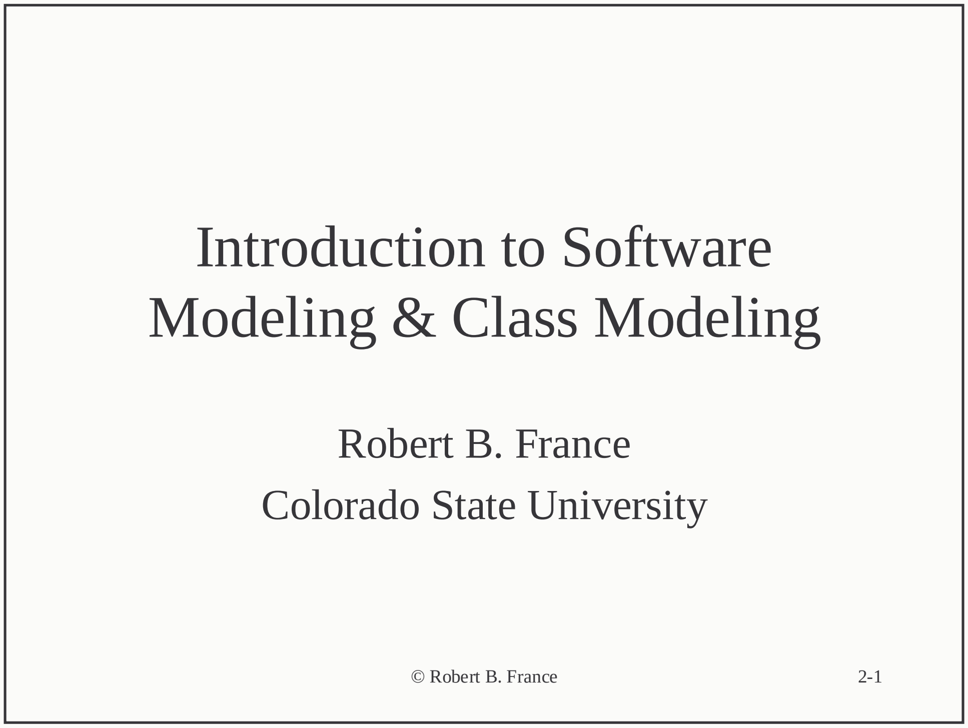
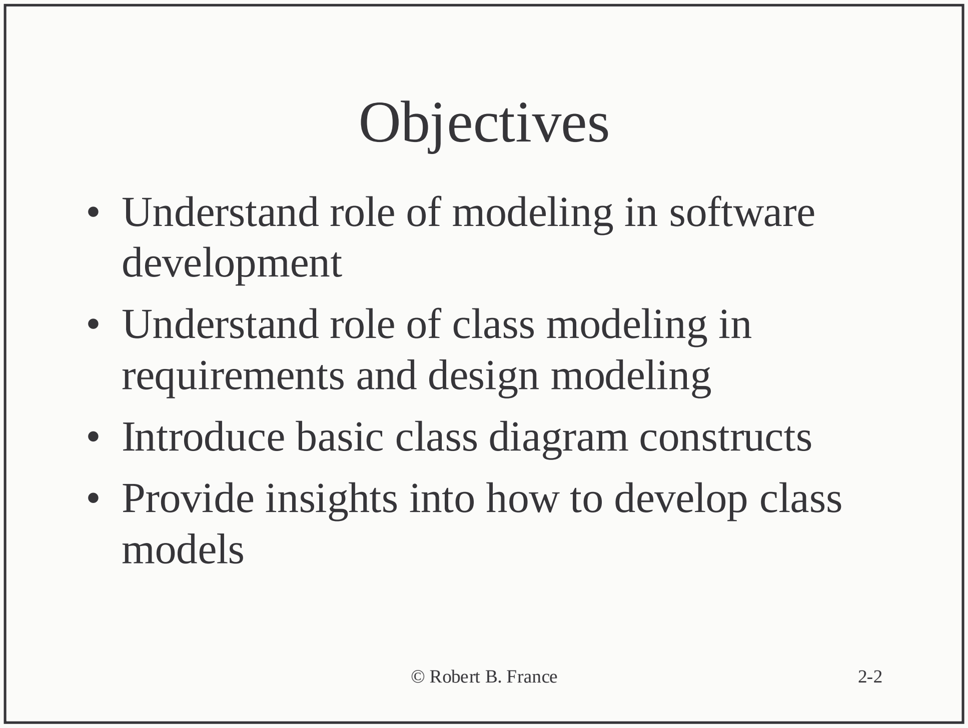


Introduction to Software Modeling & Class Modeling

Robert B. France

Colorado State University

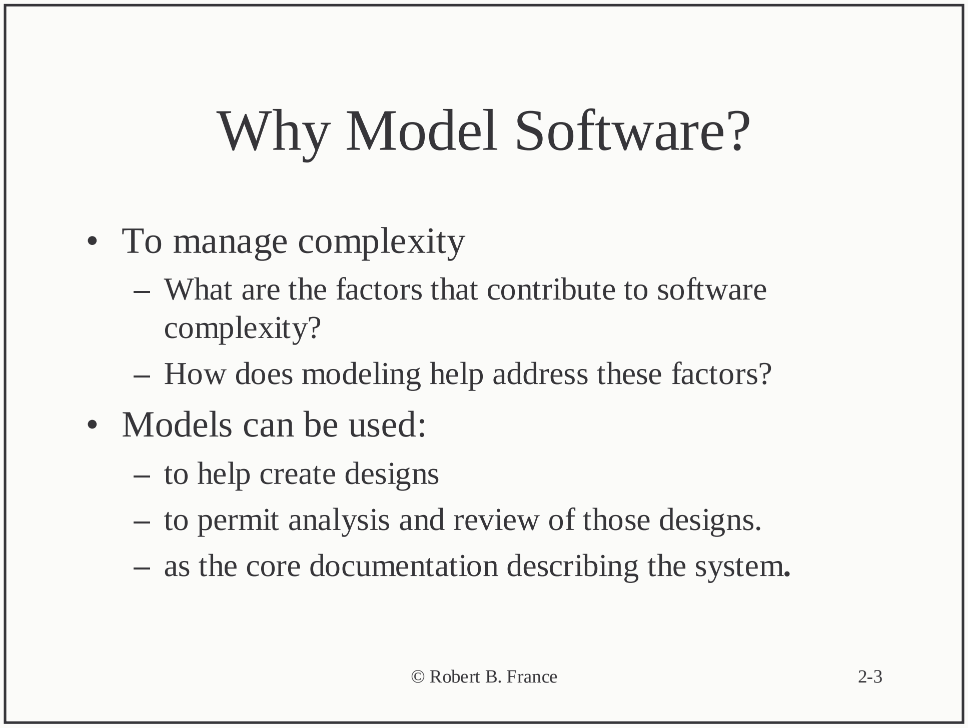
© Robert B. France



Objectives

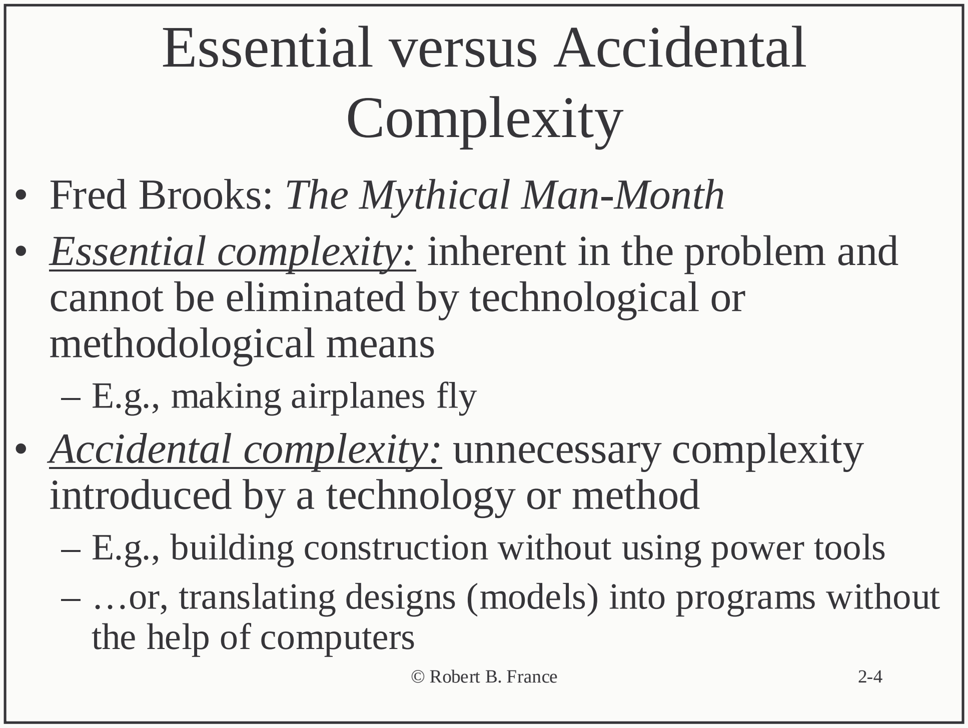
- Understand role of modeling in software development
- Understand role of class modeling in requirements and design modeling
- Introduce basic class diagram constructs
- Provide insights into how to develop class models

© Robert B. France



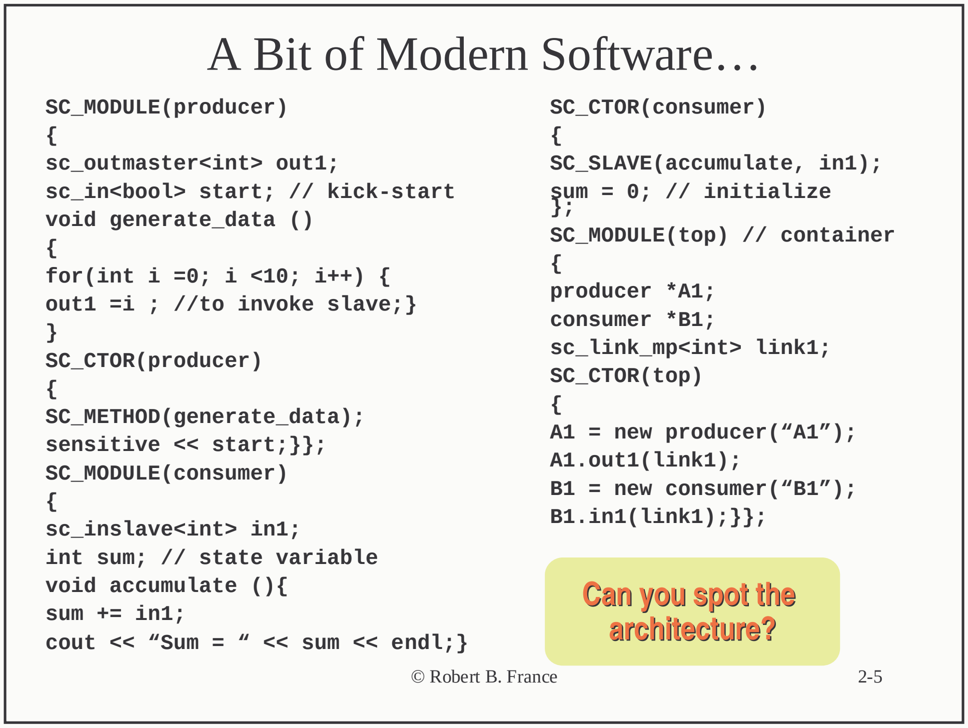
Why Model Software?

- To manage complexity
  - What are the factors that contribute to software complexity?
  - How does modeling help address these factors?
- Models can be used:
  - to help create designs
  - to permit analysis and review of those designs.
  - as the core documentation describing the system.



Essential versus Accidental Complexity

- Fred Brooks: *The Mythical Man-Month*
- *Essential complexity:* inherent in the problem and cannot be eliminated by technological or methodological means
  - E.g., making airplanes fly
- *Accidental complexity:* unnecessary complexity introduced by a technology or method
  - E.g., building construction without using power tools
  - …or, translating designs (models) into programs without the help of computers

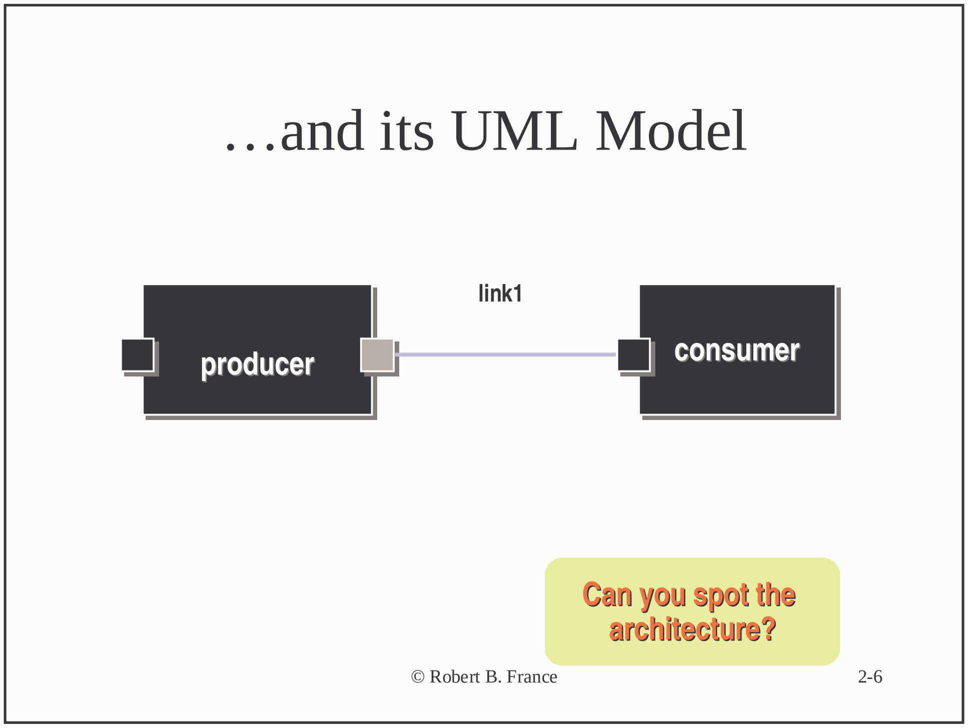


A Bit of Modern Software…

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << “Sum = “ << sum << endl;}
```

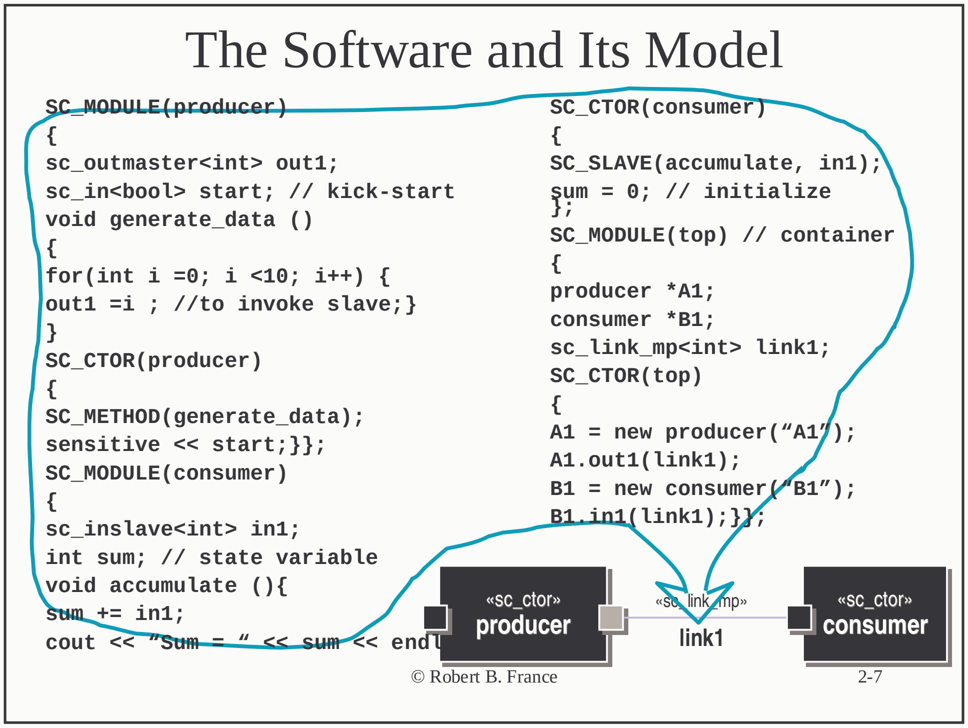
```
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer(“A1”);
A1.out1(link1);
B1 = new consumer(“B1”);
B1.in1(link1);}};
```

Can you spot the architecture?



…and its UML Model

Can you spot the architecture?

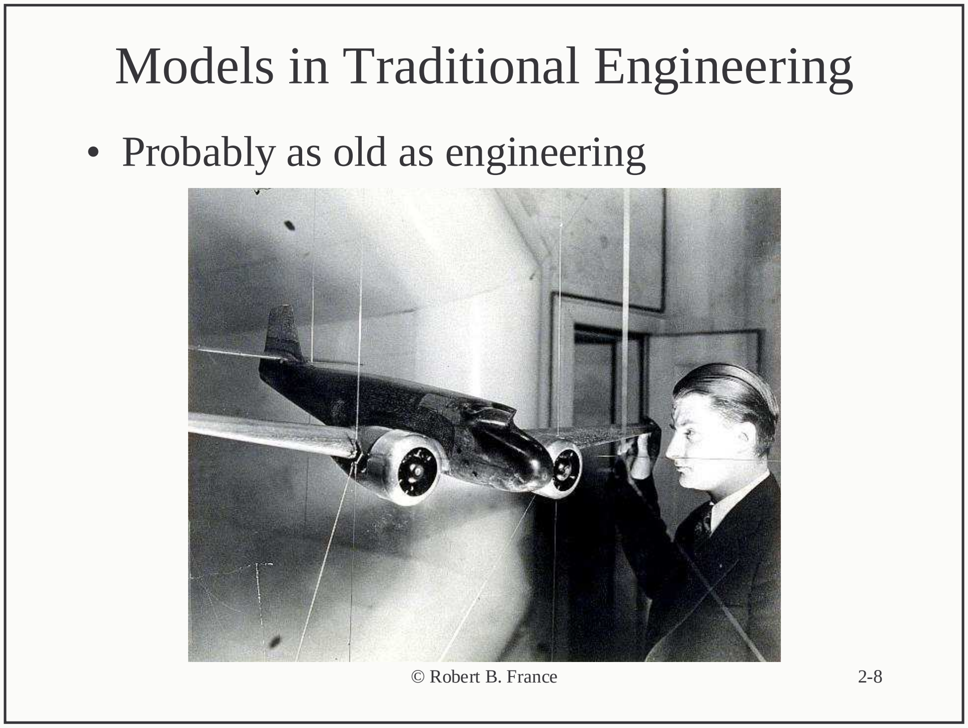


The Software and Its Model

```
SC_MODULE(producer)
{
sc_outmaster<int> out1;
sc_in<bool> start; // kick-start
void generate_data ()
{
for(int i =0; i <10; i++) {
out1 =i ; //to invoke slave;}
}
SC_CTOR(producer)
{
SC_METHOD(generate_data);
sensitive << start;}};
SC_MODULE(consumer)
{
sc_inslave<int> in1;
int sum; // state variable
void accumulate (){
sum += in1;
cout << "Sum = " << sum << endl
```

```
SC_CTOR(consumer)
{
SC_SLAVE(accumulate, in1);
sum = 0; // initialize
};
SC_MODULE(top) // container
{
producer *A1;
consumer *B1;
sc_link_mp<int> link1;
SC_CTOR(top)
{
A1 = new producer("A1");
A1.out1(link1);
B1 = new consumer("B1");
B1.in1(link1);}};
```

© Robert B. France 2-7



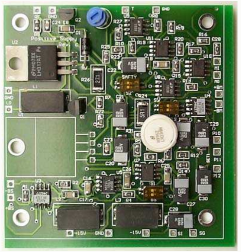
Models in Traditional Engineering

- Probably as old as engineering

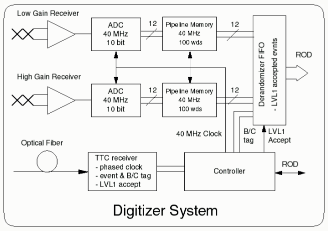
© Robert B. France 2-8

Engineering Models

- Engineering model:
  A reduced representation of some system that highlights the properties of interest from a given viewpoint



Modeled system

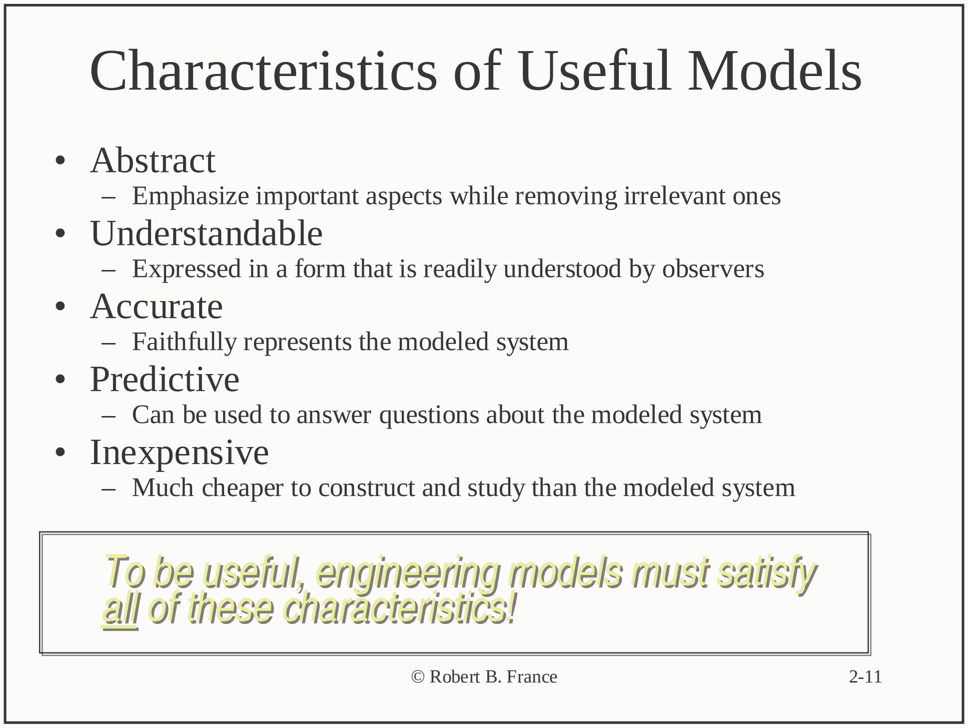


Functional Model

- We don't see everything at once
- We use a representation (notation) that is easily understood for the purpose on hand

How Engineering Models are Used

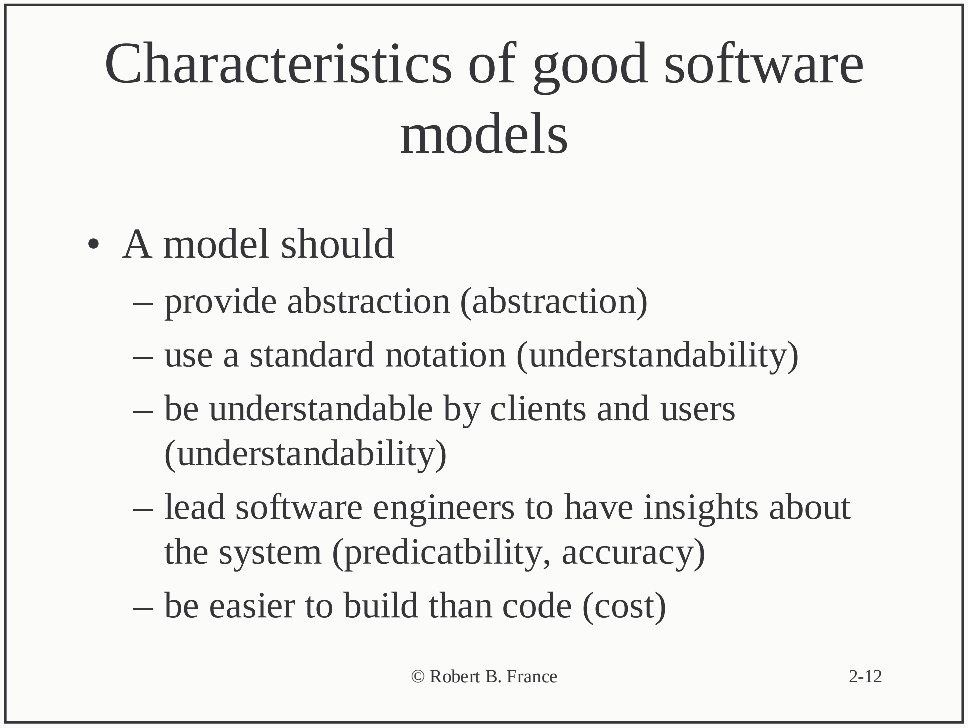
1. To help us understand complex systems
   - Useful for both *requirements* and *designs*
   - Minimize risk by detecting errors and omissions early in the design cycle (at low cost)
     - Through analysis and experimentation
     - Investigate and compare alternative solutions
   - To communicate understanding
     - Stakeholders: Clients, users, implementers, testers, documenters, etc.
2. To drive implementation
   - The model as a blueprint for construction



Characteristics of Useful Models

- Abstract
  - Emphasize important aspects while removing irrelevant ones
- Understandable
  - Expressed in a form that is readily understood by observers
- Accurate
  - Faithfully represents the modeled system
- Predictive
  - Can be used to answer questions about the modeled system
- Inexpensive
  - Much cheaper to construct and study than the modeled system

To be useful, engineering models must satisfy all of these characteristics!



Characteristics of good software models

- A model should
  - provide abstraction (abstraction)
  - use a standard notation (understandability)
  - be understandable by clients and users (understandability)
  - lead software engineers to have insights about the system (predicatbility, accuracy)
  - be easier to build than code (cost)

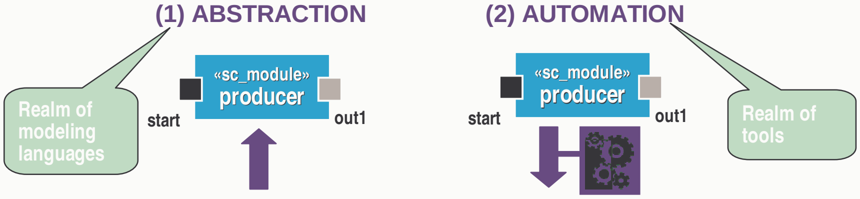
The Remarkable Thing About Software

Software has the rare property that it allows us to directly evolve models into full-fledged implementations without changing the engineering medium, tools, or methods!

The model evolves into the system it was modeling

Model-Driven Style of Development (MDD)

- An approach to software development in which the focus and primary artifacts of development are models (as opposed to programs)
- Based on two time-proven methods

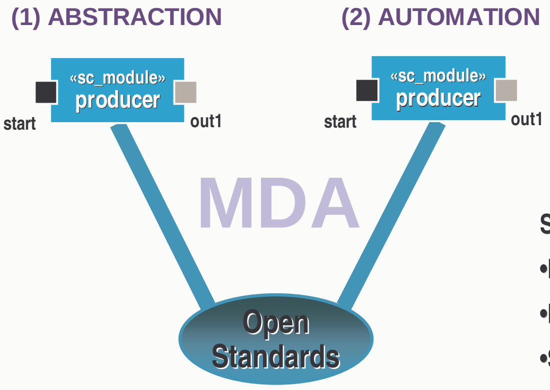


```
SC_MODULE(producer)
{sc_inslave<int> in1;
int sum; //
void accumulate (){
sum += in1;
cout << "Sum = " <<
sum << endl;}
```

```
SC_MODULE(producer)
{sc_inslave<int> in1;
int sum; //
void accumulate (){
sum += in1;
cout << "Sum = " <<
sum << endl;}
```

OMG’s Model-Driven Architecture (MDA)

- An OMG initiative
  - A framework for a set of *open* standards in support of MDD



Standards for:

•Modeling languages

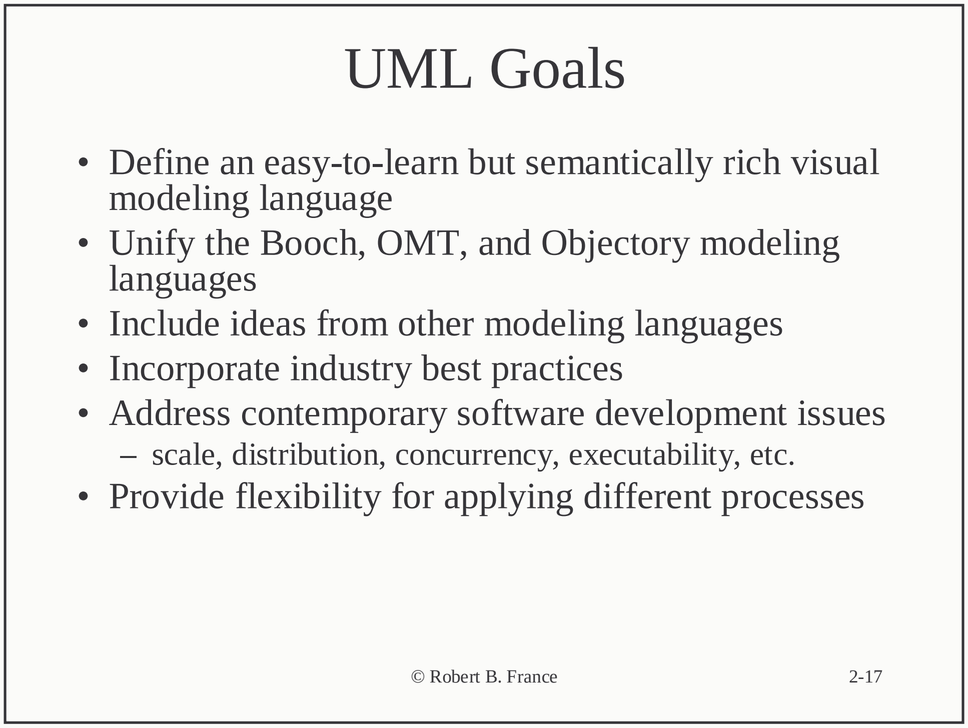
•Model transformations

•Software processes

•Model interchange…

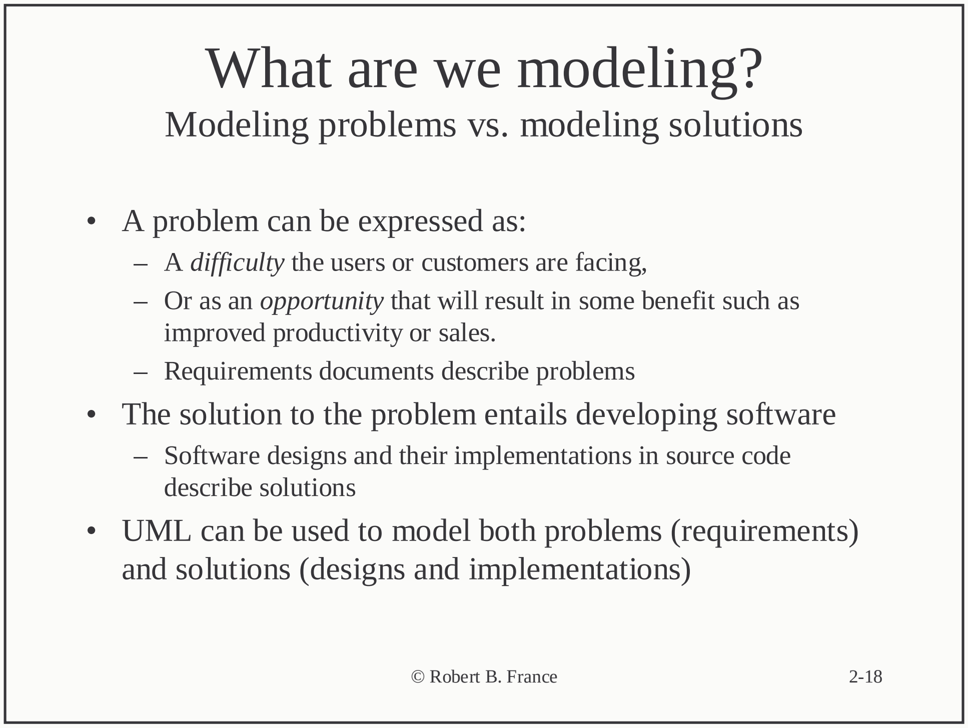
The Unified Modeling Language

- The UML is standard diagramming language to visualize the results of analysis and design.
- UML is a tool
  - Learning how to create high-quality models is not equivalent to learning the UML
  - UML is simply a language for expressing models
- The UML is *not*
  - a process or methodology
  - an object-oriented analysis and design technique
  - a modeling technique



UML Goals

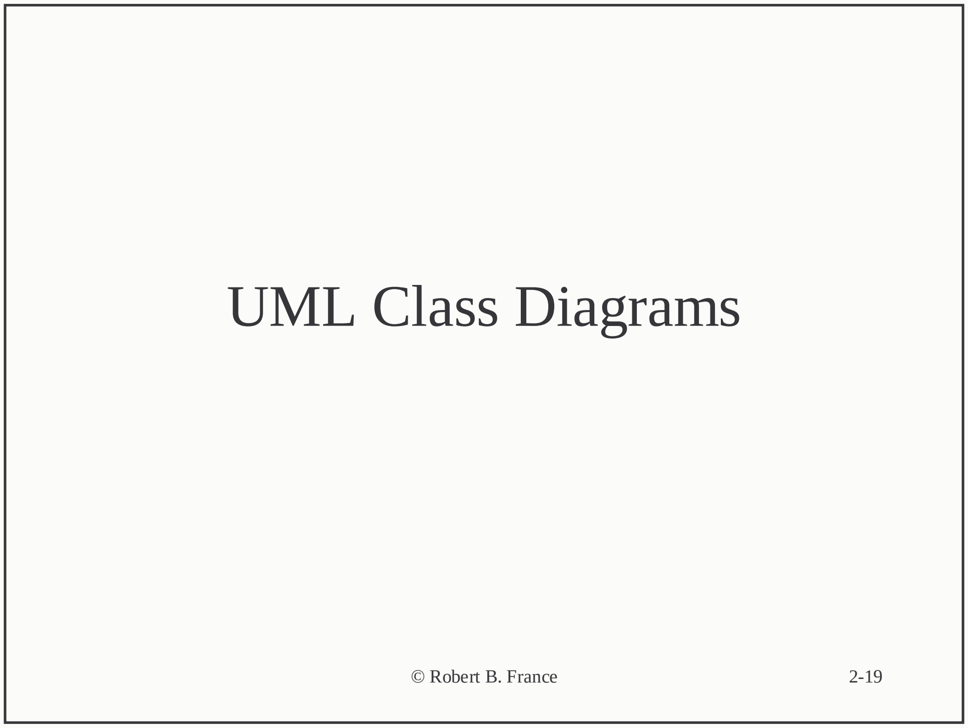
- Define an easy-to-learn but semantically rich visual modeling language
- Unify the Booch, OMT, and Objectory modeling languages
- Include ideas from other modeling languages
- Incorporate industry best practices
- Address contemporary software development issues
  - scale, distribution, concurrency, executability, etc.
- Provide flexibility for applying different processes



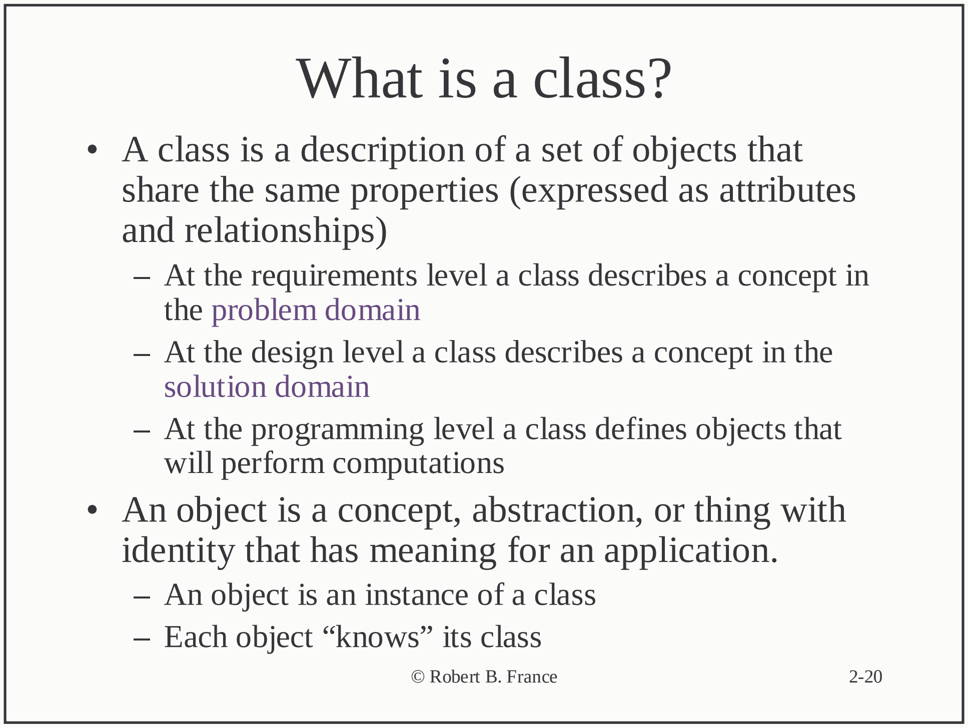
What are we modeling?

Modeling problems vs. modeling solutions

- A problem can be expressed as:
  - A *difficulty* the users or customers are facing,
  - Or as an *opportunity* that will result in some benefit such as improved productivity or sales.
  - Requirements documents describe problems
- The solution to the problem entails developing software
  - Software designs and their implementations in source code describe solutions
- UML can be used to model both problems (requirements) and solutions (designs and implementations)



UML Class Diagrams



What is a class?

- A class is a description of a set of objects that share the same properties (expressed as attributes and relationships)
  - At the requirements level a class describes a concept in the problem domain
  - At the design level a class describes a concept in the solution domain
  - At the programming level a class defines objects that will perform computations
- An object is a concept, abstraction, or thing with identity that has meaning for an application.
  - An object is an instance of a class
  - Each object “knows” its class

What is a Class Model?

- Syntactically, a class model is a structure of classes.
- Semantically,
  - a requirements class model describes problem concepts and their relationships
  - a design class model describes solution concepts and their relationships
  - an implementation class model describes program-level objects (e.g., Java objects) and their links
- Key Question: What are the objects of interest in the problem/solution space?
  - their properties (in terms of attributes and operations)?
  - their relationships?

An example of a requirements class diagram

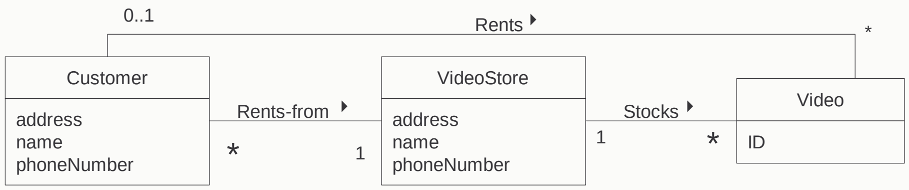
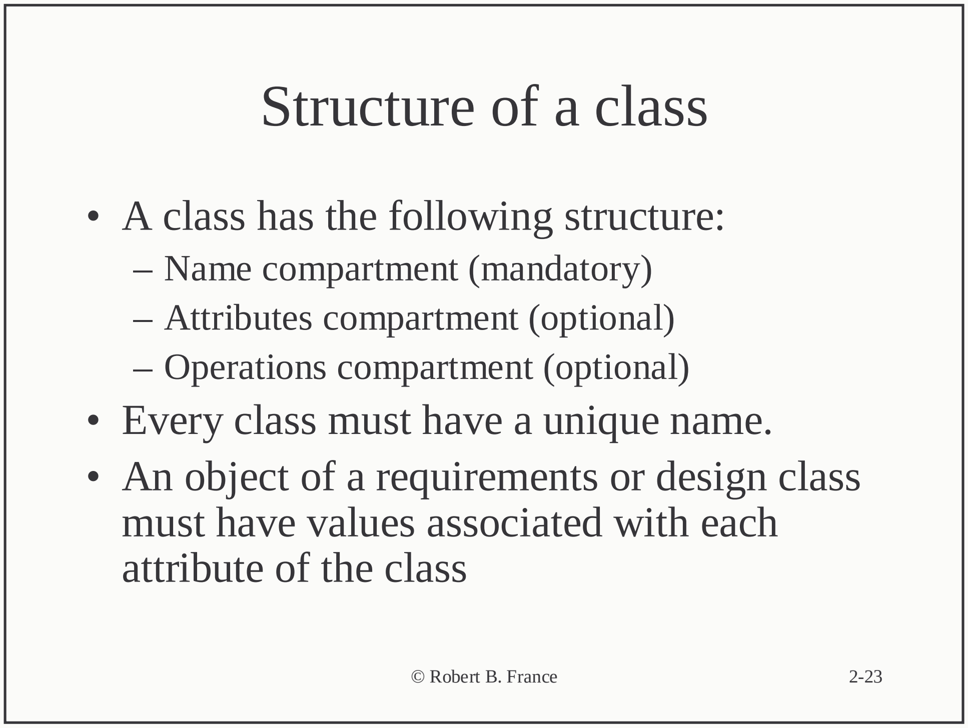
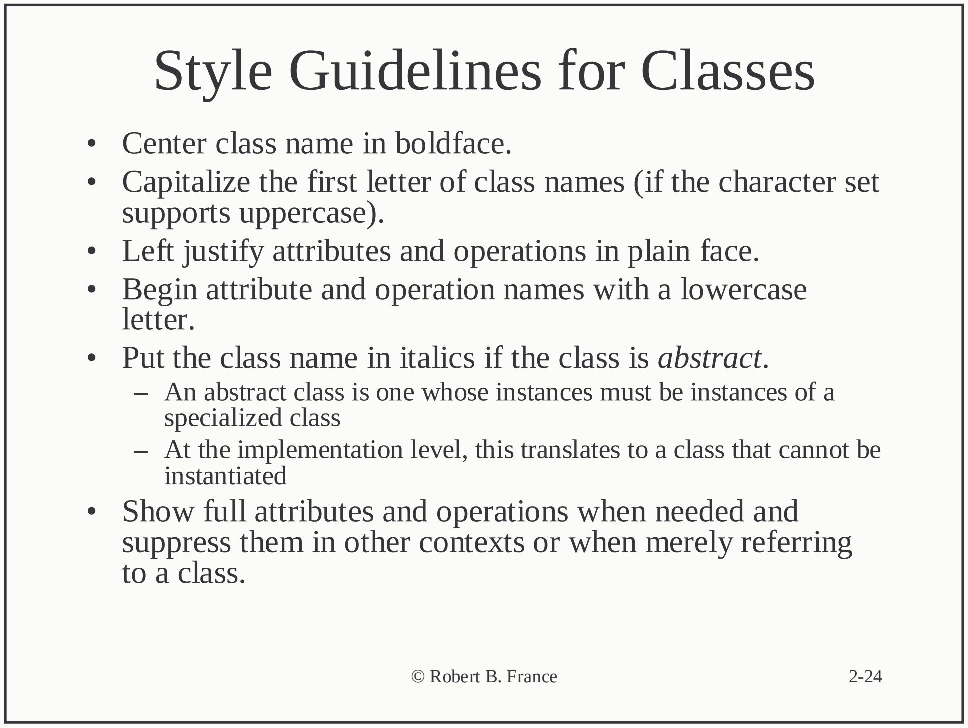


Diagram vs. model: A diagram can be a partial view of a model; large models can be described using multiple models



Structure of a class

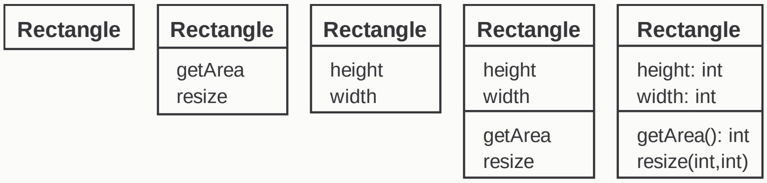
- A class has the following structure:
  - – Name compartment (mandatory)
  - – Attributes compartment (optional)
  - – Operations compartment (optional)
- Every class must have a unique name.
- An object of a requirements or design class must have values associated with each attribute of the class



Style Guidelines for Classes

- Center class name in boldface.
- Capitalize the first letter of class names (if the character set supports uppercase).
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Put the class name in italics if the class is *abstract*.
  - – An abstract class is one whose instances must be instances of a specialized class
  - – At the implementation level, this translates to a class that cannot be instantiated
- Show full attributes and operations when needed and suppress them in other contexts or when merely referring to a class.

Depicting Classes



Rectangle

Rectangle

getArea
resize

Rectangle

height
width

Rectangle

height
width

getArea
resize

Rectangle

height: int
width: int

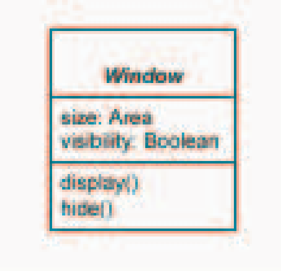
getArea(): int
resize(int,int)

Attributes

- An attribute is a named property. Each class instance associates value(s) with each attribute of a concept.

What should be an attribute?

- Properties with types that we want to treat as primitive are modeled as attributes
- Connections to other concepts are to be represented as associations, not attributes.



Operations

- An operation is a procedure that may be applied to or by objects in a class
- Each operation has a signature and a specification of its behavior.
  - Signature: operation name, a list of argument types and the result type (the argument list and return type can be suppressed in a class diagram)
  - Specification is expressed in terms of pre-and postconditions (the Object Constraint Language is used for this purpose)
  - The complete signature of an operation is:

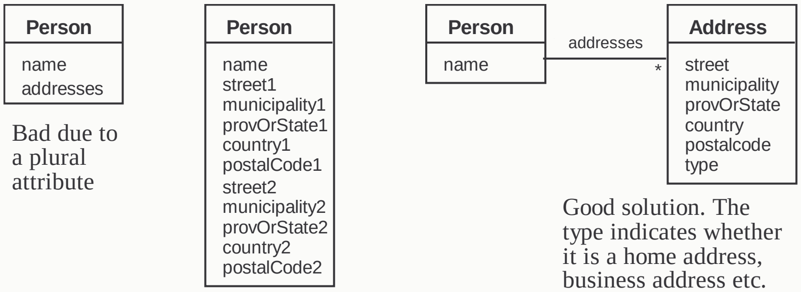
    operationName(parameterName: parameterType …): returnType
- A method is the implementation of an operation in a class.
- In this course, classes in requirements class models DO NOT contain operations
  - Rationale: in most cases, distributing operations across classes requires making design-level decision. For this reason I discourage students from putting operations in requirements-level class models (i.e., domain models).

Identifying requirements classes

- Look at a source material such as a description of requirements
- Extract the *nouns* and *noun phrases*
- Eliminate nouns that:
  - are redundant
  - represent instances
  - are vague or highly general
  - not needed in the application
- Pay attention to classes in a domain model that represent *types of users* or other actors

Identifying and specifying valid attributes

- It is not good to have many duplicate attributes
- If a subset of a class's attributes form a coherent group, then create a distinct class containing these attributes



Bad due to a plural attribute

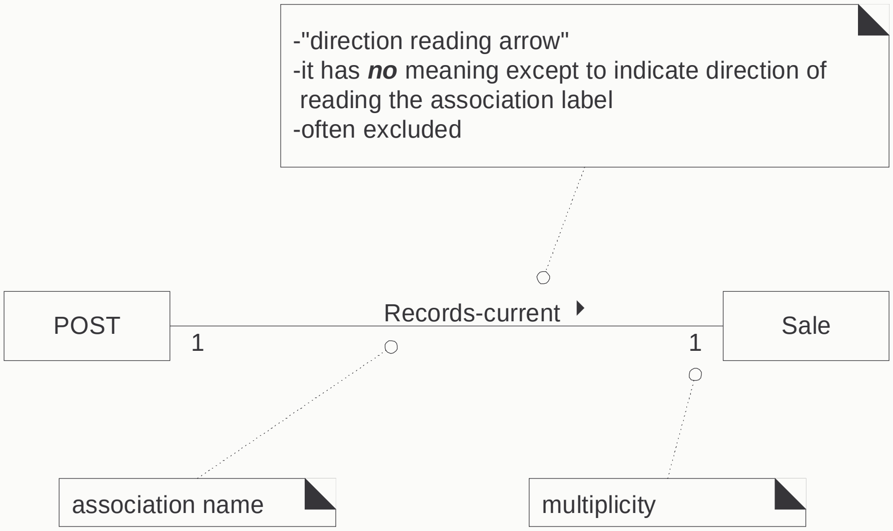
Bad due to too many attributes, and inability to add more addresses

Good solution. The type indicates whether it is a home address, business address etc.

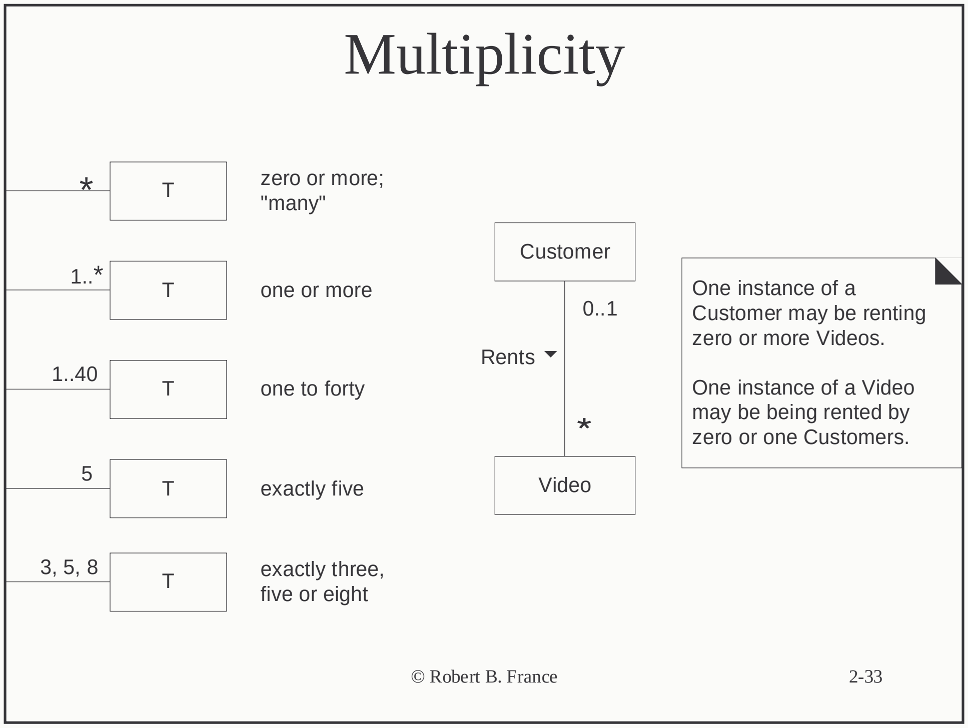
Modeling Static Relationships

- Two kinds of static relationships:
  - Associations
    - Represent structural relationships among objects
    - An association specifies a collection of links, where a link is a physical or conceptual connection among objects.
  - Generalizations
    - Represent generalization/specialization class structures
- The two kinds of relationships are orthogonal

Associations



-"direction reading arrow"
-it has ***no*** meaning except to indicate direction of
 reading the association label
-often excluded

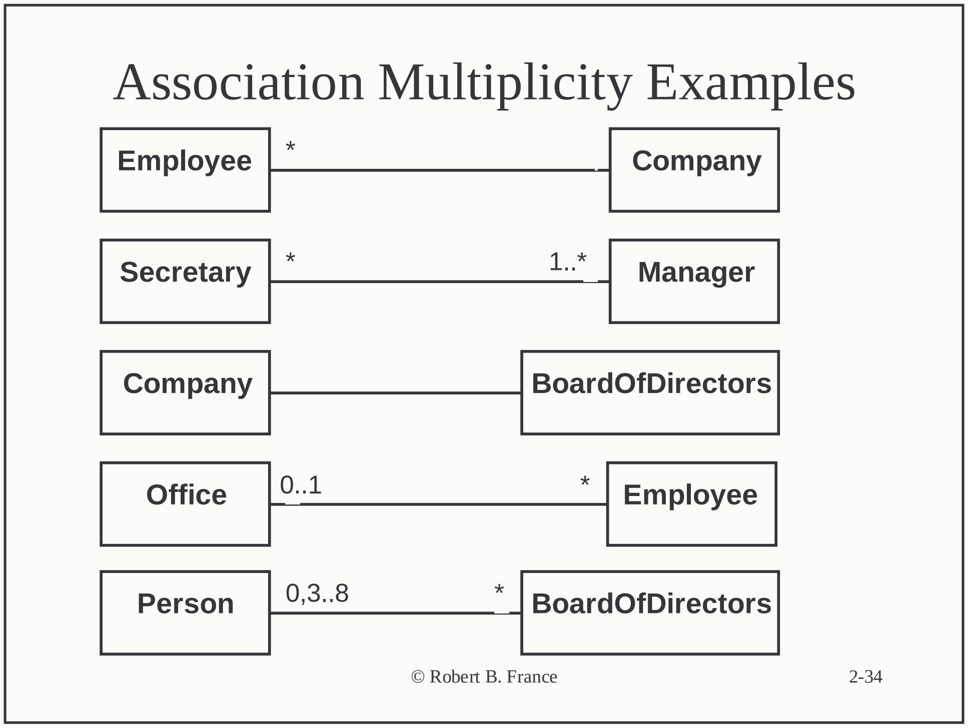


Multiplicity

*	T	zero or more; "many"
1..*	T	one or more
1..40	T	one to forty
5	T	exactly five
3, 5, 8	T	exactly three, five or eight

One instance of a Customer may be renting zero or more Videos.

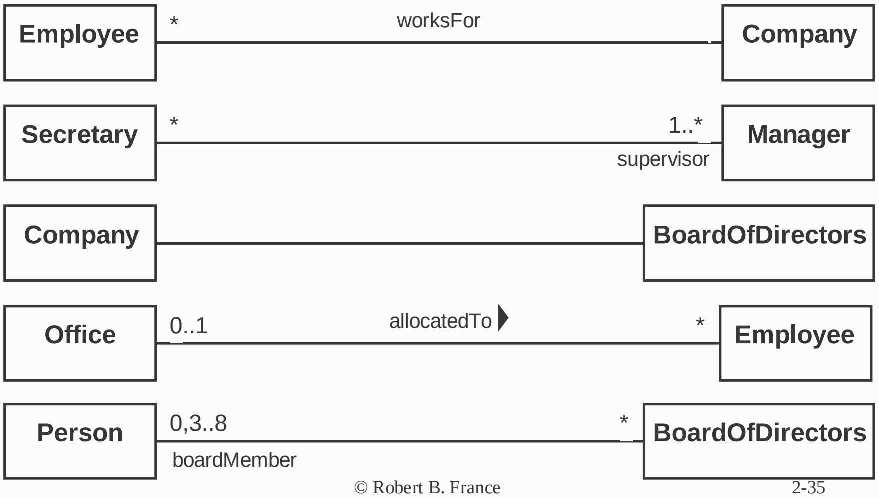
One instance of a Video may be being rented by zero or one Customers.



Association Multiplicity Examples

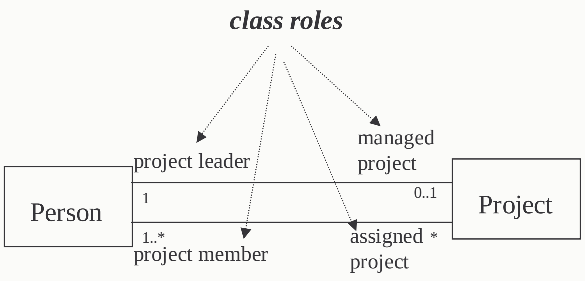
Labelling associations

– Each association can be labelled with a name that gives insight into the meaning of the association



Association Roles

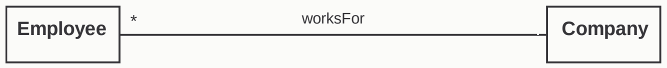
- When a class is part of an association it plays a *role* in the relationship.
- You can name the role that a class plays in an association by placing the name at the class’s association end.
- Formally, a class role is the set of objects that are linked via the association.



Analyzing and validating associations

– **Many-to-one**

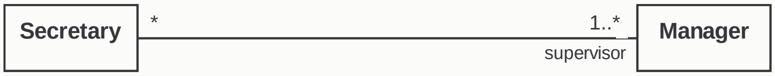
- A company has many employees,
- An employee can only work for one company.
- A company can have zero employees
  – E.g. a ‘shell’ company
- It is not possible to be an employee unless you work for a company



Analyzing and validating associations

– **Many-to-many**

- A secretary can work for many managers
- A manager can have many secretaries
- Secretaries can work in pools
- Managers can have a group of secretaries
- Some managers might have zero secretaries.
- Is it possible for a secretary to have, perhaps temporarily, zero managers?

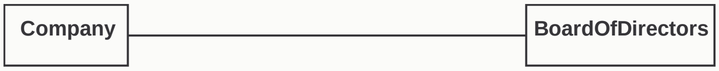


© Robert B. France

Analyzing and validating associations

– **One-to-one**

- For each company, there is exactly one board of directors
- A board is the board of only one company
- A company must always have a board
- A board must always be of some company

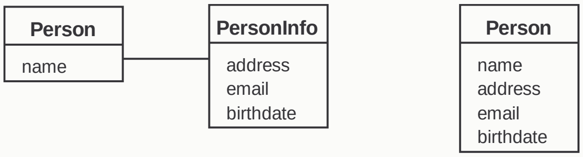


© Robert B. France

Analyzing and validating associations

- Avoid unnecessary one-to-one associations

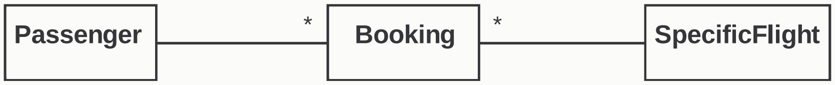
- Avoid this do this



A more complex example

– A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could *never* involve more than one passenger.

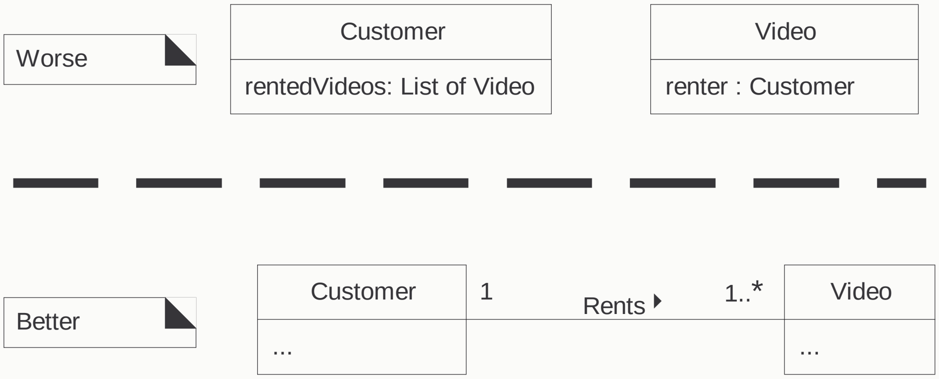
– A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking



Describing Associations

- In my course, an association must have a name or at least one role name.
  - An association without some name is meaningless! Do not rely on the reader to “fill in the blanks”
  - In the cases where there are multiple associations between two classes having an association name or role name is needed to disambiguate the model.
- Associations are often implemented as references
  - You may be tempted to model references as attributes; PLEASE avoid doing this; use association instead (also don’t use both attributes and associations to model references!)

Do Not Use Attributes To Relate Concepts


Worse

Customer
rentedVideos: List of Video

Video
renter : Customer

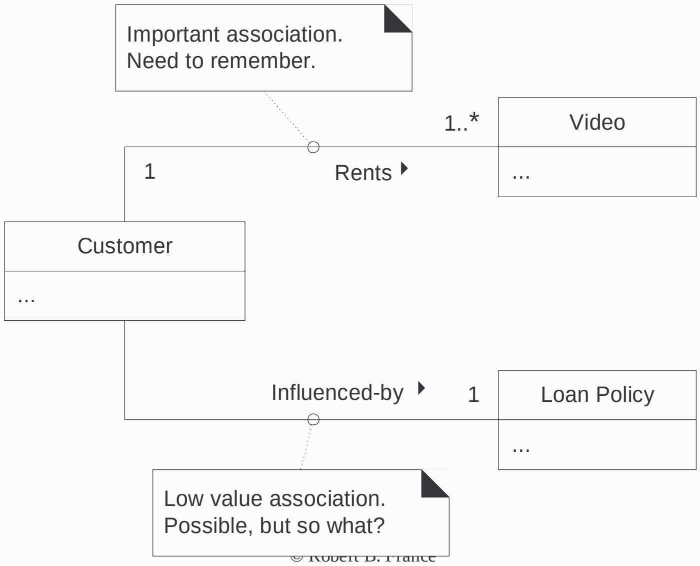
Better

Customer
...

1 Rents ▸ 1..*

Video
...

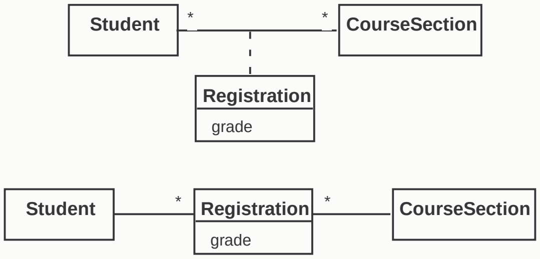
Focus on Important Associations



Association classes

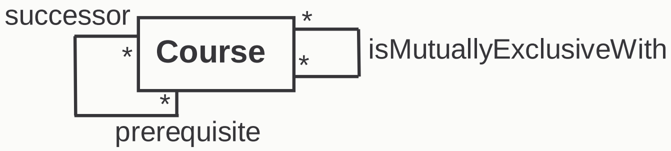
– Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes

– The following are equivalent



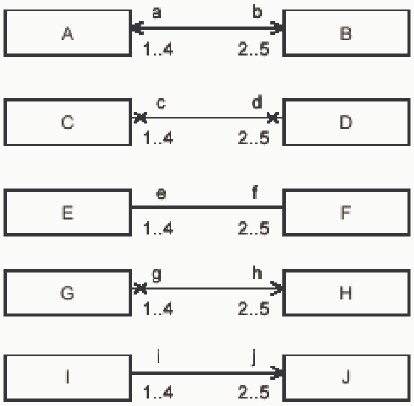
Reflexive associations

– It is possible for an association to connect a class to itself



Navigability

- One can indicate that an object "knows about" another object it is linked to by using navigation arrows on associations
  - In UML 2.0 one can also explicitly show that one object does not know about the objects it is linked to.
- Show navigability ONLY in solution models (i.e., design and implementation models)
  - Do not show navigability in requirements class models

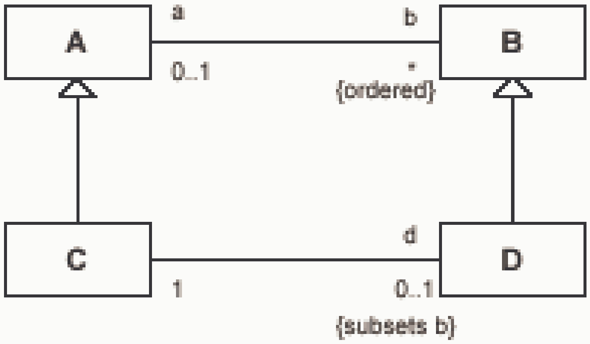


The constructs in diagrams 1, 2, and 4 are new to UML 2.0 and thus are most likely not supported by UML tools as yet.

•The top pair AB shows a binary association with two navigable ends.
•The second pair CD shows a binary association with two non-navigable ends.
•The third pair EF shows a binary association with unspecified navigability.
•The fourth pair GH shows a binary association with one end navigable and the other non-navigable.
•The fifth pair IJ shows a binary association with one end navigable and the other having unspecified navigability.

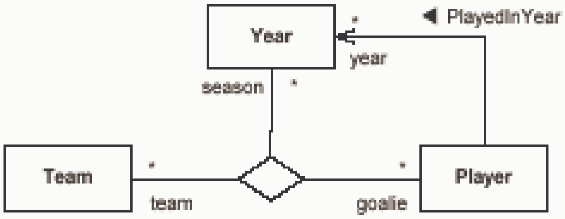
Association End Property Strings: Predefined Constraints

- {subsets <property-name>} to show that the end is a subset of the property called <property-name>.
- {union} to show that the end is derived by being the union of its subsets.
- {ordered} to show that the end represents an ordered set.
- {bag} to show that the end represents a collection that permits the same element to appear more than once.
- {sequence} or {seq} to show that the end represents a sequence (an ordered bag).



Binary and N-ary Associations

- A *binary association* relates two classes.
- An *n-ary association* relates n (n > 2) classes.
- N-ary associations can often be modeled as binary associations.



Can you interpret the following?

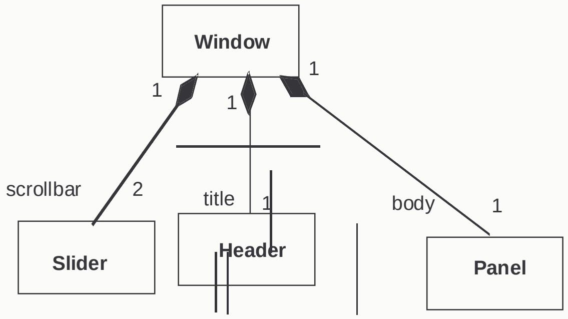
Aggregation

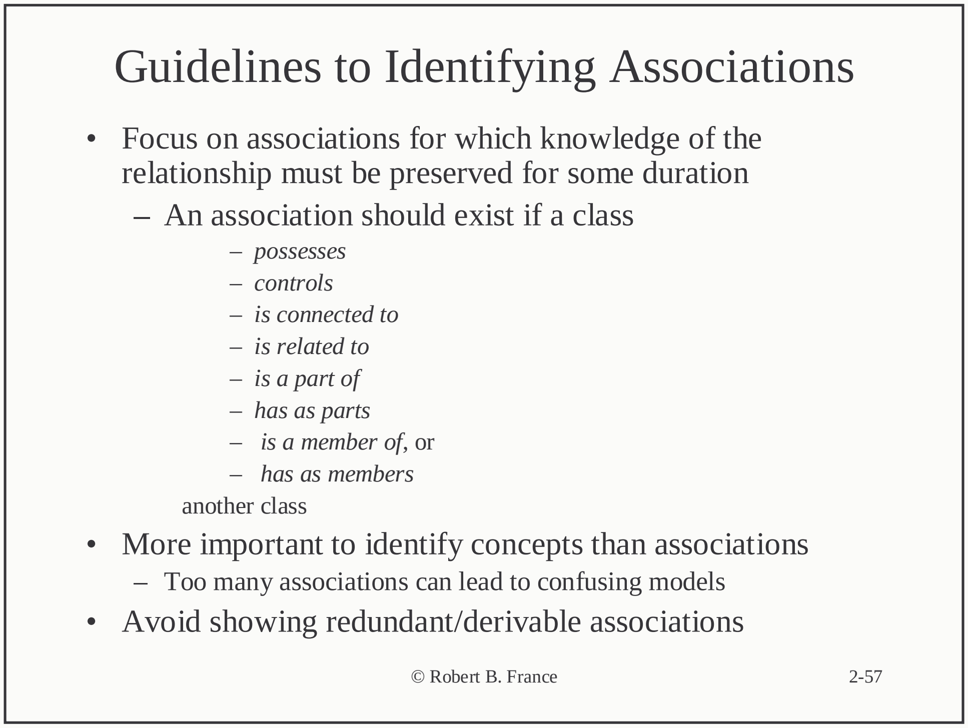
- Aggregation is a special form of association
  - reflect whole-part relationships
- In a solution model, the whole delegates responsibilities to its parts
  - the parts are subordinate to the whole
  - This is unlike associations in which classes have equal status

UML Forms of Aggregation

- Composition (strong aggregation)
  - – parts are existent-dependent on the whole
  - – parts are generated at the same time, before, or after the whole is created (depending on cardinality at whole end) and parts are deleted before or at the same time the whole dies
  - – multiplicity at whole end must be 1 or 0..1
- (weak) Aggregation

Composition

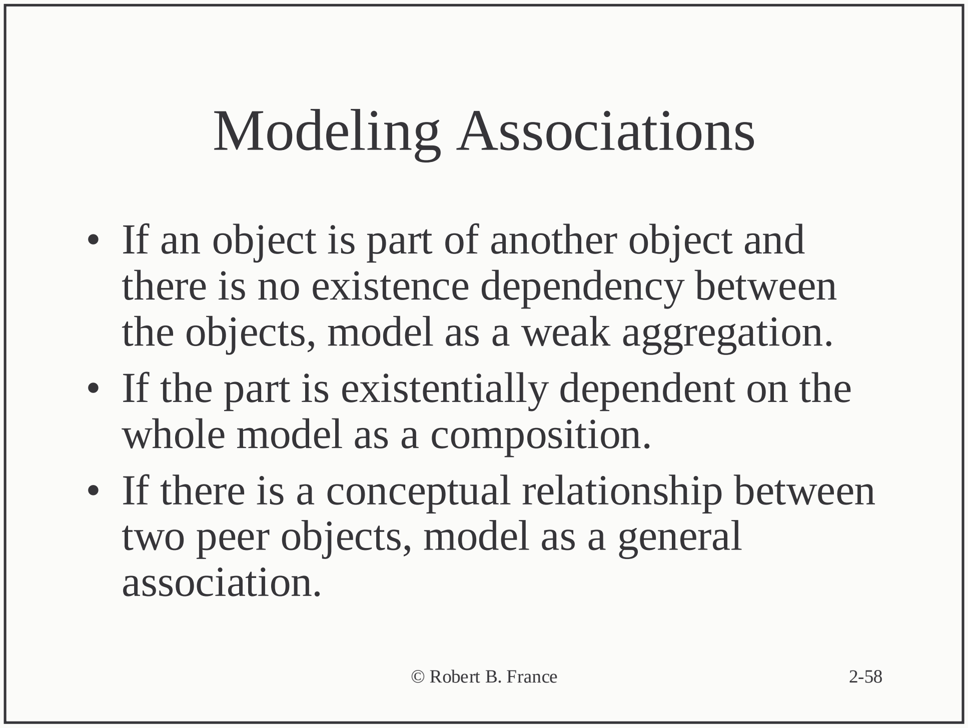




Guidelines to Identifying Associations

- Focus on associations for which knowledge of the relationship must be preserved for some duration
  - An association should exist if a class
    - *possesses*
    - *controls*
    - *is connected to*
    - *is related to*
    - *is a part of*
    - *has as parts*
    - *is a member of*, or
    - *has as members*

    another class
- More important to identify concepts than associations
  - Too many associations can lead to confusing models
- Avoid showing redundant/derivable associations

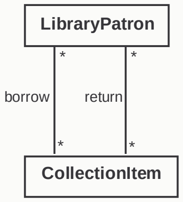


Modeling Associations

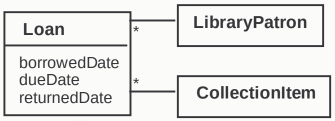
- If an object is part of another object and there is no existence dependency between the objects, model as a weak aggregation.
- If the part is existentially dependent on the whole model as a composition.
- If there is a conceptual relationship between two peer objects, model as a general association.

Actions versus associations

– A common mistake is to represent *actions* as if they were associations



Bad, due to the use of associations that are actions



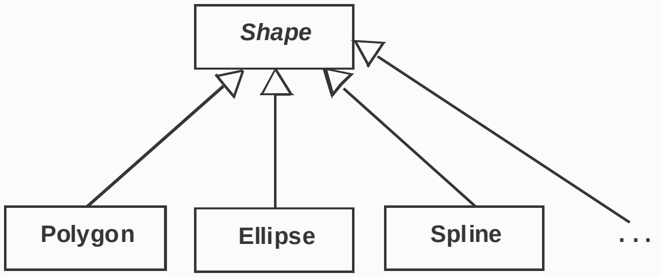
Better: The `borrow` operation creates a `Loan`, and the `return` operation sets the `returnedDate` attribute.

Generalization/Specialization

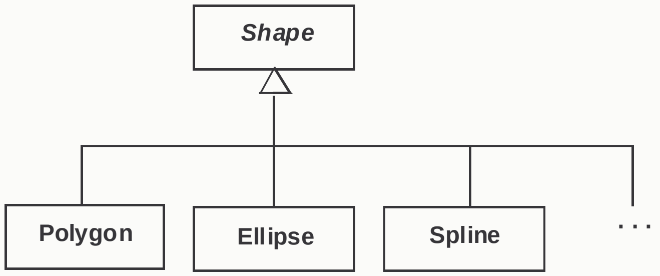
- A generalization (or specialization) is a relationship between a general concept and its specializations.
  – Objects of specializations can be used anywhere an object of a generalization is expected (but not vice versa).
- Example: *Mechanical Engineer* and *Aeronautical Engineer* are specializations of *Engineer*

Rendering Generalizations

- Generalization is rendered as a solid directed line with a large open arrowhead.
  - – Arrowhead points towards generalization
- A discriminator can be used to identify the nature of specializations



Separate Target Style



Shared Target Style

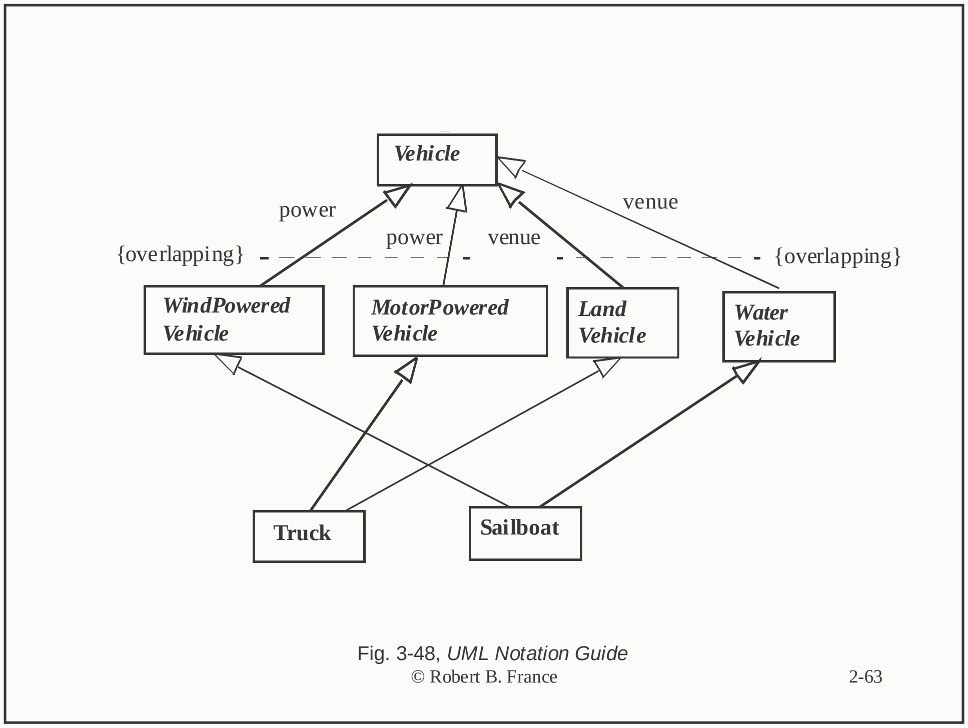
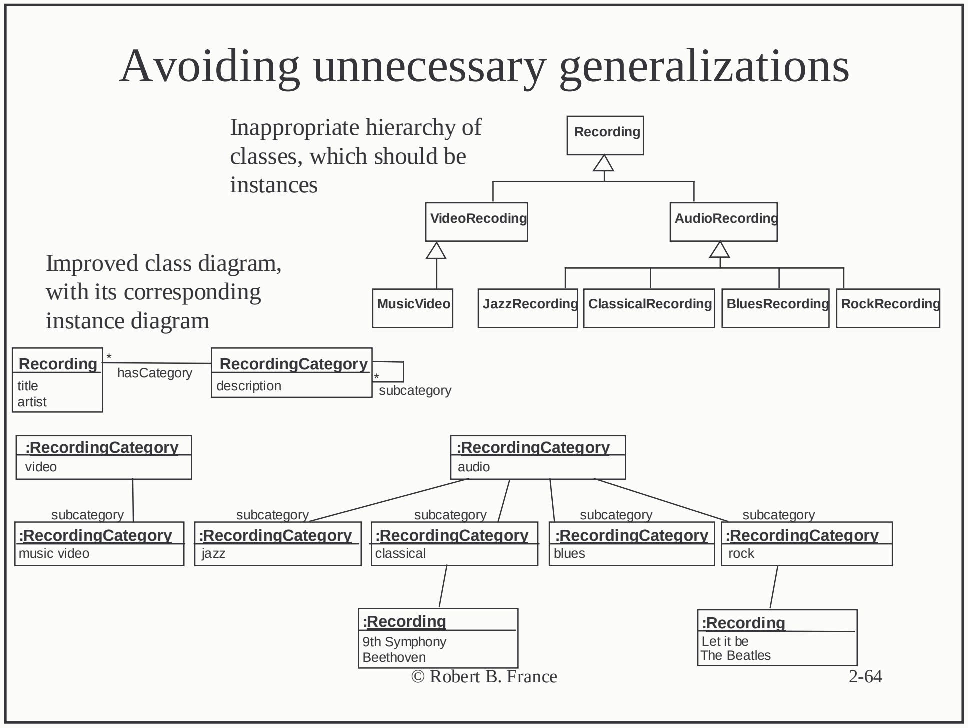


Fig. 3-48, *UML Notation Guide*

© Robert B. France

2-63



Avoiding unnecessary generalizations

Inappropriate hierarchy of classes, which should be instances

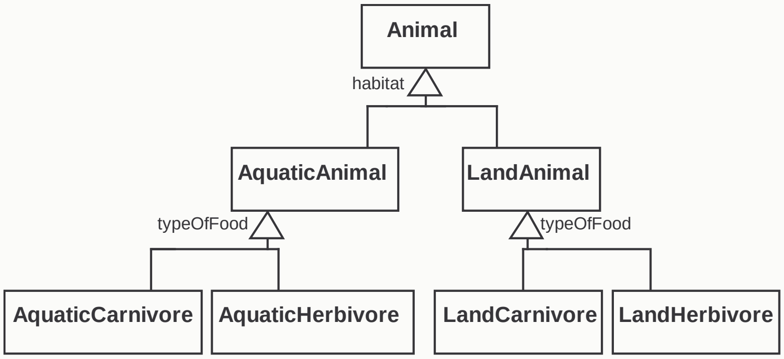
Improved class diagram, with its corresponding instance diagram

© Robert B. France

2-64

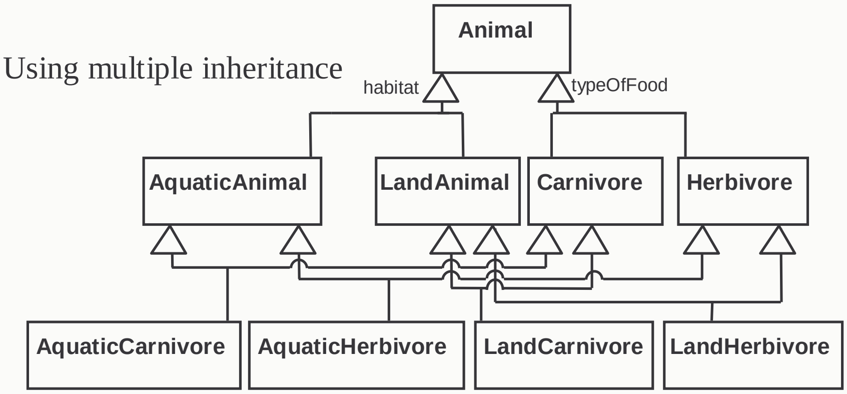
Handling multiple discriminators

– Creating higher-level generalization



Handling multiple discriminators

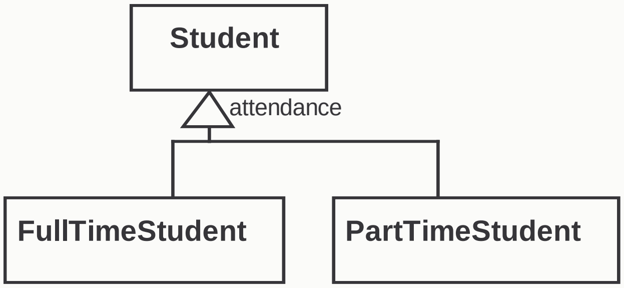
– Using multiple inheritance



– Using the Player-Role pattern (in Chapter 6)

Avoiding having instances change class

– An instance should never need to change class

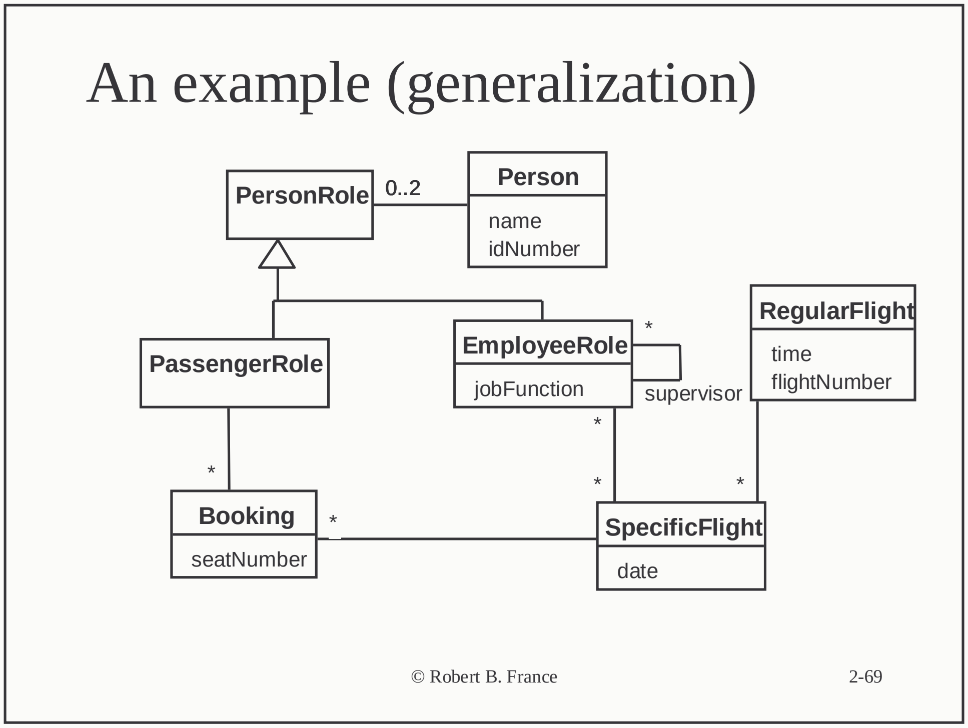


© Robert B. France

Identifying generalizations and interfaces

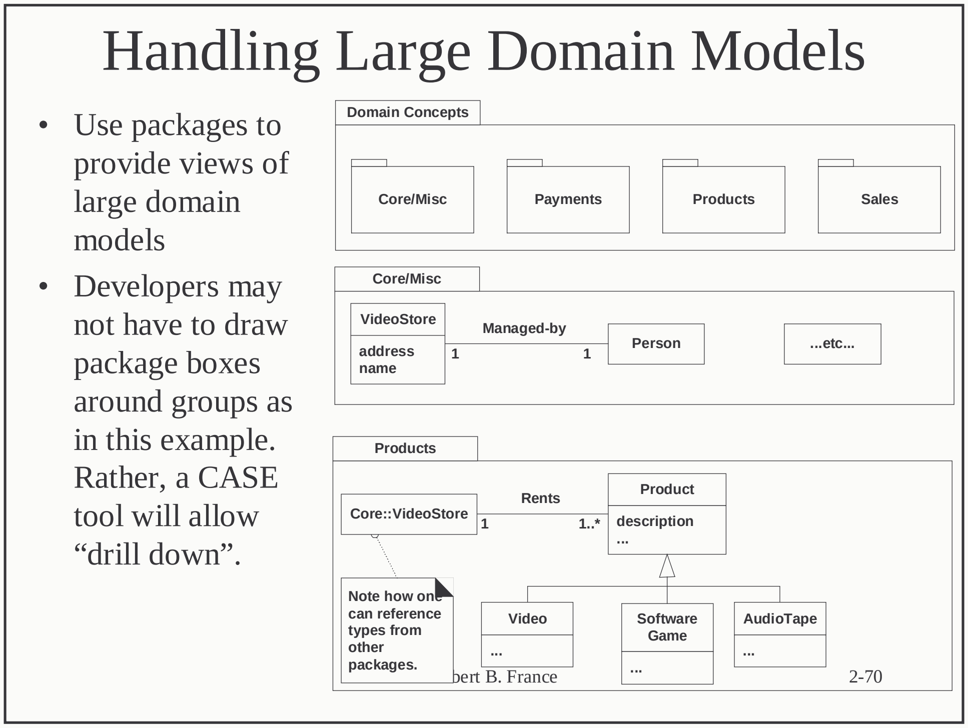
- There are two ways to identify generalizations:
  - bottom-up
    - Group together similar classes creating a new superclass
  - top-down
    - Look for more general classes first, specialize them if needed
- Create an *interface*, instead of a superclass if
  - The classes are very dissimilar except for having a few operations in common
  - One or more of the classes already have their own superclasses
  - Different implementations of the same class might be available

© Robert B. France



An example (generalization)

© Robert B. France 2-69



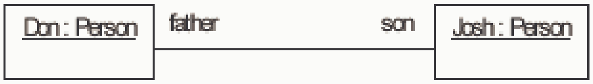
Handling Large Domain Models

- Use packages to provide views of large domain models
- Developers may not have to draw package boxes around groups as in this example. Rather, a CASE tool will allow “drill down”.

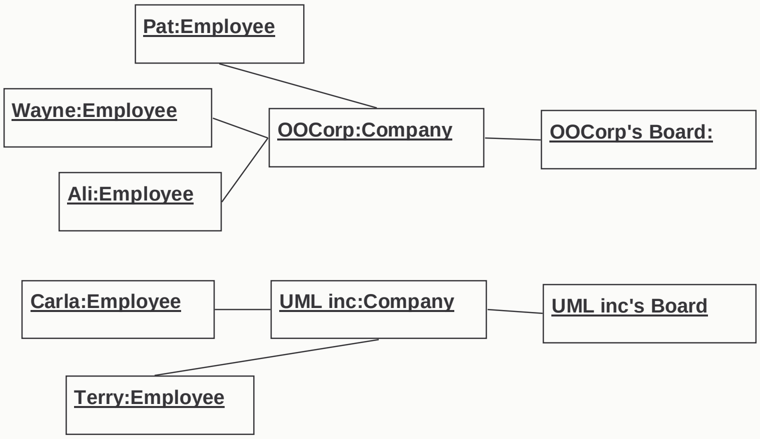
bert B. France 2-70

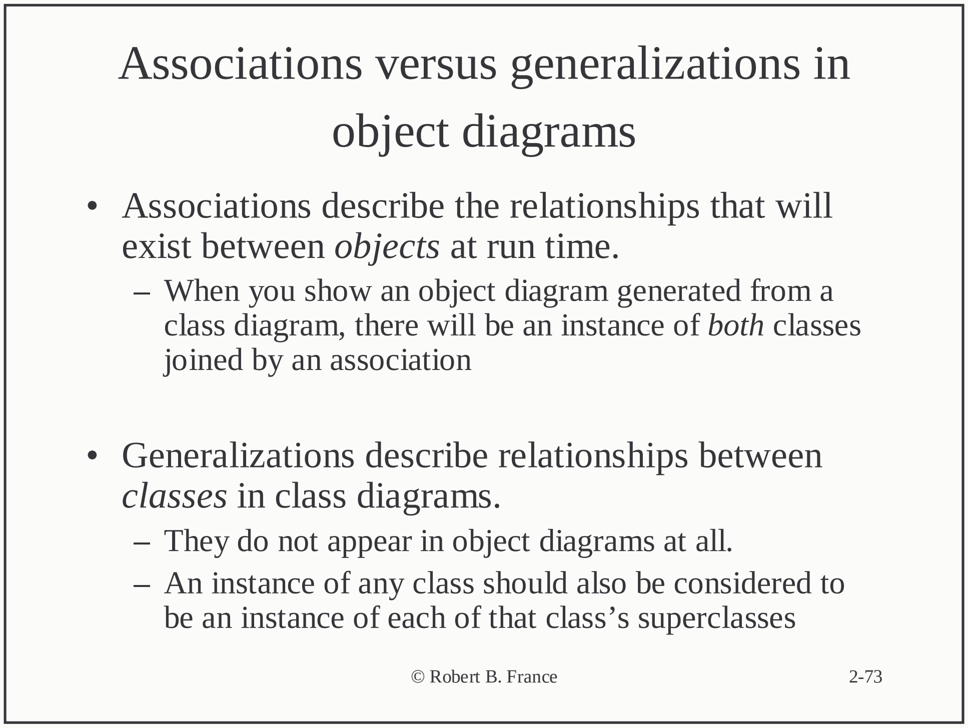
Object Diagrams

- An object diagram describes a structure of objects.
- One can view a class diagram as specifying a collection of object structures
  – Conformant object structures have objects and links that satisfy the multiplicity and other constraints specified in the class diagram.



Object Diagram Example



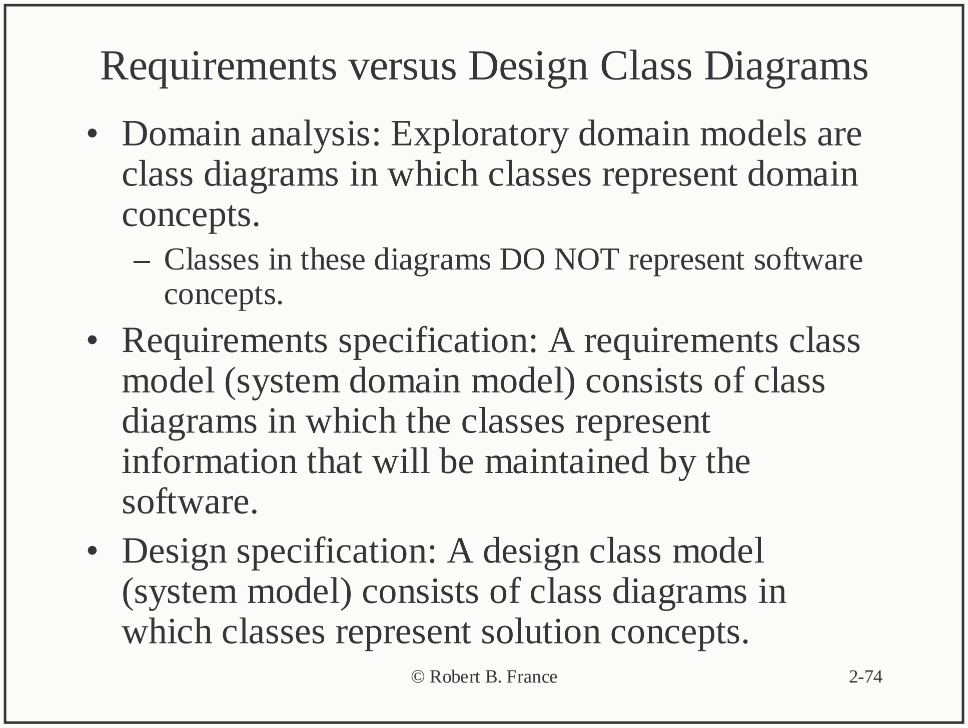


Associations versus generalizations in object diagrams

- Associations describe the relationships that will exist between *objects* at run time.
  - When you show an object diagram generated from a class diagram, there will be an instance of *both* classes joined by an association

- Generalizations describe relationships between *classes* in class diagrams.
  - They do not appear in object diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class's superclasses

© Robert B. France

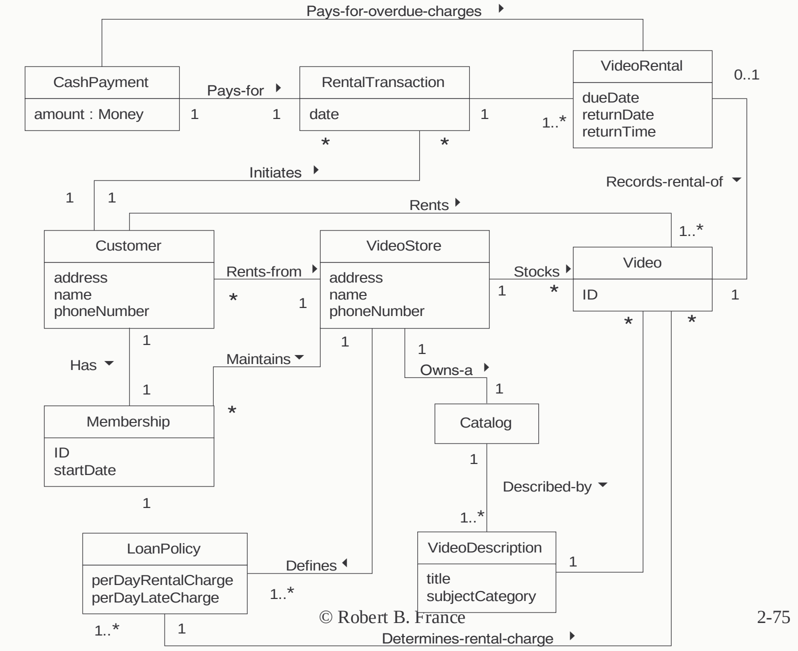


Requirements versus Design Class Diagrams

- Domain analysis: Exploratory domain models are class diagrams in which classes represent domain concepts.
  - Classes in these diagrams DO NOT represent software concepts.
- Requirements specification: A requirements class model (system domain model) consists of class diagrams in which the classes represent information that will be maintained by the software.
- Design specification: A design class model (system model) consists of class diagrams in which classes represent solution concepts.

© Robert B. France

An Example of a Requirements Class Diagram



Requirements Class Diagram Exercise

Develop a class diagram of an invoicing system that keeps track of orders and invoices orders against a stock in an inventory. Each order has exactly one order item. An order item consists of a part identifier and a quantity.

Another Exercise

Develop Requirements Class Diagram for the following application:

A school video library tracking system is to be developed. Videos can be scientific or non-scientific. Students and professors can belong to research groups. A research group must have at least 1 professor. Students that belong to a research group are called research students. A research group can consist of members with various subject area interests. Professors can check out any number of videos. Students can check out at most 2 non-scientific videos. Research students can check out only scientific videos in the subject areas represented in their research groups.

Design Class Modeling

- Requirements class models should not include operations in classes.
- To obtain a design class model consider how responsibilities are distributed across classes.
  - Determines attributes and operations of design classes.

Allocating responsibilities to classes

•A *responsibility* is something that the system is required to do.

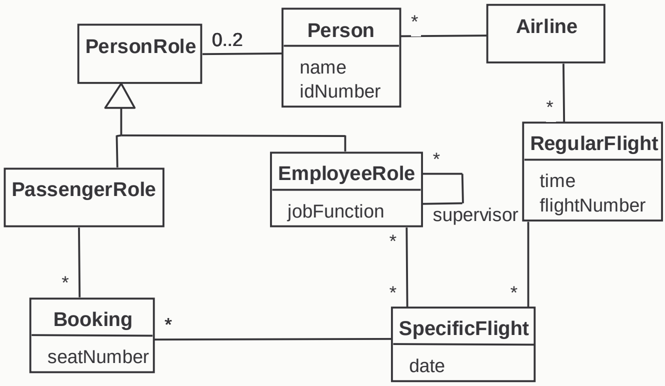
- Each functional requirement must be attributed to one of the classes
  - All the responsibilities of a given class should be *clearly related*.
  - If a class has too many responsibilities, consider *splitting* it into distinct classes
  - If a class has no responsibilities attached to it, then it is probably *useless*
  - When a responsibility cannot be attributed to any of the existing classes, then a *new class* should be created
- To determine responsibilities
  - Perform use case analysis
  - Look for verbs and nouns describing *actions* in the system description

Categories of responsibilities

- Setting and getting the values of attributes
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Computing numerical results
- Navigating and searching
- Other specialized work

An example (responsibilities)

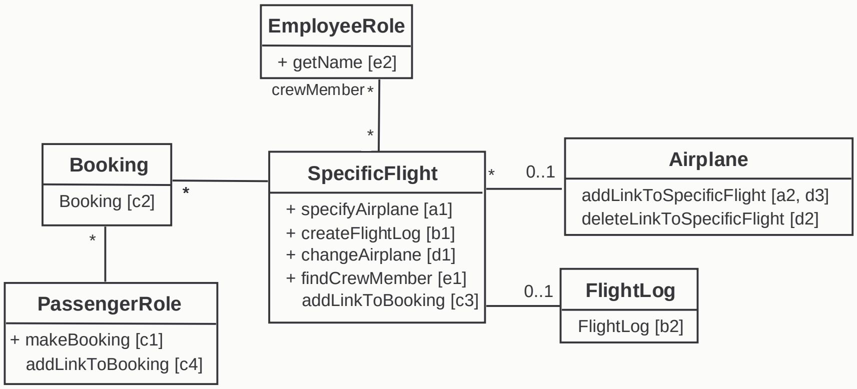
- Creating a new regular flight
- Searching for a flight
- Modifying attributes of a flight
- Creating a specific flight
- Booking a passenger
- Canceling a booking

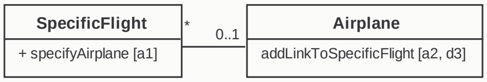


Identifying operations

- Operations are needed to realize the responsibilities of each class
  - There may be several operations per responsibility
  - The main operations that implement a responsibility are normally declared `public`
  - Other methods that collaborate to perform the responsibility must be as private as possible

An example of a design class diagram





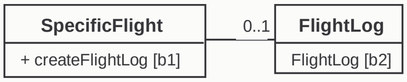
- *Making a bi-directional link between two existing objects*;
- **e.g. adding a link between an instance of `SpecificFlight` and an instance of `Airplane`.**

a1. (public) The instance of `SpecificFlight`

- **makes a one-directional link to the instance of `Airplane`**
- **then calls operation a2.**

a2. (non-public) The instance of `Airplane`

- **makes a one-directional link back to the instance of `SpecificFlight`**



- *Creating an object and linking it to an existing object*
- **e.g. creating a `FlightLog`, and linking it to a `SpecificFlight`.**

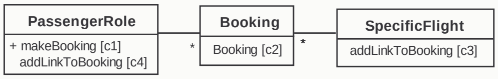
b1. (public) The instance of `SpecificFlight`

calls the constructor of `FlightLog` (operation b2)

then makes a one-directional link to the new instance of `FlightLog`.

b2. (non-public) Class `FlightLog`'s constructor

makes a one-directional link back to the instance of `SpecificFlight`.



- *Creating an association class, given two existing objects*
- **e.g. creating an instance of `Booking`, which will link a `SpecificFlight` to a `PassengerRole`.**

c1. (public) The instance of `PassengerRole`

- calls the constructor of **`Booking`** (operation 2).

c2. (non-public) Class `Booking`'s constructor, among its other actions

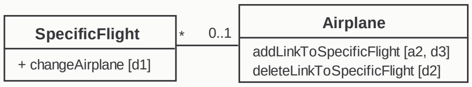
- makes a one-directional link back to the instance of **`PassengerRole`**
- makes a one-directional link to the instance of **`SpecificFlight`**
- calls operations 3 and 4.

c3. (non-public) The instance of `SpecificFlight`

- makes a one-directional link to the instance of **`Booking`**.

c4. (non-public) The instance of `PassengerRole`

- makes a one-directional link to the instance of **`Booking`**.


SpecificFlight
+ changeAirplane [d1]
*
0..1
Airplane
addLinkToSpecificFlight [a2, d3]
deleteLinkToSpecificFlight [d2]

- *Changing the destination of a link*
- **e.g. changing the** `Airplane` **of to a** `SpecificFlight`**, from** `airplane1` **to** `airplane2`

d 1. (public) The instance of `SpecificFlight`

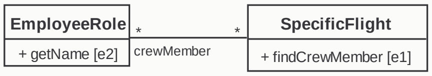
- deletes the link to **`airplane1`**
- makes a one-directional link to **`airplane2`**
- calls operation d2
- then calls operation d3.

d2. (non-public) `airplane1`

- deletes its one-directional link to the instance of **`SpecificFlight`**.

d3. (non-public) `airplane2`

- makes a one-directional link to the instance of **`SpecificFlight`**.

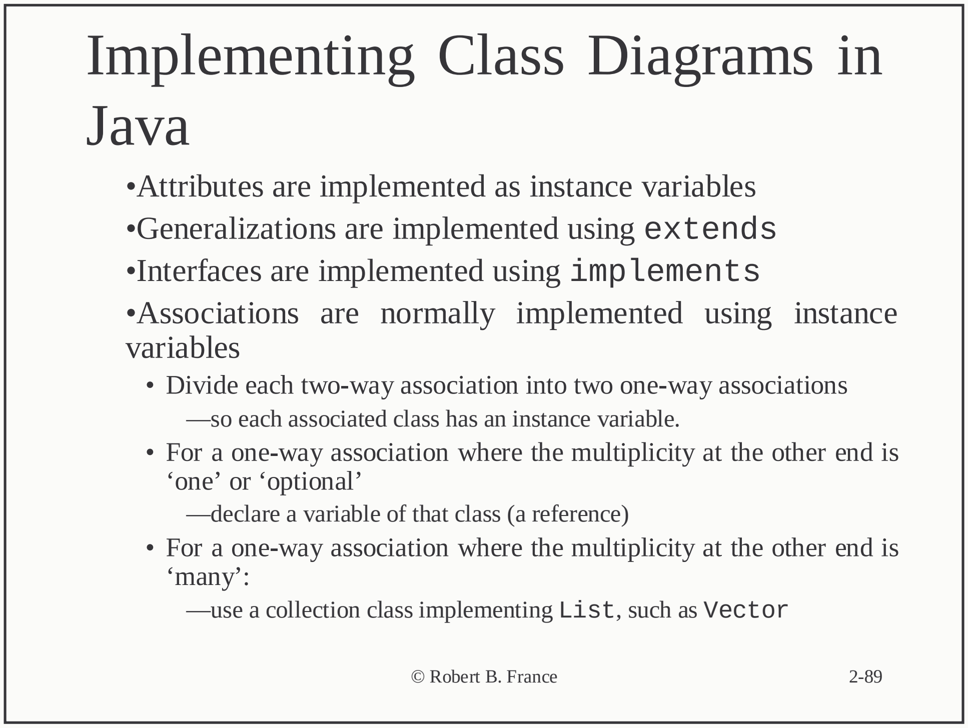


- *Searching for an associated instance*
- **e.g. searching for a crew member associated with a** `SpecificFlight` **that has a certain name.**
-

e1. (public) The instance of `SpecificFlight`

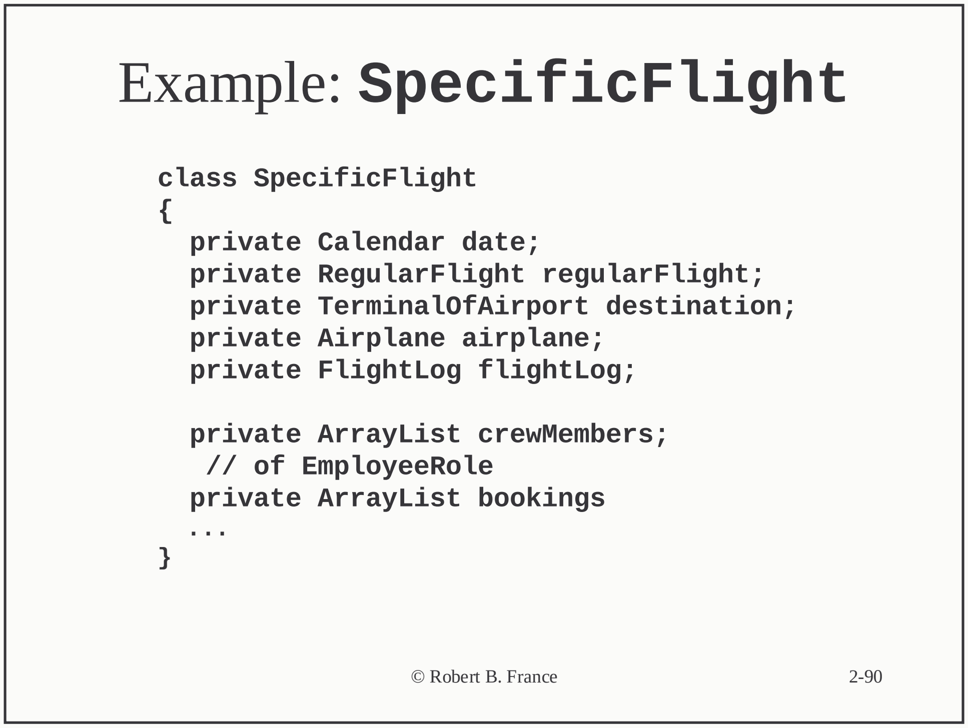
- creates an `Iterator` over all the `crewMember` links of the `SpecificFlight\`
- for each of them call operation e2, until it finds a match.

e2. (may be public) The instance of `EmployeeRole` **returns its name.**



Implementing Class Diagrams in Java

- Attributes are implemented as instance variables
- Generalizations are implemented using `extends`
- Interfaces are implemented using `implements`
- Associations are normally implemented using instance variables
  - Divide each two-way association into two one-way associations
    - —so each associated class has an instance variable.
  - For a one-way association where the multiplicity at the other end is ‘one’ or ‘optional’
    - —declare a variable of that class (a reference)
  - For a one-way association where the multiplicity at the other end is ‘many’:
    - —use a collection class implementing `List`, such as `Vector`



Example: **`SpecificFlight`**

```
class SpecificFlight
{
  private Calendar date;
  private RegularFlight regularFlight;
  private TerminalOfAirport destination;
  private Airplane airplane;
  private FlightLog flightLog;

  private ArrayList crewMembers;
   // of EmployeeRole
  private ArrayList bookings
  ...
}
```

Example: **SpecificFlight**

- // Constructor that should only be called from
- // addSpecificFlight
- SpecificFlight(
-   Calendar aDate,
-   RegularFlight aRegularFlight)
- {
-   date = aDate;
-   regularFlight = aRegularFlight;
- }

Example: **RegularFlight**

```
class RegularFlight
{
  private ArrayList specificFlights;
  ...
  // Method that has primary
  // responsibility

  public void addSpecificFlight(
    Calendar aDate)
  {
    SpecificFlight newSpecificFlight;
    newSpecificFlight =
      new SpecificFlight(aDate, this);
    specificFlights.add(newSpecificFlight);
  }
  ...
}
```