Software Engineering COMP 201

Lecturer: **Sebastian Coope**

Ashton Building, Room G.18

*E-mail: **coopes@liverpool.ac.uk***

COMP 201 web-page:

http://www.csc.liv.ac.uk/~coopes/comp201

Lecture 18 – Introductory Case Study

Introduction to UML

- During this lecture, we shall see how various features of the **Unified Modeling Language** (UML) can help to reduce ambiguities and increase understanding of a proposed system
- We will be studying an introductory case study based on a health clinic system and giving an introduction to:
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - State diagrams

The Problem

- The most difficult part of any design project is understanding the task you are attempting
- Example: you have been contacted to develop a computer system for a university medical clinic.
- The clinic needs the following types of service
  - Staff management
  - Booking appointments
  - Keeping records
- You are asked to build an interactive system which handles all of these aspects online.

Clarifying the Requirements

- Different users will have different, sometimes conflicting, priorities
- Users are not likely to have clear, easily expressed views of what they want
- It is hard to imagine working with a system of which you have only seen a description

Facts about the Requirements

- Doctors, patients, admin staff
- Appointments
- Treatments

Homework

Specify the facts about the requirements that an ideal system would satisfy.

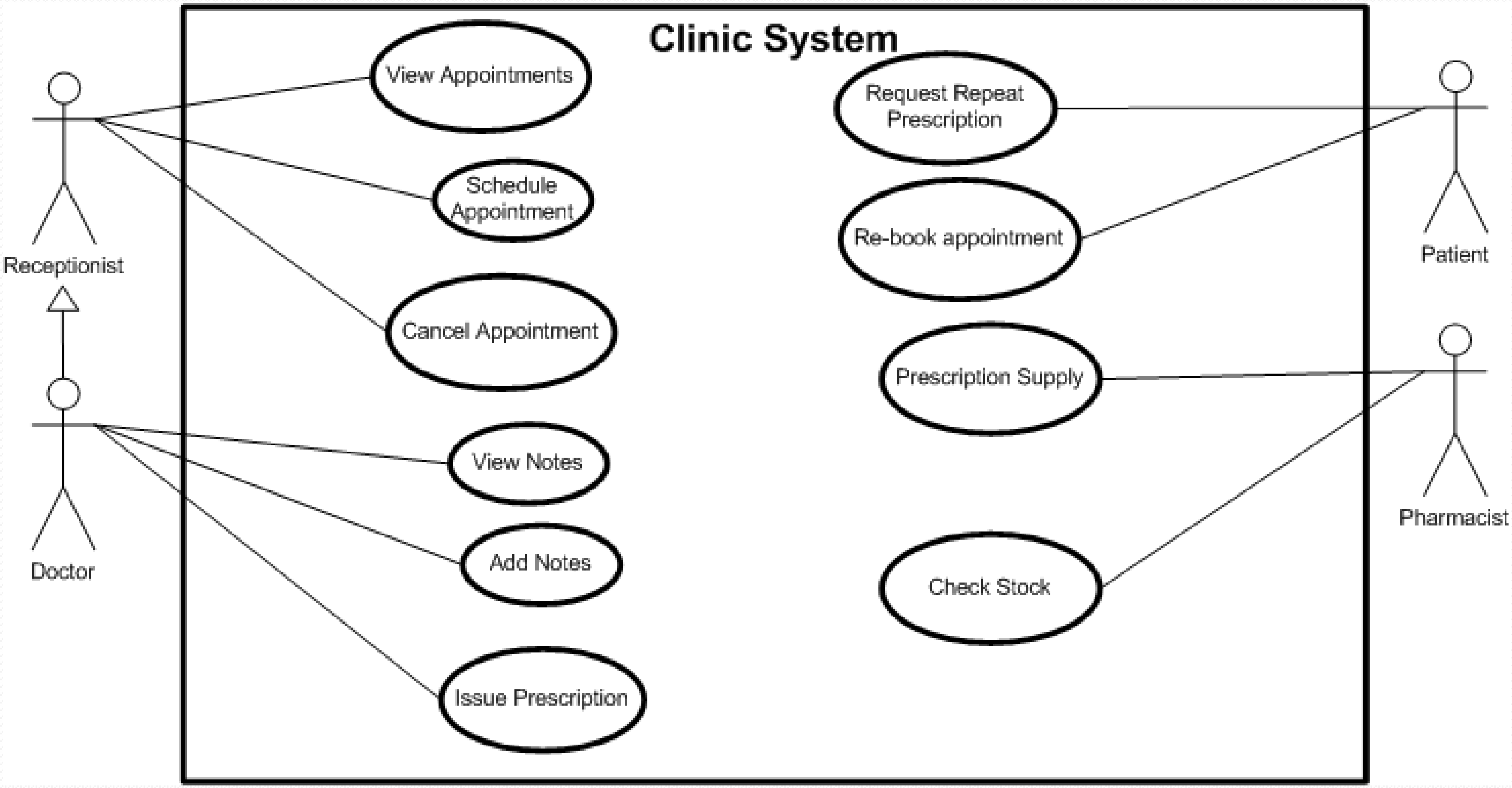
Use Case Model

- If a system is to be seen as having high quality, it must meet the needs of its users.
- So we take a user-oriented approach to systems analysis.
- We identify the users of the system and the tasks they must undertake with the system.
- We also seek information about which tasks are most important, so that we can plan the development accordingly.

What do we Mean by “Users” and “Tasks”?

- UML uses as technical terms “actors” and “use cases”
- An actor is a user of a system in a particular role *(an actor can also be an external system)*
  - For example. Our system will have an actor **Receptionist** representing the person who interacts with the system to book an appointment
- A use case is a task which an actor needs to perform with the help of the system,
  - such as **BookAppointment**

Use Case Diagram for the clinic



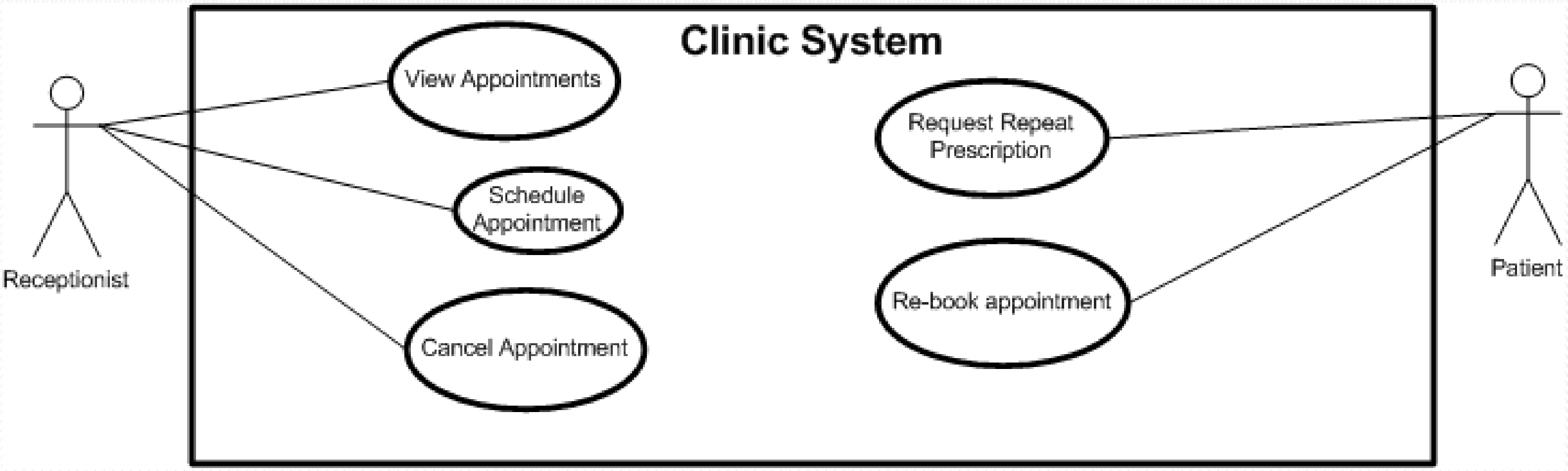
Scope and Iterations

- To limit the risk, it is better to aim to get to the ideal system in several steps or iterations.
  - The first iteration results in the delivery of a system with only the most basic and essential functionality;
  - Later iterations enhance the system
- One of the main purposes of use cases is to help identify suitable dividing lines between interactions:
  - An interaction can deliver enough of the system to allow certain use cases to be carried out, but not others

Limiting Requirements

- It is important **not to invent new requirements** for the system. After examining use-cases, it is often easy to think of new things the system could or should do, but these must first be discussed with the customer.
  - For example, perhaps it would be good to inform the doctor that a drug is out of stock via their online system? But this may not be what is wanted by the Dr, this may cause an overload of information!

Use Case Diagram for the First Iteration



- Let us suppose that after discussing priorities with the customers we decide that the first iteration of the system should provide:
  - View appointments, schedule appointment, cancel appointment, re-book appointment, request repeat prescription

Use Case Advantages

- It may be easier to identify the amount of time required to implement all the required features of the system.
  - Such details can often be optimistically overlooked.
- We can identify which requirements are important to key strategic (or most influential) personnel in the company.
  - By providing this functionality early, we can show the potential value of the software and avoid the project being cancelled.

Use Case Advantages

- We may decide to implement more risky use cases first since we would hopefully still have contingency to tackle problems that arise
  - In other words, early in the development we have more flexibility in terms of time, money, design choices etc.
- Use cases can be used to derive validation checks on the developed system, in order that it provides all required functionality.

Identifying Classes

- In the standard jargon of analysis we often talk about the ***key domain abstractions***.
- Identifying the right classes is one of the main skills of OO development.
- We start the process of identifying the *key domain abstractions* using the following approach, which is known as the ***noun identification technique***.

Identifying a list of candidate classes

Clinic, appointments and treatment system
Before seeing a doctor or nurse the patient needs to make an appointment. The appointment will be made by the receptionist, before making the appointment the patient needs to ask the patient which doctor they wish to see and if the appointment is a standard appointment or urgent appointment. The receptionist will use this information, check the appointment schedule and find a free slot and make the booking. When the patient sees the Dr, the Dr will sometimes issue a prescription. The patient may at some time request a repeat issue of their prescription. Receptionists can also cancel appointments. Each doctor has a maximum of 2000 patients registered to them.

- Take a coherent, concise statement of the requirement of the system
- Underline its *noun and noun phrases*, that is, identify the words and phases that denote things
- This gives a list of candidate classes, which we can then whittle down and modify to get an initial class list for the system

Removing Superfluous Candidates

- In this particular case, we discard:
  - **Clinic**, because it is outside the scope of our system
  - **Urgent appointment** redundant covered by appointment
  - **Time**, because it is a measure, not a thing
  - **Free slot**, because it is vague (we need to clarify it)
  - **System**, because it is part of the meta-language of requirements description, not a part of the domain

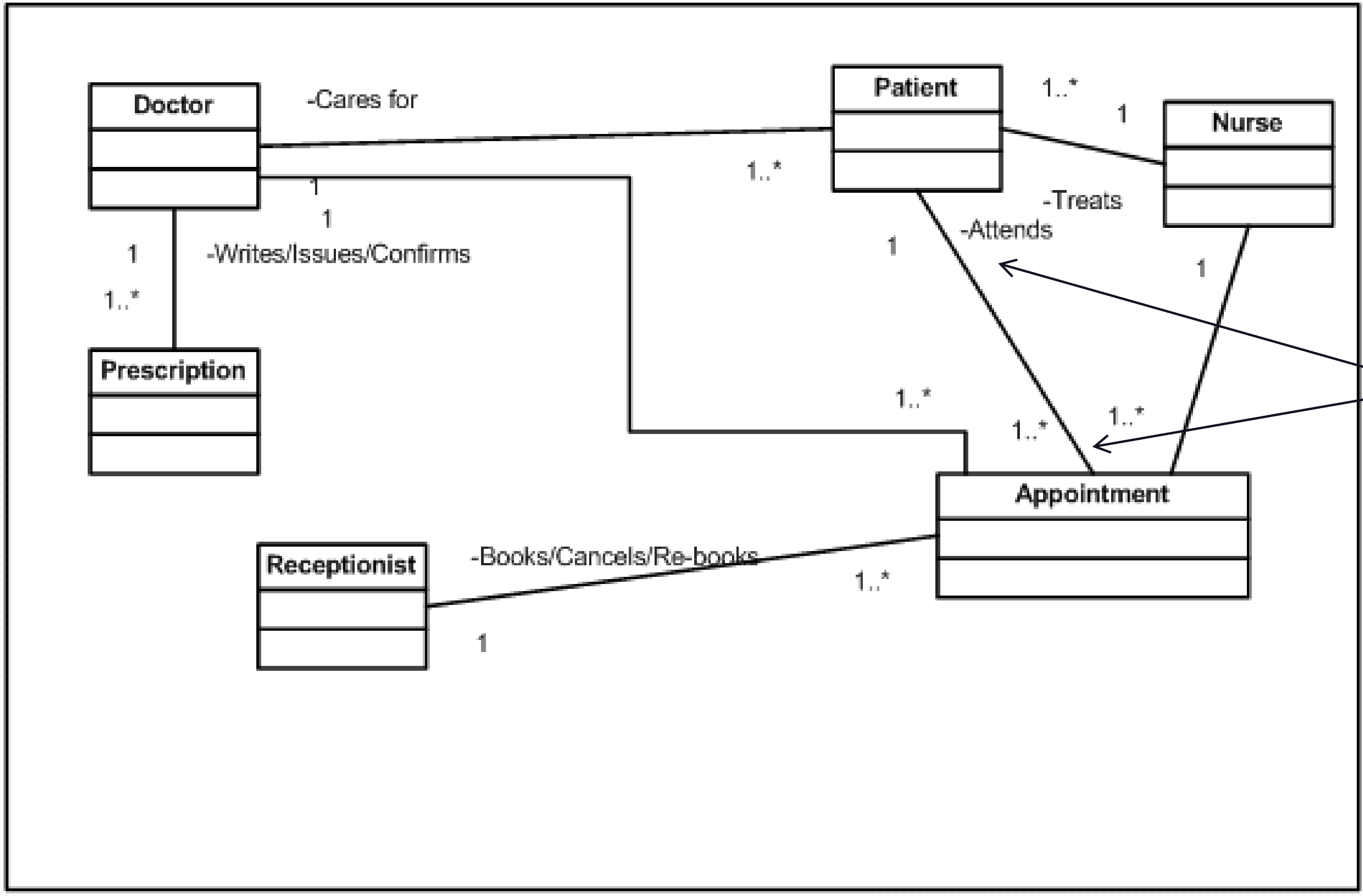
This leaves:

- Doctor
- Nurse
- Receptionist
- Patient
- Appointment
- Prescription

Relations between Classes

- Next we identify and name important real-world relationships or associations between our classes
- We do this for two reasons:
  - To clarify our understanding of the domain, by describing our objects in terms of how they work together;
  - To sanity-check the coupling in our system, i.e. make sure that we are following good principles in *modularising our design*
- We shall now see the relations for the clinic system example..

Initial Class Model of the Health Clinic

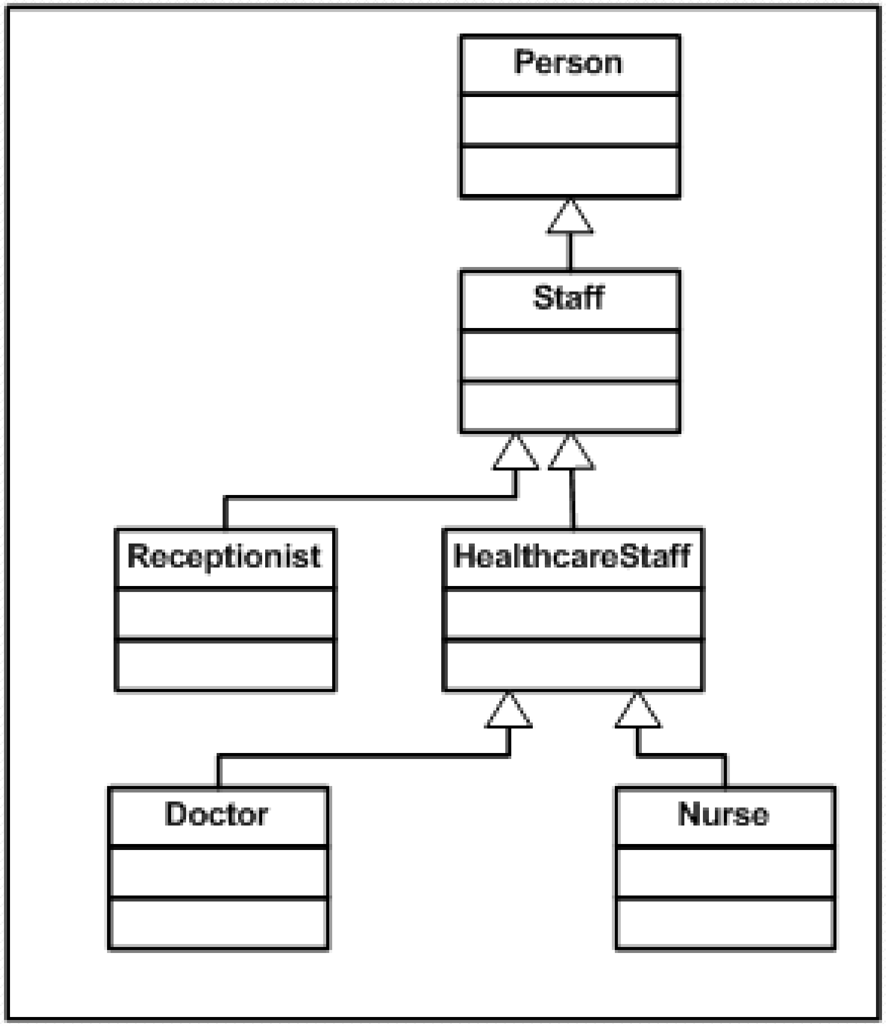


These are ***multiplicities*** which we shall study in detail next lecture..

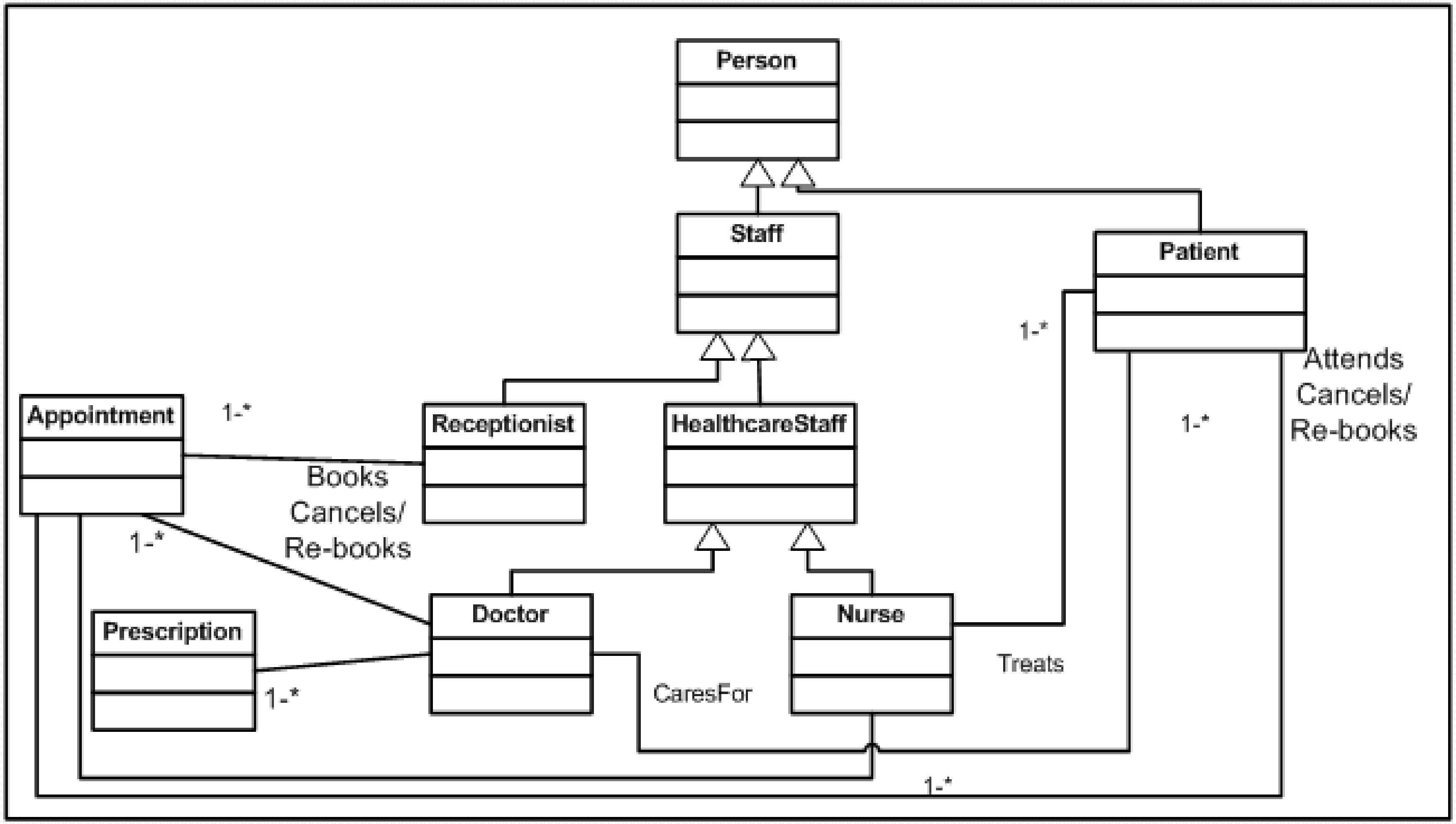
Lets Revise our Class Model

- Finally, we may notice that
  - Doctor shares all the same associations that Nurse does, and that
  - this agrees with our intuition that both nurse and doctor are health care staff.
- Recording this in the class diagram will clarify our understanding of the situation, that there is a generalization relationship between these classes
- Inheritance, Dr and Nurse are both health care staff and receptionist is also staff
- Notice Staff and HealthCareStaff are additional classes to help with inheritance

Revised Health Class Model (hierarchy)



Revised Health Class Model



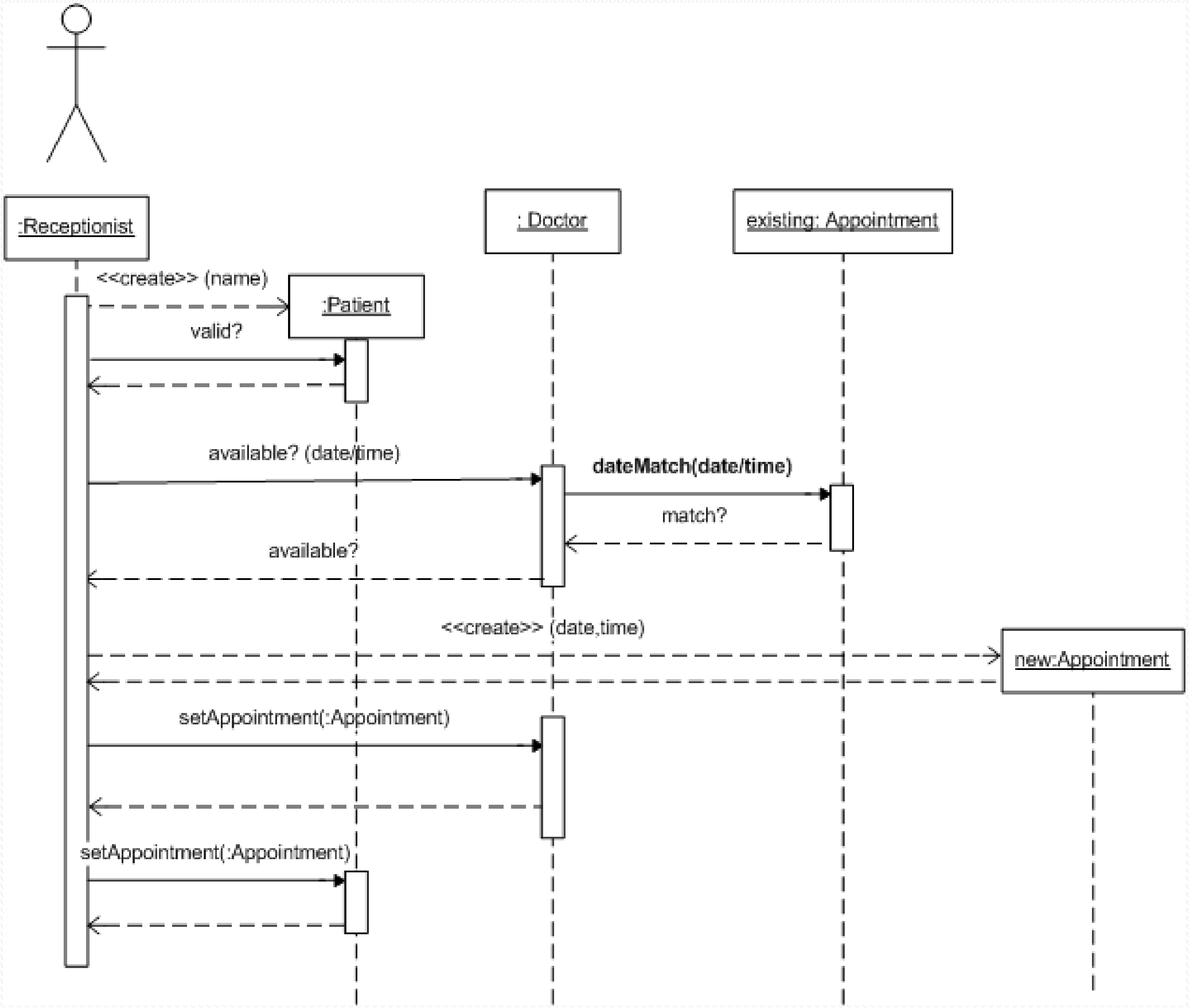
The System in Action

- A class diagram gives a **static view** of the system, but we know nothing about the **dynamic behaviour**
- In UML we can use interaction diagrams to show how messages pass between objects of the system to carry out some task
  - This will also show how the various classes realize the different use cases we identified in the use case diagram

An Example Sequence Diagram

- Consider what happens in the appointment booking scenario when a user wishes to make an appointment
  - The **receptionist must check that the person is a valid patient**
  - Then the doctor object must be checked to see if there are any available appointments
  - If there are any suitable slots available, a new appointment should be created and assigned to the doctor.
- We now see how this is recorded in a sequence diagram

Interaction Shown on a Sequence Diagram



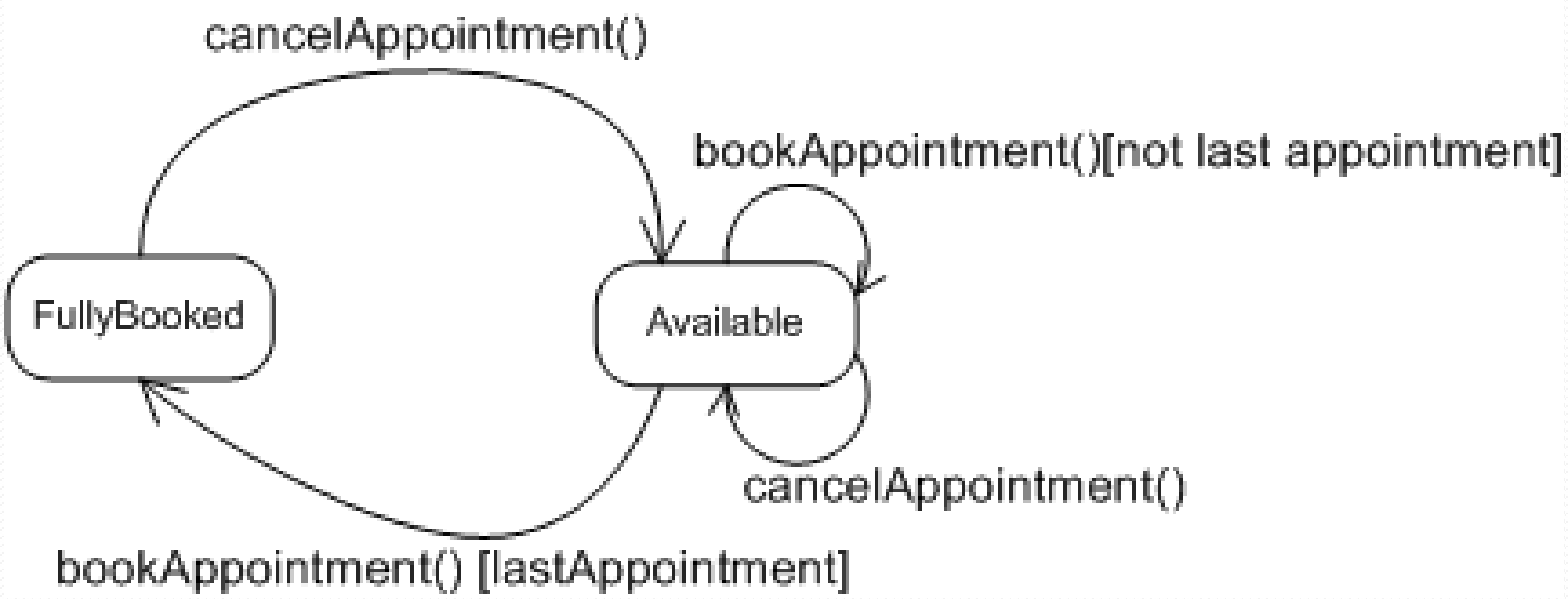
Sequence Diagrams

- We shall see more on **sequence diagrams** later, but note their general structure and that they record which actors and classes are involved in an interaction.
- In this example the interaction is very simple, there is a single execution path and nothing occurs in parallel; in more complex scenarios, sequence diagrams can clarify the working of a system to a greater extent

State Diagrams

- Objects in the system have a **state** which is all the data which it currently encapsulates
- For example, a Doctor object on the health system can be available or fully booked
- Running methods on the object can cause a change in state, i.e., by booking appointments to the doctor object we change its internal state.
- Changes in object states can be modelled by a **state diagram**..

Changes of the System: State Diagrams


cancelAppointment()
bookAppointment()[not last appointment]
FullyBooked
Available
cancelAppointment()
bookAppointment() [lastAppointment]

- State diagram for class Doctor
- We will consider state diagrams again in more detail in a later lecture

Lecture Key Points

- We have seen an *introduction* to the **Unified Modelling Language** (UML)
- We studied an introductory case study based on a health clinic system with an introduction to:
  - Use case diagrams
  - Class diagrams
  - Sequence diagrams
  - State diagrams