Lecture 6: Software requirements modelling (static)

- Class model
- Class modelling on the requirements level
- Classes mapped from vocabulary notions

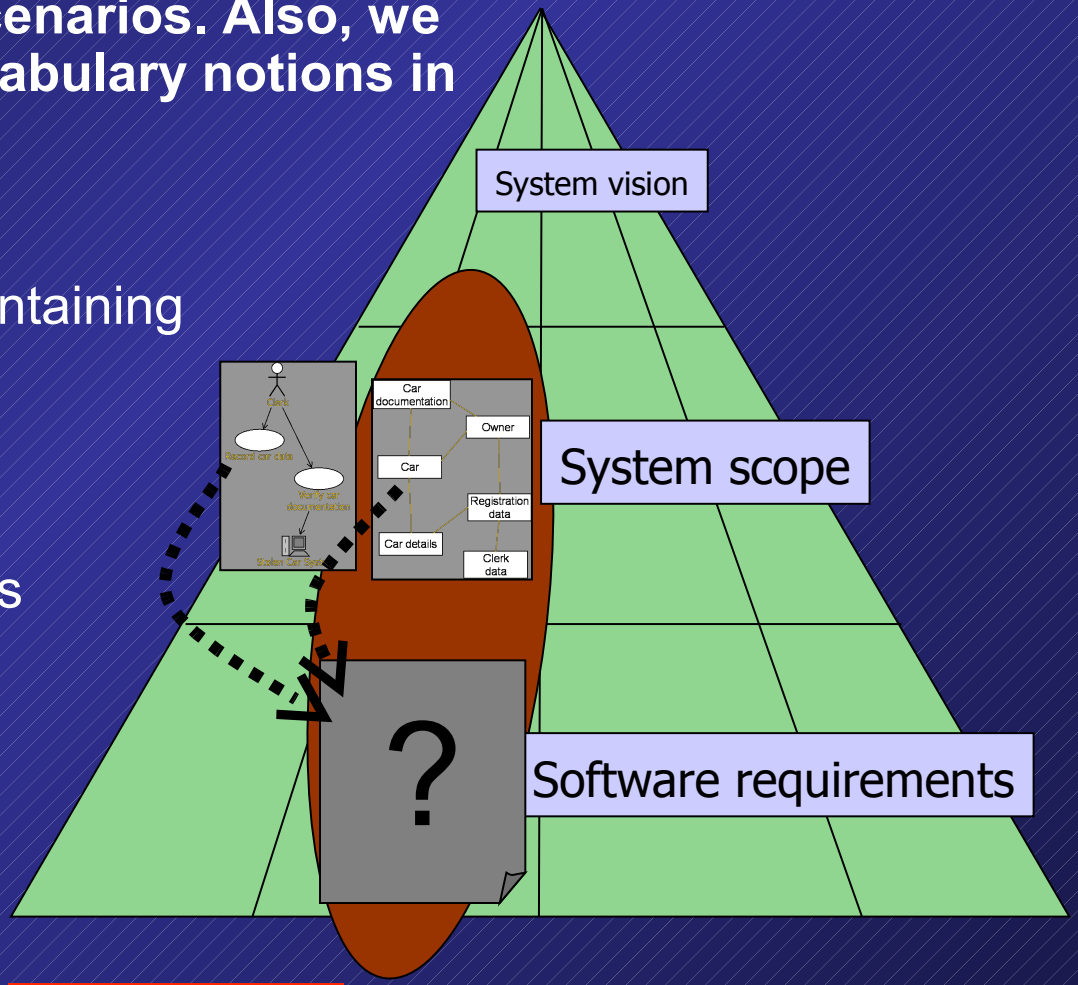
What do we want to do?

We want to define all the subjects, verbs and objects used in use case scenarios. Also, we want to describe all the vocabulary notions in great detail.

How to do it?

- draw a class diagram containing all the elements used in scenarios mapped from vocabulary notions
- add attributes, operations and relationships to describe details of the static requirements

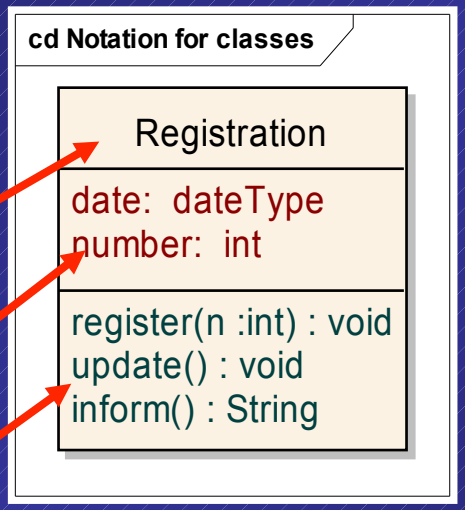
But first – notation!

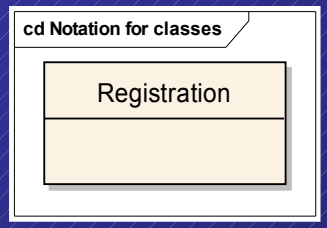


Notation for classes

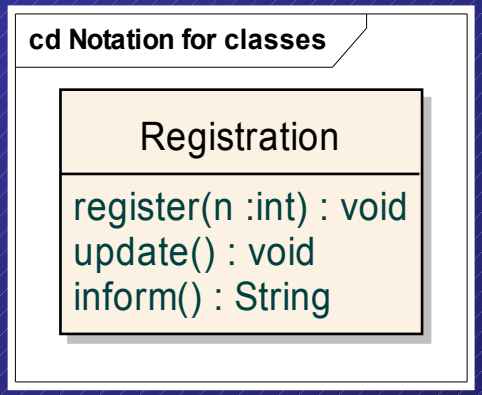
In UML class is represented simply by a rectangle divided usually into three parts (compartments).

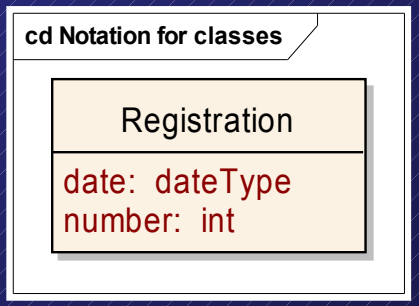
- Upper part always contains the class name
- Middle part usually contains class attributes
- Lower part usually contains class operations
- Other parts might contain tags, constraints, other attributes and operations, etc.





On different diagrams the class might contain just some compartments (compressed form representation).



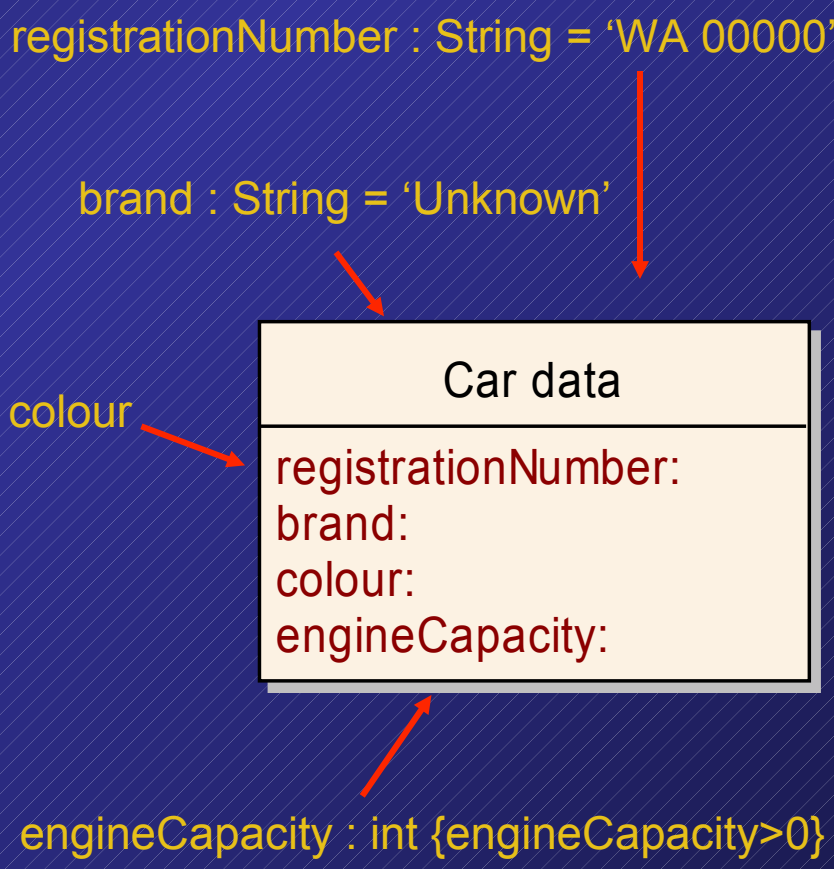


Notation for attributes

Attributes denote data elements that are included in every object of the class and is represented by a separate value. Attributes don't have their own identity – they are totally controlled by the objects of which they are parts.

Attributes in UML, as a minimum, have their names defined. Additionally we can specify their data type and initial value.

Attributes can also be denoted with constrains that narrow the possible values of the attribute.



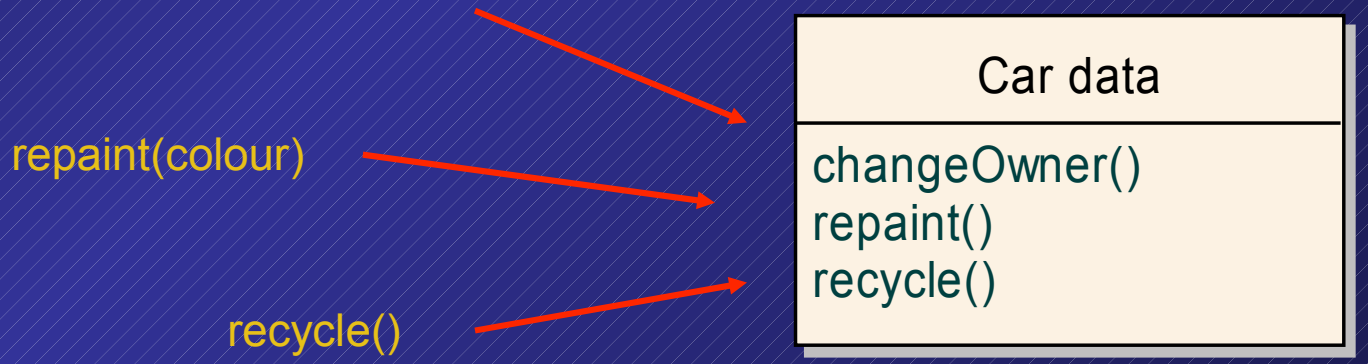
Notation for operations

Operations denote services (data processing) that are associated with the given class (performed on objects of the class).

Operations in UML, as a minimum, have their names and signatures in parentheses. Signatures contain parameters' names and types and return value type.

Operations can have their signatures empty or compressed (not shown). Operation can have constraints set on their parameters.

changeOwner(o : Owner, regNumber: String) : bool {o != NULL}



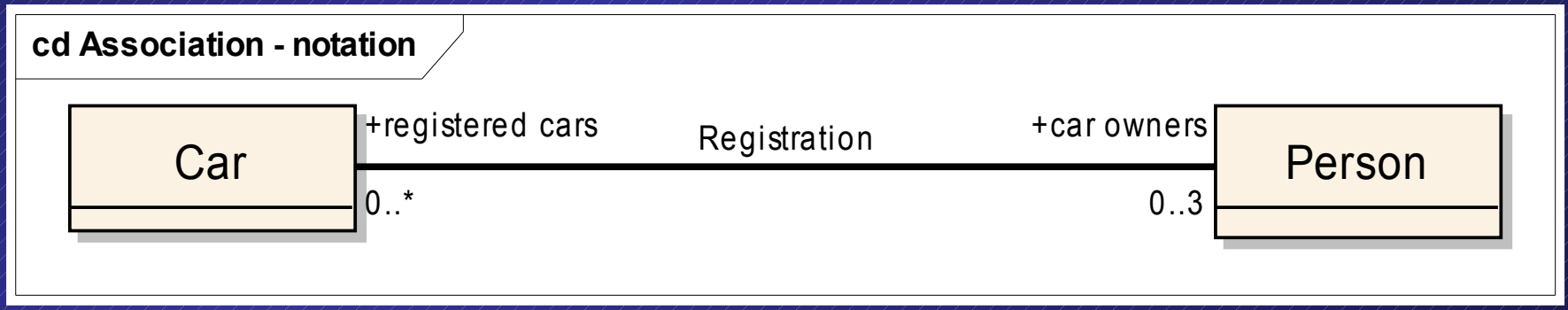
Class associations

Class association denotes that two (or more) classes (e.g. notions in the vocabulary) are in close relationship.

On the software requirements level, two classes are in association if their definitions contain references to the other class. This can be compared to denoting a “hyperlink” relationship between two notions in a “wikipedia”.

NOTE: associations have a precise meaning on the design level!

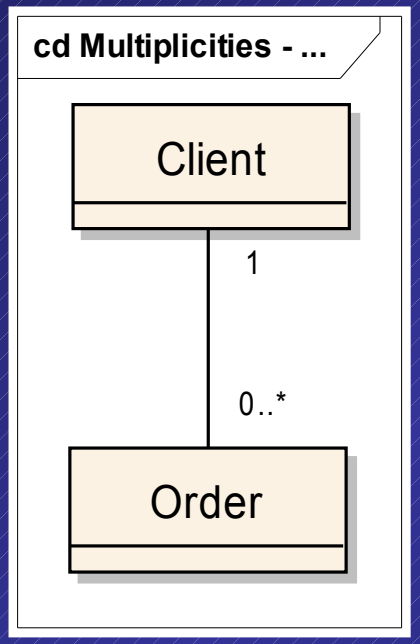
Associations are denoted with a line (might be curved) with: association name, role names, multiplicities and other.

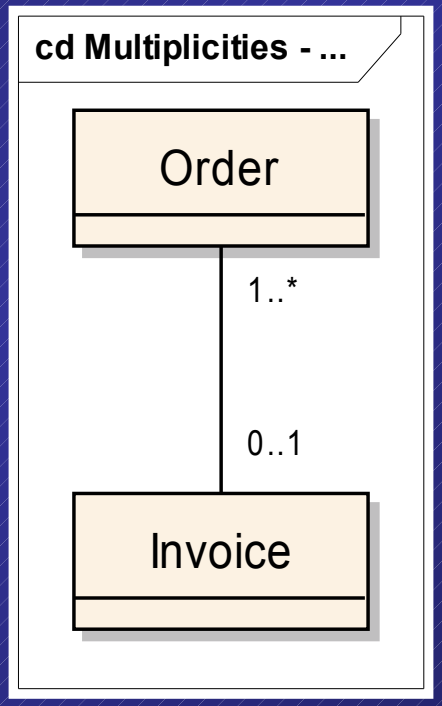


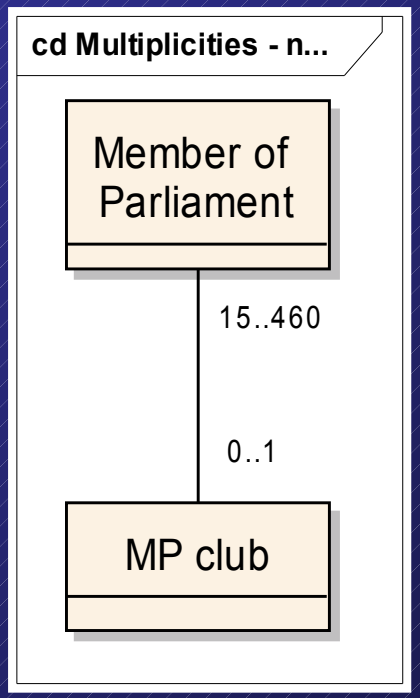
Association multiplicities

Multiplicity denotes the number of objects that might be linked with one object of the opposite class.

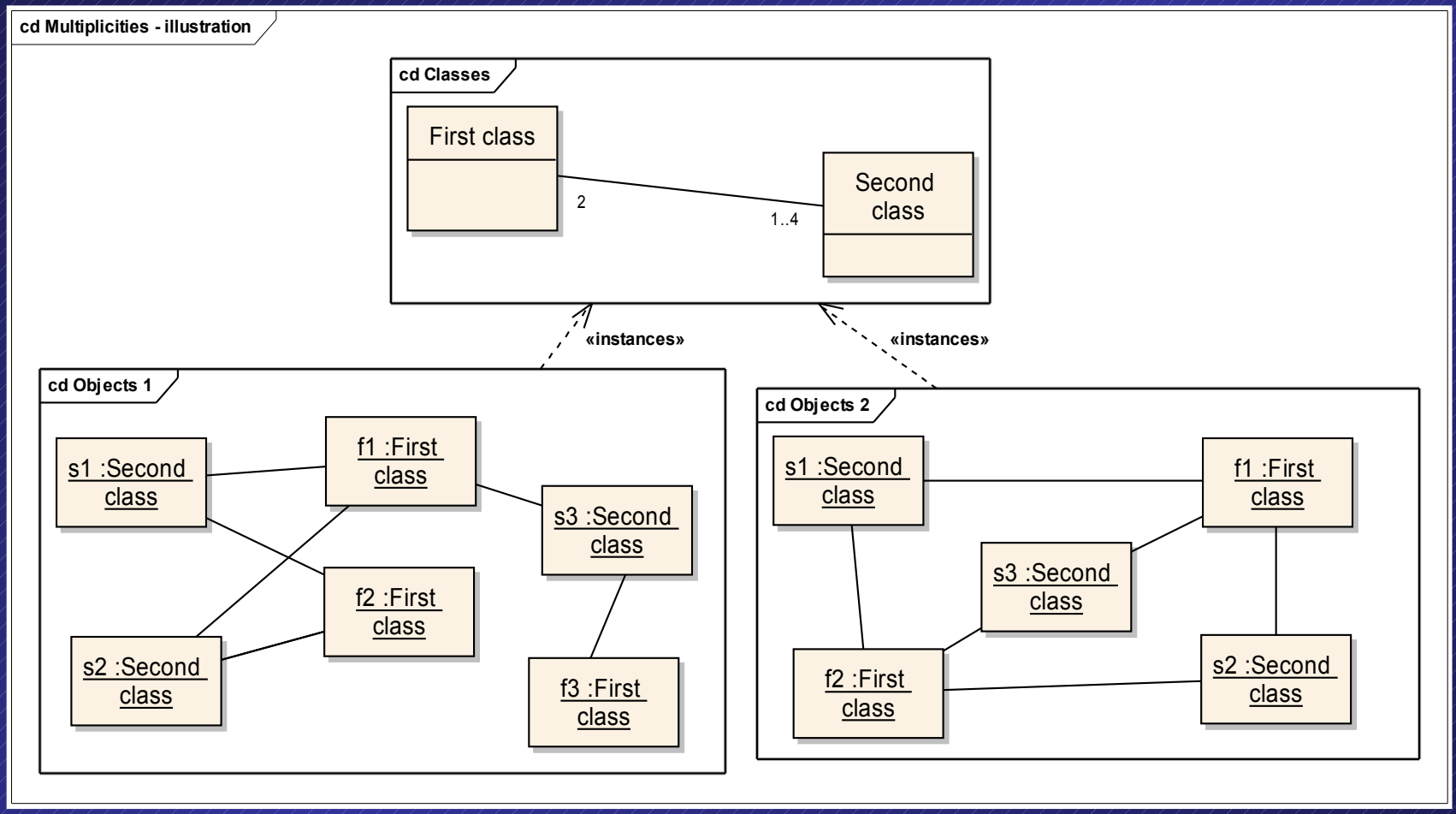
Multiplicity is written as a range (from . . to) placed near an association end. " * " denotes infinity. If the lower and upper boundary of the range are equal we can put just one value (e.g. "1" instead of "1 . . 1").







Multiplicities - example



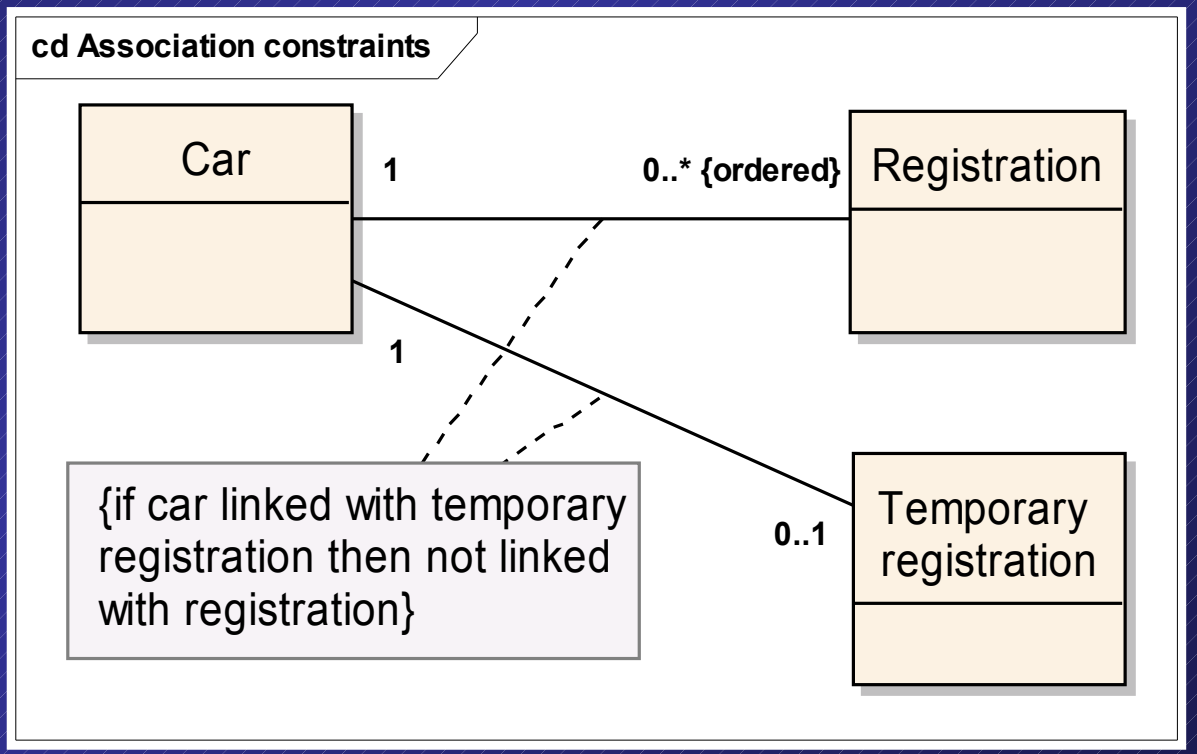
Two object diagrams illustrate multiplicites – objects there are linked correctly, according to multiplicities on the class diagram.

Association constraints

As any other elements in UML, associations can be adorned with constraints. Constraints can pertain association ends, associations as a whole or even several associations at the same time.

Here, one of the association ends is constrained (we still need to specify the semantics of Registrations being ordered).

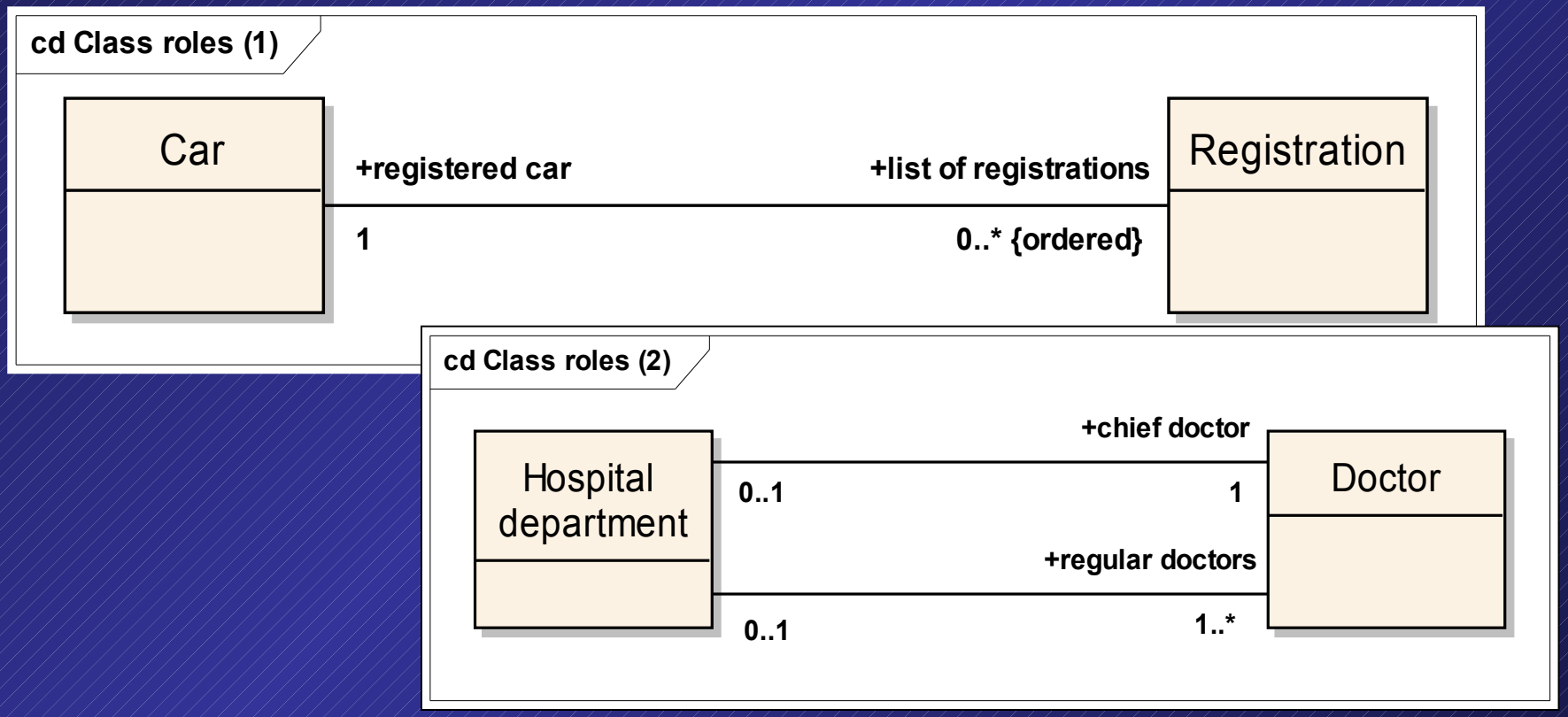
Also, two associations are constrained.



Class roles in associations

We can denote a role which is fulfilled by a class in an association.

NOTE: two classes can have several associations with different roles specified



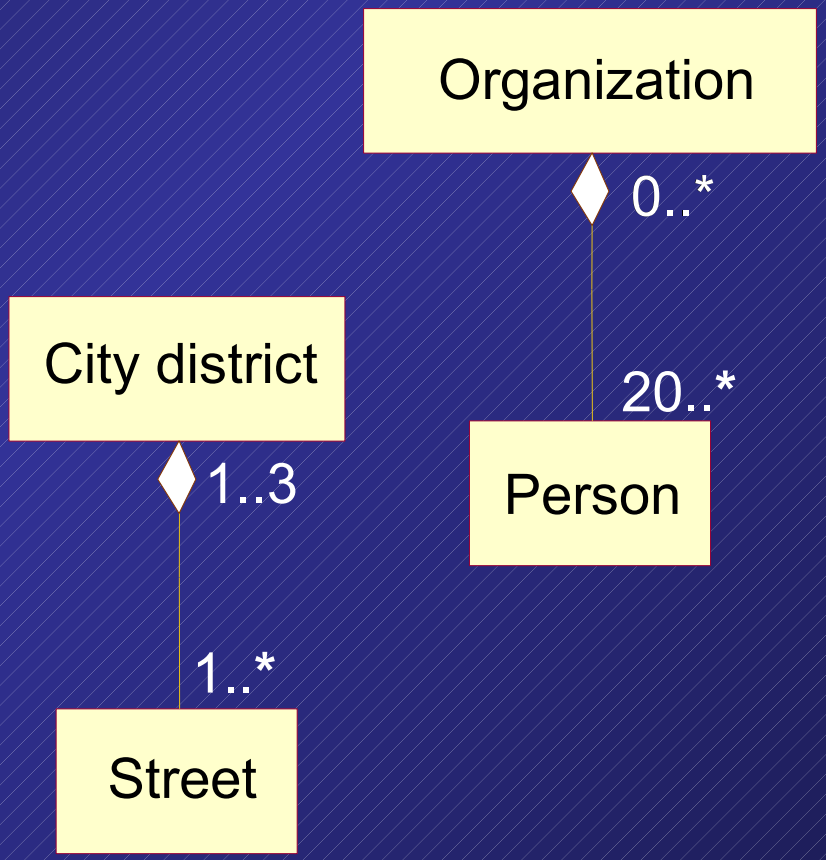
Aggregations

Aggregation is a kind of association denoting a "part-whole" relation between classes.

It represents a situation where objects of one class (the aggregate) contain objects of other class (the parts).

Aggregate in the association is marked with an empty diamond. Otherwise, aggregation has the same notation as association.

It can be noted that the parts can be linked with several aggregates.

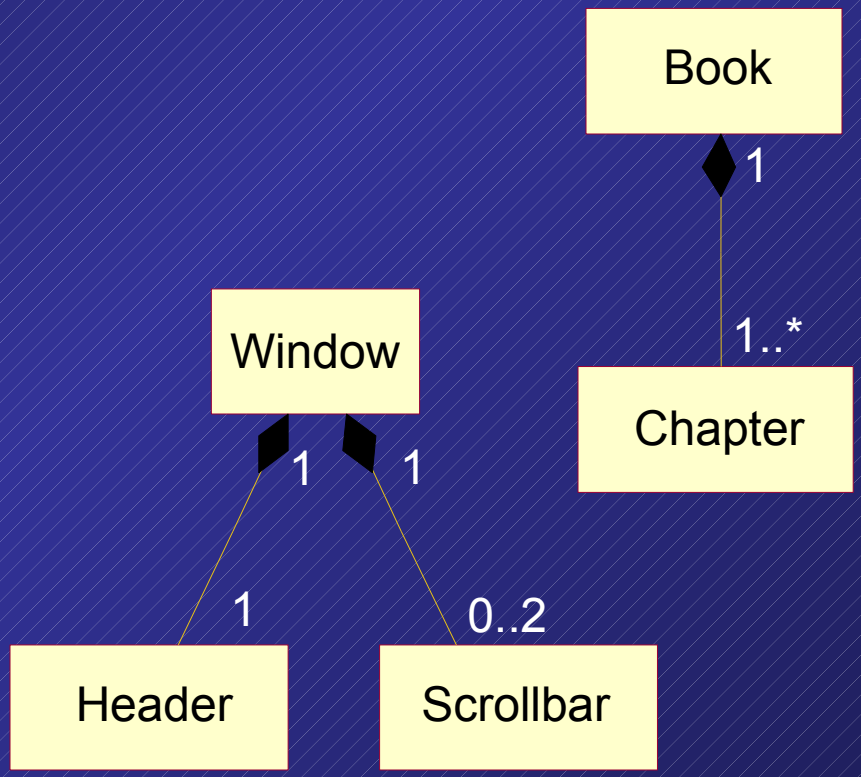


Compositions

Composition is a specific type of aggregation. It forces a strong control of the whole over its parts (including its lifetime).

Parts in the composition cannot be divided through several aggregates (thus always multiplicity "1" at the aggregate end).

The parts cannot "live" by themselves – their lifetime is bound by the lifetime of the aggregate (usually – they are created and deleted together).

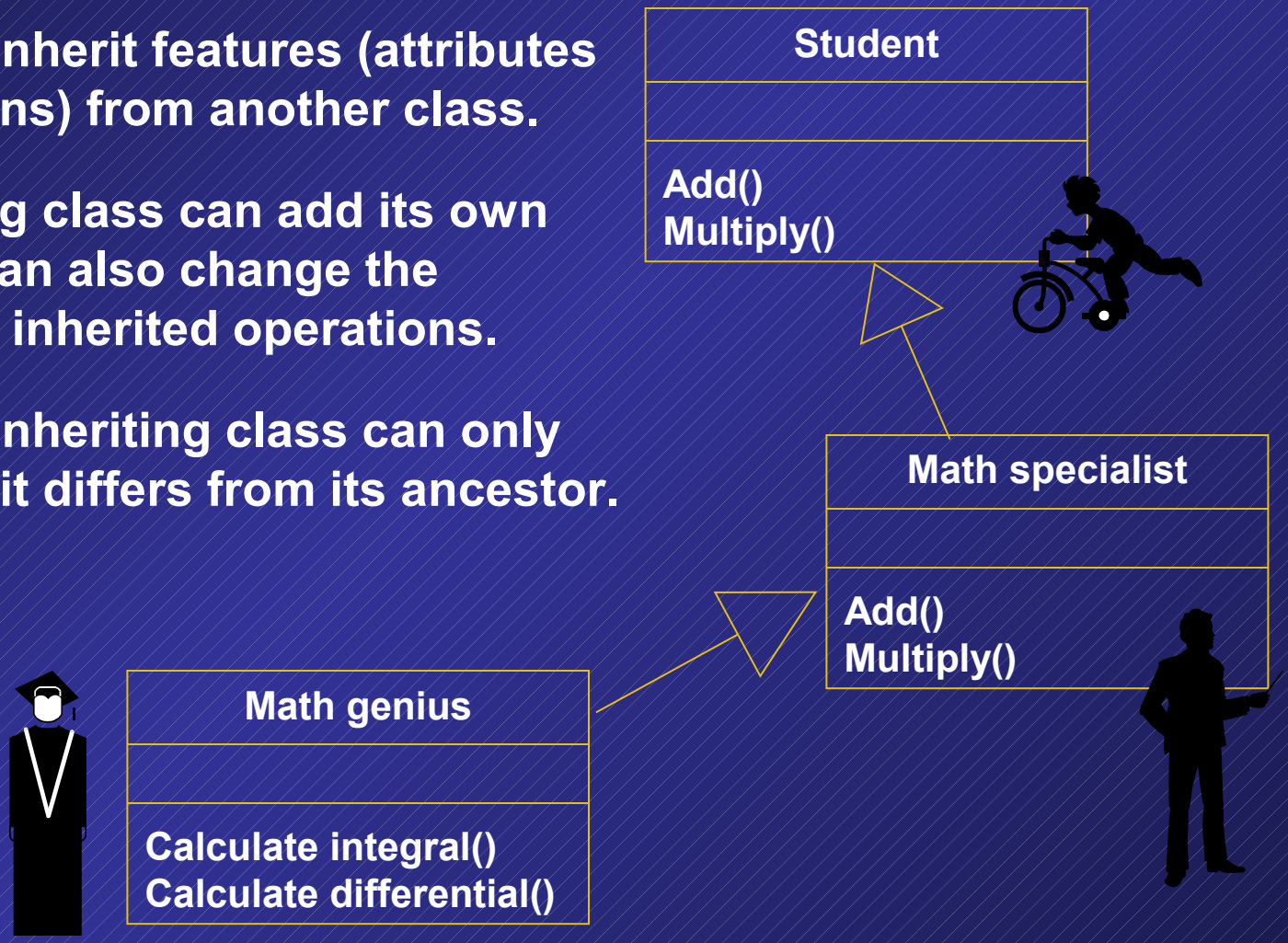


Inheritance of features

A class can inherit features (attributes and operations) from another class.

The inheriting class can add its own features; it can also change the behaviour of inherited operations.

Benefit: the inheriting class can only specify how it differs from its ancestor.

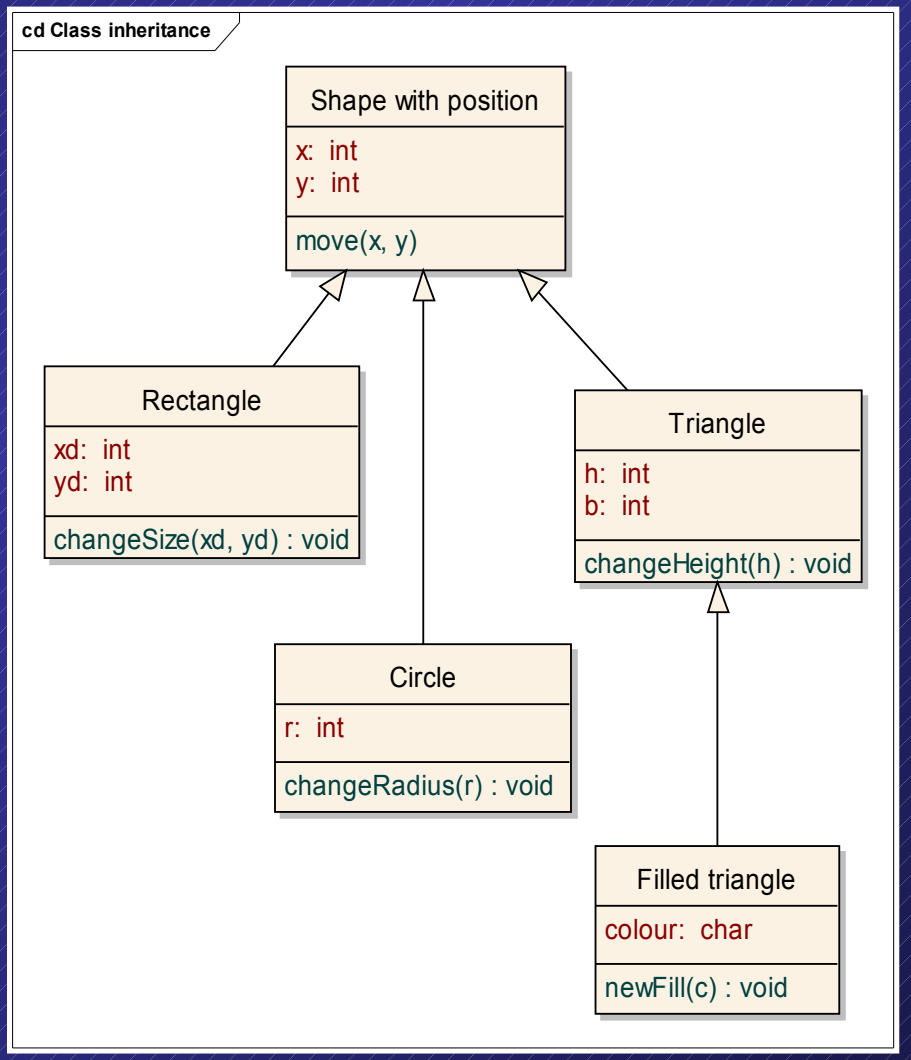


Notation and semantics for inheritance

The inheriting class inherits all the attributes of the ancestor class. It includes the attributes already inherited by that ancestor. It means that classes down in the inheritance hierarchy add new data elements.

The inheriting class inherits also all the operations. Note that these inherited operations define processing involving only the attributes available in the ancestor class.

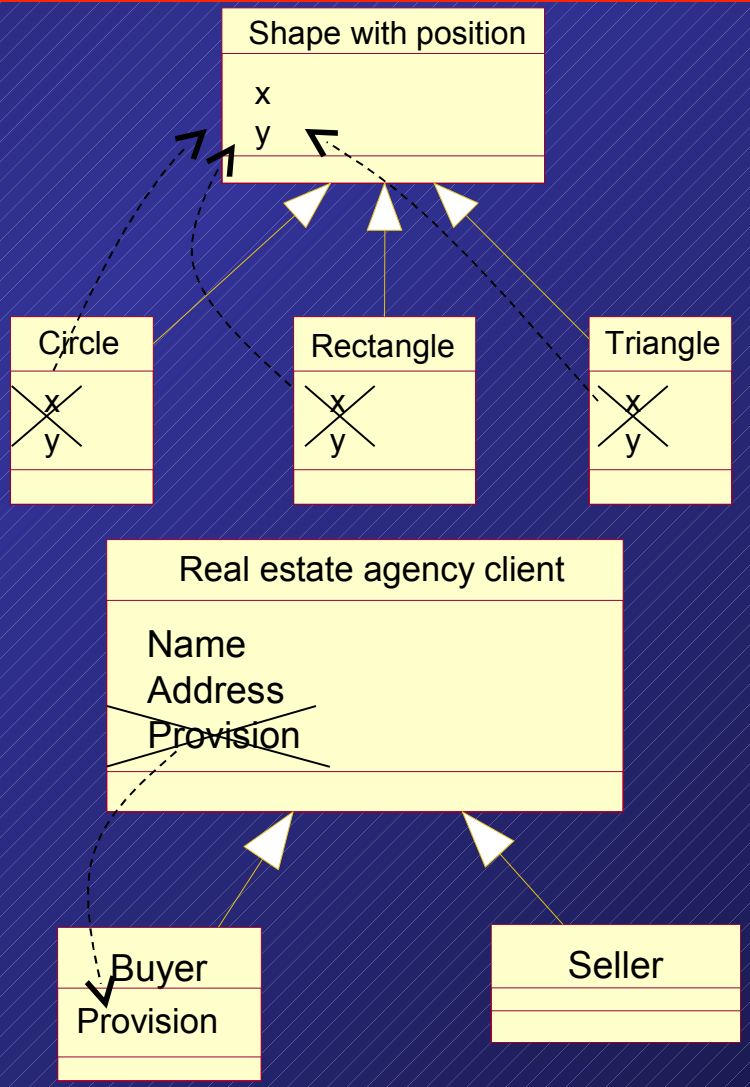
The inherited operations can be overriden in the descendant classes.



Generalization and specialization

When seeking for classes existing in the problem domain we often find similar ones. By determining similar features we can create a more general class. This way we use a technique called generalization.

On the other hand, we often find classes that are too general. Some features of such classes are used only in some objects. This situation can be resolved by using specialization. We then create two (or more) classes that are subclasses of the general one.



Distribution of responsibility

A fundamental rule that should be preserved when building class models is even distribution of responsibility. Such evenly distributed models are easy to comprehend and maintain. An important observation is the "7±2 rule". This rule is associate with the human capability to understand complex systems.

A well structured class model should be divided into reasonably sized packages of classes. Classes should have reasonable number of attributes, operations and associations.

What should we do when a class has too many elements?

- Separate some operations – new control classes emerge.
- Separate some attributes – new data classes emerge.
- Separate some associations – new bounding classes emerge.

Class diagrams should also contain reasonable number of classes.

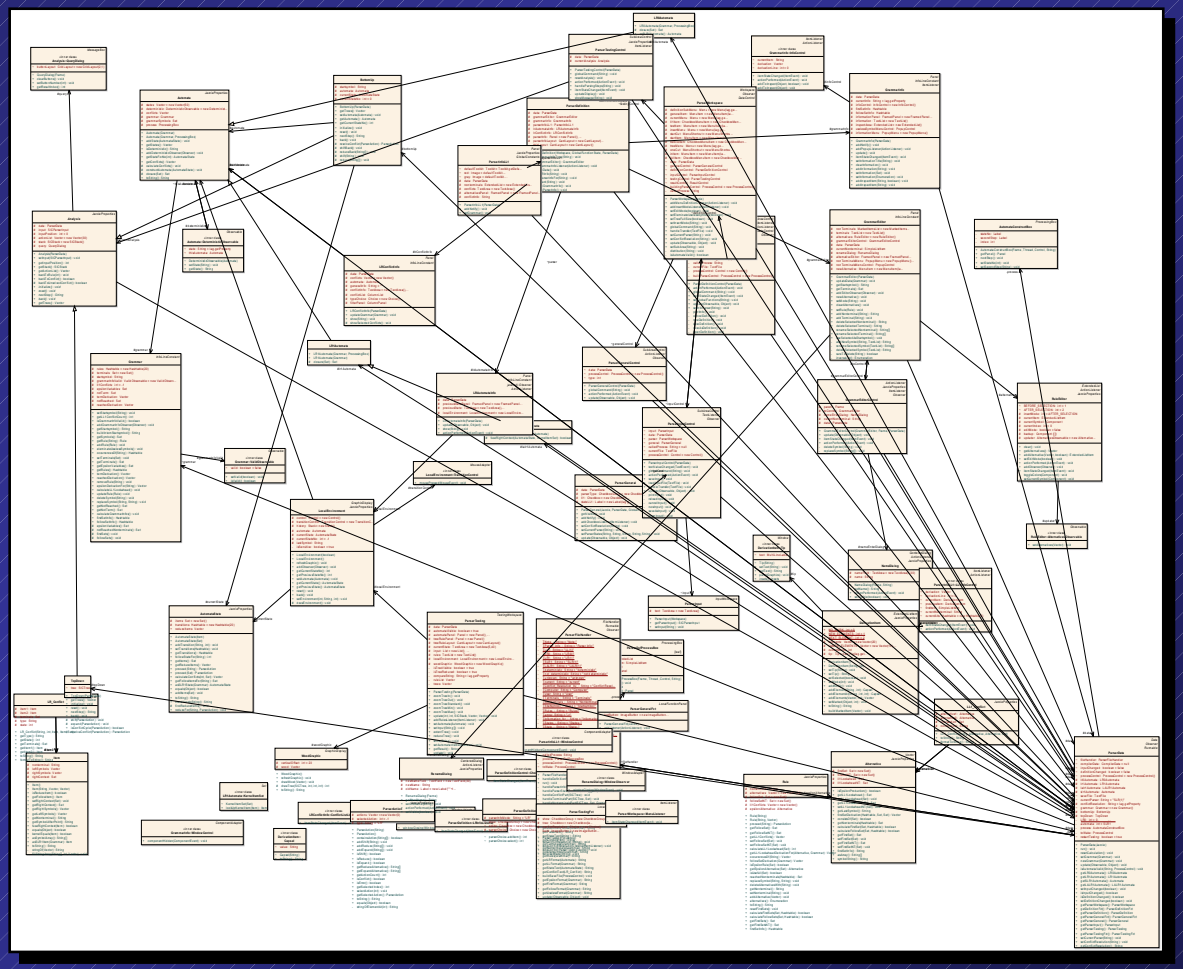
Poorly structured class diagrams – example

Common errors in class diagrams:

- too many classes
- classes with too many features
- classes with too many associations

Solutions:

- divide classes into smaller ones – aggregate when necessary
- divide class diagrams



Solution - metrics

What do „too many”, „reasonable”, „necessary”, etc. really mean?

We can determine that by using object-oriented metrics.

Some examples of OO metrics:

- Methods per Class, Lack of Cohesion Of Methods (LCoM)
- Inheritance Denpendencies, Depth of Inheritance Tree (DIT), Number of Children (NOC), Degree of Reuse of Inheritance Methods
- Coupling between object classes (CBO), (In)Stability
- Polymorphism Factor (PF), Coupling Factor (CF)
- traditional, non-OO: Lines of Code (LoC), Cyclomatic Complexity

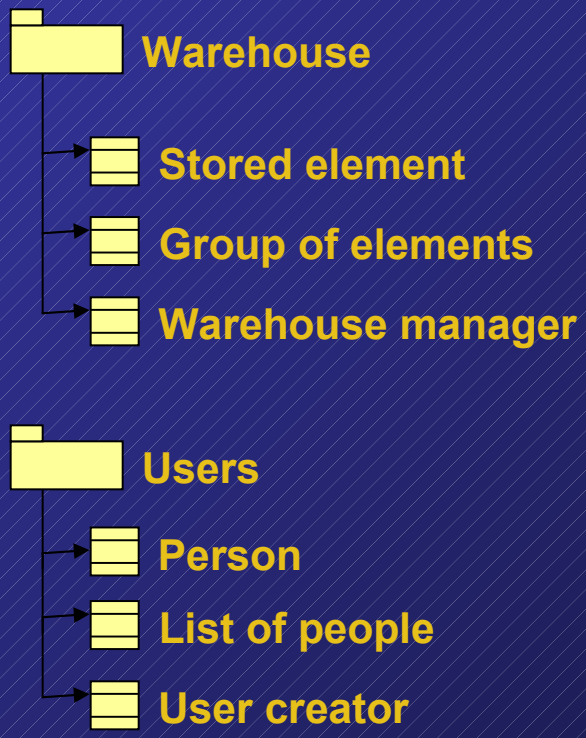
Metrics are inter-related: use of the one most of the time implicates use of the other. Be wary of *average vs. deviation* problem.

Putting order – packages

Class models (and also other models) can contain hundreds or thousands of classes. It is necessary to “pack” them into larger, more manageable units.

UML introduces the notion of a package. Packages can contain many model elements (e.g. classes). Usually, elements in a package are somehow closely related. Packages can be compared to directories in a file system.

Good rule: classes in a package have many relationships (strong coupling); classes between packages have little relationships (weak binding).



Summary – how will WE do it?

Every sentence object in a scenario should have an associated class in a class model.

- Classes should be described with attributes and operations
- Classes should be related through associations, aggregations and compositions.
- Proper inheritance hierarchies should be created whenever necessary.

Class models should be easy to understand by their reader.

- Number of classes shown at once should be reasonable.
- Models should be divided into packages.
- Classes should be reasonably sized.

A reminder: building the noun vocabulary

Discovering notions (classes) in scenarios:

- **Dean** wants to *add new lecture* to course
- **System** *asks* for semester
- **Dean** *enters* the semester
- **System** *asks for data* of the lecture
- **Dean** *enters the data* of the lecture
- **System** *adds the lecture* to the list of lectures

- **System** *assigns* the teacher to the lecture

lecture

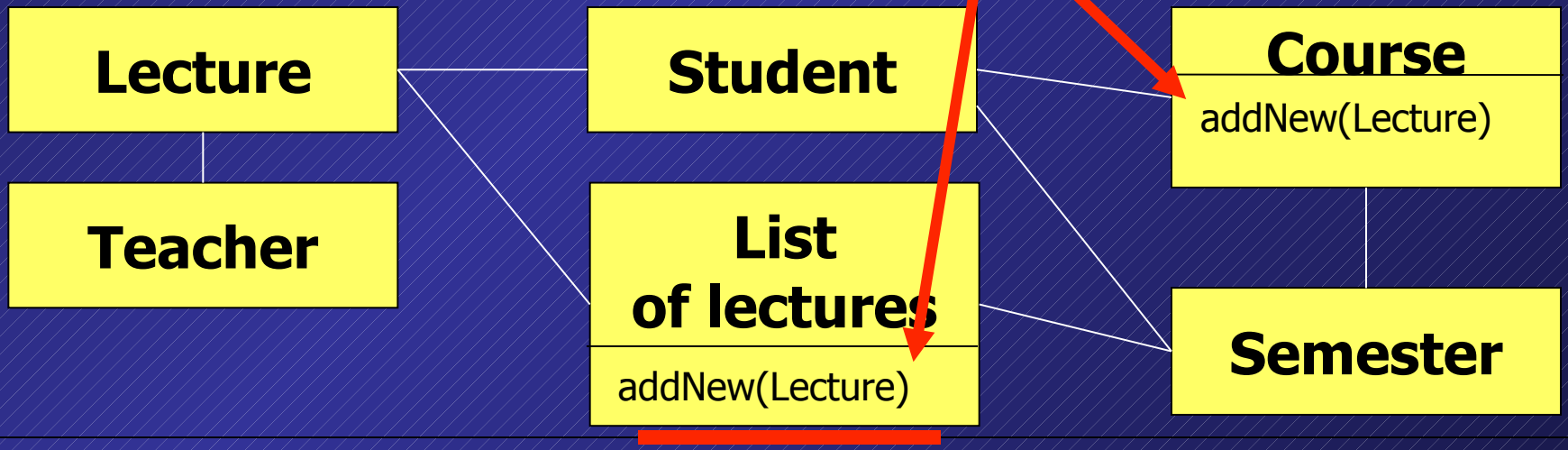
the student

Reminder: using verbs

Discovering verbs (operations) in scenarios:

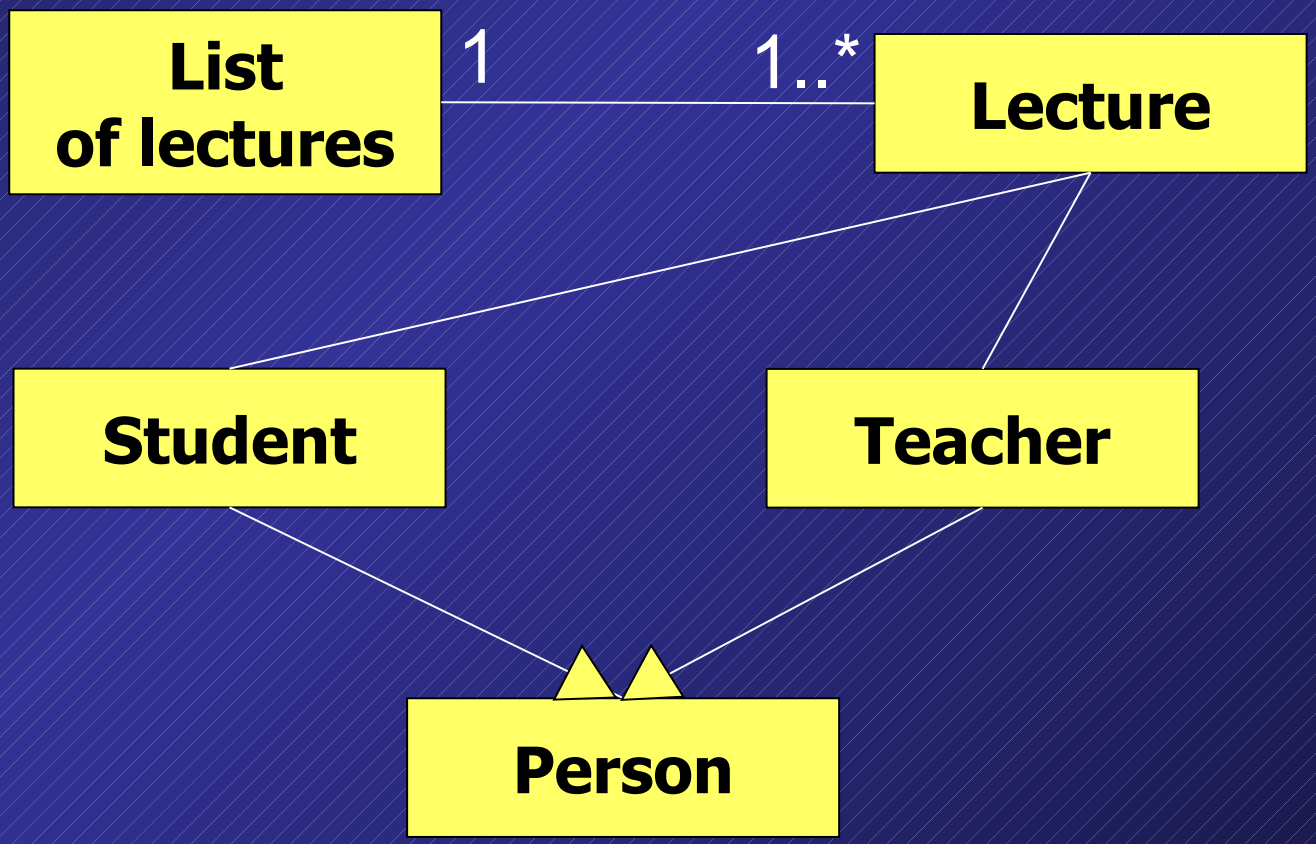
- **Dean** *wants to add* new lecture to course
- **System** *asks* for semester
- **Dean** *enters* the semester
- **System** *asks* for data of the lecture
- **Dean** *enters* the data of the lecture
- **System** *adds* the lecture to the list of lectures

add new lecture



Refining the class model

Specifying hierarchies, multiplicities, role names, etc.



Summary – how will WE do it?

Every sentence object in a scenario should have an associated class in a class model.

- Classes should be described with attributes and operations
- Classes should be related through associations, aggregations and compositions.
- Proper inheritance hierarchies should be created whenever necessary.

Class models should be easy to understand by their reader.

- Number of classes shown at once should be reasonable.
- Models should be divided into packages.
- Classes should be reasonably sized.