Design of Object Systems

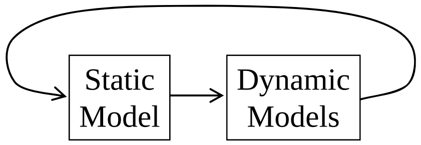
C++ Object Oriented Programming
Pei-yih Ting
NTOUCS

Introduction

- **Static model**
  - UML Tutorial: Part 1 – Class Diagrams, Robert C. Martin
    http://faculty.ksu.edu.sa/amani.h/Documents/UMLTutoria(To%20benefit).pdf
- **Dynamic model**
  - UML Tutorial: Collaboration Diagrams, Robert C. Martin
    http://www.objectmentor.com/resources/articles/umlCollaborationDiagrams.pdf
  - UML Tutorial: Sequence Diagrams, Robert C. Martin
    http://www.cs.umd.edu/~mvz/cmsc435-s09/pdf/cell-phone-sequence-chart.pdf
- **The interplay between static and dynamic models:**
  - Novice OO designers often **over-emphasize on static models –** classes, properties, interfaces, inheritance/aggregation relationships
  - **Software design is about behavior, behavior is dynamic**
  - **Object oriented design is a technique used to separate and encapsulate behaviors.**

Introduction (cont'd)

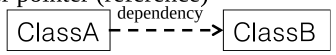
- A **static** model can**not** be proven accurate without associated dynamic models.
- **Dynamic** models, on the other hand, do not adequately present considerations of *structure* and *dependency management*.
- Require ***Quick iterations*** between static and dynamic models until they converge to an acceptable solution.



Dependency Management

- **Dependency** between ClassA and ClassB: a change in the interface of ClassB necessitates changes in the implementation of ClassA
  - ClassA has a ClassB member object or member pointer (reference)
  - ClassA is derived from ClassB
  - ClassA has a function that takes a parameter of type ClassB
  - ClassA has a function that uses a static member of ClassB
  - ClassA sends a message (a method call) to ClassB

  In each case, it is necessary to #include "classB.h" in classA.cpp.

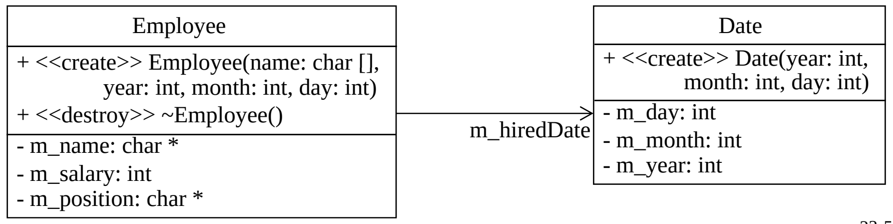


- Code reuse, an important goal, always produces dependencies.
- When designing classes and libraries it is important to make sure that we produce as few unnecessary or unintentional dependencies as possible because they slow down compilation and reduce reusability.
- Forward class declarations make it possible for classes to have ***circular relationships*** without having *circular dependencies* between header files.

UML Static Model

- **Class Diagram**
  - classes
    - attributes/properties
    - operations/interfaces/services
  - associations/relationships
    - inheritance
    - aggregation/composition

Not every part in the graph is required. It depends on what the designer intends to capture.


Employee
+ <<create>> Employee(name: char [], year: int, month: int, day: int)
+ <<destroy>> ~Employee()
- m_name: char *
- m_salary: int
- m_position: char *
m_hiredDate
Date
+ <<create>> Date(year: int, month: int, day: int)
- m_day: int
- m_month: int
- m_year: int

UML Dynamic Models

- **State Diagram**
  - Describe how a system responds to events in a manner that is dependent upon its state
- **Interaction Diagrams**
  - **Sequence diagrams**:
    - focus on the **order** in which the messages are sent
    - useful for describing the procedural flow through many objects
  - **Collaboration diagrams**:
    - focus on the **relationships** between the objects
    - useful for visualizing the way several objects collaborate to get a job done
    - useful for comparing a dynamic model with a static model

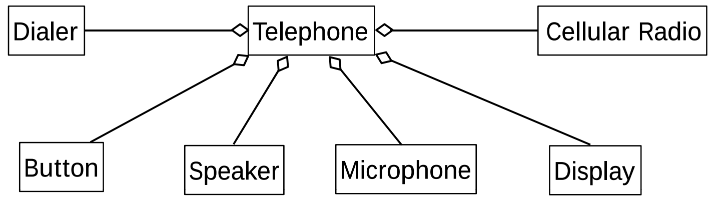
Note: Sequence and collaboration diagrams describe **the same** information and can be transformed into one another

Example: A Cellular Phone

- Consider the software that controls a very simple cellular phone.
- Specifications
  - Buttons: digits, send, accept, volume up/down, power, …
  - Dialer hardware/software: emits the appropriate tones for dialing
  - Cellular radio: RF connection to the cellular network
  - Microphone, speaker, display

- There is an intuitive **composition** relationship from the above spec.

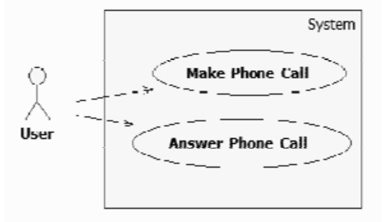
Static Model I



Is this good?? "Analogy to the real world" might **not** be sufficient.

Specifying Dynamics

- <u>**Use cases**</u>: Make Phone Call
  1. User presses the digit buttons to enter the phone number.
  2. For each digit, the display is updated to append the digit to the phone number.
  3. For each digit, the dialer generates the corresponding tone and emits it from the speaker.
  4. User presses "Send".
  5. The "in use" indicator is illuminated on the display.
  6. The cellular radio establishes a connection to the network.
  7. The accumulated digits are sent to the network.
  8. The connection is made to the called party.



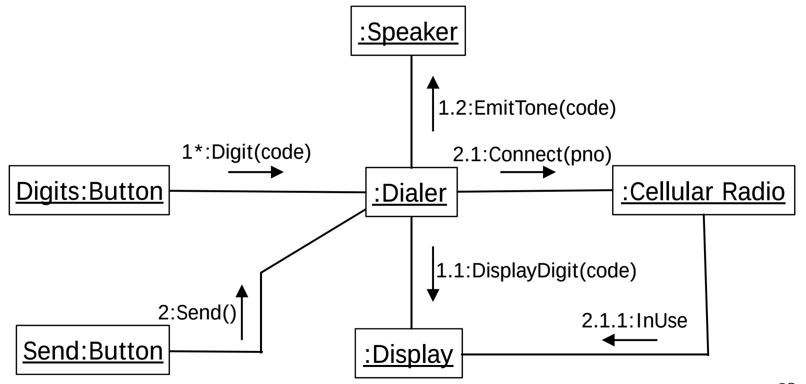
- How do the objects in the static model collaborate to execute this procedure?

Possible Dynamics

- When **digit** buttons are pressed:
  - Digit button object sends a *digit* message to ~~Telephone object.~~
  - ~~Telephone object forwards the *digit* message to~~ Dialer object.
  - Dialer object sends a *displayDigit* message to Display object to show the new digit.
  - Dialer object sends an *emitTone* message to Speaker object.
- When **send** button is pressed:
  - Send button object sends a *send* message to ~~Telephone object.~~
  - ~~Telephone object forwards the *send* message to~~ Dialer object.
  - Dialer object sends *connect* message to CellularRadio object.
  - CellularRadio object sends *inUse* message to Display object to illuminate the "in use" indicator on the display.
- Problem: Is the "Telephone object" necessary?

Collaboration Diagram

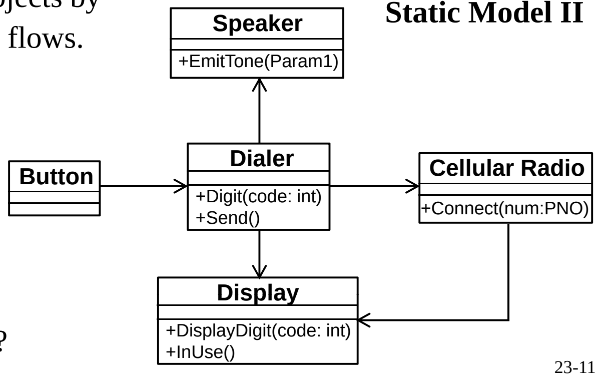
- Collaboration Diagram of the "Make Phone Call" use case
  1. **Objects**: instances of classes
  2. **Links**: instances of associations
  3. **Messages** (names, nested sequence numbers, arguments)



Reconciling the Static Model

- **Problem**: The structure of objects in the collaboration diagram does not look very much like the structure of the previous class diagram.
- Which one needs to be modified? **dynamic** or **static**
- The "Telephone" class in the previous intuitive static model is like a "god" controlling all objects by monitoring all message flows. This results in a **highly coupled design**.
- Why not change the static model to a "decentralized one" consistent with the collaboration diagram?

Static Model II

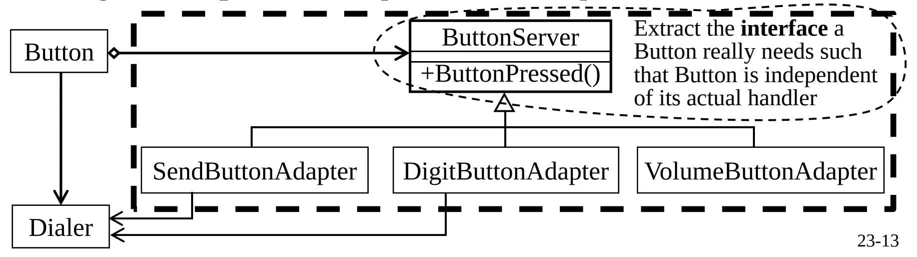


Static Model II (cont'd)

- You might feel **uncomfortable** because static model II does not seem to reflect the real world as well as the "intuitive" static model I.
  - Static model I is based upon the physical structure of the telephone.
  - Static model II is based upon the **real world behaviors** of the telephone instead of its real world physical makeup. (Again, software models the behaviors.)
- Many dynamic models usually accompany a single static model.
  - Each dynamic model explores a different variation of a use case / scenario / requirement.
  - The links between the objects in those dynamic models imply a set of associations that must be present in a static model.

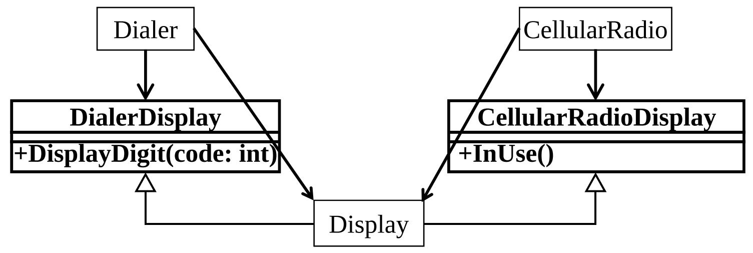
Static Model III

- ✧ **Problem 1**: Why should a class name **Button** know anything about a class named **Dialer**?
  - ✶ Does every button of this phone need to be related to the dialing function? **How about volume up/down?**
  - ✶ Shouldn't the Button class be reusable in a program that does not have any thing to do with Dialer?
  - ✶ Dependency: in the current design, when the interface of the Dialer class changes, the class Button needs to be recompiled.
- ✧ Using the **Adapted Server** pattern to decouple Button from Dialer


Button
ButtonServer
+ButtonPressed()
Extract the **interface** a Button really needs such that Button is independent of its actual handler
SendButtonAdapter
DigitButtonAdapter
VolumeButtonAdapter
Dialer

Static Model III (cont'd)

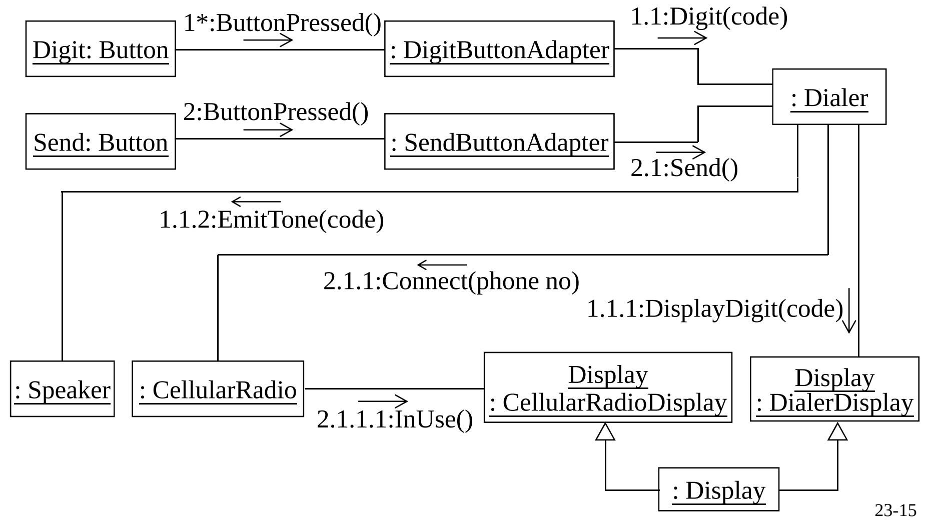
- ✧ **Problem 2**: High coupling of classes **Dialer** and **CellularRadio** through the class **Display**!
  - ✶ If the interface of Display changes in order to satisfy the requirement of Dialer, the CellularRadio will be affected (class CellularRadio depends on class Dialer); at very least, by an unwarranted recompile.
- ✧ **Interface Segregation** of the class Display



consistent with the **Single Responsibility Principle** (SRP)

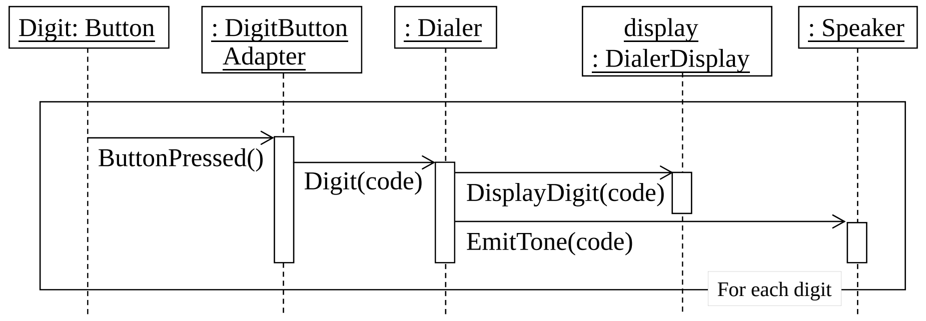
Collaboration Diagram II

- ✧ The change of static model will certainly change the dynamic model.



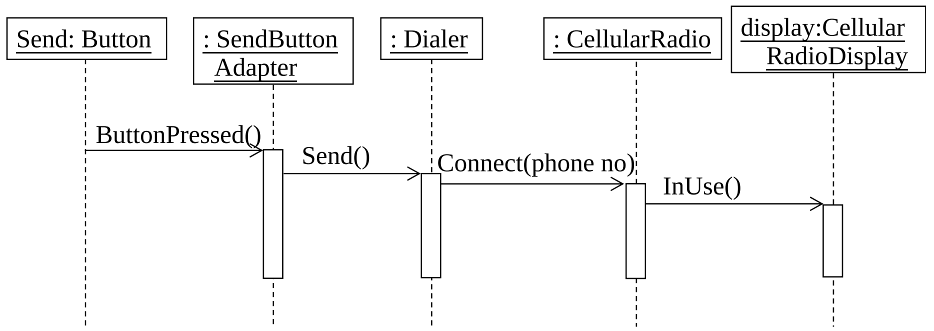
Sequence Diagram: Dialing

- ✧ Both collaboration diagram and sequence diagram specify the **dynamics** of the system: **sequence of messages sent between objects**.
  - ✶ **Collaboration diagram** emphasizes the **relationships between the objects**
  - ✶ **Sequence diagram** emphasizes the **sequence of the messages**



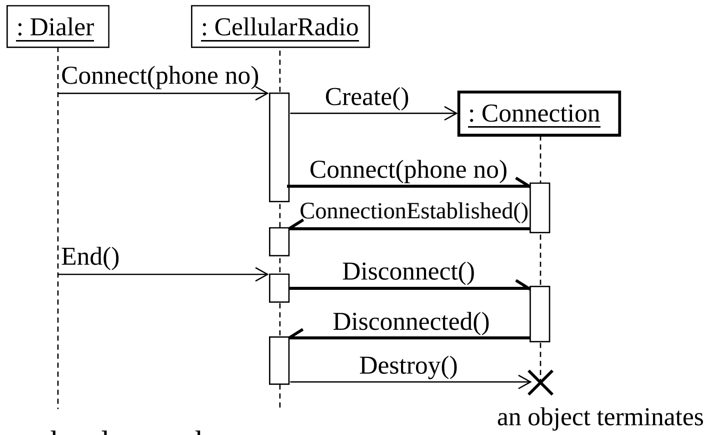
- ✧ **lifeline** ✧ **message** ✧ **iteration/looping condition**
- ✧ **activation**: the duration of the execution of a method in response to a message; a method returns to the caller at the end of the activation

Sequence Diagram (cont’d)



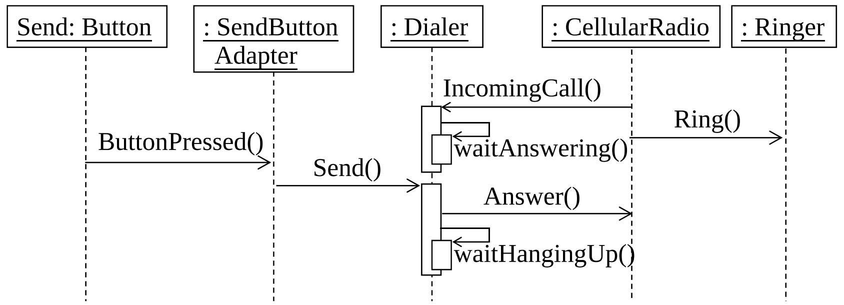
- Sequence diagram is easier to follow algorithmically.
- Usually use separate sequence diagram for each use case.
- Collaboration diagram shows the whole collaboration of objects in a single dense diagram but somewhat obscures the algorithm.

Creation and Deletion of Objects



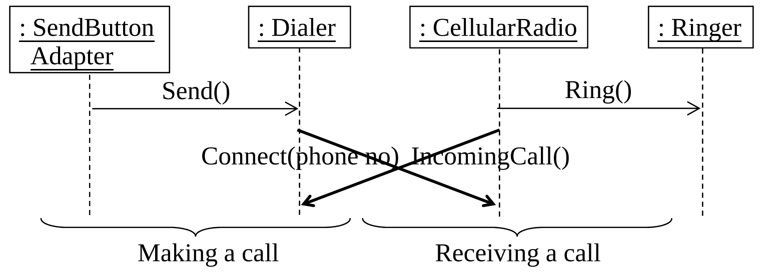
- half-arrowhead: asynchronous messages
- An asynchronous message is a message that returns immediately while the receiving object receives the message and activates in a different thread

Sequence Diagram: Answering



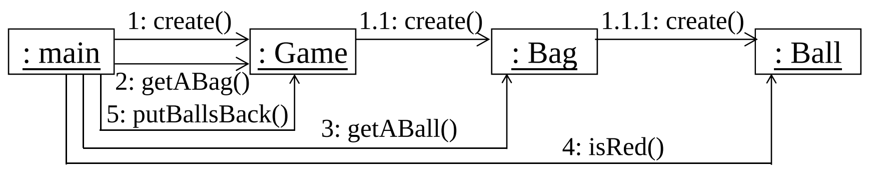
- : Dialer enters waitAnswering state after receiving IncomingCall() message. In this state, arriving Send() message denotes that user wants to answer the incoming call instead of making an outgoing call and the : Dialer enters waitHangingUp state instead of waitDialing state.
- Most activation rectangles have been omitted for clarity, only show the activation rectangles for : Dialer.

Race Condition Depicted

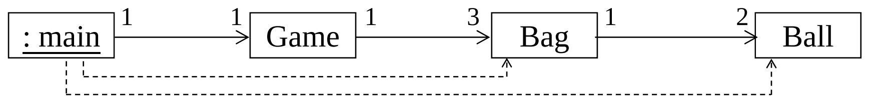


- “Making a call” is initiated by the user, while “Receiving a call” is initiated independently by another user.
- Message with a downward angle shows the elapsed time between the sending of the message and its reception.
- The **crossing of messages** indicates the race condition, which should be handled carefully by both : Dialer and : CellularRadio objects with state diagrams.

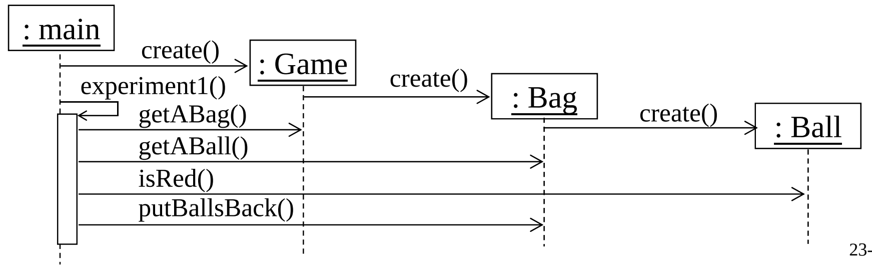
Three Bags Example



: main
: Game
: Bag
: Ball
1: create()
1.1: create()
1.1.1: create()
2: getABag()
5: putBallsBack()
3: getABall()
4: isRed()



: main
Game
Bag
Ball
1
1
1
3
1
2



: main
: Game
: Bag
: Ball
create()
create()
create()
experiment1()
getABag()
getABall()
isRed()
putBallsBack()