COP 3330: Object-Oriented Programming
Summer 2007

UML

Instructor : Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2007

School of Electrical Engineering and Computer Science
University of Central Florida

UML Basics

- UML contains 3 main types of diagrams.

1. *Static diagrams*
   - Describe the unchanging logical structure of software elements by depicting classes, objects, and data structures, and the relationships that exist between them.
2. *Dynamic diagrams*
   - Illustrate how software entities change during execution by depicting the flow of execution, or the way entities change state.
3. *Physical diagrams*
   - Show the unchanging physical structure of software entities by depicting physical entities such as source files, libraries, binary files, data files, etc., and the relationships that exist between them.

A Java Example – TreeMap.java

```
public class TreeMap
    TreeMapNode topNode = null;

    public void add(Comparable key, Object value) {
        if (topNode == null)
            topNode = new TreeMapNode(key, value);
        else
            topNode.add(key,value);
    }//end add

    public Object get(Comparable key) {
        return topNode == null ? null : topNode.find(key);
    }

    class TreeMapNode {
        private final static int LESS = 0;
        private final static int GREATER = 1;
        private Comparable itsKey;
        private Object itsValue;
        private TreeMapNode nodes[] = new TreeMapNode[2];
```

```
public TreeMapNode (Comparable key, Object value) {
    itsKey = key;
    itsValue = value;
}

public Object find(Comparable key) {
    if(key.compareTo(itsKey) == 0) return itsValue;
    return(findSubNodeForKey(selectSubNode(key), key);
}

private int selectSubNode(Comparable key) {
    return (key.compareTo(itsKey) < 0) ? LESS : GREATER;
}

private Object findSubNodeForKey(int node, Comparable
    return nodes[node] == null ? null : nodes[node].find(key);
}

public void add(Comparable key, Object value) {
    if(key.compareTo(itsKey) == 0)
         itsValue = value;
    else
         addSubNode(selectSubNode(key), key, value);
}
```

```
private void addSubNode(int node,
                        Comparable key,
                        Object, value)  {
    if(nodes[node] == null)
        nodes[node] = new TreeMapNode(key, value);
    else nodes[node].add(key, value);
}
```

The ? : operator in Java can sometimes be used in place of a conditional statement. The operator has the following form:

testexpression ? expr1 : expr2

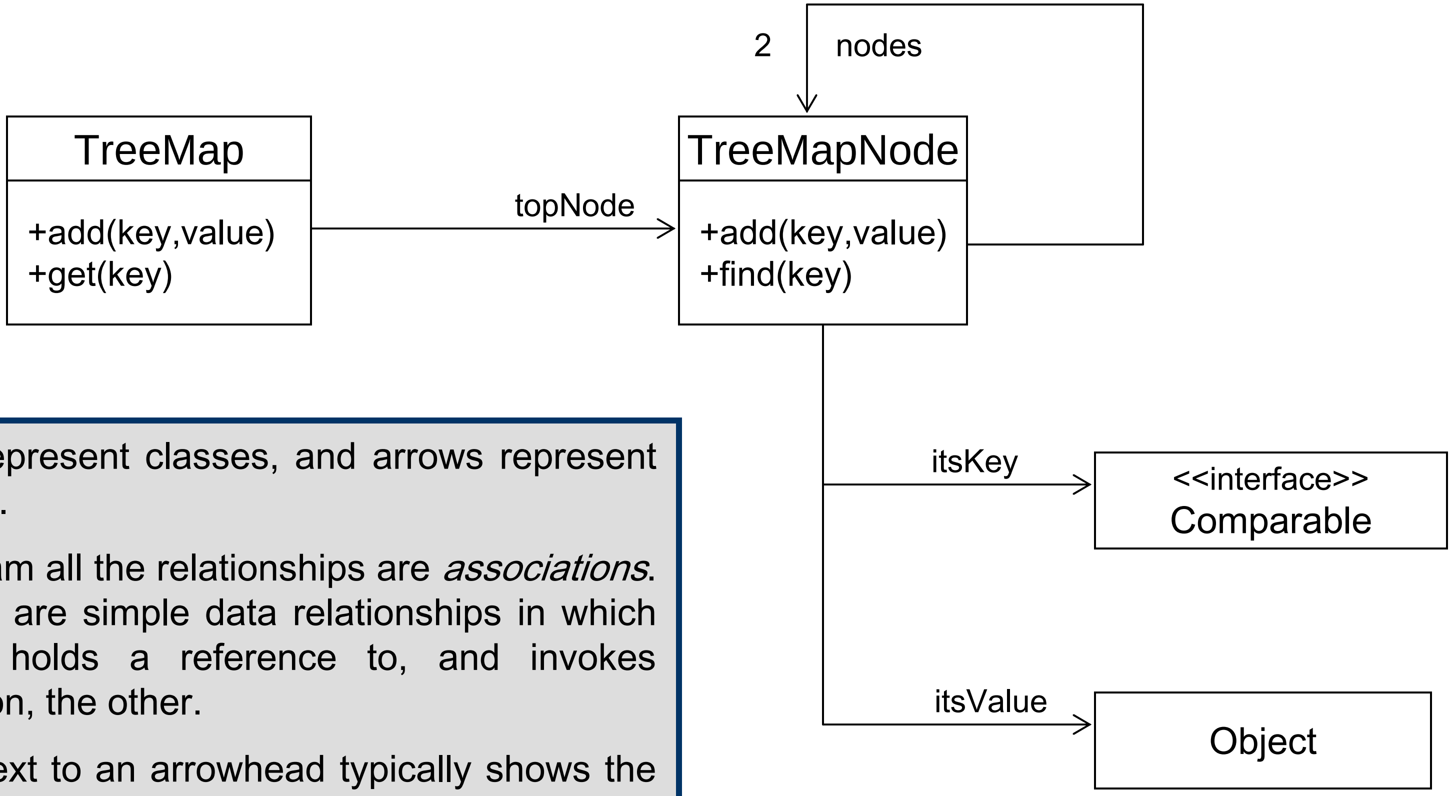
When executed, *testexpression* is evaluated first. if *testexpression* evaluates to true, then the value of the operation is *expr1*; otherwise, the value of the operation is *expr2*. A colon separates the two expressions.

```
int min = (input1 <= input2) ? input1 : input2;
```

equivalent

```
int min;
if (input1 <= input2) {
   min = input1;
}
else min = input2;
```

Class Diagram for TreeMap



Rectangle represent classes, and arrows represent relationships.

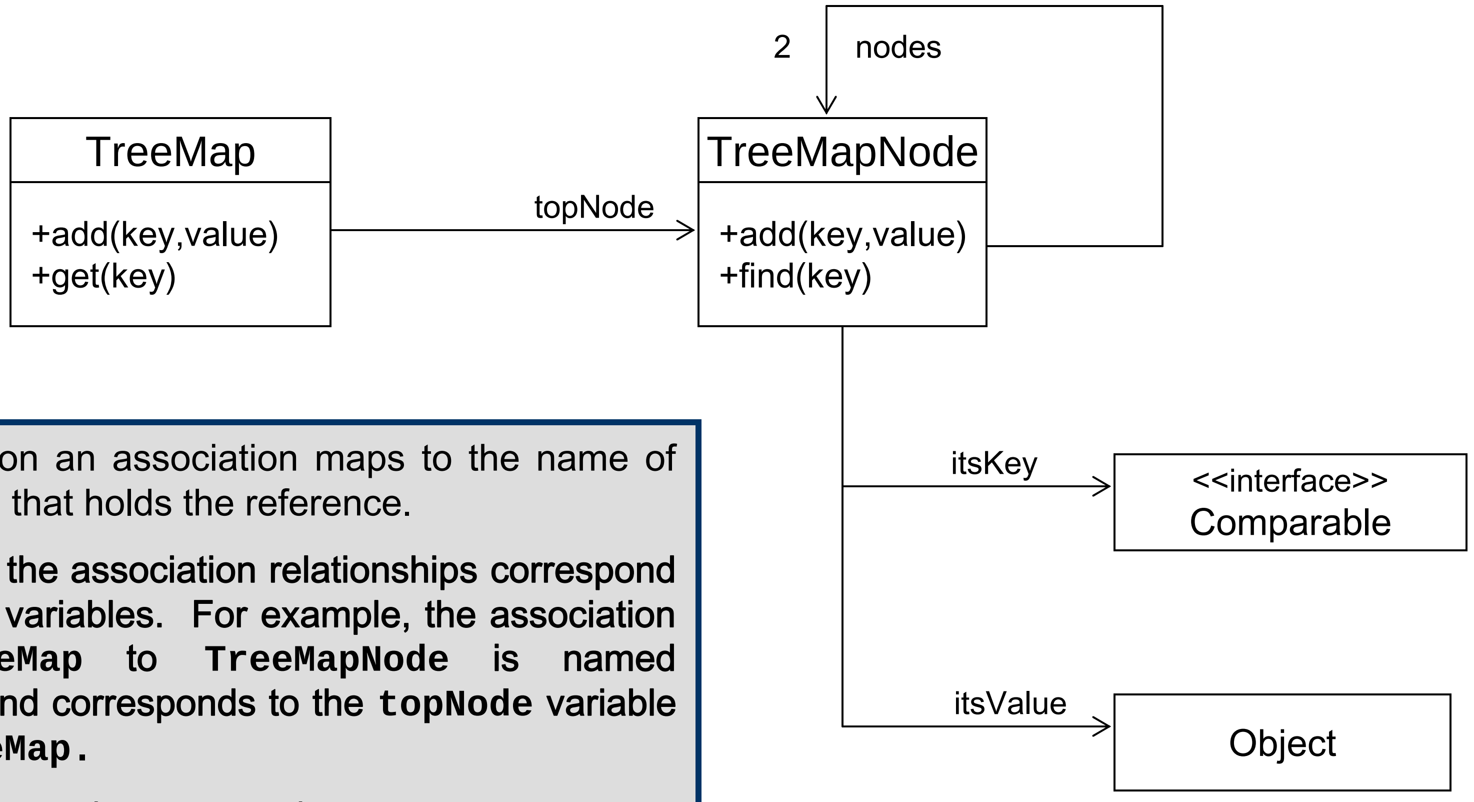
In this diagram all the relationships are *associations*. Associations are simple data relationships in which one object holds a reference to, and invokes methods upon, the other.

A number next to an arrowhead typically shows the number of instances held by the relationship. If that number is greater than one then some kind of container (usually an array) is implied.

Class Diagrams

- The class diagram on the previous page illustrates the major classes and relationships in the program.
- It shows that there is a `TreeMap` class that has public methods named add and get.
- It shows that `TreeMap` holds a reference to a `TreeMapNode` in a variable named `topNode`.
- It shows that each `TreeMapNode` holds a reference to two other `TreeMapNode` instances in some kind of container named nodes.
- It also shows that each `TreeMapNode` instance holds references to two other instances in variables named `itsKey` and `itsValue`.
  - The `itsKey` variable holds a reference to some instance that implements the `Comparable` interface.
  - The `itsValue` variable simply holds a reference to some object.

Class Diagram for TreeMap



The name on an association maps to the name of the variable that holds the reference.

Notice how the association relationships correspond to instance variables. For example, the association from **`TreeMap`** to **`TreeMapNode`** is named **`topNode`** and corresponds to the **`topNode`** variable within **`TreeMap.`**

Class icons can have more than one compartment. The top compartment always holds the name of the class. The other compartments describe functions and variables.

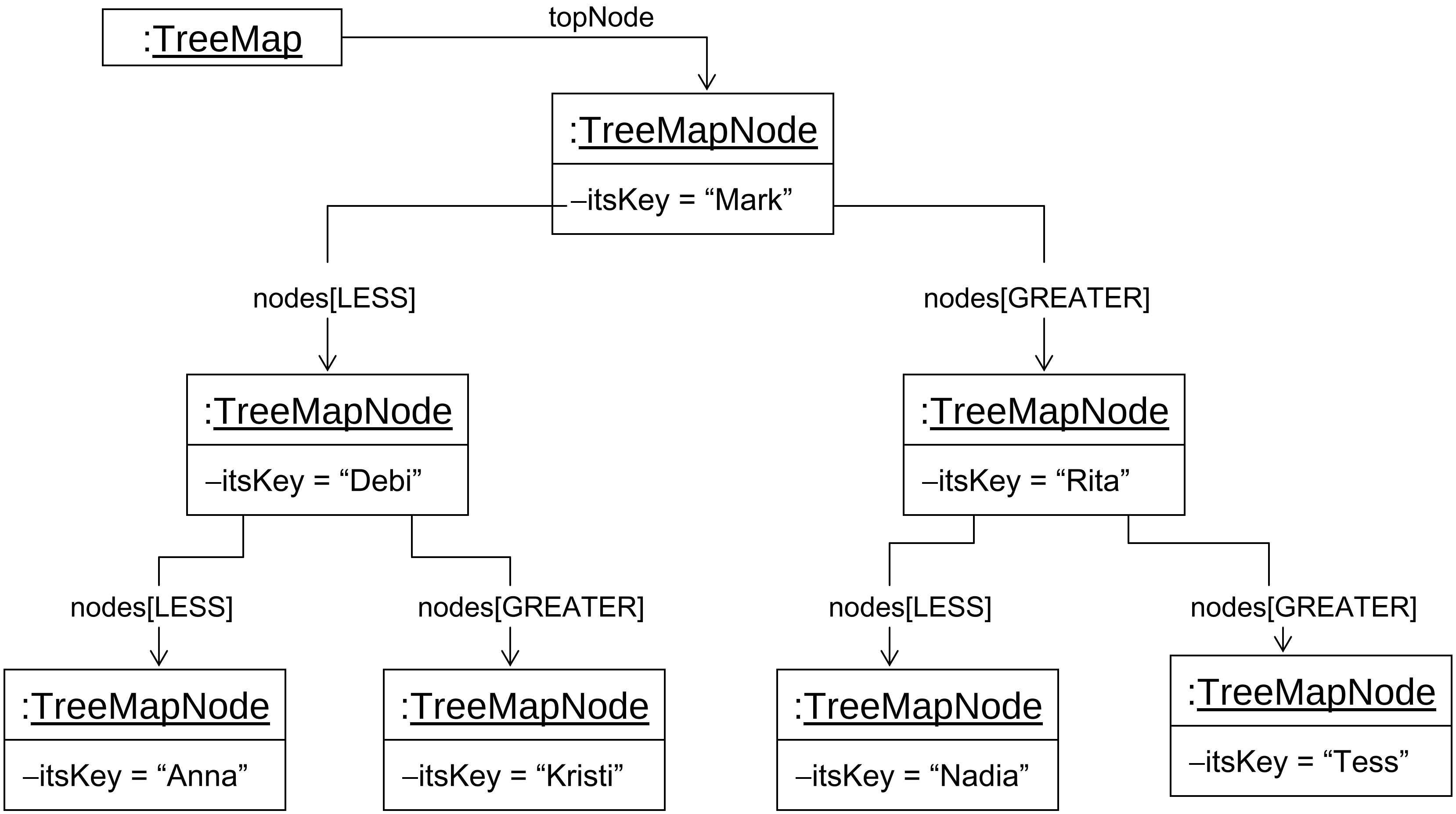
Object Diagrams

- The object diagram (example on the next page) illustrates a set of objects and the relationships at a particular moment during the execution of the system. You can view it as a snapshot of the memory.

- In object diagrams, the rectangular icons represent objects. You can tell that they are objects rather than classes because their names are underlined.
  - The name after the colon is the name of the class that the object belongs to. Note that the lower compartment of each object shows the value of that object's `itsKey` variable.

- The relationships between the objects are called links, and are derived from the associations in the class diagram (see page 8).
  - Notice that the links are named for the two array cells in the `nodes` array.

Object Diagram for TreeMap



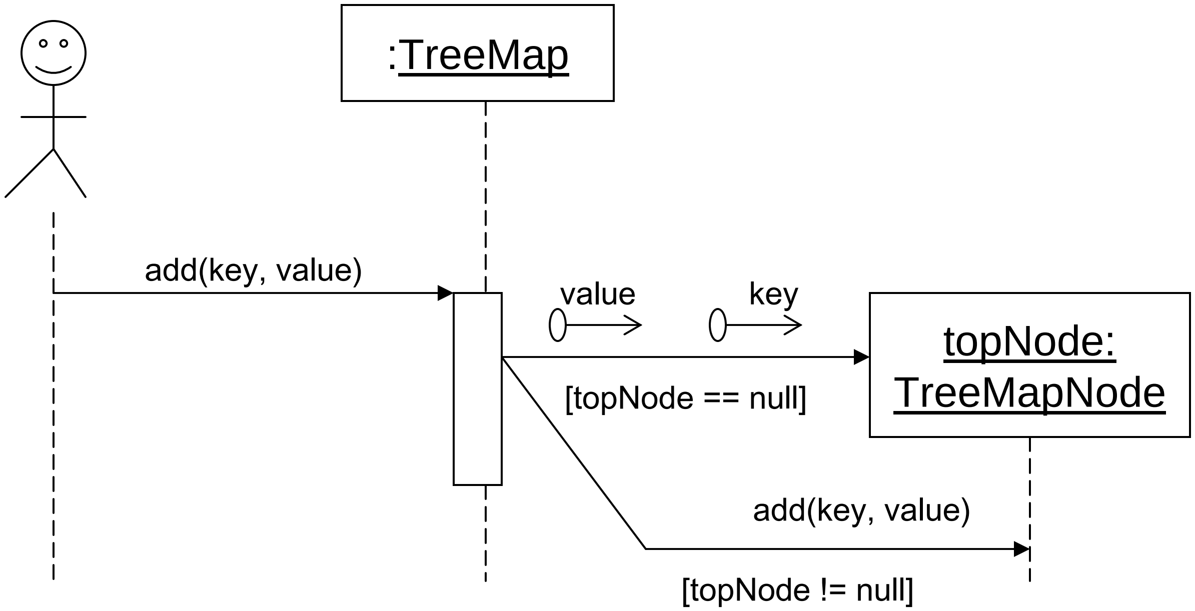
Sequence Diagrams

- The sequence diagram (example on page 13) describes how a method is implemented. The example on the next page illustrates the implementation of the `TreeMap.add` method.
- The stick figure represents an unknown caller. This caller invokes the add method on a `TreeMap` object.
  - If the topNode variable is null, then `TreeMap` responds by creating a new `TreeMapNode` and assigning it to `topNode`.
  - Otherwise, `TreeMap` sends the `add` message (the invocation) to `topNode`.

Sequence Diagrams (cont.)

- The Boolean expressions inside the square brackets are called guards. They show which path is to be taken.
- The message arrow that terminates on the TreeMapNode icon represents construction.
- The little arrows with circles are called data tokens. In this case they depict the construction arguments.
- The skinny rectangle below `TreeMap` is called an activation and depicts how much time the add method executes.

Sequence Diagram for TreeMap.add



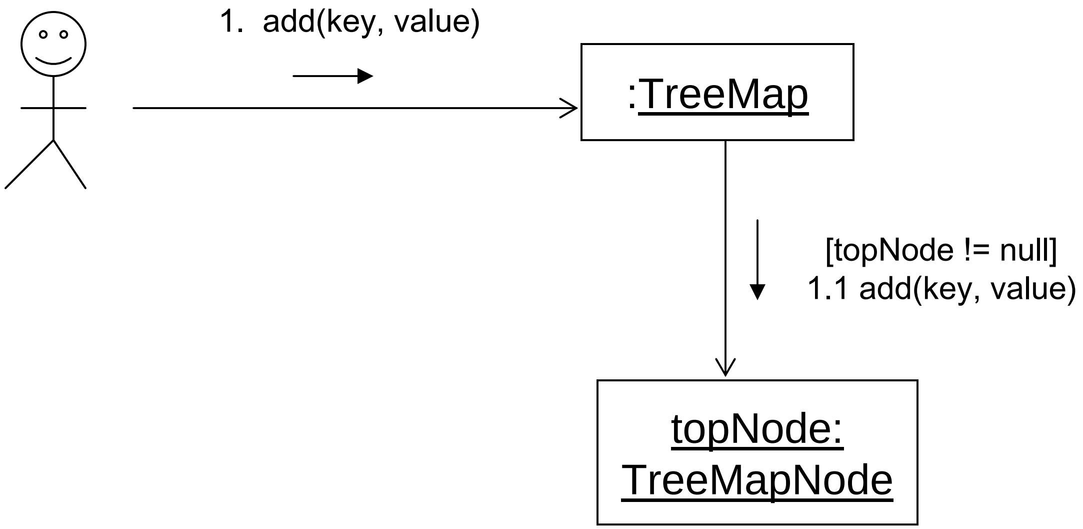
Collaboration Diagrams

- Collaboration diagrams contain the same information that sequence diagrams contain. However, whereas sequence diagrams make the order of the messages clear, collaboration diagrams make the relationships between the objects clear.
- The collaboration diagram shown on the next page illustrates the case of the `TreeMap.add` method when `topNode` is not null.
- The objects are connected by relationships called links. A link exists wherever one object can send a message to another.

Collaboration Diagrams

- Traveling over those links are the messages themselves. They are depicted as the smaller arrows. The messages are labeled with the name of the message, its sequence number, and any guards that apply.
- The dot structure of the sequence number shows the calling hierarchy.
  - The `TreeMap.add` method (message 1) invokes the `TreeMapNode.add` method (message 1.1). Thus, message 1.1 is the first message sent by the method invoked by message 1.

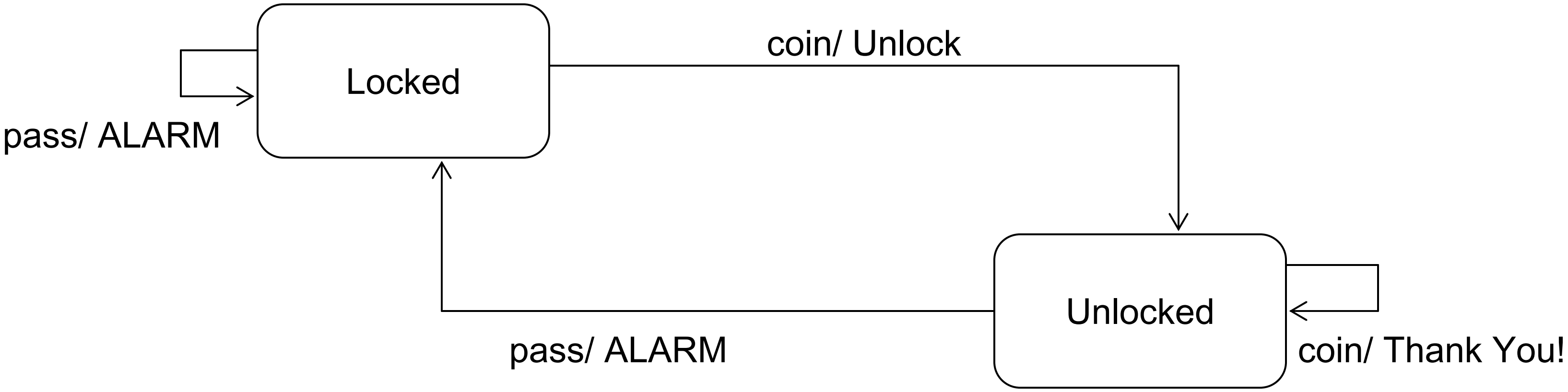
Collaboration Diagram for TreeMap.add



UML and FSA

- UML has a very comprehensive notation for finite state machines. The following example illustrates just the barest subset of this notation.
- Consider a turnstile for a subway. There are two states named `Locked` and `Unlocked`. Two events may be sent to the machine. The `coin` event means that the user has dropped a coin into the turnstile. The `pass` even means that the user has passed through the turnstile.
- The arrows are called transitions. They are labeled with the event that triggers the transition and the action that the transition performs. When a transition is triggered it causes the state of the system to change.

UML and FSA



If we are in the Locked state and a coin event occurs, then a transition to the Unlocked state occurs and the Unlock function is invoked.

If we are in the Unlocked state and a pass event occurs, then a transition to the Locked state occurs and the Lock function is invoked.

If we are in the Unlocked state and a coin event occurs, then the system remains in the Unlocked state and the Thank You! function is invoked.

If we are in the Locked state and a pass event occurs, then the system remains in the Locked state and the Alarm function is invoked.

State diagrams such as this are extremely useful for determining the way a system behaves. They provide an opportunity to explore what the system should do in unexpected cases, such as when the user deposits a coin, and then deposits another coin for no apparent reason.

Working With UML and Modeling

- Why do engineers build models? Why do aerospace engineers build models of aircraft? Why do structural engineers build models of bridges? What purposes do these models serve?

Models are built to find out if something will work.

Engineers build models to see if there designs will work.

Aerospace engineers build models of aircraft and put them in wind tunnels to see if they will fly.

Structural engineers build models of bridges to see if they will stand.

Architects build models of buildings to see if their clients will like the way they look.

Working With UML and Modeling (cont.)

- This implies that models must be testable.
  - It does no good to build a model if there are no criteria you can apply to that model in order to test it. If you can't evaluate the model, the model has no value.
- Why don't aerospace engineers just build the plane and try to fly it? Why don't structural engineers just build the bridge and see if it stands?

Because airplanes and bridges are a lot more expensive to to build than the models.

Designs are investigated with models when the models are much cheaper to construct than the real thing.

Why Build Models of Software

- Can a UML diagram be tested? Is it much cheaper to create and test than the software it represents?
  - In both cases the answer is nowhere near as clear as it is for aerospace engineers and structural engineers.
- There are no firm criteria for testing UML diagrams. we can look at it, evaluate it, apply principles and patterns to it, but in the end the evaluation is still very subjective.
- UML diagrams are less expensive to create than software is to write, but not by a huge factor.
  - Indeed, there are times when it is easier to change the source code than it is to change the UML diagram!
- When does it make sense to use UML?

Why Build Models of Software (cont.)

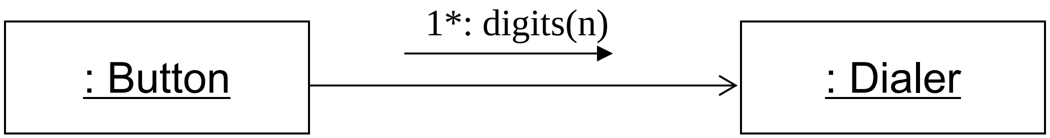
- We make use of UML whenever we have:
  1. Something definitive to test, and
  2. using UML to test it is cheaper than writing the code to test it.

- Blueprints can be drawn without digging foundation, pouring concrete, or hanging windows. In short, it is much cheaper to plan a building up front than it is to try to build it without a plan. It doesn't cost much to throw away a faulty blueprint, but it costs a lot to tear down a faulty building.
- Once again things are not so clear-cut in software. It is not at all clear that drawing UML diagrams is much cheaper than writing code. Many project teams have spent more on UML diagrams than they have on the code.

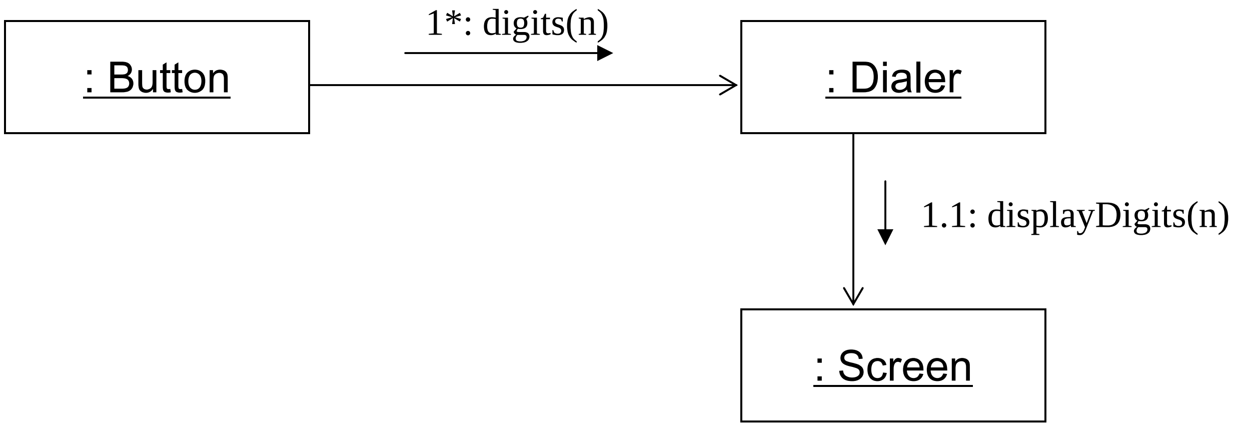
How To Start A Model

- Typically, the easiest place to start the modeling of a system is by modeling the behavior of the system using sequence diagrams.
- As a running example, let’s consider the design of the software that controls a cell phone. How does this software make the phone call?
- We might start out by imagining that the software detects each button the user presses and sends a message to some object that controls dialing.
- We’ll start out by drawing a `Button` object and a `Dialer` object and showing the `Button` sending many `digit` messages to the `Dialer`.

How To Start A Model (cont.)

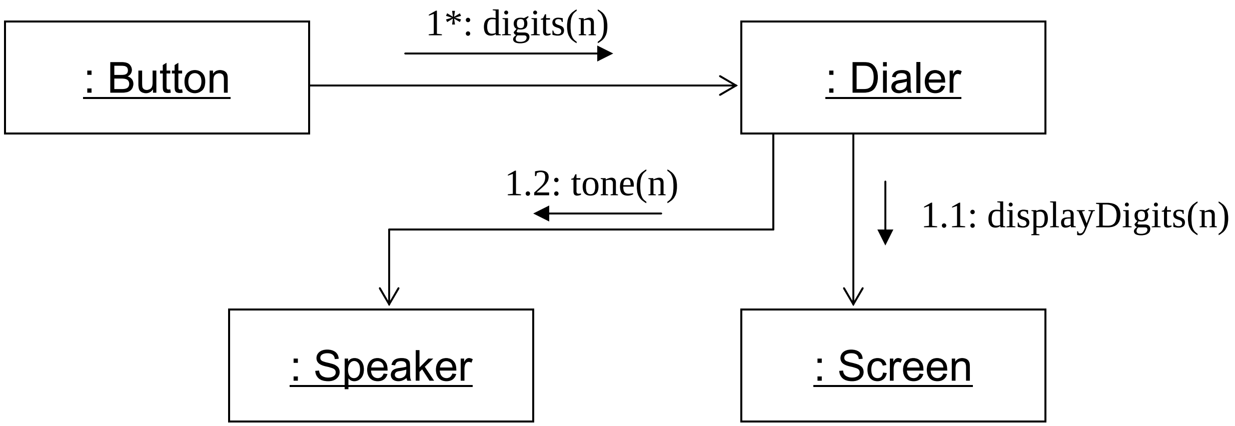


- What will the `Dialer` do when it receives a `digit` message? It will need to display the `digit` on the screen. So we'll refine our sequence diagram to send `displayDigit` to the `Screen` object.



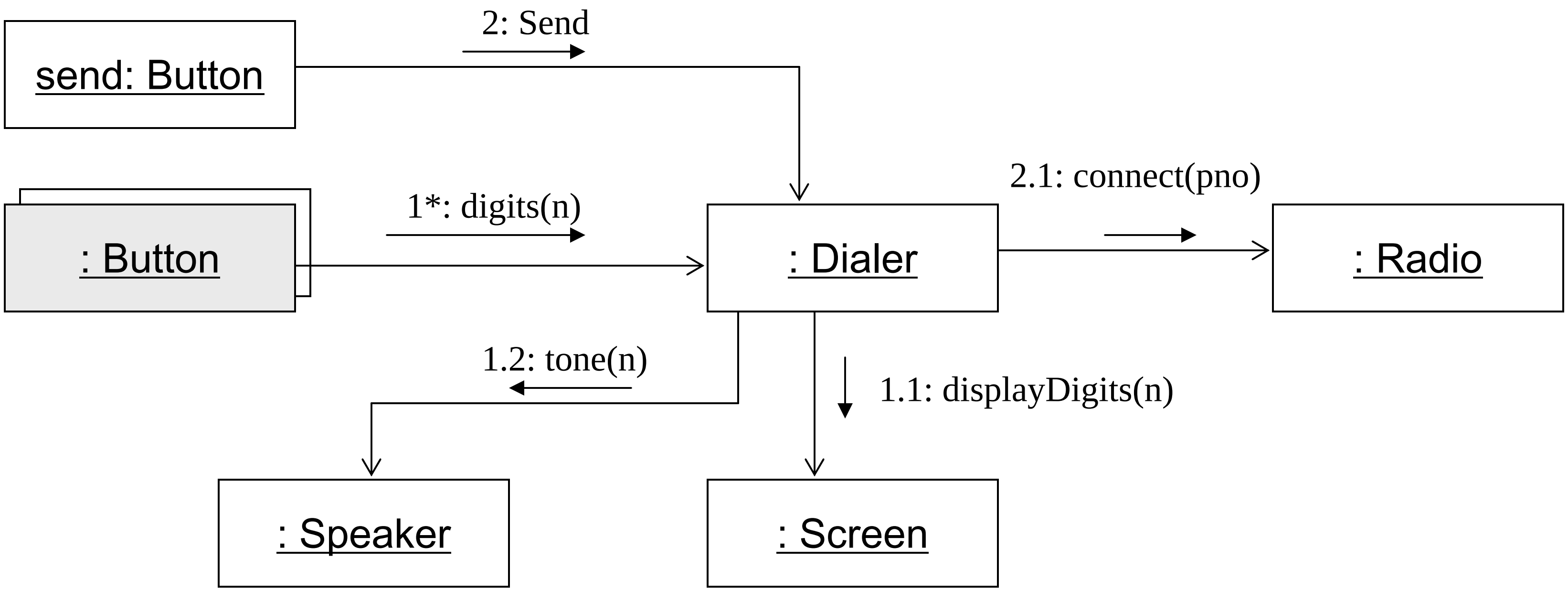
How To Start A Model (cont.)

- Next, the `Dialer` should cause a tone to be emitted from the speaker. This means that we'll need to send the `tone` message to the `Speaker` object. So we'll again refine our sequence diagram to look like the following:



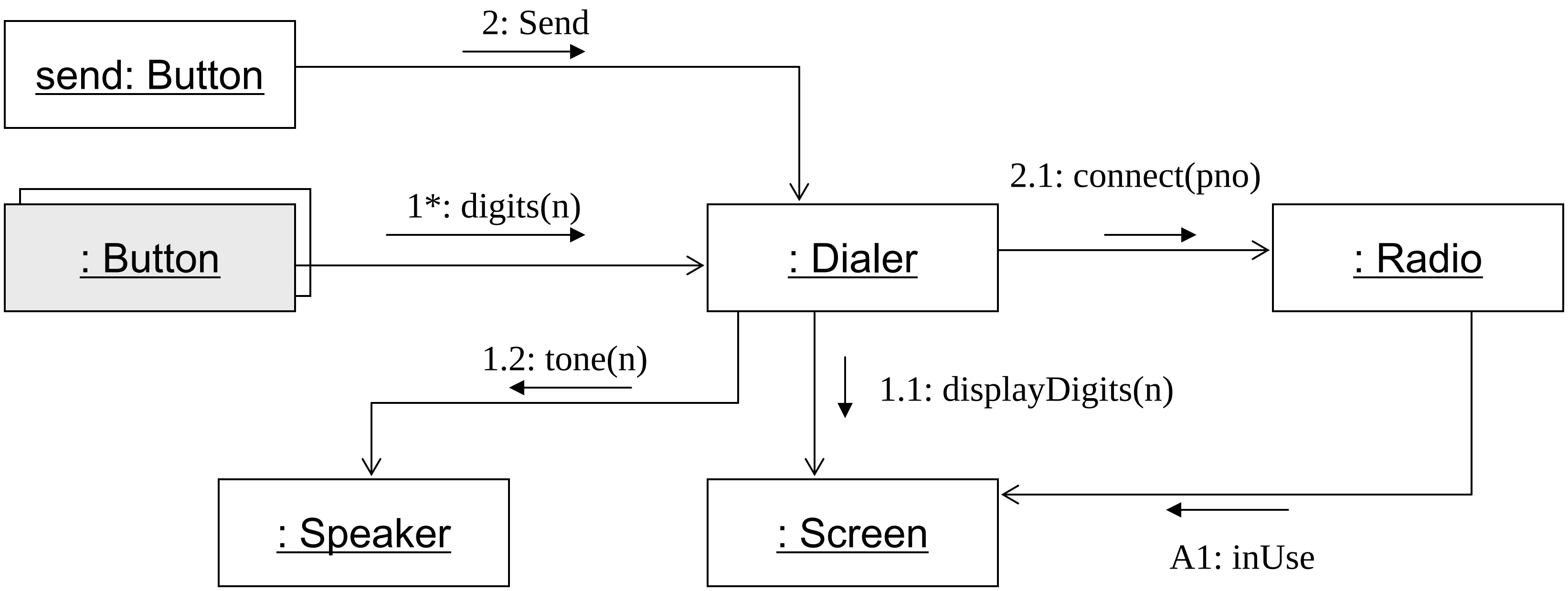
How To Start A Model (cont.)

- At some point the user will press the Send button, indicating that they want the call to go through. At that point we'll have to tell the cellular radio to connect to the cellular network and pass along the phone number than was dialed. This will update our sequence diagram to the following:



How To Start A Model (cont.)

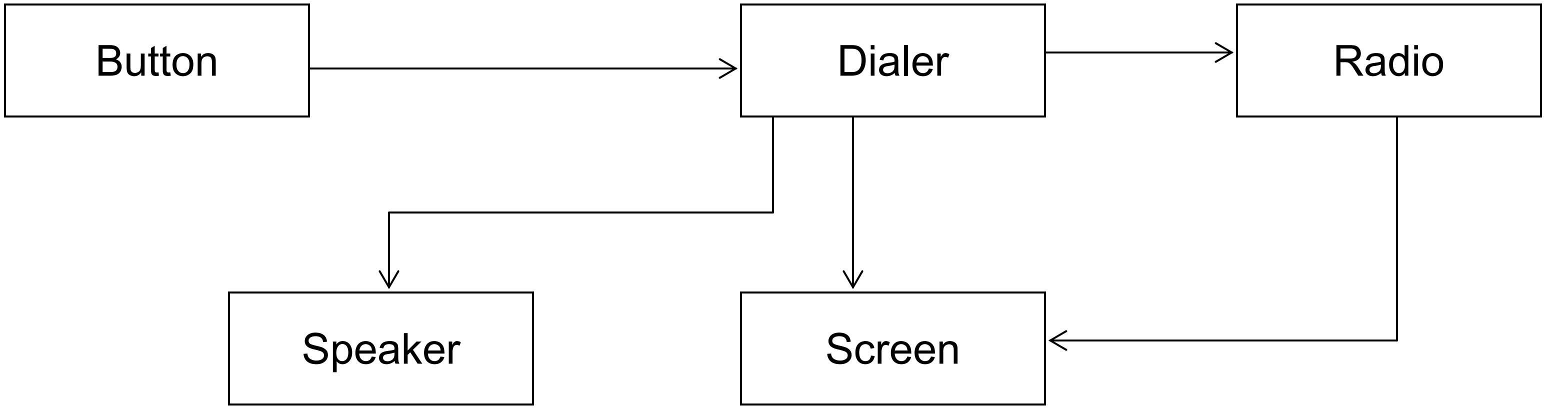
- Once the connection has been established, the Radio can tell the Screen to light up the "in use" indicator. This message will almost certainly be sent in a different thread of control (which is denoted by the letter in front of the sequence number). The final collaboration diagram would look like the following:



Class Diagram for Cell Phone Problem

- Shown below is the UML class diagram for the cell phone problem.

- Notice that the class diagram contains a class for each object that appeared in the collaboration diagram we constructed, and an association for each link in the collaboration. For right now, we'll skip any aggregation and composition details.



Further Analysis Of Our Design

- Having constructed our class diagram, let’s analyze the dependencies that are shown in the diagram. Remember, we’re going to write code based on this diagram at some point.
- Why should the `Button` depend on the `Dialer`? This association would imply the following in code:

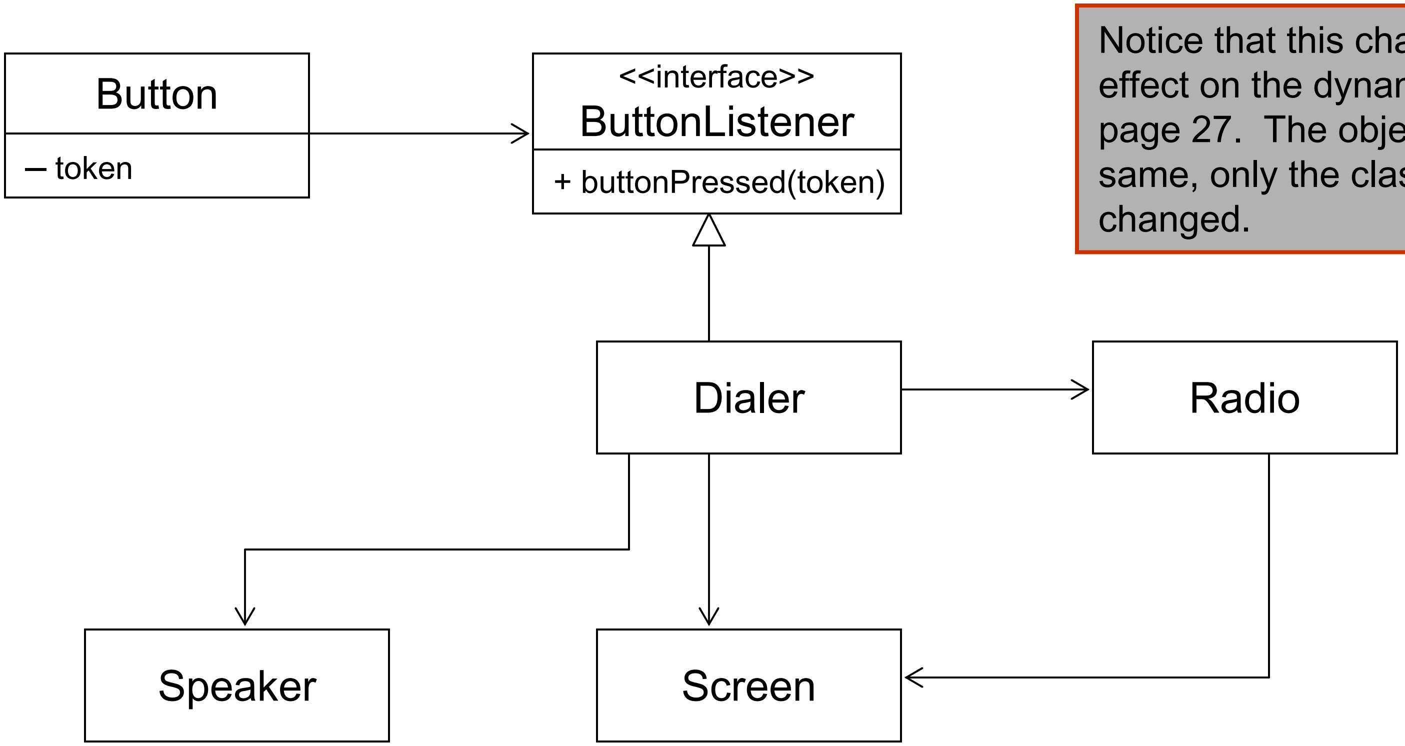
```
public class Button
{   private Dialer itsDialer;
    public Button(Dialer dialer)
              {  itsDialer = dialer; }
     ...
}
```

Further Analysis Of Our Design (cont.)

- There is no valid reason that the source code of `Button` should mention the source code of `Dialer`.
  - For example, `Button` is a class that could be used in many different contexts. We could use the `Button` class to control the on/off switch, or the menu button, or the other control buttons on the phone, not to mention the possibilities of applications other than that of a cell phone.
  - If the `Button` is bound to the `Dialer`, the we won't be able to reuse the `Button` code for other purposes.
- This problem can be fixed by inserting an interface between `Button` and `Dialer`, as shown in the UML diagram on the next page.

Isolating Button From Dialer

- Each Button is given a token that identifies it. When the Button class detects that the button has been pressed, it invokes the buttonPressed method of the ButtonListener interface, passing the token. This breaks the dependence of Button upon Dialer and allows Button to be used virtually anywhere that needs to receive button presses.

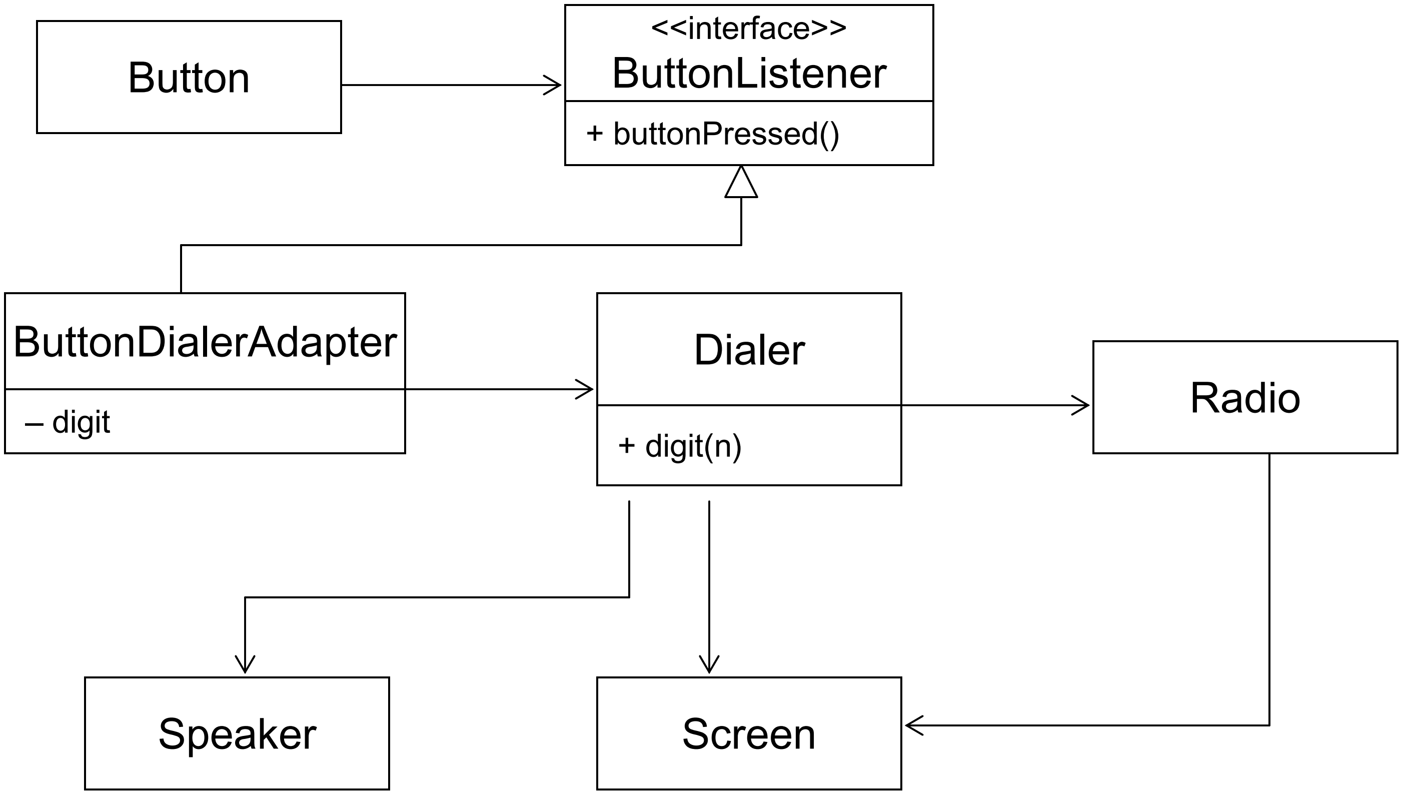


Notice that this change has no effect on the dynamic diagram on page 27. The objects are the same, only the classes have changed.

Adapting ButtonS To DialerS

- Unfortunately, the change we just made allows Dialer to know something about Button. Why should Dialer expect to get its input from ButtonListener? Why should it have a method within it named buttonPressed? What does the Dialer have to do with Button?
  - The answer is of course, nothing!
- We can solve this problem, and get rid of all the token nonsense, by using a batch of little adapters.
  - The ButtonDialerAdapter implements the ButtonListener interface. It receives the buttonPressed method and sends a digit(n) message to the Dialer. The digit passed to the Dialer is held in the adapter.
- This is illustrated by the UML diagram on the next page.

Adapting ButtonS To DialerS (cont.)



Envisioning the Code

- Once we've completed the UML diagram on the previous page, we should be able to envision the code for the ButtonDialerAdapter.

- Envisioning the code is critically important when dealing with UML diagrams. If you are drawing diagrams (modeling) and cannot envision the code that they represent, then you are in trouble with your modeling!

- Stop what you are doing and figure out how to convert it to code.

- Never let the diagrams become an end unto themselves. You must always be sure you know what code you are representing with the diagram!

- The code for the ButtonDialerAdapter is shown on the next page.

Envisioning the Code

```
public class ButtonDialerAdapter implements ButtonListener
{   private int digit;
    private Dialer dialer;
    public ButtonDialerAdapter(int digit, Dialer, dialer)
    {      this.digit = digit;
           this.dialer = dialer;
     }
    public void buttonPressed()
    {
            dialer.digit(digit);
    }
}
```

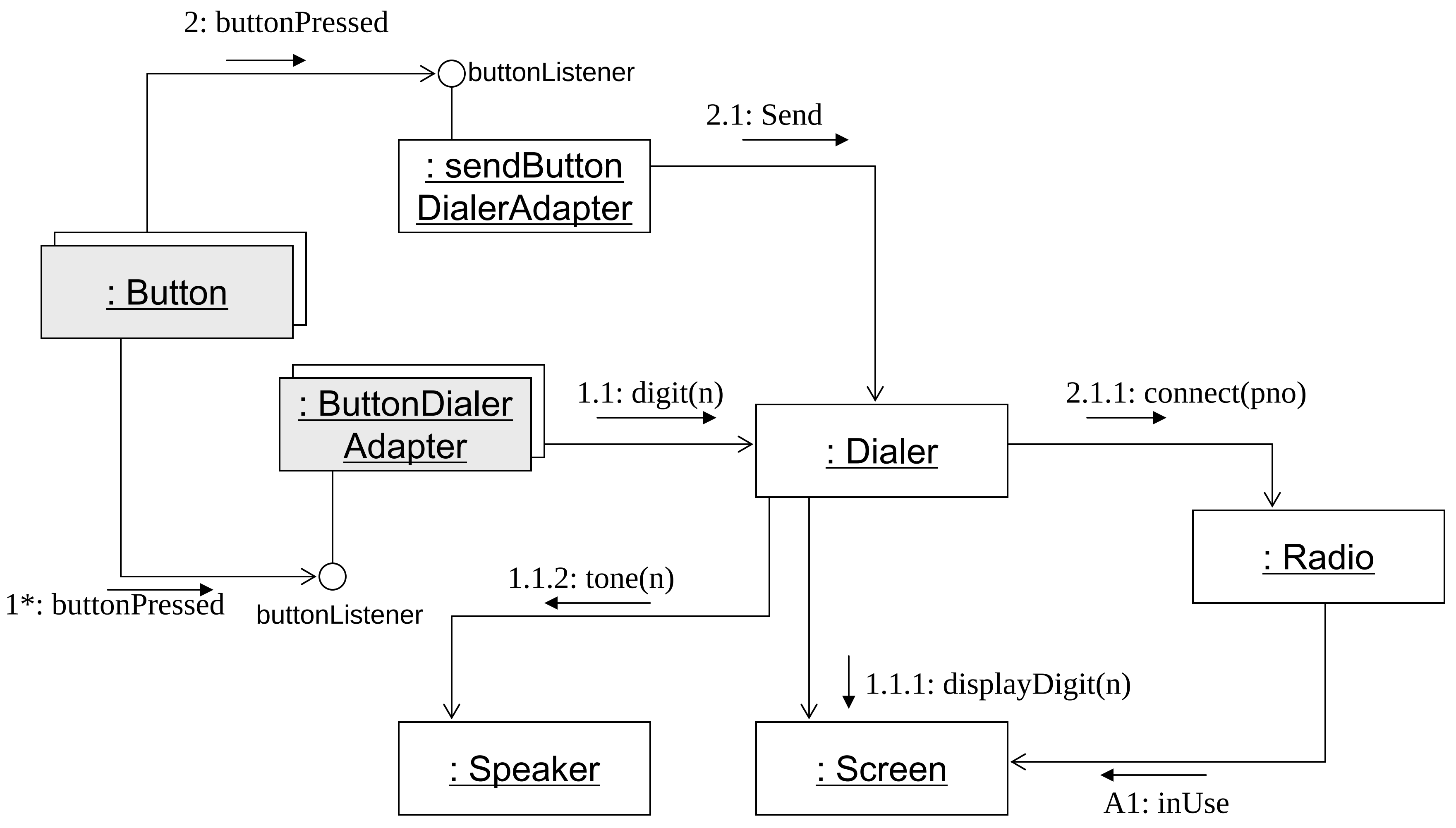
Evolution Of UML Diagrams

- Unlike the earlier change we made (adding the interface), this last change has caused the dynamic diagram (see page 27) to become invalid. The dynamic model knows nothing of the adapters we've just added.

- Since the class diagram has been updated, so too must the dynamic diagram. The updated version appears on the next page.

- This illustrates how the diagrams evolve together in an iterative fashion.
  - You start with a little bit of dynamics (objects).
  - Then you explore what those dynamics imply to the static relationships (classes).
  - You alter the static relationships according to the principles of good design.
  - Then you go back and improve the dynamic diagrams.

Updated Dynamic UML Diagram

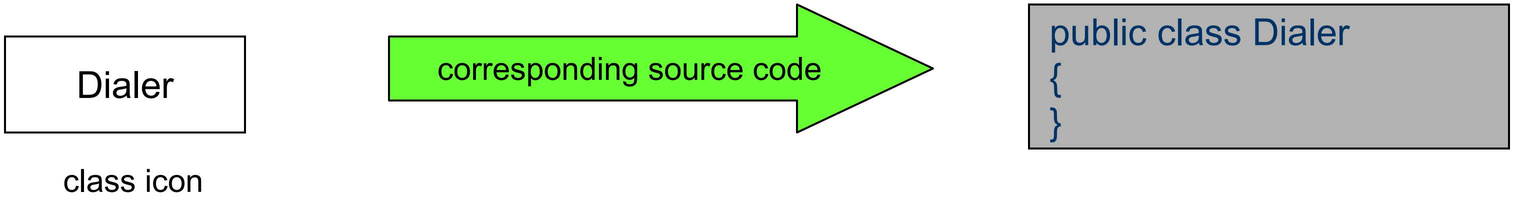


A Closer Look At Class Diagrams in UML

- UML class diagrams allow us to denote the static contents of - and the relationships between – classes.
- In a class diagram you can show the member variables, and member functions (methods) of a class.
- It is also possible to show whether one class inherits from another, or whether it holds a reference to another.
- In short, class diagrams allow us to depict all the source code dependencies between classes.
  - This is a valuable benefit. It can be much easier to evaluate the dependency structure of a system from a diagram than from source code. Diagrams make certain dependency structures visible .
  - You can *see* dependency cycles, and determine how best to break them.
  - You can see when abstract classes depend on concrete classes, and determine a strategy for rerouting such dependencies.

The Basics of Class Diagrams

- The simplest form of a class diagram consists only of a single rectangular icon which depict the class name.



Dialer

class icon

corresponding source code

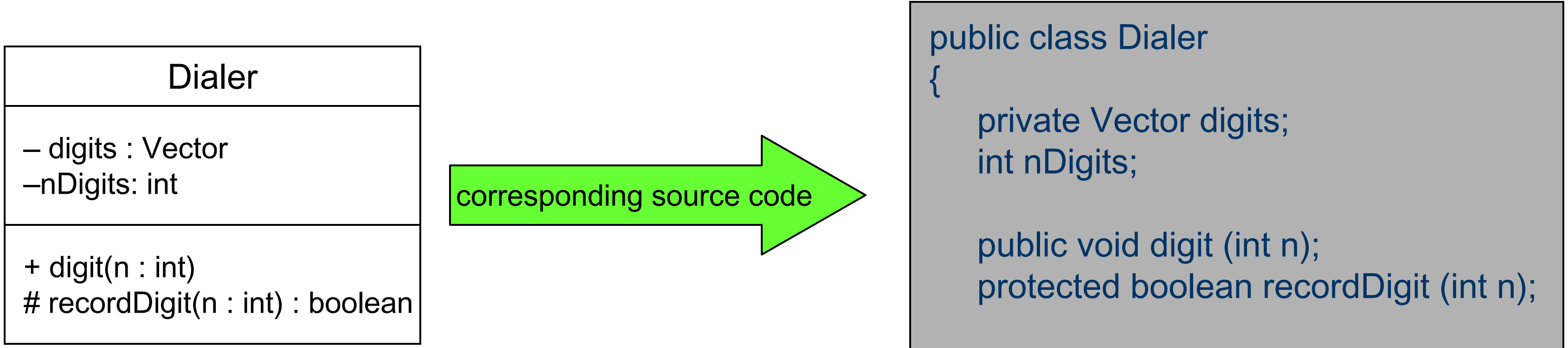
```
public class Dialer
{
}
```

  - This is a very common way to represent a class. The classes on many diagrams don’t need any more than their name to make it clear what is going on in the source code.

- A class icon can also be subdivided into compartments. The top compartment is for the name of the class, the second is for the variables of the class, and the third is for the methods of the class.

The Basics of Class Diagrams (cont.)

- The diagram below illustrates the compartments and how they translate into code.



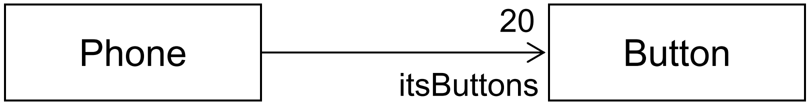
```
public class Dialer
{
    private Vector digits;
    int nDigits;

    public void digit (int n);
    protected boolean recordDigit (int n);

}
```

class icon with all 3 compartments

- For variables and functions, the preceding character represents the modifier.
  - (–) denotes private.
  - (#) denotes protected.
  - (+) denotes public.
- The return type of a method is shown after the colon following the method.

Class Diagrams - Associations

- Associations between classes most often represent instance variables that hold references to other objects. For example, in the diagram shown below, there is an association between Phone and Button.
  - The direction of the arrow indicates that Phone holds a reference to Button.
  - The name near the arrowhead is the name of the instance variable.
  - The number near the arrowhead indicates the number of references held. If no limit is set (such as a Vector, list, or some type of container) then a star is utilized to represent many.

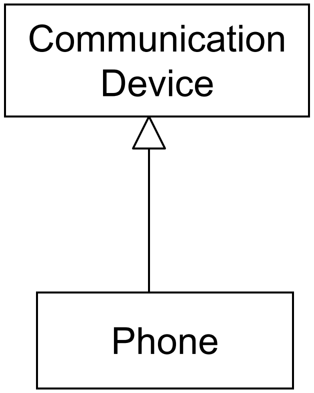


```
public class Phone
{
    private Button itsButtons[20];

}
```

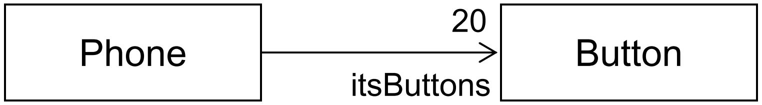
corresponding source code

Class Diagrams - Inheritance

- You have to be careful in UML when you draw arrowheads. In the diagram on the left below, the arrowhead represents an inheritance relationship. The arrowhead in the diagram on the right represents an association. If you draw the arrowheads carelessly, it will be difficult to tell whether you are depicting an association or inheritance.



Inheritance



Association

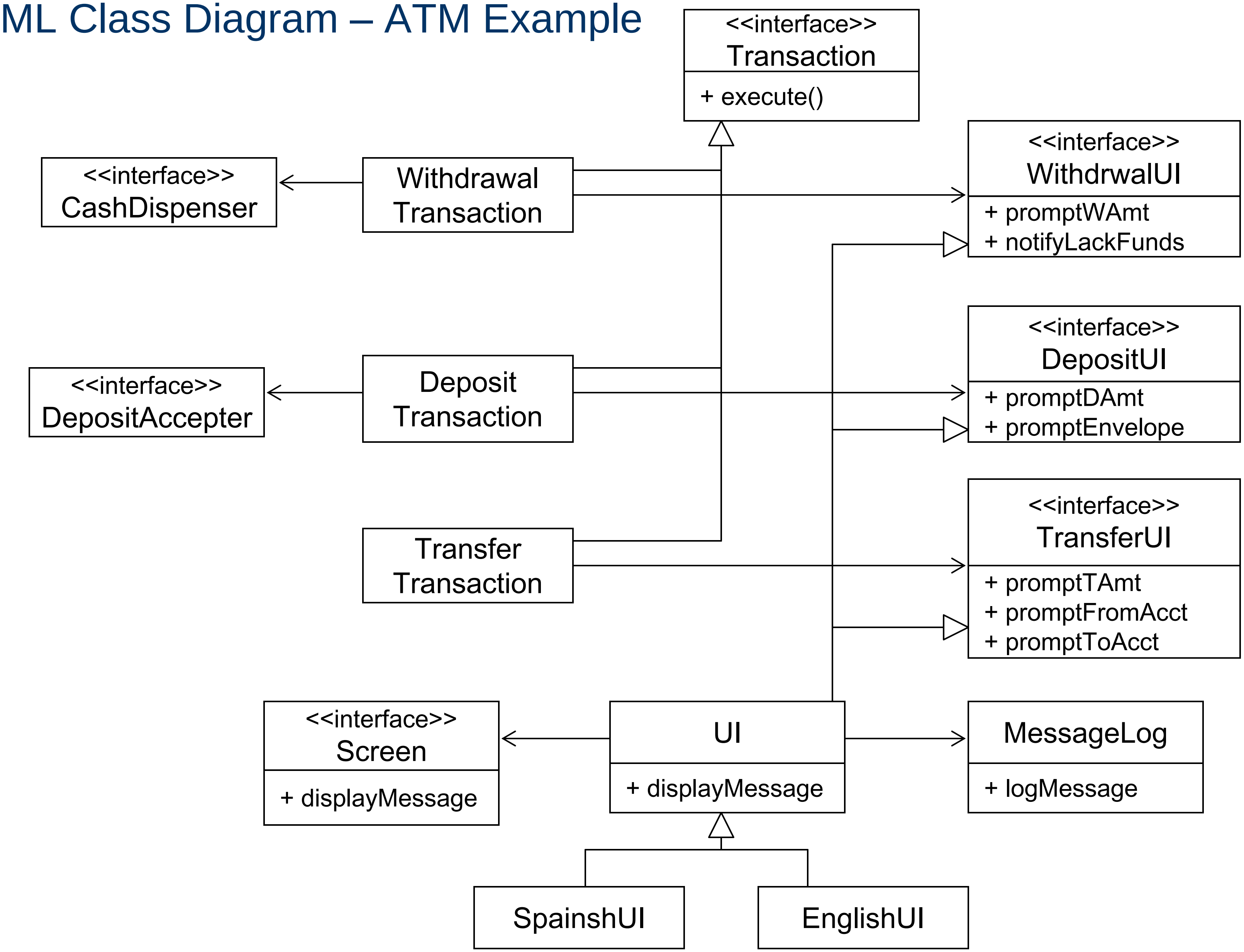
Class Diagrams – Inheritance (cont.)

- In UML, all arrowheads point in the direction of source code dependency. In the inheritance diagram on the previous slide, it is the `Phone` class than mentions the name of the `CommunicationDevice`, so the arrowhead points at `Communication Device`. Thus, in UML, inheritance arrows point at the base class.

- One technique that is fairly commonly employed to help with this potential misunderstanding is to model associations horizontally in the UML diagram and inheritance vertically.

- UML has a special notation for the kind of inheritance used between a Java class and a Java interface. Its shown as a dashed inheritance arrow.
  – Actually, it is more common to use the lollipop notation which I used in the UML diagram on page 37. Interfaces are drawn as little lollipops on the classes that implement them.

UML Class Diagram – ATM Example



Class Diagrams – Explanation of the Example

- You should note several things in the UML example on the previous page.

1. Notice the convention of horizontal association and vertical inheritance. This really helps to differentiate these vastly different types of relationships. Without this convention, it can be hard to understand the meaning out of the tangle of lines and icons.

2. Notice how the diagram is divided into three distinct zones. The transactions and their actions are on the left side of the diagram, the various interfaces are all on the right, and the user interface (UI) implementation is on the bottom.

3. See how the connections between the grouping are minimal and regular. In one case it is three associations, all pointing the same way. In the other case it is three inheritance relationships all merged into a single line. The grouping combined with the way they are connected help the reader to see the diagram in coherent pieces.

4. You should be able to “see” the code as you look at this diagram.

Class Diagrams – Explanation of the Example (cont.)

```
public class UI implements WithdrawalUI, DepositUI, TransferUI
{    private Screen itsScreen;
     private MessageLog itsMessageLog;

     public void displayMessage (String message)
     {     itsMessageLog.logMessage(message);
           itsScreen.displayMessage(message);
     }
}
```

- Does the code above look like what you would expect to see based on the UML diagram?
- Notice that all of the interfaces are clearly marked in the diagram. This makes it very clear to the reader which classes are intended to be interfaces and which are intended to be implemented.
  - For example, the diagram immediately tells you that `WithDrawalTransaction` talks to a `CashDispenser` interface. Obviously, some class in the system will need to implement the `CashDispenser`, but in this diagram we don't care which class.

Class Diagrams – Explanation of the Example (cont.)

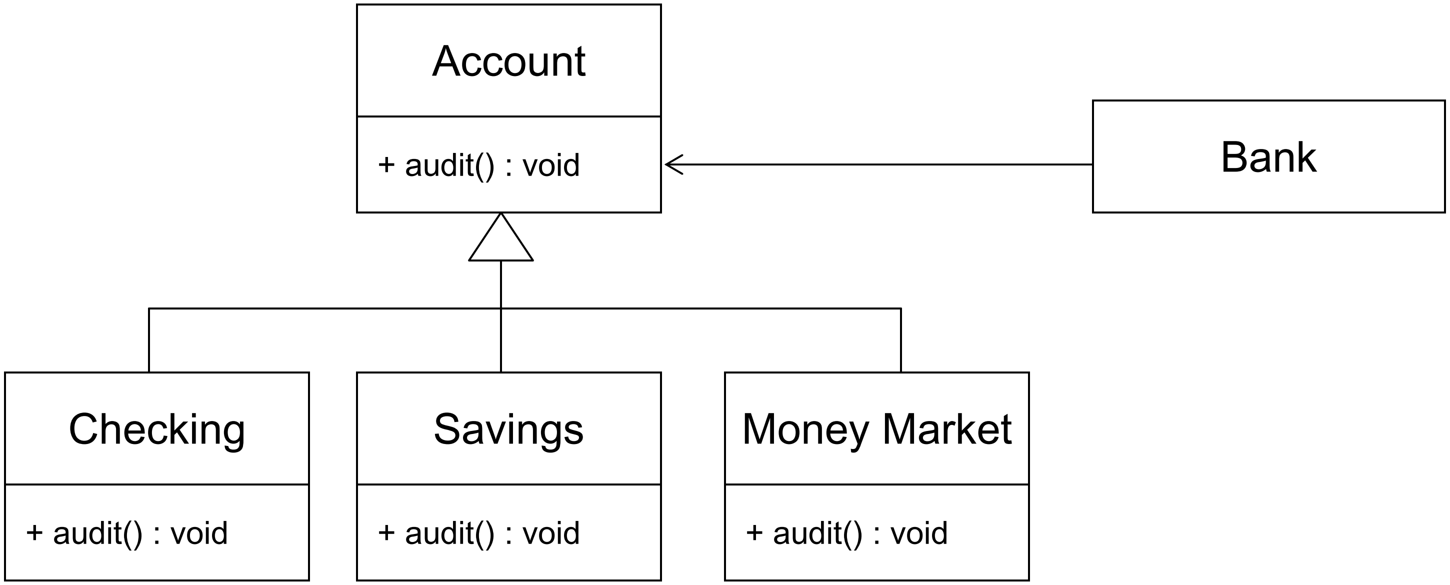
- Also notice that I haven't been particularly thorough in documenting the methods of the various UI interfaces.
  - For example, it would seem obvious that the `WithdrawalUI` will need more than just the two methods of `PromptWAmt` and `NotifyLackFunds`. For example, what about methods such as `promptForAcct` or `informCashDispenserEmpty`?
  - Putting these methods in the diagram would simply clutter up the picture. All we have included in the diagram is a representative batch of methods. These give the reader the idea of what is going on inside of the class and that is all that is really necessary at this level.

Class Diagram – Practice Problem

- Below is a description of a system. Construct a UML class diagram that incorporates the definition given.
  - A bank offers customers three types of accounts: checking, savings, and money market. Checking accounts pay 3% interest. Savings accounts must have a minimum balance of $150 and pay 5% interest. Money market accounts must have a minimum balance of $1000 and pay 8% interest. The system is to provide a report that will list all of the accounts and the interest accrued during the preceding 12 months. The system will also provide an auditor that will print all of the accounts whose balance has fallen below the required minimum for that type of account.
    - A solution appears on the next page. Try it yourself first!

Class Diagram for Bank Problem

- Shown below is the UML class diagram for the bank problem.



Further Analysis Of The Bank Design

- `Checking`, `Savings`, and `MoneyMarket` all inherited from `Account`. Each of these three classes has an "is-a" relationship with `Account`. In other words, `Checking` is-a `Account`, and so on.
- The superclass Account will provide the attributes, account number, name, current balance, etc. and behaviors (the accessor and mutator methods) common to all accounts.
- The specialized rules regarding minimum balances and the computation of interest will be delegated to the subclasses which define each account type.
- This means that the Bank class will contain Accounts and not worry about the specific types of accounts it contains.

Further Analysis Of The Bank Design (cont.)

- Consider for a moment how the design of the banking system would change it you were asked to add a no interest checking account to the system.
  - Using inheritance, this is an extremely easy task. You would need only to create a new subclass that captures the details of the new account type. All of the standard behaviors would be inherited directly from the `Account` superclass. Since the new account type is an `Account`, the `Bank` class will already know how to work with the new class. Thus, the only modification to the existing code would be the addition of the new class. In a procedural language, the changes would be much more substantial.
- The last design feature that we need to consider is the "auditor" feature.

Further Analysis Of The Bank Design (cont.)

- Did you create an `Auditor` class that would go through all the accounts stored in a Bank and check to see if they satisfy the minimum balance requirements?
  - If you did so, this is not a good design decision. The problem with this is that you would be creating a class which consists of a single method whose primary purpose is to provide functionality.
  - A better way to deal with the auditing feature is to provide an audit method in the `Bank`, `Account`, `Checking`, `Savings`, and `MoneyMarket` classes. The `audit()` method would step through the accounts in manages invoking, one by one, the specialized audit methods for each type of account.
    - The `audit()` method in the `Checking`, `Savings`, and `MoneyMarket` classes would verify that the minimum requirements have been met for this account type and, if not, print an appropriate message.

Further Analysis Of The Bank Design (cont.)

- Note that the end result of either design, that of a separate auditor class and that of auditor methods inside each of the class shown on page 49, provides the same functionality.

- However, the second design is much easier to extend. Consider again, the question of what you would have to do to add a new type of account.
  - For the second design, all you have to do is add a new subclass to the system that describes the new account and to make sure that that new class includes its own specialized audit method.
  - In the first design, you must not only add a new subclass, but you must also remember to modify the `Auditor` class to handle this new account type.

Further Analysis Of The Bank Design (cont.)

- One of the marks of a good design is that when a change is made to the specification of the problem, the change to the design is localized to a single class rather than propagating through many classes.
- When you find yourself saying things like, “We must add a new subclass A, and also modify classes B, C, and D to deal with this new subclass,” you greatly increase the likelihood for error and complicate the maintenance process.

Overview of UML

- UML defines nine different graphical diagrams. The choice of which diagrams one creates can influence how a problem is encountered and how a corresponding solution is shaped. As I've mentioned before, we will examine only the most useful components of UML that will apply to the widest range of problems. Shown below is the UML hierarchy.

1. Class diagram (static)
2. Use-case diagram
3. Behavior diagram (dynamic):
    - 3.1. Interaction diagram:
        - 3.1.1. Sequence diagram
        - 3.1.2. Collaboration diagram
    - 3.2. Statechart diagram
    - 3.3. Activity diagram
4. Implementation diagram:
    - 4.1. Component diagram
    - 4.2. Deployment diagram

Use-Case Diagrams

- The use-case diagram is a concept that was first introduced in what is now an obsolete modeling tool called object-oriented software engineering (OOSE).

- The functionality of a system is described in a number of different use cases, each of which represents a specific flow of events in the system.

- A use case corresponds to a sequence of transactions, in which each transaction in invoked from outside the system (actors) and engages internal objects to interact with one another and with the system’s surroundings.

- The description of a use case defines what happens in the system when the use case is performed.

Use-Case Diagrams (cont.)

- In essence, the use-case model defines the outside (actors) and inside (use case) of the system's behavior.

- Use cases represent specific flows of events in the system.

- The use cases are initiated by actors and describe the flow of events that these actors set off. An actor is anything that interacts with a use case: It could be a human user, external hardware, or another system.
  - An actor represents a category of user rather than a physical user. Several physical users can play the same role. For example, in terms of a Member actor, many people can be members of a library, which can be represented by one actor called Member.

Use-Case Diagrams (cont.)

- A use-case diagram is a graph of actors, a set of cases enclosed by a system boundary, communication (participation) associations between the actors and the use cases, and generalization among the use cases.
- The importance of use-case diagrams, has in large part, been exaggerated and overcomplicated.
- The main thing about use-case diagrams is to keep them simple. It is this simplicity that will make the diagram most useful. If you once proceed down the dark path of use case complexity, forever will it dominate your destiny. Use the force, and keep your use cases simple.

Use-Case Diagrams (cont.)

- As an example of a use-case diagram. consider a typical help desk.

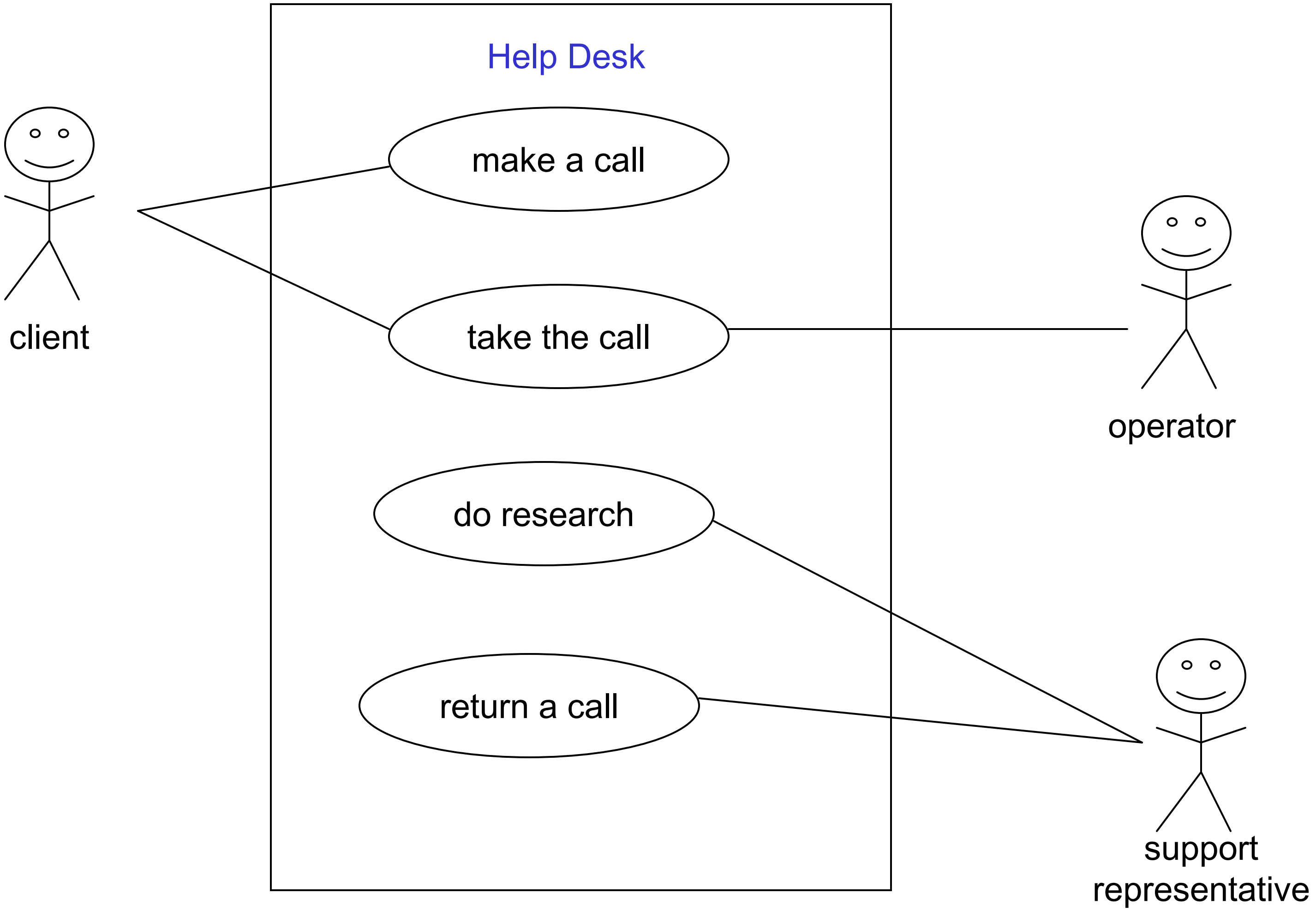
- The use-case diagram on the page 13 illustrates the relationship among the actors and the use cases within the system.

  - A client makes a call that is taken by an operator., who determines the nature of the problem.

  - Some calls can be answered immediately; other calls require research and a return call.

- A use case is shown as an ellipse containing the name of the use case. The name of the use case can be placed below or inside the ellipse.

Use Case Diagram - Example



Use-Case Diagrams (cont.)

- The following relationships are shown in a use-case diagram:
  - Communication: The communication relationship of an actor in a use case is shown by connecting the actor symbol to the use-case symbol with a solid line. The actor is said to communicate with the use case.
  - Uses: A uses relationship between use cases is shown by a generalization arrow from the use case. This is the same as in class hierarchies.
  - Extends: The extends relationship is used when you have one use case that is similar to another use case but does a but more, again similar to a subclass.

Dynamic Diagrams

- Events happen dynamically in all systems: objects are created and destroyed, objects send messages to one another in an orderly fashion, and in some systems, external events trigger operations on certain objects.
- Furthermore, objects have states. The state of an object would be difficult to capture in a static model.
- The state of an object is the result of its behavior.
  - Consider a conventional phone. When a telephone is first installed, it is in an idle state, meaning that not previous behavior is of great interest and that the phone is ready to initiate and receive calls. When someone picks up the handset, we say that the phone is "off the hook" and in the dialing state; in this state we do not expect the phone to ring: we expect to be able to initiate a conversion with someone on another phone. When the phone is "on the hook", if it rings and we pick up the handset, the phone is now in a receiving state, and we expect to be able to converse with the person that initiated the conversation.

Dynamic Diagrams (cont.)

- Each class may have an associated activity diagram that indicates the behavior of a class instance (an object). In conjunction with the use-case diagram, we can provide a script or interaction diagram to show the time or event ordering of messages as they are evaluated.

- Interaction diagrams are diagrams that describe how groups of objects collaborate to get the job done.

- Interaction diagrams capture the behavior of a single use case, showing the pattern of interaction among objects. The diagram will show a number of example objects and the messages passed between those objects within the use case.

- There are two types of interaction diagrams in UML: sequence diagrams and collaboration diagrams.

Sequence Diagrams

- Sequence diagrams are the most common of the dynamic models drawn by UML users. As with other types of diagrams, UML provides lots and lots of notational syntax to help you draw truly incomprehensible diagrams.
- We'll restrain ourselves to look only at the most useful parts of this notation and not worry about too many details.
- Sequence diagrams are a way of describing the behavior of a system by viewing the interaction between the system and its environment.
- A sequence diagram shows an interaction arranged in a time sequence. It shows the objects participating in the interaction by their lifelines and the messages they exchange, arranged in a time sequence.

Sequence Diagrams (cont.)

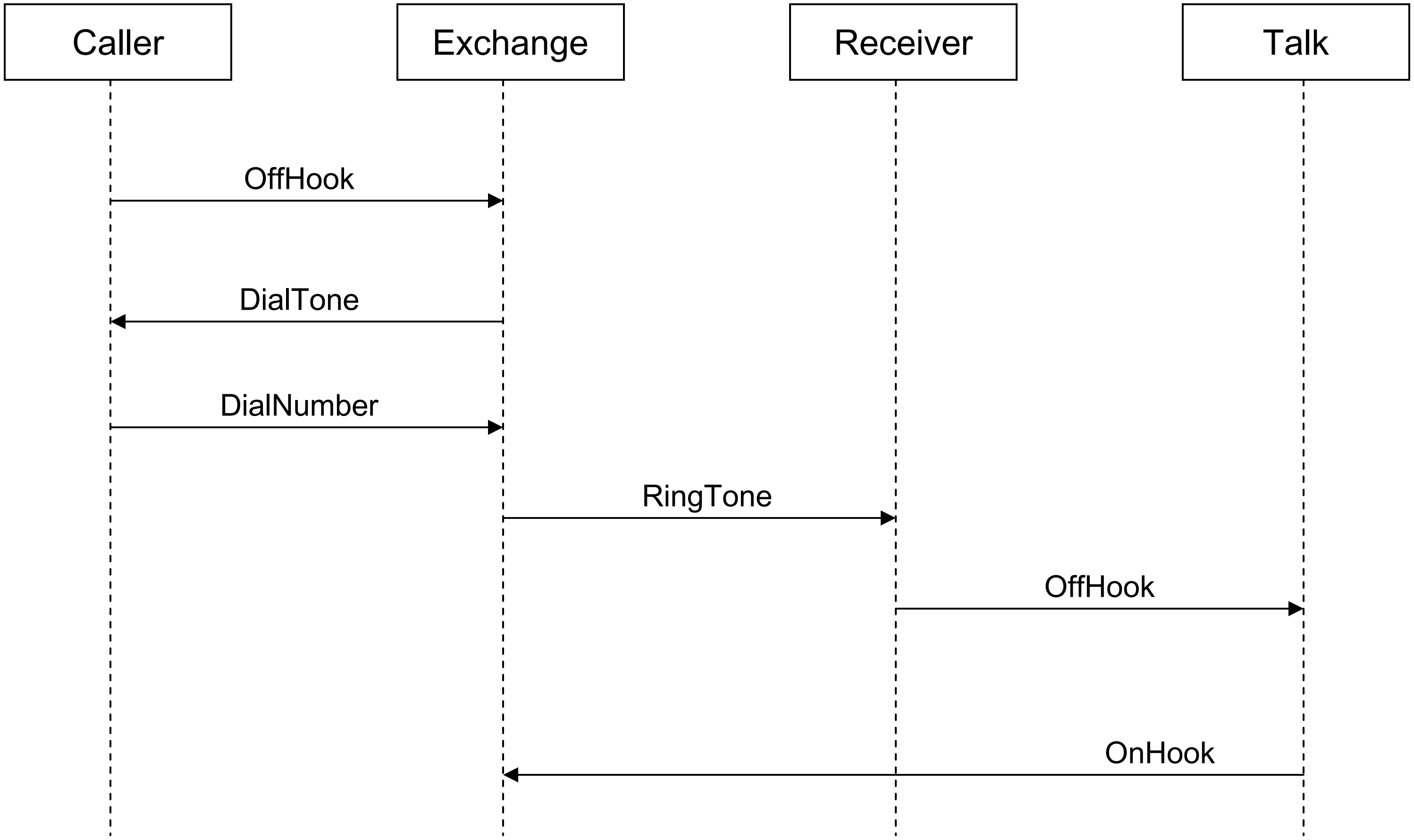
- A sequence diagram has two dimensions: the vertical dimension represents time, the horizontal dimension represents different objects.
- The vertical line is called the object's lifeline. The lifeline represents the object's existence during the interaction.
- An object is shown as a rectangle at the top of a dashed vertical line (the lifeline).
- A role is a slot for an object within a collaboration that describes the type of object that may play the role and its relationships to other roles.
- Note: A sequence diagram does not show the relationships among the roles or the association amount the objects.

Sequence Diagrams (cont.)

- Each message is represented by an arrow between the lifelines of two objects.
- The order in which these messages occur is shown top to bottom on the diagram.
- Each message is labeled with the message name. The label can also include the argument and some control information and show self-delegation (a message that an object sends to itself) by sending the message arrow back to the same lifeline.
- The horizontal ordering of the lifelines is arbitrary. Often, call arrows are arranges to proceed in one direction across the diagram, but this is not always possible and the order conveys not information.
- A sequence diagram is a good way to understand the overall flow of control in a program.

Sequence Diagram – Example (see page 15)

Telephone Call

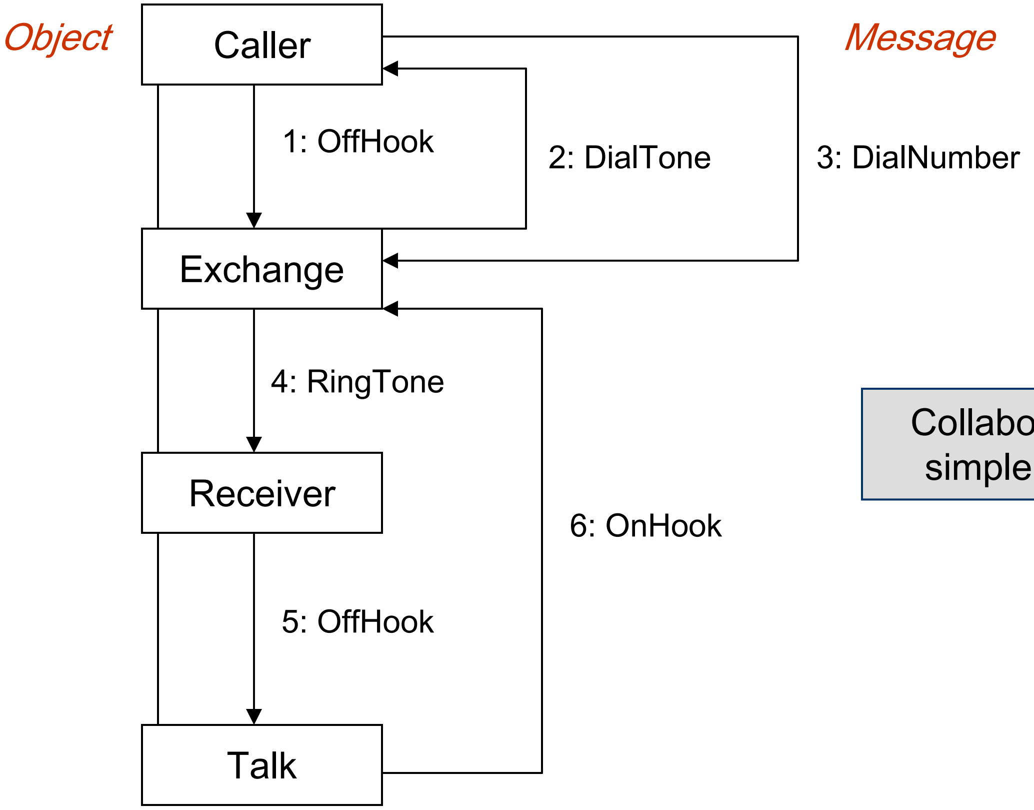


Collaboration Diagrams

- Collaboration diagrams represent a collaboration, which is a set of objects related in a particular context, an interaction, which is a set of messages exchanged among the objects within the collaboration to achieve a desired outcome.
- In a collaboration diagram, objects are shown as figures. As with sequence diagrams, arrows indicate the message sent within the given use case.
- In a collaboration diagram, the sequence is indicated by numbering the messages.
  - Some people argue that numbering the messages makes it more difficult to see the sequence than does a sequence diagram. However, since the collaboration diagram is more compressed, other things can be shown more easily. For example, how the objects are linked together.
- Alternate numbering schemes are possible with collaboration diagrams. The following two pages give two alternative schemes for the phone example.

Collaboration Diagram – Example (see page 15)

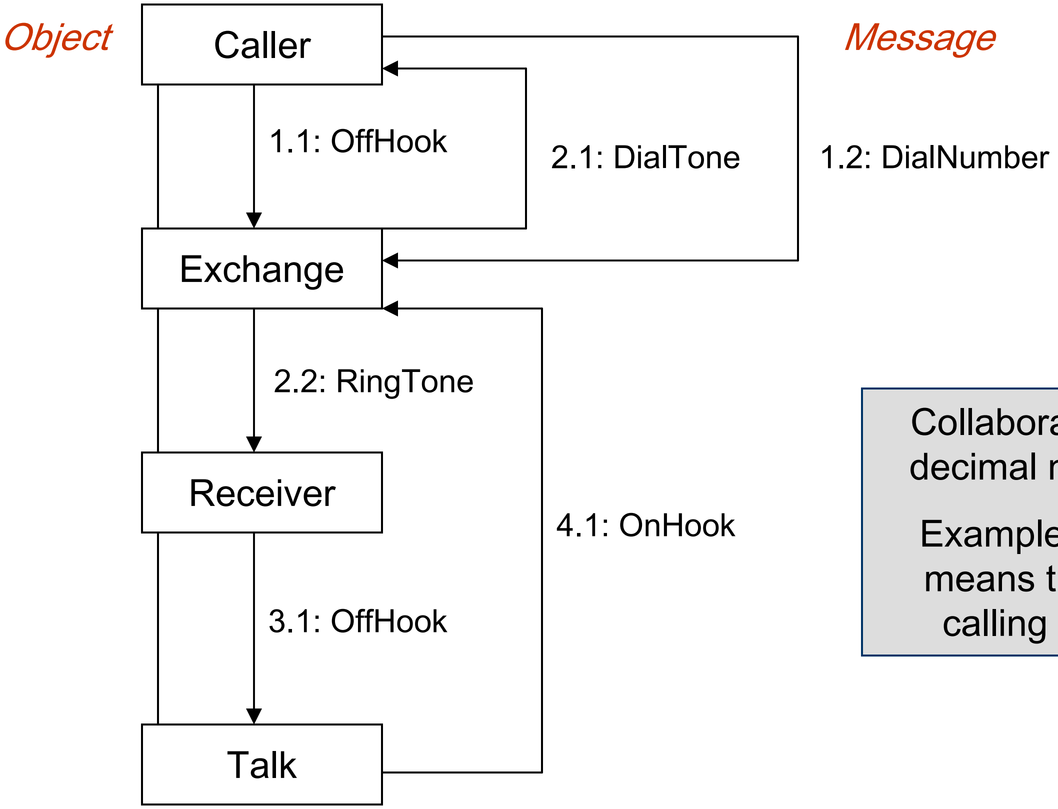
Telephone Call



Collaboration diagram using simple numbering scheme

Collaboration Diagram – Example (see page 15)

Telephone Call



Collaboration diagram using decimal numbering scheme.

Example: 1.2: DialNumber means that the Caller(1) is calling the Exchange(2).