UECS2344 Software Design: Lecture 4

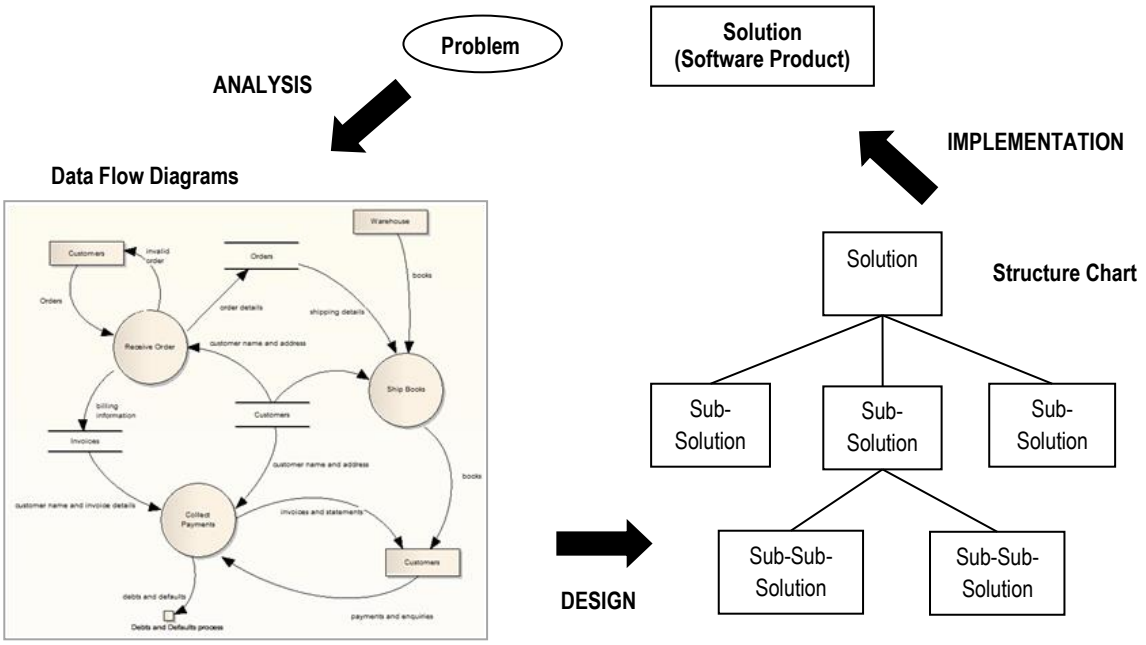
Software Systems Modelling

Modelling

Systems Development involves modelling i.e. building models of a system.

- a *model* is an a*bstraction* of a system.
- models are typically represented as diagrams

Structured Systems Development



Object-Oriented Systems Development

Adapted from Systems Analysis and Design: Object-Oriented Approach (Dennis, Wixom, Tegarden: Chapter 1

Important concepts of object-oriented systems development:

- class, object, attribute, behaviour (operation), message
- encapsulation, information hiding
- inheritance, superclass, subclass, polymorphism, dynamic binding

Unified Process and Unified Modeling Language (UML)

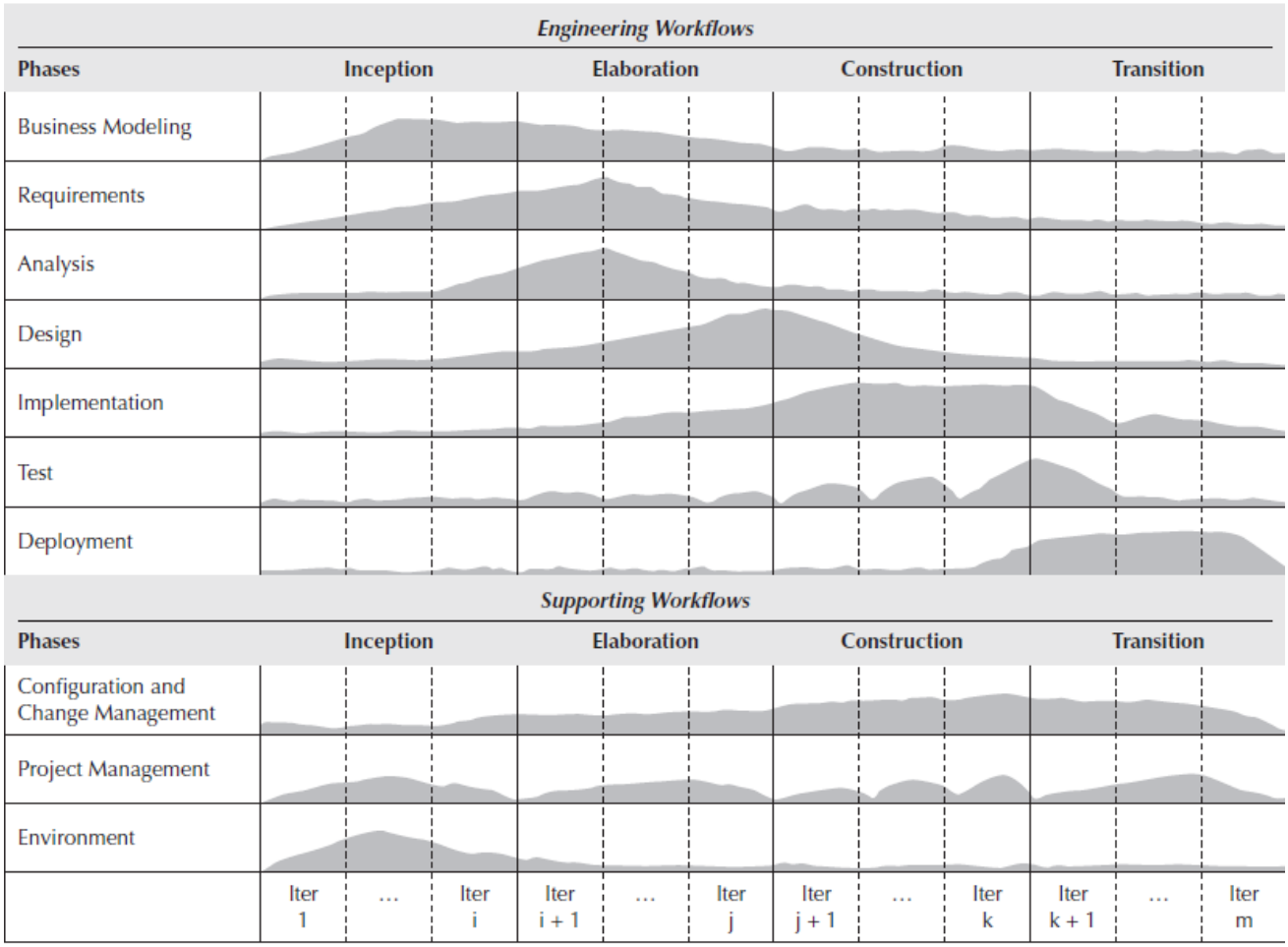
- primary contributors were Grady Booch, Ivar Jacobsen, and James Rumbaugh

Unified Process

- two-dimensional systems development process described by set of phases and workflows
    - phases: inception, elaboration, construction, and transition
    - workflows: business modeling, requirements, analysis, design, implementation, test, deployment, configuration and change management, project management, and environment
- use-case driven, architecture-centric, and iterative and incremental

Unified Process Phases

- the phases describe how an information system *evolves* through time
- the curve in the figure below associated with each workflow approximates the amount of activity that takes place during the specific phase
- depending on which phase the system is currently in, the level of activity varies over the workflows



Inception phase:

- focuses on business modeling, requirements, and analysis workflows
- primary deliverables are vision document that sets the scope of the project, identifies the primary requirements and constraints, sets up an initial project plan, and describes the feasibility of and risks associated with the project, and the adoption of the necessary environment to develop the system

Elaboration phase:

- focuses on analysis and design workflows
- deals with gathering the requirements, building the UML structural and behavioral models of the *problem domain*
- primary deliverables include UML structure and behavior diagrams and an executable of a baseline version of the information system, i.e. a partial implementation of the system that act as a foundation for remaining development

Construction phase:

- focuses on implementation workflow
- however, requirements workflow and analysis and design workflows also are involved; missing requirements are identified and the analysis and design models are finally completed
- primary deliverable is an implementation of the system that can be released for beta and acceptance testing

Transition phase:

- focuses on testing and deployment workflows
- business modeling, requirements, and analysis workflows should have been completed
- some activities that take place are beta and acceptance testing, fine-tuning the design and implementation, user training, and rolling out the final product onto a production platform
- depending on results from the testing workflow, some redesign and programming activities on the design and implementation workflows could be necessary, but they should be minimal at this point
- primary deliverable is the actual executable information system; other deliverables include user manuals, a plan to support the users, and a plan for upgrading the information system in the future

Workflows

- workflows describe tasks or activities that the software development team performs to evolve an information system over time
- they are grouped into two broad categories:
    - engineering workflows include business-modeling, requirements, analysis, design, implementation, test, and deployment workflows; deal with the activities that produce the technical product (i.e., the information system)
    - supporting workflows include the project management, configuration and change management, and environment workflow; focus on the managerial aspects of information systems development

Business Modeling Workflow:

- uncovers problems and identifies potential projects within the user organization
- aids management in understanding the scope of the project that can improve the efficiency and effectiveness of the user organization
- primary purpose is to ensure that both development team and user organization understand where and how the to-be-developed information system fits into the business processes of the user organization
- requirements gathering, and use-case and business process modeling techniques help to understand the business situation and develop the business model

Requirements Workflow:

- elicits both functional and nonfunctional requirements
- identified requirements form basis for developing use cases

Analysis Workflow:

- creates *analysis model* of the problem domain
- using the UML, the analyst creates structure and behavior diagrams that describe the problem domain classes and their interactions

- primary purpose is to ensure that both the development team and user organization understand the underlying problem and its domain

Design Workflow:

- transitions the analysis model into the *design model*
- focuses on developing a solution that will execute in a specific environment
- basically enhances the analysis model by adding classes related to the software environment
- uses activities such as detailed problem domain class design, optimization of the information system, database design, user-interface design, and physical architecture design

Implementation Workflow:

- primary purpose is to create an *executable solution* based on the design model
- in the case of multiple groups performing the implementation of the information system, the implementers must integrate the separate, individually tested modules to create an executable version of the system

Testing Workflow:

- primary purpose is to increase the quality of the system
- includes testing the integration of all modules used to implement the system, user acceptance testing, and the actual alpha testing of the software.
- practically speaking, testing should go on throughout the development of the system; basically, at the end of each iteration during the development of the information system, some type of test should be performed

Deployment Workflow:

- includes activities such as software packaging, distribution, installation, and beta testing
- might have to convert the current data, interface the new software with the existing software, and train the end user to use the new system

Unified Process is Use-Case Driven

- use-case driven means that *use cases* are the primary modeling tools defining the behavior of the system
- a use case describes how the user interacts with the system to perform some activity
- use case concept was created by Ivar Jacobson (one of the founders of UML)**.**
- use cases are used to identify and to communicate the requirements for the system to the development team who must write the system
- use cases are inherently simple because they focus on only one business process at a time; in contrast, the data flow diagrams used in traditional structured approach are far more complex because they require the systems analyst to develop models of the entire system; the system is decomposed into a set of subsystems, which are, in turn, decomposed into further subsystems, and so on; this is also known as step-wise refinement or top-down design

Unified Process is Architecture-Centric

- architecture-centric means that the underlying software architecture of the system specification drives the specification, construction, and documentation of the system
- should have at least three separate but interrelated architectural views of a system:
    - *functional,* or *external, view* describes the behavior of the system from the perspective of the user; modelled with use case diagram and use case descriptions
    - *structural,* or *static, view* describes the system in terms of attributes, methods, classes, and relationships; modelled with class diagrams

- *behavioral,* or *dynamic, view* describes the behavior of the system in terms of messages passed among objects and state changes within an object; modelled with sequence diagrams and behavioral state machine diagrams

Unified Process is Iterative and Incremental

- systems analysts develop their understanding of a user's problem by building up the three architectural views *little by little*
  - start by working with the user to create a functional representation of the system
  - next, build a structural representation of the system
  - using the structural representation, distribute the functionality of the system to create a behavioral representation of the system
- as an analyst works with the user in developing the three architectural views, the analyst iterates over each of and among the views
  - as the analyst better understands the structural and behavioral views, the analyst uncovers missing requirements or misrepresentations in the functional view
  - the functional view is changed which causes changes to the structural and behavioral views
  - all three architectural views are interlinked and dependent on each other; as each increment and iteration is completed, a more-complete representation of the user's real functional requirement is uncovered

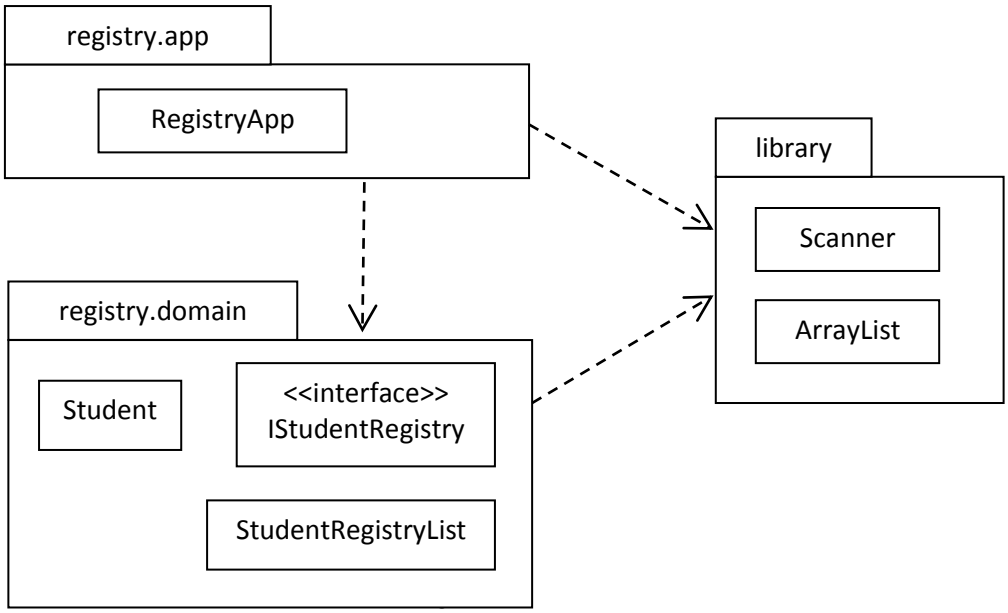
<u>Unified Modeling Language (UML)</u>

The UML has 13 diagram types of diagrams:

Diagram Name	**Used to ...**	**Primary Phase**
Structure Diagrams		
Class Diagram	**Illustrate the relationships between classes modeled in the system**	**Analysis, Design**
Object Diagram	**Illustrate the relationships between objects modeled in the system; used when actual instances of the classes will better communicate the model**	**Analysis, Design**
Package Diagram	**Group other UML elements together to form higher-level constructs**	**Analysis, Design, Implementation**
Deployment Diagram	**Show the physical architecture of the system; can also be used to show software components being deployed onto the physical architecture**	**Physical Design, Implementation**
Component Diagram	**Illustrate the physical relationships among the software components**	**Physical Design, Implementation**
Composite Structure Design Diagram	Illustrate the internal structure of a class, i.e., the relationships among the parts of a class	Analysis, Design

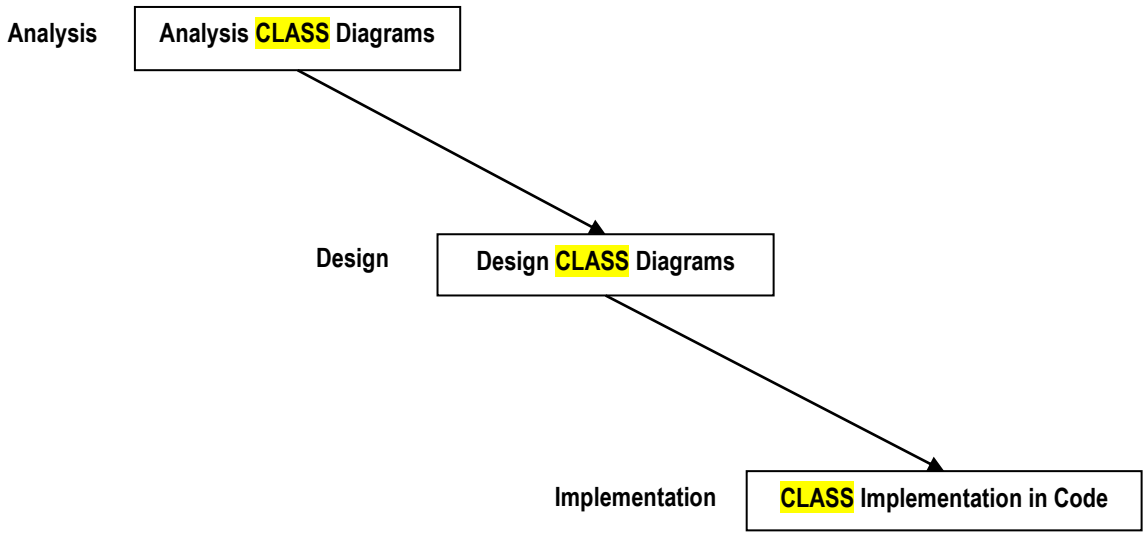
Behavioral Diagrams		
Activity Diagram	Illustrate business workfl ows independent of classes, the flow of activities in a use case, or detailed design of a method	Analysis, Design
Sequence Diagram	**Model the behavior of objects within a use case; focuses on the time-based ordering of an activity**	**Analysis, Design**
Communication Diagram	Model the behavior of objects within a use case; focus on the communication among a set of collaborating objects of an activity	Analysis, Design
Interaction Overview Diagram	Illustrate an overview of the flow of control of a process	Analysis, Design
Timing Diagram	Illustrate the interaction among a set of objects and the state changes they go through along a time axis	Analysis, Design
Behavioral State Machine Diagram	Examine the behavior of one class	Analysis, Design
Protocol State Machine Diagram	Illustrate the dependencies among the different interfaces of a class	Analysis, Design
Use-Case Diagram	**Capture business requirements for the system and illustrate the interaction between the system and its environment**	**Analysis**

Package Diagram Example – Refer to the Application in Practical 3



Advantage of Object-Oriented Approach over Structured Approach

Works with same thing during Analysis, Design, and Implementation i.e. **Classes**

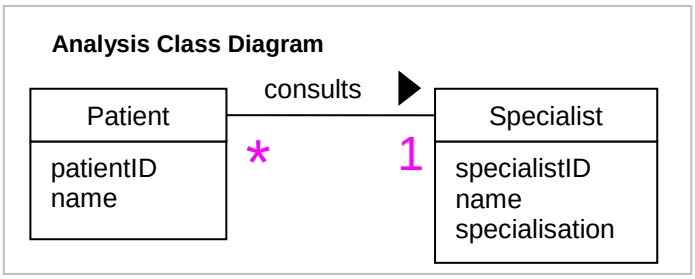


From Analysis Class Diagram to Design Class Diagram to Class Code

- Analysis Class Diagram (also called Problem Domain Model) – represents conceptual classes
- Design Class Diagram – represents software classes

Example

Analysis Class Diagram (with Multiplicity)



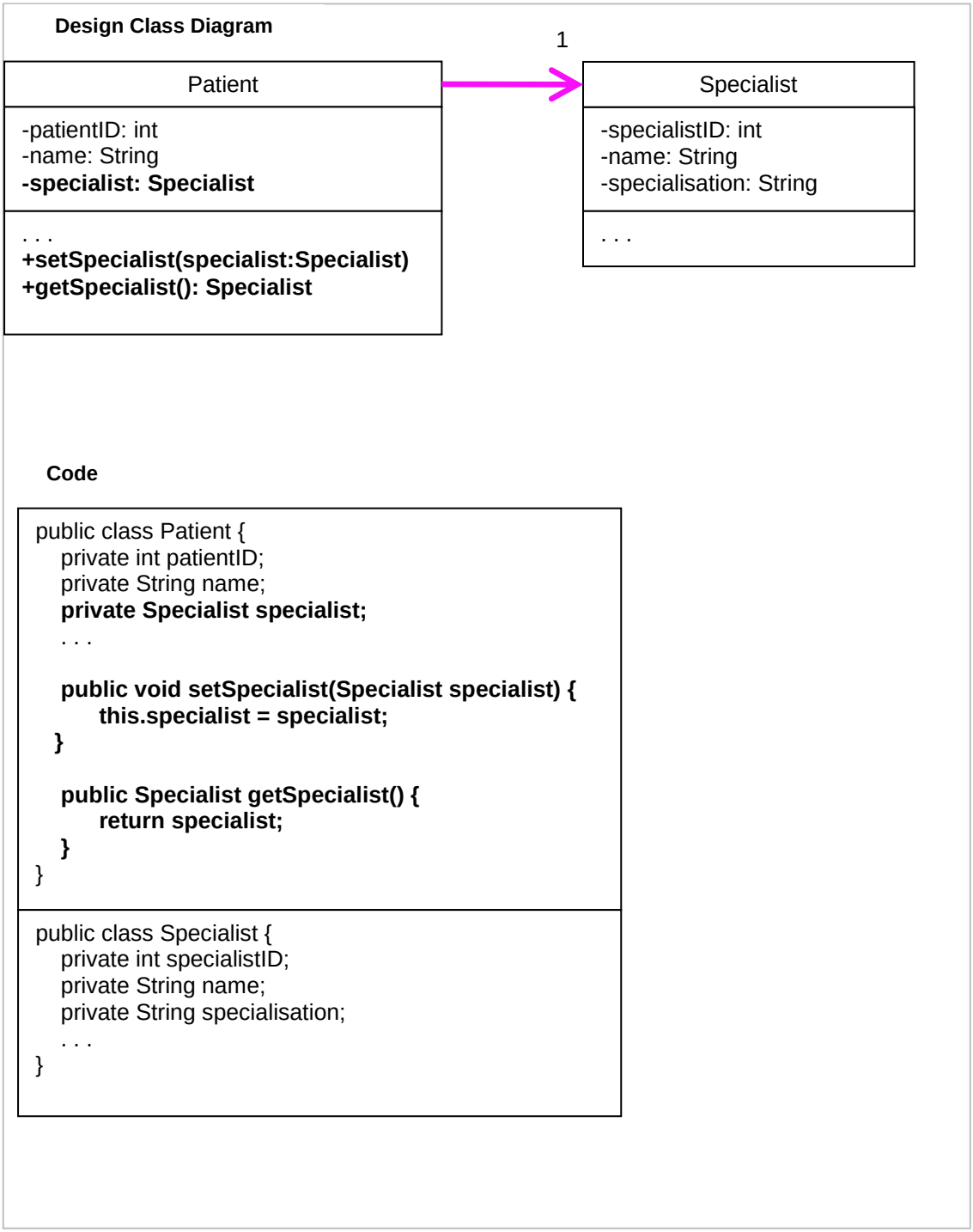
- **Multiplicity** is **one** on Specialist side (a Patient is attended by 1 Specialist only).
- **Multiplicity** is **many** on Patient side (a Specialist attends to many Patients).

Design Class Diagram (with Navigability)

Case 1: It is ***necessary*** to know, for each patient, which specialist the patient consults ***BUT*** it is ***not necessary*** to know, for each specialist, which patients the specialist gives consultation to.

Solution: Navigability is UNI-DIRECTIONAL: from Patient object to Specialist object only.

Design Class Diagram



Code

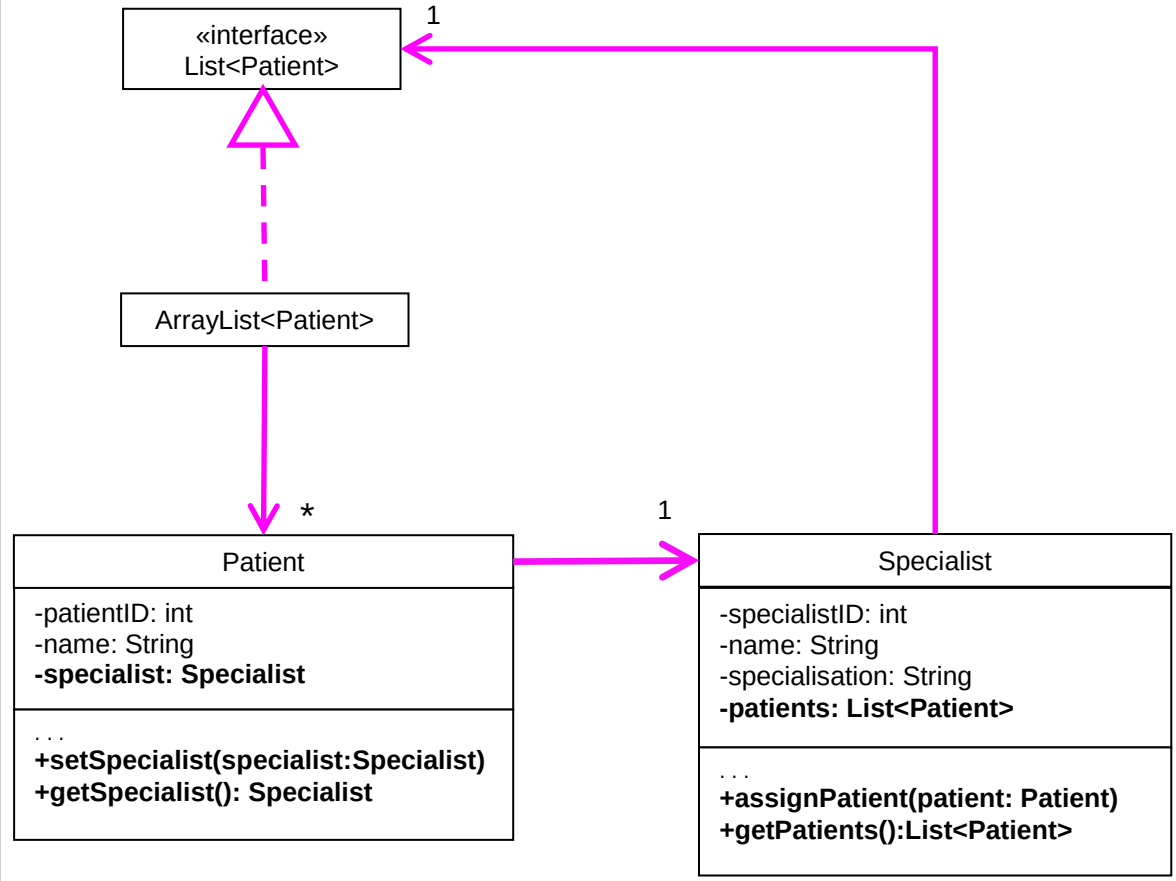
```
public class Patient {
    private int patientID;
    private String name;
    private Specialist specialist;
    . . .

    public void setSpecialist(Specialist specialist) {
        this.specialist = specialist;
    }

    public Specialist getSpecialist() {
        return specialist;
    }
}
```

```
public class Specialist {
    private int specialistID;
    private String name;
    private String specialisation;
    . . .
}
```

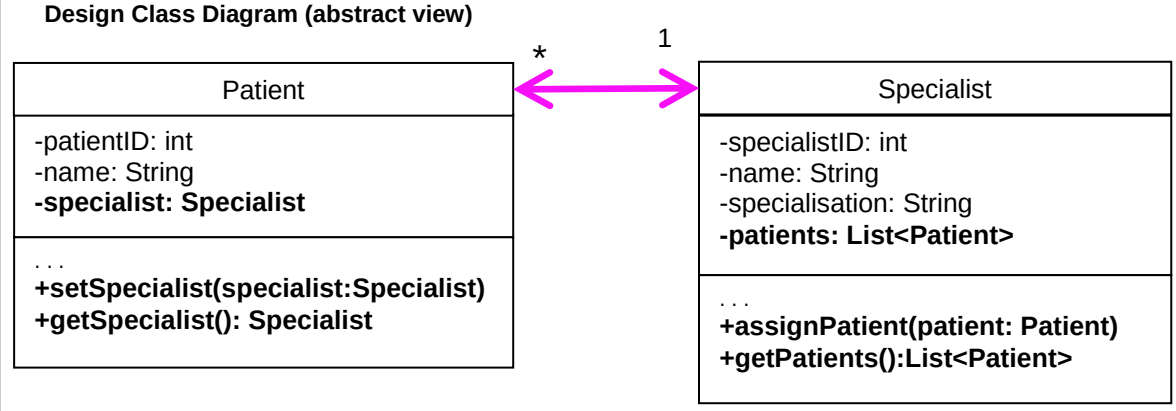
Case 2: It is ***necessary*** to know, for each patient, which specialist the patient consults ***AND*** it is ***necessary*** to know, for each specialist, which patients the specialist gives consultation to.

Solution: **Navigability is <u>BI-DIRECTIONAL</u>: from Patient object to Specialist object AND from Specialist object to Patient object.**

Design Class Diagram (detailed view)



Design Class Diagram (abstract view)



Code

```
public class Patient {
    private int patientID;
    private String name;
    private Specialist specialist;
     . . .

   public void setSpecialist(Specialist specialist) {
        this.specialist = specialist;
    }

   public Specialist getSpecialist() {
       return specialist;
   }
}
```

```
public class Specialist {
    private int specialistID;
    private String name;
    private String specialisation;
    private List<Patient> patients;

    public Specialist(int id, String name, String specialisation) {
        specialistID = id;
        this.name = name;
        this.specialisation = specialisation;
        patients = new ArrayList<Patient>();
    }
    . . .

    public void assignPatient(Patient  patient) {
        patients.add(patient);
    }

    public List<Patient> getPatients() {
        return patients;
    }
}
```

Exercise 1

Given the partial Java code for a Monopoly Game Application, produce a Design Class Diagram with navigability and multiplicity (ignore the operations)

Monopoly Game Application

Class Square

```
public class Square {
	private String name;
	private Square nextSquare;
	private int index;

	public Square(String name, int index) {
		this.name = name;
		this.index = index;
	}

	public void setNextSquare(Square s) {
		nextSquare = s;
	}

	public Square getNextSquare() {
		return nextSquare;
	}

	public String getName() {
		return name;
	}

	public int getIndex() {
		return index;
	}
}
```

Class Piece

```
public class Piece {
	private Square location;

	public Piece(Square location) {
		setLocation(location);
	}

	public Square getLocation() {
		return location;
	}

	public void setLocation(Square location) {
		this.location = location;
	}
}
```

Class Die

```
public class Die {
	 public static final int MAX_VALUE = 6;
	private int faceValue;
	. . .
}
```

Class Board

```
import java.util.ArrayList;
import java.util.List;

public class Board {
	public static final int SIZE = 40;
	private List<Square> squares;

	public Board() {
		squares = new ArrayList<Square>(SIZE);
		. . .
	}

	. . .
}
```

Class Player

```
public class Player {
	private String name;
	private Piece piece;
	private Board board;
	private Die[] dice;

	public Player(String name, Die[] dice, Board board) {
		this.name = name;
		this.dice = dice;
		this.board = board;
		piece = new Piece(board.getStartSquare());
	}

	. . .
}
```

Class MonopolyGame

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class MonopolyGame {
	private List<Player> players;
	private Board board;
	private Die[] dice;

	public MonopolyGame() {
		board = new Board();
		dice = new Die[2];
		dice[0] = new Die();
		dice[1] = new Die();
		players = new ArrayList<Player>();

		. . .
	}

	. . .
}
```

Exercise 2

Given the partial Java code for a Point-Of-Sale Application, produce a Design Class Diagram with navigability and multiplicity (ignore the operations).

Point-Of-Sale Application

Class ItemId

```
public class ItemId {
	private int id;

	. . .
}
```

Class ProductDescription

```
public class ProductDescription {
	private ItemId id;
	private double price;
	private String description;

	. . .
}
```

Class ProductCatalog

```
import java.util.HashMap;
import java.util.Map;

public class ProductCatalog {
	private Map<ItemId, ProductDescription> descriptions;

	public ProductCatalog() {
		descriptions = new HashMap<ItemId,ProductDescription>();

		//sample data
		ItemId id1 = new ItemId(100);
		ItemId id2 = new ItemId(200);

		ProductDescription desc1 =
				new ProductDescription(id1, 3.00, "product 1");
		ProductDescription desc2 =
				new ProductDescription(id2, 5.00, "product 2");

		descriptions.put(id1, desc1);
		descriptions.put(id2, desc2);
	}

	public ProductDescription getProductDescription(ItemId id) {
		return descriptions.get(id);
	}
}
```

Class SaleLineItem

```
public class SaleLineItem {
	private ProductDescription product;
	private int quantity;

	. . .
}
```

Class Payment

```
public class Payment {
	private double amount;

	. . .
}
```

Class Sale

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class Sale {
	private List<SaleLineItem> lineItems;
	private boolean isComplete;
	private Date date;
	private Payment payment;

	public Sale() {
		lineItems = new ArrayList<SaleLineItem>();
		isComplete = false;
		date = new Date();
		payment = null;
	}

	. . .
}
```

Class CashRegister

```
public class CashRegister {
	private ProductCatalog catalog;
	private Sale currentSale;

	. . .
}
```

Class Store

```
public class Store {
	private ProductCatalog catalog;
	private CashRegister register;

	public Store() {
		catalog = new ProductCatalog();
		register = new CashRegister(catalog);
	}

	public CashRegister getRegister() {
		return register;
	}
}
```