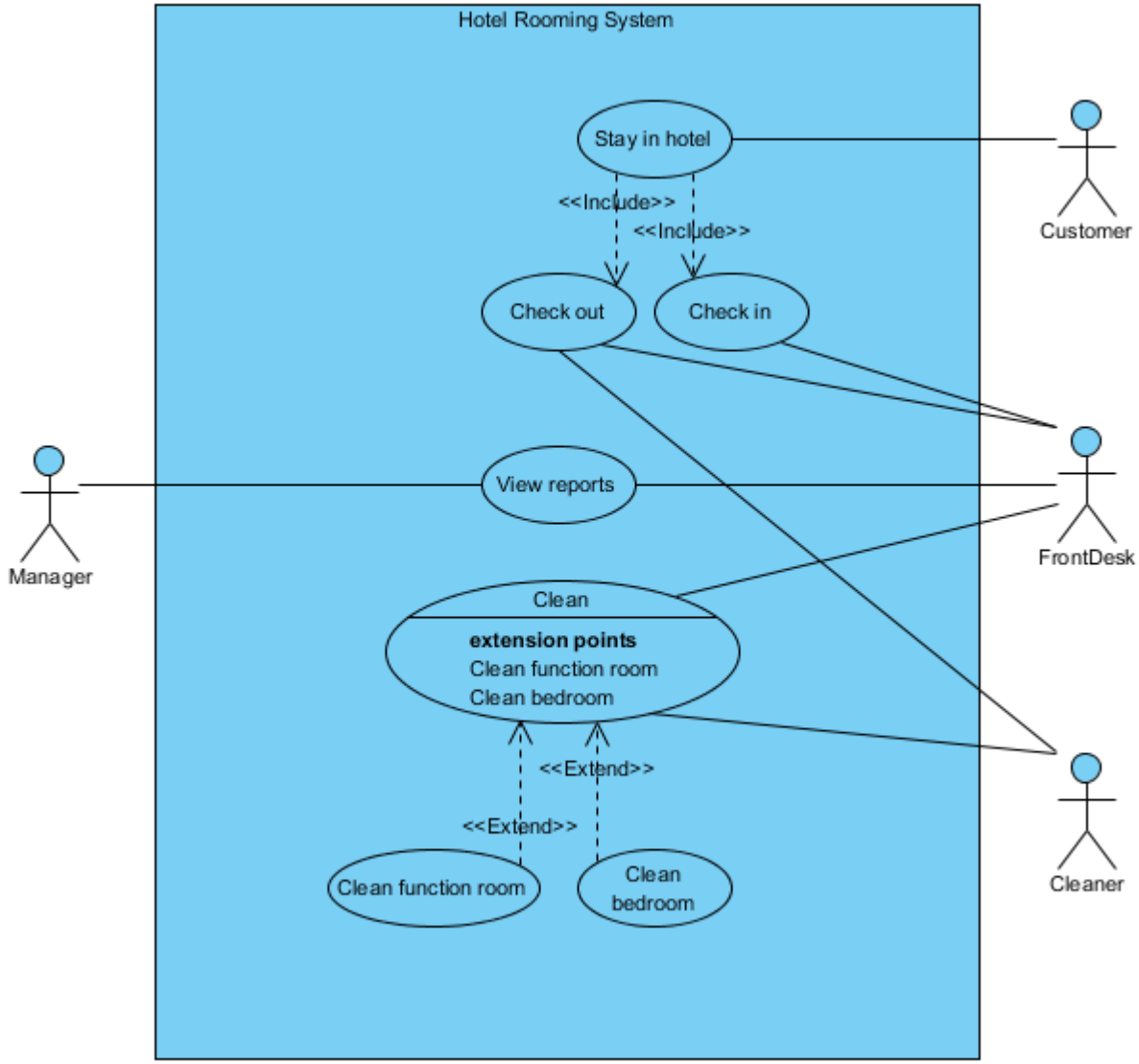
- Requirement Analysis
  - Book customers in on arrival and out on leave
  - Organise room cleaning once room is vacated
  - Rooms: 5 function rooms, 40 bed rooms (10 single, 30 double)
  - Tariff/Night: £40 single room, £55 double room, £1000 function room
  - Equipment: Projector that can be moved between the function rooms
  - System output:
    - Bedrooms occupied (with customer details)
    - Function rooms occupied
    - Which function room contains the projector
    - Income for the day

Use Case Modelling

- Use Case Diagrams
  - Telling a story in a highly structured way
    - Define actors (anyone or anything that interfaces with your system)
      - Identifying actors (can be roles or systems):
        - » Who uses the system? Who gets information from the system?
        - » Who starts/stops the system? Who installs/maintains the system?
        - » What other systems use the system?
    - Define Use Cases (procedures by which an actor might use a system)
      - Identify Use Cases:
        - » What functions will the actors want?
        - » What information is stored in the system?
        - » What actors will create, read, edit or delete this?
        - » Does the system need to notify the actor of internal changes?

Case Study: Hotel Rooming System

- Draw the Use Case Diagram for the Hotel Booking System
  - Book customers in on arrival and out on leave
  - Organise room cleaning once room is vacated
  - Rooms: 5 function rooms, 40 bed rooms (10 single, 30 double)
  - Tariff/Night: £40 single room, £55 double room, £1000 function room
  - Equipment: Projector that can be moved between the function rooms
  - System output:
    - Bedrooms occupied (with customer details)
    - Function rooms occupied
    - Which function room contains the projector
    - Income for the day


Hotel Rooming System
Stay in hotel
<<Include>>
<<Include>>
Check out
Check in
View reports
Clean
extension points
Clean function room
Clean bedroom
<<Extend>>
<<Extend>>
Clean function room
Clean bedroom
Customer
FrontDesk
Manager
Cleaner

Use Case Modelling

- Documenting Use Cases
  - Base case (happy day scenario)
    - Must have pre and post conditions
    - May be formatted to suit your style/need
  - Alternative paths
    - Every other possible way the system can be (ab)used
    - Includes perfectly normal alternative use but also errors and failures
    - How to find them?
      - Going line by line through the Base Path
      - By exception category:
        - » Actor exits application
        - » Actor cancels operation
        - » Actor provides incomplete/bad data
        - » System crashes
        - » ...

Use Case Modelling

- Use Case Documentation Example
  - Pre-condition(s): User has app on smart phone with network
  - Base path:
    1. The Use Case begins when the user opens the app
    2. The map of the user's nearest bus stops is loaded
    3. The user selects a bus stop and bus line
    4. The alarm is set
    5. Time elapses, alarm is modified when busses are off-schedule
    6. Alarm goes off
    7. User turns off alarm and the Use Case ends
  - Post-condition: The bus has left the selected bus stop
  - Alternative path(s): If at any time during step 5 the user selects a different bus line the current alarm is disabled and the basic path continues from step 4

Case Study: Hotel Rooming System

- Use Case Documentation
  - Pre-Condition
  - Base Path
  - Post-Condition
  - Alternative Path

Structural Modelling

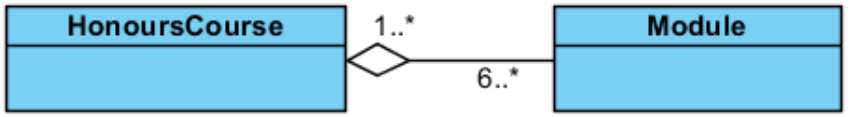
- Class Diagrams
  - Show a set of classes, interfaces and collaborations, and their relationships
  - Addresses static design view of a system
  - Classes
    - Blueprints (templates) for objects
    - Contain data/information and perform operations

Structural Modelling

- Class Diagrams
  - Relationships between classes
    - Association (most general type of relationship) [e.g. passenger - plane]
      - Weak coupling; classes remain somewhat independent of each other
    - Aggregation ("is part of" relationship) [e.g. module - honours course]
      - Used when one object logically or physically contains another; the container is called "aggregate"; components of aggregate can be shared with others
    - Composition ("has a" relationship) [e.g. car - engine]
      - Creates more complex objects by assembling simpler ones; If the whole object is copied or deleted its parts are copied or deleted together with it; the owner is explicitly responsible for creation and deletion of the parts

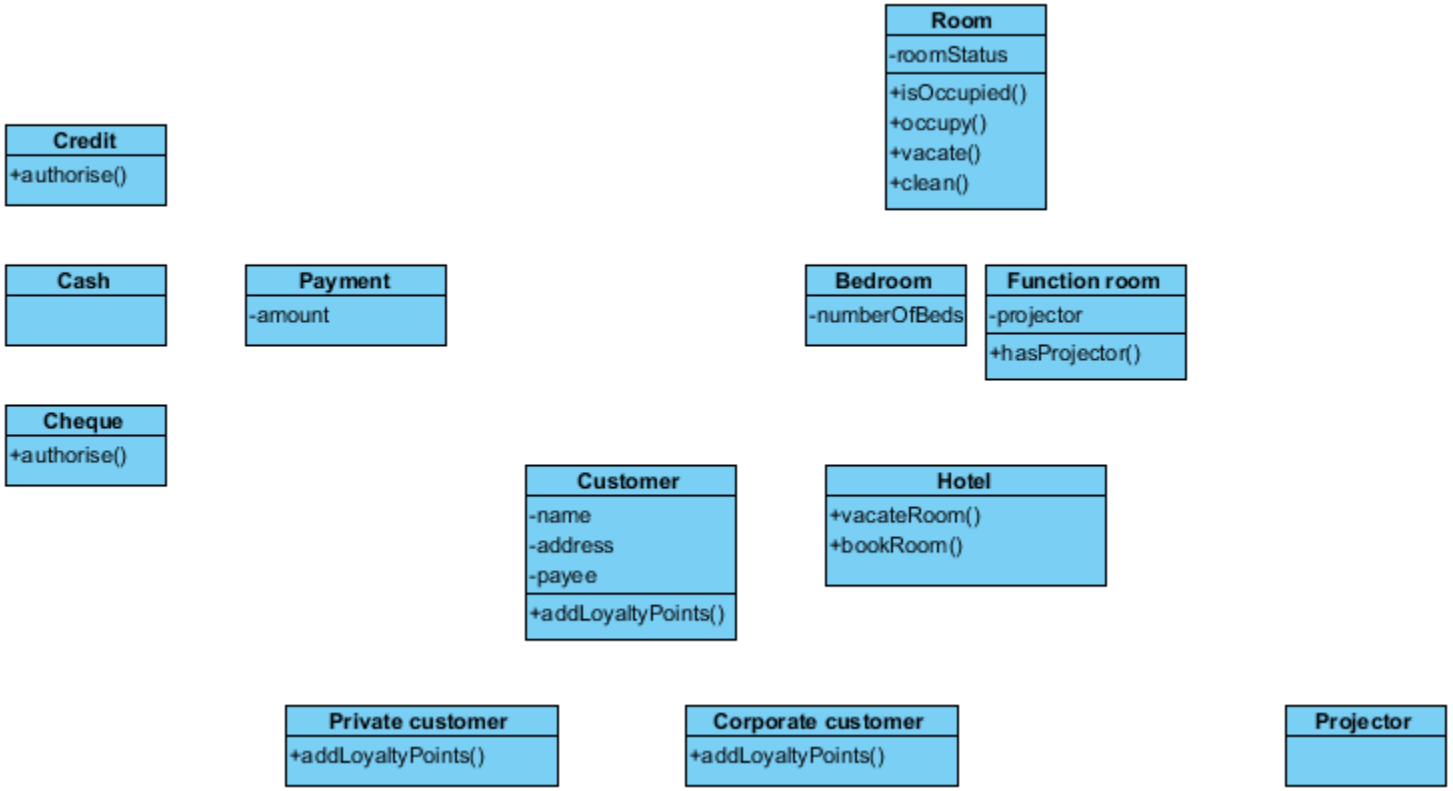
Structural Modelling

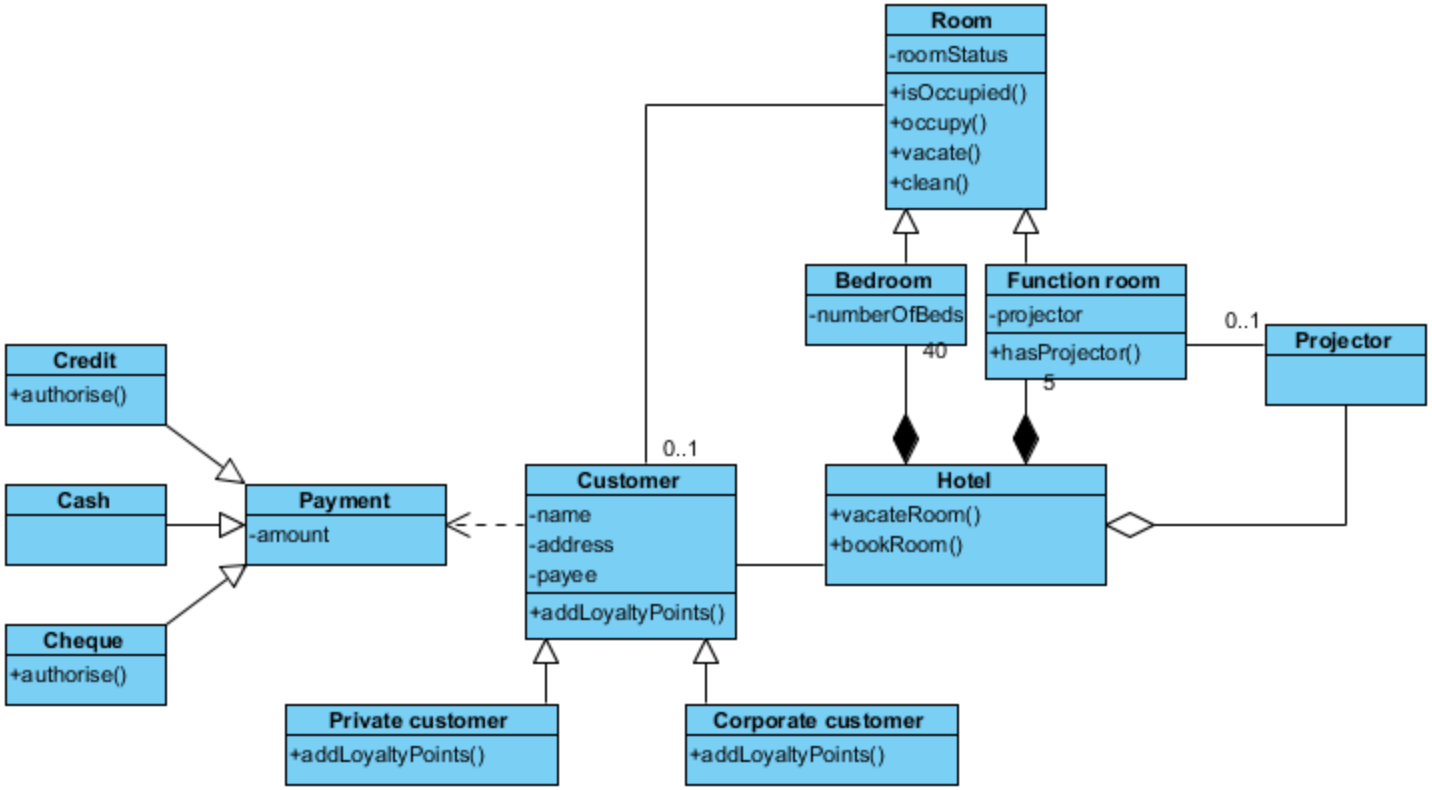
- Class Diagrams
  - Relationships between classes
    - Generalisation ("is a" relationship) [e.g. dog - mammal]
      - Implemented by inheritance
    - Dependency [e.g. client - server]
      - Dependency is semantic connection between dependent and independent model elements
  - Multiplicity
    - Multiplicity allows to specify cardinality (allowed number of instances) of described elements



Case Study: Hotel Rooming System

- Class Diagram (with one abstract class)




Room
-roomStatus
+isOccupied()
+occupy()
+vacate()
+clean()
Bedroom
-numberOfBeds
Function room
-projector
+hasProjector()
0..1
Projector
40
5
Credit
+authorise()
Cash
Cheque
+authorise()
Payment
-amount
0..1
Customer
-name
-address
-payee
+addLoyaltyPoints()
Hotel
+vacateRoom()
+bookRoom()
Private customer
+addLoyaltyPoints()
Corporate customer
+addLoyaltyPoints()

Behavioural Modelling

- Sequence Diagrams
  - Temporal representation of objects (not classes) and their interactions
  - Shows potential interactions consisting of a set of objects and the messages sent and received by those objects
  - Address the dynamic behaviour of a system with special emphasis on the chronological ordering of messages and object creation and deletion
  - Objects not classes

Behavioural Modelling

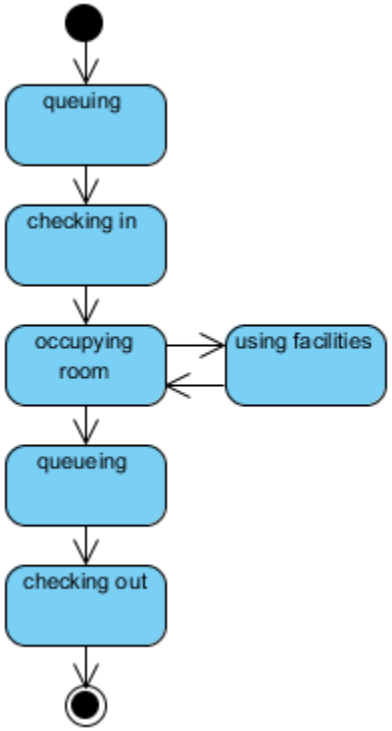
- Activity Diagrams
  - Describe how activities are co-ordinated
  - Support parallel behaviour
  - Use activity diagrams when:
    - Analysing use case
    - Dealing with multithreaded application
    - Trying to understanding workflow across many use cases

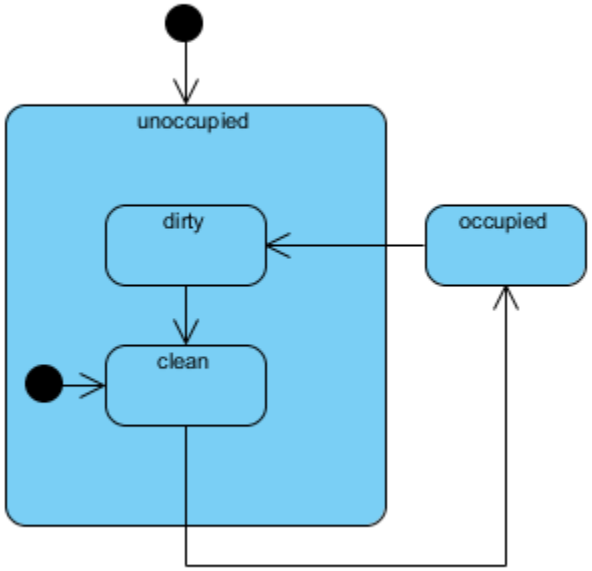
Behavioural Modelling

- State Machine Diagrams
  - Show (1) the states of a single object (2) the events or the messages that cause a transition from one state to another and (3) the action that result from a state change
  - Addresses the dynamic view of a system
  - Not applicable for every class!

Case Study: Hotel Rooming System

- State Machine Diagrams
  - Customer
  - Room
  - Payment


queuing
checking in
occupying room
using facilities
queueing
checking out


unoccupied
dirty
occupied
clean

Software Development Methods

Software Development Methods

- Agile Software Development
  - Group of software development methods based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organising, cross-functional teams

- Promotes and encourages
  - Adaptive planning
  - Evolutionary development and delivery
  - Time-boxed iterative approach
  - Rapid and flexible response to change

Agile Software Development (Agile SD)

- Agile SD is a way of thinking about project management
  - Based on iterative and incremental development
  - Requirements as well as solutions evolve together
  - Collaboration between cross-functional, self-organising teams
  - Teams kept small (5-9 people)

Agile Principles

- The agile manifesto actually has 12 main principles, but these can be condensed to the following five:
  1. Deliver Early and Often to Satisfy Customer
  2. Welcome Changing Requirements
  3. Face to Face Communication is Best
  4. Measure Progress against Working Software
  5. Simplicity is Essential

- Agile SD is the art of maximising the amount of work not done

Agile Principles Applied

- The Agile SD methods are focused on different aspects of the software development life cycle
  - Some focus on the practices
    - e.g. Extreme Programming
  - Some focus on managing the software projects
    - e.g. Scrum
  - Some prove full coverage
    - e.g. Dynamic Systems Development Method

Extreme Programming (XP)

- XP is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements through frequent releases in short development cycles
- Other elements of XP include
  - Programming in pairs
  - Unit testing of all code
  - Avoiding programming of features until they are actually needed
  - Simplicity and clarity in code

Scrum

- Agile SD method for managing software projects

http://www.youtube.com/watch?v=XU0llRltyFM

Scrum [Wikipedia 2014]

- Scrum Artefacts
  - Product backlog
    - Prioritized list of high-level requirements
  - Sprint backlog
    - Prioritized list of tasks to be completed during the sprint
  - Sprint (time-box; iteration)
    - Time period (typically 1–4 weeks) in which development occurs on a set of backlog items that the team has committed to
  - Burn down chart
    - Sprint burn down chart: Daily progress for a Sprint over the sprint's length
    - Release burn down chart: Sprint level progress of completed product backlog items in the Product Backlog

Scrum [Wikipedia 2014]

- People involved
  - Scrum Team
    - Product Owner, Scrum Master and Development Team
  - Product Owner
    - Person responsible for maintaining the Product Backlog by representing the interests of the stakeholders, and ensuring the value of the work the Development Team does
  - Scrum Master
    - Person responsible for the Scrum process, making sure it is used correctly and maximizing its benefits
  - Development Team
    - A cross-functional group of people responsible for delivering potentially shippable increments of Product at the end of every Sprint

Case Study: Hotel Rooming System

- Group Activity:
  - Get together in groups of approx. 5 and assign Scrum roles
  - Create a time planning:
    - Product backlog
    - Release backlog
    - Initialise Burndown chart
  - Hold Day-1 Scrum meeting
  - Present your time planning to the other groups

Test-Driven Development (TDD)

- TDD is a software development process that relies on the repetition of a very short development cycle.
  - The developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards
- TDD is related to the test-first programming concepts of Extreme Programming (XP) but more recently has created more general interest in its own right.
- Programmers also apply the concept of TDD to improving and debugging legacy code developed with older techniques

Case Study: Hotel Rooming System

- Group Activity:
  - How could you develop this system using a TDD approach
  - How would you get started?

Object Oriented Programming

Pointers and memory management

- A pointer is the memory address of a variable

```
#include <iostream>

using namespace std;

int main(int argc, const char* argv[])
{
     int a = 10; int b = 5;
     int* p = NULL;
     p = &a;
     b = *p;
     *p = 5;
     p = &b;
     *p = 10;
     cout << "A = "<< a << ", B is " << b << " p points to value " << *p
          << endl;
}
```

- Keyword "new" creates memory on the heap for an object and returns pointer to that object:

```
Ship* enterprise = new Ship();
```

Inheritance and Polymorphism example

- Fill in the missing code that should be at the dots.
- Note: you may need to fill in multiple lines at some points.

```
class Food
{
private:
    int calories;

public:
    ...

    int getCalories();
    ... boil() ...
    ... bake() ...
    ... fry() ...
}

class Fruit: public Food
{
private:
    list<char> vitamins;

public:
    ...

    ... peel() ...
};

class Banana: public Fruit
{
public:
    ...
}
```

Inheritance and Polymorphism example

```
class Food
{
private:
    int calories;

public:
    Food();
    Food(int c);

    int getCalories();
    virtual void boil() = 0;
    virtual void bake() = 0;
    virtual void fry() = 0;
}
```

```
class Fruit: public Food
{
private:
    list<char> vitamins;

public:
    Fruit();
    Fruilt(list<char> L);

    virtual void peel() =0;
};
```

```
class Banana: public Fruit
{
public:
    Banana();
    Banana(int c);

    void boil();
    void bake();
    void fry();
    void peel();
    List getVitamins();
}
```

Constructor example

```
Food::Food(int c): calories(c) {};

Fruit::Fruit(List v, int c): Food(c), vitamins(v) {};

Banana::Banana(List v, int c): Fruit(v, c) {};
```

Polymorphism use

- Recipe is a composition of ingredients, among other things

```
class Recipe
{
    list<Food*> ingredients;

public:
    Recipe();
    Recipel(list<Food>);

    void prepareFood();
};
```

Write two member function of class Recipe:

1. caloriesInRecipe() counts the total amount of calories contained in the ingredients.
2. vitaminsInRecipe() counts the total amount of vitamins contained in the ingredients

You may assume you have access to additional getter/setter functions.
Assume only Fruit contains vitamins

Polymorphism use answer

```
int Recipe::caloriesInRecipe()
{
    int total = 0;
    list<Food>::iterator i;
    for(i=ingredients.begin(); i != ingredients.end(); ++i)
        total += i->getCalories();
    return total;
}
```

```
list Recipe::vitaminsInRecipe()
{
    list<char> totalVitamins = list<char>(0);
    int total = 0;
    list<Food>::iterator i;
    for(i=ingredients.begin(); i != ingredients.end(); ++i) {
        Fruit F = dynamic_cast<Fruit>(i*);
        if (F) {
            list<char> V = i->getVitamins();
            list<char>::iterator j;
            for (j = V.begin(); j != V.end(); ++j)
                totalVitamins.push_back += j->getCalories();
        }
    }
    return totalVitamins;
}
```

Questions / Comments