Object Oriented Analysis & Design

Week 7

Class and Object Diagram

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 6

- Interaction diagrams describe how use cases are realized as interactions among societies of objects.
- Sequence diagrams are a temporal representation of objects and their interactions. This means that they represent objects and their interactions over a time element.
- Collaboration and communication diagrams are spatial representation of objects, links and interrelations. This means that they are not concerned with time; rather they are concerned with how objects interact in the greater dimension of space.

Content

- Introduction
- Structural Model
- Object Identification
- Class Diagram Elements
- Object Diagrams

Part 1

Introduction

Introduction

- In lesson 6 the realization of the use case diagram through interaction diagrams was discussed.
- As this is the halfway point of this course it is important to recap on a few concepts. The purpose of the recap is to remind ourselves how far we have progressed in terms of the overall object oriented analysis and design phase of the SDLC.
- In this introduction we purpose to do exactly that before indulging in the core lesson. The lesson begins with an overall recap of the OO analysis and design phase.
- This is followed by an introduction to the structural model, and how it relates to the dynamic and static models.
- Thereafter a discussion on the details of the core of the lesson follows, namely:
  - Class diagrams
  - Object diagrams

The OOAD Phases

- The OOAD phases can be described in four simple steps.
- By examining these four steps it is easy to see from an aerial perspective the progress made at this point in the course.
- Fig 1 describes the steps and how they relate to the lessons covered hitherto in this course.
- Phase 1 is the use case phase – this was covered in lesson 5 of the course. We defined our use cases and developed the use case diagram.
- Phase 2 is the define domain model phase – As Larman (2002) puts it: "A decomposition of the domain involves an identification of the concepts, attributes, and associations that are considered noteworthy. The result can be expressed in a **domain model,** which is illustrated in a set of diagrams that show domain concepts or objects.
- Phase 3 is the define interaction diagrams phase – this was covered in lesson 6 of this course; a description of the elements and interactions via interaction diagrams was discussed in details. Some example problems and solutions were also provided in the lesson.
- Phase 4 is the define class diagrams – the topic of this lesson.

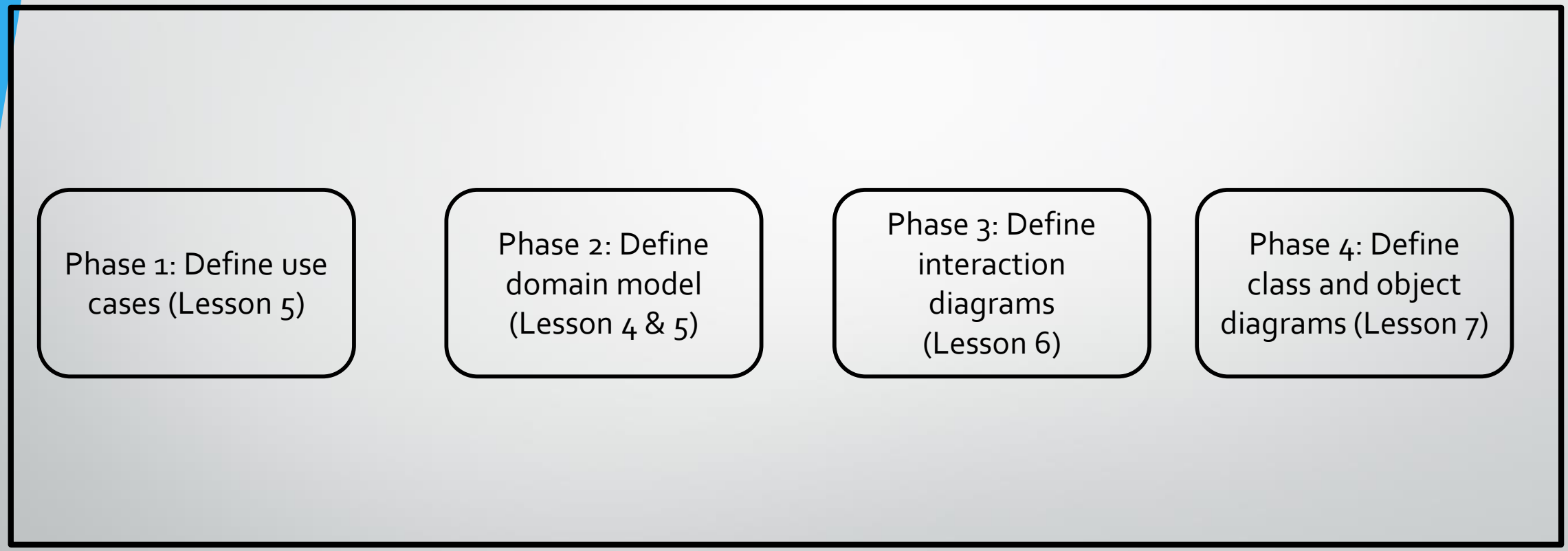

Fig 1 OOAD phases (adapted from Larman, 2002)

Part 2

The structural model

Introduction

- One of the primary purposes of the structural model is to create a vocabulary that can be used by the analyst and the users. Structural models represent the things, ideas, or concepts contained in the domain of the problem. They also allow the representation of the relationships among the things, ideas, or concepts. By creating a structural model of the problem domain, the analyst creates the vocabulary necessary for the analyst and users to communicate effectively. (Dennis et al., 2015)
- The structural model is key in analysis since it enables both the system analysts and users to visualize and understand the problem domain without any technical implementation details.
- The structural model is described using two other models: the static model and the dynamic model.
- The dynamic model is concerned with the time and ordering aspects of the system being designed. It is covered in a separate lesson.
- Nonetheless O'Docherty (2005) describes dynamic analysis of the system as follows:

Introduction (cont'd)

- O'Docherty (2005) avers that dynamic analysis is performed for the following reasons:
- "To confirm that our class diagram is complete and accurate, so that we can fix it sooner rather than later: this may involve adding, deleting or modifying classes, relationships, attributes and operations.
- To gain confidence that our modeling up to this point can be implemented in software: we're not the only ones that should be confident before we proceed, our sponsors are just as important.
- To verify the functionality of the user interfaces that will appear in the final system: it's a good idea to partition access to the system into separate interfaces, along use case lines, before we dive into detailed design."
- Recall that in earlier descriptions of object oriented analysis and design it was stated that analysis is concerned with what the system should do, while design is concerned with how the system will do it.

2.1 Static analysis

- Static analysis is concerned with what the system should do.
- In doing so we differentiate this with "how" the system should do it. This is done through the static model which is a design implementation of static analysis.
- O'Docherty (2005) suggests a series of steps that should be followed in performing a static analysis:
- Find classes – in this step the analyst will use known methods to identify the classes in the problem domain. A good place to start is to simplify underline the nouns since these are the most likely candidates for classes. Other ways of doing this are described later.
- Identifying class relationships – in this step the analyst will attempt to draw out the relationships between the different candidate classes. This is not a simple exercise; however, at this stage the aim is to just establish that there exists a relationship between the classes. The "how" of the relationship will be detailed in the static model. Relationships include association, aggregation, composition, inheritance (generalization). These relationships can be refined further as described in part 4 of this lesson.

2.1 Static analysis (cont'd)

- Draw class and object diagram – the class diagram will show the final identified classes and how they are related to one another; refinements can also be shown. However, we already know that an object is an instance of a class and that a class is a blueprint (template) that is used for instantiating objects. When we wish to use objects rather than classes an object diagram is used. The notation for the object is same as for the sequence diagram; difference being that object diagrams don't specify the relationship between the objects.
- Draw relationships and refinements – show the different relationships together with any refinements such as multiplicity, labels and comments
- Show attributes – show the applicable attributes of each class
- Show association classes – show the association classes where applicable in the class diagram.
- Differentiate between tangible and intangible objects – this refers to differentiating between intangible objects (normally theoretical, such as items in a list or catalog), and their realized tangible objects.
- Fig 2 captures the necessary components of static analysis

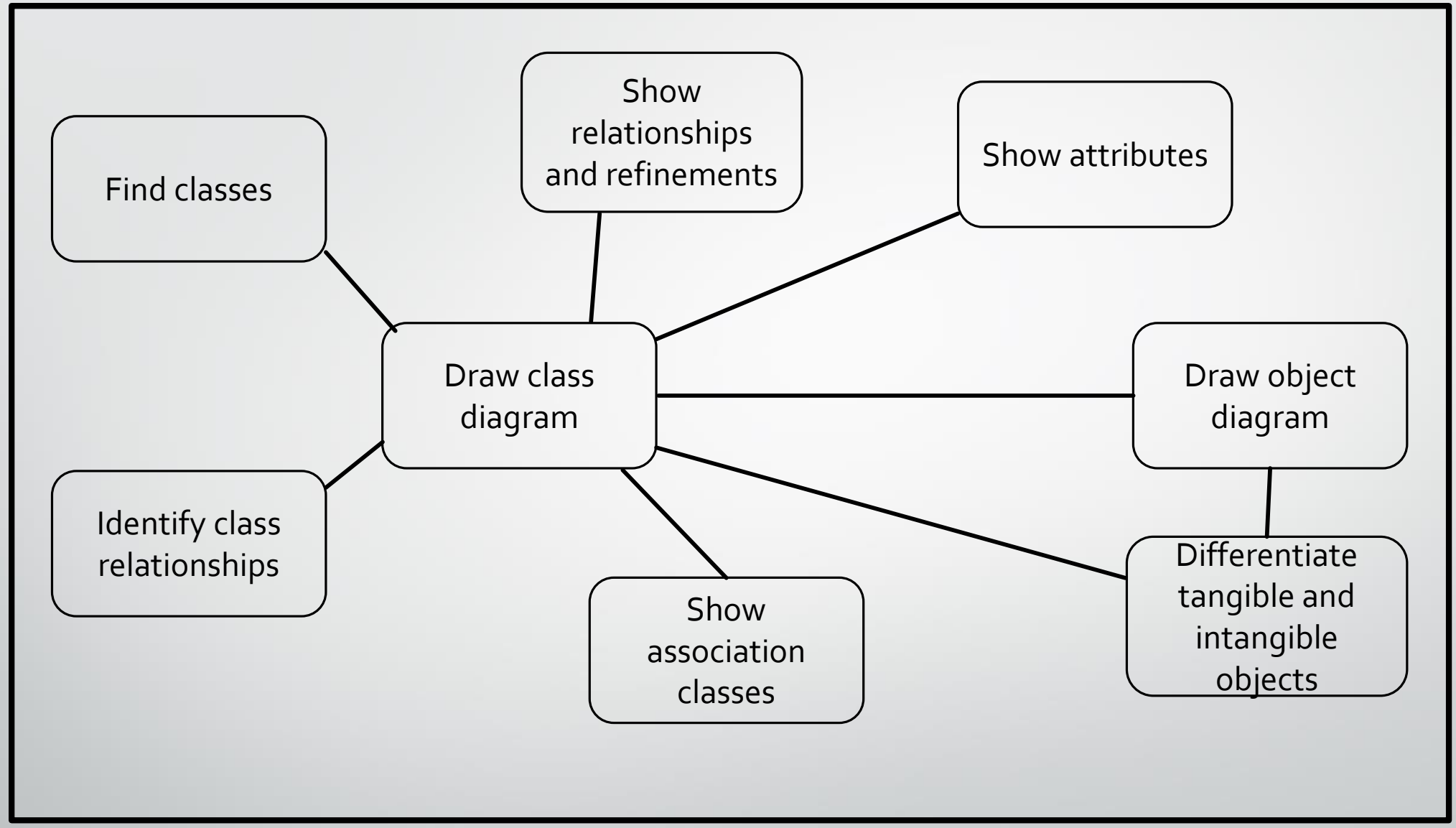

Fig 2. Components of static analysis

Part 3

Object identification

Introduction

- One of the foremost challenges in designing a class or object diagram is to identify the classes and objects themselves.
- Whereas there are different types of classes (these will be described later in the lesson) the general definition of a class is the same across the board.
- There are some methods that are used in helping to identify the classes and objects, and these are described in this part.

3.1 Textual analysis

There are a number of well known and easy to use techniques including:-

- Textual analysis
- Finding "uses"
- Finding players
- Finding generalization
- Finding analogies
- Finding reuse
- Brainstorming
- Using RUP stereotypes

3.1.1 Textual analysis

Part of speech	Example	Object-Oriented Analysis component
noun	**tree**	**object type, object, attribute**
doing verb	grows	Service
being verb	is a (plant)	Gen-Spec Structure
having verb	has (leaves)	Whole-Part Structure
transitive verb *(i.e. one which takes an object)*	provides (timber)	service
intransitive verb *(i.e. one which doesn't take an object)*	falls	exception or time dependent event
adjective	evergreen	attribute value

Table 1. Textual analysis

3.1.1 Textual analysis (cont'd)

- Textual analysis has been used in different contexts to identify the classes in the problem domain.
- The place to start from is the use cases and their descriptions. This will help the analyst to draw out information regarding different aspects of the problem domain.
- Table 1 presents an example of how to perform this analysis.
- Nouns will normally represent either objects, object types or attributes; similarly different types of verbs suggest other aspects of the problem domain. For example, "having" verbs suggest a whole-part structure (as shall be seen later in the lesson).
- This information allows us to present a comprehensive textual analysis as follows.

3.1.1 Textual analysis (cont'd)

- Once all the nouns have been highlighted:-
  - Highlight noun phrases;
  - convert plurals to singulars;
  - discard obvious nonsense;
  - remove synonyms – words that mean the same thing
    - e.g. Warehouse and Stock room could refer to the same location – but check first;
  - beware adjectives – words that describe other things
    - E.g. color, size

3.1.1 Textual analysis (cont'd)

Based on the foregoing analysis, the likely classes and objects are:

- physical objects;
- "cohesive abstractions" (such as a file);
- interfaces to the outside world;
- any categories.

3.1.1. Textual analysis (cont'd)

- Dennis et al. (2015) also suggest using the use cases to find the objects and classes. They present the following summary (adapted from Russell, 1983):
  - A common or improper noun implies a class of objects.
  - A proper noun or direct reference implies an instance of a class.
  - A collective noun implies a class of objects made up of groups of instances of another class.
  - An adjective implies an attribute of an object.
  - A doing verb implies an operation.
  - A being verb implies a classification relationship between an object and its class.
  - A having verb implies an aggregation or association relationship.
  - A transitive verb implies an operation.
  - An intransitive verb implies an exception.
  - A predicate or descriptive verb phrase implies an operation.
  - An adverb implies an attribute of a relationship or an operation.

3.1.2. Finding uses

- This method suggests picking out the candidate objects and classes and determining their use in the problem domain. This is done by asking questions such as:
- “what is it used for?”
- “what is its purpose?”
- The logic here is that a class or object should play some important role in the problem domain.
- Certainly just listing data and/or the functionality won't make it an object

3.1.3. Identifying players (actors)

With this method we look for "actors" in the problem domain:

- people or organizations that play some role in our problem domain.

  (beware of the possibility that one person plays several different roles.)

- places, in particular places where things are stored or contained. (these will imply storage places in the system such as a database object).
- Further we can also make use of event remembered objects. These are normally transactions, which the system needs to remember or record.
- Examples of these include such things as contract, delivery, deposit, loan, reservation, sale and order.

3.1.4. Finding generalization

- This method implements the powerful feature of OO called inheritance covered in lesson 2:
- Look for classes or objects that share common features (i.e. attributes, methods, and relationships)
- Take out the shared features to form a super-type and then use inheritance to form simpler subtypes to replace the original classes.

3.1.5. Finding analogies

- Ask yourself "What is the purpose of this system?"
- Once you have determined that:
  - Try and find a system that has an analogous purpose
  - Use the purpose you have identified and generalize it
- Then:
  - Examine the object types in the analogous systems and identify any that might correspond via the analogy to ones that you need
  - Add corresponding object types to your model.

3.1.6. Finding reuse

- Sometimes a previous domain-specific OO analysis has been done which can provide a framework of re-useable object types
  - with all their associated attributes, methods and relationships
- As they are domain-specific they will apply to any other application in that domain.
- If this is so:
  - Choose the relevant framework(s) for the application domain.
  - Incorporate classes (and objects) from the framework(s) into the problem domain model.

3.1.7 Brainstorming

- This discovery technique involves selecting a group of people familiar with the problem domain.
- The group then sits together and suggests classes that can be useful in this problem domain. Bellin and Simone (1997, as cited by Dennis et al., 2015) suggest the following principles to guide a brainstorming session:
  - All suggestions should be taken seriously (do not leave anything out since you "think" it might not be useful)
  - Participants should begin by jotting down all ideas quickly (without pondering over them); once this has been done then the pondering can begin.
  - The facilitator must manage the fast and furious thinking (the sharing of ideas before pondering over them). S/he can do this in a variety of ways, such as encouraging all participants to give suggestions, or using an anonymous electronic tool to encourage participation, and so on).

3.1.8. Using RUP stereotypes

- Arlow and Neustadt (2013) suggest the use of RUP stereotypes; it involves considering 3 distinct types of analysis classes:

1. Boundary – this is a class that mediates interaction between the system and its environment. Specifically these are the classes that communicate with external actors and are therefore at the "boundary" of the system. These classes are found by considering what classes mediate between the system and its environment. According to the RUP there are 3 types of boundary classes:
   - User interface classes (between system and humans; thus if the actor represents a human it indicates a user interface class)
   - System interface classes (between system and other systems; thus if the actor represents a system it indicates a system interface class)
   - Device interface classes (between system and external devices, such as sensors; thus if the actor represents a device it indicates a device interface class)

3.1.8. Using RUP stereotypes (cont'd)

2. Control – this is a class that encapsulates use case behavior. These classes coordinate system specific behavior that corresponds to one or more use cases. They are found by considering the behavior of the system as described by the use case and working out how that behavior should be partitioned among the classes. Simple behavior can be distributed between boundary/entity classes, while complex behavior is best handled by introducing a controller class.
3. Entity – they express the logical structure of the system and represent persistent information such as people. These classes have the following characteristics:
   - Cut across many use cases
   - Are controlled by control classes
   - Provide information to, and accept information from, boundary classes
   - Represent persistent things managed by the system such as client
   - Are persistent.

3.1.9 Getting the final list

- Finally, to get to your list of final classes and objects for your system, weed out any false candidates for the classes and objects.
- To draw up a list of candidate classes:
  - use textual analysis to find a first list;
  - identify the "uses" technique to find any other classes;
  - identify the "players";
  - check to see if there is there is generalization;
  - examine the list you came up with during brainstorming;
  - check if there are entity, boundary or control classes that you identified;

3.1.9. Getting the final list (cont'd)

- Then:
  - look for analogies; and
  - look for re-use.

- Finally:
  - check your list of candidate classes against your agreed criteria to weed out "false" classes and objects - this gives you the list of classes and objects to include in your system.

Part 4

Class Diagram Elements

Introduction

- In the last section we examined different ways to identify the classes and objects in our problem domain.
- This is a very important (and sometimes tedious) exercise as you may have gathered. One of the most important facts to take home from that section is that in developing the class diagram nothing is left to chance.
- This is so since leaving out one class because we may think it's not important can cause real problems during the implementation phase when we write our code. The overall effect is that the system might not solve the problem we set out to from the beginning!
- In this part we examine the elements and notation that make up the class diagram before finally combining them to show a full class diagram with all its elements.

4.1 What is the class diagram

- A class diagram shows the existence of classes and their relationships in the static view of a system. It is the static model of the system being developed.
- The UML modeling elements in class diagrams are:
  - Classes, their structure and behavior.
  - relationships components among the classes like association, aggregation, composition, dependency and inheritance
  - Multiplicity and navigation indicators
  - Role names or labels

4.2 Class

- A class can be defined as:
- A description of a set of objects that share the same
  - attributes,
  - operations,
  - relationships and
  - semantics
- A software unit that implements one or more interfaces
- (Ojo and Estevez, 2005)

4.2 Class (cont'd)

- Class – the class is the major component of the class diagram (naturally). There are four types of classes that can be represented in the class diagram:
- Concrete class: A concrete class is a class that is instantiable; that is it can have different instances. This means that this is a class from which objects can be created.
- Abstract class: An abstract class is a class that has no direct instance but whose descendants classes have direct instances. If an abstract class can define the protocol for an operation without supplying a corresponding method we call this as an *abstract operation.* An abstract operation defines the form of operation, for which each concrete subclass should provide its own implementation. These types of classes are used mostly for creating descendant classes that in turn can be instantiated.
- Root class: this is a class written with the *root* stereotype and can't be a sub class.
- Leaf class: this class is written with a *leaf* stereotype and can't be a superclass.

4.2 Class (cont'd)

- The basic notation for a class in the UML is a solid-outline rectangle with three compartments separated by horizontal lines.
- The three compartments are divided as follows:
  - The top compartment contains the class name and any other general properties of the class (usually using the << property >> notation)
  - The middle compartment holds a list of attributes of the class
  - The bottom compartment holds a list of operations of the class
- In many literature alternative styles are used; the UML allows the use of this notation, for example:
  - Suppressing the attributes compartment
  - Suppressing the operation compartment
  - Suppressing both the attribute and operation compartments.
- Fig 3 shows an example of classes drawn using the UML notation.

4.2 Class (cont'd)

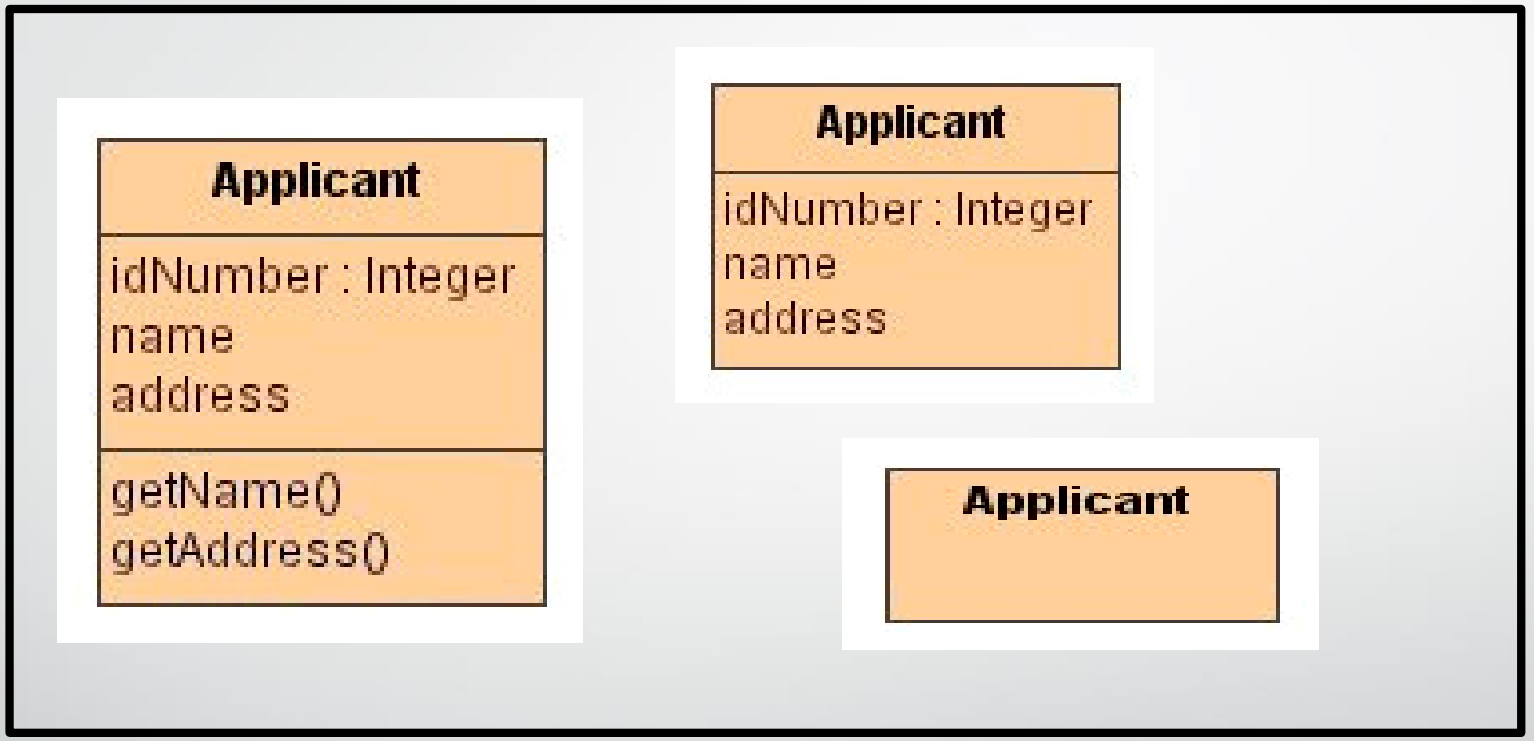

Fig 3. Class notation (Ojo and Estevez, 2005)

4.3 Stereotype

- A stereotype is a mechanism allowing to extend the semantics of UML
- It is used to present more information about an artifact
- The notation for a stereotype is the name of a new element within the matched guillemets, e.g. <<thread>>
- Attributes or operation lists in a class may be organized into groups with stereotypes.
- A number of stereotypes of classes and data types: thread, event, entity, process, utility, metaclass, and so on. (Ojo and Estevez, 2005)
- Table 2 presents some stereotype descriptions.

Stereotype	Description
Thread	An active class which specifies a lightweight flow that can execute concurrently with other threads within the same processes.
Process	An active class which specifies a heavyweight flow that can execute concurrently with other processes.
Control	A class which owns almost no information about itself. It represents a behaviour rather than resources and directs the behaviour of other objects almost having no behaviour of its own.
Entity	A class which represents a resource in the real world. It describes its features and their current condition (their state) and preserves its own integrity regardless of where and when it is used.
Utility	A class whose attributes and operations are all class scoped. That is a class which no instance.
Metaclass	A classifier whose objects are all classes.
Powerclass	A classifier whose objects are the children of a given parent.
Enumeration	A user defined data type that defines a set of values that do not change.

Table 2. Stereotype descriptions (Ojo and Estevez, 2005)

4.4 Association

- The UML defines association as “"the semantic relationship between two or more classifiers that involve connections among their instances”. (Larman, 2002).
- An association relationships denotes a semantic connection between two classes
- It shows BI directional connection between two classes.
- It is a weak coupling as associated classes remain somewhat independent of each other. An example of an association is student reads books, a person has a company, and a customer uses an ATM.
- The UML notation for an association relationship is a straight line between the two classes as demonstrated in fig 4.

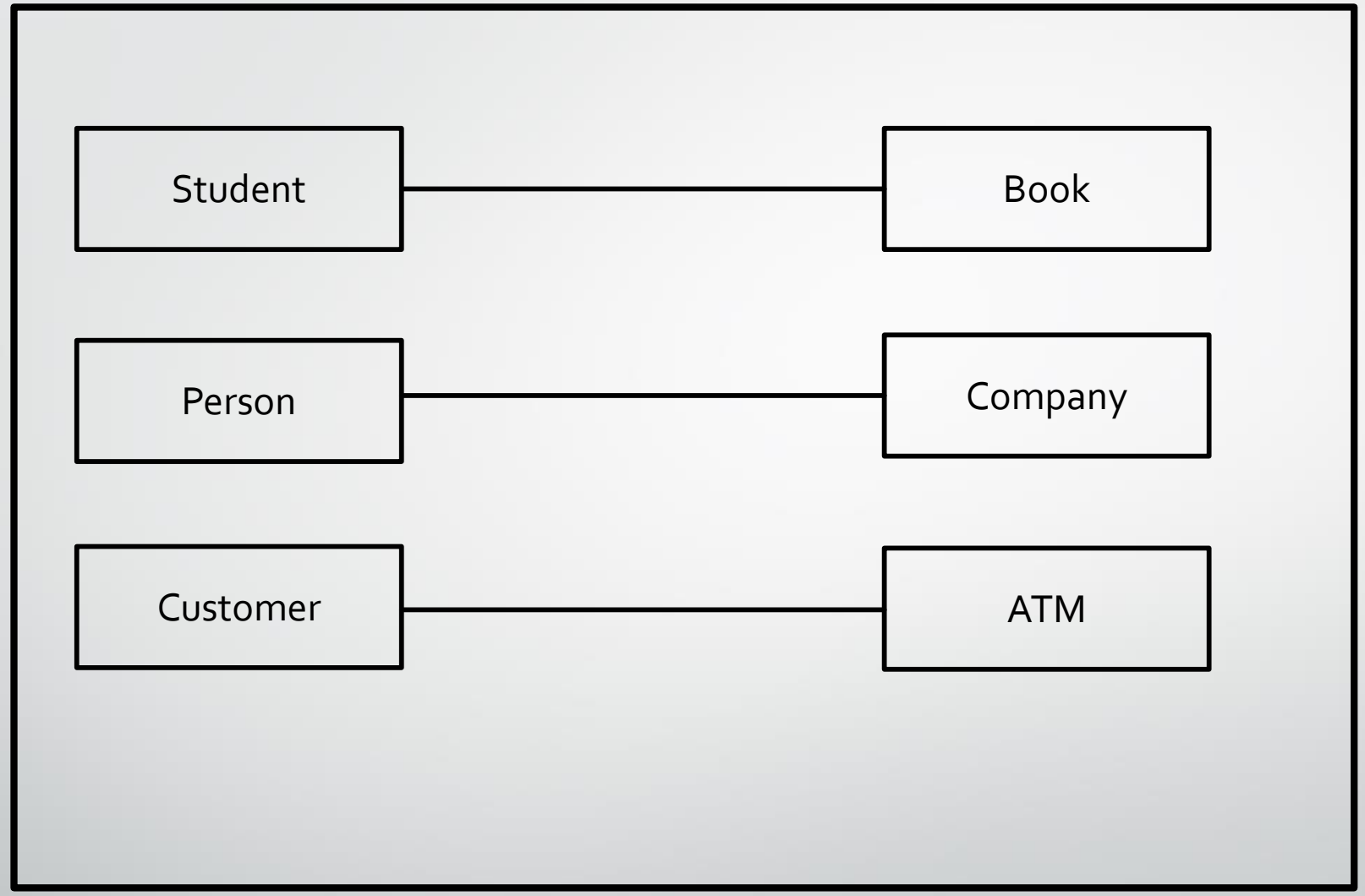

Fig 4. Association relationship

4.5 Aggregation

This is a special type of association

- The association with label "contains" or "is part of" is an aggregation
- It represents "has a " relationship
- It is used when one object logically or physically contains another
- The container is referred to as the aggregate, and it has an unfilled diamond at its end
- The components of aggregate can be shared with others
- It expresses a whole - part relationships.
- Examples of aggregation includes: a car (whole) has an engine (part); a customer (whole) has an ATM card (part).
- Fig 5 shows the relationships using UML notation.

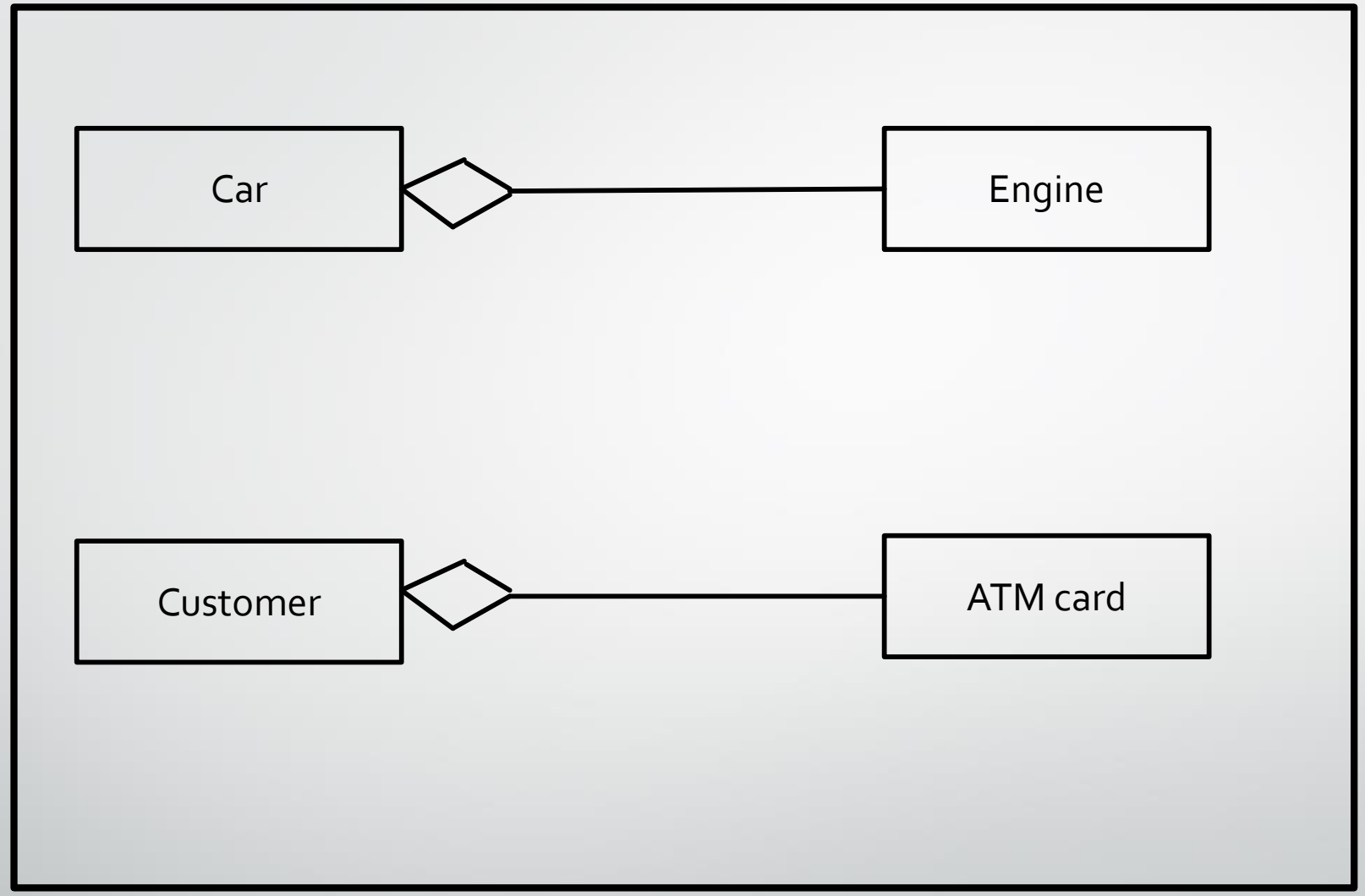

Fig 5. Aggregation relationship

4.6 Composition

This is a strong form of aggregation

- It expresses the stronger coupling between the classes
- The owner is explicitly responsible for creation and deletion of the part
- Any deletion of whole is considered to cascade its part
- The aggregate has a filled diamond at its end, which emphasizes the fact that this is a strong form of aggregation.
- An example of aggregation is the relationship between a university and its departments. Deleting the university cascades its departments; also deleting a window (like in OS) cascades to all open GUIs on it. Fig 6 shows an example of composition.

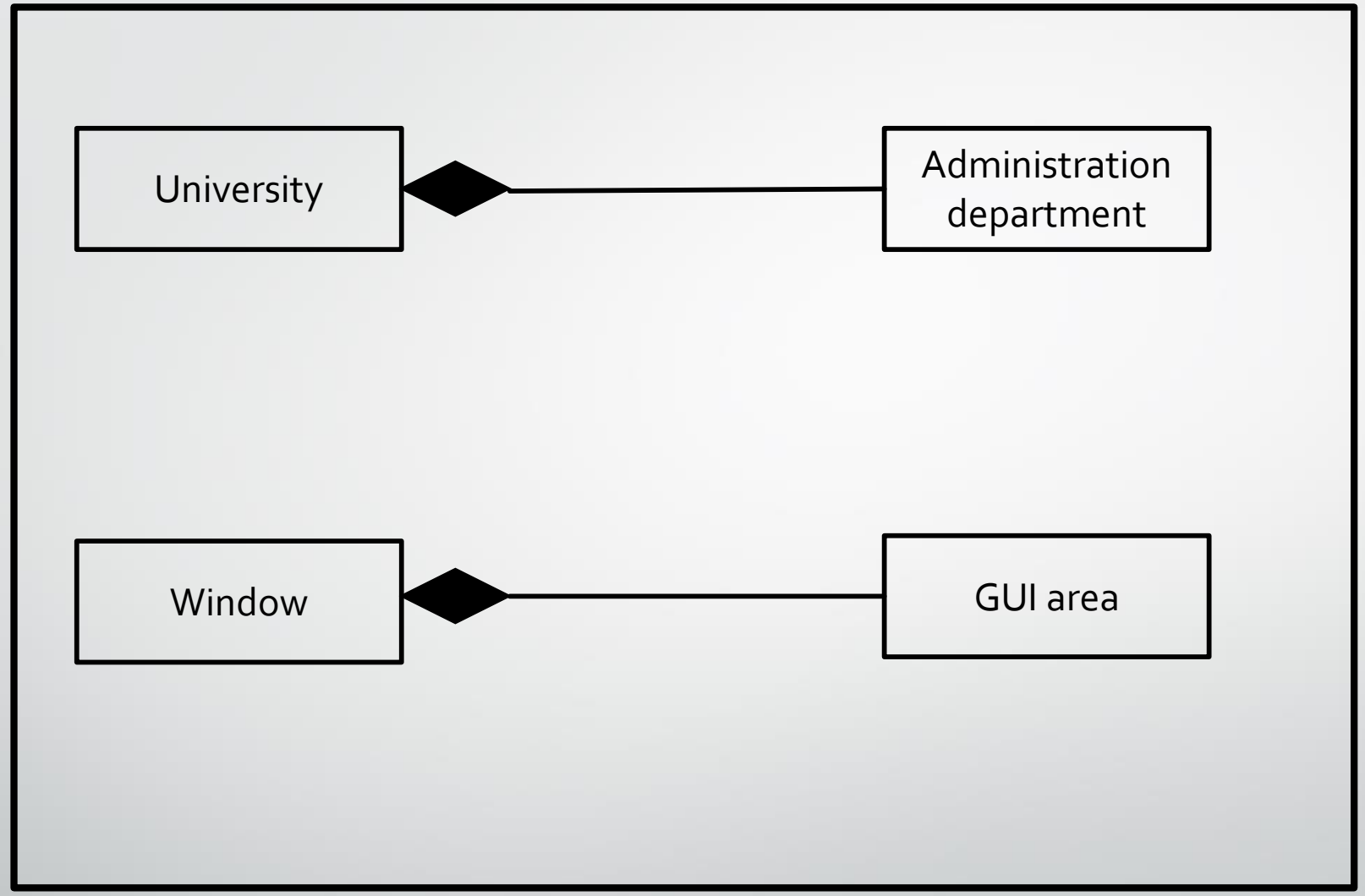

Fig 6. Composition relationship

4.7 Dependency

- It represents a relationship between two elements such that a change to one element (called the source) will affect another (called the target) (Ojo and Estevez, 2005).
- The UML notation for this is a dotted arrow from the source to the target (unidirectional).
- Examples of dependency include; a client depends on the server; a procedure depends on a policy.
- Fig 7 shows the dependency relationship.

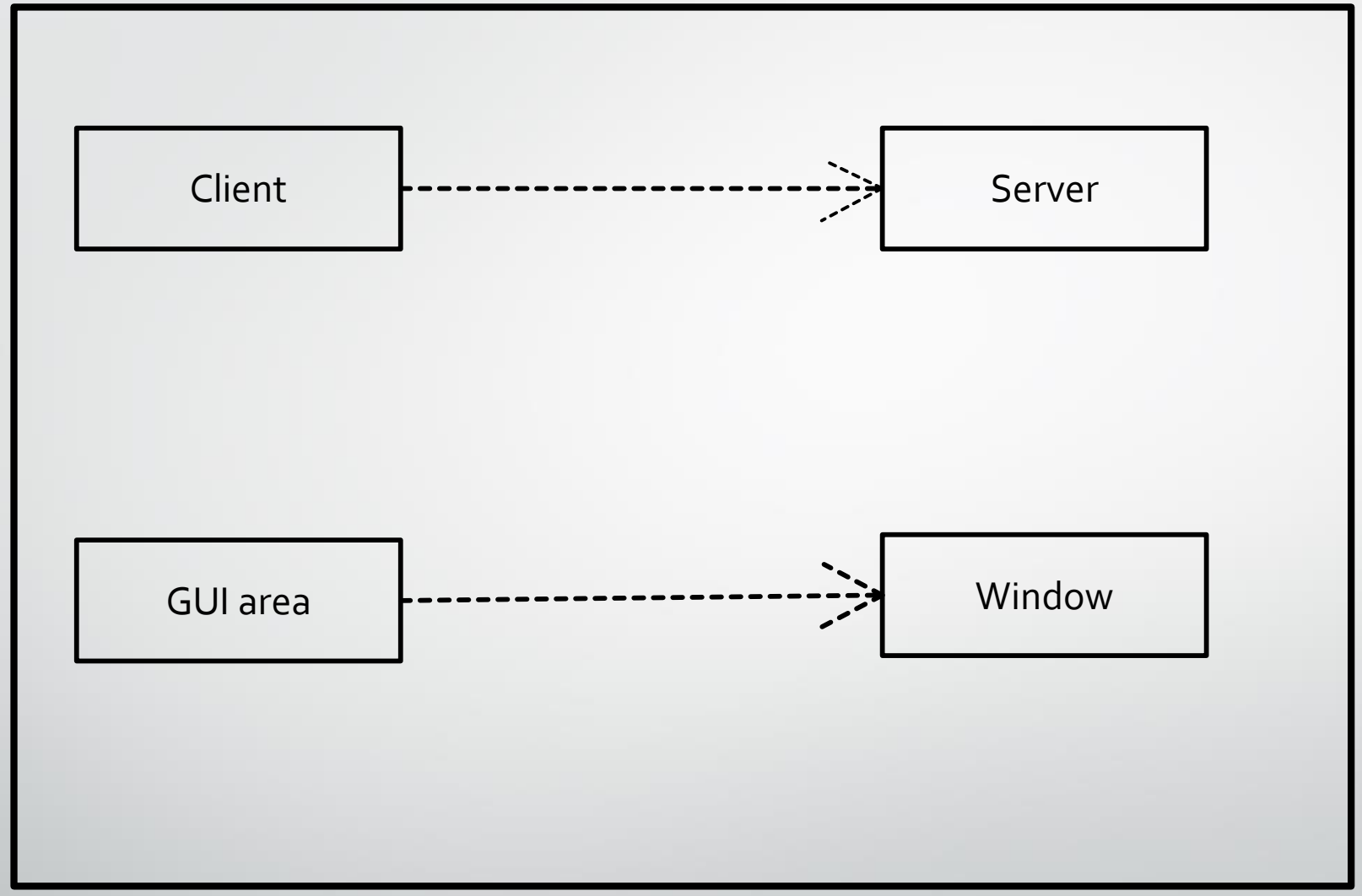

Fig 7. Dependency relationship

4.8 Inheritance (Generalization)

- The inheritance relationship helps in managing the complexity by ordering objects within trees of classes with increasing levels of abstraction. Notation used is solid line with arrowhead, as shown in fig 8.
- Generalization and specialization are points of view that are based on inheritance hierarchies.
- Generalization is equivalent to “kind-of” or “type-of” relationship, while specialization is the opposite of generalization.

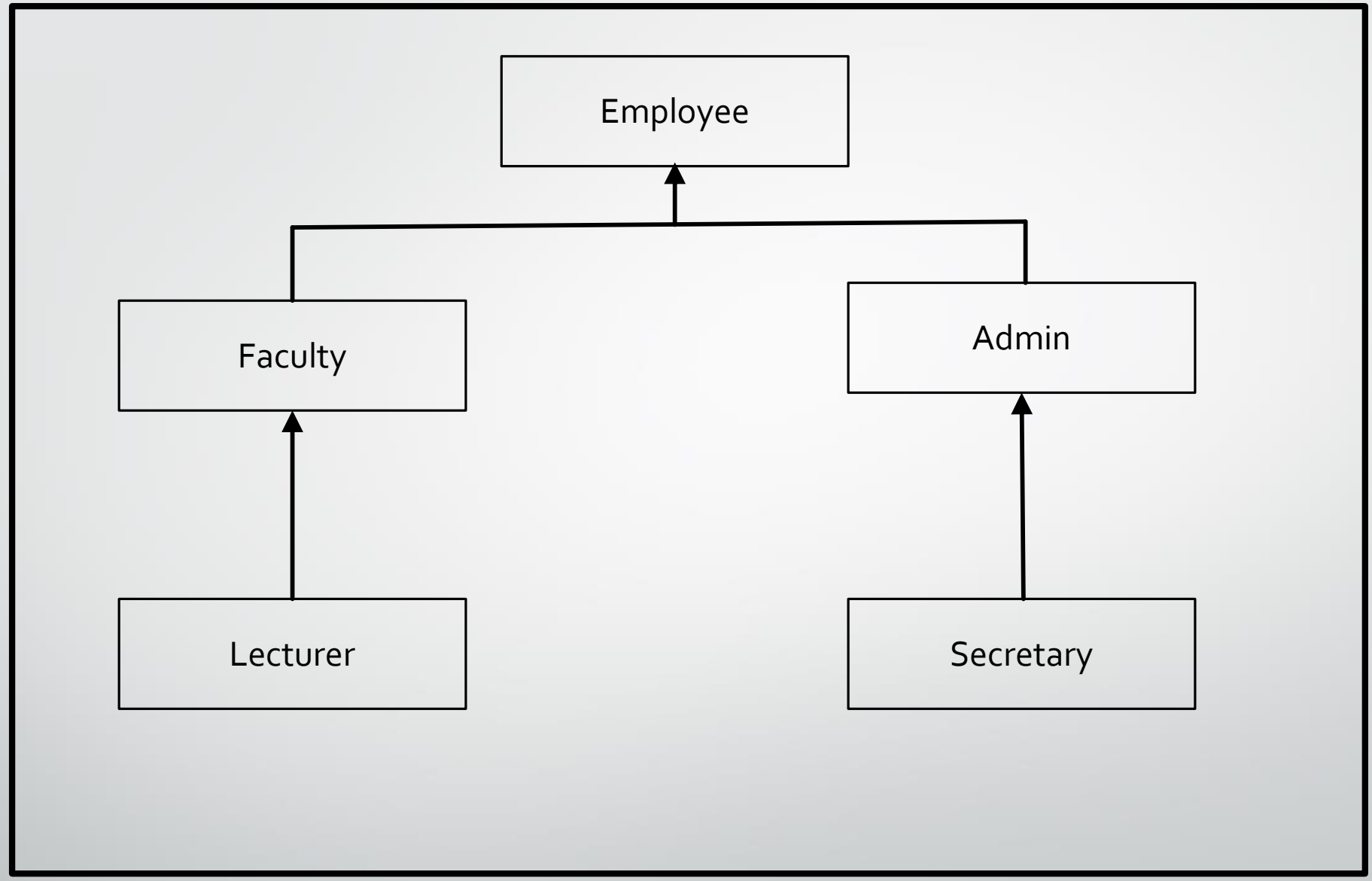

Fig 8. Generalization

4.9 Instantiation

- This relationship is defined between parameterized class and actual class.
- Parameterized class is also referred as generic class.
- A parameterized class can’t have instances unless we first instantiated it.
- Fig 9 shows an example of instantiation

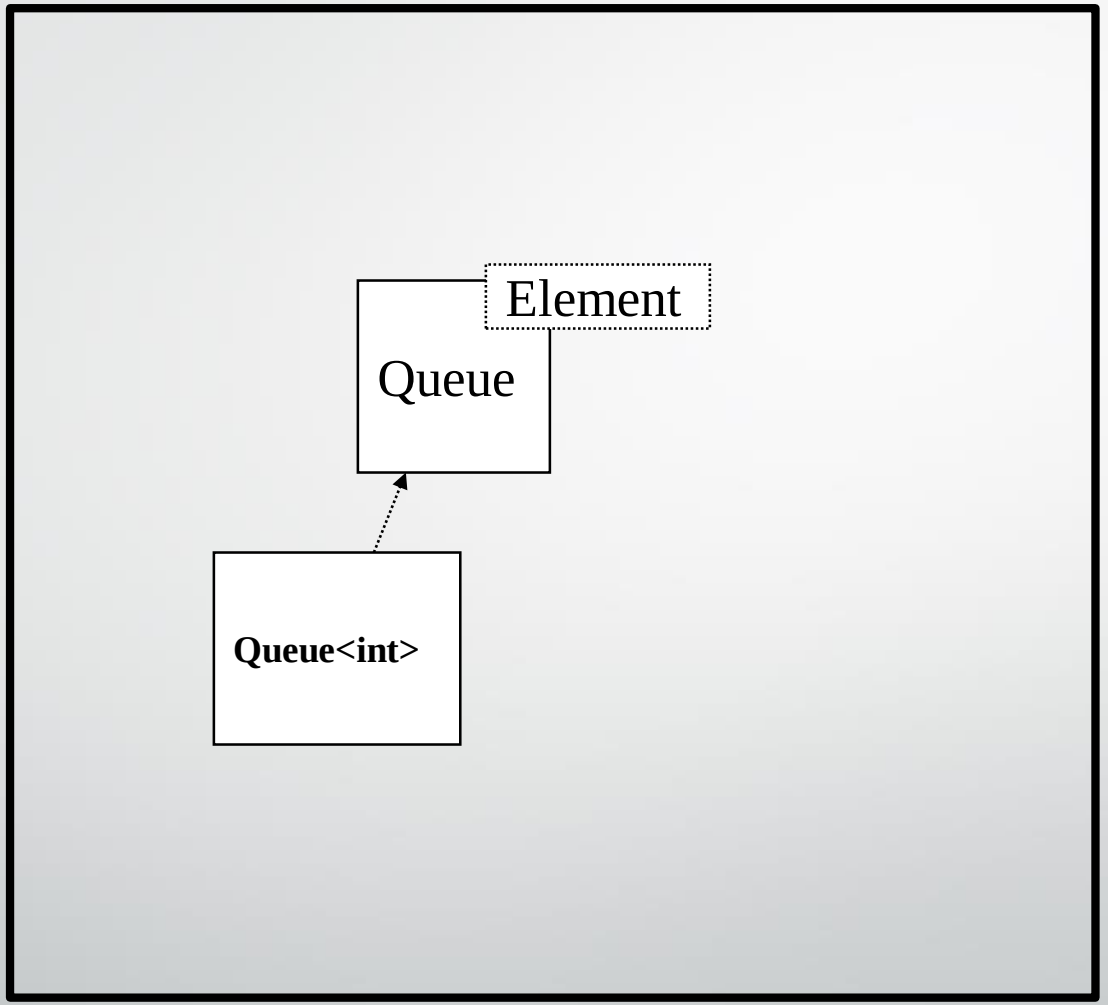

Fig 9. Instantiation

4.10 Multiplicity (Cardinality)

- The multiplicity relationship shows how many objects of one class can be associated with one object of another class.
- Number of instances of each class involved in the dialogue is specified by cardinality.
- **Common multiplicity values:**

Symbol	**Meaning**
1	One and only one
0..1	Zero or one
M...N	From M to N (natural integer)
0..*	From zero to any positive integer
1..*	From one to any positive integer

- Fig 10 shows an example of multiplicity

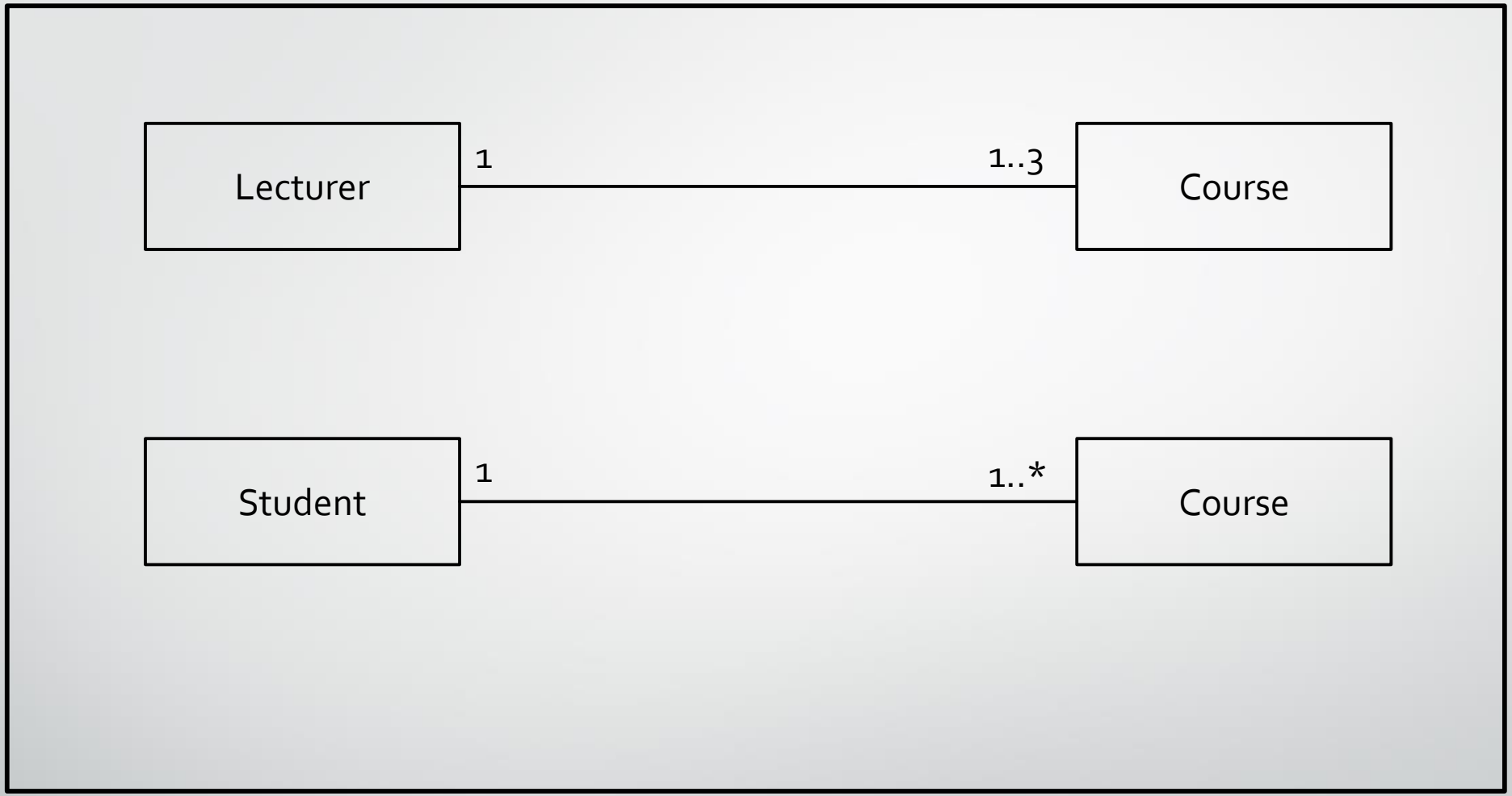

Fig 10. Multiplicity

4.11 Refinement attributes

- Stereotypes are part of the range of extensibility mechanism provided by UML
- It permits user to add new model element classes on top of the kernel predefined by UML.
- Constraints are functional relationship between the entities and object model. The entities include objects, classes, attributes, association, links.
- A constraint restricts the values that entities can assume.
- UML doesn't specify a particular syntax for constraints, other than they should appear between braces, so they may therefore be expressed using natural language, pseudo code, navigation expression or mathematical expression
- UML1.2 does prefer the use of a constraint language OCL i.e. Object Constraint Language, which is subset of UML.

4.11 Refinement attributes (cont'd)

Example:Constraints

- Number of withdrawal transaction should be less than five per day.

Transaction

Constraint on the same class.

{No. of transaction <=5 /day}

- No window will have an aspect ratio i.e. (length/width) of less than 0.8 or > 1.5

Window length/width

A constraint between the properties of the same object

{0.8<=length/width <= 1.5}

4.11 Refinement attributes (cont'd)

Qualifier:

- UML provides a role of constraint notation to indicate different kind of collections that may be inherent in the analysis model
- Common role constraints for multi valued roles include

  {ordered} Collection is maintained in sorted manner

  {bag} Collection may have multiple copies of same item.

  {set} Collection may have at most one copy of given item.
- Some constraints may be combined such as: {ordered set}

4.11 Refinement attributes (cont'd)

- Another common design scheme is to use a key value to retrieve an item from the collection. This is called as qualified association and the key value as qualifier.
- A qualified association is the UML equivalent of a programming concept variously known as associative arrays, maps, dictionaries
- A qualified association relates two object classes and a qualifier
- The qualifier is a special attribute that reduced the effective multiplicity of an association.
- One to many and many to many association may be qualified.
- Check for many to many relationship, if any, normalize with qualifier or association class.
- Check for the scope forming abstract classes and template classes.
- Check for helper functions.

Thought can be given for using the design patterns.

4.12 Building up to the class diagram

- The collaboration diagram can be used to build up to the class diagram. In the collaboration diagram we have shown the objects, their interaction and detailed message signature.
- This information is carried forward to the class diagram.
- At this point, we group the similar objects and form classes.
- Messages get mapped to responsibilities for respective classes.
- Find the attributes for every class.
- Transform the links to appropriate relationships.
- Relationship is further refined with respect to multiplicity and navigability.
- Fig 11 shows a complete class diagram

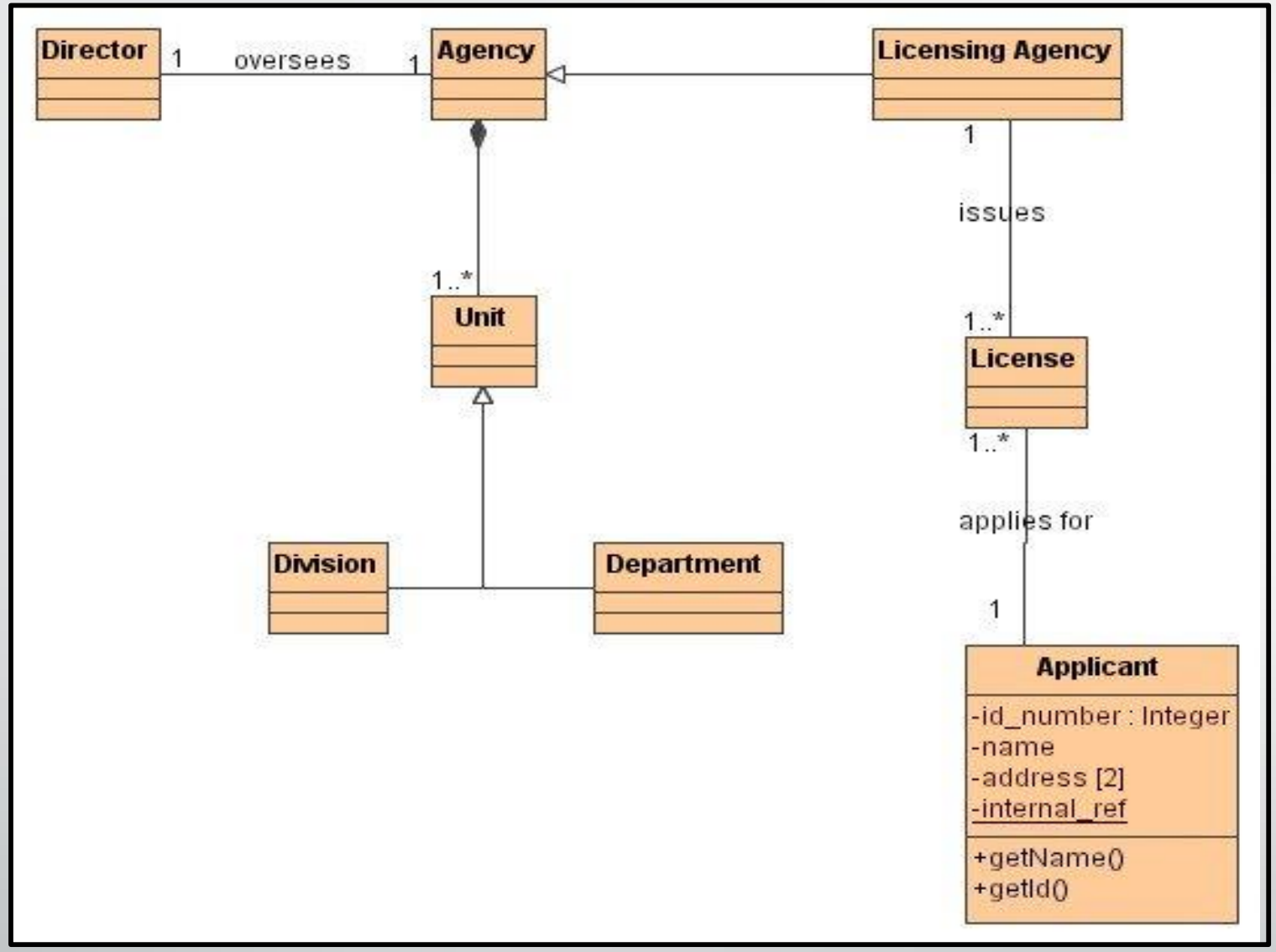

Fig 11. Class diagram (Ojo and Estevez, 2005)

Part 5

Object diagram

5.1 Introduction

- The object diagram models the instances of classes contained in class diagrams
- It shows a set of objects and their relationships at one time
- Modelling object structures involves taking a snapshot of a system at a given moment in time
- The object diagram is an instance of a class diagram or the static part of an interaction diagram
- It contains objects and links (Ojo and Estevez, 2005)

5.2 Usage of the Object Diagram

- Object diagrams are used to:
  - visualize
  - specify
  - construct
  - document
- The existence of certain instances in a system, together with their relationships.

5.3 Designing the Object Diagram

- The following steps show how to create an object diagram:
- Identify the function/behavior of interest that results from interaction of classes, interfaces and other artifacts
- For each function/behavior, identify the artifacts that participate in the collaboration as well as their relationships
- Consider one scenario that invokes the function/behavior, freeze the scenario and render each participating object
- Expose the state and attribute values of each object, as necessary to understand the scenario
- Expose the links among these objects
- Fig 11 shows an example of an object diagram.

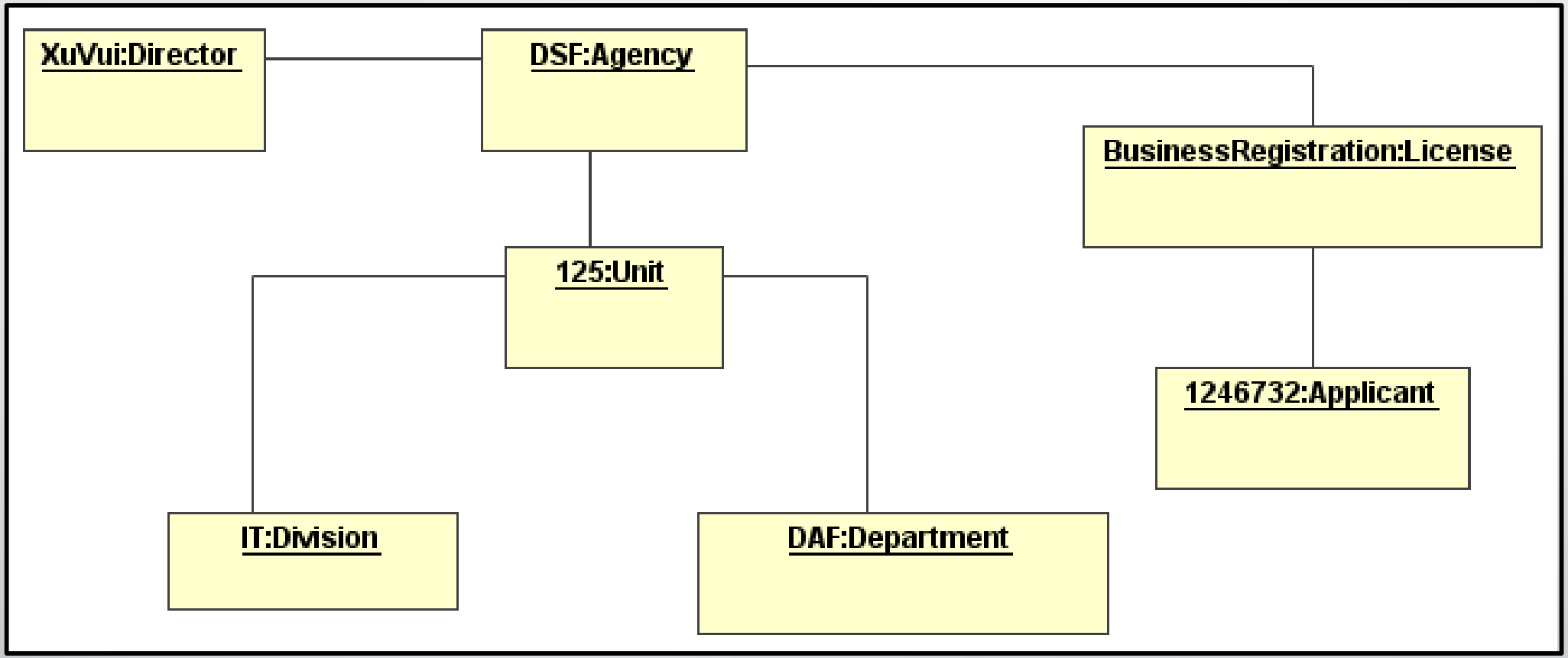

Fig 11. Object diagram (Ojo and Estevez, 2005)

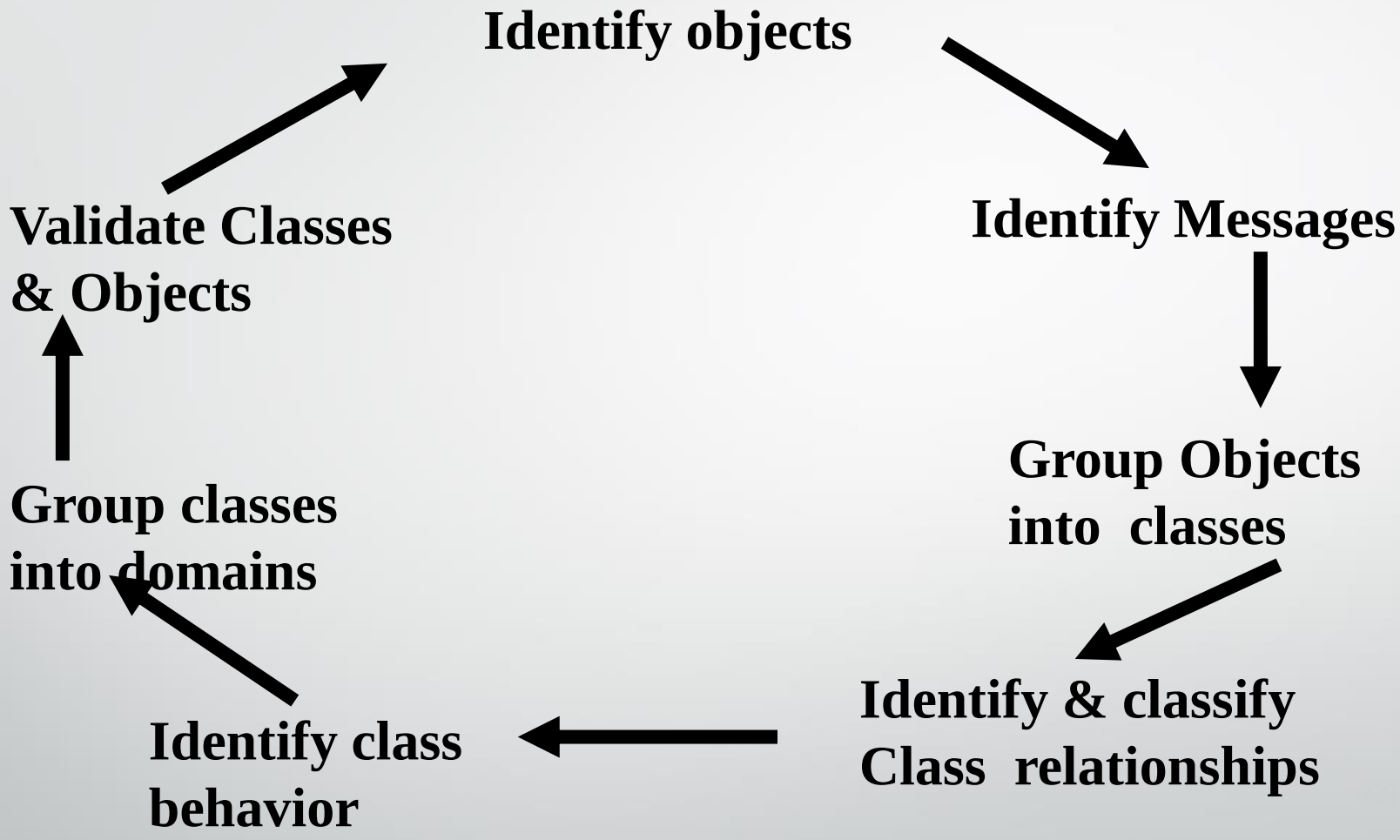

Fig 12. OOAD an iterative and incremental approach

Summary

- The structural model is described using two other models: the static model and the dynamic model.
- Static analysis is concerned with what the system should do. In doing so we differentiate this with "how" the system should do it. This is done through the static model which is a design implementation of static analysis.
- Object identification is done using the following techniques: textual analysis, finding "uses", finding players, finding generalization, finding analogies, finding reuse, brainstorming, and using RUP stereotypes.
- The UML modeling elements in class diagrams are: Classes, their structure and behavior; relationships components among the classes like association, aggregation, composition, dependency and inheritance; multiplicity and navigation indicators; role names or labels.
- A constraint restricts the values that entities can assume.
- Object diagrams are used to: visualize, specify, construct, and document, the existence of certain instances in a system, together with their relationships.

References

- Arlow, J., & Neustadt, I. (2013). *Uml 2 and the unified process: Practical object-oriented analysis and Design* (Second). Addison-Wesley.
- Dennis, A., Wixom, B. H., Tegarden, D. P., & Seeman, E. (2015). *System analysis & design: An object-oriented approach with Uml*. Wiley.
- *IS352 Object Oriented Systems Analysis and Design slides*. (2007). Lecture.
- (n.d.). *Object Oriented Analysis and Design using UML*. ms.
- Larman, C. (2002). *Applying Uml and patterns: An introduction to object-oriented analysis and design and the Unified Process*. Prentice Hall.
- O'Docherty, M. (2005). *Object-oriented analysis and Design: Understanding system development with Uml 2.0*. John Wiley & Sons.
- Ojo, A., & Estevez, E. (2005). (rep.). *Object-Oriented Analysis and Design with UML - Training Course* (Vol. 1).