M19COM Software Development & Design

Week 8: Object Oriented Design with UML

Dr Faiyaz Doctor

References.

"Applying UML and Patterns" (3rd Edition) by Craig Larman. Prentice-Hall

"Object Oriented Design with the Unified Process", Salzinger, Jackson & Burd

Objective

- Create design class diagrams (DCDs) and interaction diagrams in UML
- Identify the classes, methods,
- Types of relationships shown in a DCD.
- Using an example to illustrate.
- Will cover interaction diagrams at a later date, also suggested for further reading

Class Diagrams

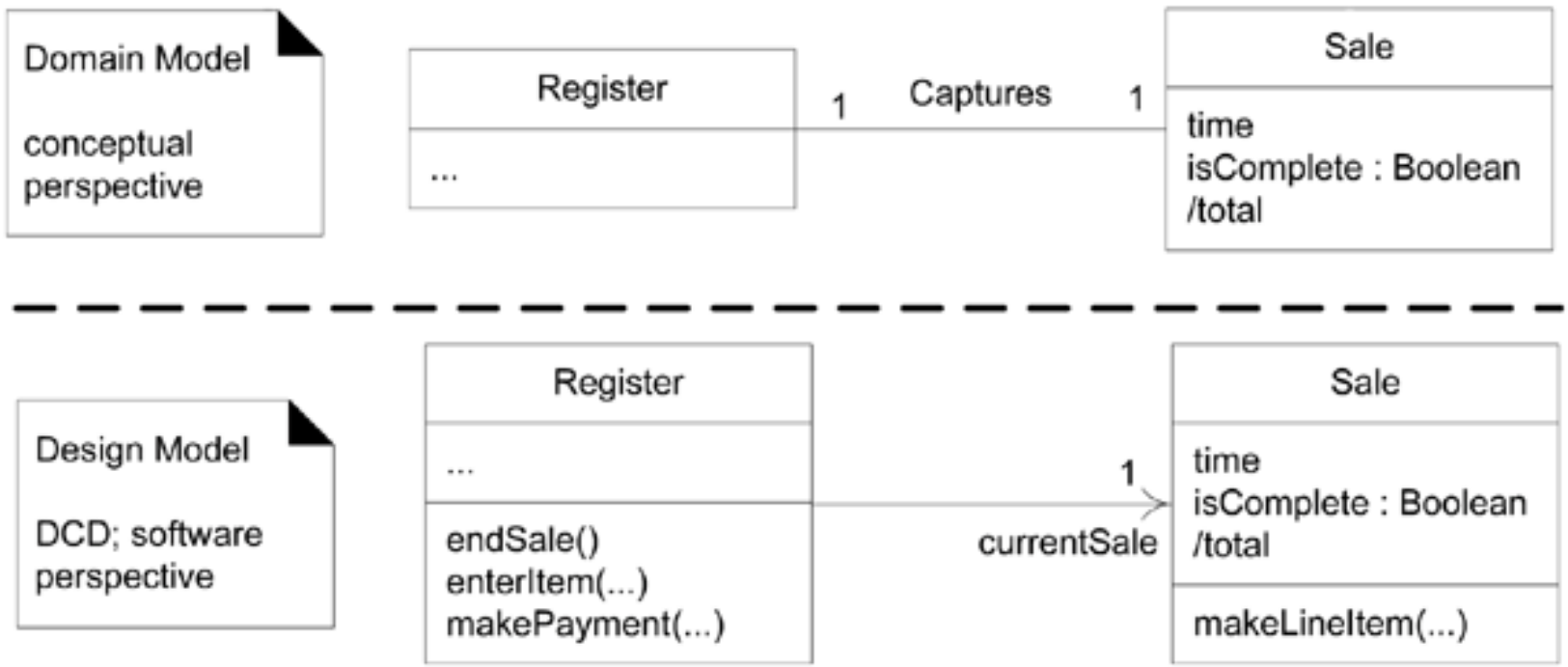
- The UML has notation for showing design details in static structure.
  - **Class diagrams**
- The definition of design class diagrams occurs within the design phase.
  - **The UML does not specifically define design class diagram.**
  - **It is a design view on SW entities, rather than an analytical view on domain concepts.**

Influences on DCD

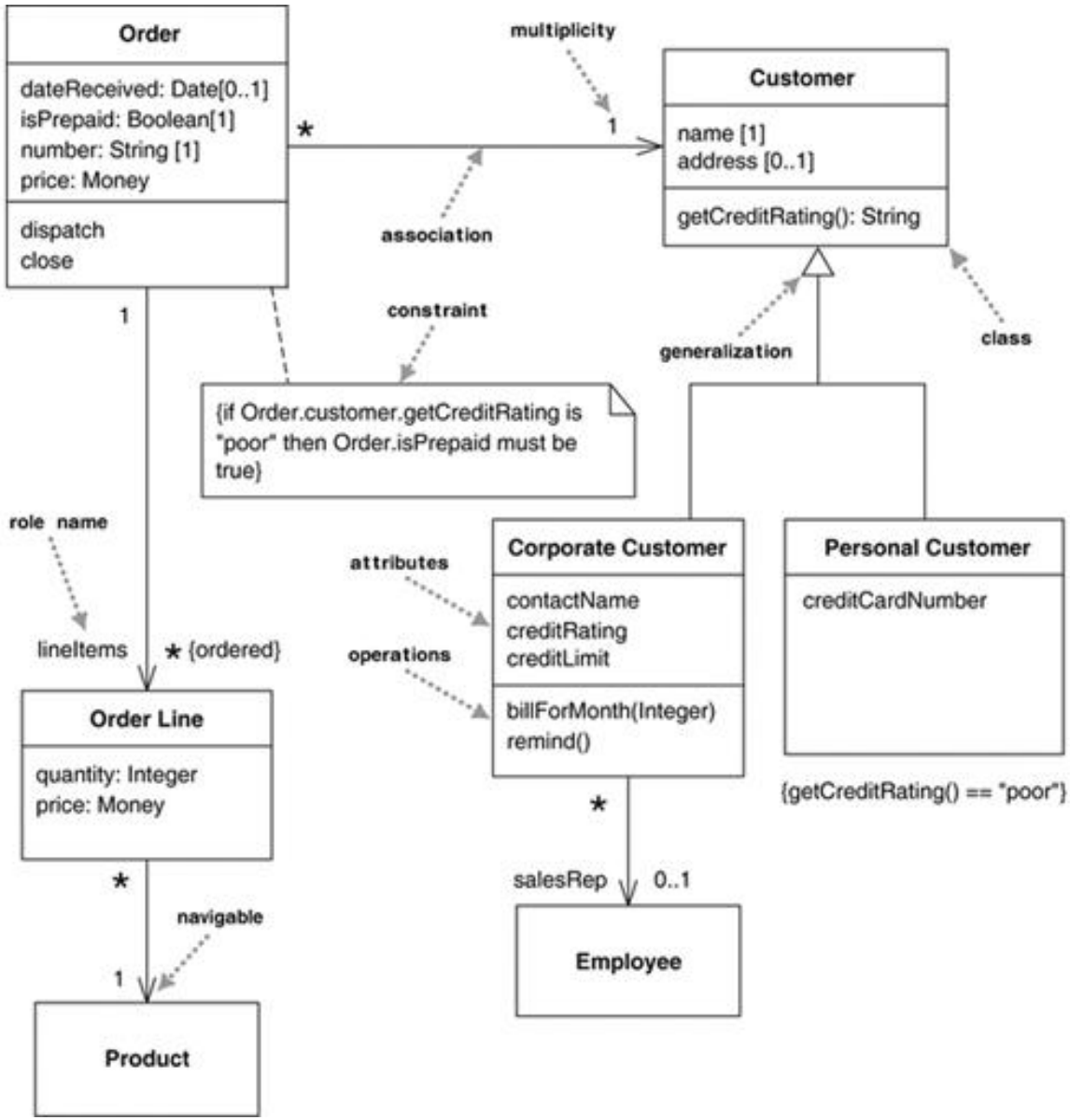
The creation of design class diagrams is dependent upon the prior creation of:

- **Conceptual model**
  - Adds detail to the class definitions.
- **Interaction diagrams**
  - Identifies the **SW classes** that participate in the solution, plus the **methods** of classes.
  - Often created in parallel with DCDs

UML class diagrams in two perspectives



Example of a DCD



- Illustrates specifications for SW classes and interfaces
- Typical information included:
  - Classes, associations, and attributes
  - Interfaces, with their operations and constants
  - Methods
  - Attribute type information
  - Navigability
  - Dependencies

How to make a DCD?

Identify Classes

- Identify all the classes participating in the SW solution by analyzing the interaction diagrams.

Draw Classes

- Draw them in a class diagram.

Add Attributes (From Domain Model)

- Duplicate the attributes from the associated concepts in the domain model.

Add Methods/Operations

- Add method names by analyzing the interaction diagrams.

Elaborate Attributes

- Add type information to the attributes and methods.

CASE STUDY TO ILLUSTRATE THE DESIGN PROCESS

A Dental Appointments System

Actor 1, the receptionist, has 3 use cases

- make appointment
- change appointment
- cancel appointment

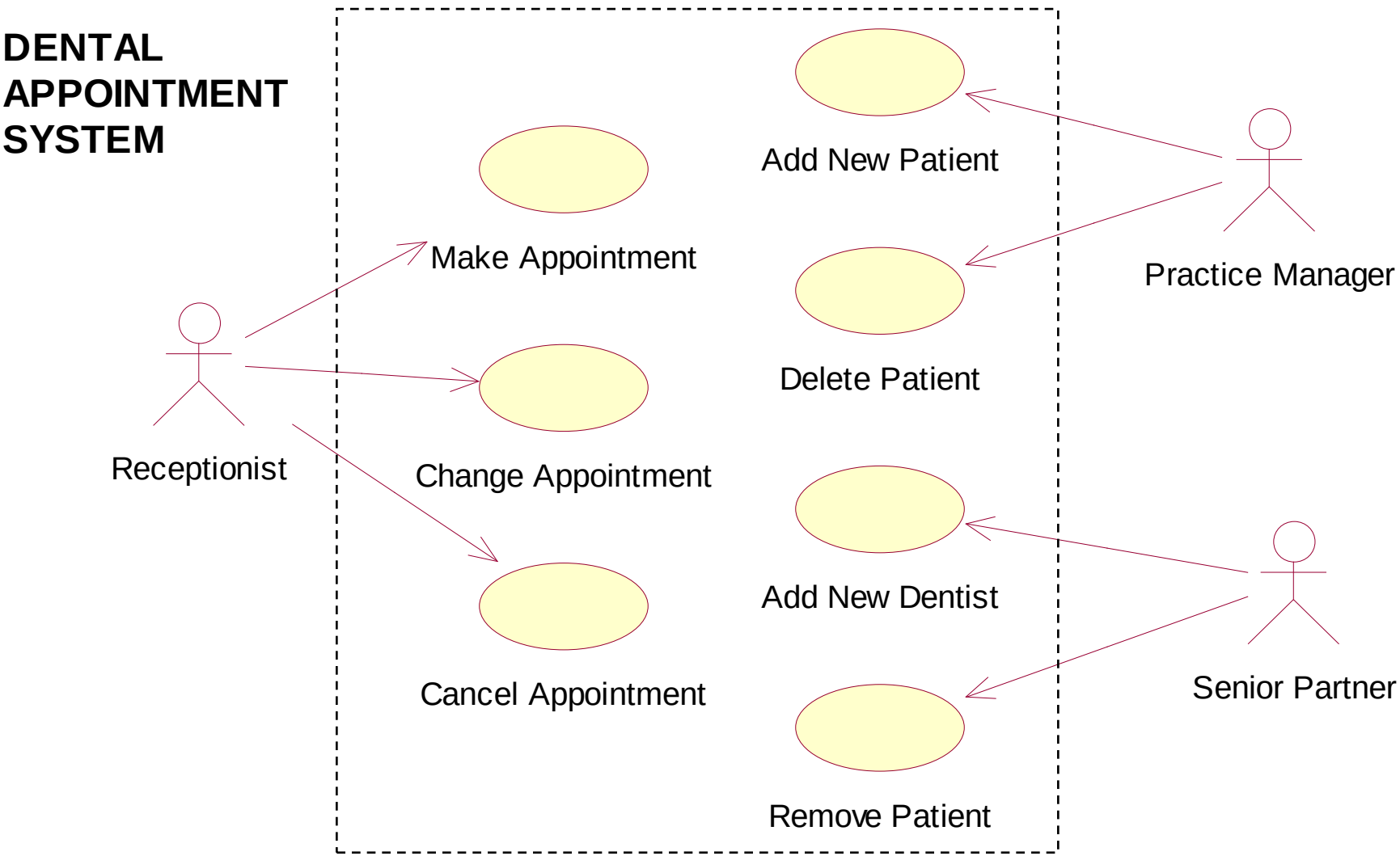
Actor 2, the practice manager, has 2 use cases.

- add new patient
- delete patient

Actor 3, the Senior Partner, has 2 use cases

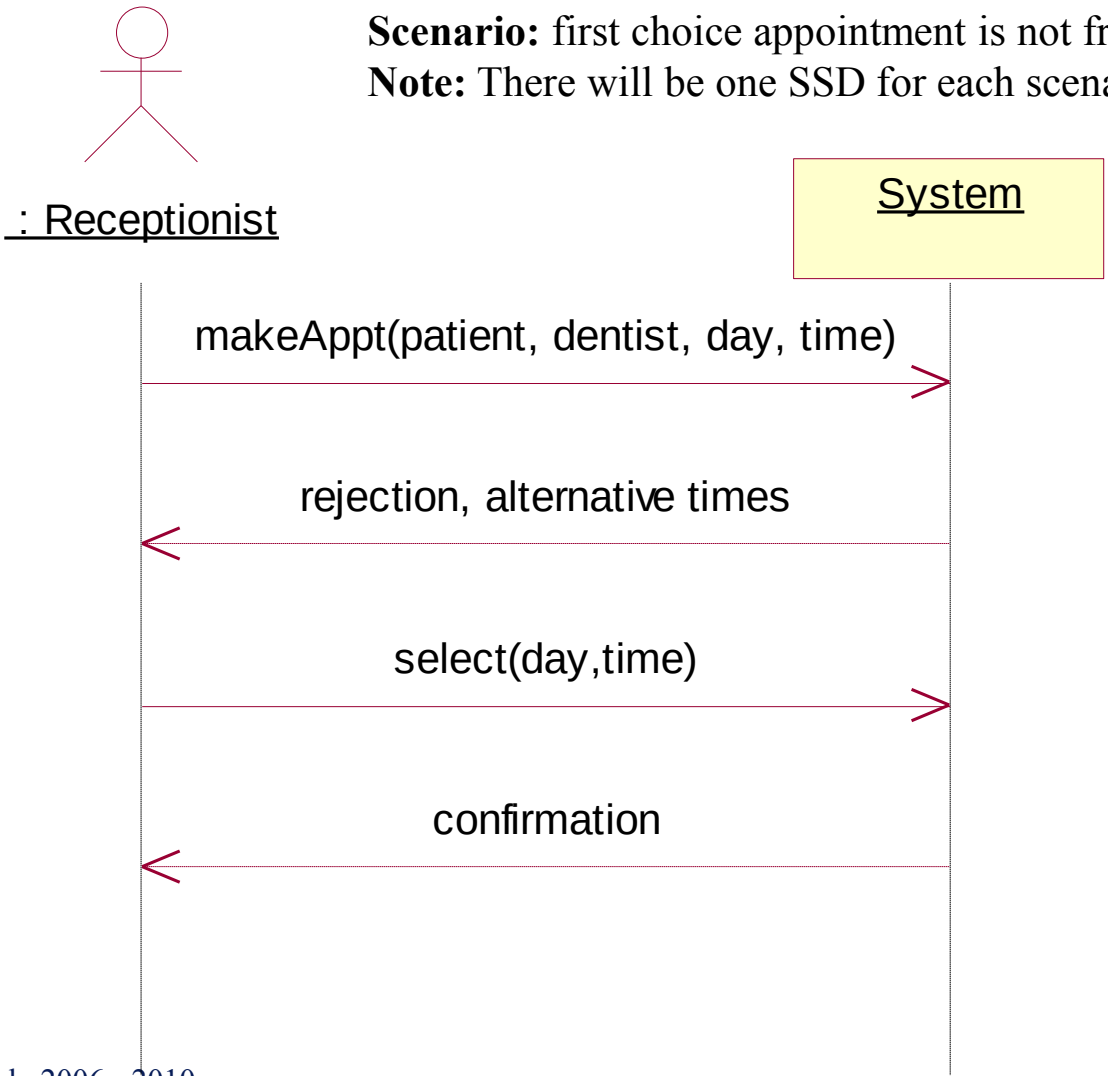
- add new Dentist
- delete Dentist.

Domain Model: Use Case Diagram



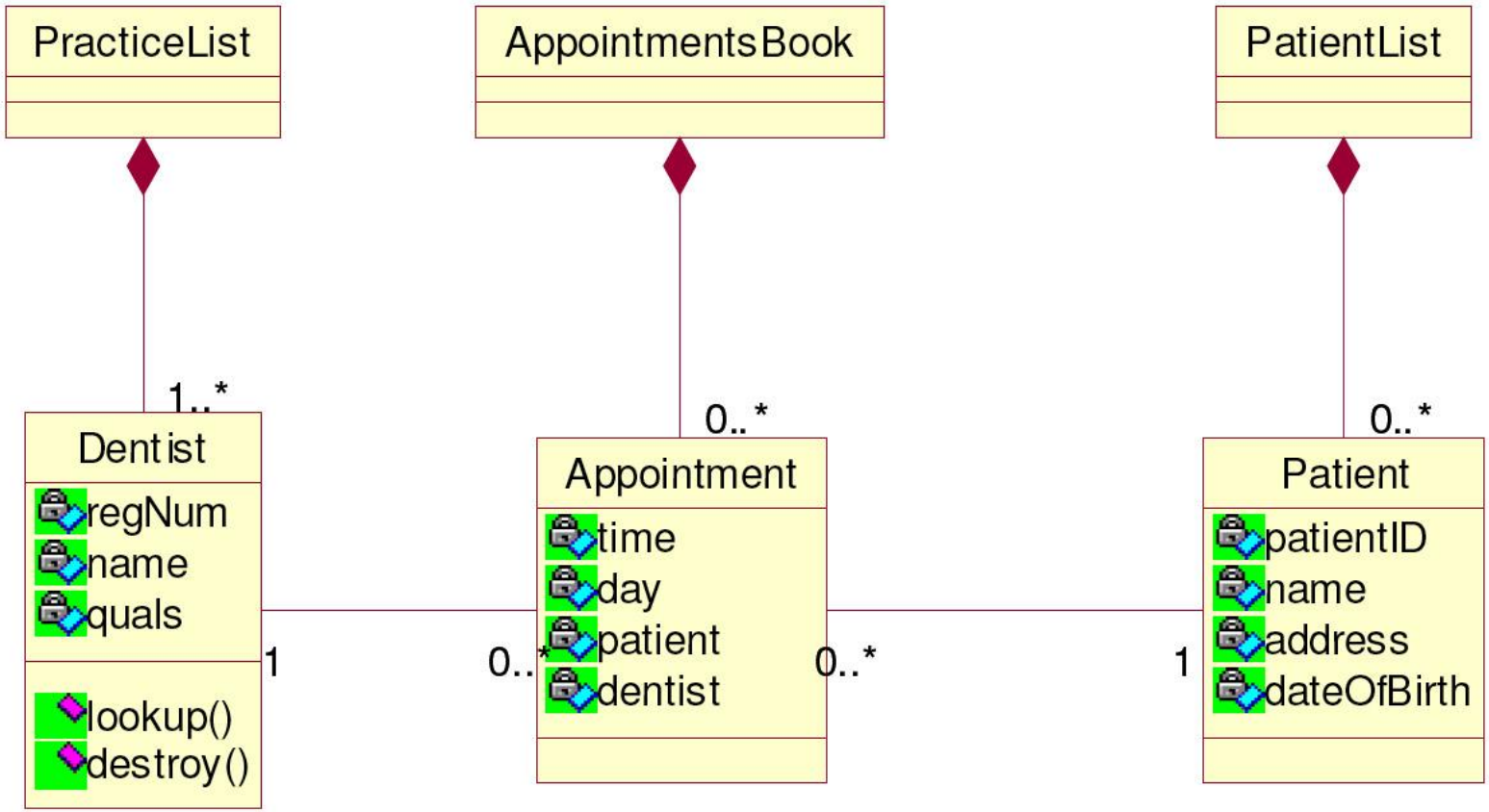
A System Sequence Diagram (SSD) for the use case “Make Appointment”.

Scenario: first choice appointment is not free.
Note: There will be one SSD for each scenario of each use case.



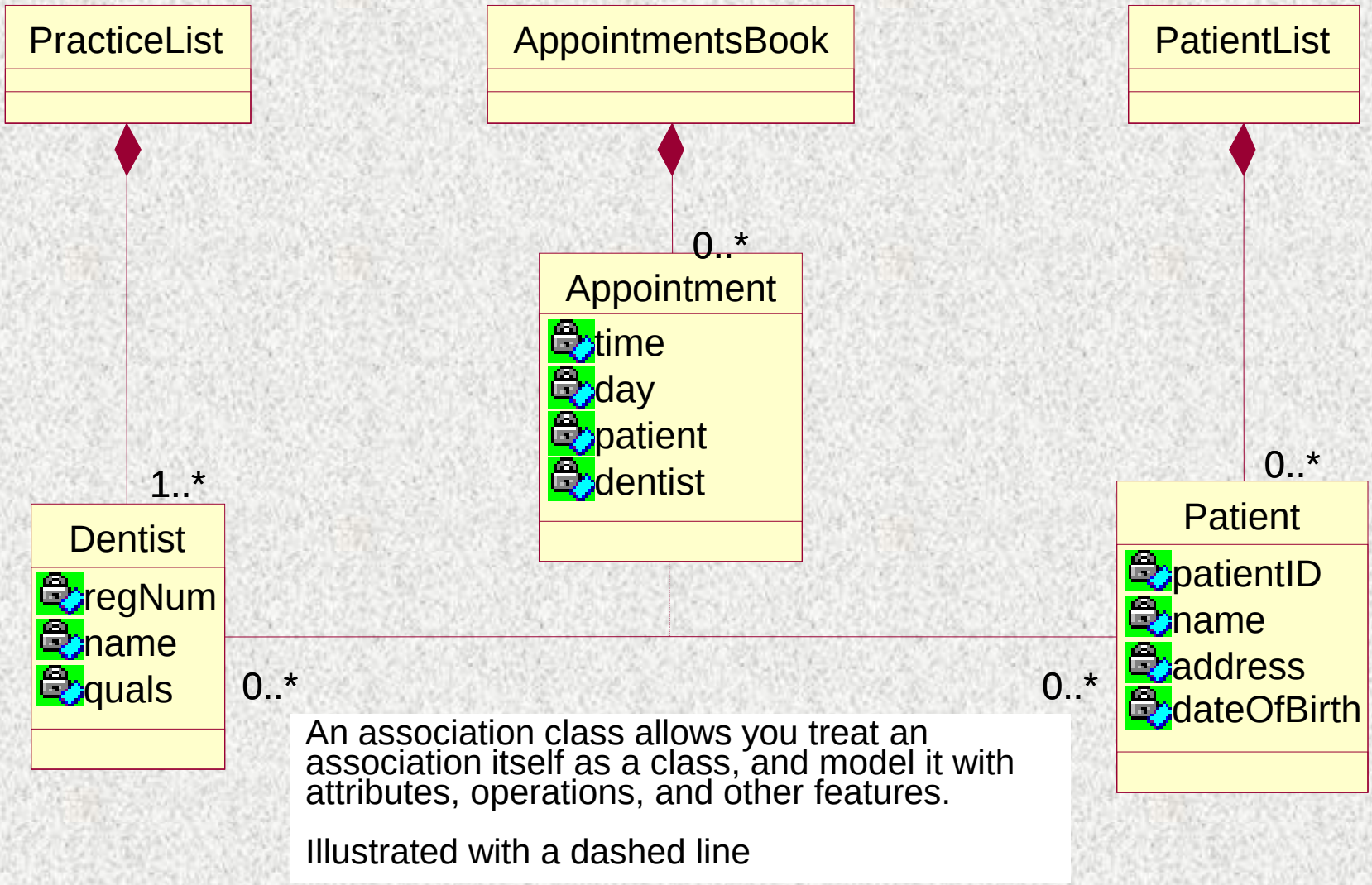
DOMAIN MODEL: Class Diagram

Dental Appointment System Class Diagram. Version 1



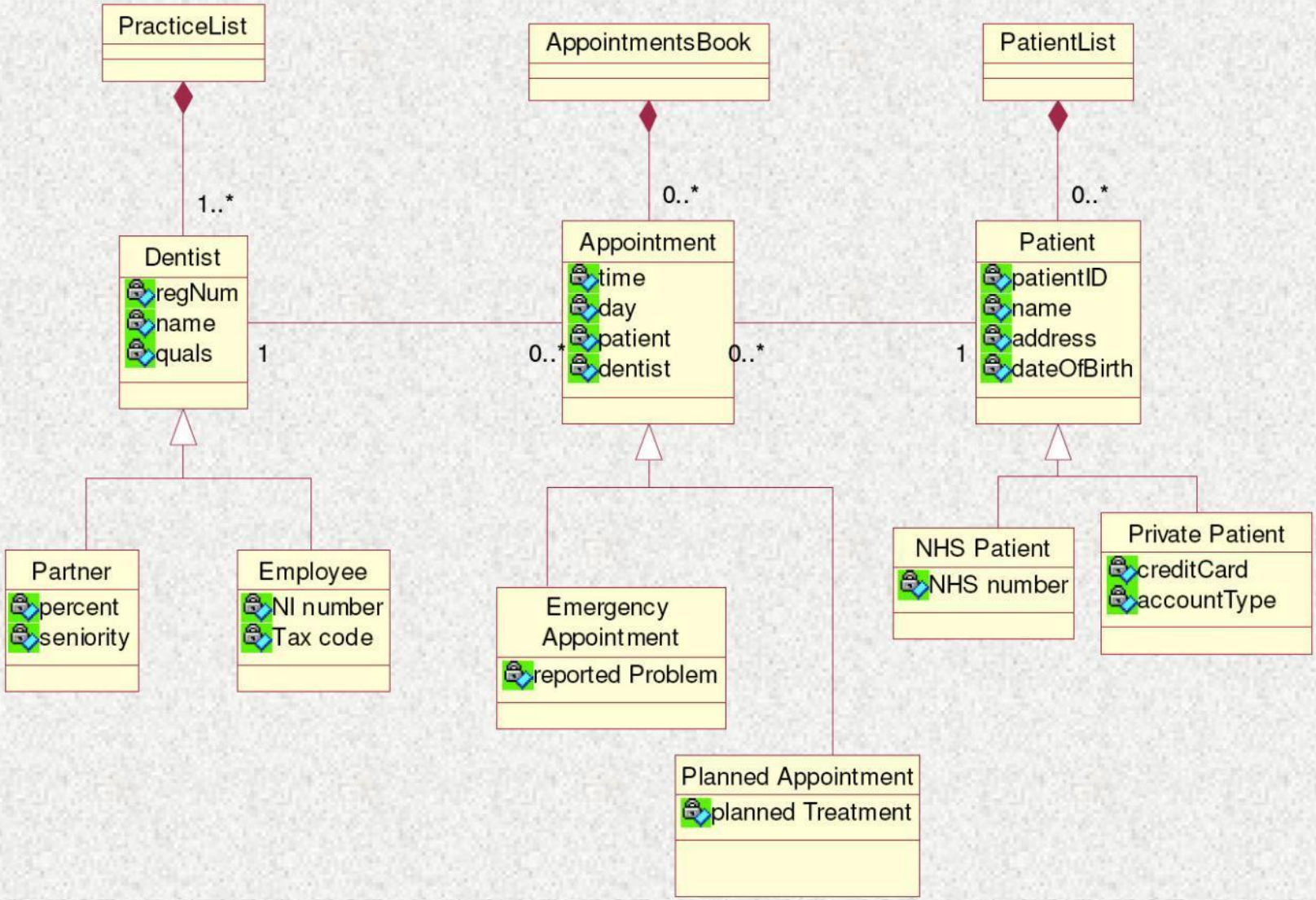
Review of UML Features

shows Appointment as an **Association Class**.



Review of UML Features

Potential **INHERITANCE**



Review of UML Features

The "is-a" and "has-a" and realisation rules

Note the difference between these.

Inheritance (follows the "is-a" rule)

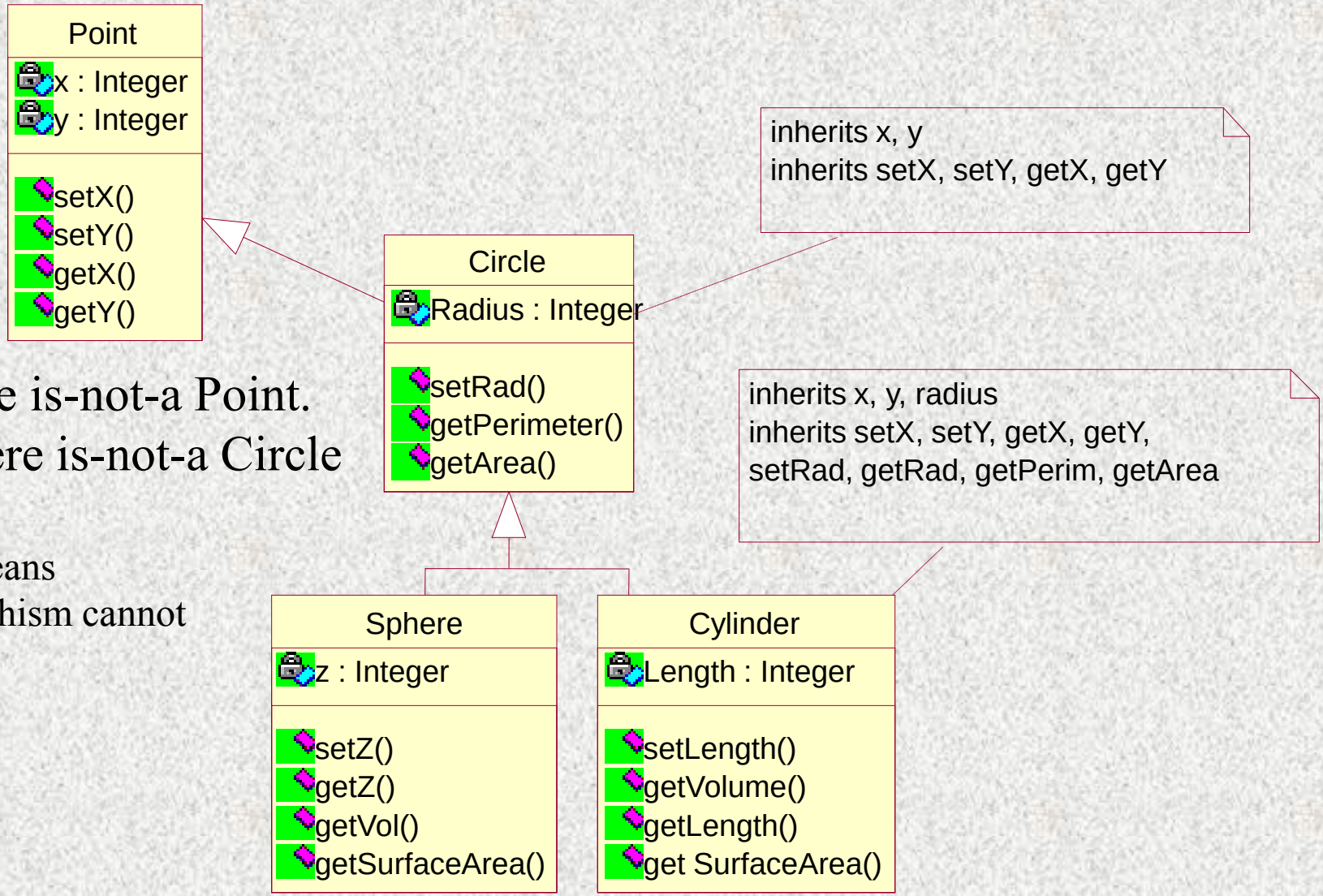
- a Partner **is a** Dentist
- a PrivatePatient **is a** Patient
- *but a Patient is not a PatientList*

Composition (follows the "has-a" rule)

- a PracticeList **has-a** Dentist (many of them).
- an AppointmentBook **has-a**n Appointment (many of them)

Review of UML Features

Poor Use of Inheritance



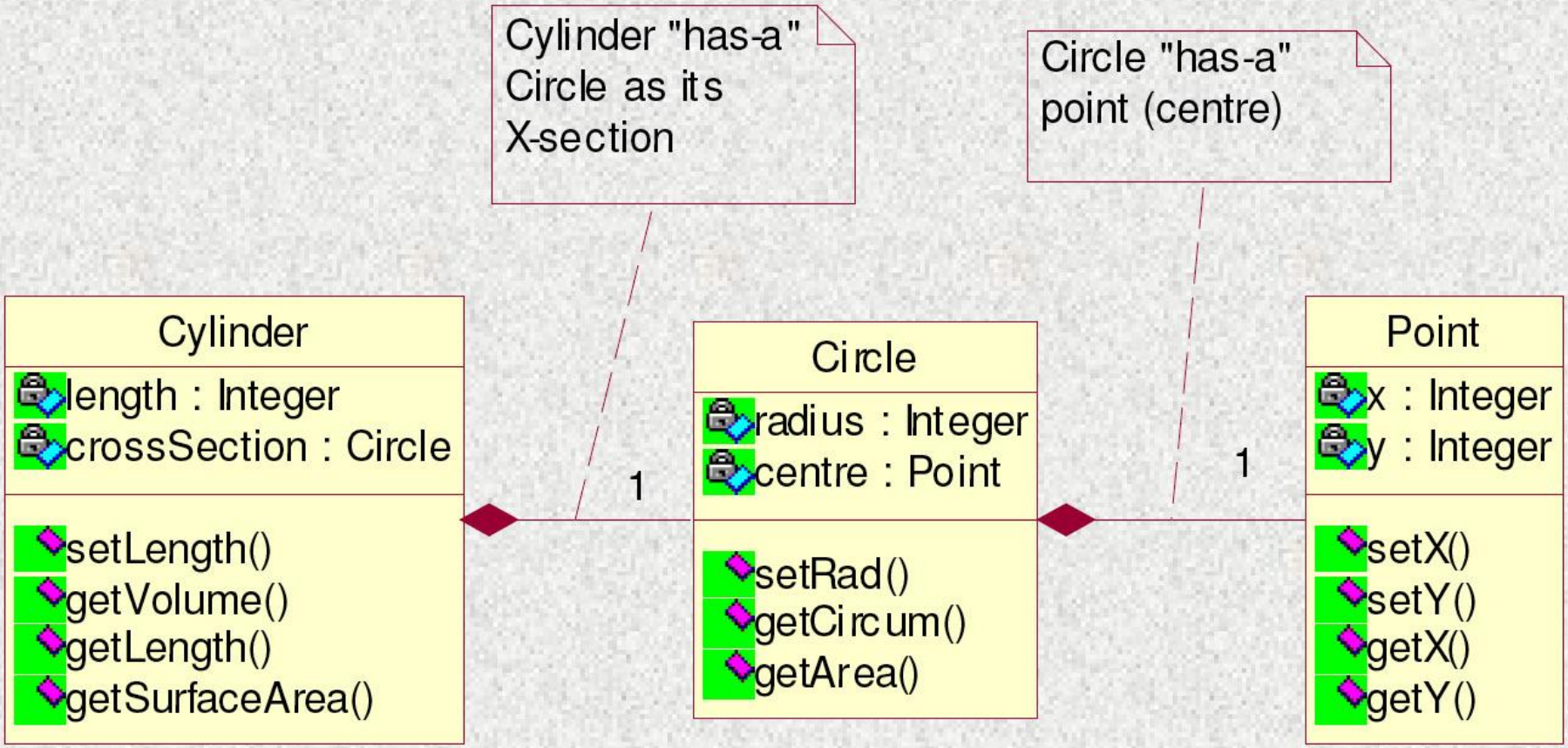
Poor:

- a circle is-not-a Point.
-A Sphere is-not-a Circle

Which means Polymorphism cannot be used.

Review of UML Features

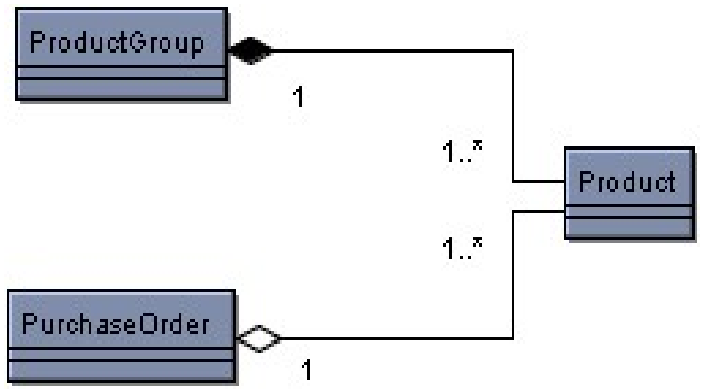
Better Use of Composition



Composition and Aggregation

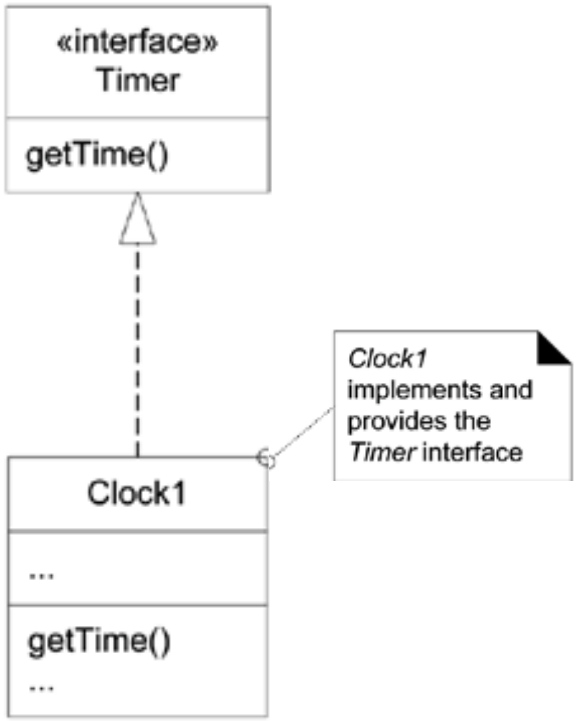
- The **composition** association is represented by the solid diamond.
- *ProductGroup* is **composed** of *Products*.
- This means that if a *ProductGroup* is destroyed, the *Products* within the group are destroyed as well.

- The **aggregation** association is represented by the hollow diamond.
- *PurchaseOrder* is an **aggregate** of *Products*.
- If a *PurchaseOrder* is destroyed, the *Products* still exist.



Notations to show interfaces in UML

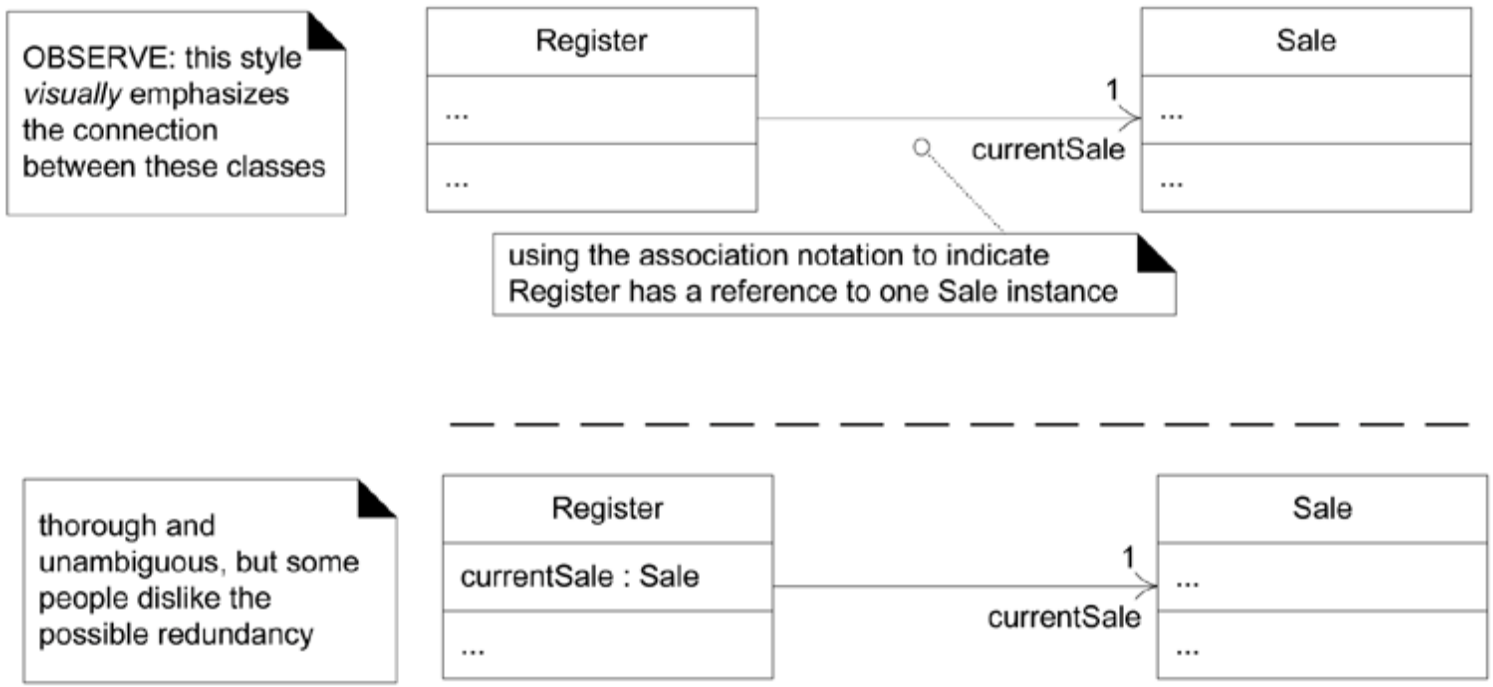
- Interface Realization



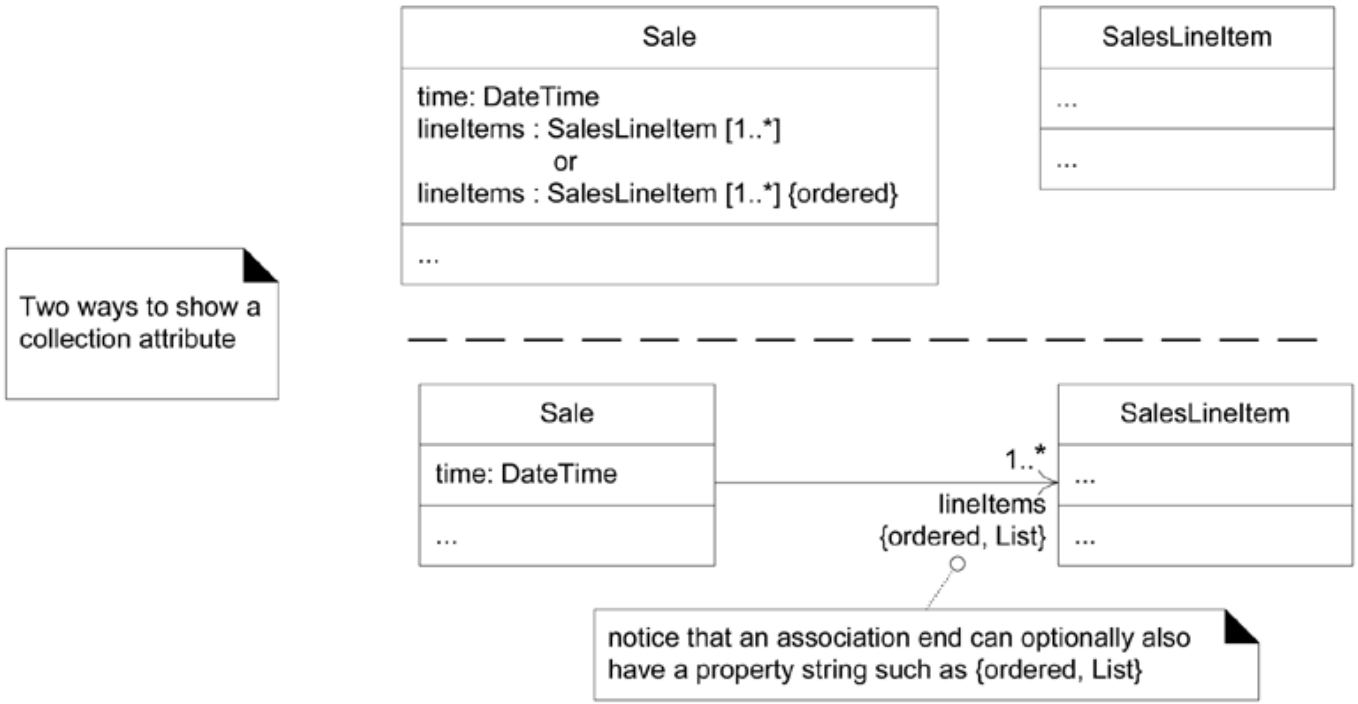
Realization is a semantic relationship

Specifies a contract that another class guarantees to carry out by implementing the abstract behaviours defined in the interface

Association line notation for a UML attribute



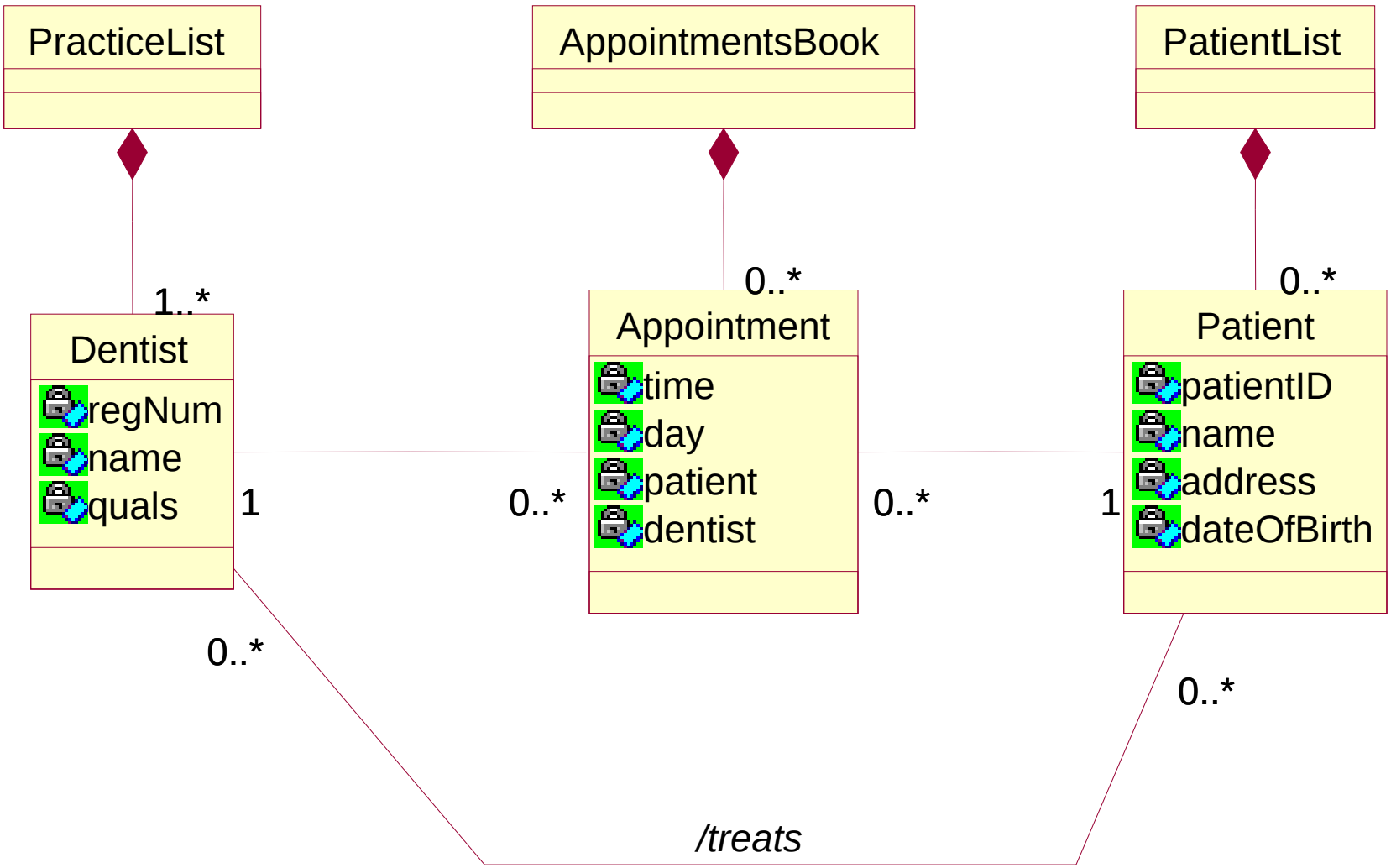
Two ways to show a collection attribute in the UML



- A property string such as {ordered} or {ordered, List} is possible.
- {ordered} is a UML-defined keyword that implies the elements of the collection are ordered.
- Another related keyword is {unique}, implying a set of unique elements.

Review of UML Features

Derived Association



Associations example 1

There is a "many-to-many" association between Module and Student. In particular, a Student can study many modules.

Navigation

- Navigability is a property of the role which indicates that it is possible to navigate uni-directionally across the association from objects of the source to target class.

- Navigability implies visibility.

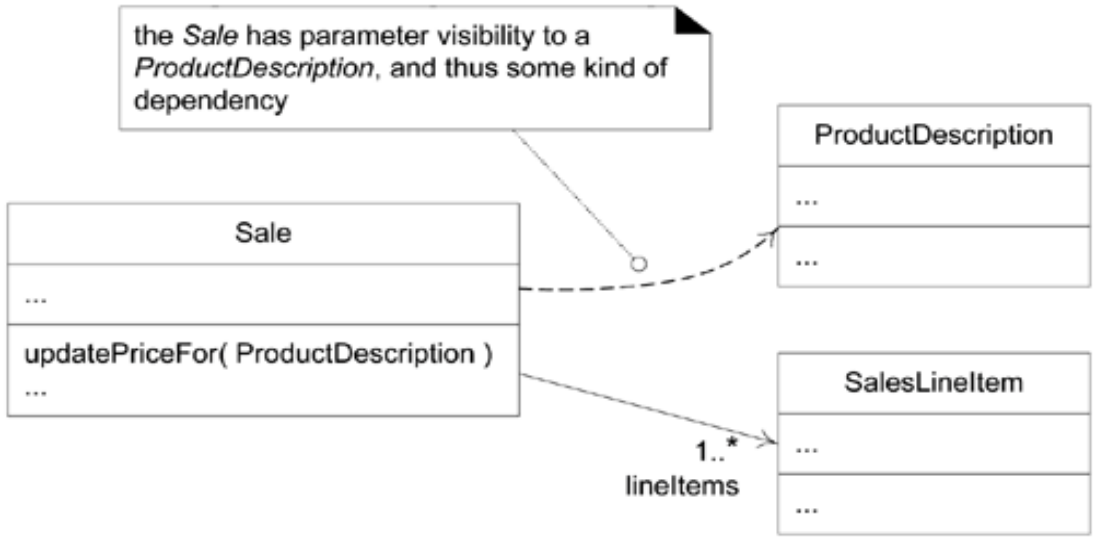
Dependency

- A dependency is a using relationship, specifying that a change in the specification of one thing may affect another thing that uses it.

- Indicates that a client element (of any kind, including classes, packages, use cases, and so on) has knowledge of another supplier element and that a change in the supplier could affect the client.

- Dependency can be viewed as another version of coupling, a traditional term in software development when an element is coupled to or depends on another.

Showing Dependencies

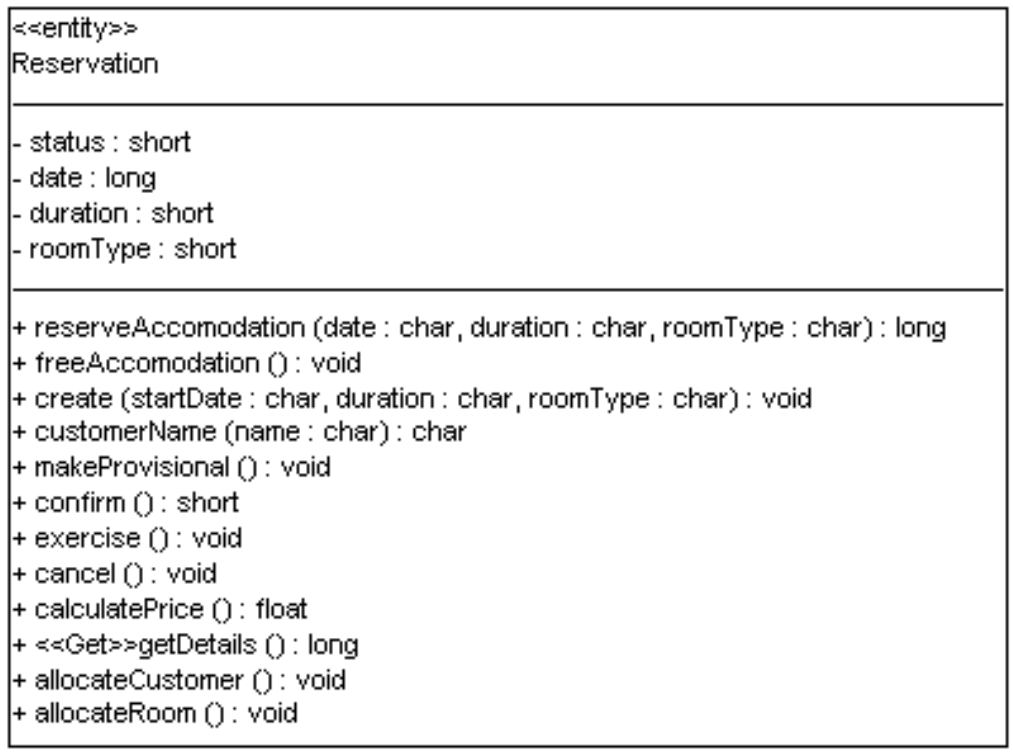


- There are many kinds of dependency
  - having an attribute of the supplier type
  - sending a message to a supplier; the visibility to the supplier could be:
    - An attribute, a parameter variable, a local variable, a global variable, or class visibility (invoking static or class methods)
  - receiving a parameter of the supplier type
  - the supplier is a superclass or interface

Operation vs. Method

- An operation is not a method
- A UML operation is a declaration, with a name, parameters, return type, exceptions list, and possibly a set of constraints of pre-and post-conditions.
- **it isn't an implementation**

- Methods are implementations
- Operation contracts in UML terms the definition of constraints for UML operations



Showing Methods in Class Diagrams

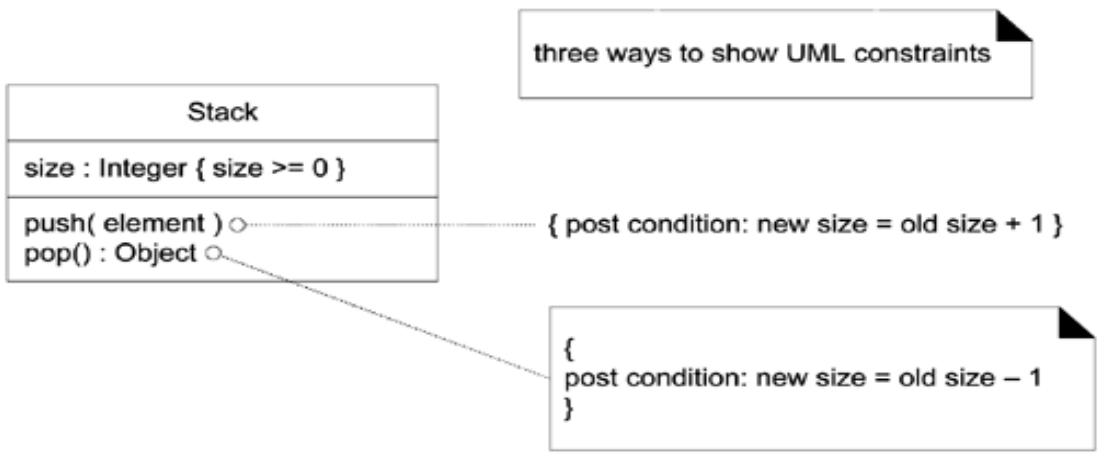
- Method Illustrations
  - in interaction diagrams, by the details and sequence of messages
  - in class diagrams, with a UML **note symbol** stereotyped with «method»

A note symbol may represent several things, such as:

a UML note or comment, which by definition have **no semantic impact**
a **UML constraint**, in which case it must be encased in braces '{…}'
a **method body**- the implementation of a UML operation

Constraints

- Constraints may be used on most UML diagrams, but are especially common on class diagrams.
- A UML constraint is a **restriction or condition** on a UML element.
- It is visualized in text between braces;
  - for example: { size >= 0 }.
- The text may be natural language or anything else,
  - such as UML's formal specification language, the Object Constraint Language (OCL).



Keywords

- A UML keyword is a textual adornment to categorize a model element.
- Few sample predefined UML keywords include:

Keyword	Meaning	Example Usage
«actor»	classifier is an actor	in class diagram, above classifier name
«interface»	classifier is an interface	in class diagram, above classifier name
{abstract}	abstract element; can't be instantiated	in class diagrams, after classifier name or operation name
{ordered}	a set of objects have some imposed order	in class diagrams, at an association end

Now back to the "Design" Theme

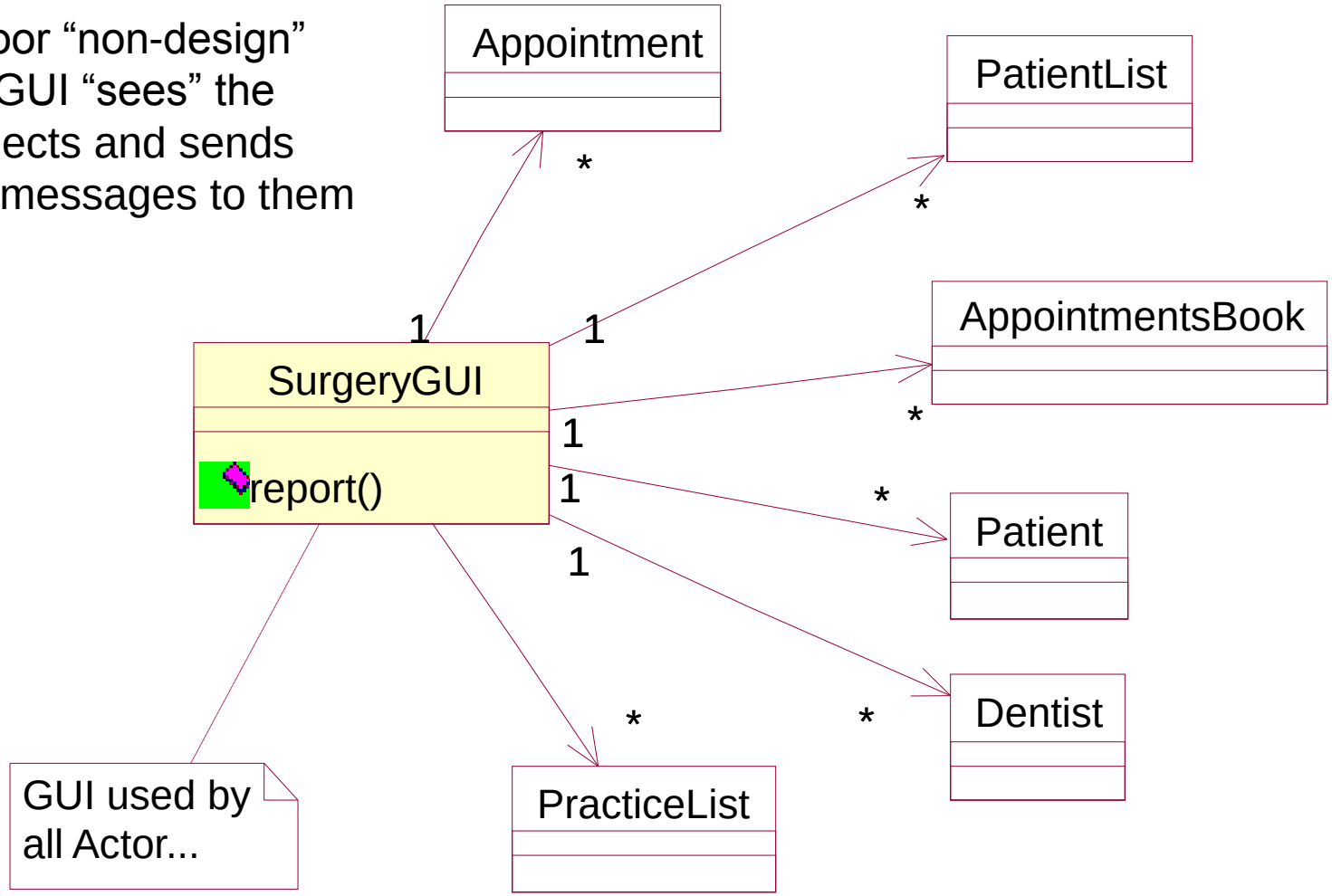
OO DESIGN STAGE

- Designer produces an OO Model of the intended software solution.
- This new model is based on the preceding domain (conceptual/analysis) model but is not the same.
- This new OO model is the one the team will implement (in Java or C++?)

- **SEPARATION OF CONCERNS**
- **HIGH COHESION**
- **LOW COUPLING**
- MODIFIABILITY
- MODULARITY
- ABSTRACTION
- INFORMATION HIDING & DETAIL HIDING:
- ENCAPSULATION:
- SOFTWARE REUSE

Design Example 0: (Worst Case)
for the Dental Appointments System

This is a poor “non-design” where the GUI “sees” the domain objects and sends controlling messages to them



Problems with this Design

- It is an extremely poor design (but it could be made to work and meet the functional requirements).
- The so-called "GUI" object does everything. There is no separation of concerns, low cohesion, high coupling and poor modifiability. The GUI handles:
  - GUI concerns of the receptionist
  - GUI concerns of the practice manager
  - Business/control logic concerns of all five use-cases.

Overview of OO Design Stage
(using simple layered approach)

- Build a **Static Model** of the Design based on the Domain Model
  - Retain **Domain Classes** from analysis stage [basis of *domain layer*]
  - Add **Boundary Classes** to handle communication and interaction with actors [basis of the *UI layer*]
  - Add **Control Classes** to co-ordinate functionality carried out by the use cases [basis of *Application layer*].
  - Add **Broker Classes** to solve the problem of storing persistent data to a database [part of the *Technical Services Layer*] (will not cover these today)
- Build a **Dynamic design model** comprising use case realisations.

Entity/Data Classes [Domain Layer]

- Carried forward from the domain model.
- Map from “real-world” business classes identified in the conceptual model.
- Represent business concepts about which we need to store data (and sometimes called data classes).

Boundary Classes [User Interface Layer]

- Boundary Classes model interactions between the system and its actors (human roles, or external systems).
- For human roles they are often abstractions for GUI's.
- For external software or hardware they are abstractions for
  - API's (application programming interface).
  - hardware device drivers.

Data Broker Classes

[part of Technical Services layer]

- Objects (class instances) stored in working memory are not persistent. *i.e. if the power is switched off the data is lost.*

- The Data Broker pattern solve the problem of storing and retrieving persistent objects to and from a database by using data broker classes. *We will not cover this today.*

Controllers or Control Objects
[Application Layer]

- "A non-user interface object responsible for handling a system event. A controller defines the method for the system operation". *(Craig Larman)*

- Encapsulate behaviour whereby other objects collaborate and interact with each other. *(based on Booch and others)*

- Pure software abstractions: they do not map from "real world" objects (unlike boundary classes and entity classes).