

**Software Design**  
**CSE 335, Spring 2006**

**Object-oriented programming: Role-based design**

Stephen Wagner

Michigan State University

# Button Example

- Reusable classes with dynamic behavior
- How to design button without knowing class of objects that are to receive `buttonPressed` messages
- Solution
  - Invented an interface class that declares nothing but the `buttonPressed` operation
  - Designed class `button` with respect to this interface
  - Receiver *classes* to implement the interface
  - Receiver *objects* to *register* with the `Button` object

## Qualities of our Design

- class `Button` is very reusable
- We can understand the `Button-ButtonListener` *collaboration* with little knowledge of `Button` and no knowledge of `DocManager`.
- Clear mechanism for adapting arbitrary class to implement the `ButtonListener` interface.
- Programs structured differently than programs that “compute a function”.

# Separation of Concerns

- De-couple interdependent parts of a problem

# Separation of Concerns

- De-couple interdependent parts of a problem
  - Only way to manage complexity

# Separation of Concerns

- De-couple interdependent parts of a problem
  - Only way to manage complexity
- Pull different parts of the problem apart

# Separation of Concerns

- De-couple interdependent parts of a problem
  - Only way to manage complexity
- Pull different parts of the problem apart
  - Simple concept, but hard to do

# Separation of Concerns

- De-couple interdependent parts of a problem
  - Only way to manage complexity
- Pull different parts of the problem apart
  - Simple concept, but hard to do
  - Divide and Conquer is a SoC strategy

# Abstraction

- **Defn:** Process by which we identify the important aspects of the phenomenon and ignore its details
- Example
  - STL class `vector` versus *bounded sequence*
  - The latter is an abstraction
- Two popular abstractions over objects
  - Class: set of objects with same characteristics
  - Role/collaboration: set of objects that collaborate by sending messages back and forth to achieve some goal or purpose

# Collaboration

- **Defn:** Cohesive pattern of interaction (i.e. message exchange) among multiple objects to achieve some goal or purpose

# Collaboration

- **Defn:** Cohesive pattern of interaction (i.e. message exchange) among multiple objects to achieve some goal or purpose
- Examples
  - Message exchange between `printButton` and `fileMgr`
  - `printButton` sends `buttonPressed` message to `fileMgr`

# Collaboration

- **Defn:** Cohesive pattern of interaction (i.e. message exchange) among multiple objects to achieve some goal or purpose
- Examples
  - Message exchange between `printButton` and `fileMgr`
  - `printButton` sends `buttonPressed` message to `fileMgr`
- Refers to a *set of objects*, not classes

# Roles

- **Defn:** Abstraction of an object in a particular collaboration

# Roles

- **Defn:** Abstraction of an object in a particular collaboration
  - Defines only that subset of the object's characteristics that are meaningful to the collaboration

# Roles

- **Defn:** Abstraction of an object in a particular collaboration
  - Defines only that subset of the object's characteristics that are meaningful to the collaboration
  - Each object may play many different roles

# Roles

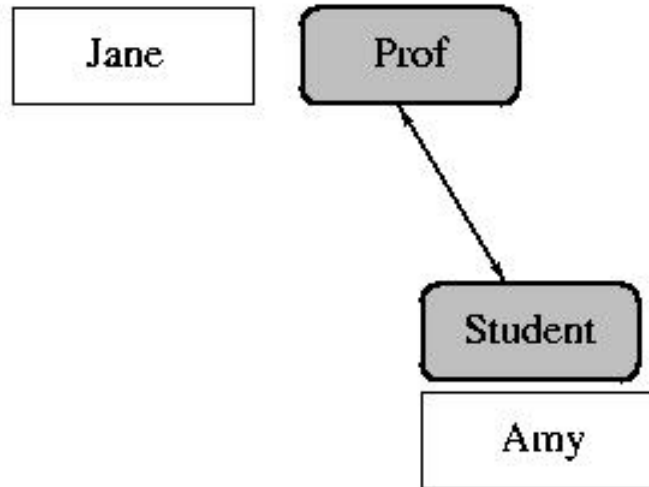
- **Defn:** Abstraction of an object in a particular collaboration
  - Defines only that subset of the object's characteristics that are meaningful to the collaboration
  - Each object may play many different roles
  - A given role may be played by different types of objects

# Roles

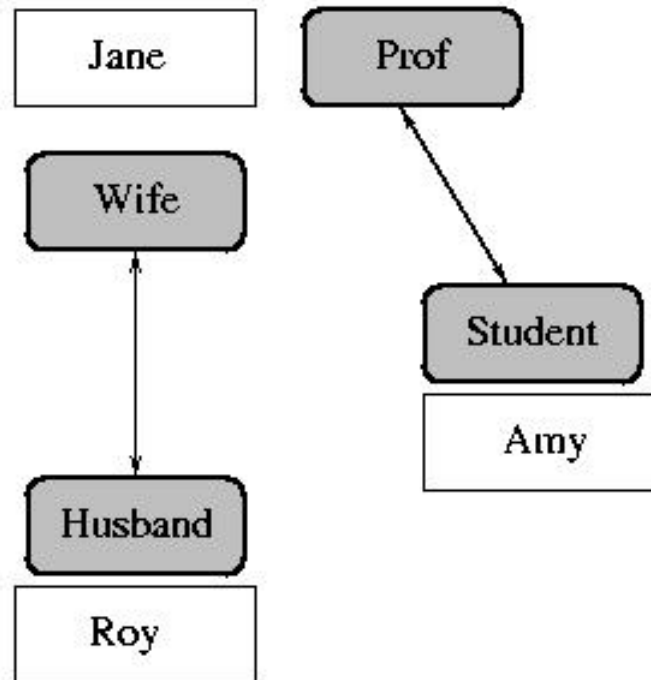
- **Defn:** Abstraction of an object in a particular collaboration
  - Defines only that subset of the object's characteristics that are meaningful to the collaboration
  - Each object may play many different roles
  - A given role may be played by different types of objects
- Example: `fileMgr` object might:
  - play the *ButtonListener* role for the `printButton` object
  - play the *Model* role for the `viewPort` object
- Allows us to easily synthesize new collaborations



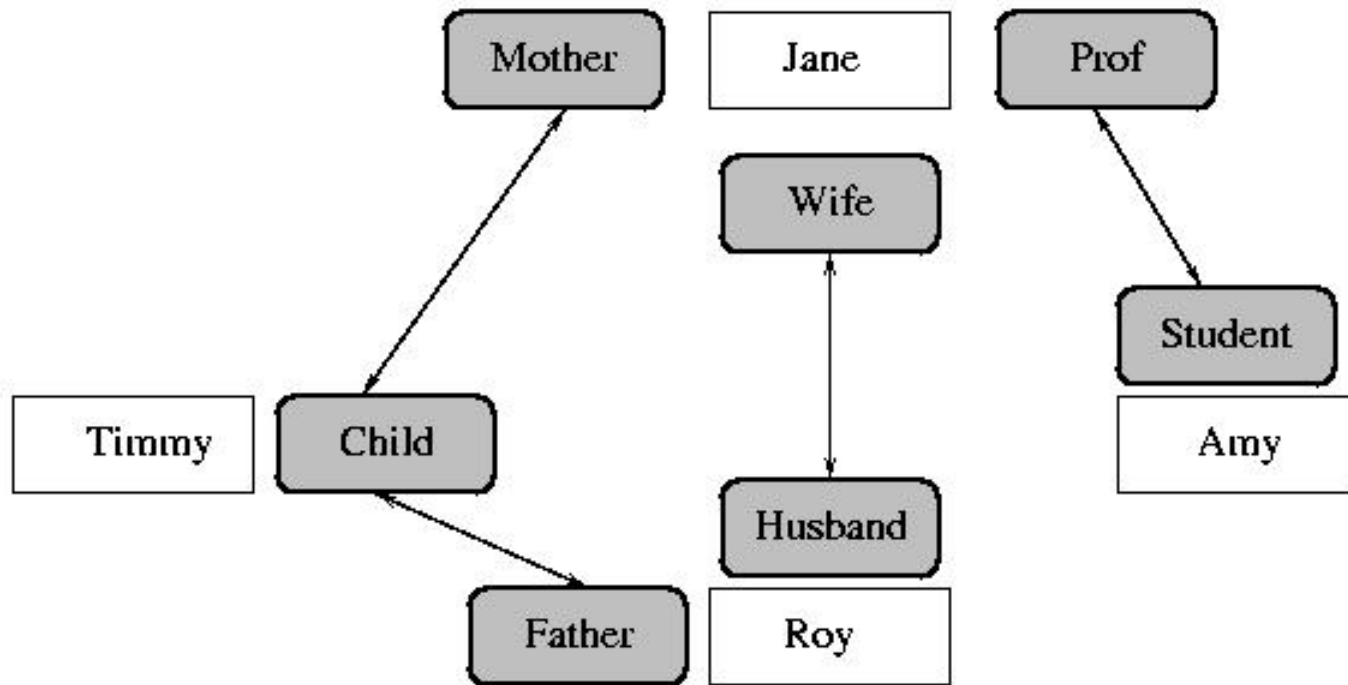
# Roles



# Roles



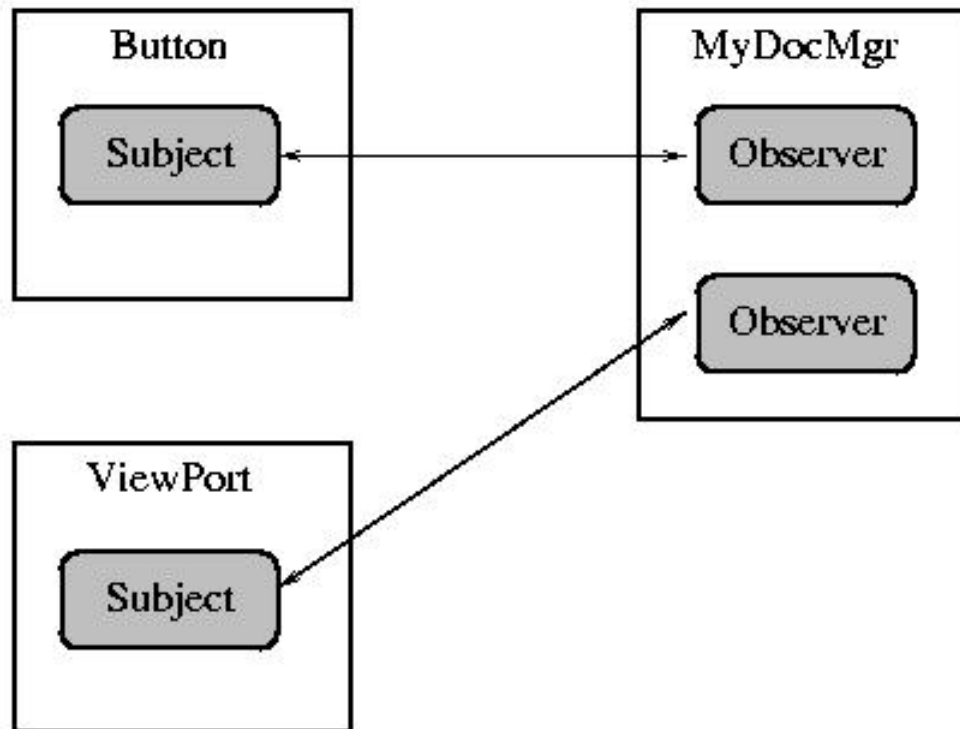
# Roles



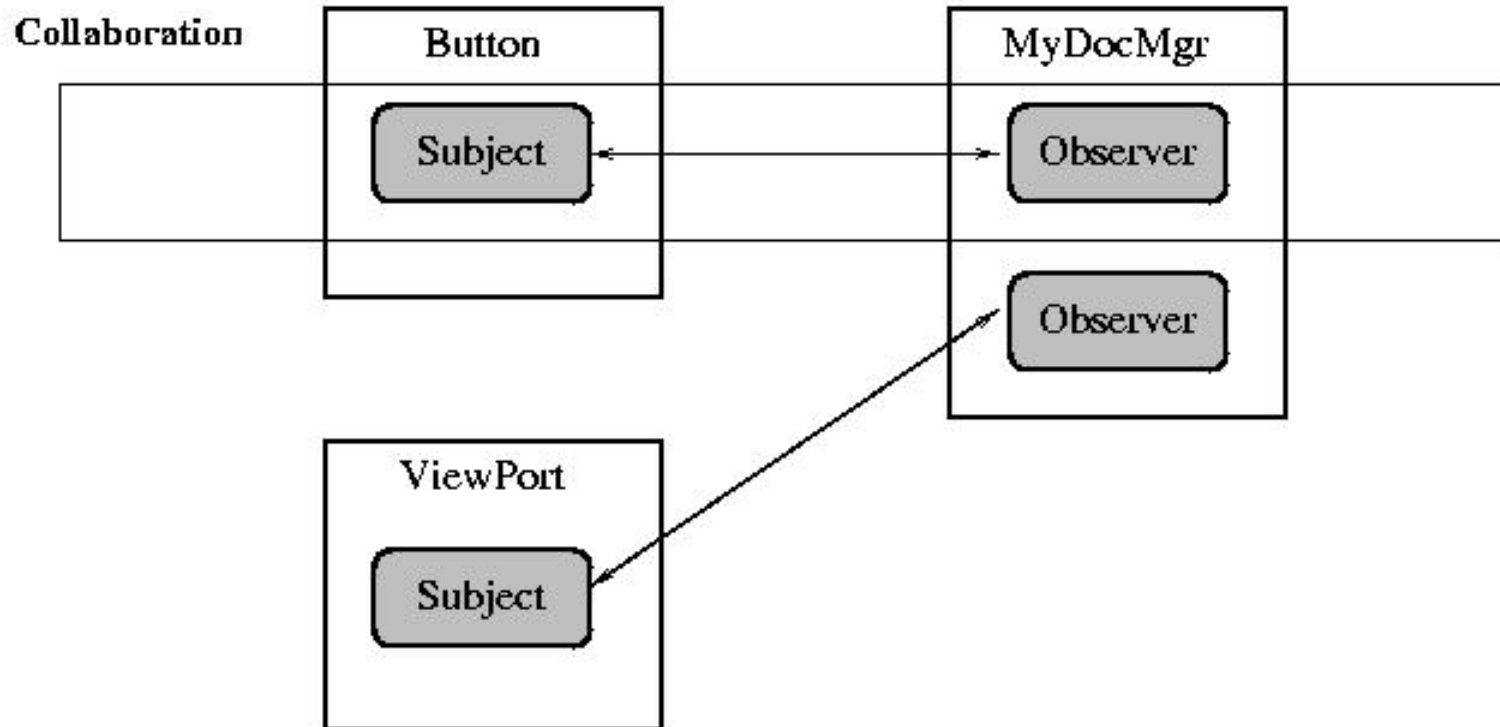
# Roles

- Jane, Amy, Roy, Timmy are all people
  - There are all instances of the *Person* class
- They each play different roles
  - The same person can play multiple roles
  - The same role can be played by multiple people

# Roles across Objects



# Roles across Objects



# Roles and Separations of Concern

- We can design classes independently
  - We do not need to predict all the different types of interactions an instance of the class may have
  - Focus on the specific task/goal of the class
  - Interactions with other objects will be implemented as roles
- Example: DocMgr
  - Class provides access to a document
    - \* add, delete, modify lines, etc.
  - very specific purpose
  - does not know about any other objects

# Collaboration Synthesis

- Role based designs simplify the construction of new collaborations, known as *synthesis*
- Each object in a running program may play a role in multiple collaborations

# Collaboration Synthesis

- Role based designs simplify the construction of new collaborations, known as *synthesis*
- Each object in a running program may play a role in multiple collaborations
  - If object O must play roles  $\{R_1, R_2 \dots R_n\}$  then

# Collaboration Synthesis

- Role based designs simplify the construction of new collaborations, known as *synthesis*
- Each object in a running program may play a role in multiple collaborations
  - If object  $O$  must play roles  $\{R_1, R_2 \dots R_n\}$  then
  - the class of object  $O$  must implement the interfaces that specify these roles

# Collaboration Synthesis

- Application construction in this paradigm involves:
  - Identifying the collaborations required to accomplish system goals
  - Synthesizing custom classes to declare objects that play roles in these collaborations

## Example

- Suppose we want to develop a graphical browser that allows users to view and print documents
  - DocMgr class manages documents
  - Button class implements a button
  - ViewPort class displays information on the screen
- How many collaborations will a single document be involved in?

## Example: Class ViewPort

```
class ViewPort :
    public F1_Multiline_Output {

    void    registerModel(ViewPortModel *vpm)
           { model=vpm; }

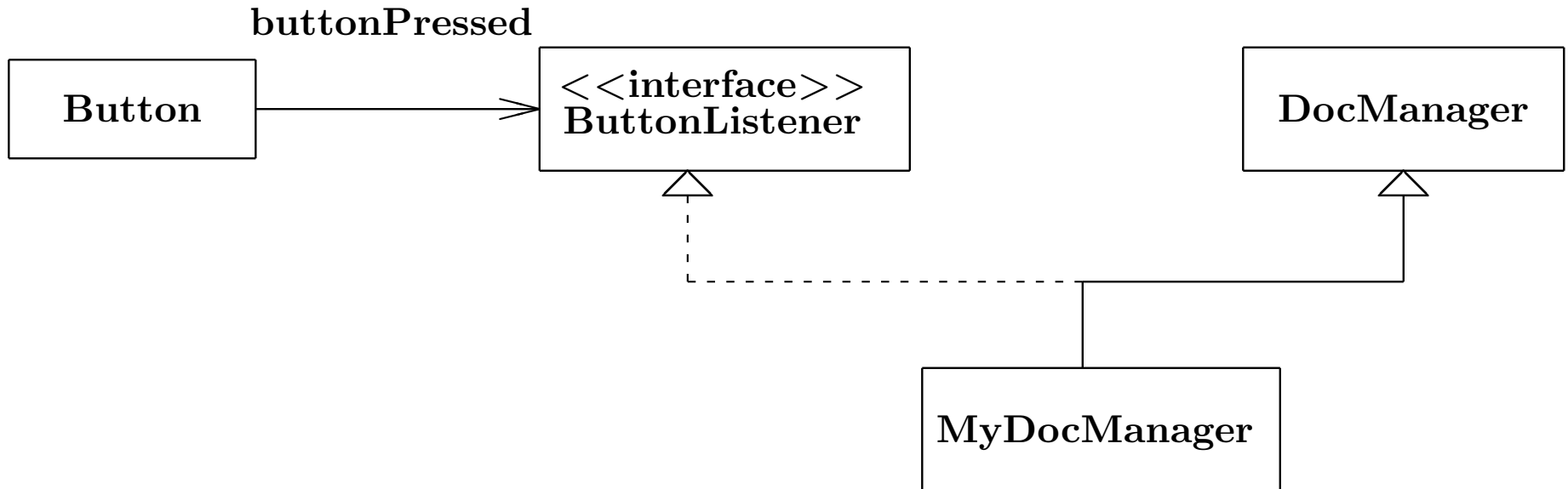
    protected:
    ViewPortModel * model;
    void resize();
};
```

## Example: Class DocMgr

```
class DocMgr {
    public:
    void    printDocument() const;
    unsigned docSize() const;
    const string & docLine(unsigned) const;

    void    insertLine(unsigned, const string &);
    void    appendLine(const string &);
    void    deleteLine(unsigned);
};
```

# Class Diagram Notation



## Synthesis of Multiple Roles

```
class MyDocMgr : public DocMgr,  
                public ButtonListener,  
                public ViewportModel {  
  
    public:  
    void buttonPressed (const string & s)  
        { if (s=="print") { DocMgr::printDocument() } }  
  
    bool retrieve(unsigned lineNo, string & line) const  
        { bool retVal= (lineNo < DocMgr::docSize());  
          if (retVal) line=DocMgr::docLine(lineNo);  
          return retVal; }  
};
```

## Motivation: Explaining a Design

- Modern OO systems require lots of collaborations
- Understanding such systems requires visualization
  - the interconnection of the objects (structure)
  - the dynamic interactions among these objects (behavior)
- How to visualize in a meaningful way?

# Sequence Diagrams

- Illustrate one instance of one collaboration among multiple objects
  - Each object may play a role in other collaborations

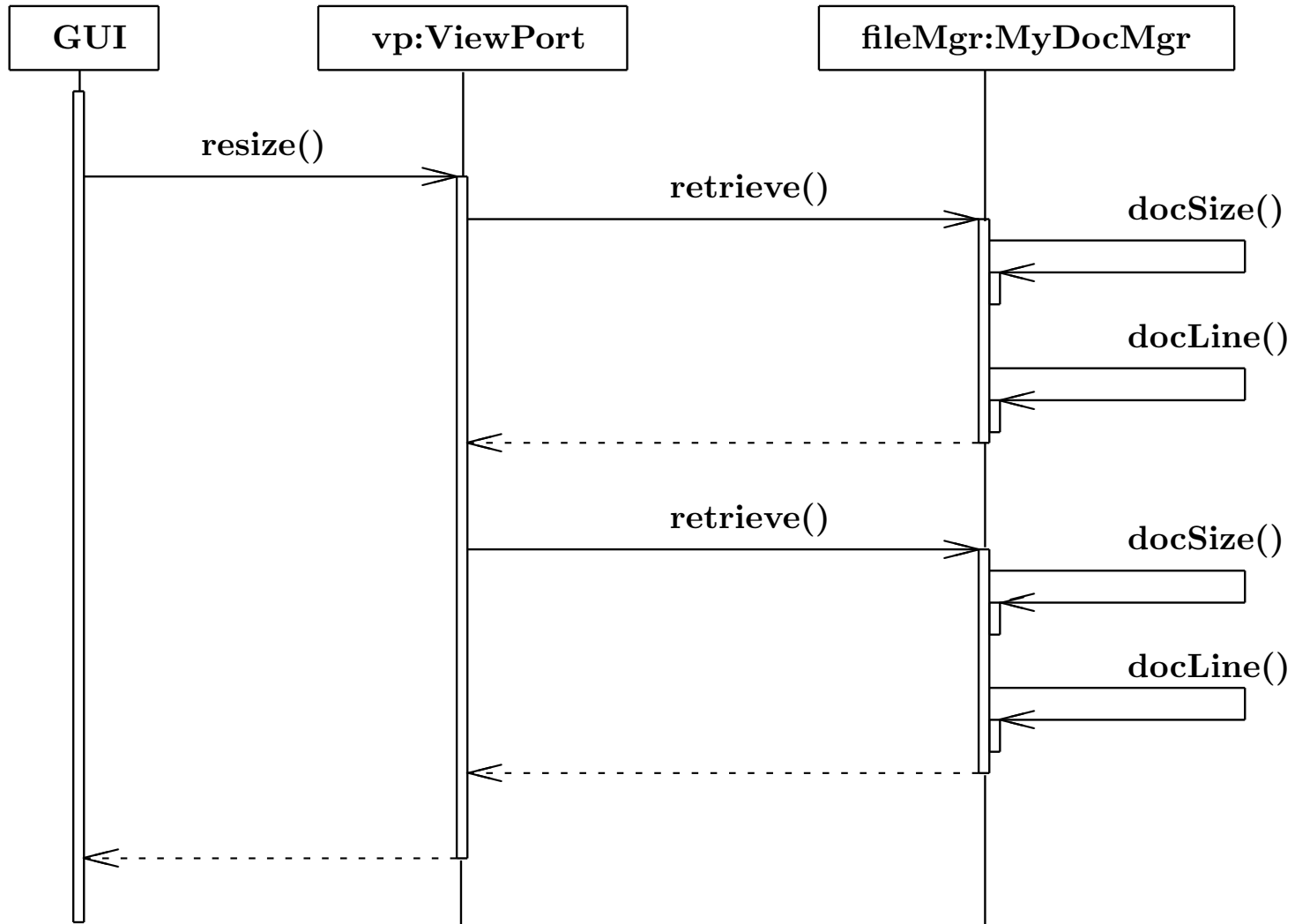
# Sequence Diagrams

- Illustrate one instance of one collaboration among multiple objects
  - Each object may play a role in other collaborations
- Notation

# Sequence Diagrams

- Illustrate one instance of one collaboration among multiple objects
  - Each object may play a role in other collaborations
- Notation
  - Use of spatial location to reflect time dimension
  - Use of vertical bars to denote object “activity”
  - Graphic denotation of returns from operations

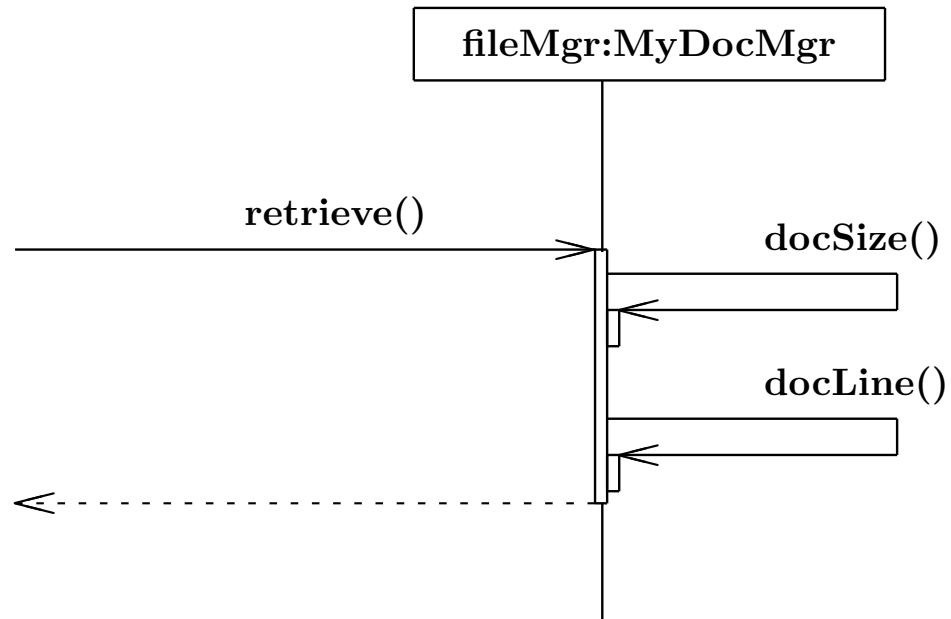
# Example Sequence Diagram



# Sequence Diagrams

- Left to right: caller/callee
- Focus of control
- Life lines

```
bool MyDocMgr::retrieve(unsigned lineNo,  
    string & line) const  
{  
    bool retVal=(lineNo < DocMgr::docSize());  
    if (retVal) line=DocMgr::docLine(lineNo);  
    return retVal;  
}
```

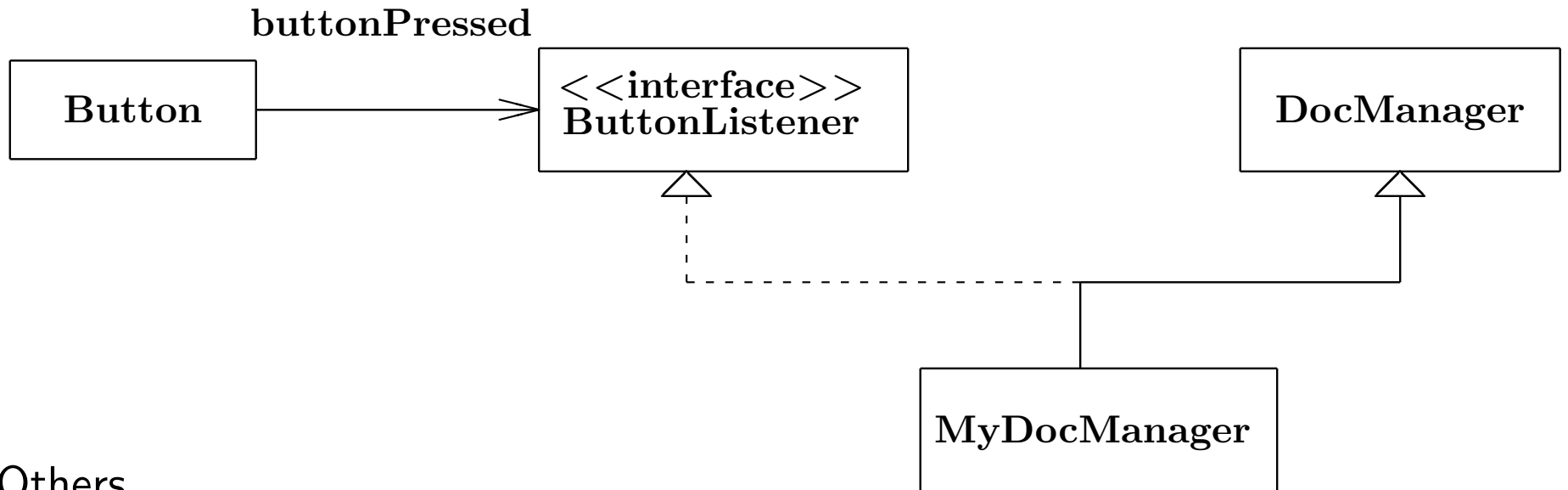


## Exercise

Draw a sequence diagram to illustrate the collaboration between `printButton` and `fileMgr`

# UML

- Different diagrams to represent different aspects of a design
- Sequence Diagrams capture behavior
- Class Diagrams



- Others