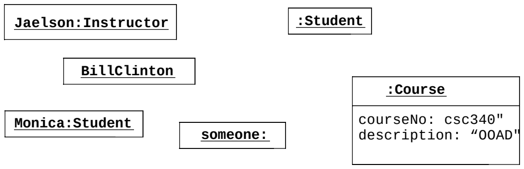
X. Sequence and Collaboration Diagrams

Object Diagrams
Class Responsibilty Cards (CRCs)
Sequence Diagrams
An Example
Collaboration Diagrams

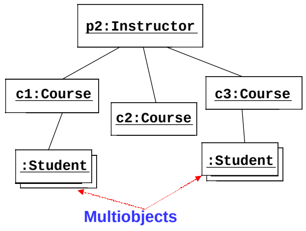
Object Diagrams

- Model the instances of things described by a class.
- Each object diagram shows a set of objects and their inter-relationships at a point in time.
- Used to model a snapshot of the application.
- Each object has an optional name and set of classes it is an instance of, also values for attributes of these classes.



Multiobjects

A ***multiobject*** is a set of objects, with an undefined number of elements



Communication and Collaboration Between Objects

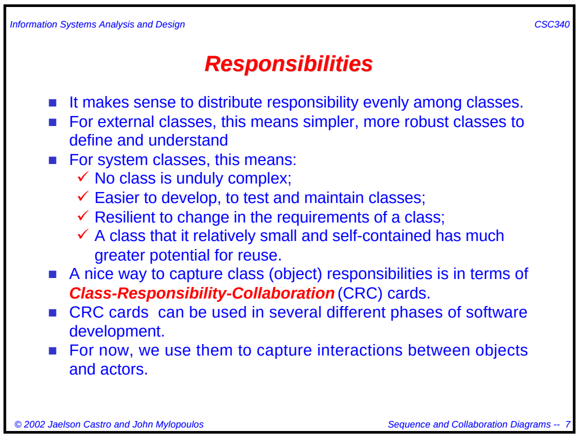
- Communication and collaboration among objects is a fundamental concept for object-orientation.
- We want to decide which objects are responsible for what (within or without the system).
- In addition, we want to know how external users and external systems (“actors”) interact with each other and the system.
- As well, it is often convenient to model interactions between actors; for example, the interactions between actors carrying out a business process.

Object Interaction and Collaboration

- Objects “own” information and behaviour, defined by operations; system objects contain data and methods which are relevant to their own ***responsibilities***. They don’t “know” about other objects’ information, but can ask for it.
- To carry out business processes, objects (system or otherwise) have to work together, I.e., collaborate.
- Objects collaborate by sending messages to one another thereby calling operations of the other object.
- Objects can only send messages to one another if they “know” each other, I.e., there is an association between them.
- A ***responsibility*** is high level description of something instances of a class can do. A responsibility reflects the knowledge or information that is available to that class, either stored within its own attribute or requested via collaboration with other classes.

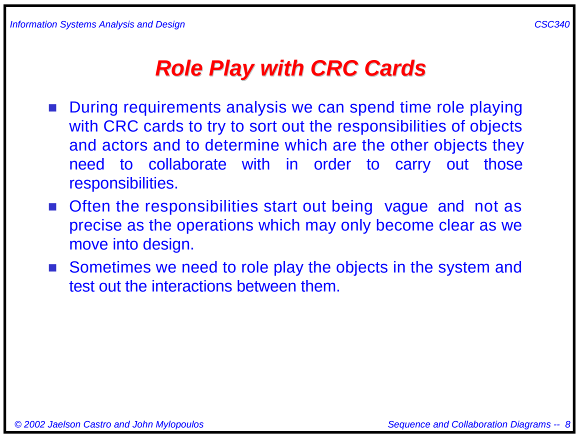
VIN -- Very Important Note

- During requirements, the system is modelled in terms of a small number of coarse-grain classes and objects which describe how the system interacts with its environment.
- During design, the system is modelled in greater detail in terms of many fine-grain classes and objects.
- To keep things clear, we will use icons to represent external objects and actors, and boxes to represent system objects.



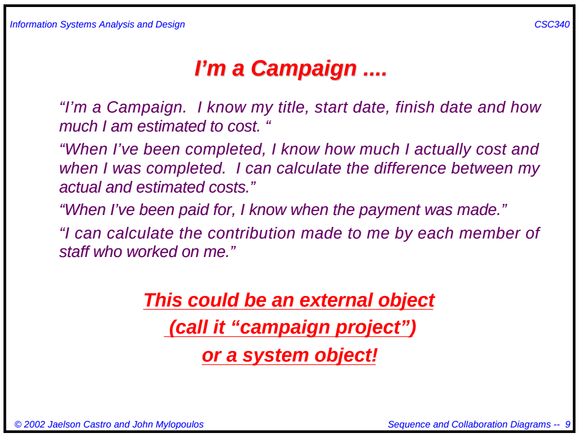
Responsibilities

- It makes sense to distribute responsibility evenly among classes.
- For external classes, this means simpler, more robust classes to define and understand
- For system classes, this means:
  - ✓ No class is unduly complex;
  - ✓ Easier to develop, to test and maintain classes;
  - ✓ Resilient to change in the requirements of a class;
  - ✓ A class that it relatively small and self-contained has much greater potential for reuse.
- A nice way to capture class (object) responsibilities is in terms of ***Class-Responsibility-Collaboration*** (CRC) cards.
- CRC cards can be used in several different phases of software development.
- For now, we use them to capture interactions between objects and actors.



Role Play with CRC Cards

- During requirements analysis we can spend time role playing with CRC cards to try to sort out the responsibilities of objects and actors and to determine which are the other objects they need to collaborate with in order to carry out those responsibilities.
- Often the responsibilities start out being vague and not as precise as the operations which may only become clear as we move into design.
- Sometimes we need to role play the objects in the system and test out the interactions between them.



I'm a Campaign ....

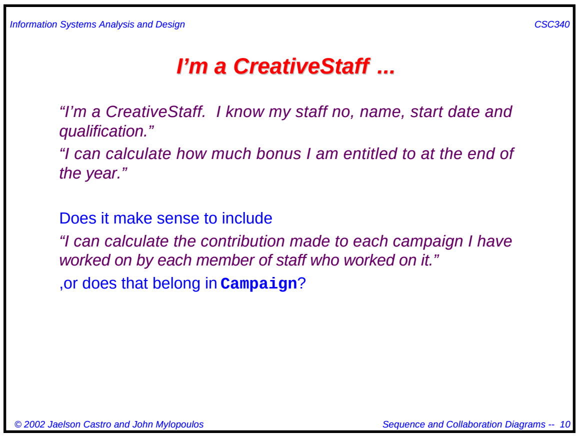
"I'm a Campaign. I know my title, start date, finish date and how much I am estimated to cost. "

"When I've been completed, I know how much I actually cost and when I was completed. I can calculate the difference between my actual and estimated costs."

"When I've been paid for, I know when the payment was made."

"I can calculate the contribution made to me by each member of staff who worked on me."

This could be an external object
(call it "campaign project")
or a system object!



I'm a CreativeStaff ...

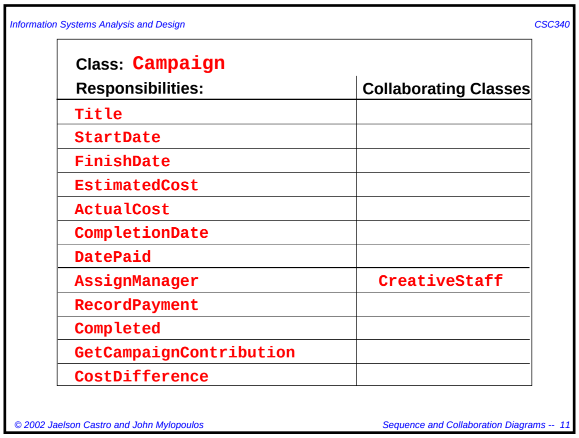
"I'm a CreativeStaff. I know my staff no, name, start date and qualification."

"I can calculate how much bonus I am entitled to at the end of the year."

Does it make sense to include

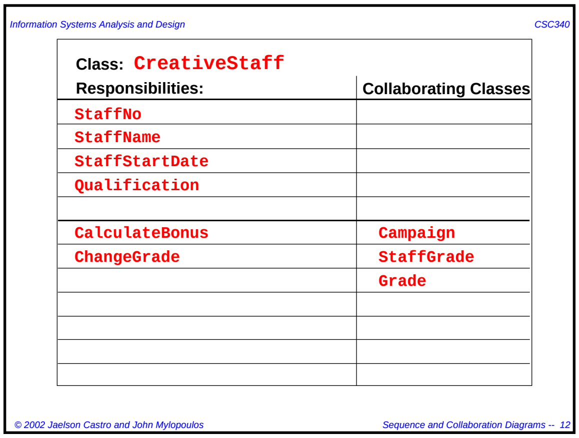
"I can calculate the contribution made to each campaign I have worked on by each member of staff who worked on it."

,or does that belong in **`Campaign`**?



Class: `Campaign`

Responsibilities:	Collaborating Classes
Title	
StartDate	
FinishDate	
EstimatedCost	
ActualCost	
CompletionDate	
DatePaid	
AssignManager	CreativeStaff
RecordPayment	
Completed	
GetCampaignContribution	
CostDifference	



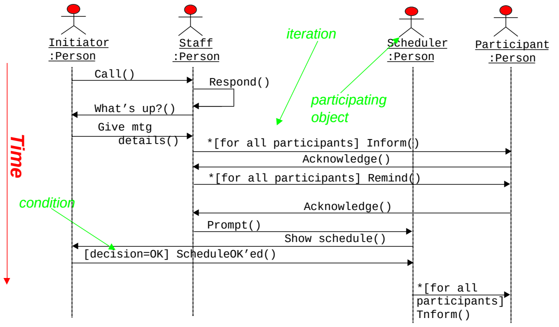
Class: `CreativeStaff`

Responsibilities:	Collaborating Classes
StaffNo	
StaffName	
StaffStartDate	
Qualification	
CalculateBonus	Campaign
ChangeGrade	StaffGrade
	Grade

Sequence Diagrams

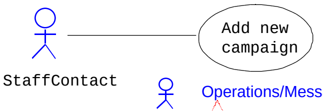
- Sequence diagrams describe in detail how actors use use cases; they can also model external business processes the new system will support (e.g., processing a book order)
- An ***interaction*** is a behavior that consists of a set of messages exchanged between external and system objects.
- Interactions consist of one or more ***messages***. Interactions may be synchronous (e.g., calling someone on the phone), or asynchronous (e.g., sending someone email).
- Sequence diagrams defined during requirements analysis should ***not***:
  - ✓ include design objects;
  - ✓ specify message signatures in any detail;

Sequence Diagrams: The Basic Idea

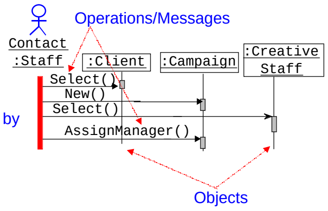


Example: Add a New Campaign

- Getting back to the use case "Add a new campaign"

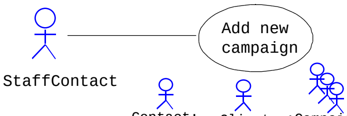


Add new campaign
✓Find client by name
✓Create new campaign
✓Find creative staff member by name
✓Assign campaign manager

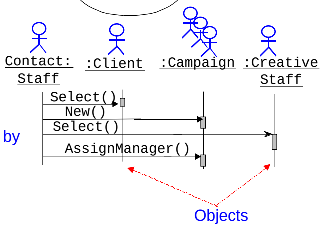


Add another New Campaign

- Getting back to the use case "Add a new campaign"

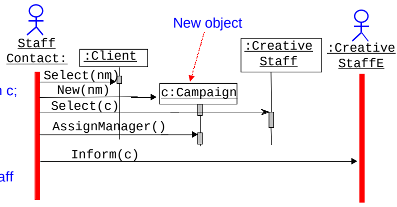


Add new campaign
✓Find client by name
✓Create new campaign
✓Find creative staff member by name
✓Assign campaign manager

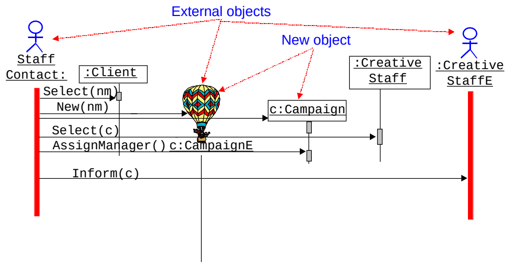


A More Realistic Example

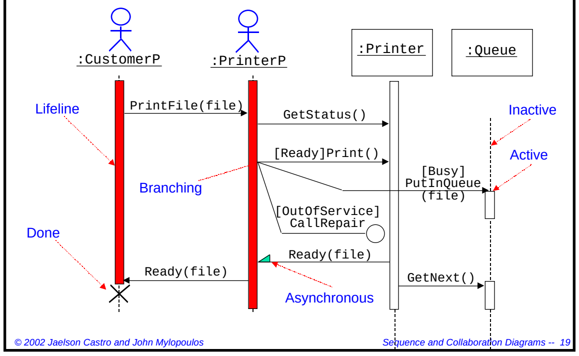
Add new campaign
✓Find client by name;
✓Create new campaign c;
✓Assign creative staff member to c;
✓Assign campaign manager;
✓Inform the creative staff person.



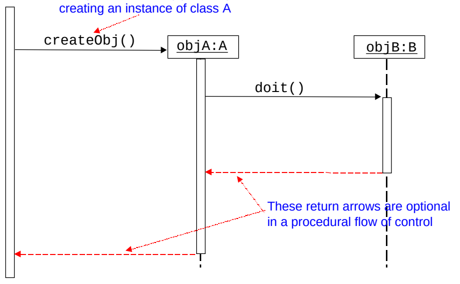
An Even More Realistic Example



Another Example: Print Shop



Flow of Control



This procedure call (operation) is creating an instance of class A

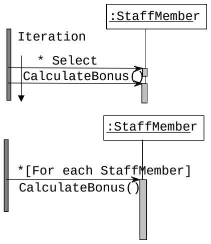
These return arrows are optional in a procedural flow of control

Iteration

- Iteration (repetition of an operation) is shown with an asterisk
- Each StaffMember will be selected in turn
- Once selected, the CalculateBonus message will be sent to the one currently selected
- There is only one loop!

```
Calculate Staff Bonuses
     Description
     Start
        For Each StaffMember
           Select next Staff Member
           Calculate Bonus for Staff
           Member
```

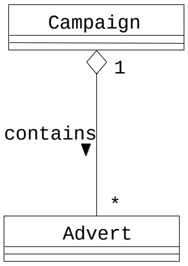
or



Drawing Sequence Diagrams

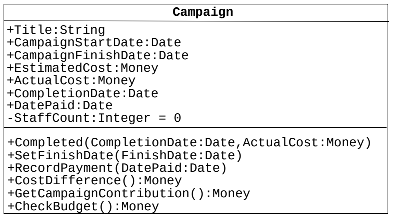
- For a particular use case, start by identifying which objects and actors might be involved.
- You may not get this right, but you can always change it.
- Imagine that there is a use case required by Agate called Check Campaign Budget
- Each Campaign has an EstimatedCost attribute and each Advert has an EstimatedCost attribute.
- The purpose of the use case is to check that the total estimated cost of all the adverts is less than that for the campaign as a whole.
- ...Which objects are involved here?

Campaign and Advert

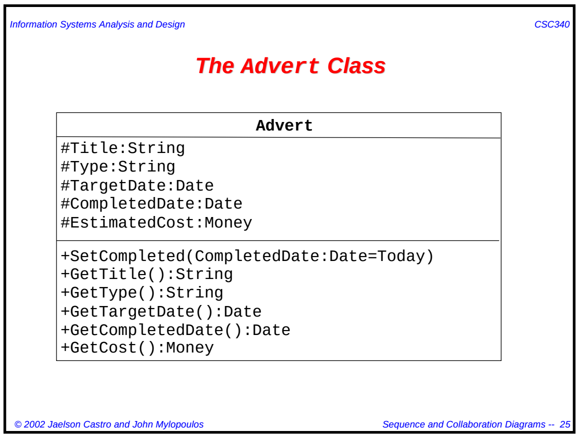


Class diagram showing aggregation

The Campaign Class

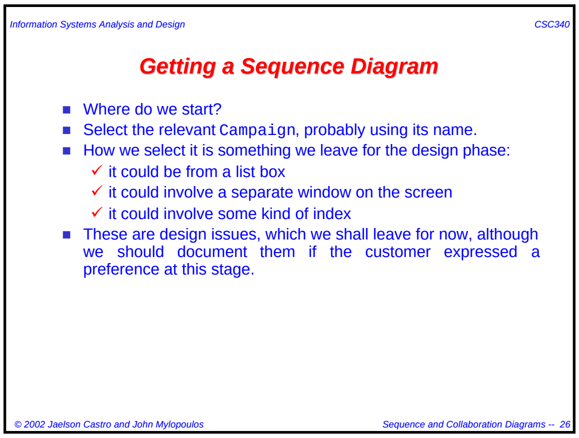


```
Campaign
+Title:String
+CampaignStartDate:Date
+CampaignFinishDate:Date
+EstimatedCost:Money
+ActualCost:Money
+CompletionDate:Date
+DatePaid:Date
-StaffCount:Integer = 0
+Completed(CompletionDate:Date,ActualCost:Money)
+SetFinishDate(FinishDate:Date)
+RecordPayment(DatePaid:Date)
+CostDifference():Money
+GetCampaignContribution():Money
+CheckBudget():Money
```



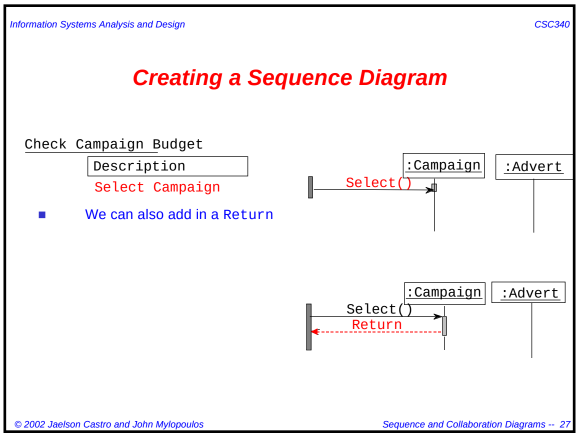
The `Advert` Class

```
Advert
#Title:String
#Type:String
#TargetDate:Date
#CompletedDate:Date
#EstimatedCost:Money
+SetCompleted(CompletedDate:Date=Today)
+GetTitle():String
+GetType():String
+GetTargetDate():Date
+GetCompletedDate():Date
+GetCost():Money
```



Getting a Sequence Diagram

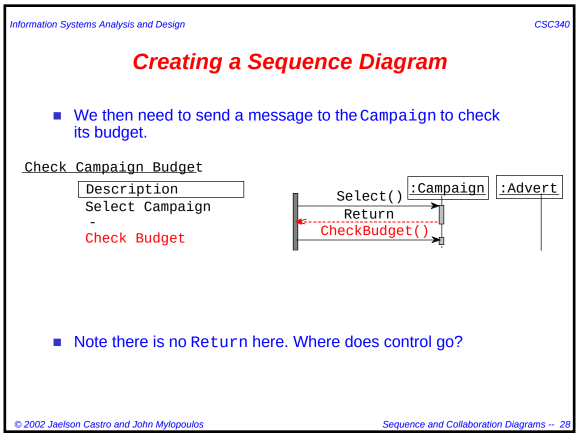
- Where do we start?
- Select the relevant `Campaign`, probably using its name.
- How we select it is something we leave for the design phase:
  - ✓ it could be from a list box
  - ✓ it could involve a separate window on the screen
  - ✓ it could involve some kind of index
- These are design issues, which we shall leave for now, although we should document them if the customer expressed a preference at this stage.



Creating a Sequence Diagram

```
Check Campaign Budget
        Description
        Select Campaign
```

- We can also add in a `Return`

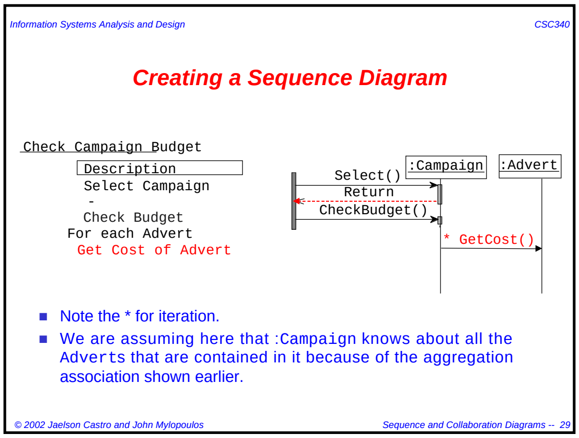


Creating a Sequence Diagram

- We then need to send a message to the `Campaign` to check its budget.

```
Check Campaign Budget
        Description
        Select Campaign
        -
        Check Budget
```

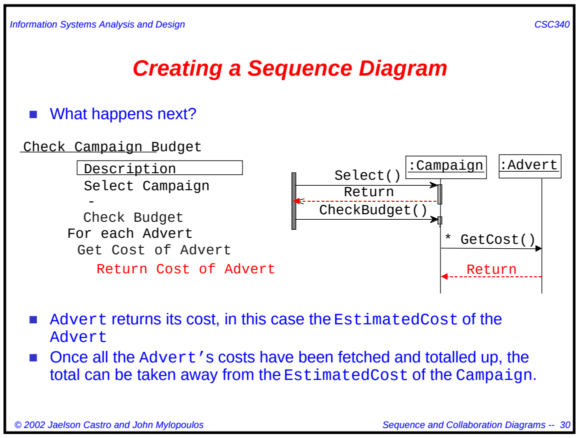
- Note there is no `Return` here. Where does control go?



Creating a Sequence Diagram

```
Check Campaign Budget
        Description
        Select Campaign
        -
        Check Budget
      For each Advert
        Get Cost of Advert
```

- Note the * for iteration.
- We are assuming here that `:Campaign` knows about all the `Advert`s that are contained in it because of the aggregation association shown earlier.



Creating a Sequence Diagram

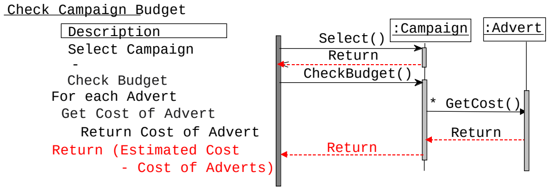
- What happens next?

```
Check Campaign Budget
        Description
        Select Campaign
        -
        Check Budget
      For each Advert
        Get Cost of Advert
          Return Cost of Advert
```

- `Advert` returns its cost, in this case the `EstimatedCost` of the `Advert`
- Once all the `Advert's` costs have been fetched and totalled up, the total can be taken away from the `EstimatedCost` of the `Campaign`.

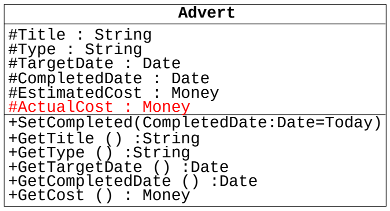
Creating a Sequence Diagram

- This has to happen for every Advert in the Campaign, so there's a loop



- Once all the Advert's costs have been fetched and totalled up, the total can be taken away from the EstimatedCost of the Campaign.

...Back to Class Diagrams...



```
Advert
#Title : String
#Type : String
#TargetDate : Date
#CompletedDate : Date
#EstimatedCost : Money
#ActualCost : Money
+SetCompleted(CompletedDate:Date=Today)
+GetTitle () :String
+GetType () :String
+GetTargetDate () :Date
+GetCompletedDate () :Date
+GetCost () : Money
```

- We could add a new attribute to Advert called ActualCost, which is set when an Advert has been completed.
- Now GetCost() can return the ActualCost if it exists, otherwise it uses EstimatedCost().

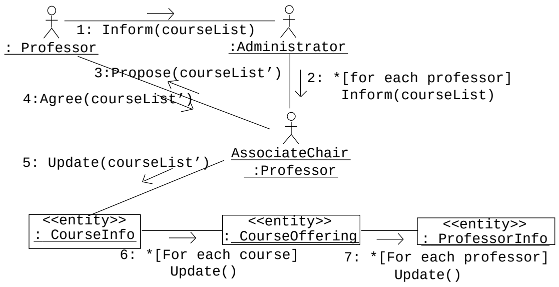
How to Use Sequence Diagrams

- In general, you may need several sequence diagrams to describe a single use case.
- A use case may involve complex control logic; sequence diagrams on the other hand should remain easy to read and understand.
- For a complex use case, use several sequence diagrams, each of which describes a possible scenario for the use case.

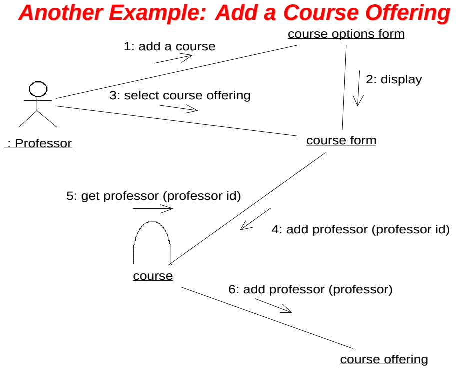
Collaboration Diagrams

- These diagrams are comparable to sequence diagrams. In fact, you can map every sequence diagram to an equivalent collaboration diagram and vice versa.
- Collaboration diagrams show interaction without the time dimension, but do include object links.
- Like sequence diagrams, collaboration diagrams are intended to model scenaria; each scenario describes a possible sequence of events and actions.
- Sequence diagrams are helpful because they capture visually the sequence of events over time.
- Collaboration diagrams capture more directly the interactions between actors and objects.
- **Note:** All operations shown on collaboration and sequence diagrams must be present in the destination classes.

Example: Select Courses to Teach



Another Example: Add a Course Offering


course options form
1: add a course
2: display
3: select course offering
: Professor
course form
5: get professor (professor id)
4: add professor (professor id)
course
6: add professor (professor)
course offering

Additional Readings

- [Booch99] Booch, G. et al. *The Unified Modeling Language* User Guide. Chapters 15, 18, 27. Addison-Wesley.
- [Jacobson92] Jacobson, I. et al. *Object-Oriented Software Engineering: A Use-Case Driven Approach*. Addison-Wesley.
- [Fowler00] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modelling Language.* Chapter 5. Addison-Wesley.