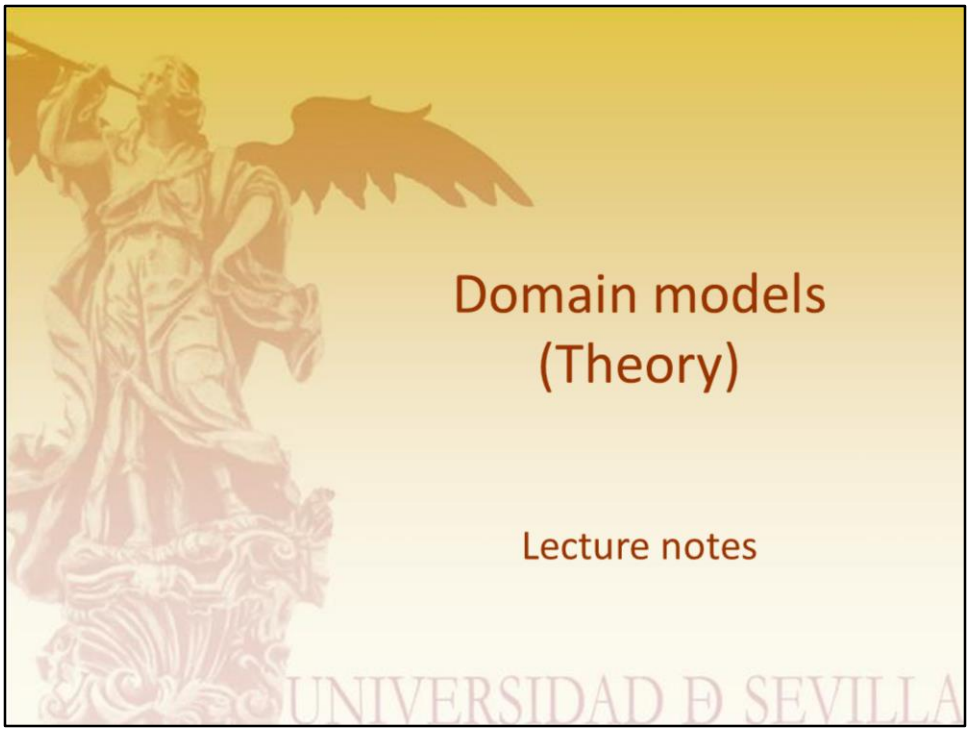


Welcome to this lecture! Today, our goal's to delve into the theory of domain models.

--

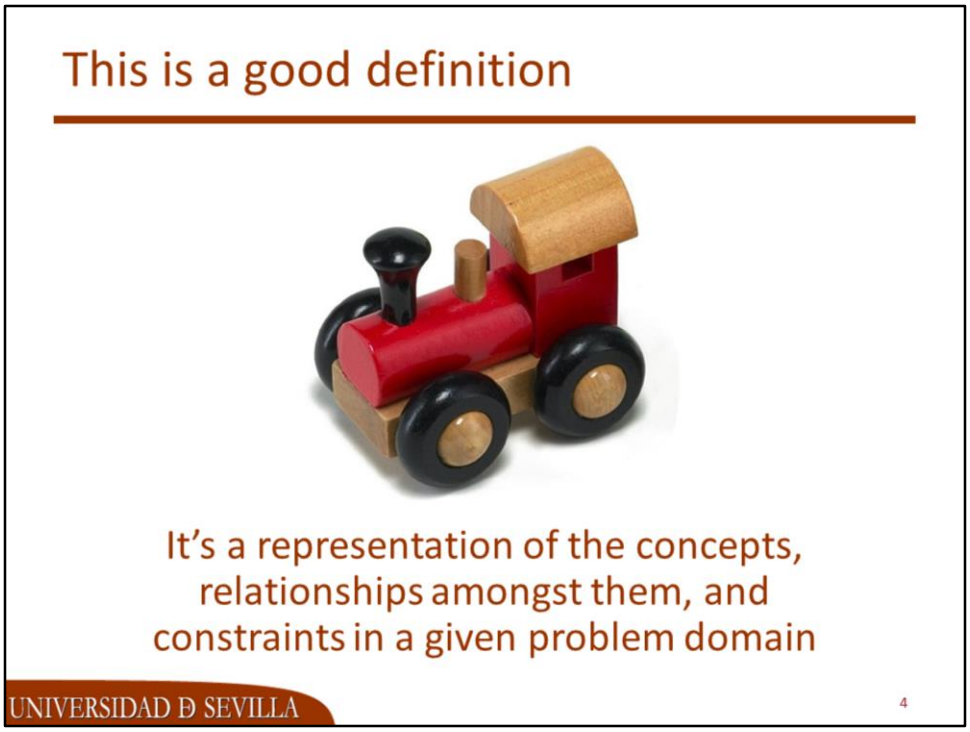
Important note!

What's a domain model?

UNIVERSIDAD Ð SEVILLA

3

It's then a good idea that you try to produce a definition before going on. Please, make a point of defining what a domain model is before peeking at the following slides.



This is our definition: it's a representation of the concepts, relationships amongst them and constraints in a given problem domain. This might sound very abstract, but it isn't difficult to understand. A problem domain, or just domain, is a term that is commonly used to refer to a problem, to a project, or to an area of interest. For instance, if you think of a domain called "Medical Record System", then you're likely to think of concepts like "Doctor", "Nurse", "Patient", or "Medicine", and relationships like "a Doctor prescribes a Medicine to a Patient", "a Nurse looks after a Patient", and the like; if you think of a domain called "Credit Scoring System", then you're likely to think of concepts like "Customer", "Clerk", "Loan", "Mortgage", or "Financial Profile", and relationships like "a Customer asks for a Mortgage", "a Mortgage was approved by a Clerk", and the like.

How are they devised?

UNIVERSIDAD Ð SEVILLA

5

How do you think a domain model is devised? Please, try to produce an answer by yourself before peeking at the following slides.

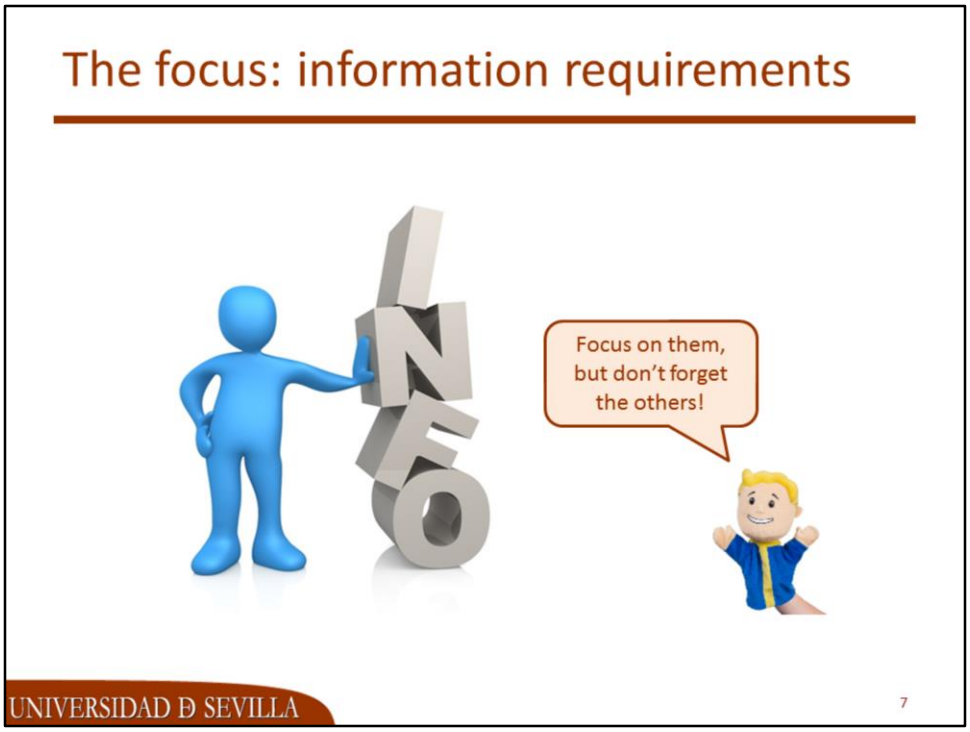
Starting point: your requirements

- Ilities
  - The non-functional aspects of a system
- Information
  - The data that a system has to process
- Use cases
  - The tasks that the actors can perform

UNIVERSIDAD Đ SEVILLA

6

The starting point are your requirements, and we already know that they are commonly organised into three groups, namely: ilities (aka non-functional, extra-functional, quality, system-wide, or cross-cutting requirements), information requirements, and use cases. In a previous lesson, we provided an insight into each category and justified why, contrarily to the common practice, we put ilities at the top.



Our focus to devise our domain models will be on the information requirements. But wait! Having a focus doesn't mean we can forget about the other requirements. You need to have a holistic view of the requirements and you can't forget the ilities and the functional requirements. Although they are not the focus, you need to know them very well before you start devising a domain model.

Step 1: create a conceptual model

UNIVERSIDAD Ð SEVILLA

8

The first step is to create a conceptual model, which is a UML-based representation of your information requirements that is appropriate for your developers. You can use your requirements templates to talk to your customer; they are written in plain natural language and they are very useful to talk to a person who is not a specialist in Software Engineering, but in the problem domain. Unfortunately, natural language isn't compact or graphical enough for developers. Developers need a view of the information requirements that allows them to quickly focus on a specific part of the requirements without having to go through a long document in natural language. Take a look at the sketch in this slide: we're pretty sure that you understand this sketch and identify the pieces and their relationships; it's a conceptual model. The conceptual models on which we're going to work are obviously very different and more complex, but they play the same role.

Step 2: create a UML domain model

UNIVERSIDAD Ð SEVILLA

9

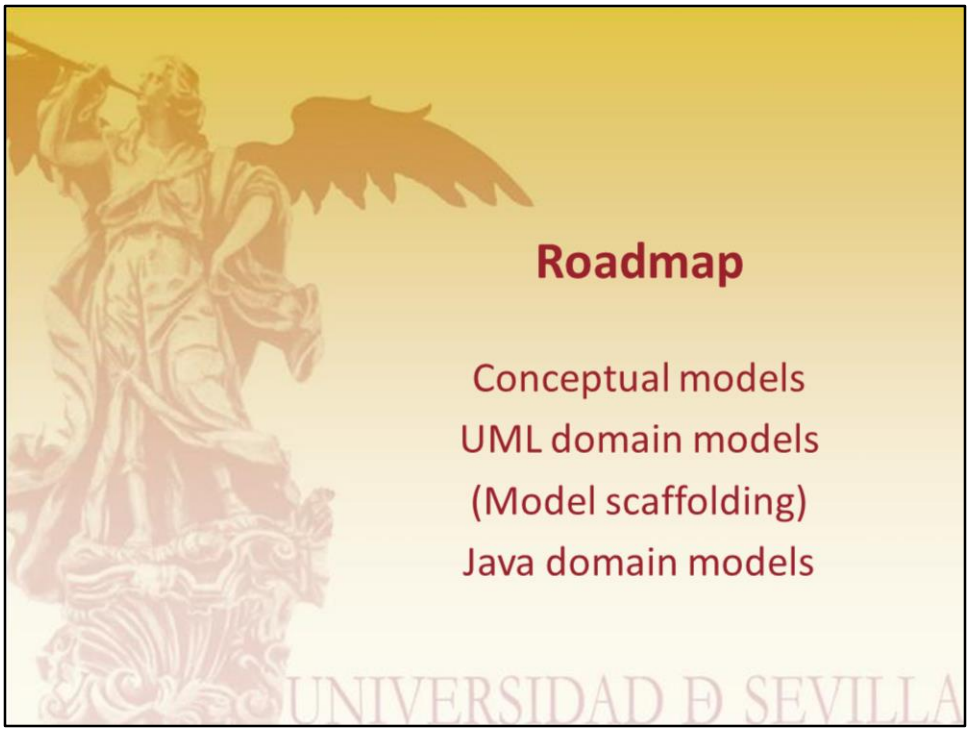
The conceptual model is very close to the information requirements, that is: to your customer's parlance. An advanced customer might understand a conceptual model with the help of a UML expert. Unfortunately, this isn't enough for a developer. A typical developer needs to know about user accounts, association navigability, wrapper versus primitive types, how to deal with overlapping specialisation, which designs are efficient and which are not, and so on. Obviously, this is not the kind of things you can discuss with your customer, but they are absolutely necessary to produce an implementation of the conceptual model. The second step consists of applying a series of transformations to our conceptual model to produce a so-called UML domain model that has enough information for a developer to produce an implementation. It's like in this picture: it's very similar to the one in the previous slide, but adds a number of details that builders need to know before starting to build a house.

Step 3: create a Java domain model

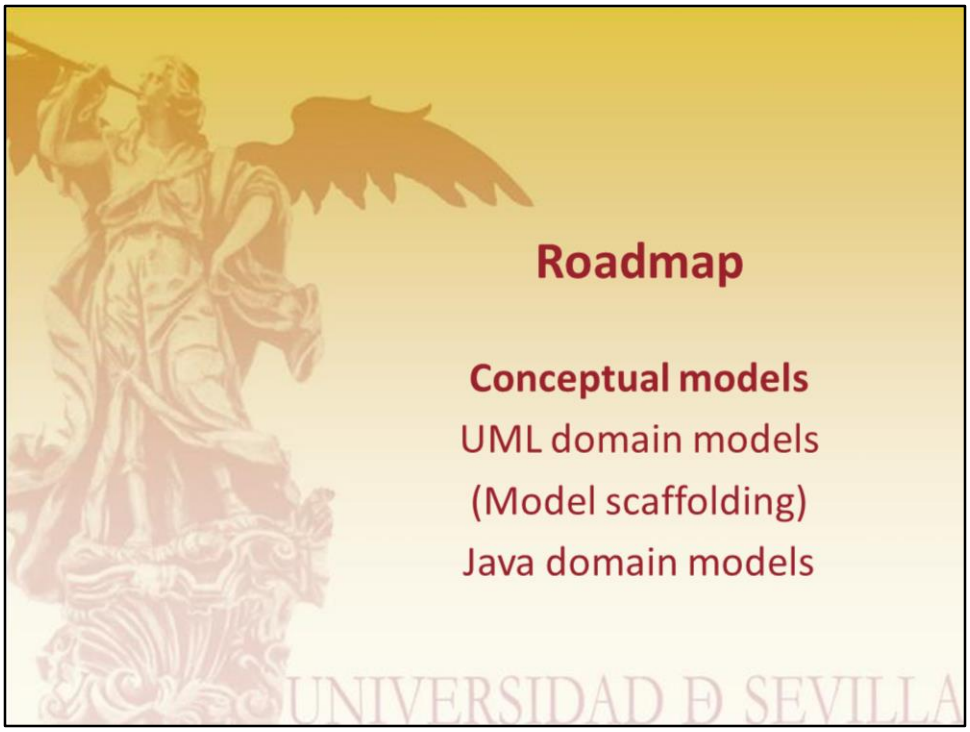
UNIVERSIDAD Ð SEVILLA

10

Since a UML domain model has every tiny detail required to implement it, that means that it's time to start coding. We'll usually refer to the results of the implementation as the Java domain model. Please, note that producing a requirements elicitation document requires quite a lot of natural intelligence; that's why requirements engineers exist and are paid so well. Producing a conceptual model and a UML domain model requires some natural intelligence, but the process is quite systematic in general; that's why designers exists and are paid not too bad. Producing an implementation from a domain model is quite a simple task; that's why there exist hordes of programmers who are not paid very well.



By now, we should have a good understanding of our goals in this lesson. It's now time to start walking our roadmap: first, we will report on conceptual models, then on how to produce a UML domain model building on a series of transformations, and, finally, we'll delve into how to produce a Java implementation of the UML domain model. After presenting UML domain models and before delving into the details of Java domain models, we'll pay a little attention to a concept known as model scaffolding. Please, read on.



Let's start with conceptual models.

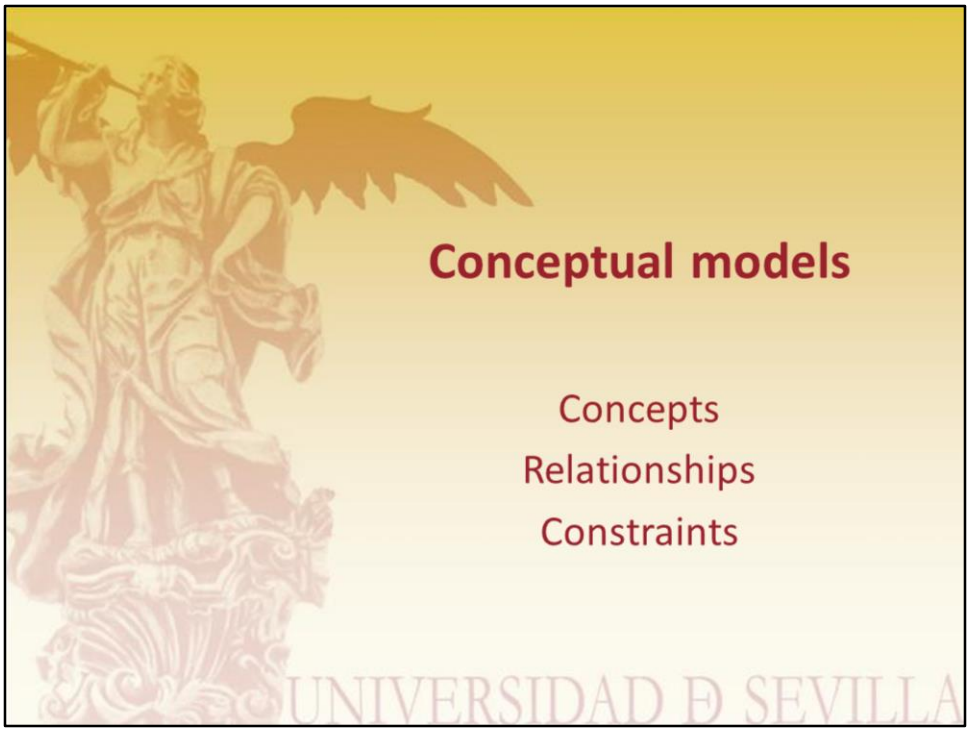
Conceptual models

It is a representation of your information requirements (wherever they are specified) using a developer-oriented language

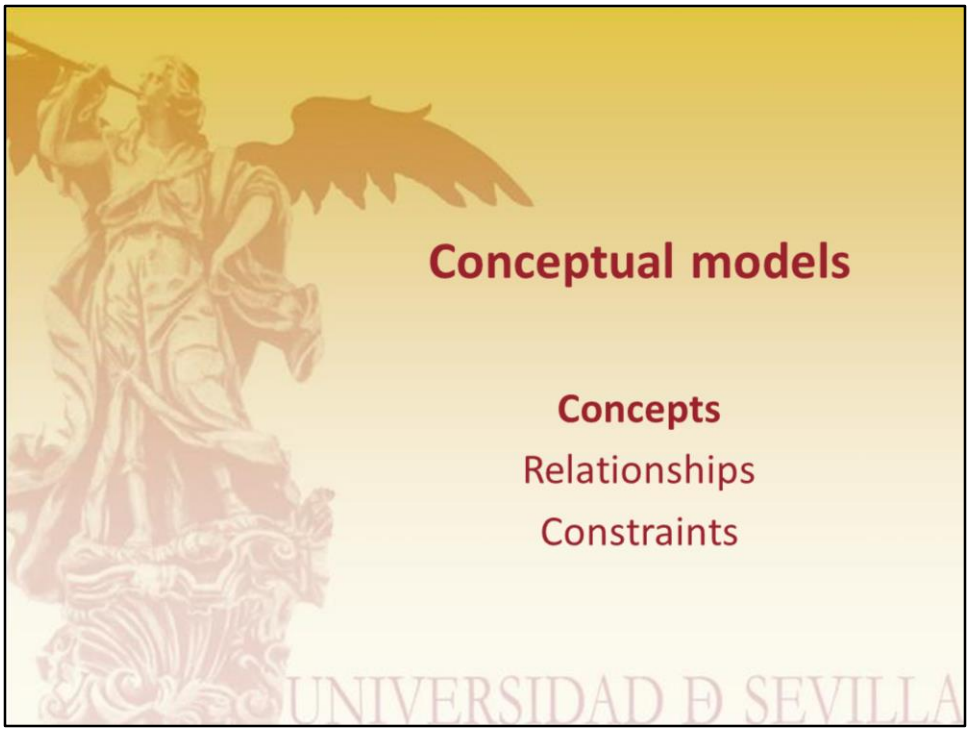
UNIVERSIDAD Ð SEVILLA

13

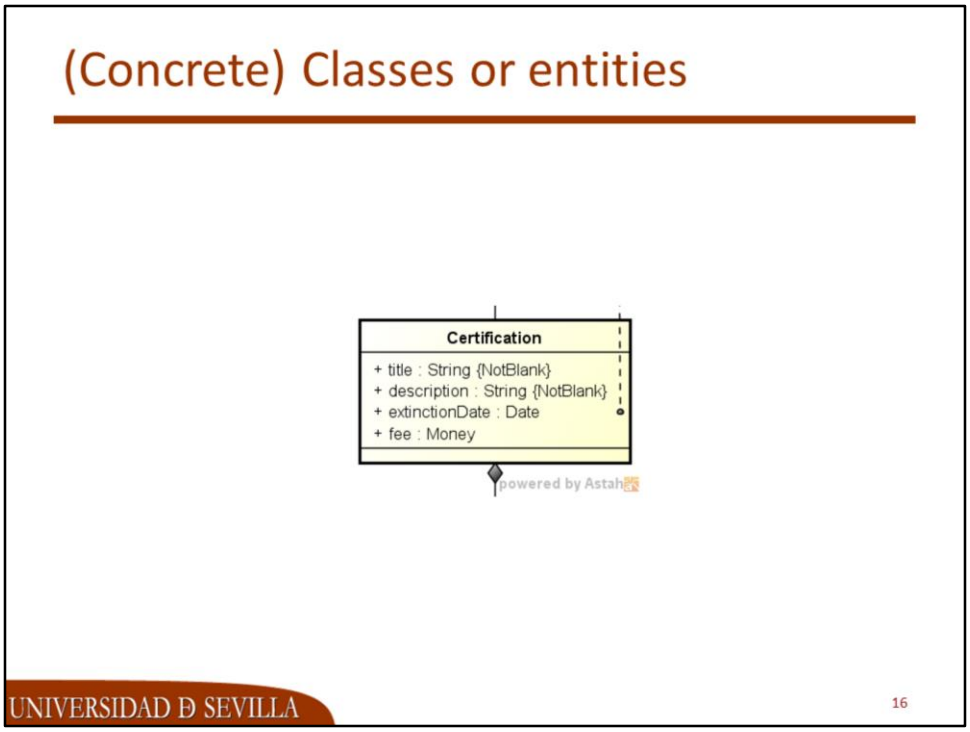
Conceptual models focus on representing information requirements using a formal language that is appropriate for developers. Note that we emphasise information requirements, independently from where they are specified in your requirements elicitation document; please, bear in mind that most information requirements are in the section entitled “Information Requirements”, but some of them derive from the “Functional Requirements” section and others from the “Non-Functional Requirements” section; recall that you must focus on the first section, but keep an eye on the others.



Ok, let's now delve into the details of conceptual models. We'll review concepts, relationships, and constraints.

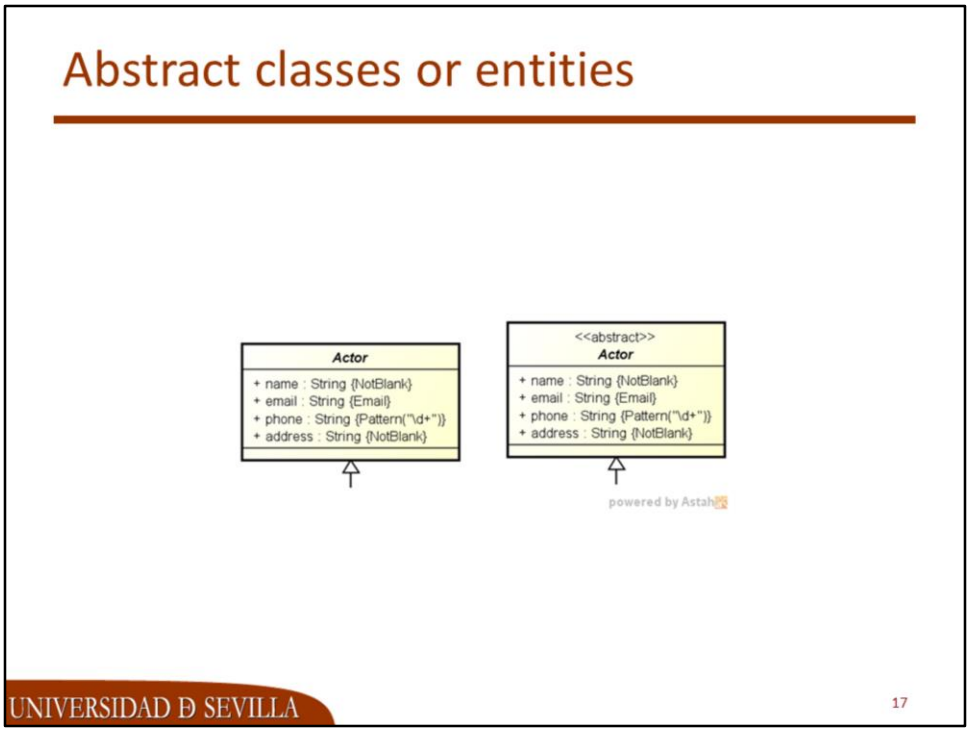


Let’s start with the concepts.

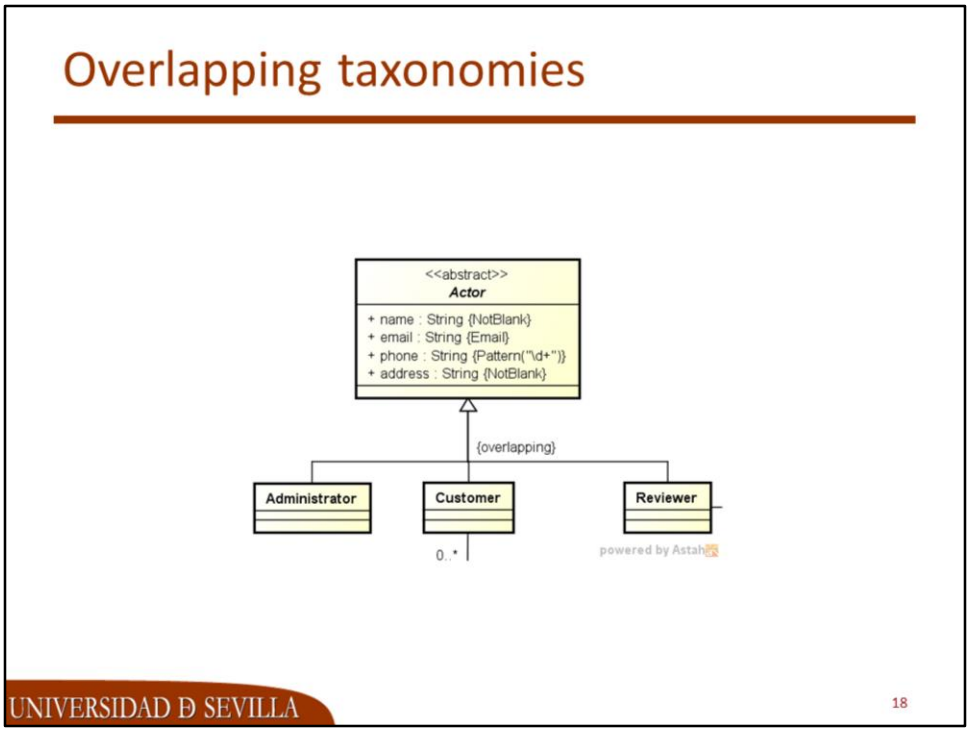


A concept is a term that commonly refers to the entities or objects in a given problem domain. For instance, if you think of a "Certification System" domain, then you're likely to think of concepts like "Certification", "Customer", "Announcement", or "Exam". They are represented by means of classes, aka entities. Classes must have a name, which is usually a singular noun, a collection of attributes, which represent simple features (strings, integers, dates, and so on), and may also have methods, but this isn't usual at all in a conceptual model. By default, when we refer to "classes", we implicitly assume that they are concrete classes, that is: classes that represent actual objects in the domain, i.e., classes that can be instantiated. For instance, in our example, we show a class called "Certification"; it's a concrete class, which means that there are objects in the domain that fit within this class; there are things to which our customer refers to as certifications, and these are actual objects.

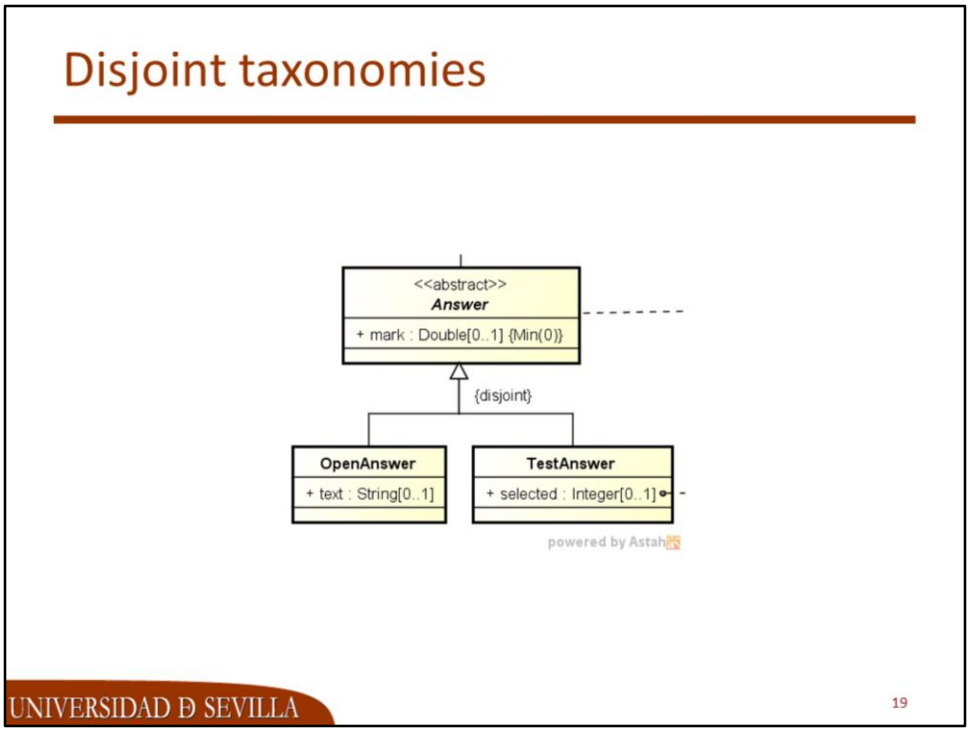
NOTE: from now on, we'll use the "Certification System" domain as a running example.



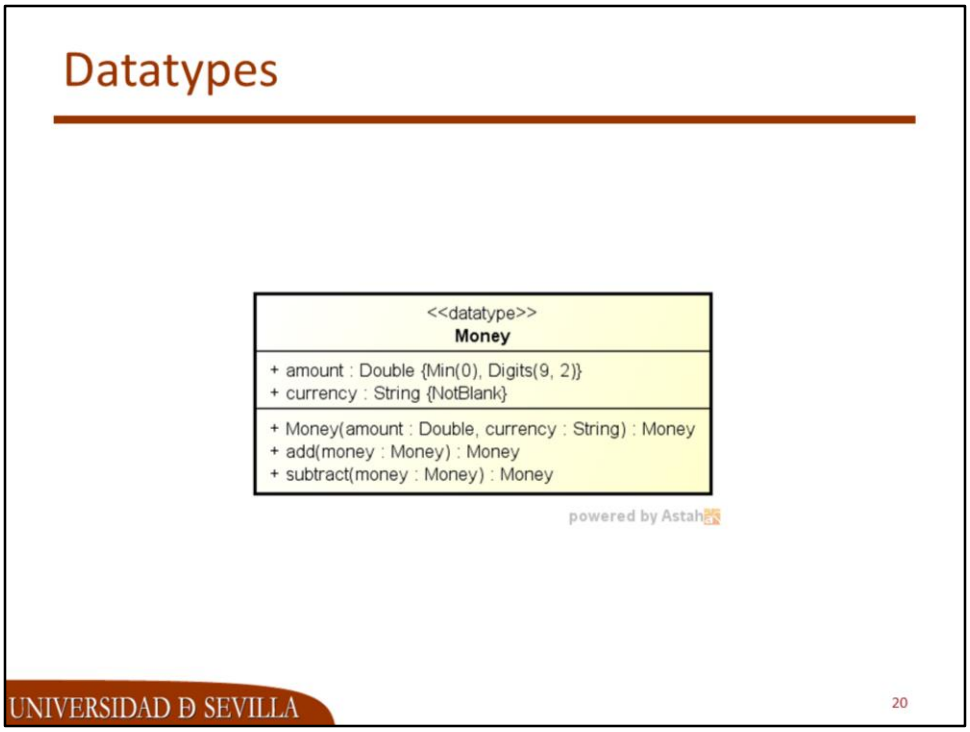
Contrarily, there are abstract classes, aka abstract entities, that are intended to represent the root of a taxonomy. For instance, in our example, we may have administrators, customers, reviewers, and other people to which our customer refers collectively to as the actors. Obviously, it's a good idea to introduce a class in your conceptual model to represent actors. The difference with class "Certification" is that class "Actor" is a simple classifier; it provides a little meaning, but not enough to discern which person you refer to. You need additional concrete classes to represent administrators, customers, or reviewers. Please, note that abstract classes are represented in UML using a slanted font to write their names or with stereotype "<<abstract>>". The former notation is sometimes confusing, since it is not commonly easy to realise the slanted font; that's why we usually lean towards using the stereotype.



Ok, let's go ahead. We introduced the idea previously, but it's now time to delve into the details. We told you that abstract classes help introduce taxonomies. They are abstract classifiers and thus represent is-a relationships amongst a number of classes. Note that taxonomies are overlapping by default in UML, which means that an object may belong to several classes at a time. For instance, assume that John Doe is an actor; what specific kind of actor? Ok, he can be an administrator, but notice that nothing prevents John Doe from being a customer, as well. Assume that you're John Doe and that you work for Acme Certifications, Inc. as an administrator; why can't you be a customer of Acme Certifications, Inc.?



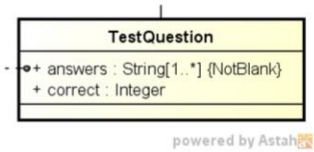
Obviously, not every taxonomy is overlapping. For instance, think of the exam papers that a customer produces when he or she sits for a certification exam in a given announcement. If we assume that questions can be either open questions or test questions, then it makes sense to think of “Answer” as an abstract class that specialises into either “OpenAnswer” or “TestAnswer”. Obviously, an answer must be either an open or a test answer, but not both.



Classes aren't difficult to understand, are they? Let's introduce a new modelling artefact: datatypes. A datatype is a concept that represents a value object, aka object value or embeddable object. Such objects share a common feature: they don't have an identity; more than that: they don't need an identity. Value objects represent abstract concepts like integer or real numbers, strings of characters, calendar dates, or amounts of money. Can you find a single difference between integer "2" and integer "2"? Can you find a difference between date "October 4, 2013" and "October 4, 2013? Can you find a difference between the amount of money "99.95 €" and the amount of money "99.95 €"? We assume that your answer to the previous questions is no, but before coming to additional conclusions, please, keep reading.

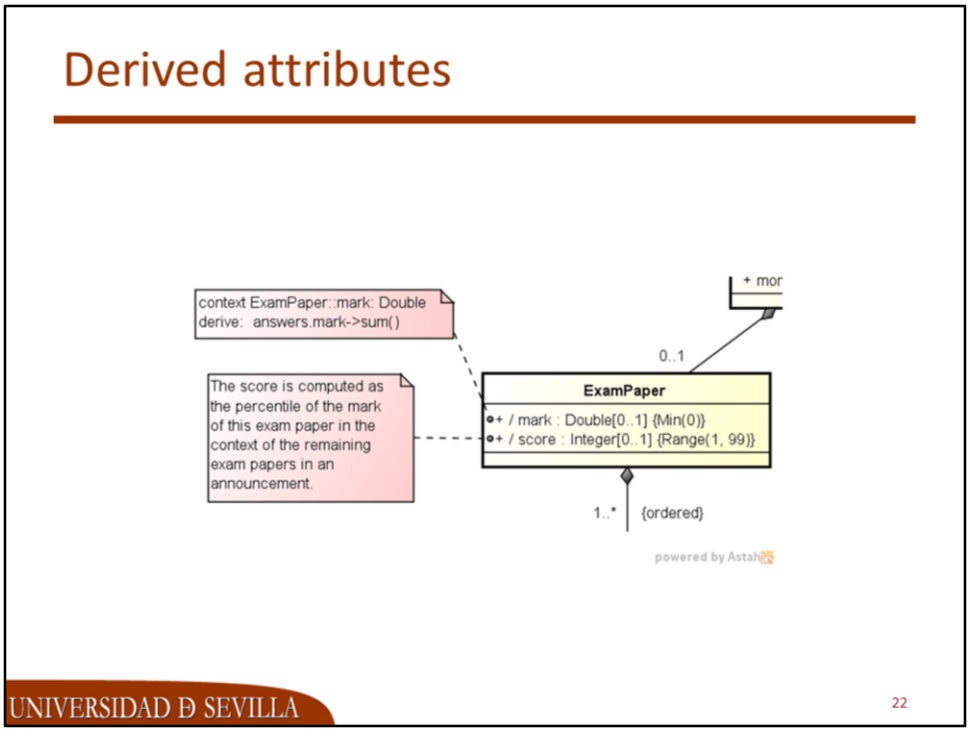
NOTE: please, note that it's very common that datatypes have methods, which is something very unlikely to happen with entities. The reason is that datatypes commonly encapsulate some functionality to work with their values which fits very well this kind of models. In this example, the functionality associated with "Money" objects is to add or subtract money.

Attribute multiplicity

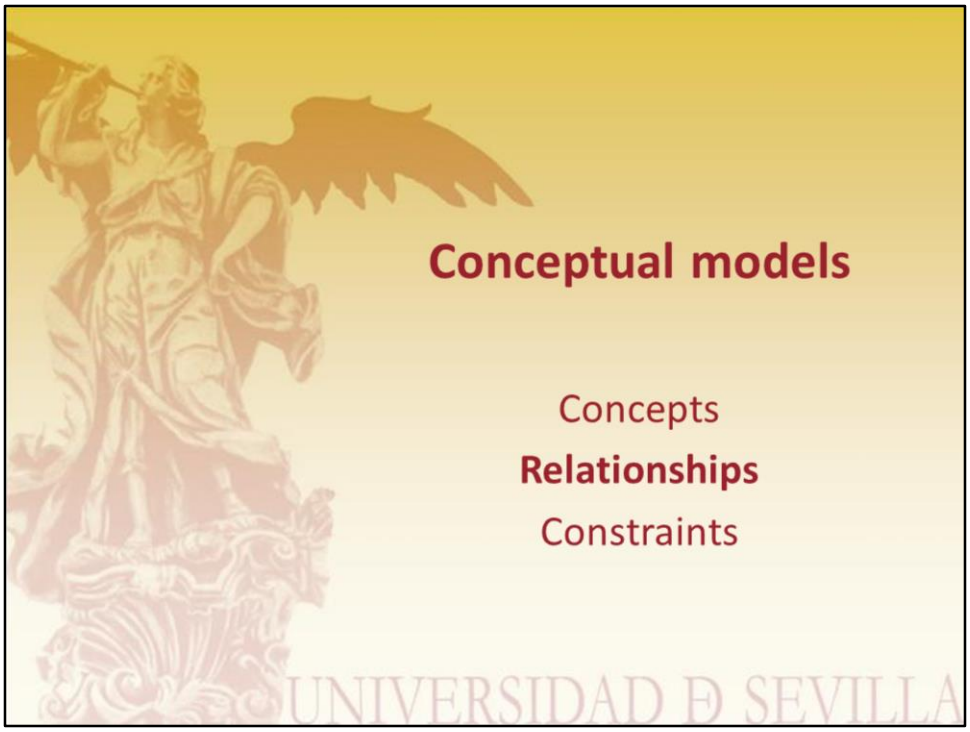


- Obligatory (default):
  - "[1]"
- Optional:
  - "[0..1]" or "[?]"
- Multiple:
  - "[1..*]" or "[+]"
- Optional multiple:
  - "[0..*]" or "[*]"

A few slides ago, we mentioned that a class or a datatype may have attributes that represent the features of the actual objects that are represented by that class or datatype. Please, note that the type of an attribute in a conceptual model must be datatype and that an attribute commonly stores a single instance of that datatype; there are, however, cases, in which an attribute needs to have optional, multiple, or optional-multiple multiplicity (some people refer to multiplicity as cardinality; it's the same, don't worry). For instance, let's analyse a class called "TestQuestion" that, obviously, represents test questions in an exam. It makes sense to think of that class as having an attribute called "answers" that represents the answers shown to a customer, and an integer that indicates which of them is correct. Note that the "answers" attribute has multiple multiplicity.

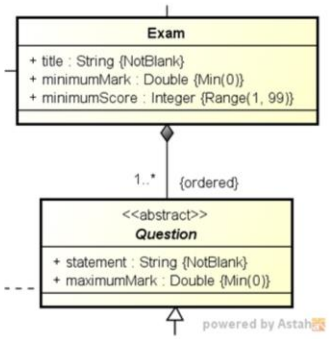


There are a few cases in which it makes sense to have a derived attribute in a class. Such an attribute is unnecessary, since it represents information that can be computed from other attributes in the conceptual model, but it makes sense to represent it because of simplicity. In the example in this slide, we present a class in our conceptual model that represents exam papers. It makes sense to add an attribute called “mark” to represent the mark of the exam paper, and an attribute called “score” to represent its score. The mark of an exam paper is something like 3 points or 9 points; the score indicates the percentile of a mark out of the set of marks in the exam papers produced in an announcement (please, review your lecture notes on statistics if you don’t know what a percentile is). Strictly speaking, none of these attributes is necessary, but we think that they make sense because when our customer talks about the exam papers, he usually refers to their marks and scores. Note that a derived attribute has to be defined by means of a note, which can be written in OCL or natural language, whatever best suits your needs.

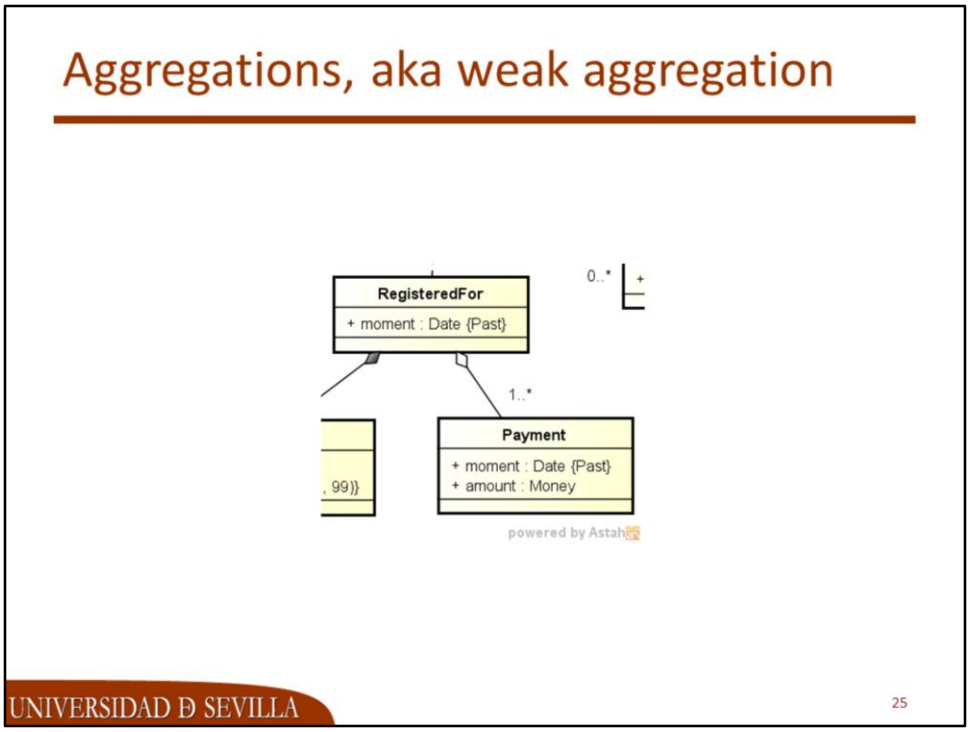


In the previous section, we've analysed how to represent concepts. It's now time to analyse how to represent the relationships amongst them.

Compositions, aka strong aggregation

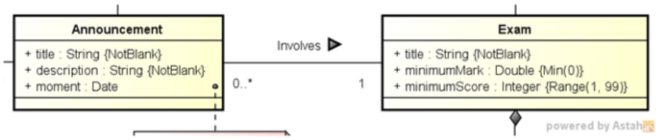


Let's start with compositions, which are also known as strong aggregations. A composition represents a part-of relationship between a whole class and its part classes. For instance, in this slide, we present a part of our running model in which we represent that an exam is composed of questions. The decision on whether a relationship is a composition or not is usually quite confusing to our students. There's a very simple rule of thumb that you should use all the time: can the part exists independently from the whole? If the answer is no, then you're likely to be examining a composition. For instance, does it make sense that a question exists without the corresponding exam? In this example, the answer is no, so this is a strong indication that the relationship between classes "Exam" and "Question" is a composition. Note that it's very important to emphasise that this is the answer in the context of this project. Another customer might have a database of questions out of which the exams are composed; in that case, a question may exist in the database independently from whether it's a part of an exam or not. You always have to check your requirements before making a decision; and, if you are in doubt, please, talk to your requirements engineer.

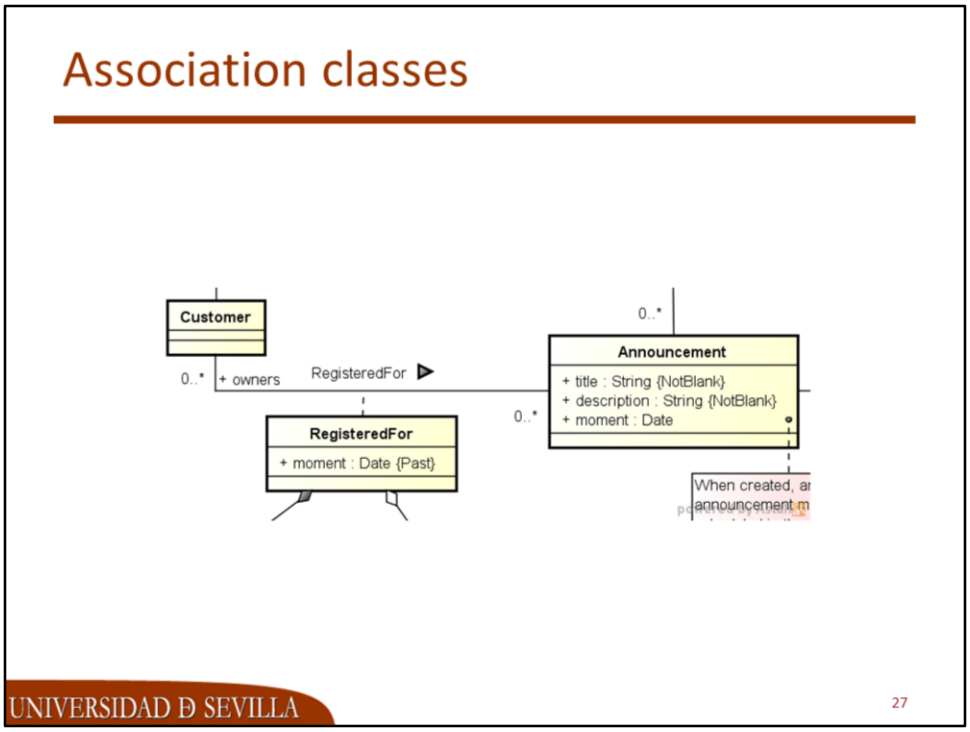


Aggregations are intimately related to compositions; they are also called weak aggregations. They also represent part-of relationships between a whole class and its part class; the difference is that the parts can exists independently from the whole; that is, a part may exist even thought it doesn't belong to any whole. In our running example, think for instance of a requirement that reads: "Payments are assigned to registrations by administrators once they are checked". This requirement means that the system must allow to register a payment that is not associated to any registration; it'll be associated when an accountant certifies that the payment was made and is correct. Think of an additional requirement: "The fee of a registration may be cleared by means of one or several payments". This requirement means that a user may decide to pay a registration by means of a single money transfer; but another one might decide to use a transfer and a cheque; and another customer might decide to make a single payment to pay two different registration fees. If such requirements need to be modelled, then we might use the UML diagram in this slide.

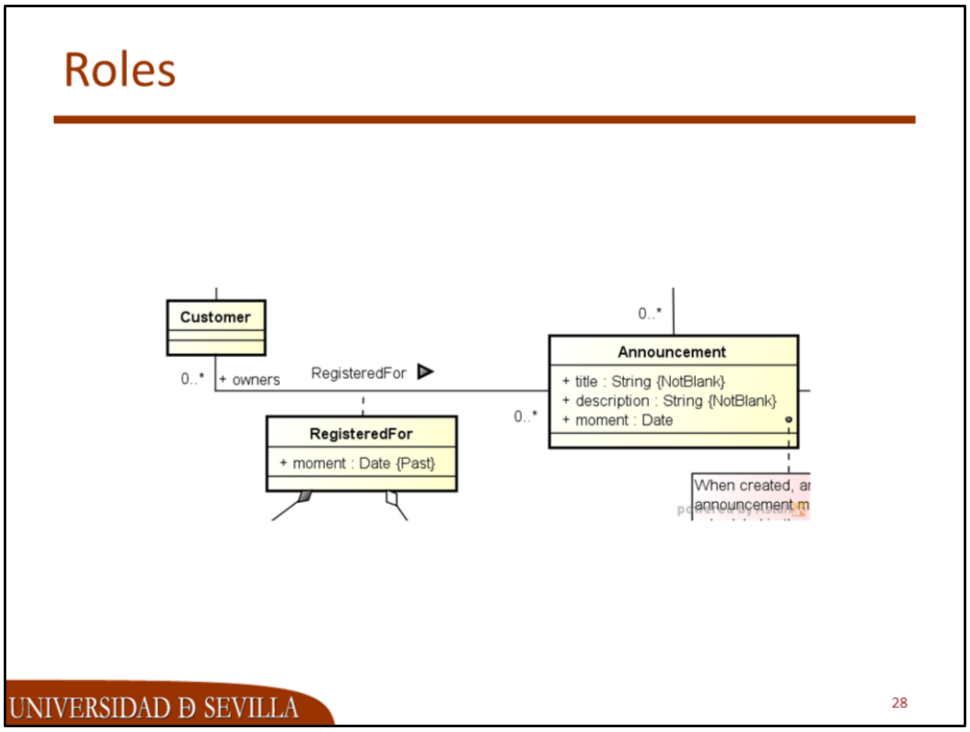
Associations



Let's now delve into associations. An association represents any kind of relationship between two classes, provided it isn't a composition or an aggregation. The life cycle of the associated objects is absolutely independent from each other. For instance, in this slide, we present a part of our running model in which we present classes "Announcement" and "Exam". An announcement is a public note in which Acme Certifications, Inc. announces that an examination to attain a certification's going to be organised; that announcement, obviously, involves an exam, which, in turn, is composed of questions. Note that it is very important that associations have a meaningful name that describes its semantics. Note that it's common to draw an arrow head next to the name of an association; it indicates how the association is read naturally. For instance, in this case, the association may be read as follows: "an announcement involves an exam". You obviously can read it in the inverse direction using passive voice: "an exam is involved into an arbitrary number of announcements, possible none".

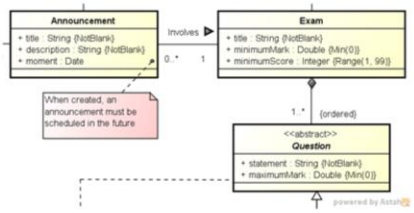


There's a particular kind of classes that is called association classes. They represent the attributes that are inherent to an association. Please, realise that associations are first-class citizens regarding conceptual modelling; that means that they are not sub-ordinated to classes, but are first-level constructs of UML. That implies that you may need some attributes to describe the features of an association and these attributes are provided by an association class. In this slide, we present an example: a customer has registered for an announcement, which means that he or she is willing to take an exam to attain a certification. The registration itself is modelled as an association between classes "Customer" and "Announcement", and the moment when the registration takes place is modelled as an attribute in the association class called "RegisteredFor". We're pretty sure that you would come to a different model; maybe a model in which class "Customer" has an association to a class called "Registration" and then class "Registration" has an additional association to class "Announcement". Soon you'll realise that this model isn't correct unless you add additional constraints. We'll return to this problem later, when we examine UML domain models. Please, it's important that you get accustomed to using association classes because they're very common in practice.



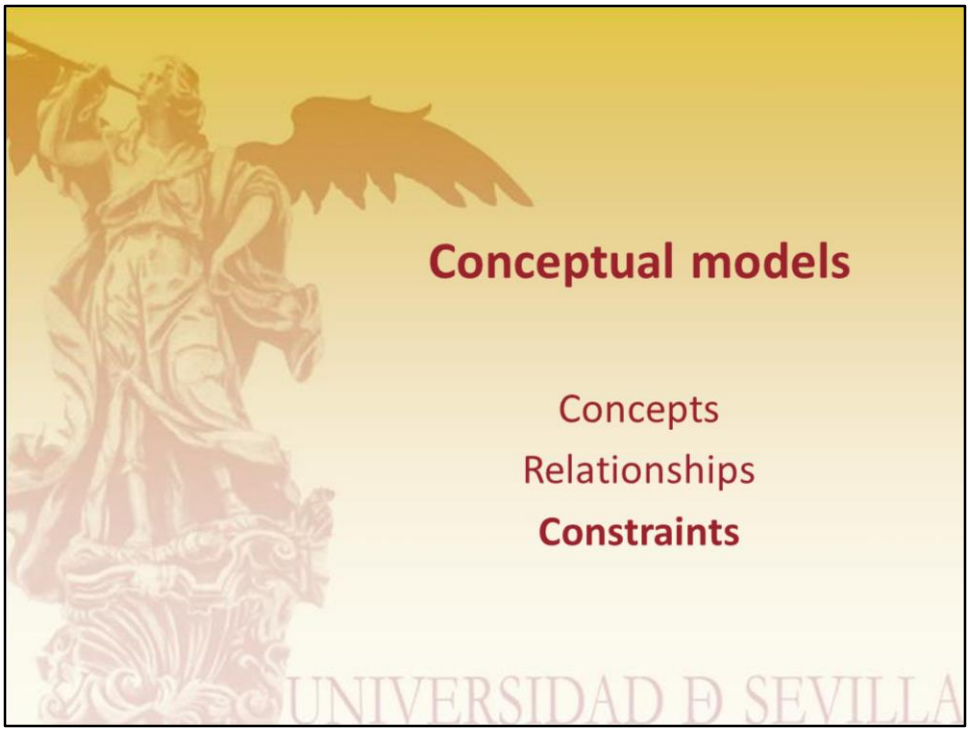
Regarding associations, it's interesting to note that you may attach roles to their ends. Note that every association end has a default role that is the lowercase name of the corresponding class (in plural if necessary). More often than not, explicit roles work very well; you should override them only if your customer changes his or her vocabulary when he or she refers to a concept using two different names: a name to refer to the concept itself and an additional name to refer to it in the context of an association. Please, make sure that you understand that this is not very frequent, but it happens. For instance, when Acme Certifications, Inc. refers to their customers, they use term "Customer", which is the reason why we use this term in our sample model; however, when they refer to their customers in the context of a registration, they use term "owner"; they say things like "the owner of a registration that wasn't paid in due date …" instead of "the customer of a registration that wasn't paid in due date …". This justifies that you override the default role in the customer end of association "RegisteredFor"; the other end can keep its default name "announcement". Another case in which overriding a default role name is a must is recursive associations. Think of a project management problem domain in which a worker can supervise other workers; it's not difficult to see that you need an association called "Supervises" that relates an instance of class "Worker" to some other instances of that class; in such cases, you need to override the default role names of the resulting model or, otherwise, it'll be a mess.

Multiplicities

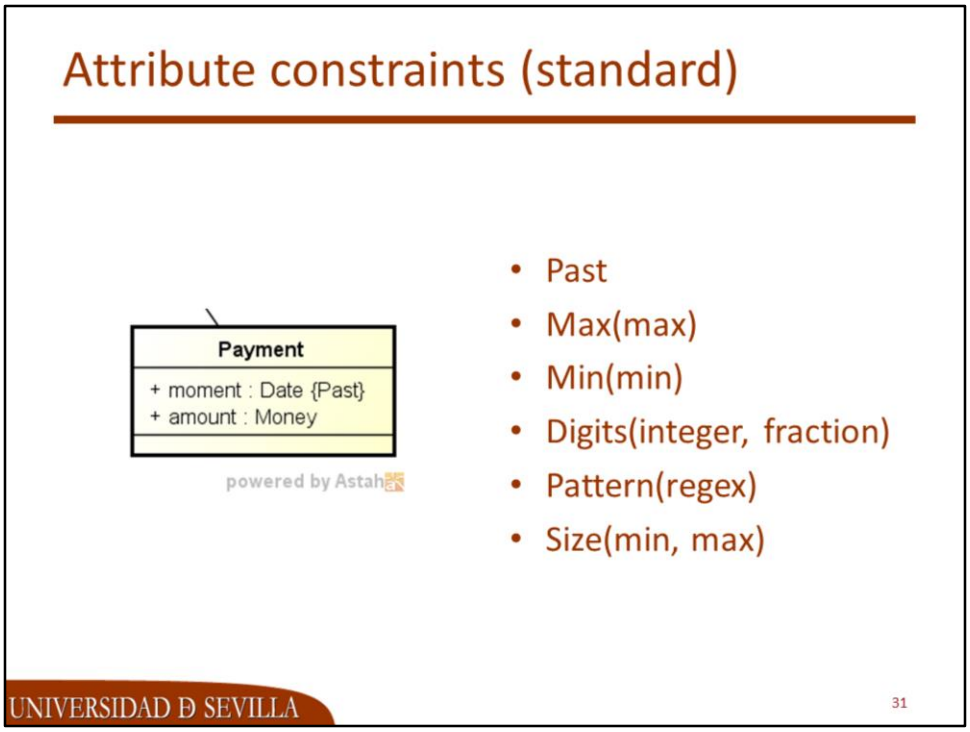


- Obligatory:
  - "1"
- Optional:
  - "0..1" or "?"
- Multiple:
  - "1..*" or "+"
- Optional multiple:
  - "0..*" or "*" (default)

Association ends also have multiplicities: obligatory, optional, multiple, or optional multiple. For instance, in this piece of model, we show that an announcement involves exactly one exam, but an exam may be involved in an arbitrary number of announcements (possibly none); similarly, we specify that an exam is composed of at least one question.



Concepts and relationships are the core of a conceptual model, but it's very likely that you need constraints to complete it.

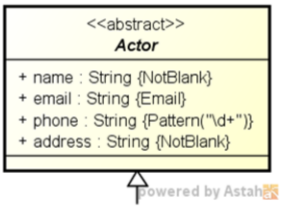


So far, we haven't talked about constraints, but we have presented many. They are the keywords in brackets next to some attributes. For instance, in this slide, we present the "Payment" class, which has two attributes: "moment" and "amount". Note that "moment" is of type "Date" and has constraint "{Past}", which means that a moment is valid as long as the date it holds is in the past. The most common constraints are presented in this slide:

- Past: must be a date in the past.
- Max(max): must be a number whose value must be smaller or equal to max.
- Min(min): must be a number whose value must be greater or equal to min.
- Digits(integer, fraction): the number of digits in the integer and fraction part of a number must not exceed the specified figures.
- Pattern(pattern): must match regular expression "pattern". There's a version of this constraint that helps write several patterns easily: Pattern.List{Pattern(p1), …, Pattern(pn)}. (Recall that Java does not allow to instantiate an annotation several times, which is the reason why you resort to "Pattern.List".)
- Size(min, max): size must be between the specified boundaries (included).

They all are supported by the Javax recommendations, so we consider them "standard" constraints.

Attribute constraints (not standard)

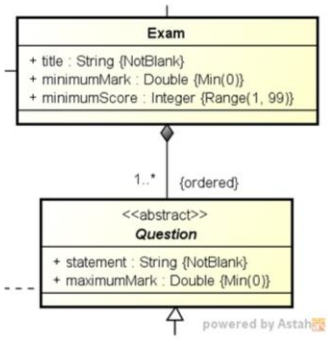


- Length(min, max)
- NotBlank
- NotEmpty
- SafeHtml
- Range(min, max)
- Email
- Url
- CreditCardNumber

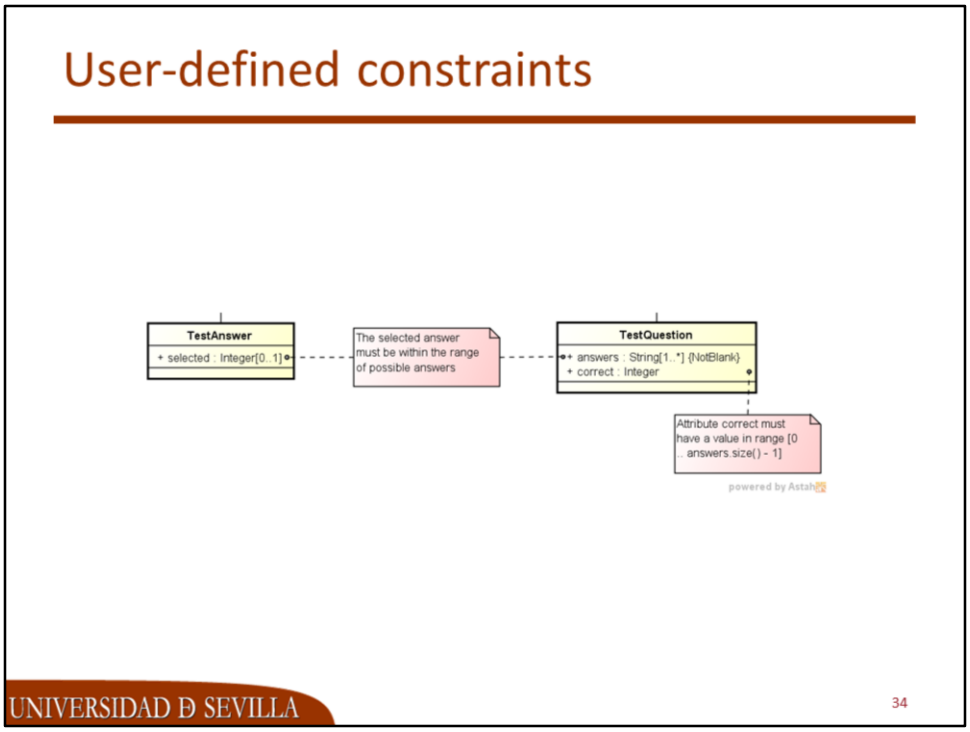
In addition to the Javax constraints, there are a number of non-standard constraints that are widely supported by the industry. They are listed in this slide:

- Length(min, max): the length of a string must be within the expected range (included).
- NotBlank: must not be a null string, an empty string, or a string that consists exclusively of blanks.
- NotEmpty: must not be a null or empty collection.
- SafeHtml: must not be a string that contains unsafe HTML code, such as <script> blocks.
- Range(min, max): must be a number within the expected range (included).
- Email: must be a lexically/syntactically valid email address.
- Url: must be a lexically/syntactically valid URL address.
- CreditCardNumber: must be a lexically/syntactically valid credit card number.

Composition/aggregation constraints

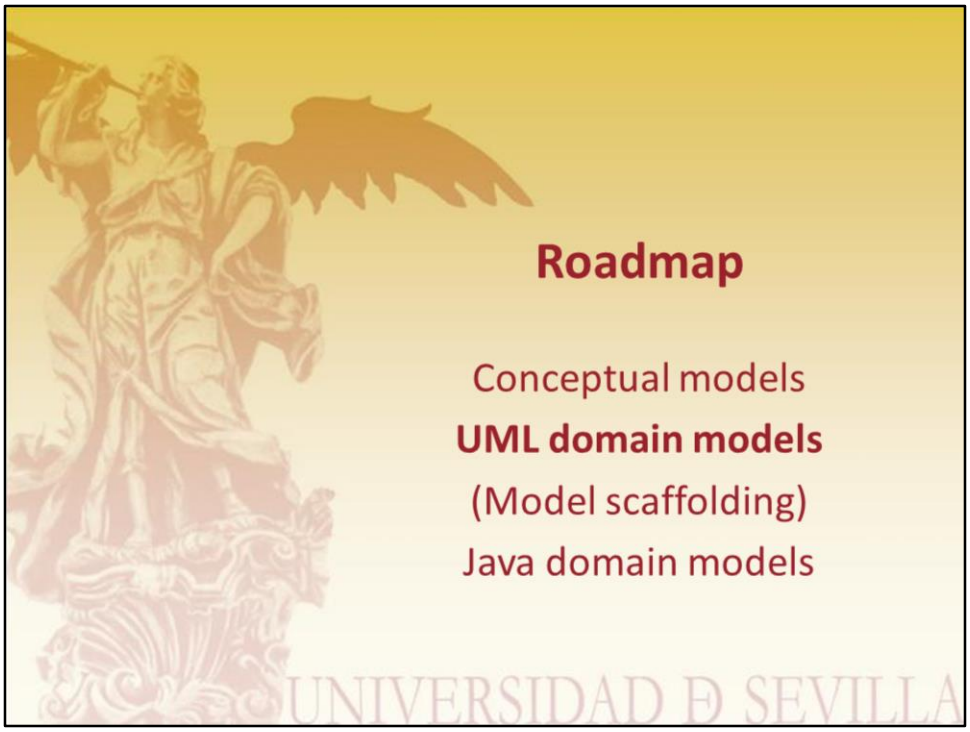


Regarding compositions and aggregations, there's a common constraint called "ordered". It means that there's an order in the collection of parts that a whole composes or aggregates. For instance, the piece of model in this slide states that an exam is composed of one or more questions, and that this composition is ordered. Note that the model is inherently ambiguous: ok, the composition is ordered, but ordered with regard to what? It's common that your requirements engineer doesn't record how the order's set simply because there's no way to specify that. For instance, if he or she asked Acme Certifications, Inc. about how questions are ordered, they might answer that "the questions are entered in order". Full stop. That usually means that we'll later have to introduce something in our model to help implement the order.



Before concluding this section on constraints, we must write a few words on user-defined constraints. Note that bracket constraints are powerful and very useful, but not enough to represent every constraint in a typical real-world conceptual model. More often than not, we'll have to resort to user-defined constraints, which are written in notes that are connected to the attributes or the classes involved. For instance, in this slide, we present a user-defined constraint that states that the value of attribute "selected" is an integer that must be within the range of possible answers specified in the corresponding "TestQuestion"; e.g., if a test question has three answers, then the selected answer in a test answer must be in range one through three; the other user-defined constraint is similar, since it establishes that attribute "correct" in class "TestQuestion" must be within the appropriate range. If possible, OCL should be used to express user-defined constraints, but don't get obsessed: the ultimate goal's clarity not using OCL.

NOTE: you're going to learn about OCL in subject Requirements Engineering.



Fine, this has been quite a good overview of the artefacts that UML provides to create conceptual models. It's now time to delve into UML domain models.

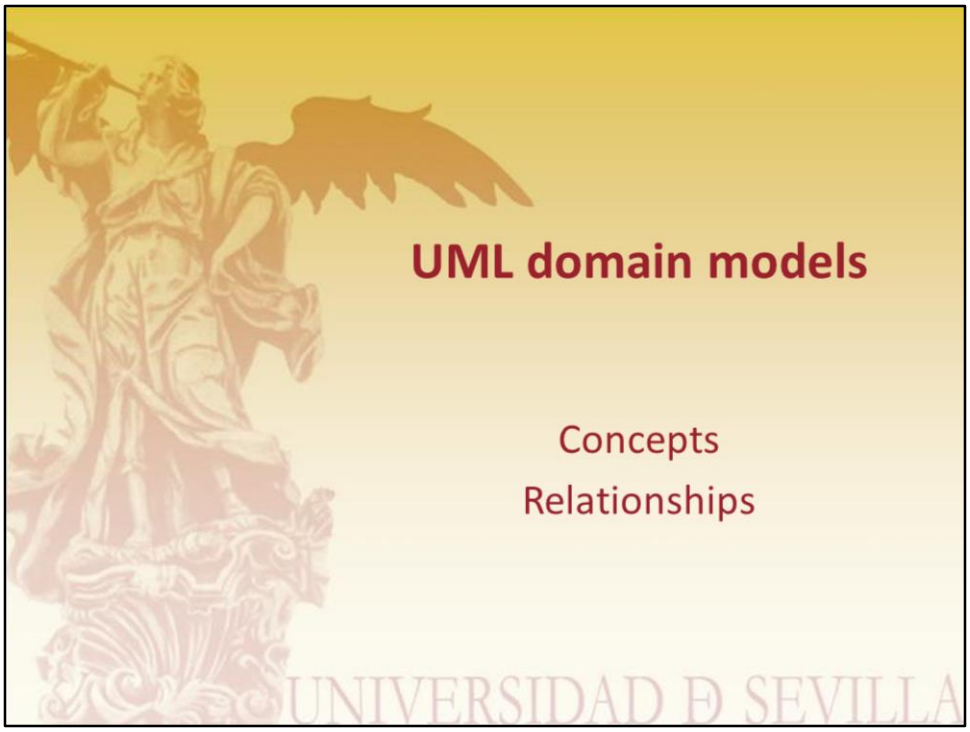
UML domain models

It enhances a conceptual model with features that help implement it using current technologies

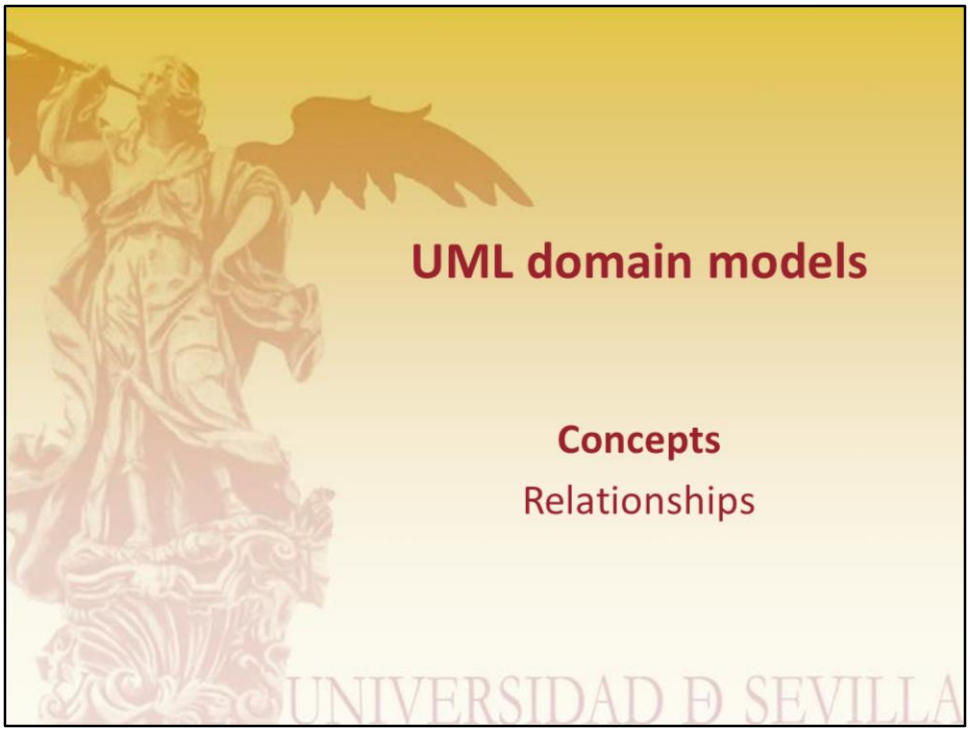
UNIVERSIDAD Ð SEVILLA

36

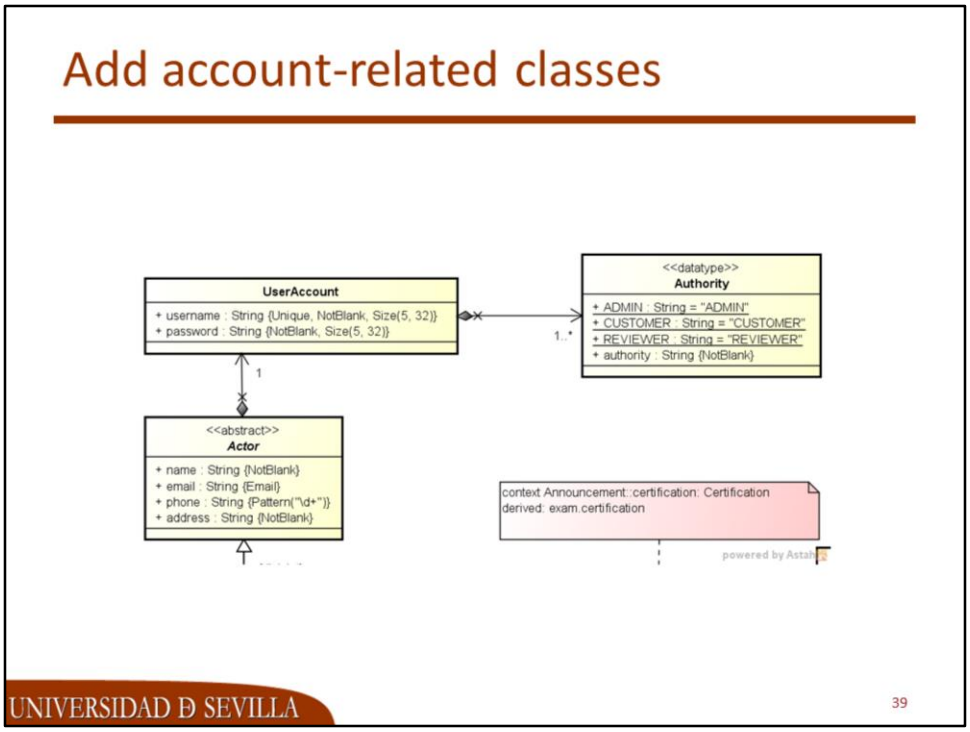
To create a UML domain model, we must enrich the conceptual model with some features that help implement it using current technologies. Can you remember the sketch of a house that we introduced in the previous section? It was quite a simple sketch, but enough to understand the overall idea. The sketch in this slide is a house, too; but has many additional details that make it easier to build it. In this section, we'll explore a number of systematic transformations to transform a conceptual model into a UML domain model; that is, a number of transformations to add details to a conceptual model.



To explore UML domain models, we have to study some transformations regarding concepts and relationships; constraints do not typically require any transformations.

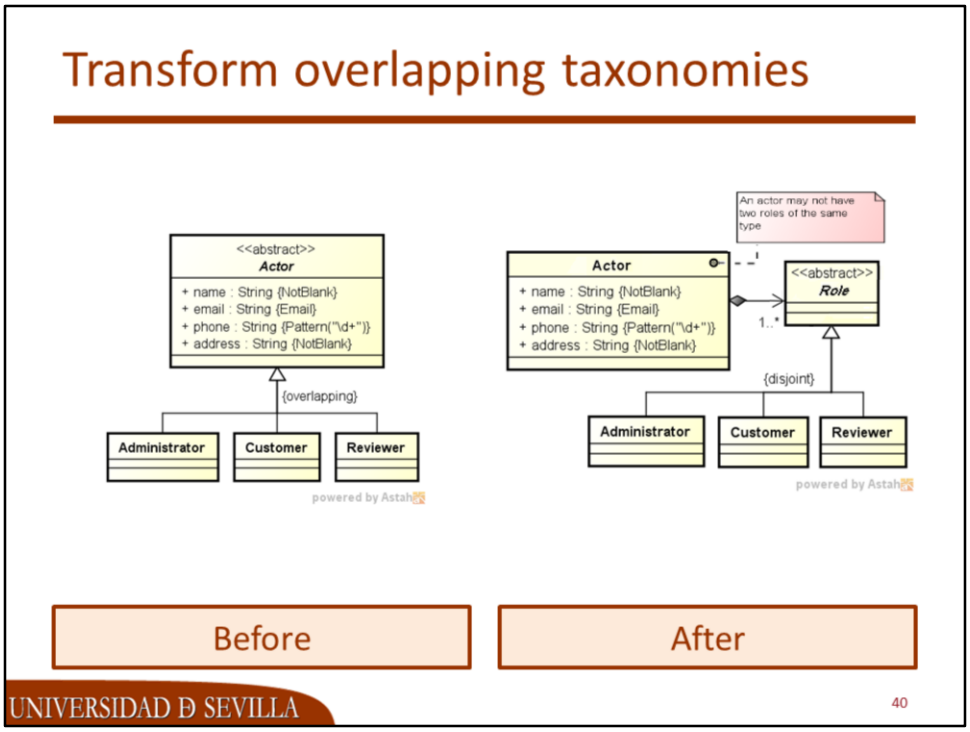


Let's start with the transformations that have an impact on the concepts that are represented in a conceptual model.

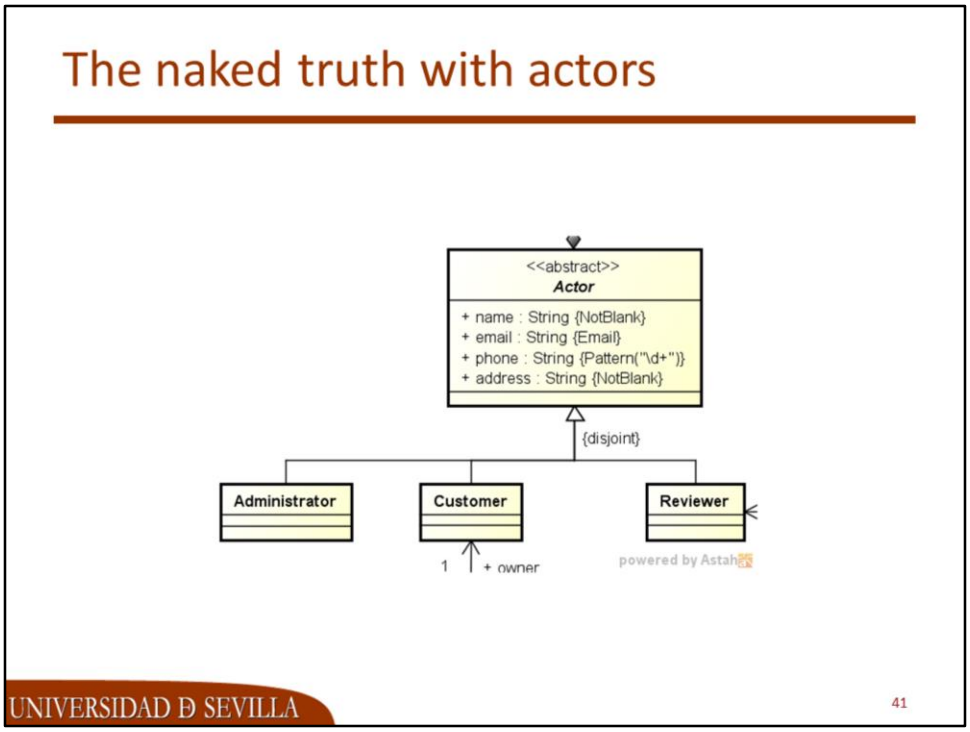


Add account-related classes and compositions to link your actors to user accounts or vice-versa. These classes are very dependent on the framework used to implement your web information system. It's not common that you need to implement these classes from scratch. We provided them to you in lecture L01/S02 "The Working Environment", and it's time to provide a few additional details on them. The key class is "UserAccount", which basically models a username and a password (more precisely, a password hash); each user account composes a number of "Authority" objects (authority is a synonym for role). By default, we provide an implementation that supports the following authorities: "ADMIN" and "CUSTOMER"; if you need additional authorities, please, feel free to add them; in our running example, we need to support administrators, customers, and reviewers. We assume that you got the overall idea: an actor has a user account, and a user account composes several authorities or roles. Given that for granted, let's delve into a couple of details:

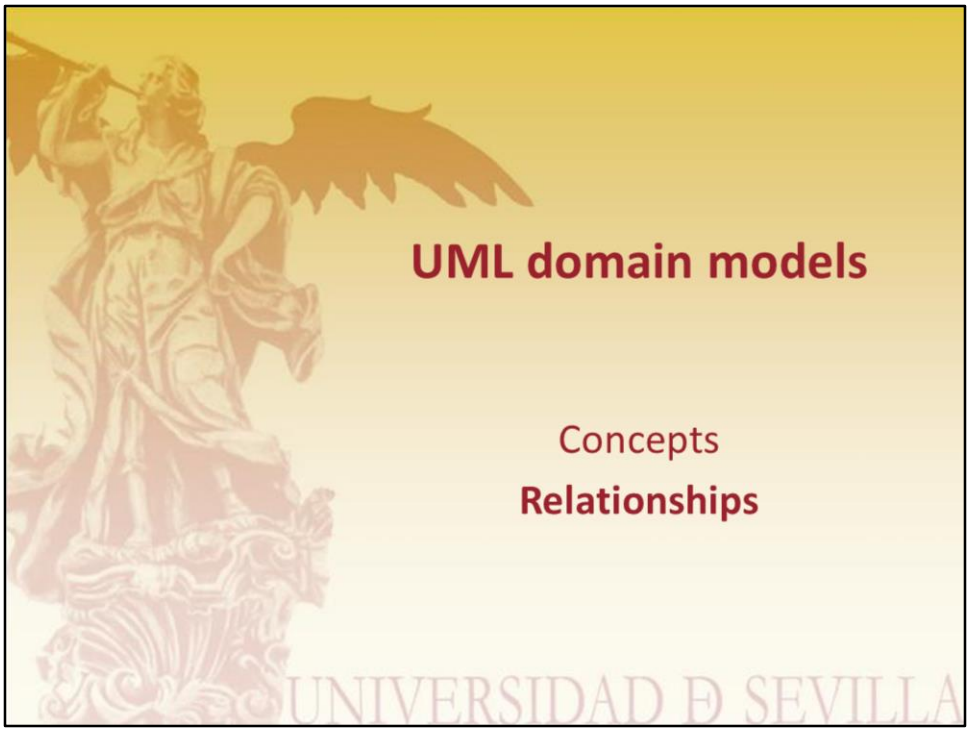
- Note that attribute "username" in class "UserAccount" has a constraint of the form "{Unique}", which we haven't studied so far; this is a fictitious constraint that we'll use to express that there can't be two different objects of class "UserAccount" with the same value for attribute "username". If you're thinking of primary keys and so on, please, forget about that; this constraint simply means that the values of this attribute are unique, nothing else.
- Note that new authorities are introduced in class "Authority" using static attributes of type "String"; for instance, we've introduced a new authority called "REVIEWER". We know that this design is a little cumbersome; simply learn that by heart, and assume that it's a technology limitation.



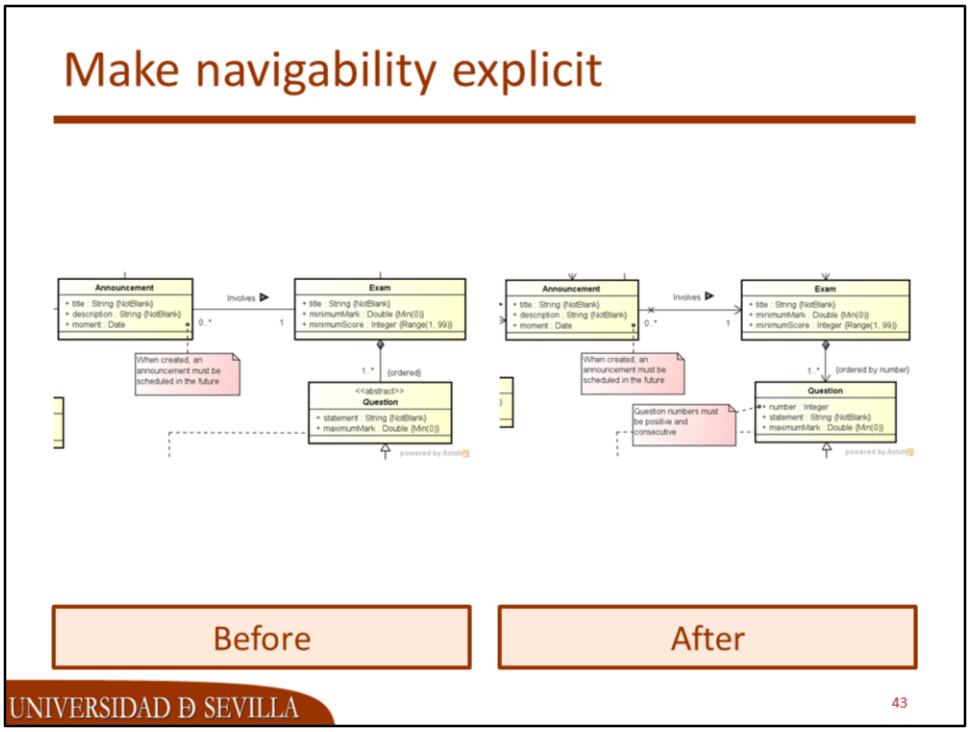
Now, it's time to review overlapping taxonomies. They are very common in practice, but there's a problem: Java doesn't support them natively. That means that we need to transform them appropriately, as shown in this slide. The transformation consists of creating a new taxonomy in which there's a new parent class called "Role", which is the ancestor of the previous subclasses of "Actor"; note now that the relationship between classes "Actor" and "Roles" is a composition relationship. Thus, an actor that is conceptually modelled as, say, an administrator and a customer, is transformed into an actor that has two roles: administrator and customer. Before concluding, please, note a couple of details: a) it makes sense to add a user-defined constraint to make it sure that an actor may not have two instances of the same type of role (it doesn't make sense that an actor is an administrator, an administrator, and a customer), and b) class "Actor", which was originally abstract, has to become a concrete class.



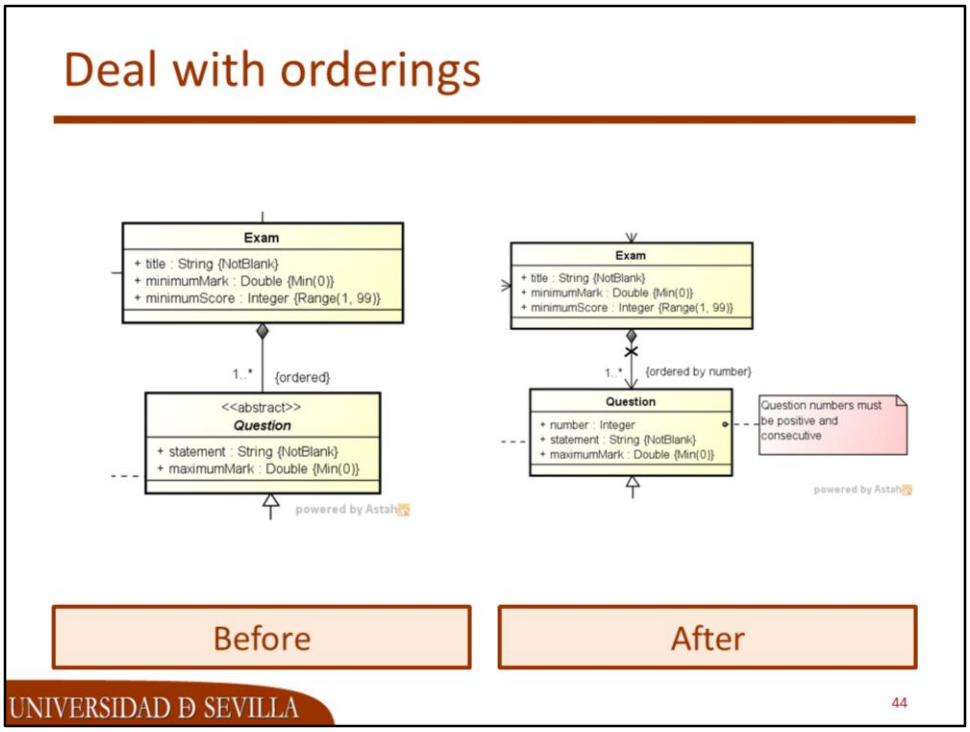
The previous transformation is pretty nice… but reality is cruel! You can apply the previous transformation, but unfortunately you'll soon discover that it's not as easy as expected to support user accounts that may have several roles. Our suggestion is that you may try the previous transformation; we'd say that you should. But you're very likely to get into a lot of trouble. Thus, regarding overlapping taxonomies that represent actors, we recommend a simpler alternative: model the specific kinds of actors in your system as a disjoint taxonomy. The limitation is that a user who is both an administrator and a customer will require two different user accounts; this limitation is very common in practice, so no-one will say that you're not professional enough because you use this model. Let's wait for a little more time until the technology matures enough to support user accounts with multiple roles seamlessly.



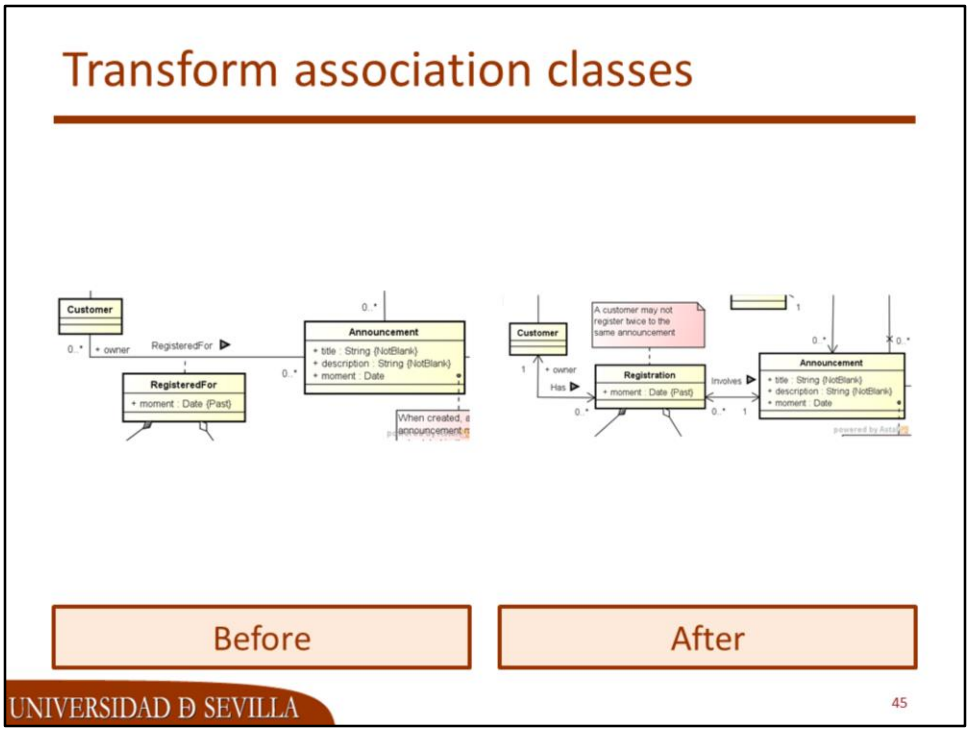
Simple enough, isn't it? Let's now delve into the transformations that are related to the relationships in a conceptual model.



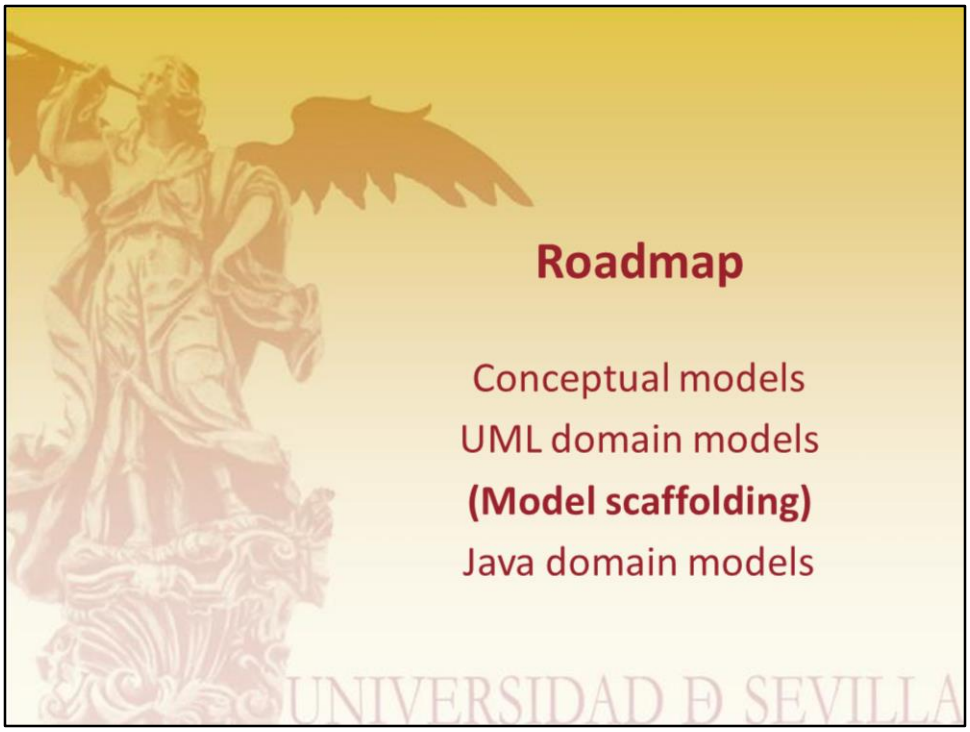
We're sure you've already realised that the associations that we have presented in the previous slides have arrow heads at their ends. That's navigability. By default, associations in a conceptual model are bidirectional, which means that given an object of class, say, "Announcement" you can navigate to the corresponding object of class "Exam", and that given an object of class "Exam" you can navigate to the collection of objects of class "Announcement" that are associated with it. That's great from a conceptual point of view, but it has some flaws from a computational point of view: most of them are related to efficiency. In the example in this slide, for instance, we have made it explicit that given an announcement it is possible to navigate to its corresponding exam; simply put, given "a" of type "Announcement", you can write "a.getExam()" to fetch its associated exam; however, given "e" of type "Exam", you can't write "e.getAnnouncements()" to fetch the announcements in which a given exam is involved. Why not? -- you may wonder? Can you remember one of the very first slides in which we mentioned that you need to have a holistic view of your requirements? That means that the focus to devise a domain model is the section on information requirements of your requirements elicitation documentation, but you need to keep an eye on your ilities and use cases, as well. Whenever you have to decide if an association between classes "A" and "B" must be bidirectional or unidirectional, ask yourself the following question: is there a use case in which you need to navigate from "A" to "B"? Then make it navigable from "A" to "B". Is there a use case in which you need to navigate from "B" to "A"? Then make it navigable from "B" to "A". No navigation from "A" to "B" or vice versa? Ain't good!



Can you remember ordering constraints on compositions and aggregations? We mentioned that it is common that requirements engineers can't easily elicit how a composition or an aggregation is ordered, which makes these constraints incomplete. Unfortunately, we can't go ahead and create a domain model and implement it unless we make it explicit how compositions and aggregations are ordered. Usually, the problem's solved if you add an attribute on which you can set the order to the part class. For instance, in this slide we show a piece of our sample model that represents exams, which are in turn composed of ordered collections of questions; note that transforming this model into a domain model implies adding a new attribute called "number" that is of type integer; it's easy to establish an order on this attribute, plus a user-defined constraint to make it explicit that numbers must be positive and consecutive. Right, this might seem far-fetched; it's very unlikely that the requirements engineer couldn't elicit this requirement, but our sole purpose was to illustrate the idea, right?

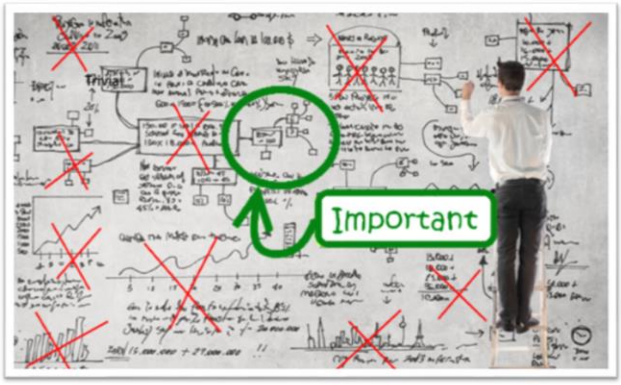


Recall that an association class models the attributes of an association. Unfortunately, associations don't exist in Java, which implies that association classes don't exist, either. The transformation is shown in this slide: you first need to transform association class "RegisteredFor" into a regular concrete class whose natural name is "Registration"; then, you need to add a relationship from class "Customer" to class "Registration" and an additional one from class "Registration" to class "Announcement", but make sure that you adjust the multiplicities so that the resulting model faithfully represents the original one. Note, too, that you need to add a user-defined constraint to class "Registration" to make sure that a customer doesn't have two registrations for the same announcement. Before concluding this transformation, please, note that it usually makes sense to change the name of the association class when it's transformed; since an association class is an association, its name is usually a verbal expression; when you transform it into a class, it's common to change its name into a noun.



Great! We now know how to create a UML domain model, something we can hand to our developers so that they can start coding. However, before presenting Java domain models, it's necessary to provide a few details regarding something called model scaffolding.

What's model scaffolding?

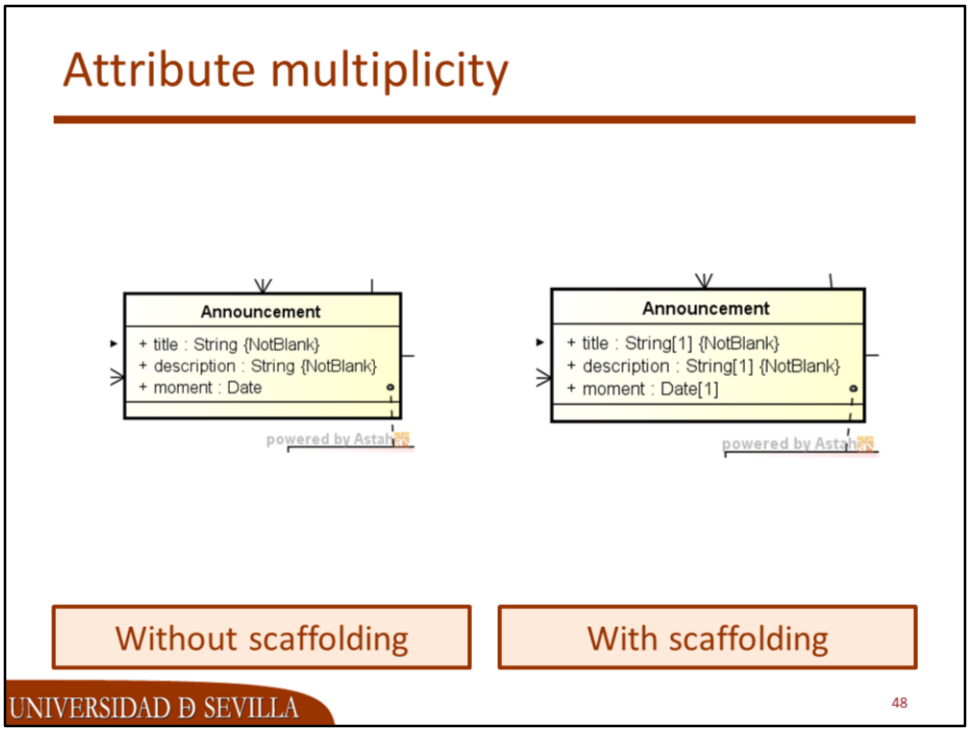


Scaffolding a model means that you make it explicit things that are trivially implicit, which has a negative impact on readability.

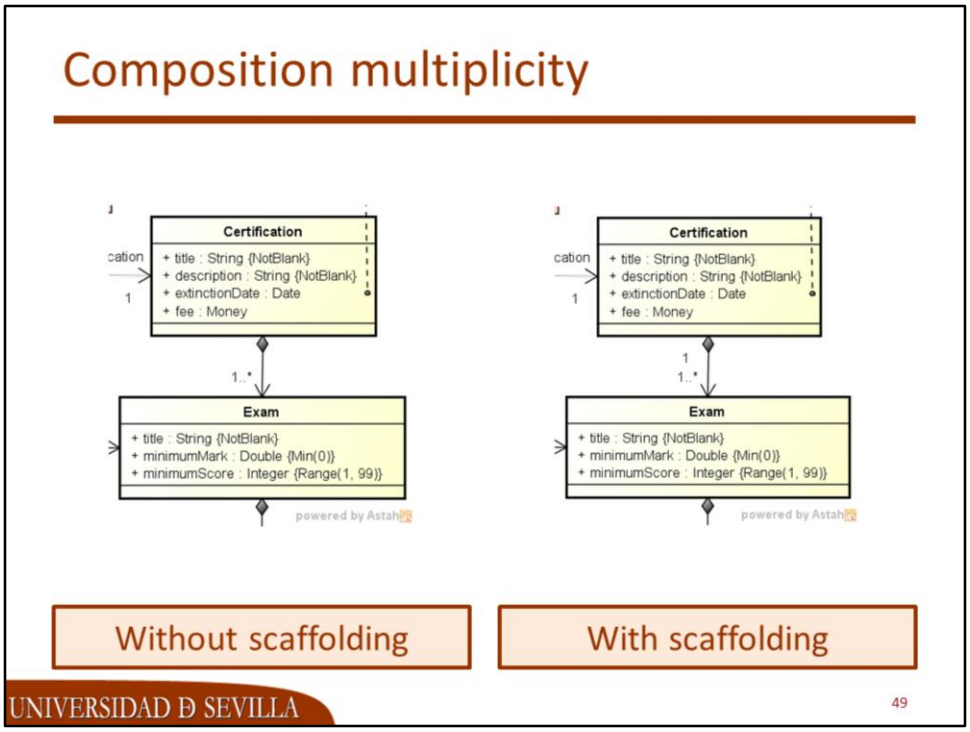
UNIVERSIDAD Ð SEVILLA

47

We're pretty sure you're not familiar with this term, and looking it up in a dictionary won't help, either. The idea is simple: you scaffold a model if you make it explicit things that are trivially implicit. Simply put: it makes it difficult to see the forest (the important things in the model) for the trees (the trivial implicit things made explicit). Peek at the following slides and you'll understand the idea very easily.

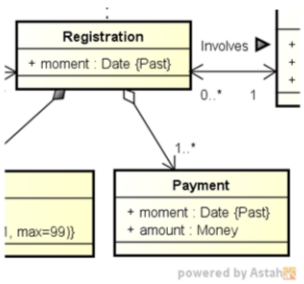


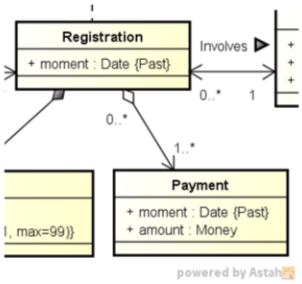
Recall that you can specify the multiplicity of an attribute. For instance "title: String[1]" means that "title" is an attribute that stores exactly one string; right, that's trivially implicit, so you don't to need to make it explicit. The model on the left wasn't scaffolded; the model on the right was.



In this slide, we present a model according to which a certification composes a number of exams. Note that we model this relationship as a composition, which implicitly indicates that an exam belongs to exactly one certification; then, it's not necessary to indicate that the multiplicity on the upper end of the composition is "1", like in the model on the right.

Aggregation multiplicity

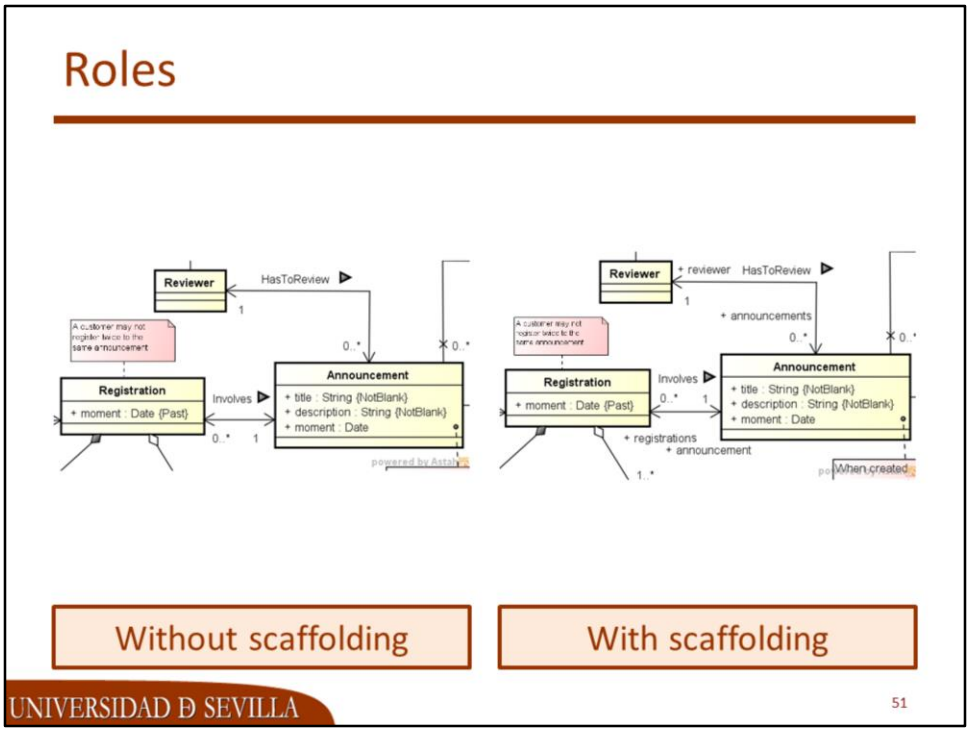




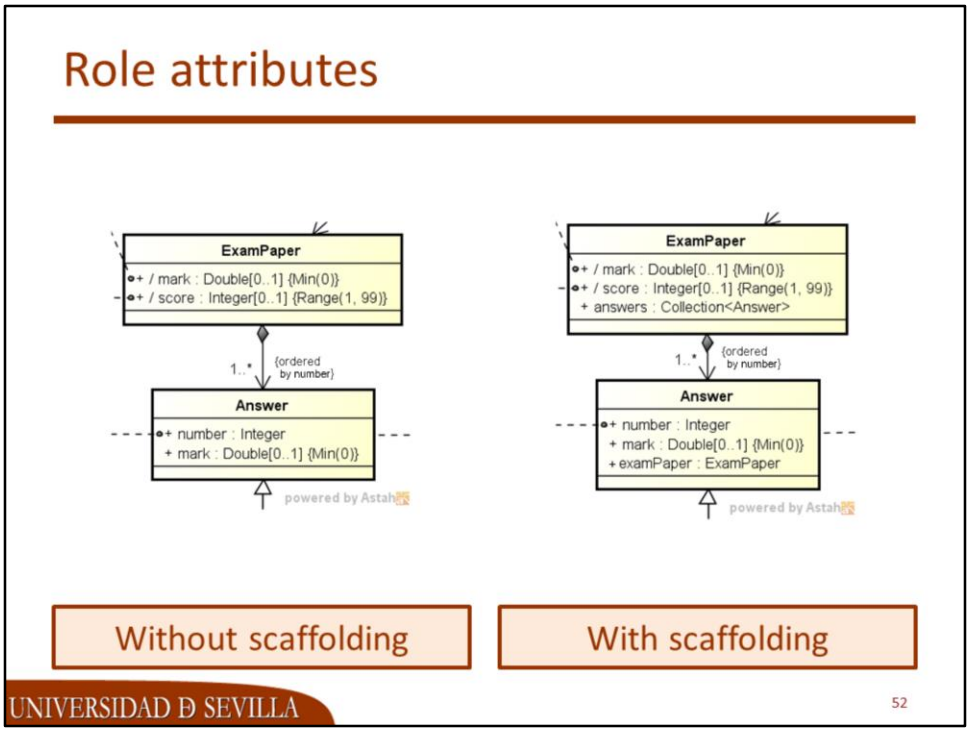
Without scaffolding

With scaffolding

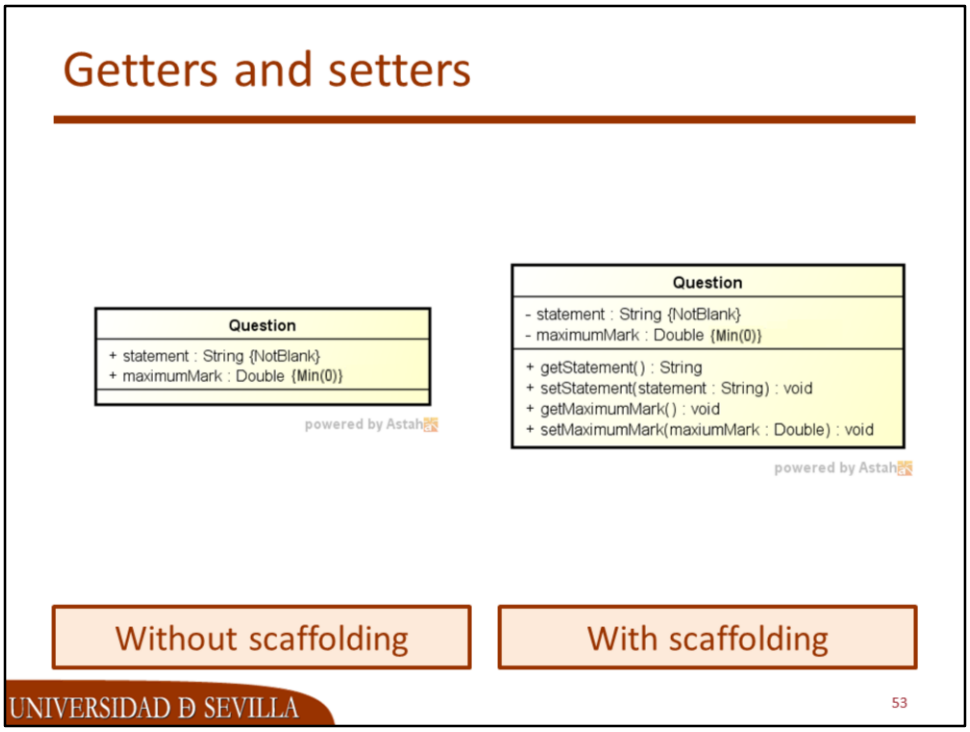
Similarly, an aggregation implies that a part may belong to several wholes at a time, or none at all. Then the model on the right makes it explicit a multiplicity that is trivially implicit.



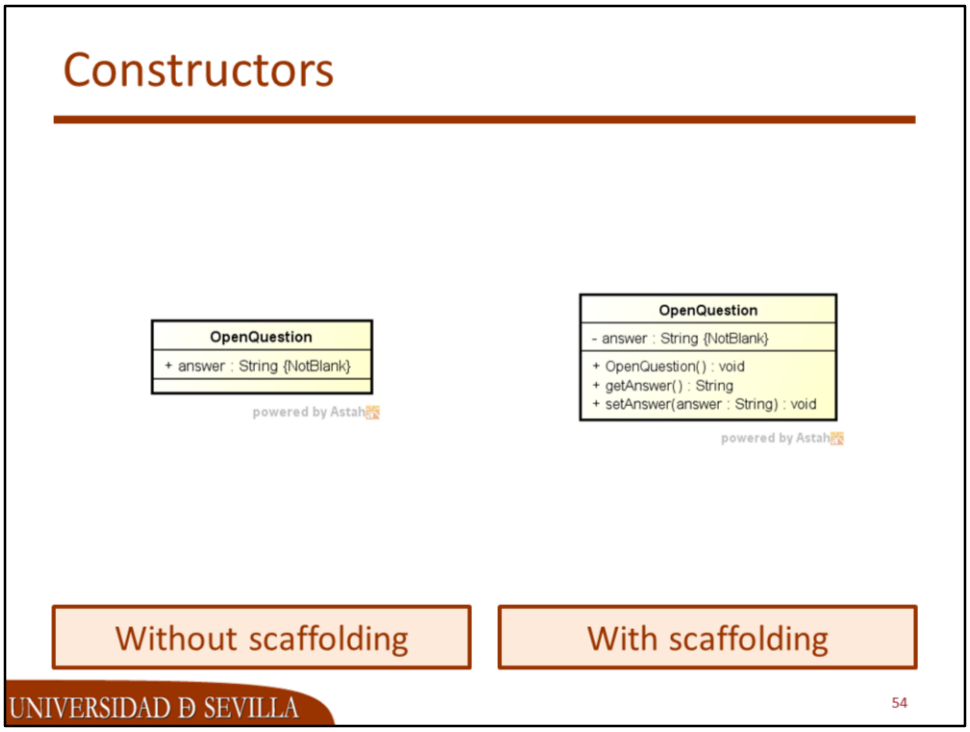
Our students love to scaffold roles. Please, remember that roles must be overridden only when their default names aren't appropriate. If the default name is appropriate, do not override it; leave it implicit in your model.



Our students also tend to scaffold role attributes. Intuitively, every time you specify a relationship between two classes and specify its navigability, the classes have implicit attributes to implement that navigability. For instance, take a look at the model on the left: we specify that an exam paper is composed of an ordered collection of answers and that this composition is bidirectional (please, recall that there's a bug in Astah that prevents us from drawing the arrowhead at the whole end of a composition or aggregation). That means that class "ExamPaper" has an implicit attribute called "answers" that is of type "Collection<Answer>", and that Answer has an implicit attribute called "examPaper" that is of type "ExamPaper"; you shouldn't scaffold your model and make it explicit like in the model on the right.

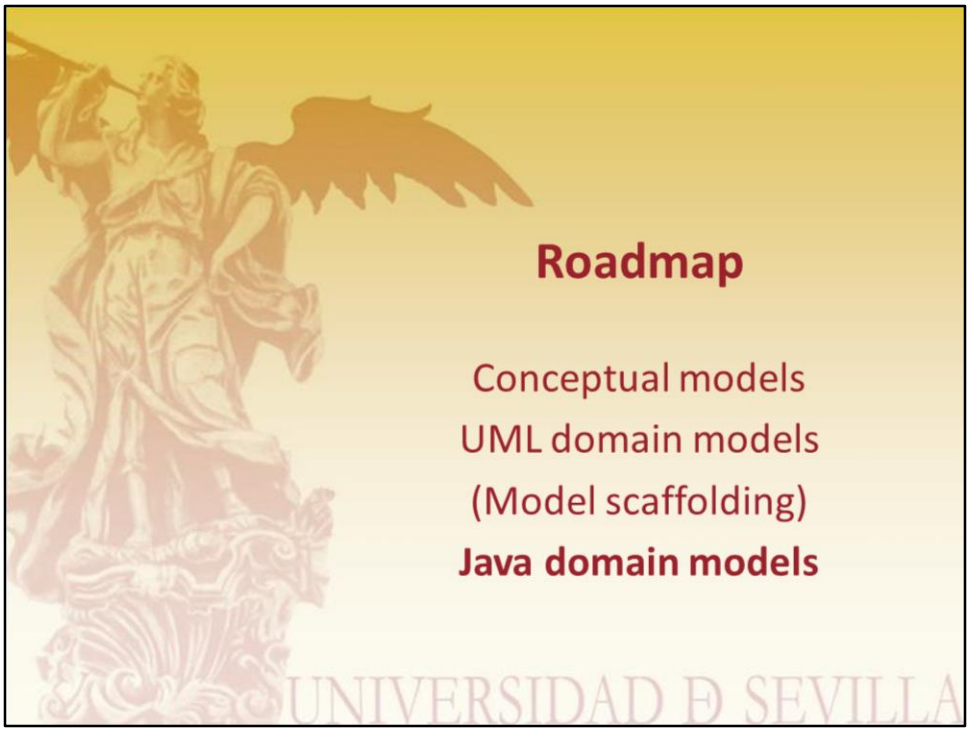


Our students love getters and setters, too. Please, note the difference between the model on the left and the model on the right. Note that understanding a UML model requires to agree on the semantics of its artefacts (locally). Some people might cry out: “the class on the left is not correct, you can’t have public attributes”. Wait a minute, the class on the left is a model; as such, it represents something at an appropriate abstraction level. Writing “+ statement: String” does not entail that the corresponding Java class must have a public attribute called “statement”. We may agree on that public attributes in a UML domain model are transformed into private attributes with setters and getters in the Java model. Ok, in Java, you’ll have to write getters and setters, but why to scaffold your model with hundreds of getters and setters that are good for nothing.



Similarly to getters and setters: do not include constructors in your UML domain models. We can safely assume that every Java class will have a default constructor.

In general, scaffolding your models is considered a bad practice because it overloads them with irrelevant details; and the problem is that these irrelevant details may easily hide important ones. So, it's very important that you bear this in mind: you must not scaffold your models! If you scaffold them, then they won't be considered valid.



We're close to the end of the presentation. Once we know about conceptual models, the transformations required to transform them into UML domain models, and a little about how inappropriate it is to scaffold a UML model, it's now time to report on Java domain models.

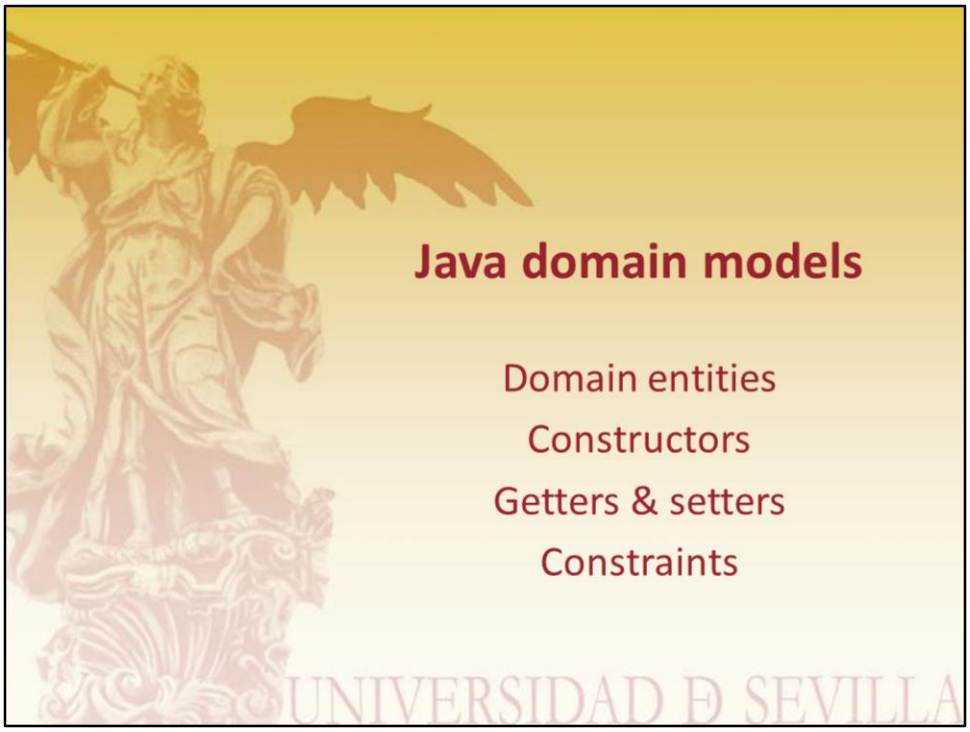
Java domain models

It provides the classes that implement a UML domain model

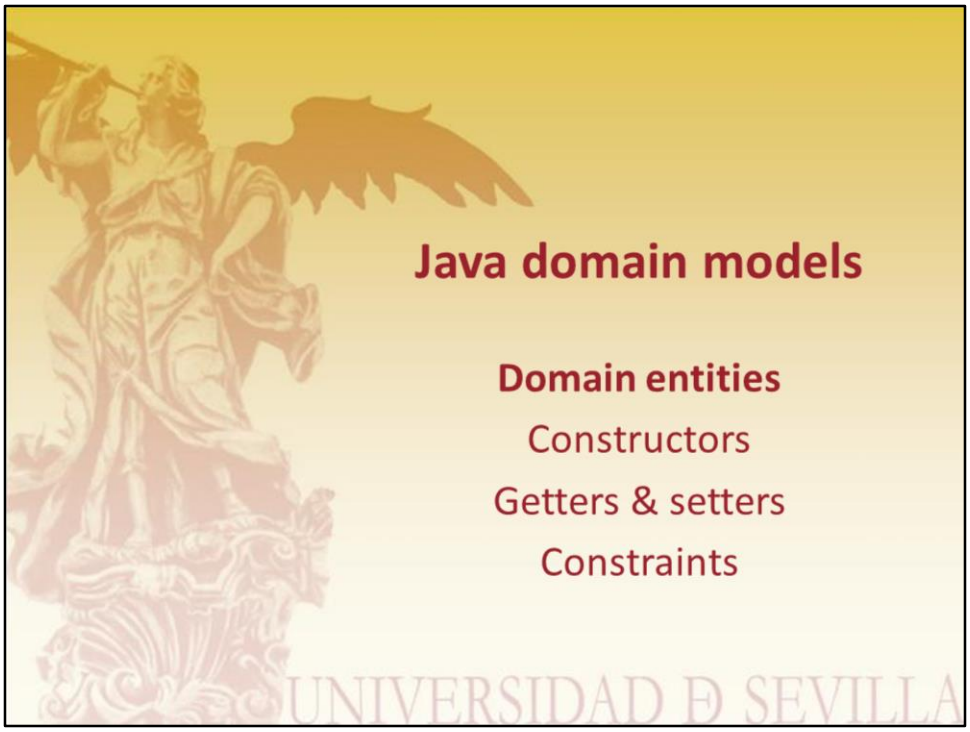
UNIVERSIDAD Ð SEVILLA

57

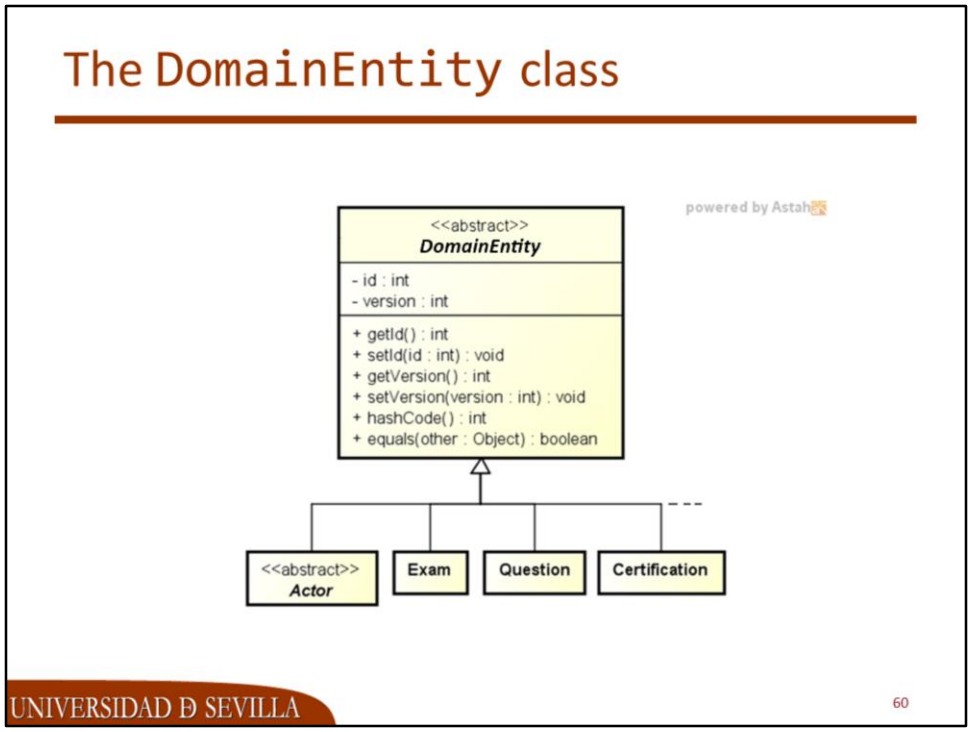
Can you remember the sketch of the house with which we introduced conceptual models? Later, we used a version with more details to illustrate the idea of UML domain models. Now we present a picture of a house. We shall start with the UML domain model and will apply a series of transformations that will result in a Java model. (If Java model sounds strange to you, you can say Java program, Java library, or whatever helps you understand the idea that this is something you can compile and run).



This is our roadmap regarding Java domain models. We'll start with something called domain entities, then we'll report on constructors, getters and setters, and, finally, on constraints.

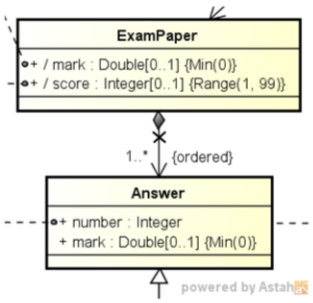


Let's start with domain entities.



This is a sketch of class "DomainEntity". (Note that we're scaffolding the class because it represents a Java model in which little can be left implicit; almost everything in Java must be made explicit.) Note that we have to make "DomainEntity" an ancestor of classes other than your datatypes, which endows these classes with two additional attributes called "id" and "version". The former represents the identity of an entity, which is modelled as an integer number; the latter represents its version. You shouldn't have too many problems to understand the idea behind the "id" attribute, but the "version" attribute requires a little more explanation. Note that an entity typically goes through several stages during its lifecycle; that means that you can create an exam, then set the title, minimum score, and so on, later you may find that you made a mistake when you entered the title and you have to correct it… that's very frequent. Note that the "id" attribute should be immutable throughout the lifecycle of an entity, since it represents its identity and it does not change once an entity is created; the "version" attribute, however, changes every time the attributes of an entity are modified; this helps count how many changes an entity has undergone during its entire lifecycle. Why should we be interested in tracking versions? The answer's pretty simple: we need it to prevent concurrency problems. Think of an administrator who is editing an exam and another administrator who edits the same exam concurrently (maybe because they didn't co-ordinate well, but this is far too a frequent issue to ignore it); the former edits the exam and saves it, which increases its version number; the latter edits it and whenever he or she saves it, the system can easily check that the version that is stored in the database doesn't correspond to the version the second administrator has edited. This results in a so-called concurrency problem that's very easily detected by means of a version attribute. In addition to the "id" and the "version" attributes, "DomainEntity" also overrides methods "hashCode" and "equals" so that they work well with entities.

Check attribute types




```
public class ExamPaper
    extends DomainEntity {
    …
    private Double mark;
    private Integer score;
    …
}

public class Answer
    extends DomainEntity {
    …
    private int number;
    private Double mark;
    …
}
```

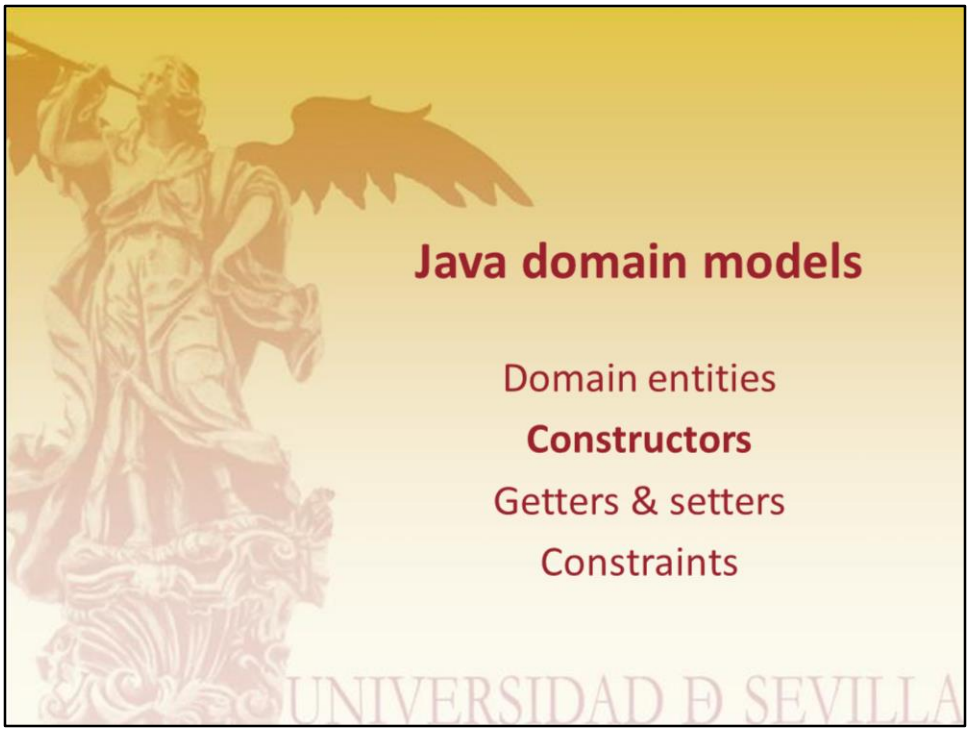
UML domain model

Java domain model

UNIVERSIDAD D SEVILLA

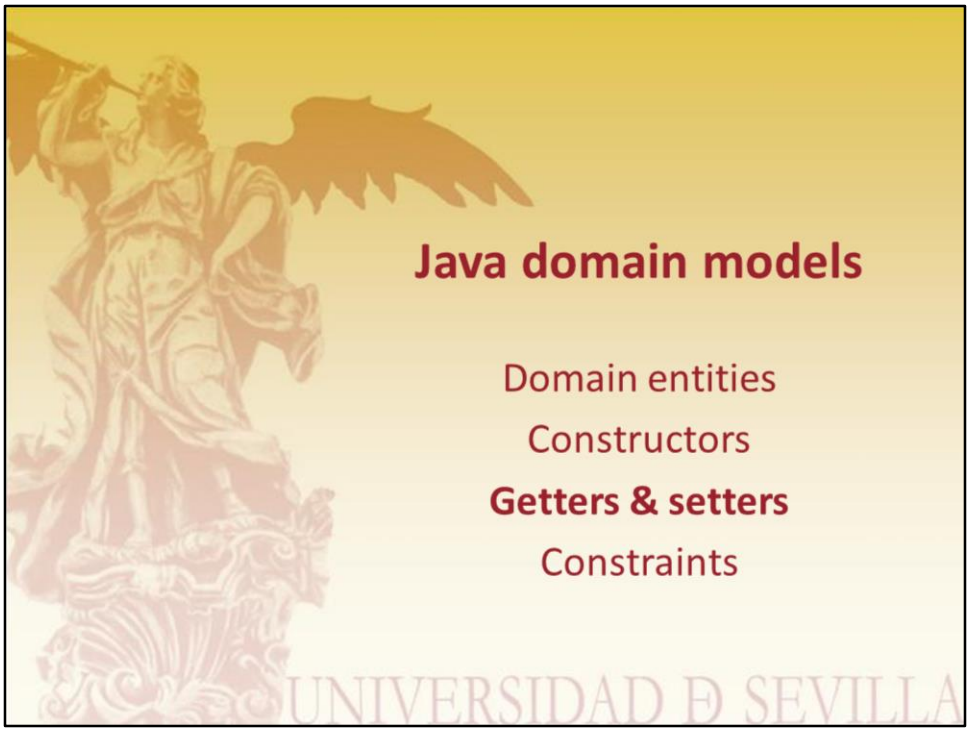
61

Please, note that we've always used datatypes like "Integer", "Double", or "Boolean" in the conceptual or the UML domain models we've used so far. Java provides implementations for these datatypes with exactly the same names, and they are known as wrapper types. Every wrapper type has a corresponding primitive type; for instance, "int" is the primitive type for "Integer", "double" for "Double", and "boolean" for "Boolean". The difference between a wrapper type and a primitive type is simple: an instance of a wrapper type is an object, which implies that it may be "null", whereas an instance of a primitive type is a value, which implies that it cannot be "null". We now must make a decision on whether UML datatypes like "Integer", "Double", or "Boolean" must be implemented using wrapper types or primitive types. The decision is very simple: are you dealing with an optional attribute? Use a wrapper type then; otherwise, use primitive types. There's a good example in this slide: note that the mark and the score of an exam paper are optional attributes; the reason is simple to understand: they can't be computed until the exam is marked and all of the exams in an announcement have been marked, respectively. That means that you need a type that can account for a "null" value to represent a mark or a score that has not been computed, yet, that is, "Double" and "Integer". The same applies to the "mark" attribute in class "Answer". Let's now analyse attribute "number" in class "Answer"; every answer must have a number, which uniquely identifies the corresponding question. The number of an answer is not optional, which means that it must have a value. As a conclusion, you should implement this attribute using primitive type "int".



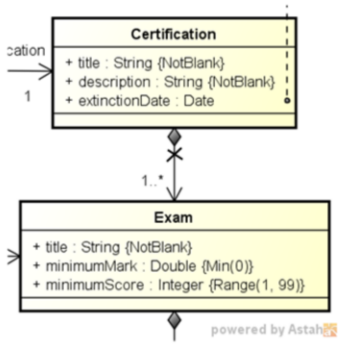
Let's now delve into the constructors.

We recommend that you should think twice of constructors. They are useful in other contexts, but not when modelling domain entities. In general, constructors are not necessary when using our framework. If you wish to add a default empty constructor just to be able to set a breakpoint when an object is created, go ahead, but you aren't likely to find many other uses of constructors. Chiefly, avoid non-default constructors, they are completely useless in our context.



Let's now report on getters and setters.

Getters and setters

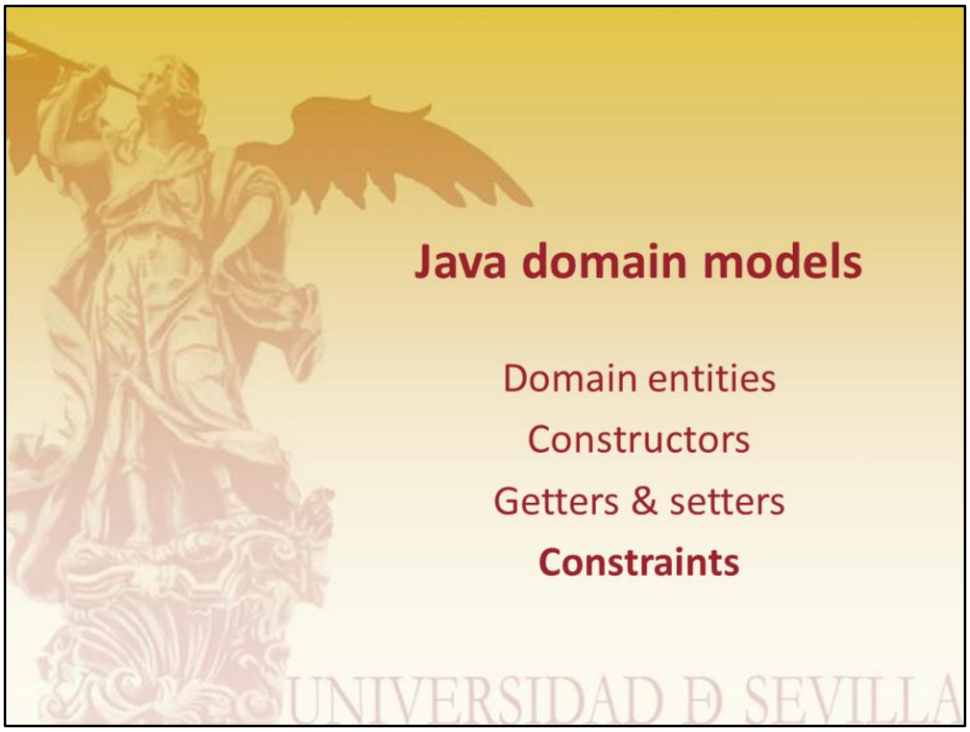



```
private String title;
private Collection<Exam> exams;

String getTitle() {
  return title;
}

void setTitle(String title) {
  this.title = title;
}

Collection<Exam> getExams() {
  return exams;
}

void setExams(Set<Exam> exams) {
  this.exams = exams;
}
```

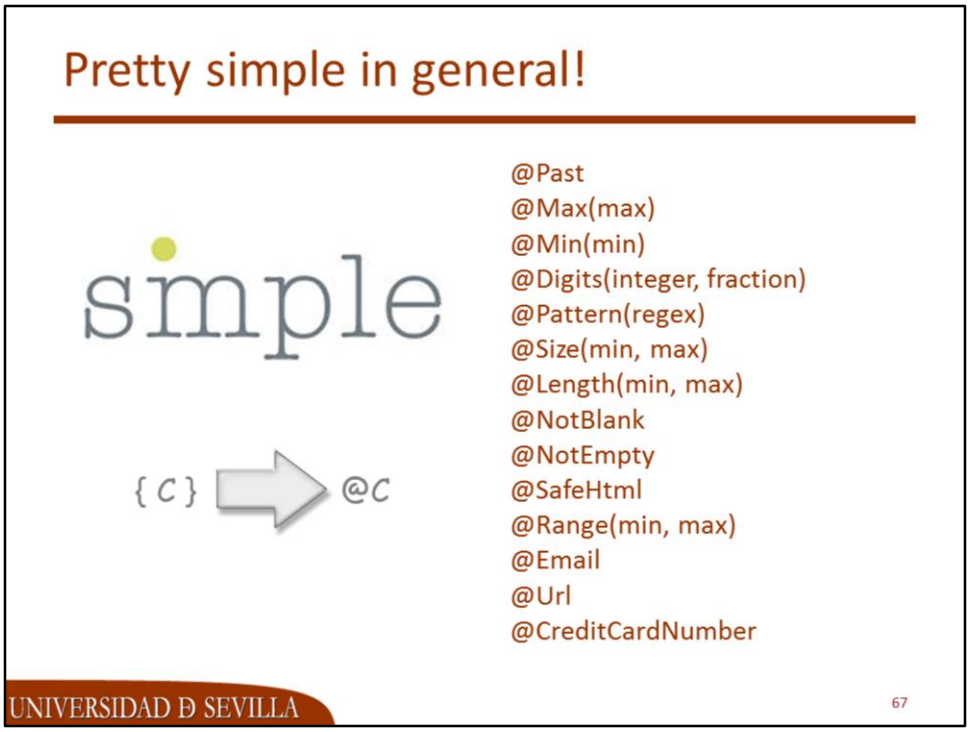
UNIVERSIDAD Ð SEVILLA

65

Ok, let's start with getters and setters. Please, remember that a few slides ago we told you that an attribute being public in a UML domain model doesn't entail that it must be implemented by means of a public attribute in the corresponding class. Please, transform every attribute in your UML domain model into a private attribute in the corresponding class plus a getter and a setter method. Don't forget role attributes! Remember that every composition, aggregation, or association with a navigable end results in an implicit role attribute in the other class. Neither should you forget about derived attributes!

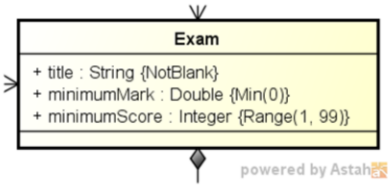


Finally, it's time to delve into how to implement constraints.



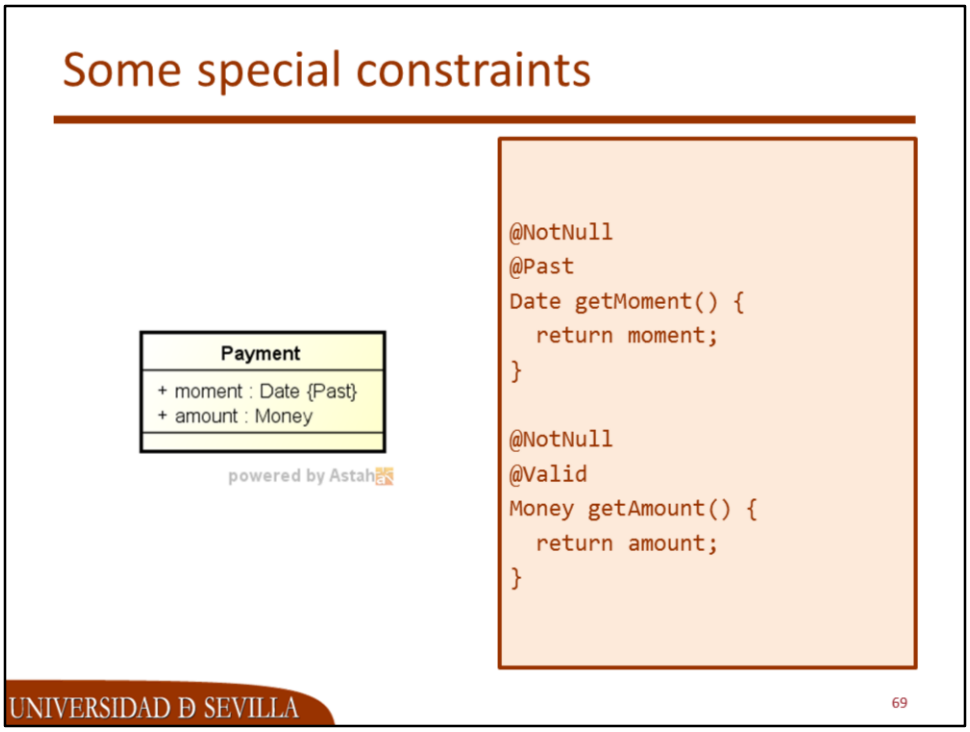
Implementing a constraint is pretty simple. Generally speaking, for every constraint of the form "{ C }" in your UML domain model, there's a corresponding "@C" annotation in Java that implements it.

Attach constraints to getters

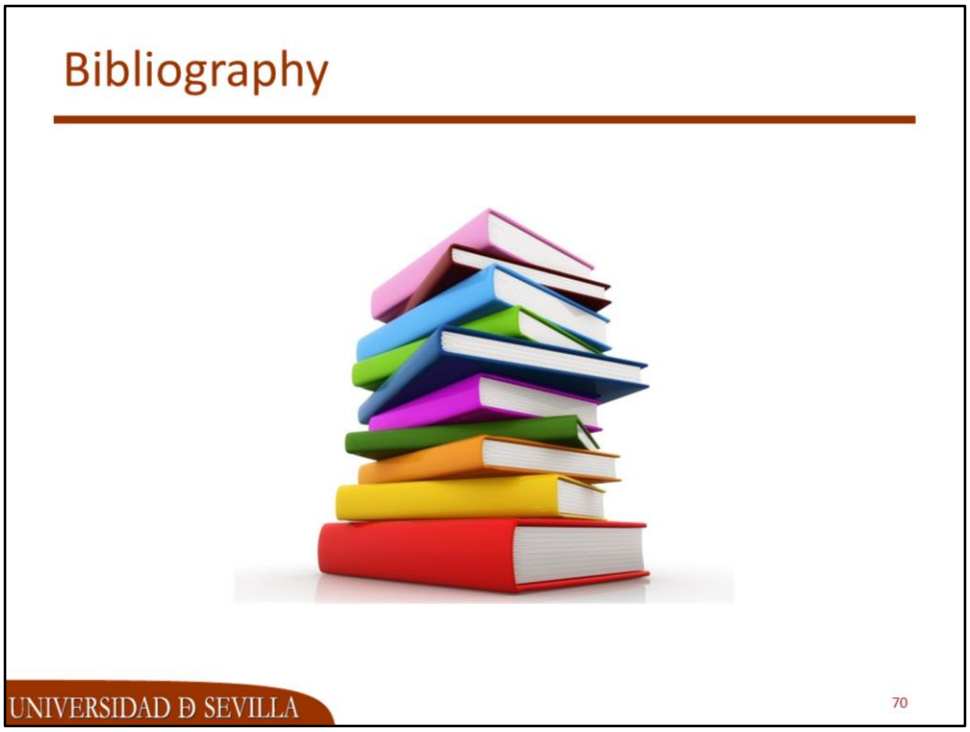



```
@NotBlank
String getTitle() {
  return title;
}

@Min(0)
double getMinimumMark() {
  return minimumMark;
}

@Range(min = 1, max = 99)
int getMinimumScore() {
  return minimumScore;
}
```


Constraints are attached to getters, like in this slide. It's pretty simple so far; unfortunately, there's an issue.



There are two special constraints that are only available in your Java domain models: "@NotNull", which requires an attribute not to have a null value, and "@Valid", which requires an attribute to reference an object that fulfils its corresponding constraints. They might seem confusing on a first reading, but they are simple to understand. Take a look at the class that models payments: it has an attribute called "moment", which is of type "Date" and has a constraint that requires it to be in the past, and an attribute called "amount", whose type is a datatype called "Money". Note that attribute "moment" has multiplicity one in the model, which means that it cannot be null; contrarily, the "getMoment" getter returns a reference to an object of type "Date", which can be null; in order to constraint the values returned by "getMoment" not to be null, you must make it explicit by means of annotation "@NotNull". The problem with attribute "amount" is not the same: "Money" is a user-defined datatype that may have attributes with constraints; by default, constraints are not validated recursively, which means that an object of type "Payment" with an invalid amount would pass the validation process; in order to require recursive validation, we must add "@Valid" annotations where appropriate.



Should you need more information on this lesson, please take a look at the following reference:

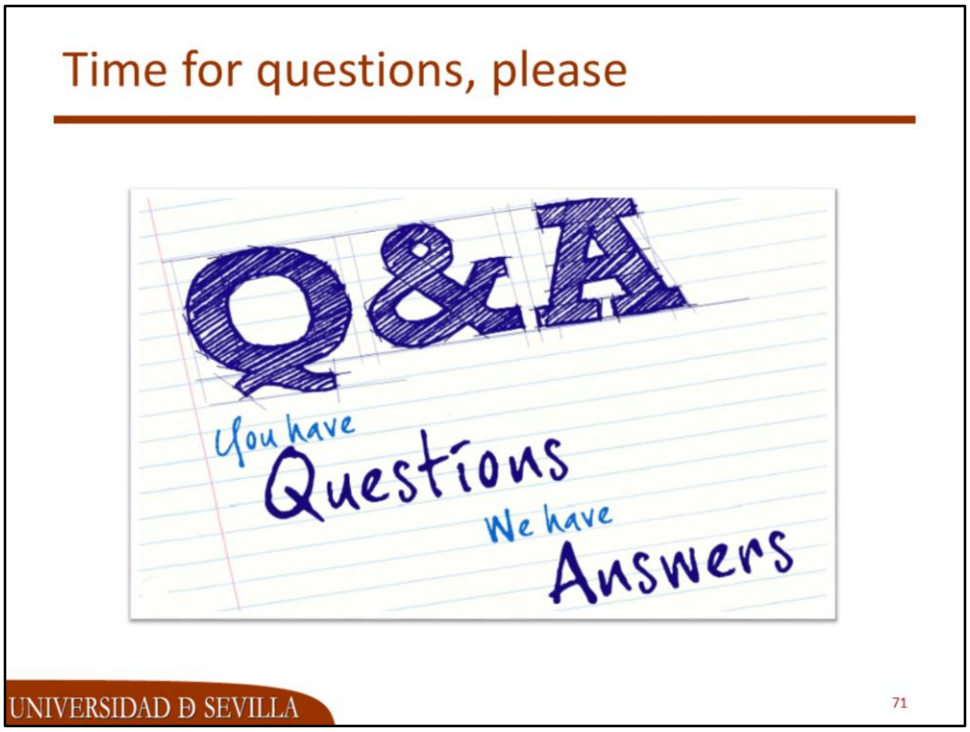
Systems analysis design UML version 2.0
Alan Dennis, Barbara Haley Wixom, David Tegarden
J. Wiley & Sons, 2012

If you need help with Java annotations, please, take a look at the following reference:

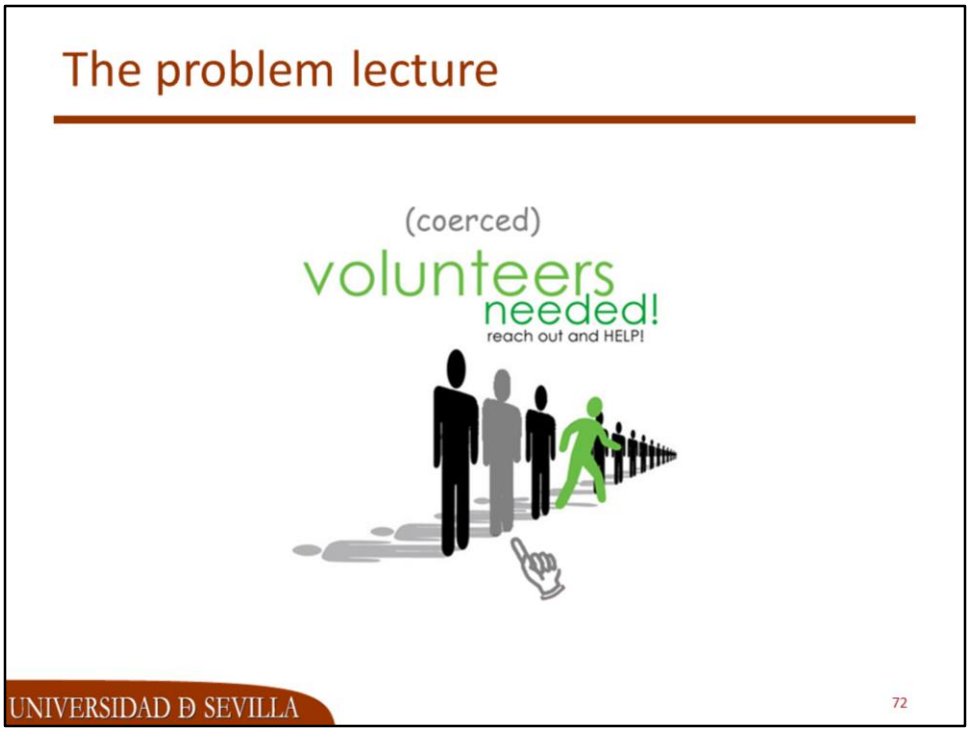
Java 7 Programming
Poornachandra Sarang
McGraw-Hill, 2012

This bibliography is available in electronic format for our students at the USE's virtual library. If you don't know how to have access to the USE's virtual library, please, ask our librarians for help.

We also recommend that you should take a look at http://docs.jboss.org/hibernate/validator/4.0.1/reference/en/html_single/ to learn more about the constraints that we've presented in this lesson.



Time for questions, please. Note that we won't update the slides with the questions that are posed in the lectures. Please, attend them and take notes.



Next week, we have a problem lecture and we need volunteers! Who's first in line?

That's all for today! Thanks for attending this lecture!