M19COM Software Development & Design

Week 8: GRASP Design Patterns

Dr Faiyaz Doctor

Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution
- A pattern is a description of the problem and the essence of its solution
- It should be sufficiently abstract to be reused in different settings
- Patterns often rely on object characteristics such as inheritance and polymorphism

Pattern elements

- Name
  - A meaningful pattern identifier
- Problem description
- Solution description
  - Not a concrete design but a template for a design solution that can be instantiated in different ways
- Consequences
  - The results and trade-offs of applying the pattern

Design Patterns

... you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented design more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

Gamma et al [1995]

Broad Categories of Design Pattern

- Specific Design Patterns for more specific but common contexts/problems.
  - e.g. the “Observer” and “data broker” patterns
  - Usually specific enough to include a UML model of the outline solution. What people mean when they refer to design patterns.

- More General Patterns giving fundamental approaches for assigning responsibilities to objects.
  - e.g. the “GRASP” set of patterns.
  - These repackage fairly basic well-known design guidelines.

The GRASP Patterns

General
Responsibility
Assignment
Software
Patterns

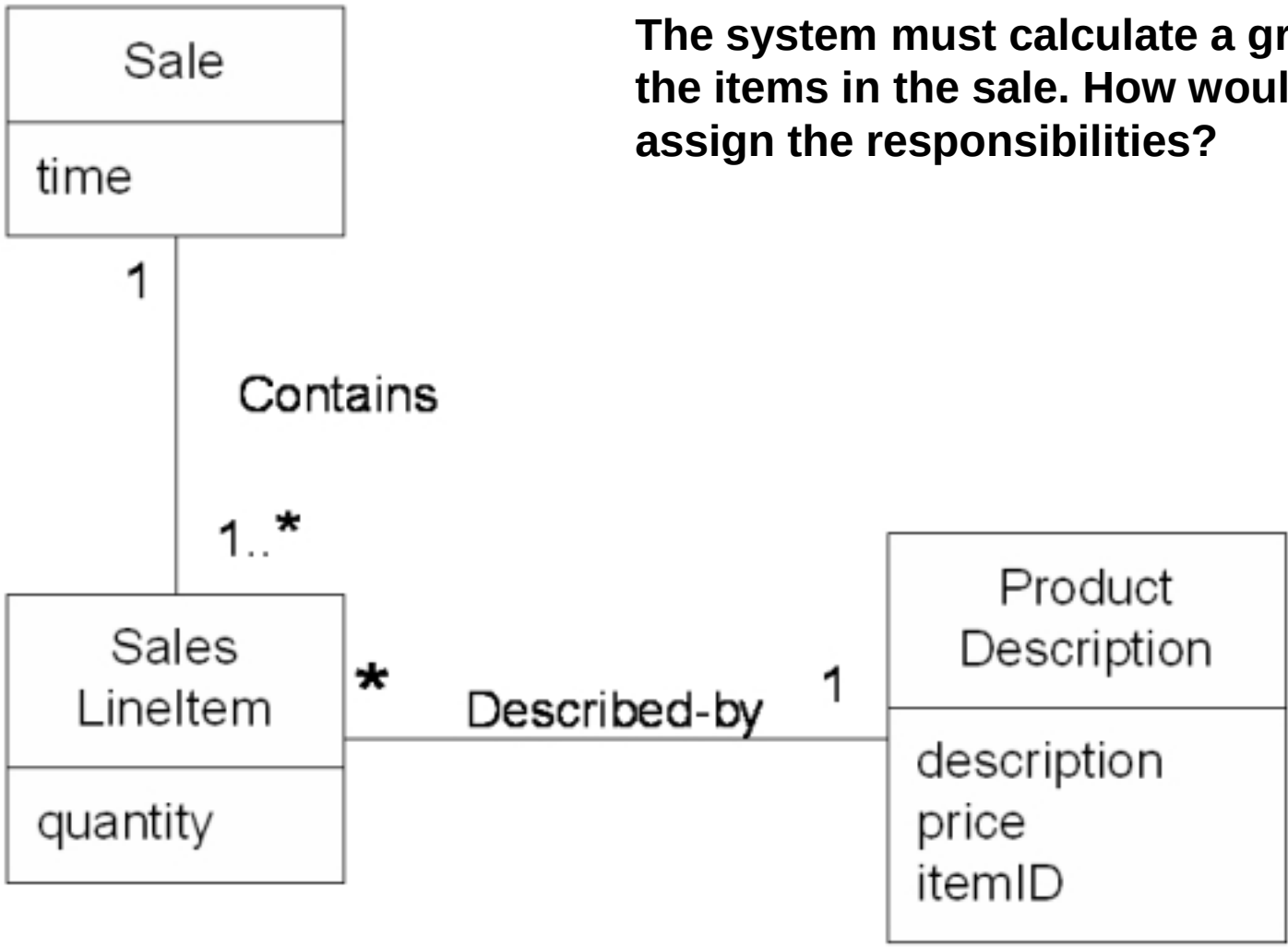
- Expert
- Creator
- High Cohesion
- Low Coupling
- Polymorphism
- Controller
- Pure Fabrication
- Indirection
- Don’t Talk to Strangers

- http://sourcecodemania.com/grasp-design-patterns/

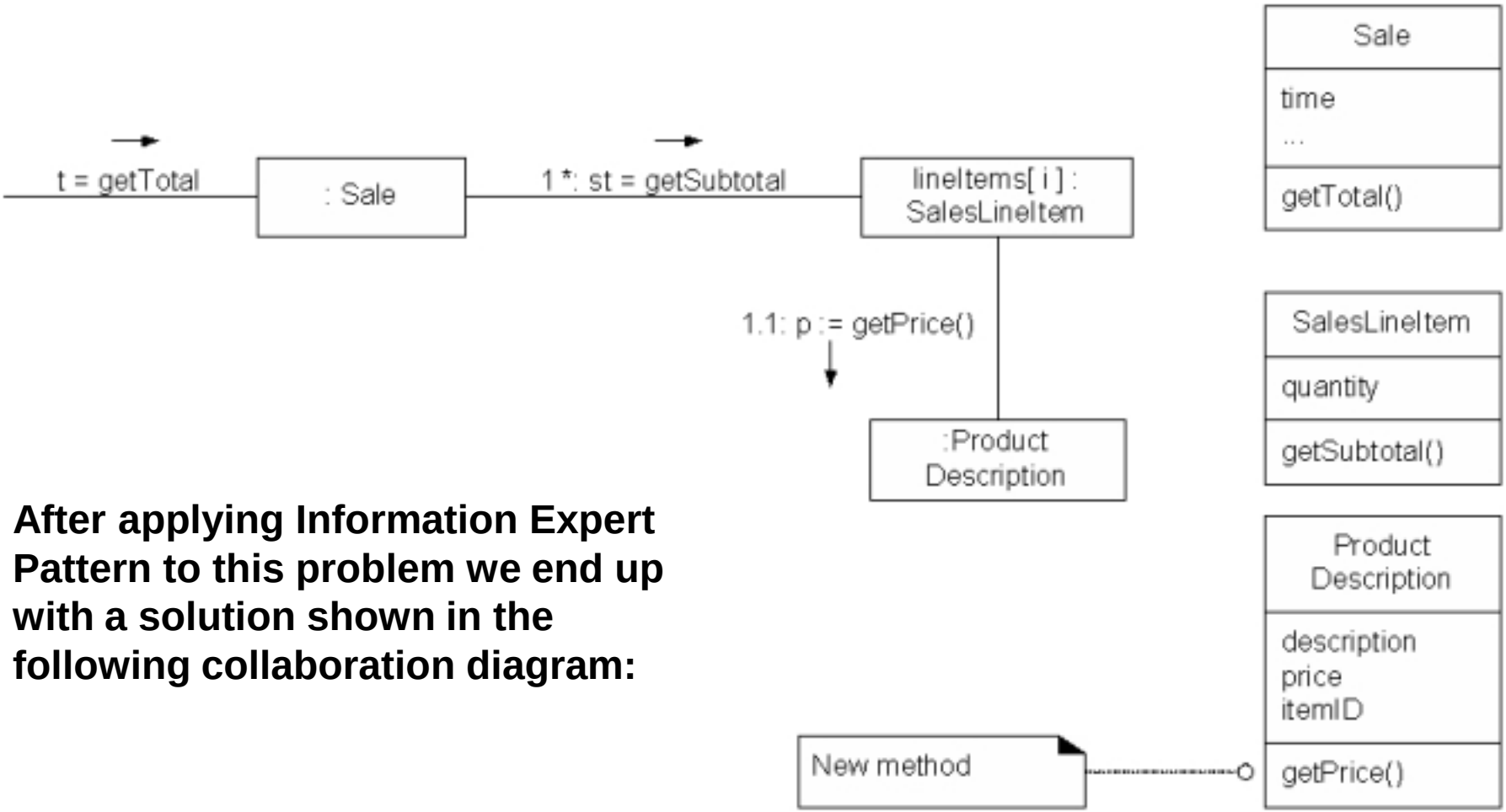
GRASP Pattern 1: **Expert**

- **Problem:** What is the most basic principle by which responsibilities are assigned in object oriented design?
- **Solution:** Information Expert helps us decide, once we know the task (responsibility), which class to make responsible for carrying out the task.
- Assign responsibility to the information expert – the class that has the information necessary to fulfil the responsibility.

Based on "Applying UML and Patterns" by Craig Larman



The system must calculate a grand total of the items in the sale. How would you assign the responsibilities?



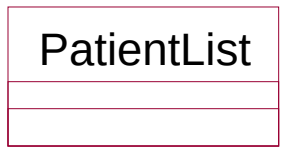
After applying Information Expert Pattern to this problem we end up with a solution shown in the following collaboration diagram:

http://web.cs.wpi.edu/~gpollice/cs4233-a05/CourseNotes/maps/class4/InformationExpert.html

Example of the Expert Pattern.

Actual Problem: We wish to extend the dental appointments system. Which class should be responsible for knowing the total number of patients.

Actual Solution: The information expert on this is the class PatientList. We add a method getNumPats to that class.



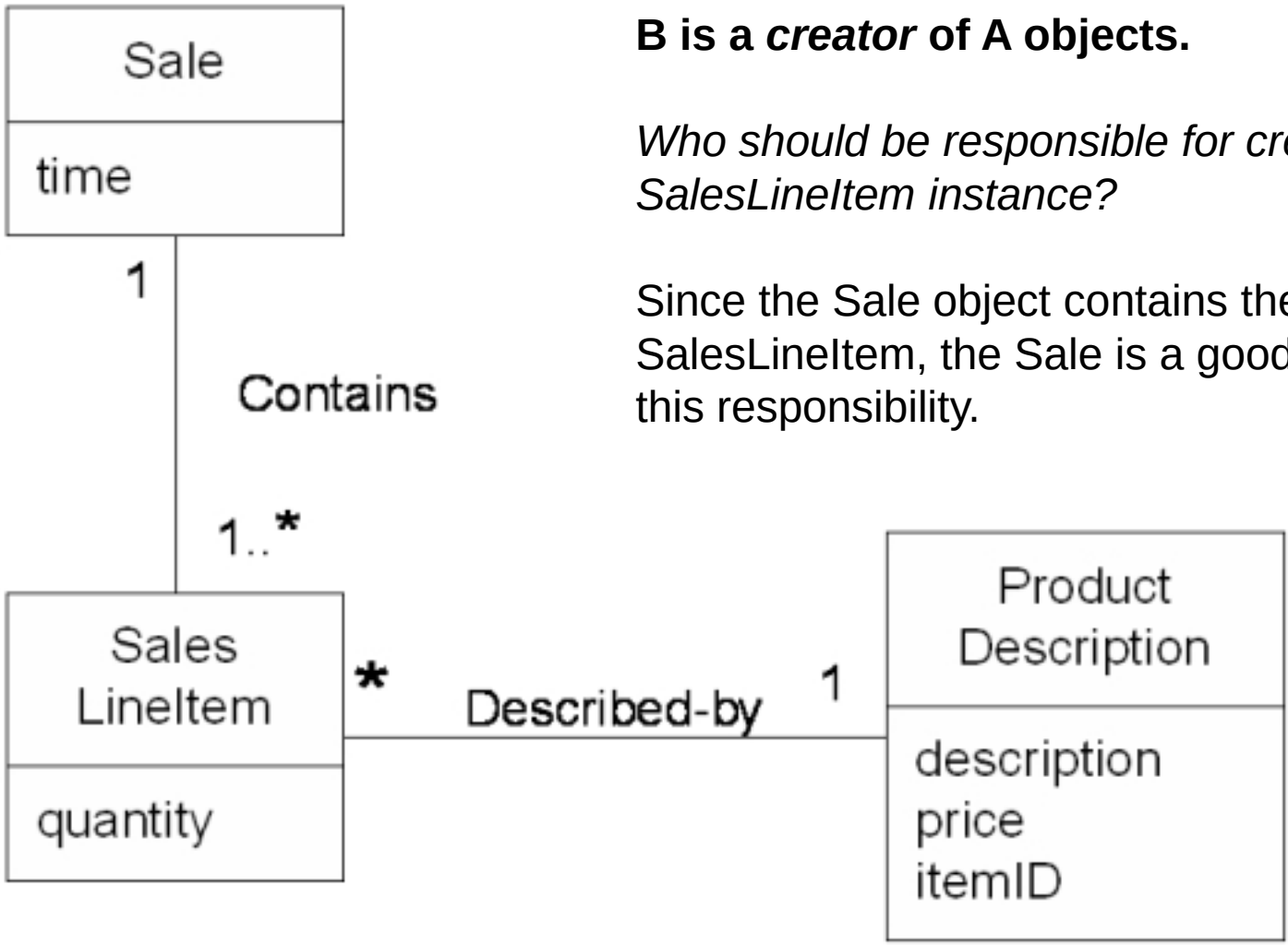
GRASP Pattern 2: **Creator**

Problem: Who should be responsible for creating a new instance of some class?

Solution: Assign this responsibility to class B if one of the following is true:

- **B aggregates A objects**
- **B contains A objects**
- **B records instances of A objects**
  (i.e. it has attributes of class A)
- **B closely uses A objects**
- **B has the initialising data that will be passed to A when it is created** *(This means it is an Expert too)*

Based on "Applying UML and Patterns" by Craig Larman



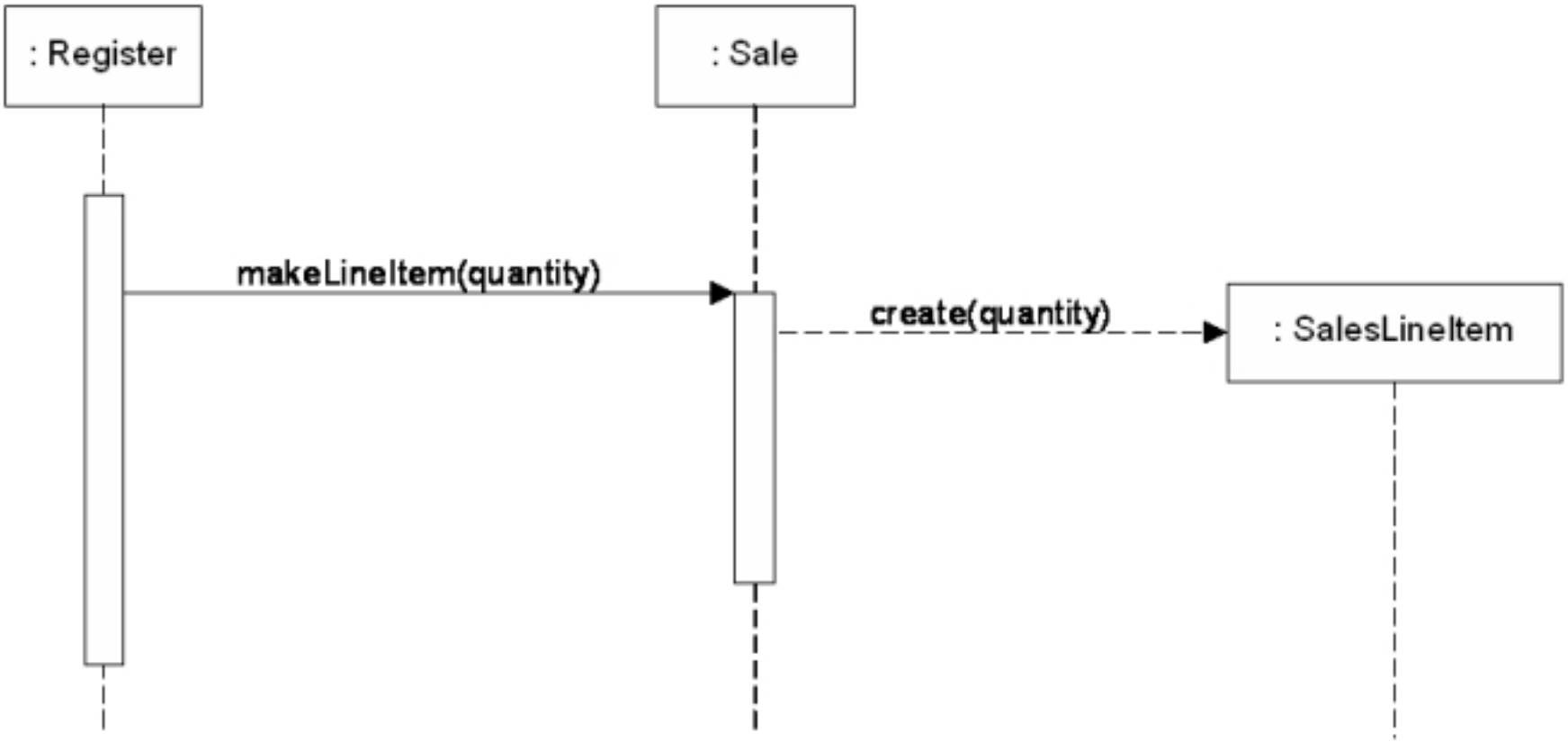
B is a *creator* of A objects.

Who should be responsible for creating a SalesLineItem instance?

Since the Sale object contains the SalesLineItem, the Sale is a good candidate for this responsibility.

http://web.cs.wpi.edu/~gpollice/cs4233-a05/CourseNotes/maps/class4/Creator.html

The sequence diagram showing this is….



When creation requires significant complexity, such as:

- *using recycled instances for performance reasons,*
- *conditionally creating an instance from one of a family of similar classes based upon some external property value, and so forth,*

You might want to delegate the creation to a class that is specifically designed for such a purpose. Consider the Factory pattern not covered today

Example of the Creator Pattern.

Actual Problem: In our dental appointments system which class should be responsible for creating a new appointment.

Actual Solution: The AppointmentsBook is an aggregation of Appointment objects so it should have responsibility for creating new Appointment objects.

GRASP Pattern 3: **Low Coupling**

Problem: How to support low dependency and increased reuse.

- Related classes should not be tightly dependent on the internal details of each other,
- Changes ripple through the system, and the system is potentially harder to understand

Solution: Assign the responsibility so that coupling remains low, isolates changes, classes can be reused in different applications

GRASP Pattern 4: **High Cohesion**

Problem: How to keep complexity manageable, do classes have well defined role in the system, are contents of the class related to it?

Solution: Assign the responsibility so that cohesion remains high. If you find a subset of methods and fields that could easily be grouped separately under another class name, then these should be extracted to a new class.

Based on "Applying UML and Patterns" by Craig Larman

GRASP Pattern 5: **Polymorphism**

Problem: How to handle alternative behaviour that depends on type.

Solution: Assign responsibility for the alternative behaviour – using polymorphic operations – to the types (sub-classes) for which the behaviour varies.

Example of the Polymorphism pattern

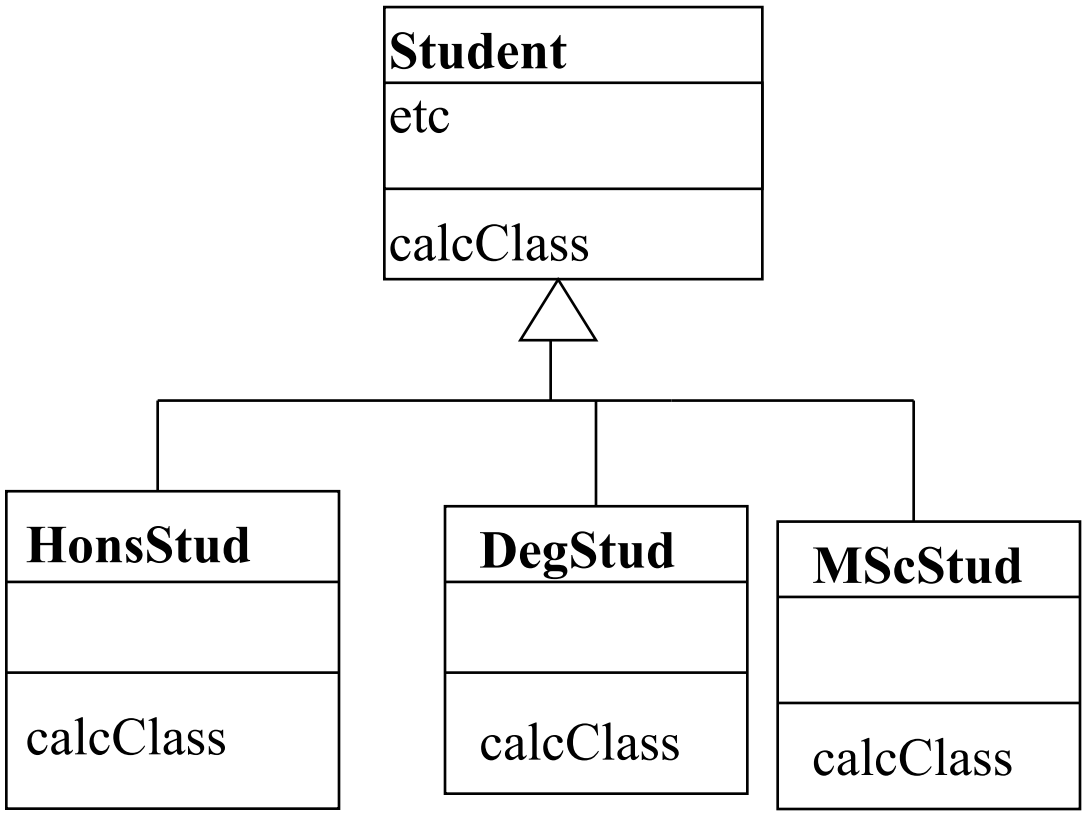
Solution: Make each type a subclass of Student. Student will have an operation “calcClass”. In subclass HonsStud we override/redefine this inherited operation to define how it is calculated for Hons students, and in DegStud to reflect how it is done for Degree students.

Our code then becomes merely:

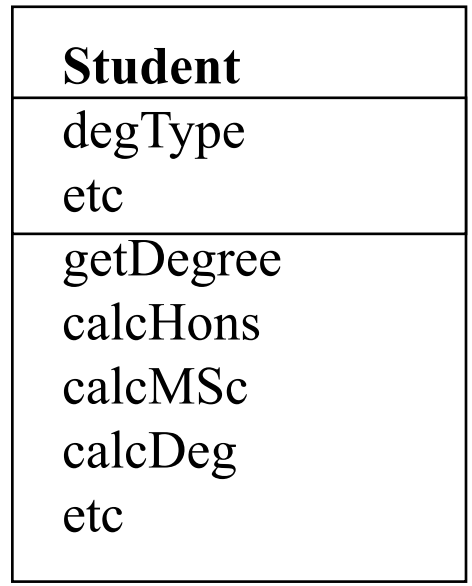
```
stud.calcClass
```

and we do not need to check what type of student it is every time.

Example of the Polymorphism pattern



Polymorphism Pattern



Non Polymorphic approach

Example of the Polymorphism pattern

Problem: We need to calculate the degree classification of HonsStud and DegStud in different ways.

Further Explanation: We might write this code:

```
if stud.getDegree() ="hons" then
stud.calcHons()
else stud.calcDeg();
```

This is poor & difficult to extend as we add more types (e.g. MSc and HND students).

GRASP Pattern 6: **Controller**

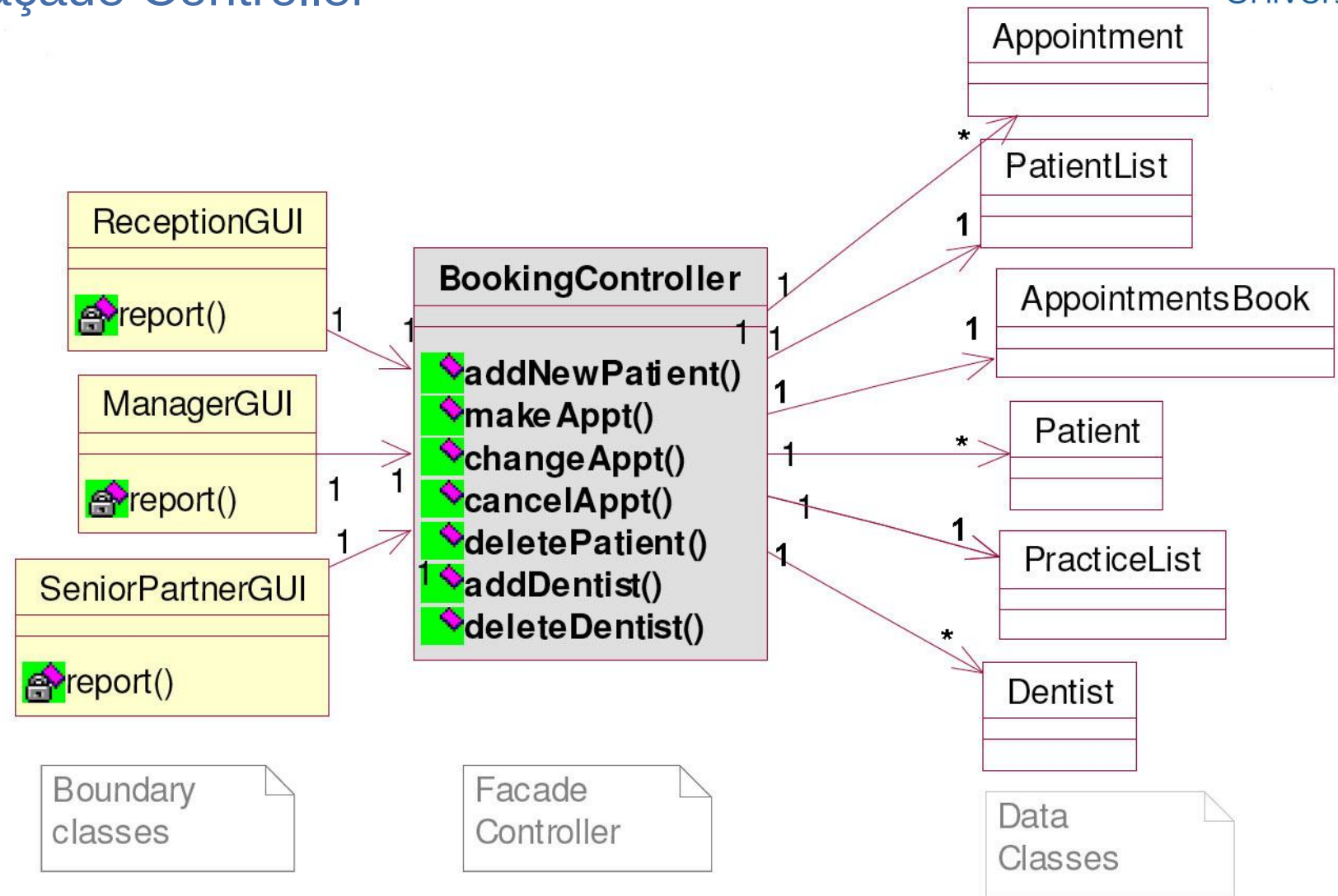
Problem: Who should be responsible for handling a system event? (e.g. button clicked in GUI to initiate a use-case).

Solution: Assign responsibility for handling a system event to one of the following types of class:

- **Façade Controller**
- **Role Controller**
- **Use Case Controller**

Based on "Applying UML and Patterns" by Craig Larman

Design Model 1 using a Façade Controller

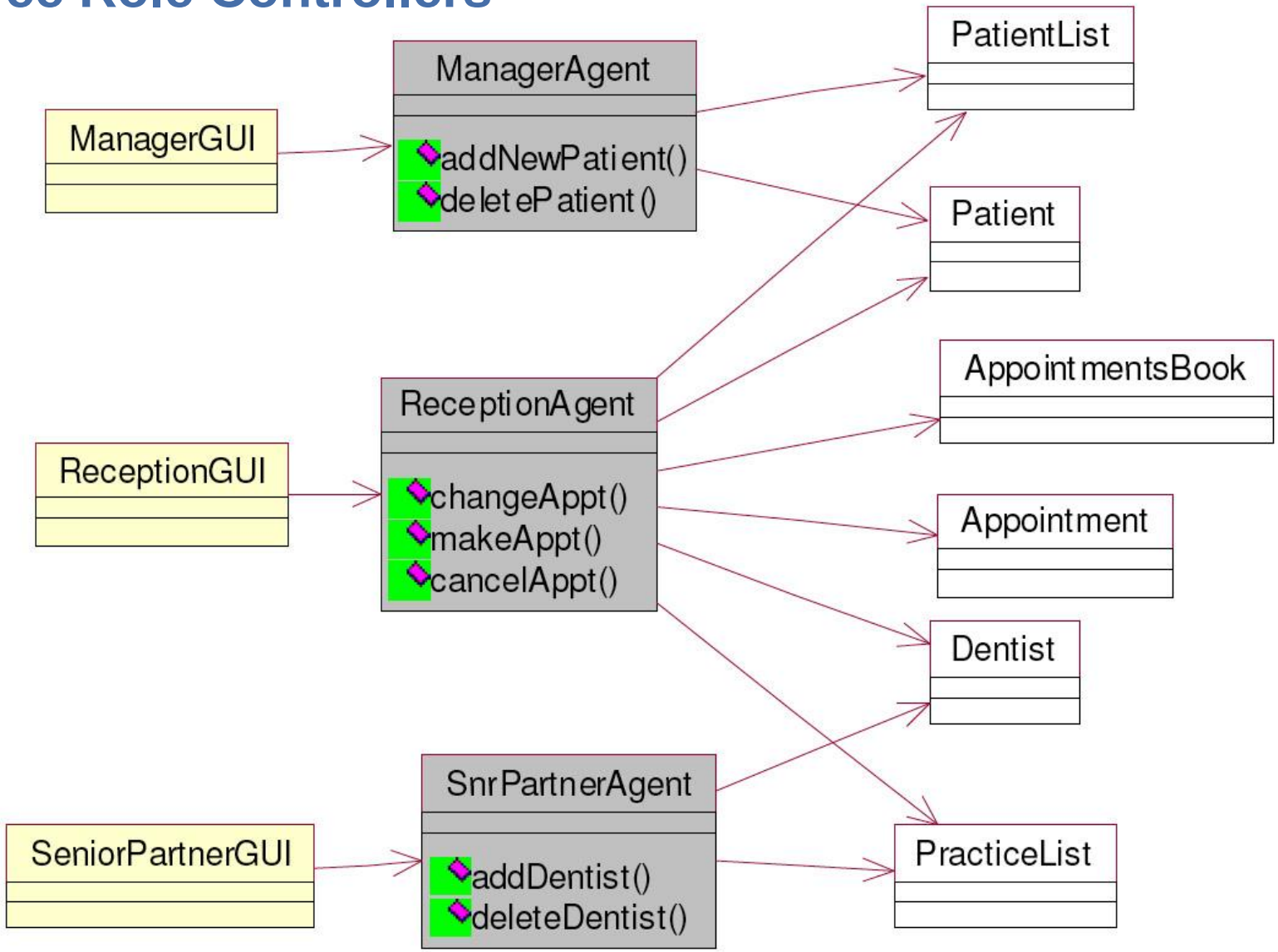


Note: This is not the best design

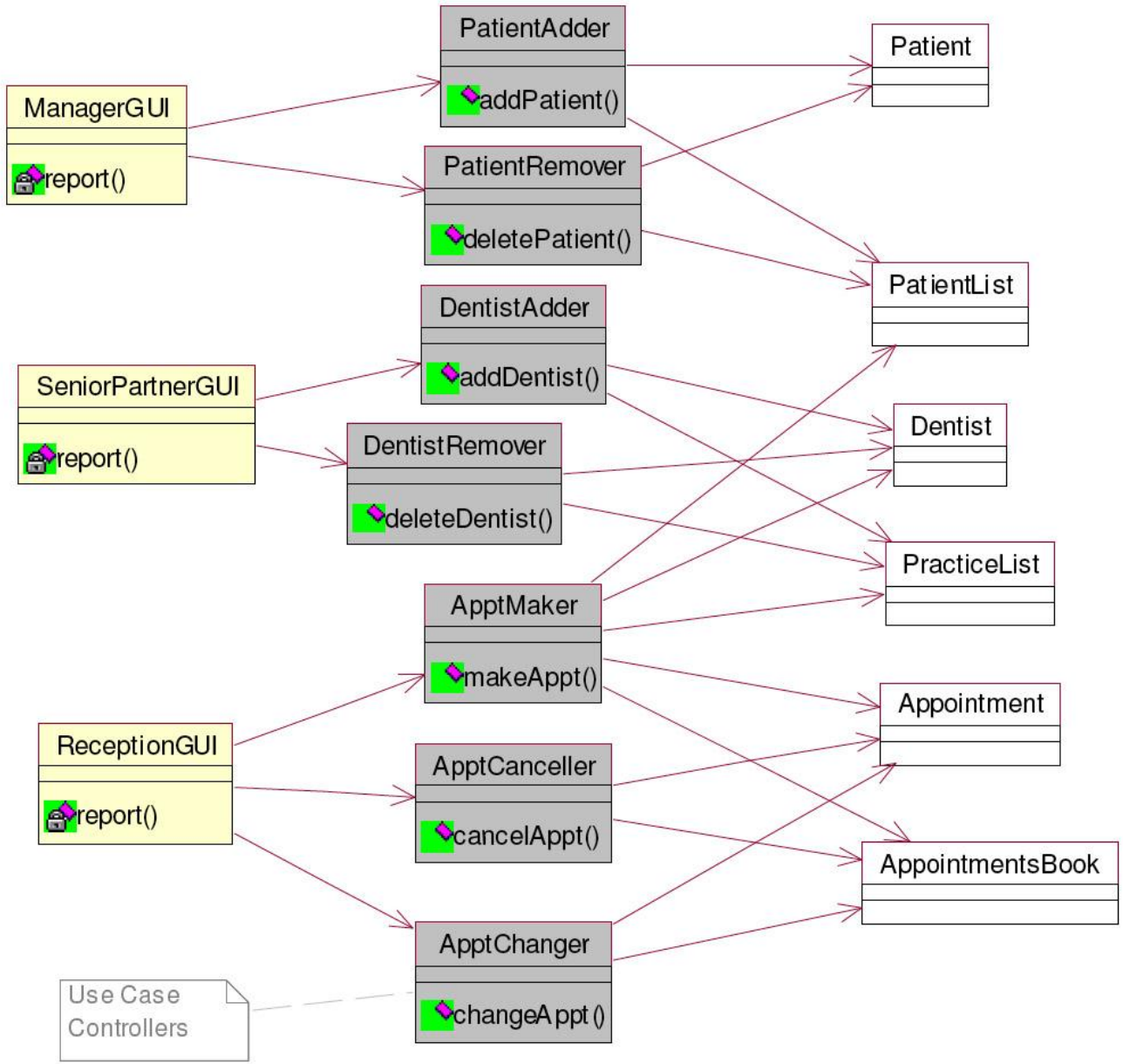
The Three Types of Controller applied to the Dental Appointments System

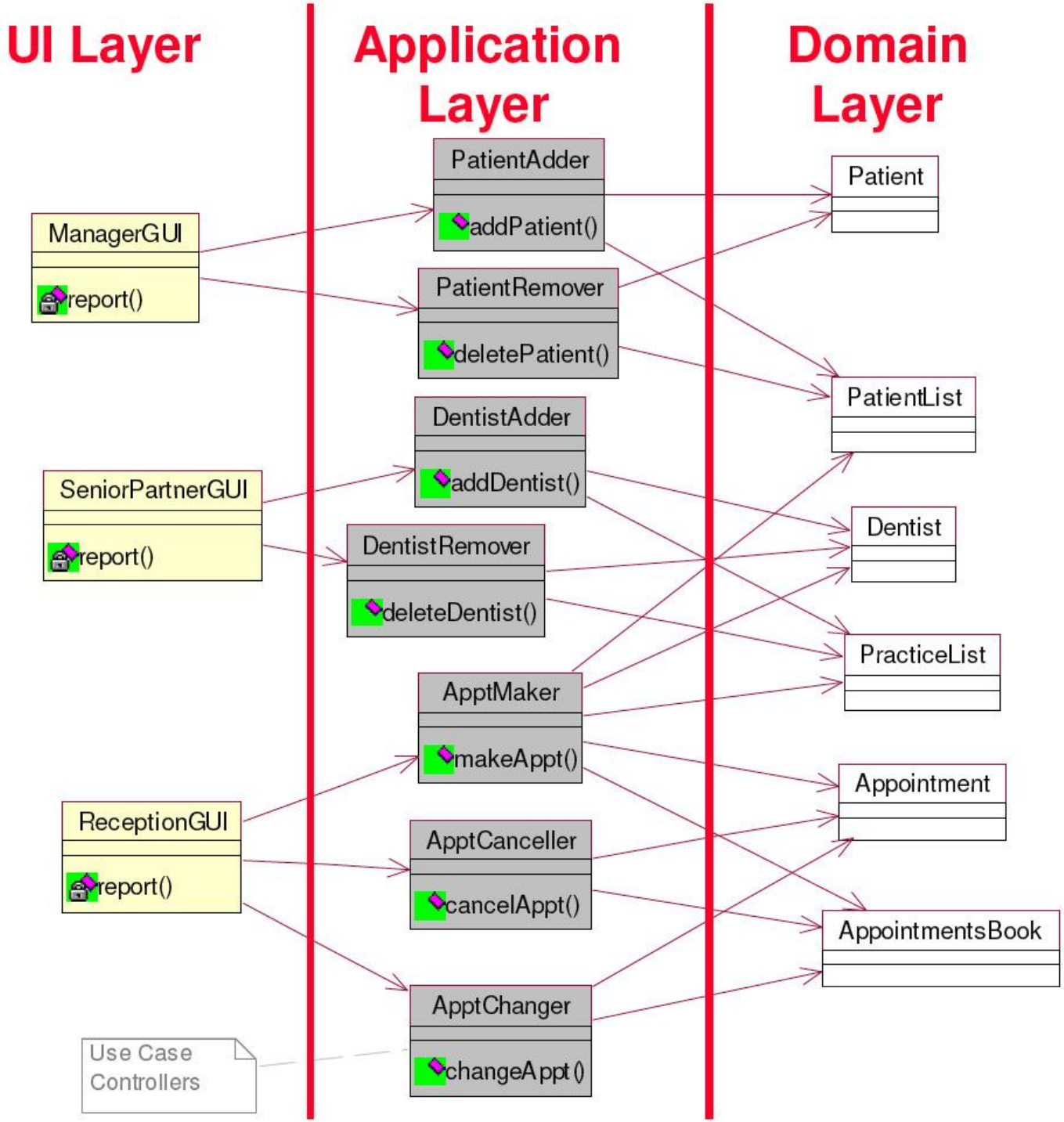
- **Façade Controller:** One class represents the "system" or business and controls everything in that system.
- **Role Controller:** One class represents a role (i.e. actor) and controls all the functionality required by that actor.
  Our system needs 3 controllers to co-ordinate the functionality (use cases) needed by our three separate actors.
- **Use Case Controller:** One class represents one use case and controls the functionality in carrying out that use case.
  Our system needs 7 controllers: one for each use case.

Design Model 2 using Three Role Controllers



Design Model 3 using Seven Use Case Controllers




UI Layer
Application Layer
Domain Layer
ManagerGUI
report()
SeniorPartnerGUI
report()
ReceptionGUI
report()
PatientAdder
addPatient()
PatientRemover
deletePatient()
DentistAdder
addDentist()
DentistRemover
deleteDentist()
ApptMaker
makeAppt()
ApptCanceller
cancelAppt()
ApptChanger
changeAppt()
Use Case Controllers
Patient
PatientList
Dentist
PracticeList
Appointment
AppointmentsBook

Good Points of the Three Designs

- GUI concerns are separated from the business/control logic concerns.
- The GUI and Control concerns are separated from the business objects whose primary concern is to store data.
- The interface concerns of one actor are separated from the interface concerns of the other actor.

Relative Merits of the Three Design Models

- **Façade Controller**
  With this approach large systems may end up with a "bloated controller"‡ with:
  - **poor cohesion**. It controls all high and low level detail of many unrelated functions
  - **high coupling**. It communicates with all classes in the system. Acts as a "central switchboard" for all data in the system.

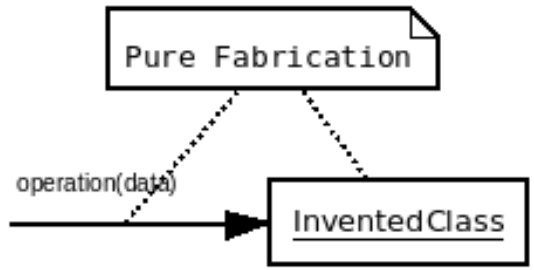
- **Role Controller** & **Use Case Controller**
  Generally lower coupling and higher cohesion.
  - narrower self-contained functionality
  - communicate with fewer classes.

‡ *"Applying UML and Patterns" by Craig Larman.*

GRASP Pattern 7: **Pure Fabrication**

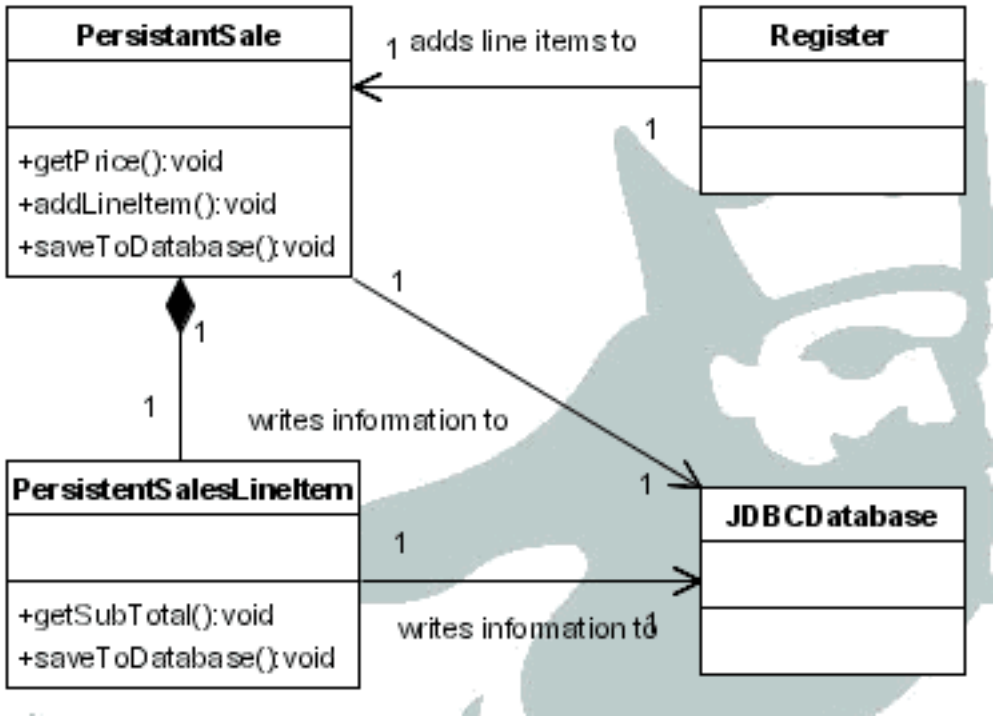
Problem: Which class do you allocate responsibility to when you are desperate but do not want to violate High Cohesion and Low Coupling?

Solution: Assign a highly cohesive set of responsibilities to an artificial class that does not represent anything in the problem domain – something made up in order to support high cohesion, low coupling and reuse.



Examples: Control Objects and Persistent Storage Broker Objects are both examples of this. These type of objects are pure fabrications that do not exist in the business/problem domain.

Based on "Applying UML and Patterns" by Craig Larman

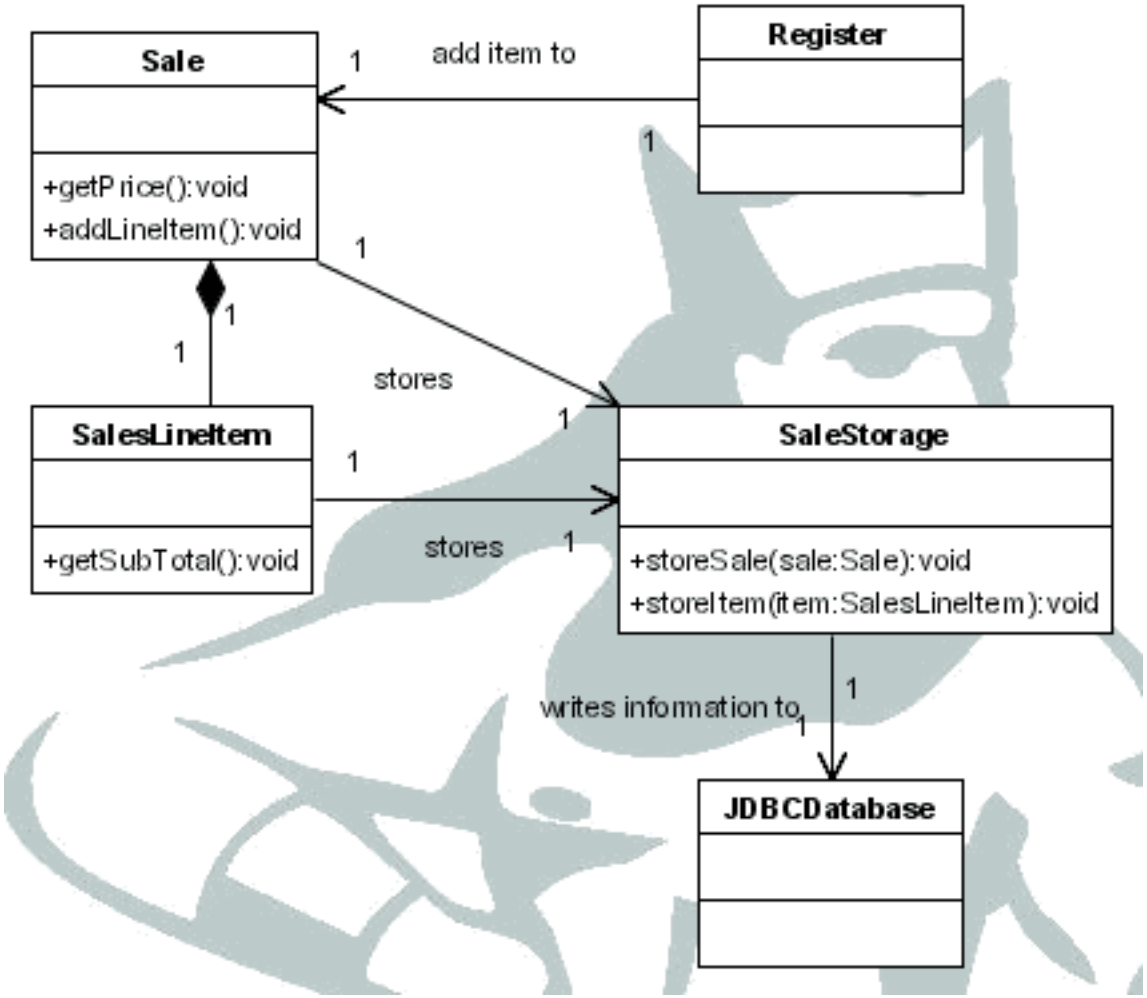


Who stores Sale information in a database?

Information Expert might suggest the Sale, that knows all information (price, items):

Bad Cohesion: Sale does two things!

http://www.christmann.ws/ucis342/class9/class9.html



Create a new class: SaleStorage:

Good Cohesion: Sale just knows "sale" stuff and the *Fabricated* class SalesStorage just puts Sale information into a database.

Coupling is no worse: Sale still talks to one other class, Register still talks just to Sale.

Coupling is better: Sale is indepenent of how it is stored (JDBC, SQLServer, etc.).

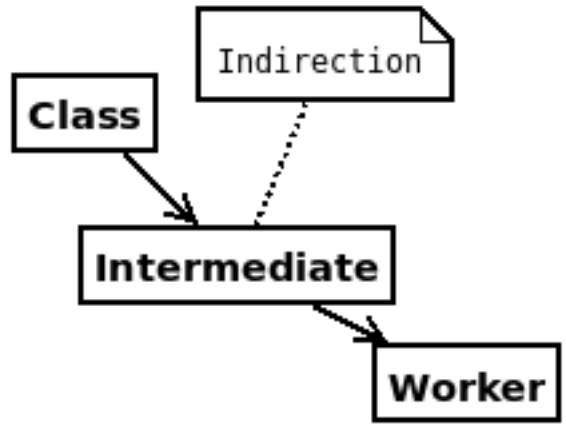
http://www.christmann.ws/ucis342/class9/class9.html

GRASP Pattern 8: **Indirection**

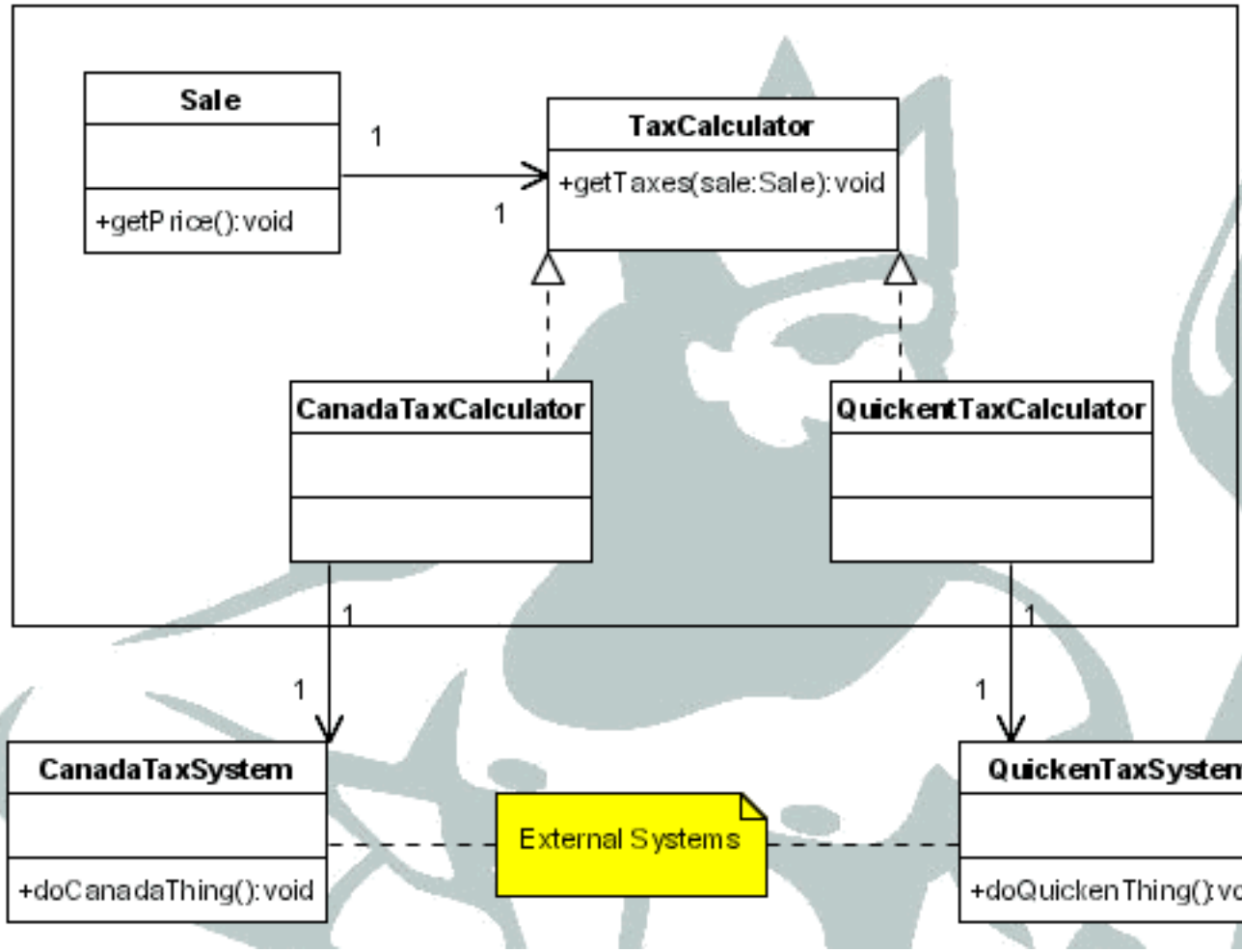
Problem: To which class do we assign responsibility in order to avoid direct coupling? How do we de-couple objects so that Low Coupling is supported and reuse potential remains.

Solution: Assign this responsibility to an intermediate object which mediates between other components or services so that they are not directly coupled.

Example. A Sale needs to communicate with external 3rd party tax systems through there own specific API's.



Based on "Applying UML and Patterns" by Craig Larman



Indirect access to an abstraction without knowing the precise concrete implementation of the abstraction (worker).

Polymorphism makes sure you get the right concrete behavior.

Example: GoF Adaptor, Proxy pattern covered later in the course

We create TaxCalculator interface with concrete implementations which represent the indirection classes, and reduces direct coupling between Sale and 3rd party API's for Canada and Quicken Tax System.

http://www.christmann.ws/ucis342/class9/class9.html

GRASP Pattern 9: Don't Talk to Strangers

Problem: To which class do we assign responsibility in order to avoid knowing about the existence and structure of indirect objects (strangers), and avoid highly coupled classes?

Solution: Assign responsibility to a client's direct object to collaborate with an indirect object, so that the client does not need to know about the indirect object.

- Avoids coupling a client to indirect objects
- knowledge of the internal structure of direct objects

Follow the "Law of Demeter"

Adapted from "Applying UML and Patterns" by Craig Larman

LAW OF DEMETER

"Who" is method X (in object A) allowed to talk to?

A method must only send messages to

1. *this* object (e.g. A itself)
2. An object sent to X in its argument list.
3. An attribute of its object A
4. An element of a "collection" attribute of its object A.
5. An object which X has created itself.

LAW OF DEMETER

Student
ccode year module [ ]
calcGrade(cwork,exm) etc etc

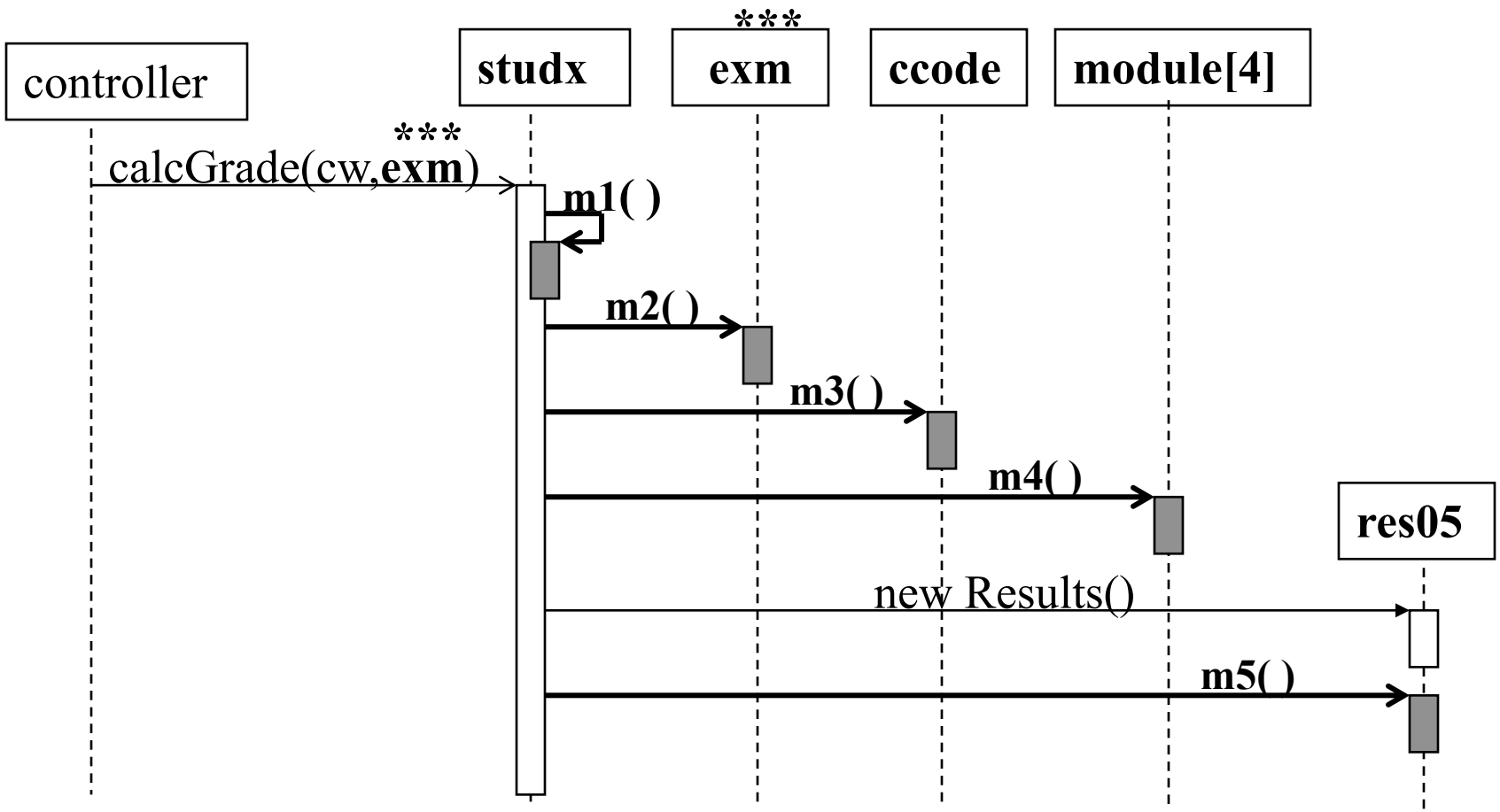
Object *stud* is of class *Student*.

Who can this method *calcGrade* talk to?

calcGrade may send messages to:

1. the object *stud* itself (e.g. use private method)
2. To objects *cwork* or *exm* (which were sent to *calcGrade* as arguments)
3. An attribute of *stud* (e.g. *ccode* or *year*)
4. Array element *module[3]* (for example) – from the collection attribute *module*.
5. Assume *calcGrade* creates a *Results* object called *res05.* It is then allowed to send further messages to *res05*.

Law of Demeter: allowed messages



Shows examples of the five types of allowed message

http://www.aspiringcraftsman.com/2008/06/distilling-law-of-demeter/

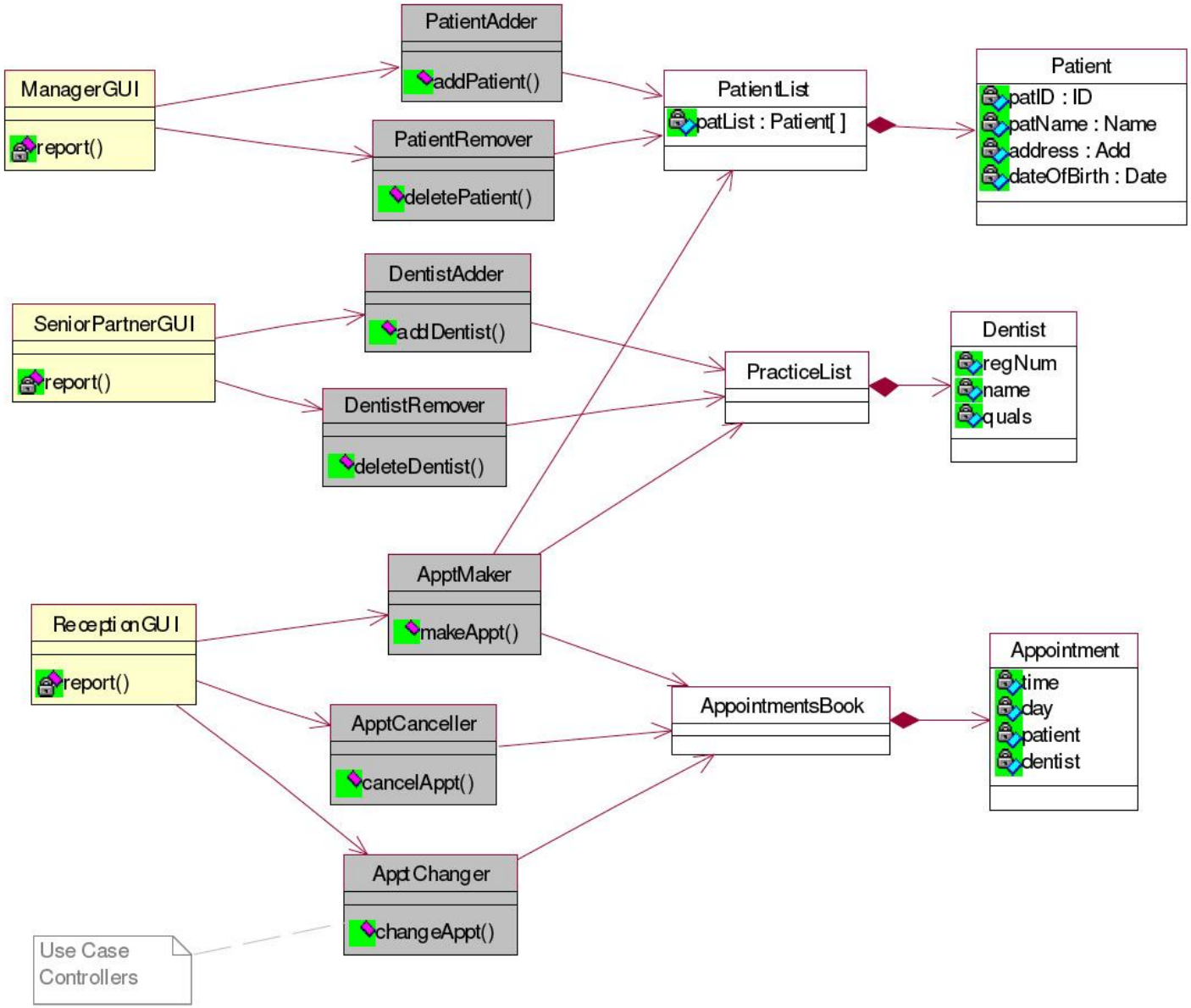
Towards a Better Design 4

None of the preceding designs is optimum.

We can improve the use-case controlled Model 3 as below:

- Apply the Creator pattern so PatientList is responsible for creating new Patient objects. *This fits the Creator Pattern because PatientList is an aggregation of Patient objects.*
- Apply the Expert pattern so that PatientList has overall responsibility for access to all Patient objects. *This fits the Expert Pattern because it has the information needed.*
- Similarly for the AppointmentBook & Appointment, PracticeList & Dentist

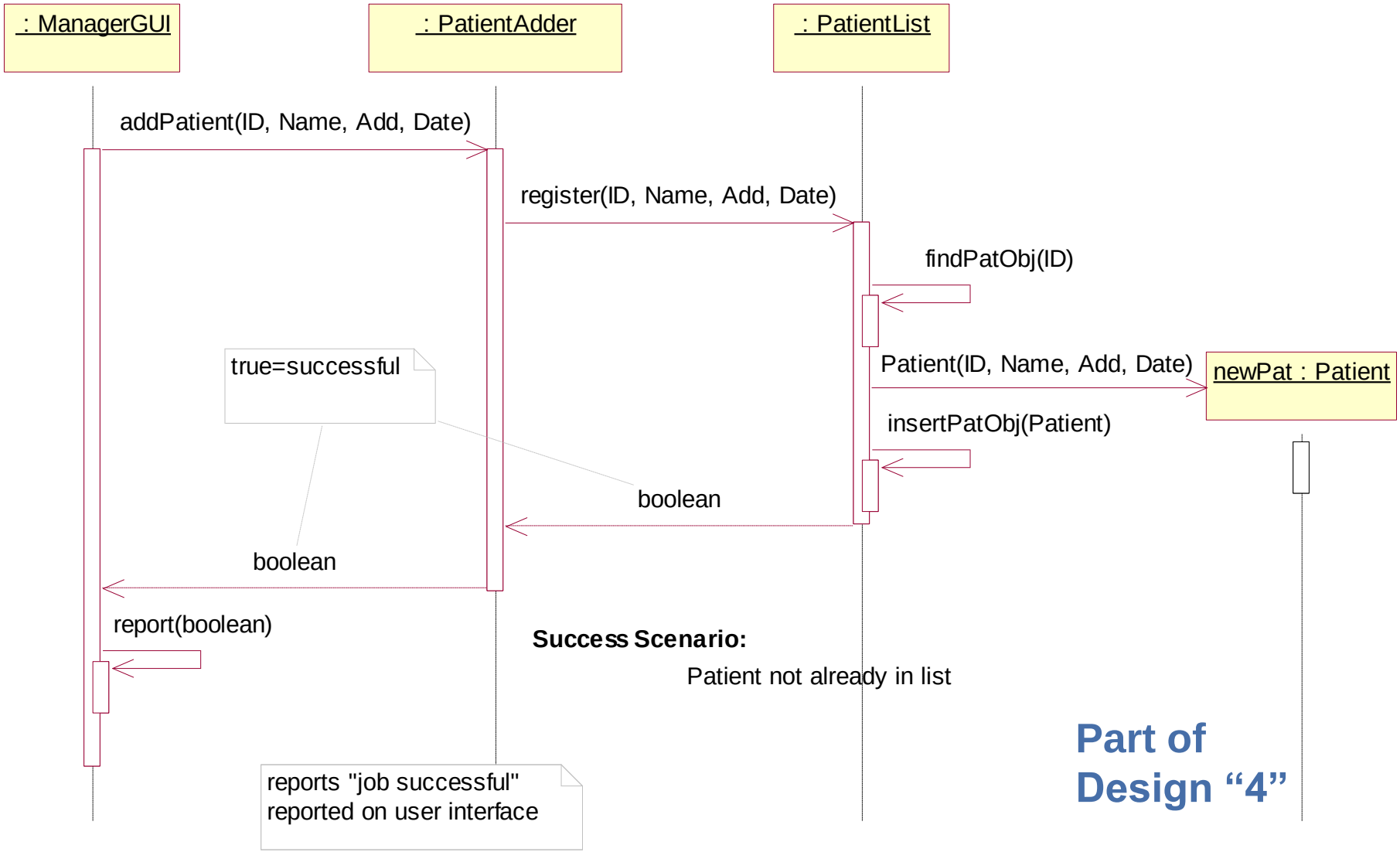
Design Model 4



Use Case Realisations

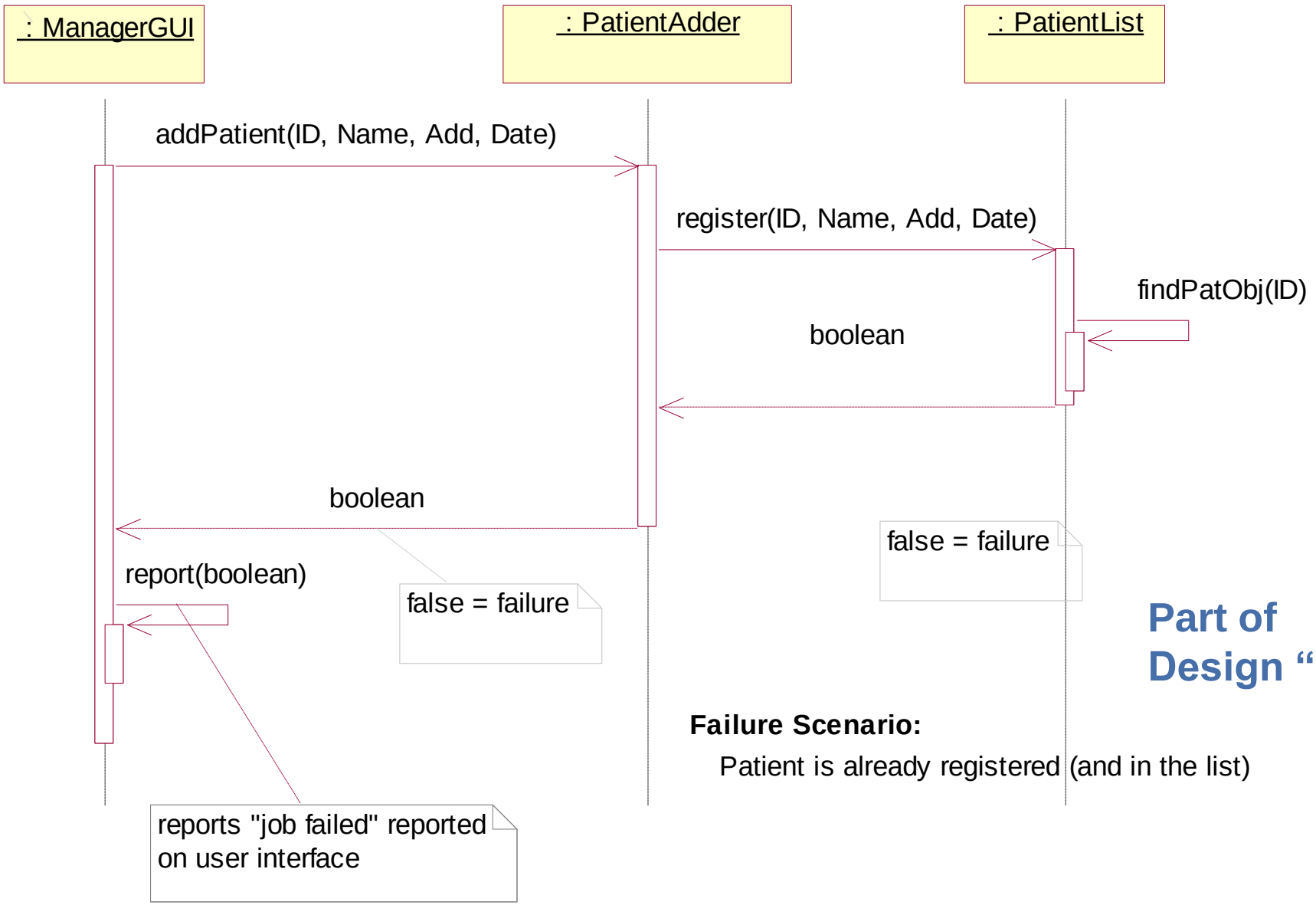
- This means producing detailed design solutions for for each scenario of each Use Case.
- A sequence diagram is produced for each scenario:
  - the main/typical scenario (e.g. success scenario)
  - the variant or conditional scenario (e.g. failure scenario)
- As we work through each scenario we have to decide:
  - which object sends messages to which object
  - which object is to be made responsible for which tasks
- This determines the methods each class must provide including
  - their arguments,
  - the types of those argument,
  - the return types of the methods

Sequence Diagram:
Add New Patient (Success Scenario)



Part of Design "4"

Sequence Diagram:
Add New Patient (Failure Scenario)

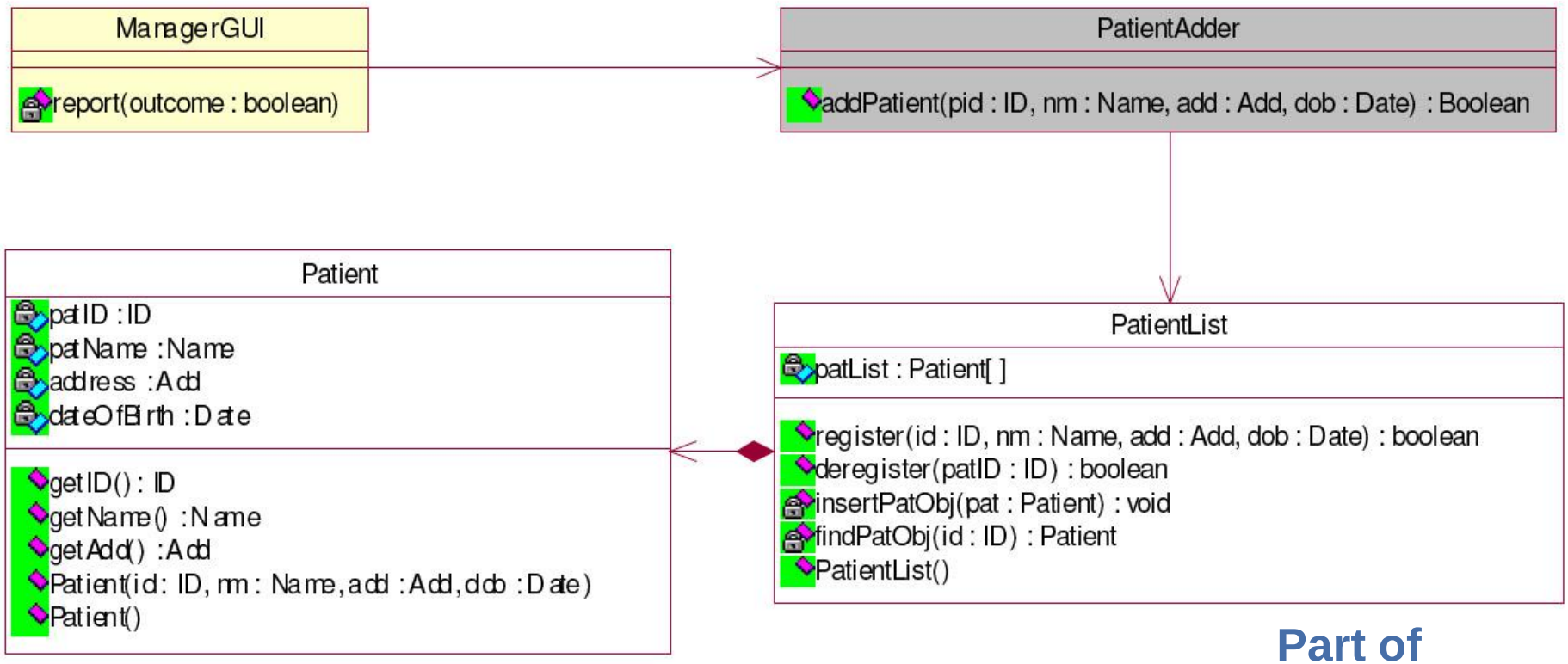


Part of Design "4"

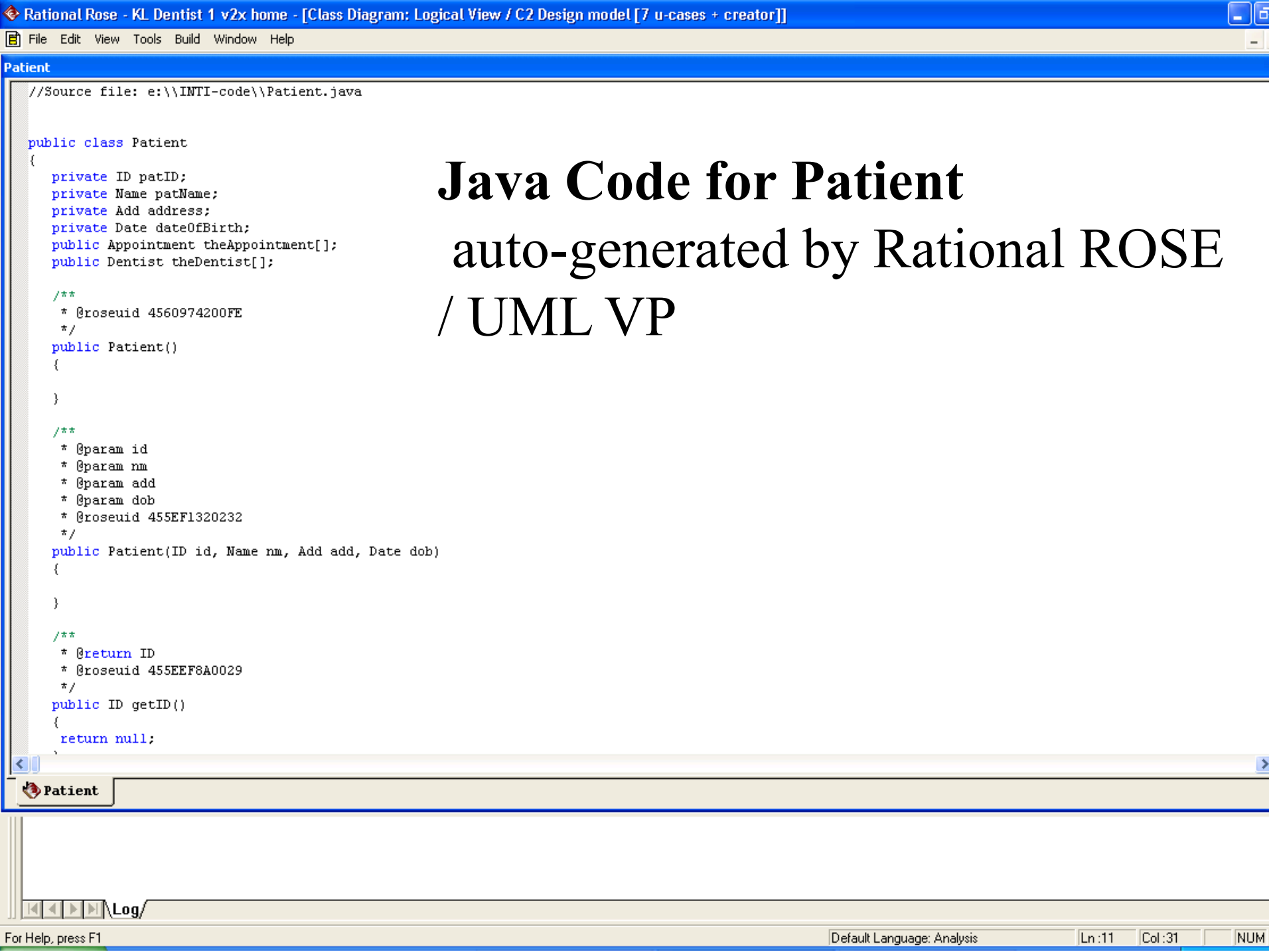
Failure Scenario:

Patient is already registered (and in the list)

Detailed Design of the Classes involved in "Add New Patient" Use Case

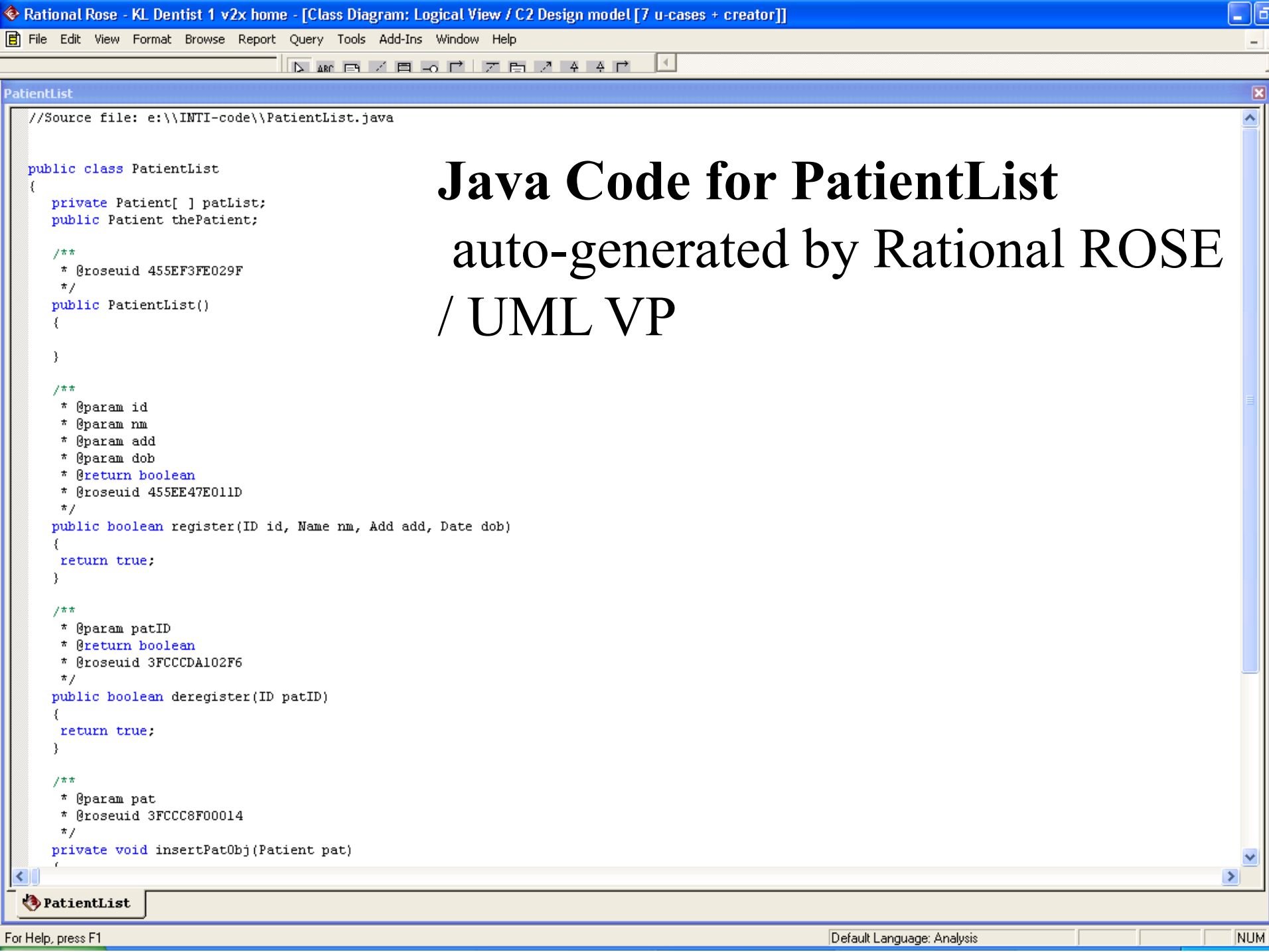


Part of Design "4"



Java Code for Patient

auto-generated by Rational ROSE / UML VP



Java Code for PatientList

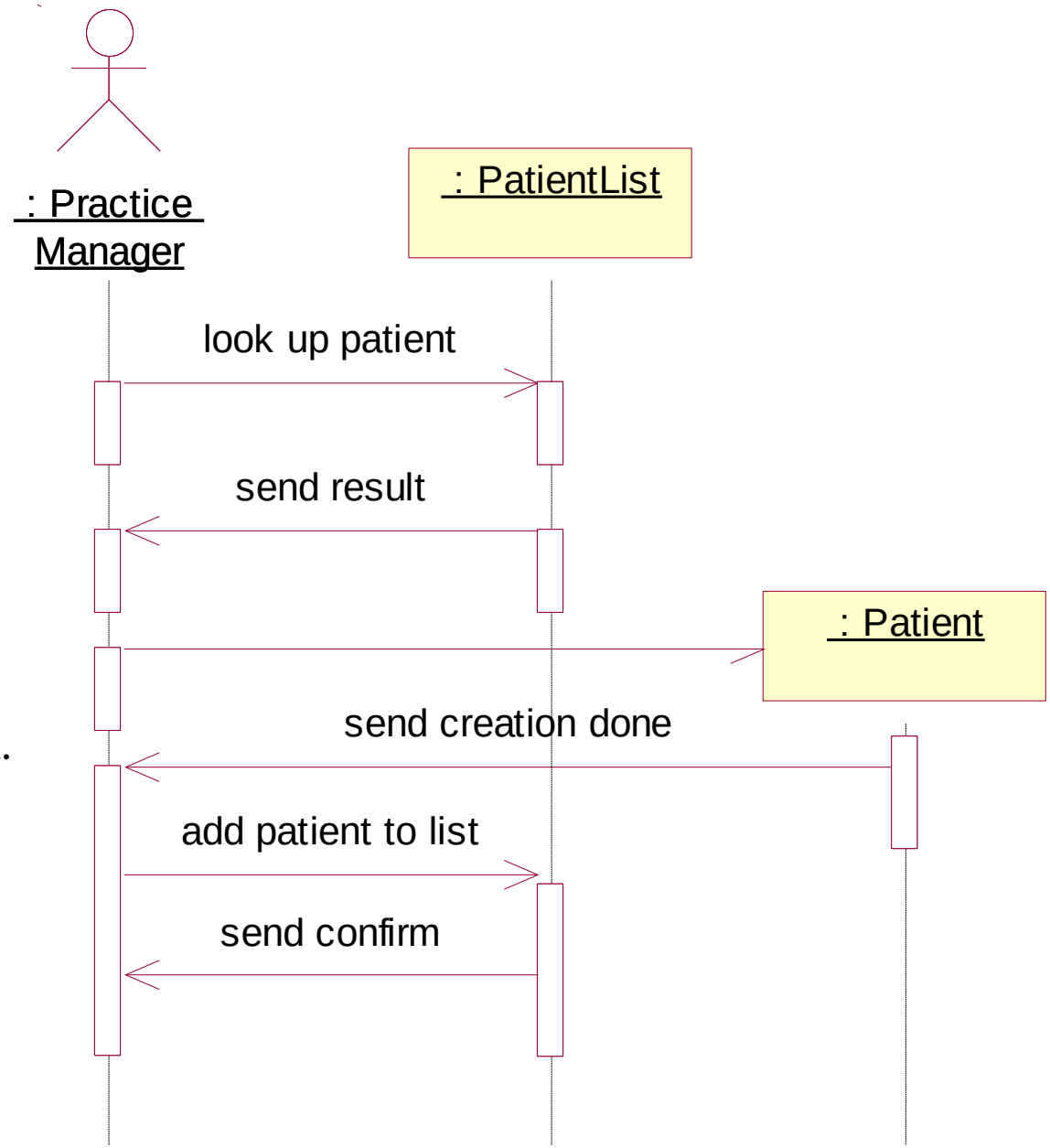
auto-generated by Rational ROSE / UML VP

PROBLEM DIAGRAM

Students often produce this type of informal diagram.

- Messages are informal text, not real methods.
- Return values shown as separate methods.
- No continuous "locus of control". Control reappears spontaneously after a break.
- Actors impossibly interact directly with data classes. No boundary classes or control classes

This diagram does not represent a software design.

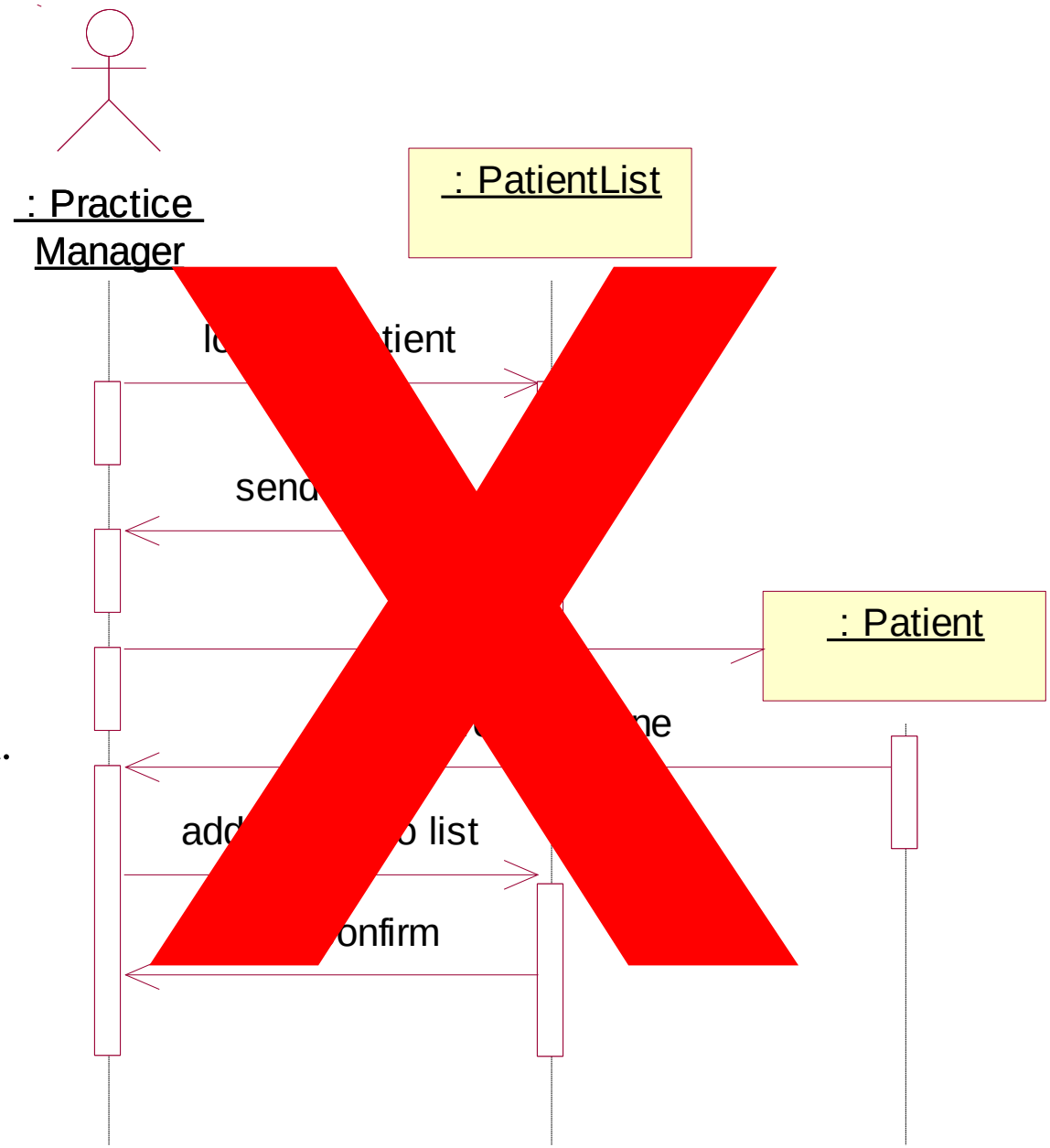


PROBLEM DIAGRAM

Students often produce this type of informal diagram.

- Messages are informal text, not real methods.
- Return values shown as separate methods.
- No continuous "locus of control". Control reappears spontaneously after a break.
- Actors impossibly interact directly with data classes. No boundary classes or control classes

This diagram does not represent a software design.



Some Exercises to try?

Produce Use Case realisations for other use cases and Update the detailed class designs to match

- delete patient
- make appointment
- change appointment
- cancel appointment

Key points [on Patterns]

- Design patterns are high-level abstractions that document successful design solutions. It is the design idea that is reused, not the code.
  - We still get faster software development at lower cost and with lower risk.
- Design patterns address common problems like the multiple observers problem. It often includes a UML model of the outline solution.
- Much more general patterns encapsulate conventional design guidelines such in the GRASP patterns.

Summary of OO Design Stage

- Build a static model of the design based on domain classes, boundary classes, control classes, data broker classes.
- Build a dynamic design model comprising use case realisations.
- Assign responsibilities to the classes based on
  - conventional software design guidelines (separation of concerns, coupling, cohesion, etc)
  - and/or the GRASP patterns.