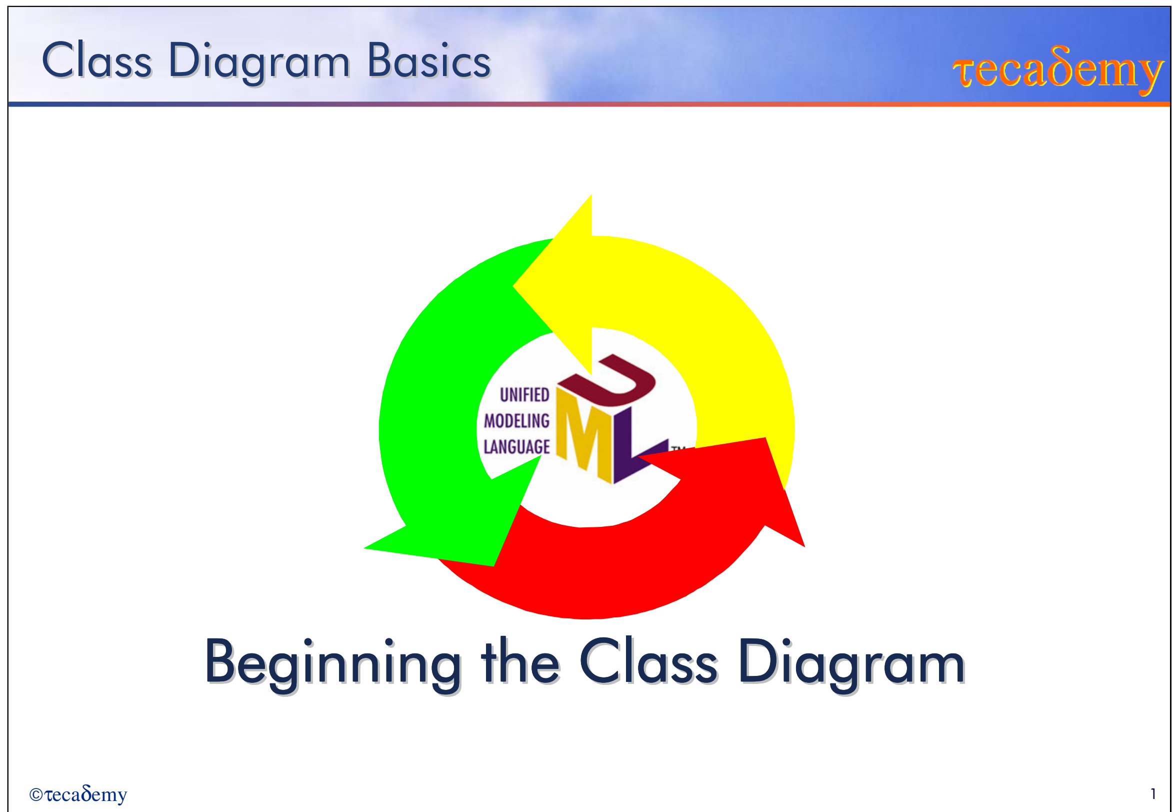

Class Diagram Basics
τecaδemy
UNIFIED
MODELING
LANGUAGE
UML
Beginning the Class Diagram
©τecaδemy
1

Beginning the Class Diagram

τecaδemy

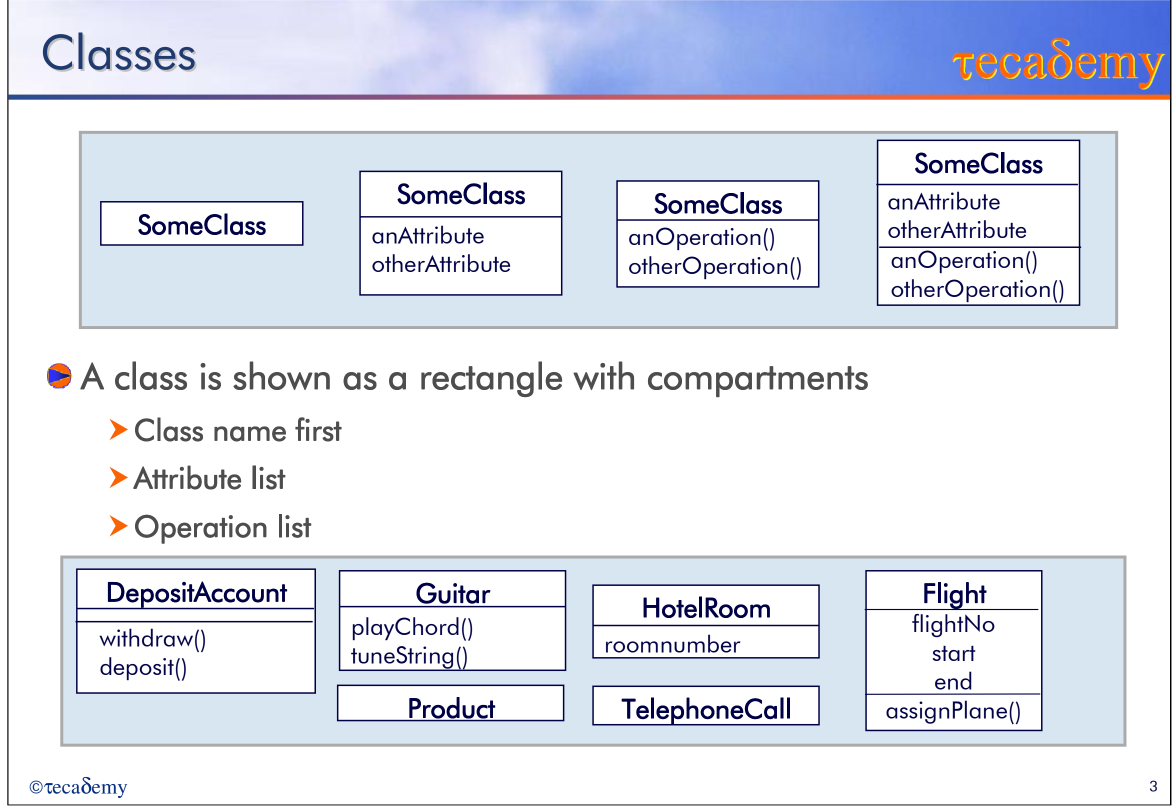
Objectives

- Familiarisation with basic class diagram notation
- Learn to identify class names from the domain vocabulary

Contents

- Class diagram
- Classes
- Attributes
- Operations
- Objects
- Identifying classes
- Heuristics

©τecaδemy



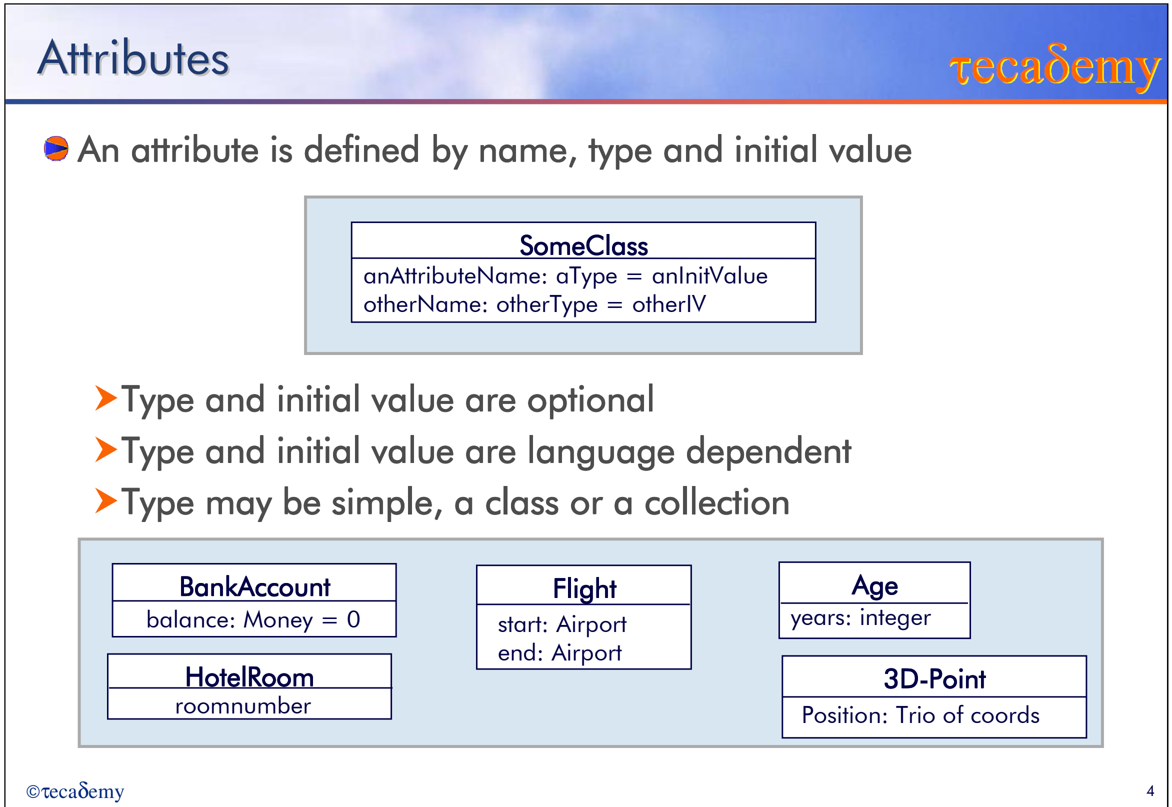
Any of the above examples of a class may be used, depending on the purpose of the diagram. Certainly, early on in the process, classes are typically represented with just their name. Even when the project is finished or in advanced stage, a diagram with classes represented only by name is useful in giving the bigger picture.

Class names should be drawn from the business vocabulary. They should be singular, discrete, countable nouns. Plurals and proper nouns should be avoided.

Showing attributes and operations will help in understanding the role of a class, and in differentiating it from other classes, perhaps with similar names.

There is often a problem with 'real estate', i.e. the amount of space on a screen or sheet of paper. Suppressing attributes and operations compartments enables more classes to be drawn (or more space for relationships or accompanying text). However, beware of putting to much onto a single diagram (even if you can make it fit), as busy or complex diagrams are more difficult to understand, and often make people 'turn off'.Alternatively a compartment may be represented in a skinny fashion when attributes/operations are suppressed.

Named compartments can be used on the class diagram. Typically this is used to depict a fourth compartment showing overall class responsibilities or exceptions thrown by the class. It can in fact be used to add any predefined or user defined model properties e.g. business rules, bean attributes.

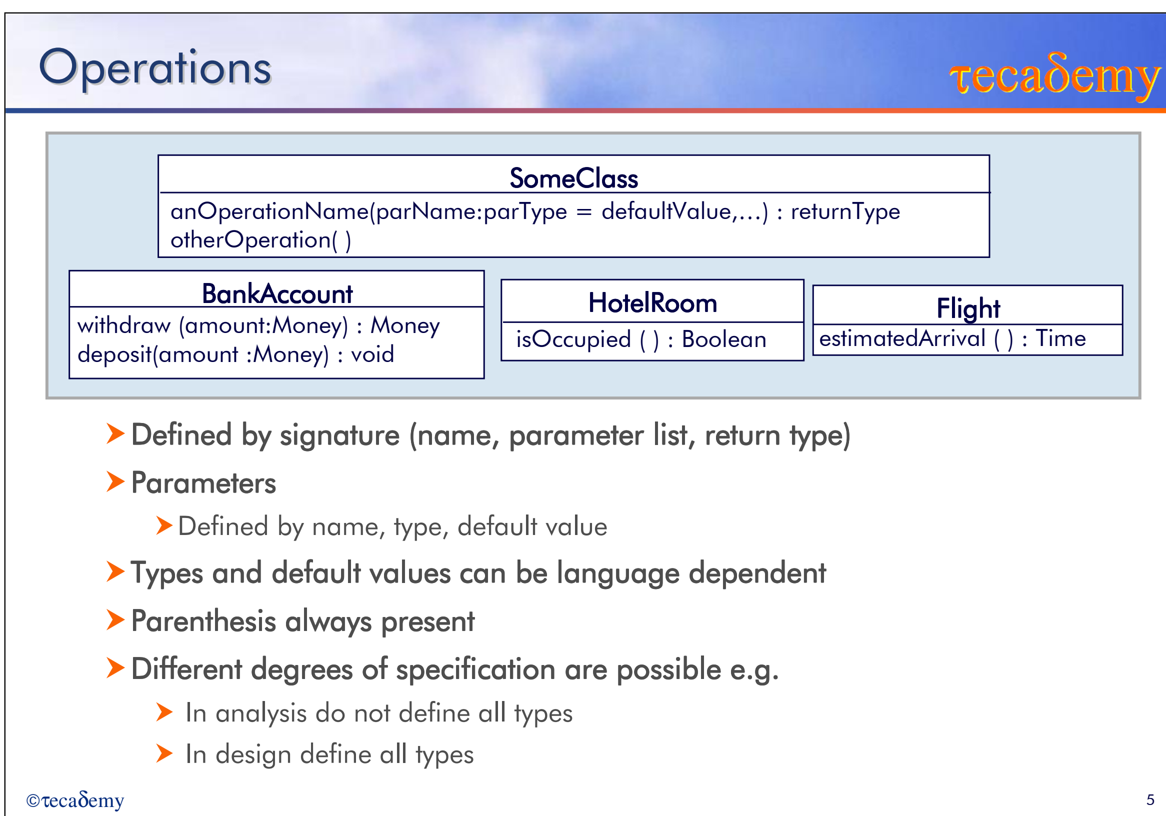


An attribute indicates that an object is responsible for knowing a certain piece of information. The information in an attribute is normally at a lower level of granularity and importance than an object. Thus, a Bank Account object is a bigger and more important thing than the balance.

All the attribute items are shown optionally. That is, a particular diagram may omit a complete attribute, or show just its name, or its name and type, or its name, type and default value.

More importantly, the type and default value may be left undefined in the data dictionary. Ultimately, the type must be defined for implementation. All attributes should be assigned an initial value, either explicitly or through a constructor(create method). This can take place as part of the design process or as part of the implementation.

The types used for attributes should be defined consistently across a project or organisation. They could initially come from a language (such as C/C++ int, char), or from a foundation class library (usually language specific), such as the C++ Standard Template Library, or from a project or company-wide standard definition. They may be provided with generic types, such as number, colour, string, etc., until implementation decisions have been made.

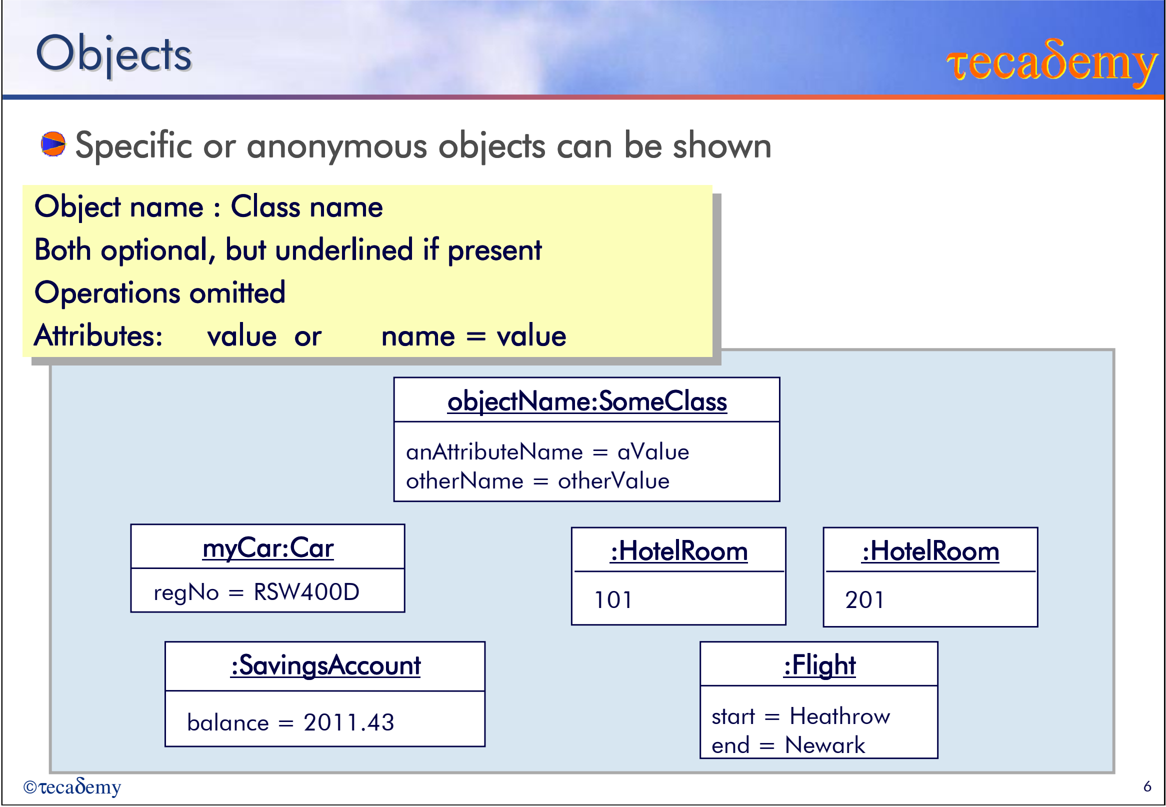


An operation indicates that an object is responsible for providing a certain piece of system functionality. The parameters show information that is passed to the object by the caller of the operation. The return type shows information passed back by the object to the caller.

Each parameter may either be raw information, in the same manner as attributes, or may be a reference to an object. A default value implies that if the sender does not supply the value, the receiver uses the default. (If the implementation language does not support this, the default value is used by the sender.)

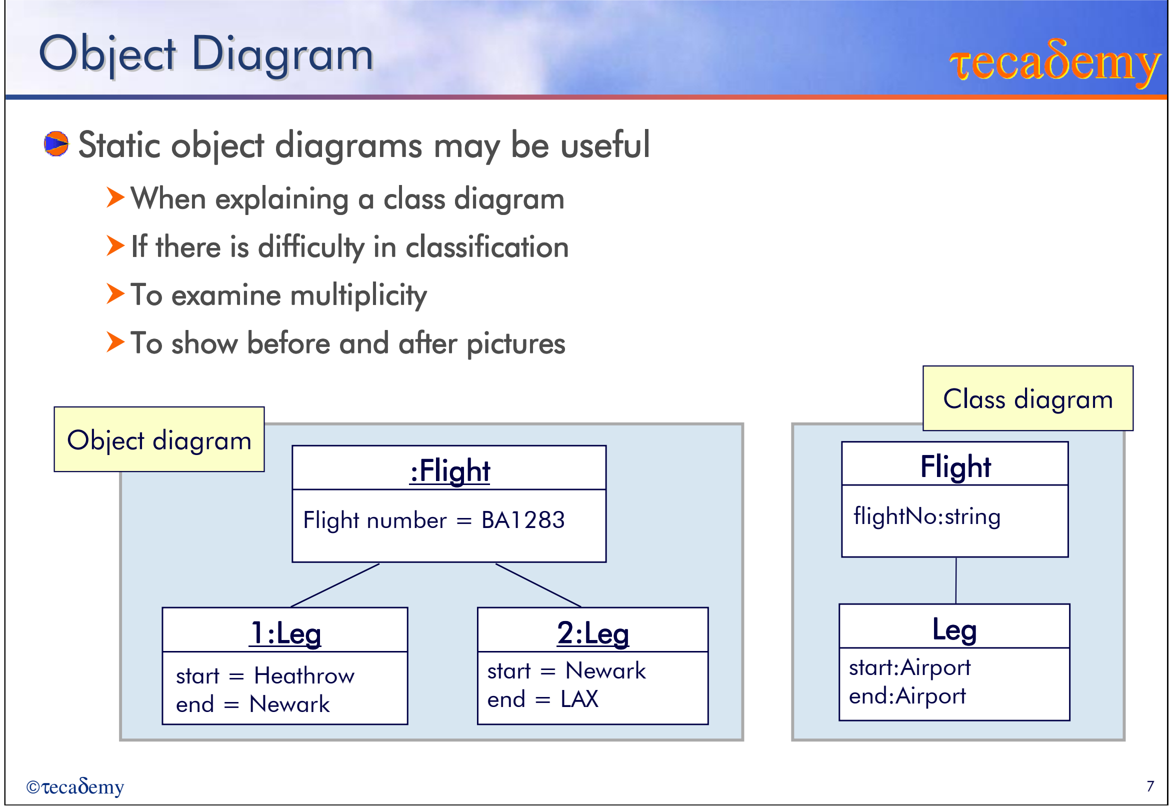
The return is usually constrained to being a single value of the stated type. If more than one value is needed, a new type (class) may be created, and an object of that type is returned.

Both the rendering and existence of parameters, default values and return types are optional, as they are for attributes.



Objects are not often shown in class diagrams, although they are seen in sequence and collaboration diagrams. In general, an instance or example in the UML is underlined, whereas its definition is not. The name of the class should be preceded with a colon when representing an object.

When objects are shown, they can either be a specific example object, with a made-up name, which perhaps represents something in the real world, or an anonymous object that contains just the name of the class. They are usually shown with values for their attributes. Attribute names are normally shown if they are not obvious.



It may be useful to show example objects to clarify a class, to display an association between classes or to examine objects prior to classification. They could be used to provide examples of specific multiplicities; for example, why a flight would ever need more than one leg on its journey. They could also be used for before and after pictures of an operation or a scenario.

Stereotypes

tecaδemy

- Gives extra information about the 'type' of Class
- Typical stereotypes:
  - «Entity» – mainly information-rich, not much dynamic behaviour
  - «Boundary» – lives at system boundary, handles communication with outside world
  - «Control» – lots of behaviour, manages other objects
  - «Wrapper» – encapsulates and provides interface with legacy system
  - «Utility» – set of behaviours to provide a function, no instances
  - Etc. etc… you can create your own.

Client

«entity»
Client

Client

«control»

«boundary»

©tecaδemy 8

To understand what a stereotype is, think about your job. Regardless of your actual job description, what *kind* of job is it? Managerial? Clerical? Front Office? Technical? To say someone is in a clerical job we don't have to know what it is they *actually* do for a living, but we will have a fair idea what kind of work it entails and, importantly, what it does not. We would be surprised to find clerical grades deciding policy or conducting annual appraisals, but those are precisely the types of things we'd expect from managers. The terms 'Clerk' and 'Manager' are *stereotypes* – not job descriptions in themselves, but a kind of classification nonetheless.

Classes have stereotypes too. If a Class name can be thought of as a job description for its instances, then its stereotype tells us a bit more about the kind of behaviour we can expect from it. Assigning a stereotype to a class can help 'fix' the meaning of that class and prevent us from assigning inappropriate behaviours to it.

Notationally, the stereotype is shown in double-angle brackets (called *guillemets*) above the name of a class. Some have their own icons which can be used if preferred.

Some commonly occurring stereotypes are shown in the slide above. They are optional – if an object doesn't readily conform to a stereotype, don't assign one.

Identifying 'Candidate' Classes

tecaδemy

- To provide a base for further examination
- Classes need to be in the domain vocabulary
  - Users are the ultimate arbiters
- Use post-it notes on a wall or whiteboard
- It is a group activity
  - But keep focused on finding the essential classes
  - Keep focused on the use case under investigation

©tecaδemy 9

The purpose of identifying classes is to create a reasonable collection of classes from which more-detailed investigations can start. The collection of classes is a working prototype that expects further detail to be added to classes, as well as new classes to be added and existing classes to be removed. Of course, this prototype is not intended to be thrown away.

Programming language, computer science or other technical terms may not be used during the problem investigation (analysis) stage, unless they are part of the problem domain (for example, writing a compiler).

Identifying classes should be a group activity, as this allows the rapid exchange and refinement of ideas, as well as benefiting from more than one person's knowledge and vocabulary. However, it is easy to become diverted into other topics, such as "What attributes and/or operations does an object have?", "What superclasses/subclasses are there?", "What dynamic behaviour is there?", etc. These example topics are all fine to justify whether to keep or discard a class, but you need to keep on track in order to create an overall model, before delving into detail. Try not to get side-tracked into other Use Cases. Side-tracked discussions will be forgotten, and will then have to be redone while those Use Cases are being done.

Identifying 'Candidate' Classes

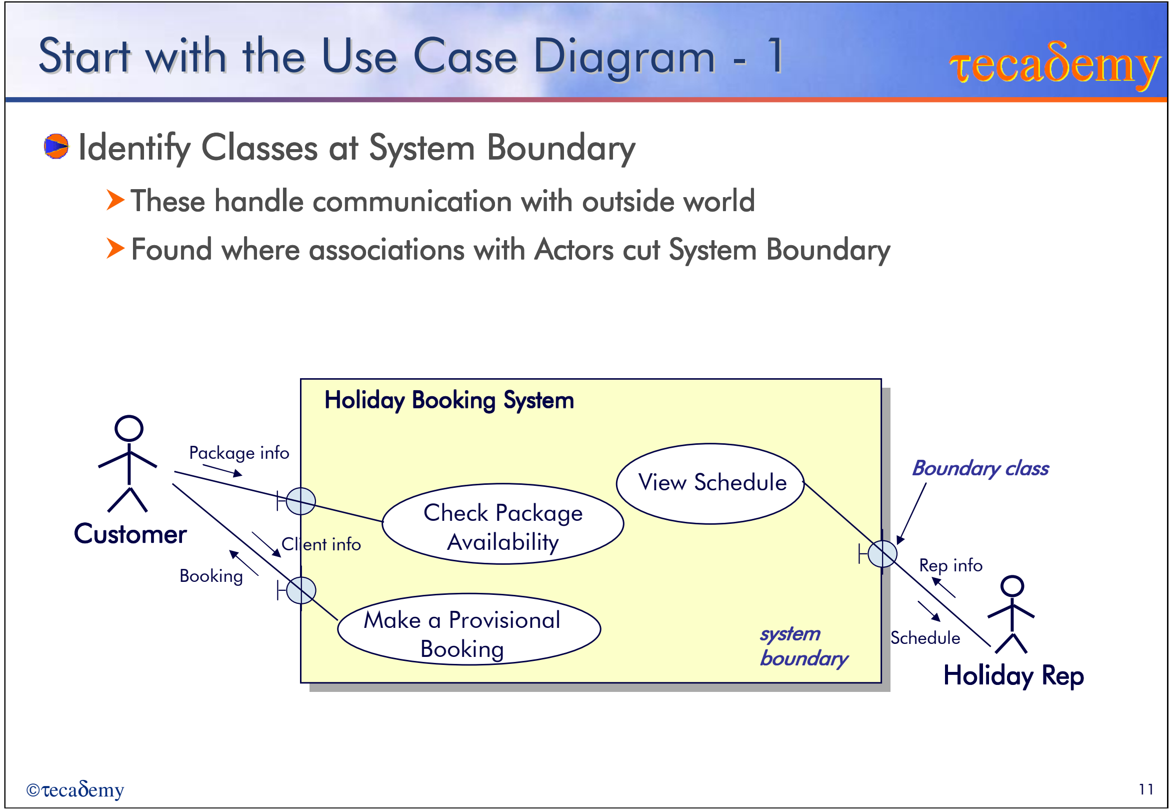
tecaδemy

- Start with Use Case Diagram to find stereotypical Classes
- Then, for each use case:
  - Identify the nouns and noun phrases
  - Choose the best name from synonyms
  - Demote unimportant nouns to attributes
  - Keep terms inside the system boundary
- Finally
  - Brainstorm new relevant classes
  - Make sure each class is defined in the repository

©tecaδemy

10

Each of the points in the above slide will be covered in the forthcoming pages.



The Use Case Diagram can be a rich source of candidate classes. For instance, classes are needed to handle communication to and from the outside world. These classes are called *boundary classes* and are found where associations with Actors cut the system boundary.

Start with the Use Case Diagram - 2

tecaδemy

- Look at information crossing System Boundary
  - If input, entity class needed to hold information
  - If output, entity class needed to produce information

Entity class
Holiday Booking System
Customer
Package info
Client info
Booking
PackageInfo
ClientInfo
Booking
Check Package Availability
Make a Provisional Booking
View Schedule
RepInfo
Schedule
system boundary
Rep info
Schedule
Holiday Rep

©tecaδemy 12

Similarly, investigation of the information passing across the boundary can point to classes needed to hold or produce that information. For example, in the diagram above, if a Holiday Rep supplies his or her details to the system, the system will return the Rep’s tour schedule. The system will need a class to hold an Rep’s details, and a class to hold the schedule.

Start with the Use Case Diagram - 3

τecaδemy

- Entity classes might be needed to hold detail about Actors
  - Actors are *outside* of the system
  - Entity classes are *inside* the system
  - Name classes to emphasise difference

Entity class
Holiday Booking System
Agent
Package info
Client info
Booking
AgentInfo
Check Package Availability
Make a Provisional Booking
View Schedule
RepInfo
Rep info
Schedule
Holiday Rep
system boundary

©τecaδemy 13

Actors are classes that are *outside* of the system boundary, and should not, therefore, be included in our list of candidate classes as these are inside our boundary. However, the system might need information *about* an Actor (e.g. logon information) and this can be held in an entity-type class. The name of this class should reflect the fact that this is *information about an Actor* and not the Actor itself.

Start with the Use Case Diagram - 4

tecaδemy

- Control classes provide behaviour
  - Consider a control class for each use case or group of use cases
  - Behaviour in control classes might be redistributed later

Holiday Booking System
Agent
Package info
Client info
Booking
Check Package Availability
PackageChecker
View Schedule
Scheduler
Make a Provisional Booking
PackageBooker
Control class
system boundary
Rep info
Schedule
Holiday Rep

©tecaδemy 14

A system can be thought of in much the same terms as an organisation. Boundary classes are analogous to the front desk in an office building – they exist to receive messages from or pass messages to the outside world. Entity classes are essentially clerical in function – they are good at filing and retrieving information. But every organisation needs managers to make it function – and from a system point of view, it is the *control class* that manages a process. Since these are described by use cases, consider adding a control class to manage each use case or, where use cases are obviously related, to manage the group.

The use of control classes at this stage can be controversial – some developers like them, some don't. It is often the case that behaviour assigned to these classes might be redistributed at a later stage in development, as commonalities are identified (which might lead to the creation of specialist utility classes, for instance) or to take advantage of behaviour provided by classes in existing libraries (re-use). So don't get too attached to them!

One thing is uncontroversial – *something* has to run things. So there should be at least *one* control class in your list of candidates – even if it is called 'Main'…

Then the Use Cases…

τecaδemy

- Use a small number of related Use Cases
- Identify all the relevant nouns and noun phrases
  - Observe context
  - Highlight, Underline, Copy to Post-it notes
- At this stage, don't do any other processing
  - Removal of duplicates, looking for synonyms, identifying hierarchy
- Provides a good list of candidate classes

©τecaδemy

Start with a small group of Use Cases that have some functional relationship. Then follow the steps on the following pages…

1. Underline Nouns/Noun Phrases

tecaδemy

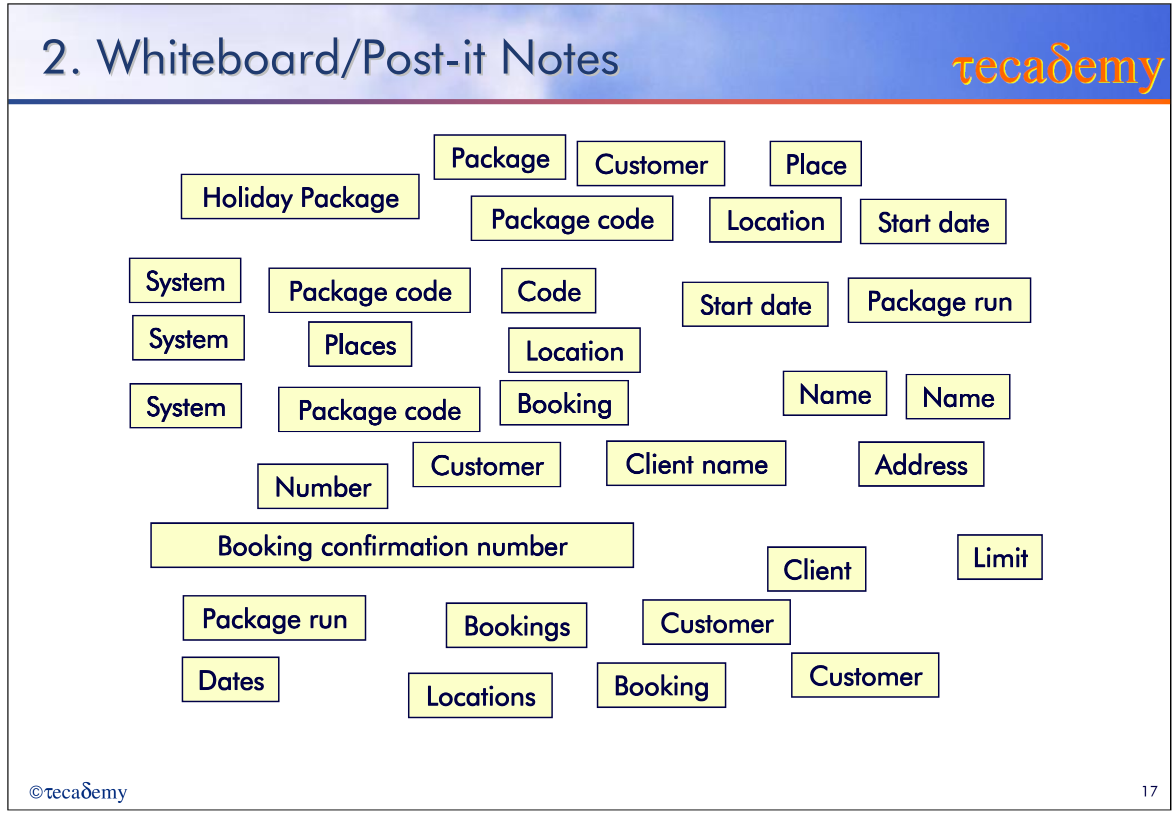
Use Case: Make a Provisional Holiday Booking
Actor: Customer
Flow of Events
1. The Use Case starts when the Customer requests a place on a particular holiday package supplying the package code, location and start date.
2. The system uses the package code, location, and start date to find the specific package run with places available.
3. The system temporarily holds a place.
4. The system notifies the customer that the place is available.
5. The customer accepts place and supplies name and address details.
6. The system makes the temporary held place provisional for that particular client.
7. The system notifies the customer of the booking confirmation number, the use case ends.

©tecaδemy 16

Nouns are the 'things' under investigation. Noun phrases provide more description. Normally, a small number of closely-related Use Cases would be analysed, providing a list of perhaps fifty items.

Creating this list should be kept separate from the future discussions of whether or not to keep specific classes. This ensures that the whole picture can be seen, before making decisions.

Watch for nouns out of context. For instance, in the sentence 'discounts for early payments will be taken into account', the term 'account' could be construed as a noun – rewording the sentence to 'discounts for early payments will be considered' turns the noun into a verb without affecting the underlying semantics. As a rule, keep only the nouns and noun phrases without which the Use Case could not be written.



Write each noun phrase in large writing onto a Post-it note. Place these on a wall or whiteboard, or on a large table. This will let people interact, allow repeated interactions (even unlimited 'undo's and 'redo's), and avoids rewriting long lists.

Use Case: Make a Provisional Holiday Booking

Actor: Customer

Flow of Events

1. The Use Case starts when the Customer requests a place on a particular holiday package supplying the package code, location and start date.

2. The system uses the package code, location, and start date to find the specific package run with places available.

3. The system temporarily holds a place.

4. The system notifies the customer that the place is available.

5. The customer accepts place and supplies name and address details.

6. The system makes the temporary held place provisional for that particular client.

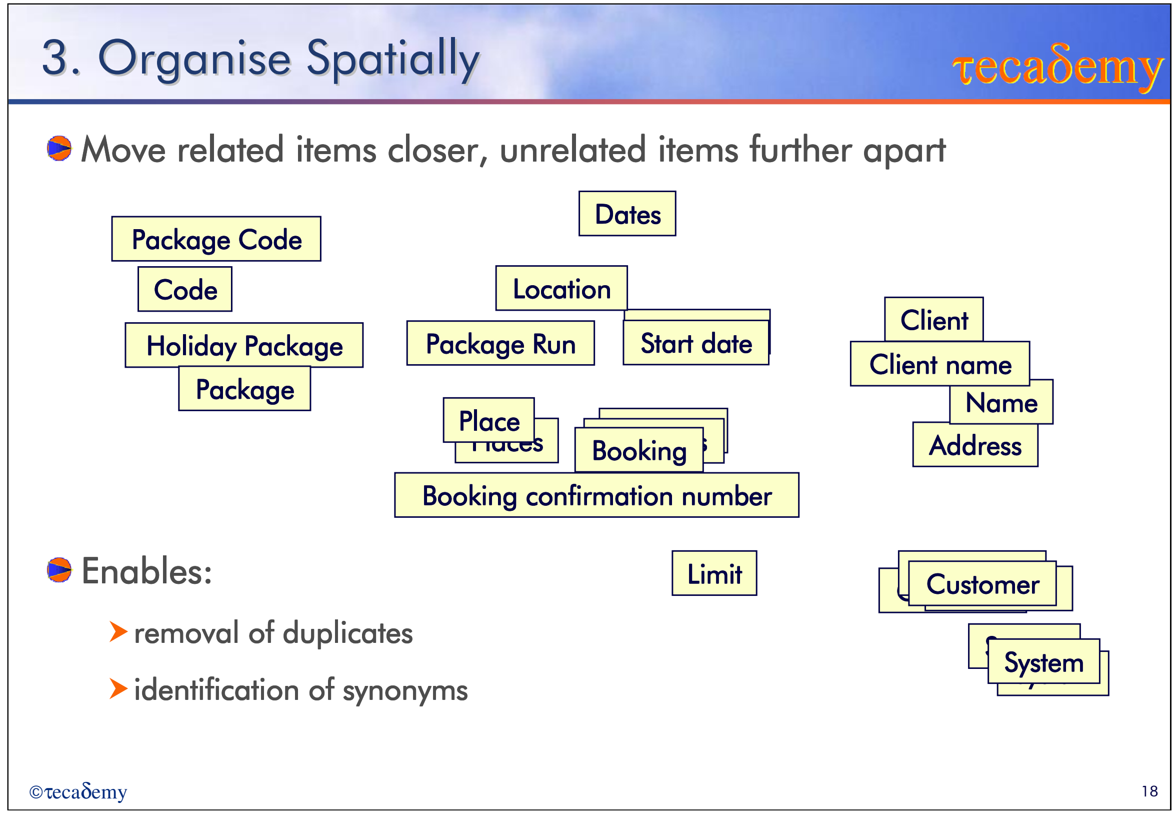
7. The system notifies the customer of the booking confirmation number, the use case ends.

Alternatives

A1. Package run is not found. Customer is allowed to retry.

A2. The number of bookings on the package run has reached the limit. Customer is offered alternative dates and locations.

A3. Customer rejects booking.Temporary booking is cancelled.



Without regard to the particular reason why items are related, try and organise them so that more-closely related items are closer together, and more-loosely related items are further apart. There will never be a single correct answer to this, but some arrangements will appear better in this respect than others. Aim for one that everyone can live with.

The aim is to provide smaller areas that contain more-closely related items, that can be considered in more detail. Specifically, identically-named items should be adjacent, so that duplicates can be easily discarded. Further closely-related items can be investigated, in order to determine whether different names refer to the same concept.

Here follow our reasons why the above items above are close together:

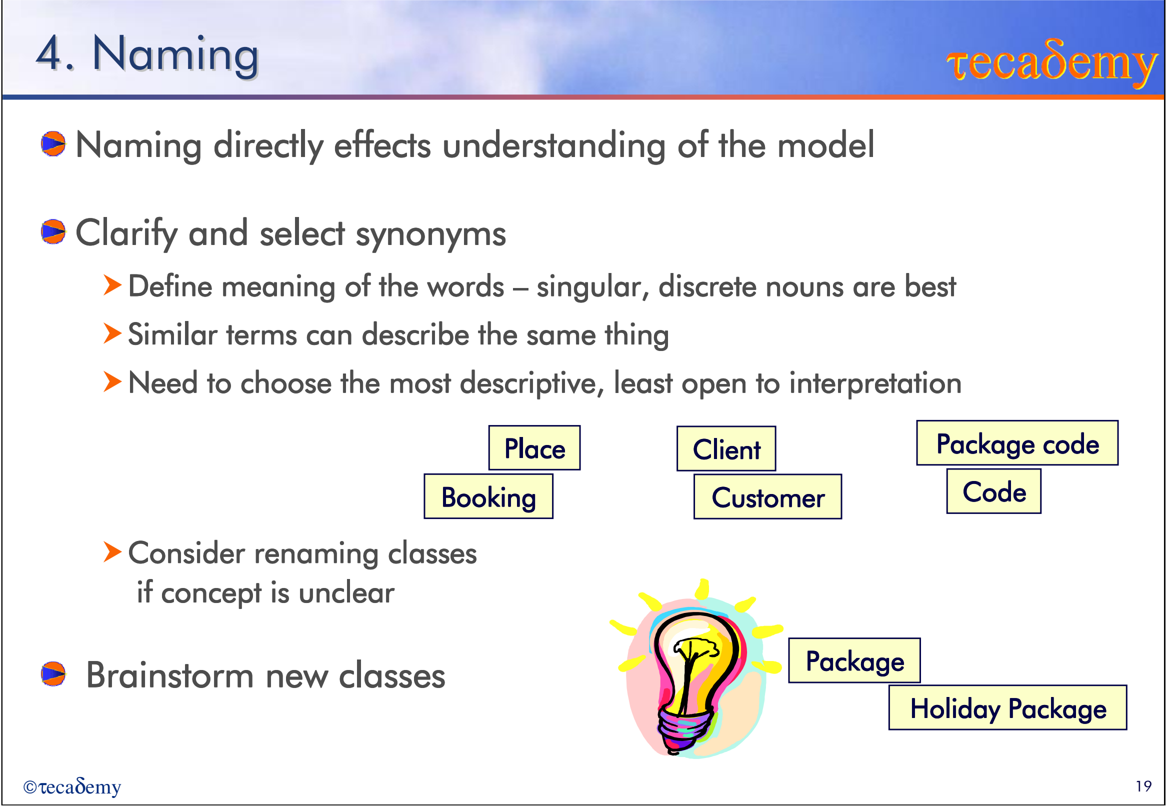
Booking Reference Number, Reference, Number and Place all relate to a booking.

Location and start date specific to a package run

A Client lives at an address

Customer provides client's name

System, Customer, are all physical

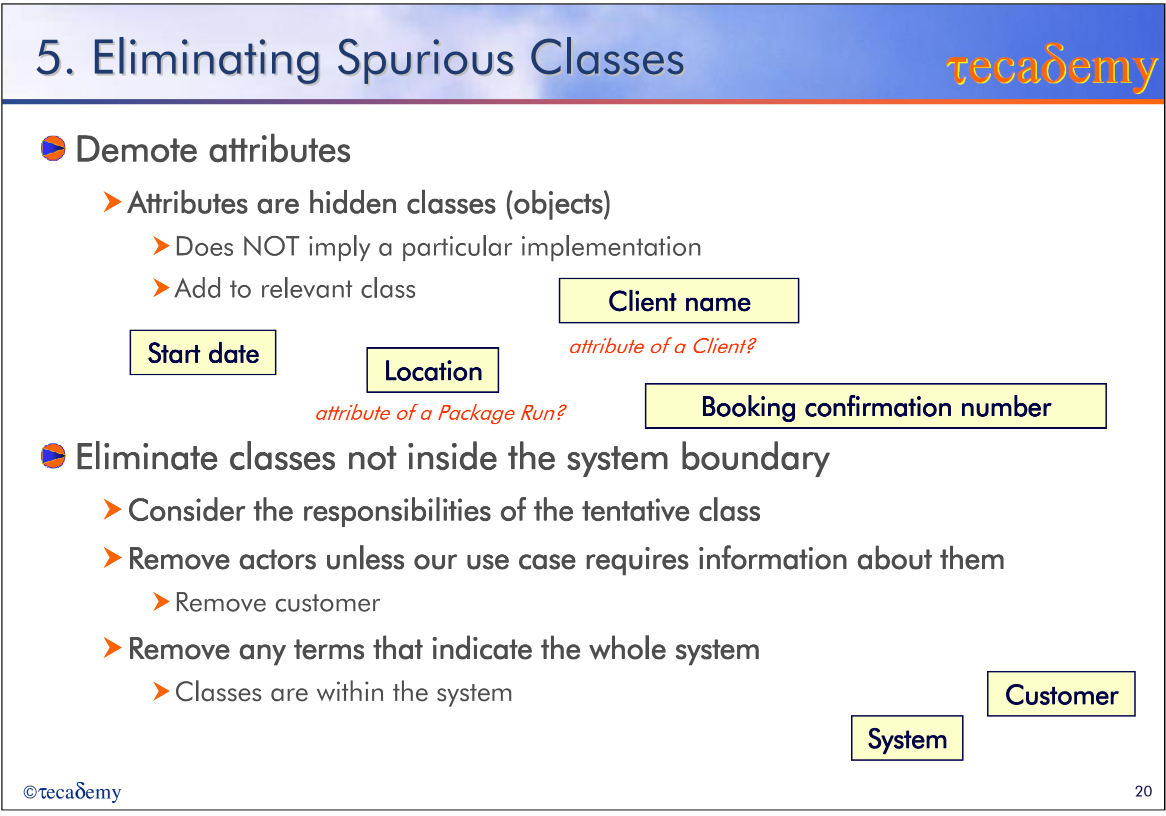


The purpose of the model is to promote understanding. Naming has a direct effect on the way the problem is understood and therefore is critically important in modelling.

During analysis the aim is to find names that are unambiguous and can be understood by domain experts. By using the same concepts as the business domain, the model will be easier to comprehend, maintain and extend.

Try to find classes for which you can identify instances – don't generalise too early. Choose singular nouns rather than plural – try the **'is-a'** test ('Helen **is-a** Client', not 'Helen **is-a** *Clients*'). Discrete (or *countable*) nouns are better than mass nouns – to say 'a Water' doesn't sound right in English – 'a Liquid' does. Proper nouns (like 'Water') also tend to apply to objects rather than classes – Water **is-a** Liquid.

Nouns need not be restricted to those found in specific use cases. System users and developers often have a good grasp of what the system should contain. A good way of capturing this knowledge is to brainstorm ideas to get a list of related nouns. This is best done by simply capturing related nouns, without any judgement, and annotating any items that come to mind, including surreal ones. This list can then be examined according to the same criteria used for nouns from use cases, but with the additional constraint that they should be within the scope of the use cases under consideration. New use cases may be identified and captured. Ensure of course, that any accepted nouns are in user vocabulary.



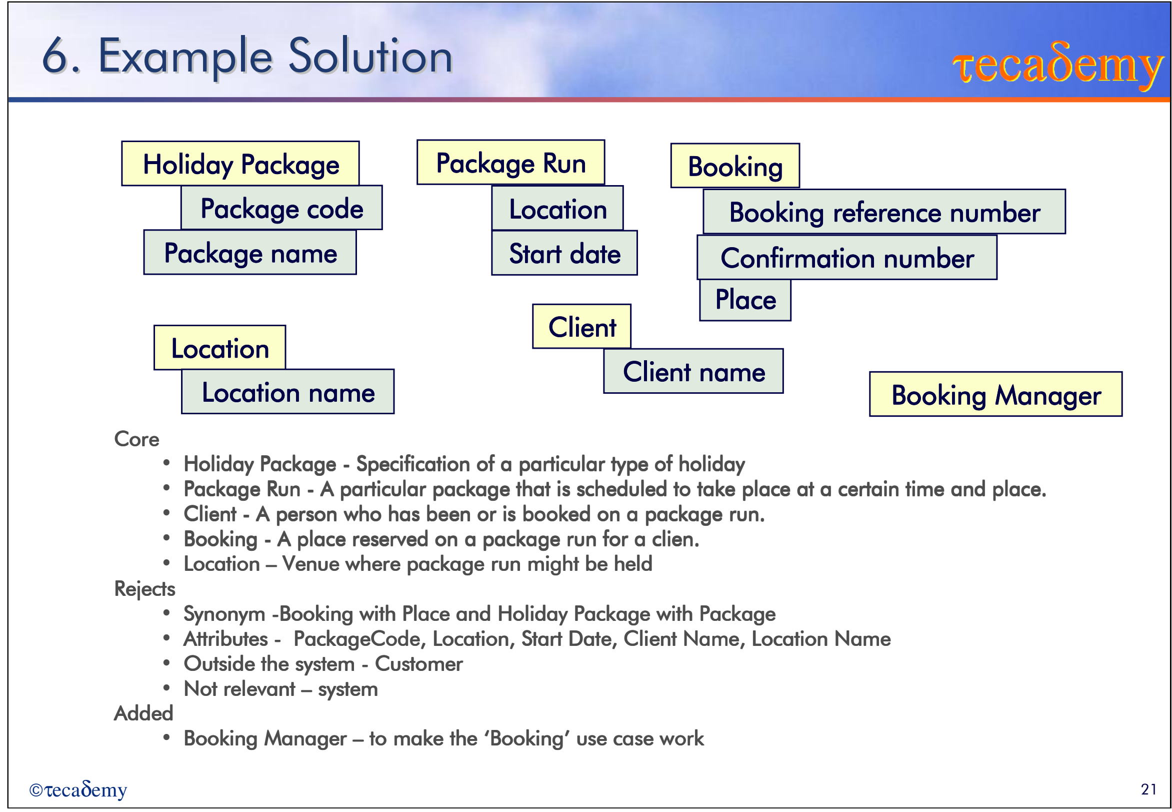
Hiding unimportant nouns as attributes enables better visibility of the first-class nouns that remain, and also ensures that the second-class ones hidden as attributes are not completely forgotten. The attributes may very well return to become classes, when more localised areas of the class diagram are more deeply investigated.

The system boundary documents our understanding of what is inside, and what is outside the scope of the system under investigation. Actors are usually removed from our list of candidate classes. The system does not normally care which real human is interacting with it.

Thus, actors are normally removed, unless in this use case we need information about them. We limit this to use cases under current consideration to avoid premature decision making.

Similarly, any terms that indicate the whole system (System, Booking System) are normally removed, as we are interested in the things inside the system, rather than the enclosure itself. It could be that the system under investigation is wider than realised.

If in any doubt, it is better to keep the item under discussion in play as a candidate class. 'When in doubt, don't chuck it out!'



This slide shows an example solution for a candidate class list for the Course Booking use cases. The selected classes are in white. For each class that is core a one line description show the intent of those classes.

Describing Classes

tecaδemy

- Detailed description of each class usually added to glossary
  - Avoids misunderstandings.
  - May identify new classes/attributes/operations
  - Detailed description added to repository
- Intent - What does an object of the class represent?
- Extent - Which objects are included/excluded?
- Example:

> **ClientInfo «entity»:** Information about a person who has booked on a Package Run. E.g. Helen Backe who is booked on the City Break in Barcelona. Excludes other contacts from organisations who administer the booking in some way; e.g. other travel agents. People who have not been on any holiday but have booked and then cancelled. Includes people who have been on previous holidays.

©tecaδemy

22

Words or phrases in a language can evoke very detailed mental images, but are very often imprecise and open to interpretation to the reader. Class names benefit from the abstraction and encapsulation of the term used, but at the same time suffer from the mental baggage in the reader's mind. To minimise the possibility of misinterpretation, a more-detailed description is required. Additionally, the name itself may be elaborated.

Each class description should define the intent and extent of the class, with examples as necessary. The intent should capture the purpose of the class; what an object of the class represents. The extent should describe the set of objects that belong to the class, and equally importantly, those objects that do not belong to the class. Rules of membership may be explicitly defined.

Detail

τecaδemy

- Don't expect detail at this point
  - May have a few attributes from demoted classes
  - Don't waste time trying to find every last class
- These are only Candidate Classes
  - You probably won't have found them all
  - You'll probably have some you won't need – keep them for now
- Scenarios will help us decide which to keep
- Scenarios will help us find operations and relationships
- Additional attributes will be identified to support operations

©τecaδemy

Class Heuristics

tecaδemy

- Should
  - actively participate in the system
  - use business vocabulary
  - keep related behaviour and data together
  - have a clear intent
  - model real world
  - have unambiguous names
- Typically
  - are named with a singular countable (discrete) noun
  - have a well-defined responsibility

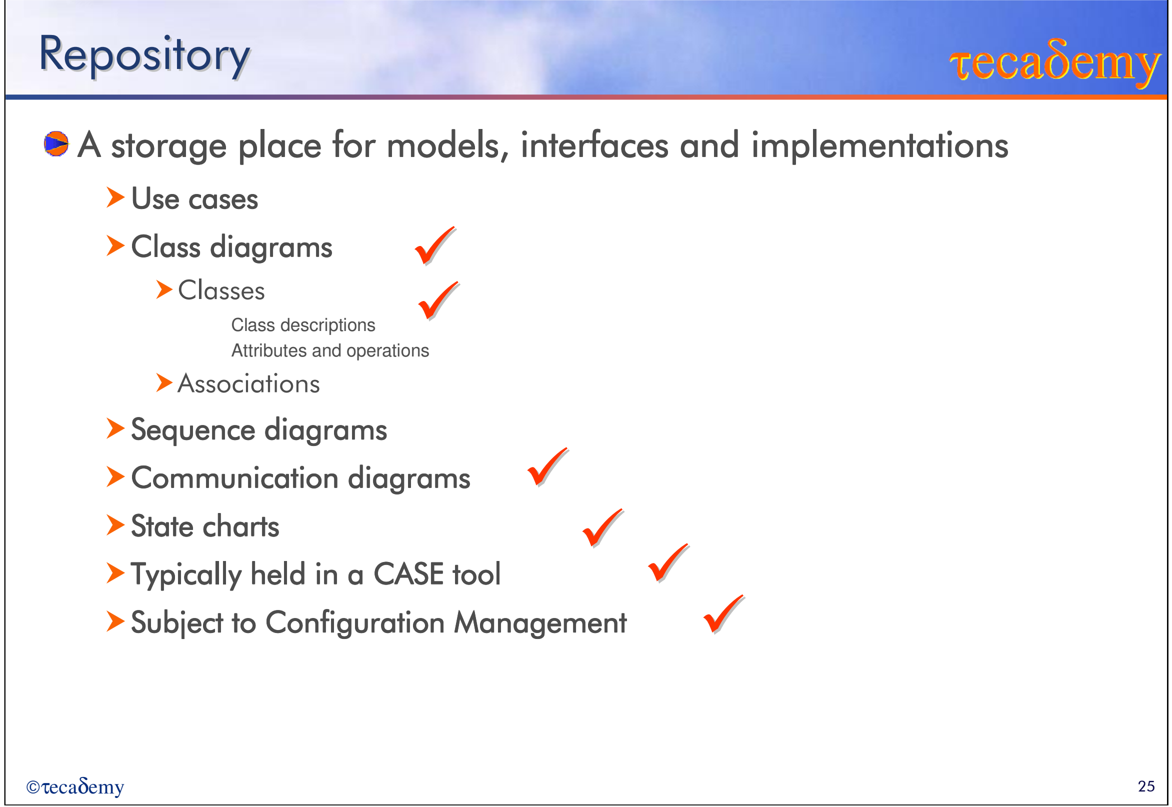
©tecaδemy 24

When identifying classes consider the responsibilities of that Class? What would the real world thing do and is any of that functionality required within the system. By defining the intent of the classes identified, it can be ensured that the whole team or indeed the same person has a consistent view of not only the name of the class but the semantics behind it. This functionality needs to be inside the system boundary.

Choose names carefully, as they will affect the way everyone thinks about and understands the model and eventually the code itself. Choose names that are meaningful and represent the intent of the class. Strive to use words from the business domain. This should not only create a model that is more meaningful in terms of its class names but many of the relationships between the classes will become easier to model and understand too.

As a picture of the class responsibilities and some of the class attributes emerges always remember to keep behaviour and data together; if a responsibility needs some data try to add the responsibility to the class that owns the data. By modelling the real world this will naturally happen. Responsibility assignment can be guided by how the real world thing would be manipulated or interacted with.

Be wary of modelling an operation as a class. For example, 'CreditCardVerifier' is not a good class name. What is it we are verifying ? A better name for the class would be the thing we are verifying, e.g. the Credit Card.



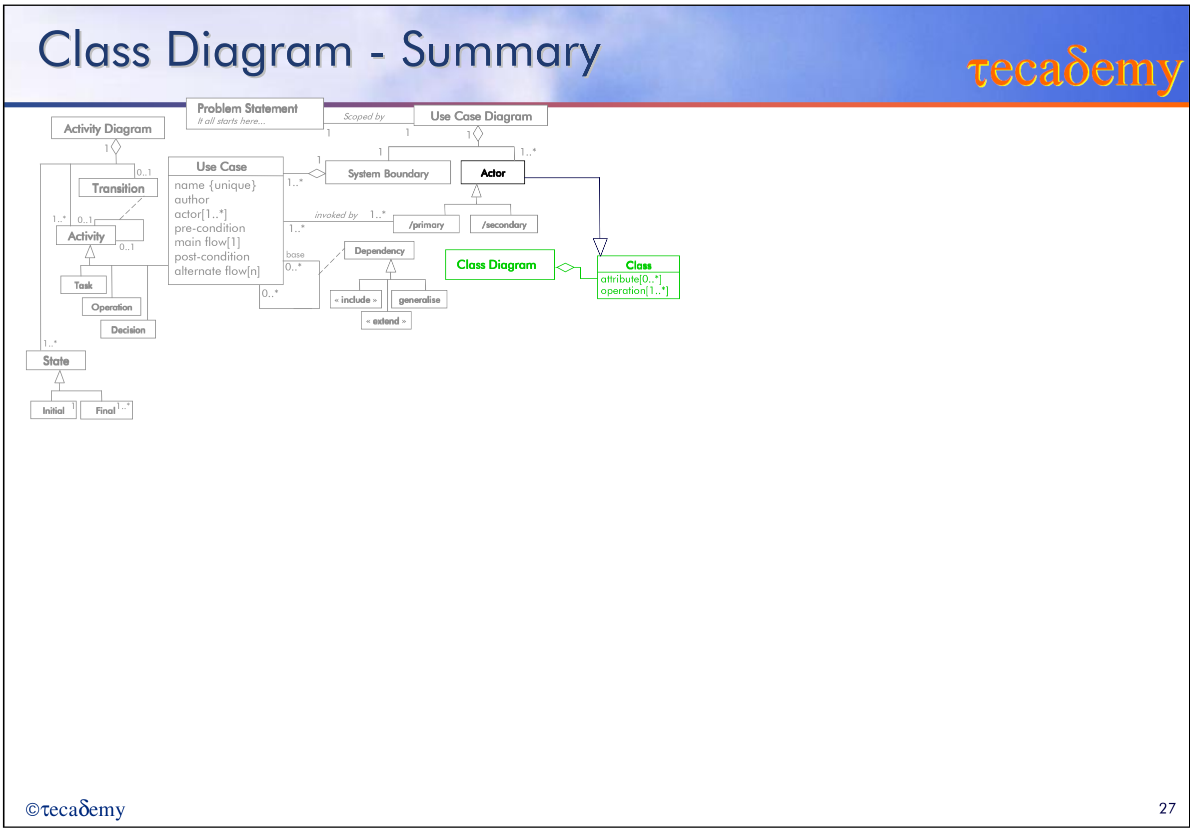
The graphical models cannot hope to visualise everything about a system. They are simply views of some of the information held in the repository.

Chapter Summary

τεcaδemy

- UML provides notation for classes, attributes and operations
- UML also provides a notation for objects
- Classes can be found by analysing use cases
- Class names should be meaningful and use domain vocabulary – but, when in doubt, don't chuck it out!
- Don't expect too much detail after the initial identification of classes
- Analysis/design is an evolutionary process, so expect changes

©τεcaδemy



Use cases can be a rich source of domain-specific terms and concepts, some of which could be regarded as candidates for class status. Some terms might indicate properties of a class; an attribute by a noun, an operation by a verb. These can be shown on the initial *class diagram* and serve as the basis for discussions aimed at discovering other classes.

Actors are classes, too; but they are classes that exist outside of our system boundary and are therefore out of scope. They are not included in a class diagram except for contextual purposes.

Case Study – Exercise 7

τecaδemy

- Objective
  - To practice finding 'Candidate' Classes.
- Turn to and complete exercise 7 in the exercise booklet.

©τecaδemy