Week 10 - Revision

1. Describe three differences between traditional waterfall methodologies and agile software development

Answer:

- A waterfall model follows a linear sequential model consisting of four phases namely, requirements analysis, design, implementation and testing where each phase must be completed prior to the start of the next phase, whereas an agile model flips the linear axis sideways, builds software in iterations where each iteration implements all the four phases on a set of features
- A waterfall model is rigid and not open to changes in requirements whereas an agile software model is open and adaptable to changing requirements
- Customer involvement in a waterfall model is typically at the start and end of the software life-cycle, while agile methods are characterised by continuous involvement throughout the life-cycle, prioritizing work-items and providing feedback on each iteration deliverable

2. In object oriented design, explain the meaning of encapsulation and state its benefits

Answer:

Encapsulation is hiding an object's internal state so that the state (internal instance values) can only be accessed or modified through the object's methods. Benefits of encapsulation:

- ensures that an object's state remains in a consistent state
- increases reusability of the object's class
- by hiding the implementation of the class, a change to the class does not cause a rippling effect on the application

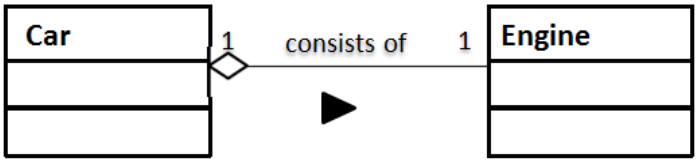
3. In the context of OO design, with the help of an example, explain what an association relationship is and the types of this relationship. Provide a UML representation of this example as well.

Answer:

When two classes share a "has-a" relationship such that one class "contains" another class, this relationship is known as an ***association*** relationship. This relationship can be further refined as ***aggregation*** and ***composition***.

If the contained item (part) is an integral part of the container, then the relationship is said to be a composition relationship. e.g., pages in a book, line-items in an order

If the contained item can exist on its own, then the relationship is said to be an aggregation relationship e.g., a lecturer in a university or a student at a university or an engine in a car



4. Briefly describe the importance of software architecture

Answer:

Software architecture provides the "big picture" or macroscopic organization of the system to be built where we partition the system into logical sub-systems or parts. This provides a high-level view of the system to be built in terms of these parts and how they relate to each other. Software architecture tends to focus on the non-functional requirements ("cross-cutting concerns") and decomposition of functional requirements, while design is more low-level and focuses on the implementation of these functional requirements. Some of the key benefits of defining a software architecture include:

- Partitioning a complex system into sub-systems helps to tackle complexity by "divide-and-conquer"
- Helps to focus on creative parts, see if some parts already exists and avoids "reinventing the wheel"
- Supports flexibility and future evolution by decoupling unrelated parts, so each can evolve separately ("separation of concerns")
- Pre-determine key non-functional requirements (scalability, reliability, performance, usability etc..)
- Promotes understanding and communication among stakeholders, end-users, architects and developers

5. Explain what is the meaning of an architectural style and how is it typically described?

Answer:

Architectural style is a pattern of structural organisation, which defines how the system must be decomposed into its parts and how these parts relate to one another. Basically, an architectural style is defined by:

1. **Components:** A collection of **computational units** or elements that "do the work" (e.g., classes, databases, tools, processes etc.)
2. **Connectors**: Enable communication between different components of the system (e.g., function call, remote procedure call, event broadcast etc.,) and uses a specific **protocol**
3. **Constraints: Defines how the components can be combined to form the system**
    - define where data may flow in and out of the components/connectors
    - topological constraints that define the arrangement of the components and connectors

e.g., **Client-Server Architecture** is a basic architectural style for distributed computing which has

- Two distinct component types:
    - A server that provides specific services e.g., database or file server
    - A client component that requests these services
    - Client and Server could be on same machine or different machines
- **Connector** is based on a **request-response** model

Examples of client-server architecture: File Server, Database server, Email Server

6. Name the architectural style used in each of the following applications

a. Real-time updates on stock prices, weather updates, sporting results
b. Architecture of Spotify in 2010 - Only 8% of music playback comes from Spotify's servers. The rest comes from a distributed network of nodes (Spotify clients).
c. Compilers (source code ->lexical analysis -> syntax analysis -> optimiser -> code generator -> machine code
d. Twitter allows users to follow any other Twitter user (with a publicly available profile) without requiring the other user to follow them back. e.g., user A follows other Twitter users B and C

e. Traditionally, the updates for any Windows device were delivered directly from Microsoft's *Windows Update* servers. While this is the most secure way of getting un-tampered files, it's not the fastest delivery method that you can use. Windows 10 computers and devices can connect to each other and get updates not only from Microsoft's dedicated servers, but also from other Windows 10 devices that have already downloaded parts of the updates

Answer:

a. Publish Subscribe
b. Peer-to-peer
c. Pipe and Filter
d. Publish-Subscribe
e. Peer-to-peer

7.

(a) **Refactoring is a key practice to help high quality in XP. Describe what is refactoring**.

Refactoring is a technique practised in software development for improving the design of an existing code base. It is a process of applying a series of transformations to change the internal structure of software to make the code simpler, easier to understand and avoid code smells without affecting the functionality (i.e. the external observable behaviour of the software)

(b) Explain the role of refactoring play in supporting this principle in XP

XP and other Agile Software development practices place emphasis on continuous customer involvement through the software development cycle and delivering value to customers by being able to quickly respond to changes in requirements. If the client requires a change in the system requirements, the existing code base must be able to accommodate these changes with ease. Refactoring thus becomes particularly necessary for agile software development because the main goal of refactoring is to make the initial code simpler. Constant refactoring aims to build code that is simple, easy to understand and flexible and this enables the team to quickly adapt existing code to new and changing requirements.

One of the core XP principles is **high quality** and high quality of software allows us to develop features faster and cheaper, and constant refactoring helps achieve this high quality. Hence, one of the key practices emphasised in XP is continuous refactoring, a practices where refactoring is integrated into the daily routine of the team.

Examine the code below to answer sections (c) and (d).

(c) Identify two design issues in the code below.
(d) Apply the necessary refactoring and SOLID design principles to enhance the maintainability and reusability of the code

```
class SemesterEnrolment(object):
    def __init__(self):
        self._courses = []

    @property
    def courses(self):
        return self._courses

    def add_course(self, course):
        self.courses.append(course)
```

```
    def generate_local_student_bill(self):
        result = 0

        for course in self.courses:
            result += course.fee * 1

        return result

    def generate_oversea_student_bill(self):
        result = 0

        for course in self.courses:
            result += course.fee * 2

        return result

    def generate_scholarship_student_bill(self):
        result = 0

        for course in self.courses:
            result += course.fee * 0.5

        return result


class Course(object):

    def __init__(self, name, fee):
        self._name = name
        self._fee =fee

    @property
    def fee(self):
        return self._fee

    @property
    def name(self):
        return self._name


s = SemesterEnrolment()
s.add_course(Course('COMP 1531', 5000))
s.add_course(Course('COMP 2041', 5100))
print(s.generate_local_student_bill())
```

Answer:

There is code redundancy in the methods `generate_local_student_bill()`, generate_scholarship_student_bill(), generate_overseas_student_bill(). Code is duplicated in these methods and the only distinction between these methods is the scaling factor applied to the fees for each type of student. A better design would be to refactor the duplicated code into a separate method.

```
from abc import ABC, abstractmethod
class Course(object):

    def __init__(self, name, fee):
        self._name = name
        self._fee =fee
```

```
    @property
    def fee(self):
        return self._fee

    @property
    def name(self):
        return self._name


class BillGenerator():

    def header(self, name, status):
        return 'Fee for current semester for ' + name + '\n' + 'Student status:
' + status + '\n'

    def scale_factor(self):
        return 1

    def generate_bill(self,name,status,courses):
        total_fee = 0
        result = self.header(name,status)
        for course in courses:
            course_fee = course.fee
            result += '\t' + name + '\t' + str(course_fee) + '\n'
            total_fee += course_fee * self.scale_factor()
        result += 'Total fee for semester:' + str(total_fee)
        return result

class OverSeasStudentBillGenerator(BillGenerator):
    def scale_factor(self):
        return 2

class ScholarshipStudentBillGenerator(BillGenerator):
    def scale_factor(self):
        return 0.5

class SemesterEnrolment(object):

    def __init__(self,name,status,billgenerator):
        self._name = name
        self._courses = []
        self._status = status
        self._billgenerator = billgenerator

    @property
    def courses(self):
        return self._courses

    @property
    def name(self):
        return self._name

    def add_course(self, course):
        self.courses.append(course)

    def generate_bill(self):
        result = self._billgenerator.generate_bill(self._name, self._status,
self._courses)
        return result
```

```
s = SemesterEnrolment('chris',"overseas",OverSeasStudentBillGenerator())
s.add_course(Course('COMP 1531', 6200))
s.add_course(Course('COMP 2521', 7200))
print(s.generate_bill())
```

8. Describe the pipe and filter architectural style and provide two benefits of this style

Answer:

Pipe & Filter is an architectural style for stream processing. It consists of:

- A number of **filters** (components) that transform or filter an input data stream before passing the output stream to another filter component via **pipes** (connectors).
- All the filter components work concurrently
  An example of Pipe & Filter is the processing of Unix Shell commands (`ls dir |sort|grep keyword`)

Some of the benefits of this architectural style include:

- Simple composition: any two filters can be recombined, if they agree on data formats
- Support Reuse: decouples different data processing steps so that they can evolve independently of one another and can be used to create new compositions of filters
- Flexible and easily maintained: filters can be easily recombined or replaced by new filters
- Concurrency processing of data streams

9. Analyse the case-study below:

(a) Give an ER design to model this scenario

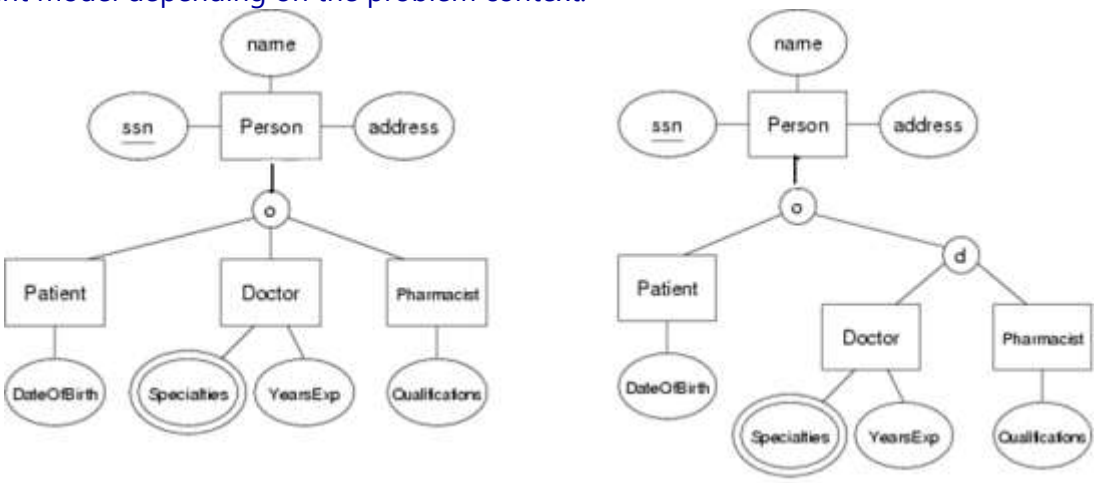
- Patients are identified by an SSN, and their names, addresses and ages must be recorded.
- Doctors are identified by an SSN. For each doctor, the name, specialty and years of experience must be recorded.
- Each pharmacy has a name, address and phone number. A pharmacy must be managed by a pharmacist.
- A pharmacist is identified by an SSN, he/she can only work for one pharmacy. For each pharmacist, the name, qualification must be recorded.
- For each drug, the trade name and formula must be recorded.
- Every patient has a primary physician. Every doctor has at least one patient.
- Each pharmacy sells several drugs, and has a price for each. A drug could be sold at several pharmacies, and the price could vary between pharmacies.
- Doctors prescribe drugs for patients. A doctor could prescribe one or more drugs for several patients, and a patient could obtain prescriptions from several doctors. Each prescription has a date and quantity associated with it.

Note: *To produce an ER design, follow a step-by-step procedure. First underline all nouns and verbs. Then look at the nouns, and distinguish between an entity-set and an attribute of the entity-set. Draw these entity-sets with attributes and underline the key. Next, identify relationships, and connect the relevant entity-sets. Next, think about the cardinality (Students can use either the explicit N:M notation or the arrow-head notation. An arrow-head denotes a single cardinality). Finally, think about the level of participation.*

Answer: The ER diagram is modelled in two steps: (i) First, model the classes of people (ii) Next, model the overall ER diagram

Modelling the classes of people:

There are two possibilities towards modelling this as shown in figures (1a) and (1b). In the first model, a Person who is a Patient can also be a Doctor and a Pharmacist (as indicated by the overlapping symbol "o". In the second model, a Person can be a Patient and a (Doctor or Pharmacist (as indicated by the disjoint symbol "d") i.e., someone who is a Doctor cannot also be a Pharmacist. In modelling, choose the relevant model depending on the problem context.



Figures 1a and 1b

Modelling the overall ER diagram

Model the overall ER diagram, ignoring the attributes of Patient, Doctor and Pharmacist for clarity. Also, the Person entity is not used, as it is not directly involved in any relationship

.

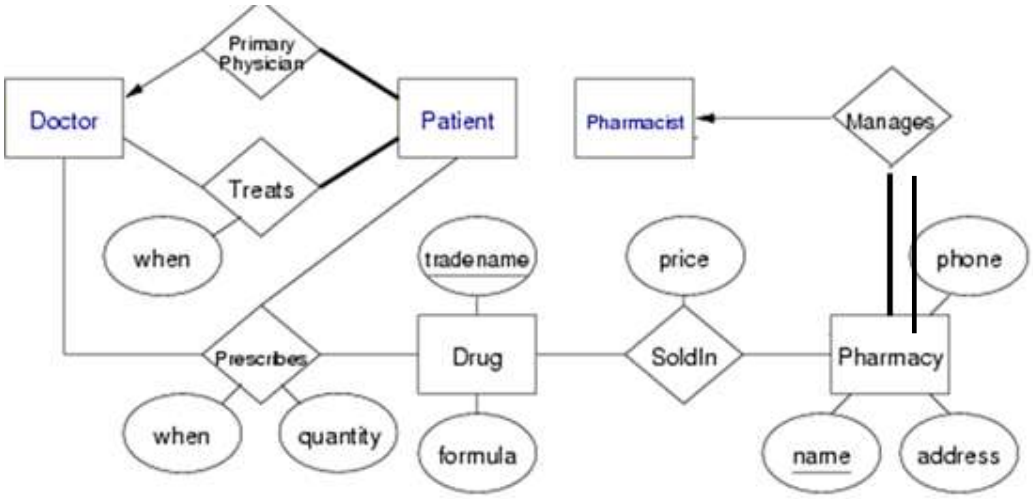


Figure 2

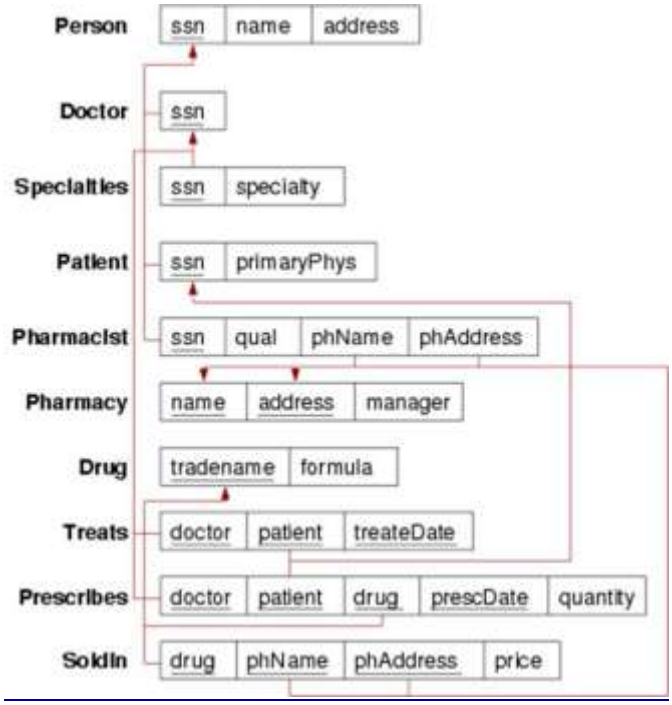
(b) Map the ER design to a relational model

Using the model of the Person class hierarchy as shown in Figure (1a), map the ER design above in Figure (2), to relational form using box schema notation. What aspect of the ER design cannot be modelled by the relational model? *(In mapping the person class hierarchy from the ER design to relational model, use the ER style mapping shown in lectures)*

Answer:

i. Relational Model using the box schema notation is shown below. In this model, the person class hierarchy is modelled using the **ER-style mapping.**

ii. The relational model cannot represent the total participation constraints for patients (i.e. every patient must be treated by at least one doctor.



(c) In the above ER model, you can only model a prescription having a single line-item i.e. each prescription can have only one list one Drug (and date and quantity). How would you change the above ER model so that two or more drugs prescribed by the same doctor for the same patient on the same day can be part of the same prescription?

Answer: The revised model shows two or more drugs prescribed as part of the same prescription by the same doctor for the same patient on the same date.

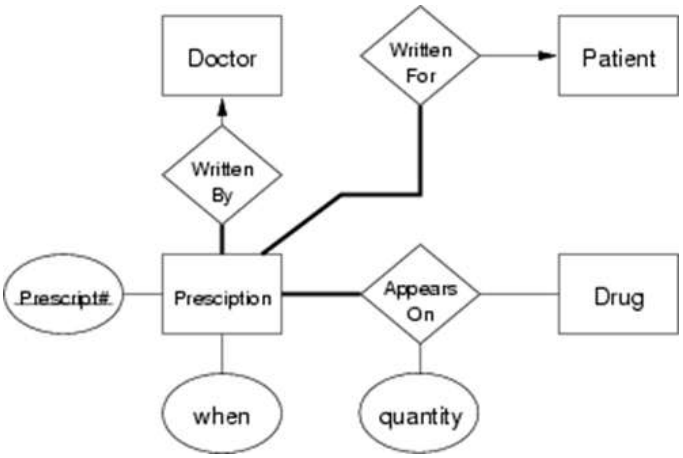
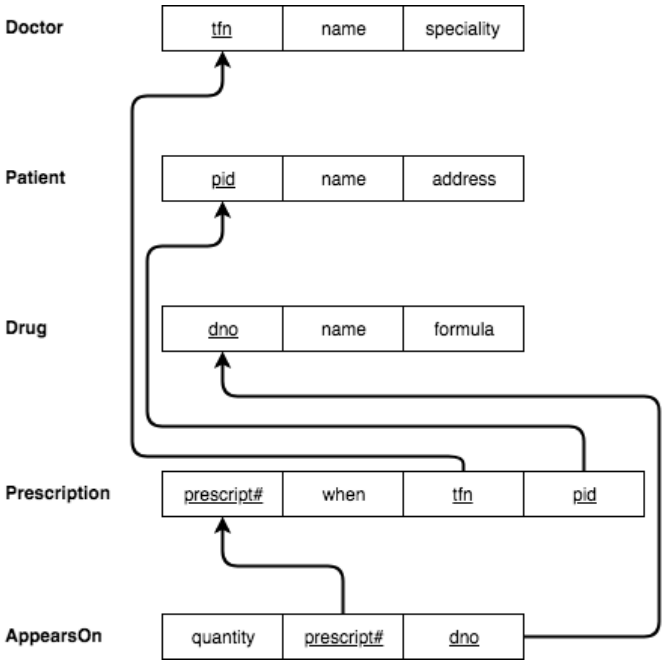


Figure 3

Some of the inherent assumptions in the above model:

- every prescription has some kind of unique identifying number
- every prescription must be written by a particular doctor
- every prescription must be written for a particular patient
- a prescription must list ***at least*** one drug (total participation), and can list several drugs

(d) Now, map the revised ER model in figure (3) to relational form in the box schema notation



Additional Questions to Think About:

10. What are the major features of a waterfall methodology?
11. Give one example each of a software engineering project that would benefit from a waterfall and a agile model and justify
12. What are the drawbacks of adopting an agile methodology for a project?
13. In an XP Project Planning game, what are the key aims of the release plan and iteration plan?
14. Describe briefly one way to track a project velocity?
15. Distinguish with the help of an example a static UML diagram and a dynamic UML diagram
16. Describe the RGB technique of writing a user-story
17. Describe how INVEST is used as a technique to assess how good a user-story is
18. Describe three symptoms of software rot
19. What is refactoring? Explain the role of refactoring in XP?
20. Describe two key characteristics of good software design
21. With the help of an example, briefly describe one way the SRP SOLD principle might be violated