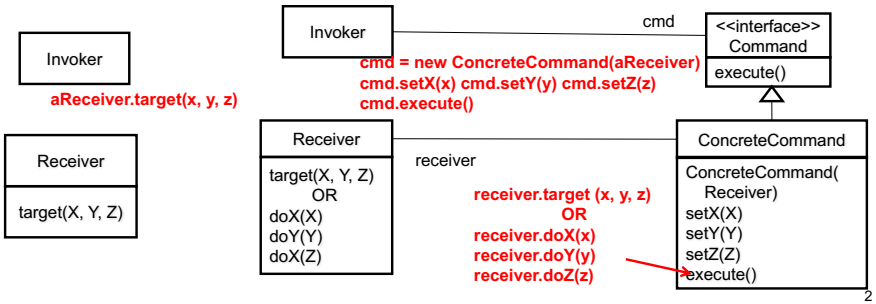
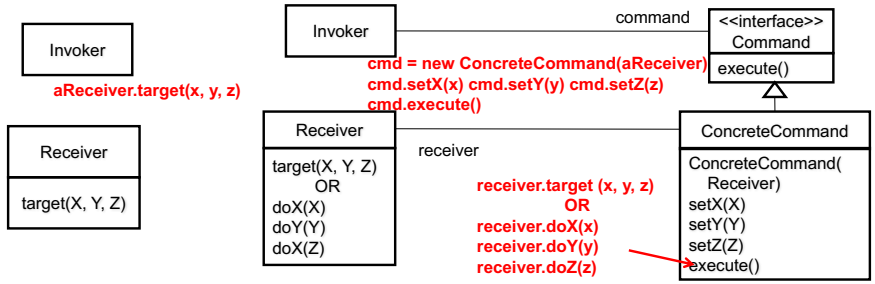
Command Design Patten

Command Design Pattern

- Intent
  - Encapsulate a request/command (a method call) and its relevant information (e.g., parameters) as a class.
    - Replace a method call with a class.

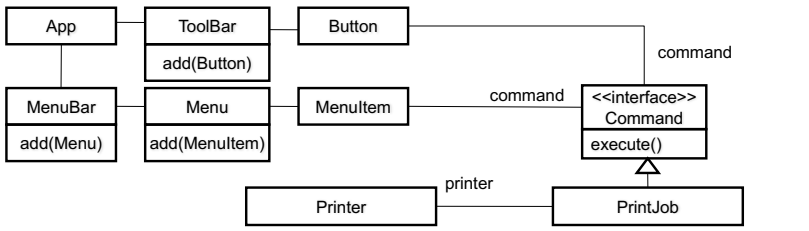
Simple method call **Command design pattern**

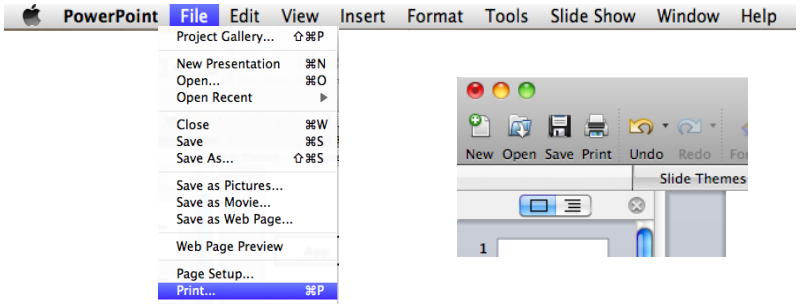


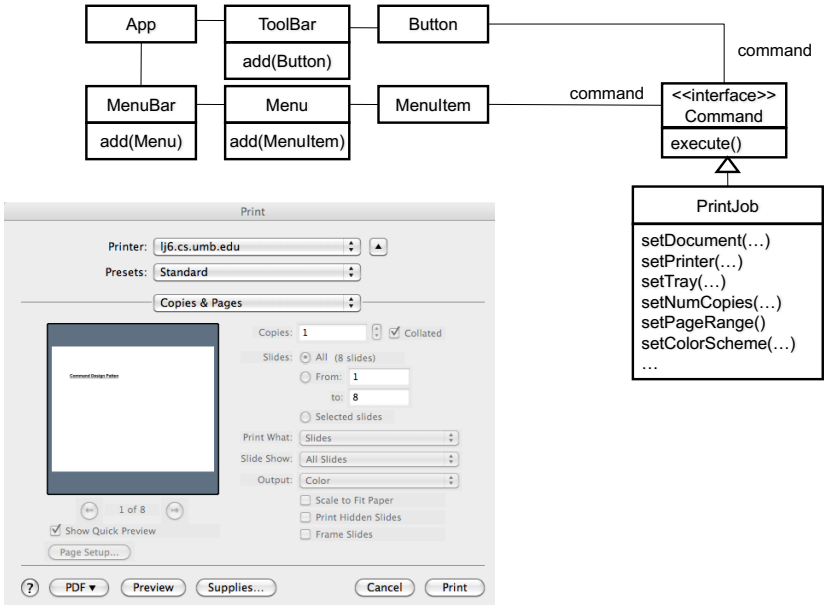


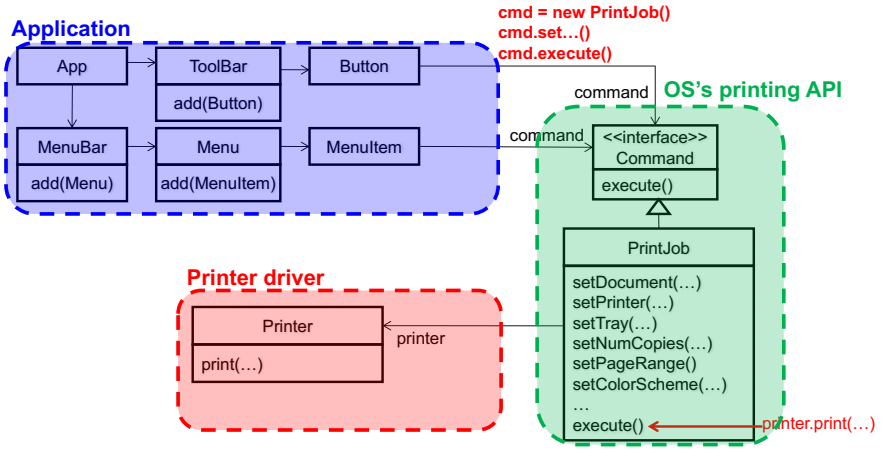
- Benefits
  - Loosely couple an invoker and a receiver
    - An invoker doesn't have to know how to perform a command.
    - Invokers can be intact when receivers are changed.
  - Make it easy to add new commands
    - No need to change invokers and receivers.

An Example: Print Command





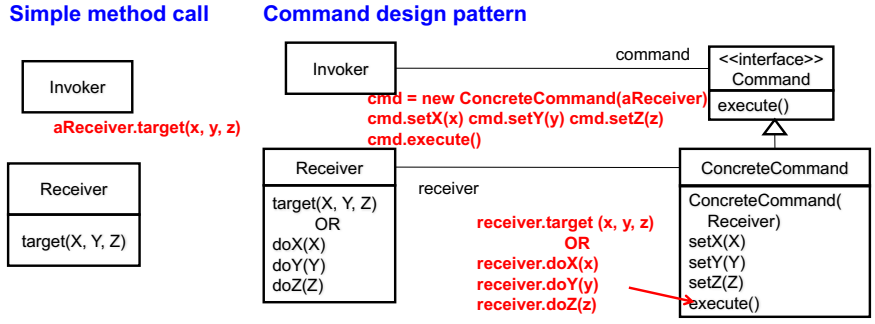




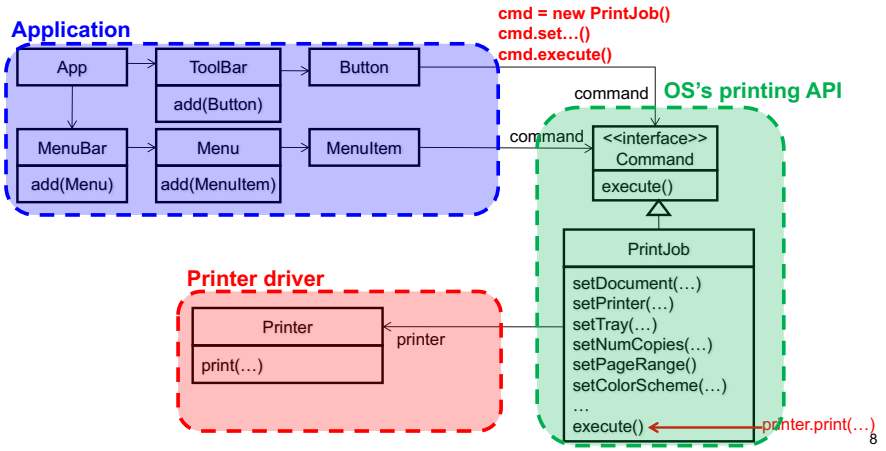
- Loosely couple invokers (a button and a menu item) and a printer
  - Invokers don't have to know how to perform a command.
    - They don't have to know the underlying printing facility (printer drivers, etc.)
  - Invokers can be intact when the printer (its driver) is changed.
- Make it easy to add new commands such as faxing, PDF/PS generation and "Send PDF via email" and "Send PDF via text."
  - No need to change the printer and invokers

When to Use *Command*?

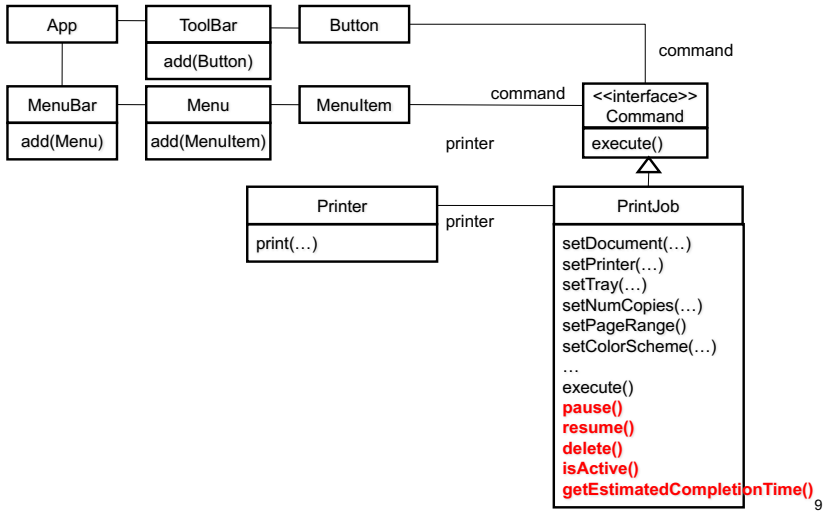
- When do you want to use *Command*, rather than a regular method call?
  - Why not using a regular method call?



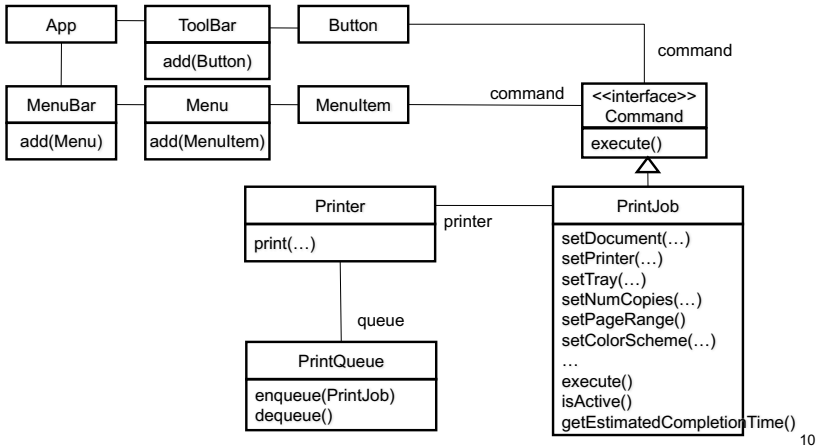
- When you want invokers and receivers to be loosely-coupled.



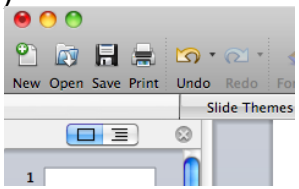
- When a command has many relevant information/parameters.
- When you have many invokers for each command.
- When you want to perform some operations on a command.

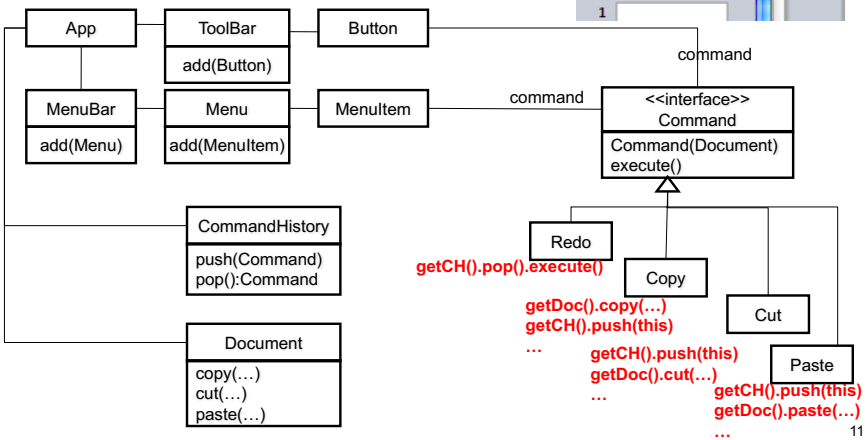


- When you would like to manage (or keep track of) multiple commands

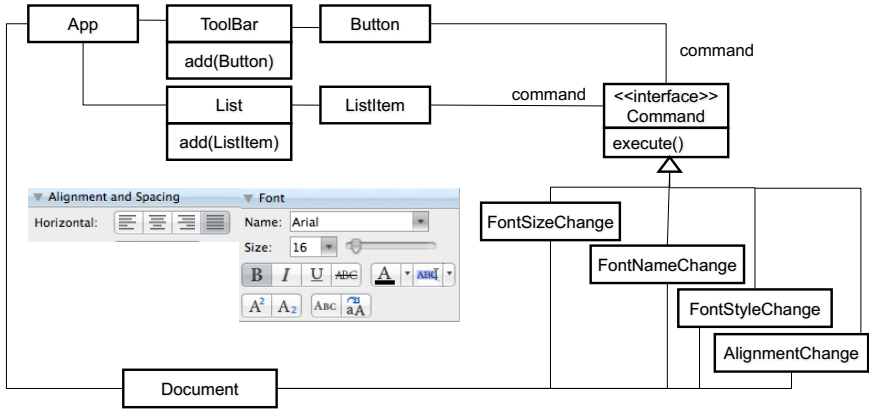


- When you would like to record (or log) command history
  - e.g., for implementing “undo” and “redo” function.



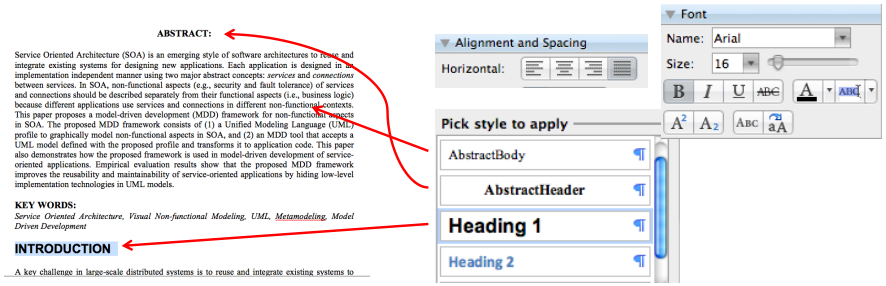


Another Example

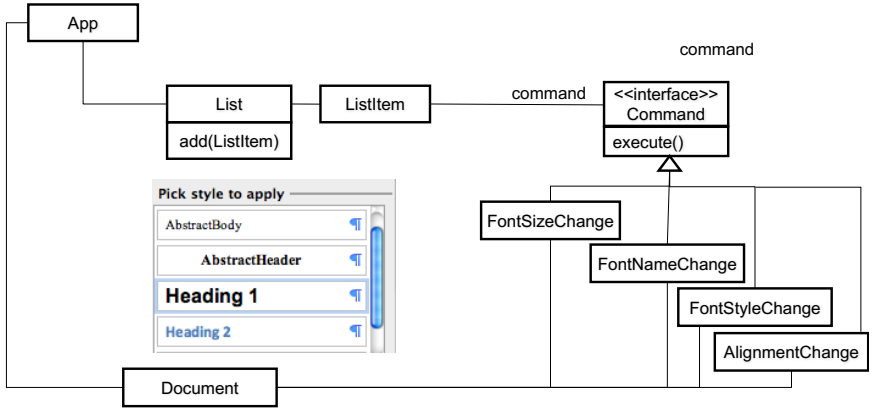


- These 4 command classes correspond to the buttons for changing *font size*, *font name*, *font style* and *alignment*.

A Sequence of Commands (Composite Command)

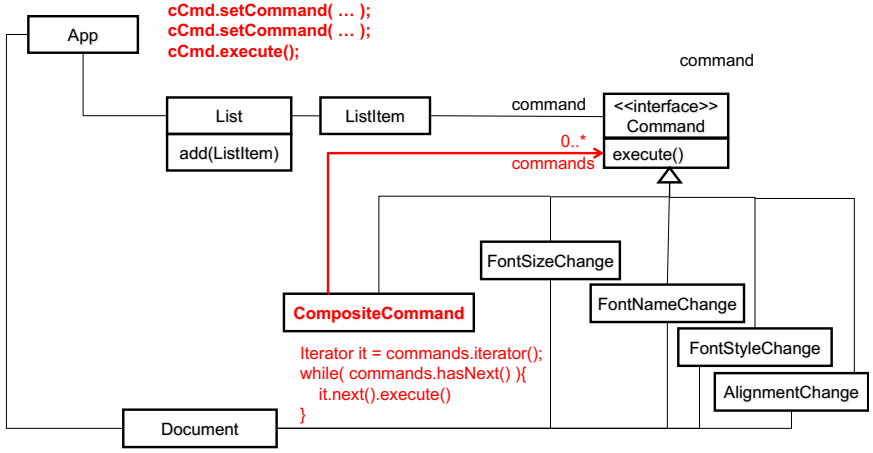


- Clicking "Abstract Header" style = performing the following 4 commands
  - Font size = 12pt
  - Font name = Times New Roman
  - Font style = bold
  - Alignment = center
- Clicking "Heading 1" style = performing the following 5 commands
  - Font size = 16pt
  - Font name = Arial
  - Font style = bold
  - Alignment = left
  - Effect = all caps



- How would you design command classes to change formatting styles?
  - e.g., "Abstract Header" (12pt, Times, bold, center)

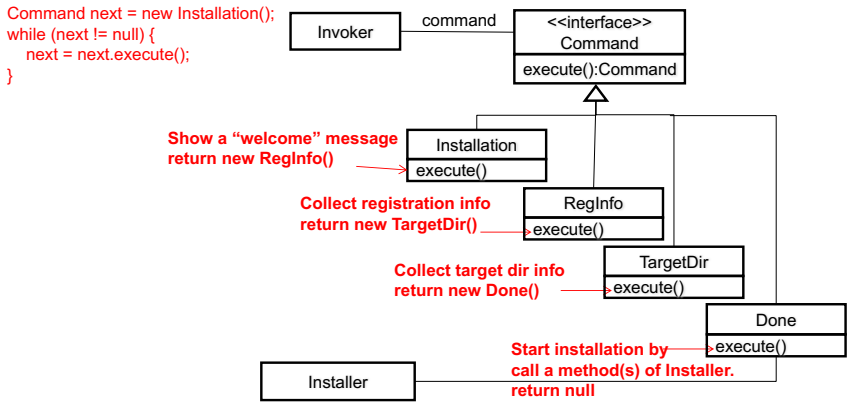
Command cCmd = new CompositeCommand("Abstract Header");
Create 4 command objects to change font size, font name, font style and alignment.
For each command object, set a parameter value(s) (e.g. 12pt, Times, etc.).
cCmd.setCommand(…);
cCmd.setCommand(…);
cCmd.setCommand(…);
cCmd.execute();



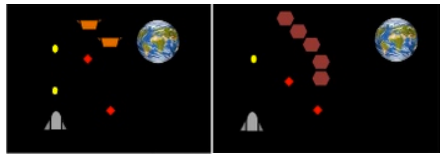
Command + Composite design patterns (plus Iterator)

One More Example: Wizards

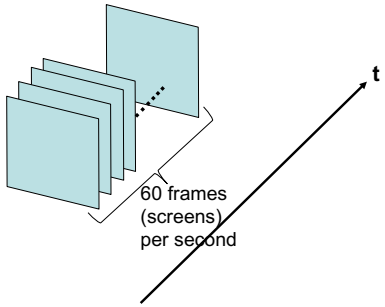
- Wizard
  - e.g., installation wizard, project creation wizard, refactoring wizard
  - Shows a sequence of modal dialogs to collect a set of data from the user
    - e.g., one dialog to enter registration information (user name, serial number, etc.)
    - Another dialog to specify the directory to install an app in question
    - Another dialog to enable and disable application components/features
    - Moves one dialog to another with the "next" and "back" buttons
  - Starts a particular action/process only when the "finish" button is clicked on the last dialog.

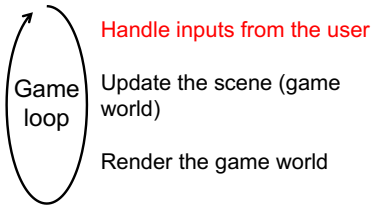


Imagine a Simple 2D Game

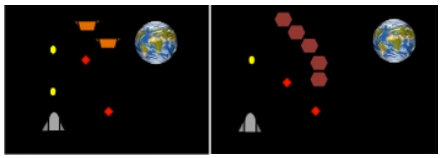


- The game loop is iterated repeatedly.
- Each iteration takes care of rendering one frame (or screen).
- Typical frame rate (FPS)
  - 60 iterations per second
  - 1.6 msec (1/60 sec) per frame

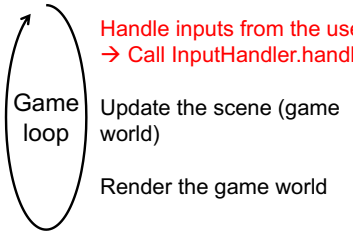




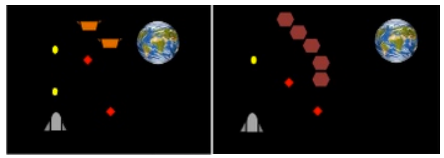
Handling User Inputs



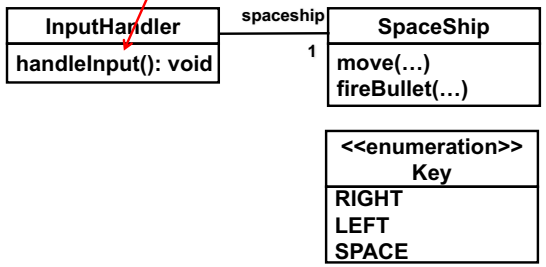
- 3 types of inputs
  - The user can push the right arrow, left arrow and space keys.
    - R arrow to move right
    - L arrow to move left
    - Space to fire a bullet
- InputHandler
  - Collects user inputs and respond to them.
  - handleInput()
    - identifies a keyboard input since the last game loop iteration (i.e. since the last frame).
    - One input per frame (i.e. during 1.6 msec)



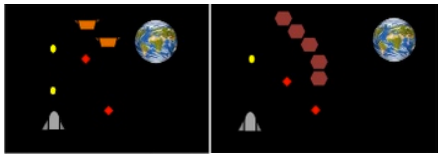

```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput();
  ...
}
```

```
if( isPressed(Key.RIGHT) )
  spaceship.move(…);
else if( isPressed(Key.LEFT) )
  spaceship.move(…);
else if( isPressed(Key.SPACE) )
  spaceship.fireBullet(…);
```



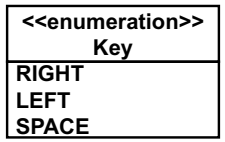
Not That Good… Why?



```
if( isPressed(Key.RIGHT) )
  spaceship.move(…);
else if( isPressed(Key.LEFT) )
  spaceship.move(…);
else if( isPressed(Key.SPACE) )
  spaceship.fireBullet(…);
```

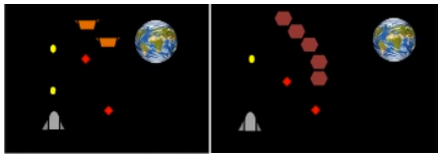
Game loop

Handle inputs from the user
→ Call InputHandler.handleInput()



- User inputs and Spaceship's actions are tightly coupled.
  - Invocations of move() and fireBullet() are hard-coded in handleInput().
  - Hard to allow the user to change key bindings.

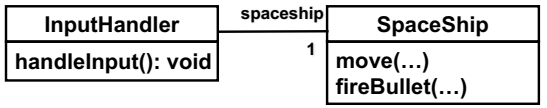
A Design Revision Needed

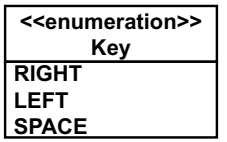


```
if( isPressed(Key.RIGHT) )
  spaceship.move(…);
else if( isPressed(Key.LEFT) )
  spaceship.move(…);
else if( isPressed(Key.SPACE) )
  spaceship.fireBullet(…);
```

Game loop

Handle inputs from the user
→ Call InputHandler.handleInput()

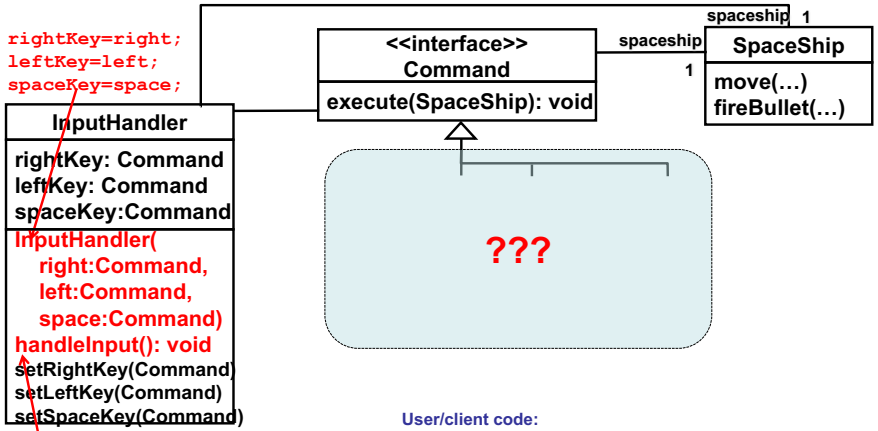




- Making user inputs and Spaceship's actions loosely coupled.
  - Making Spaceship's actions to be interchangeable for each keyboard input
  - Need to call move() and fireBullet() ***indirectly***.
  - Need an intermediate class in b/w InputHandler and SpaceShip

SpaceShip's Actions as Command Classes

```
rightKey=right;
leftKey=left;
spaceKey=space;
```



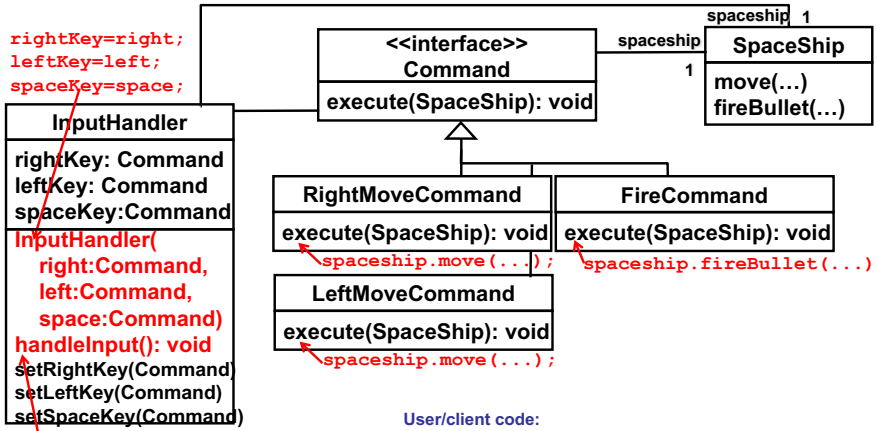

```
if(isPressed(Key.RIGHT))
  ???
else if(isPressed(Key.LEFT))
  ???
else if(isPressed(Key.SPACE))
  ???
```

User/client code:

```
InputHandler ih =
  new InputHandler(new RightMoveCommand(...),
                   new LeftMoveCommand(...),
                   new FireCommand(...));
while(true){
  ih.handleInput();
  ... }
```


SpaceShip's Actions as Command Classes

```
rightKey=right;
leftKey=left;
spaceKey=space;
```




```
spaceship.move(...);
spaceship.fireBullet(...)
spaceship.move(...);
```

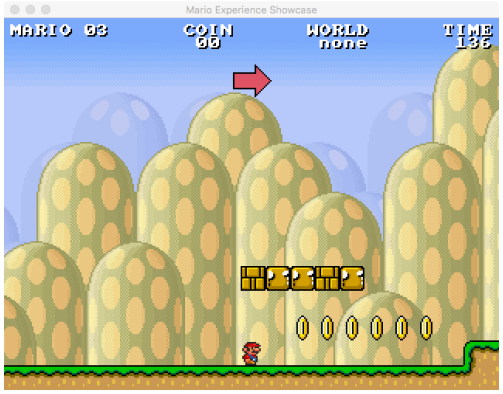
```
if(isPressed(Key.RIGHT))
  rightKey.execute(spaceship);
else if(isPressed(Key.LEFT))
  leftKey.execute(spaceship);
else if(isPressed(Key.SPACE))
  spaceKey.execute(spaceship);
```

User/client code:

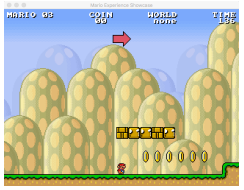
```
InputHandler ih =
  new InputHandler(new RightMoveCommand(...),
                   new LeftMoveCommand(...),
                   new FireCommand(...));
while(true){
  ih.handleInput();
  ... }
```

Imagine a Simple 2D Game

- Super Mario
  - http://www.marioai.org/
  - http://www.platformersai.com



Handling User Inputs



Handle inputs from the user
→ Call InputHandler.handleInput()

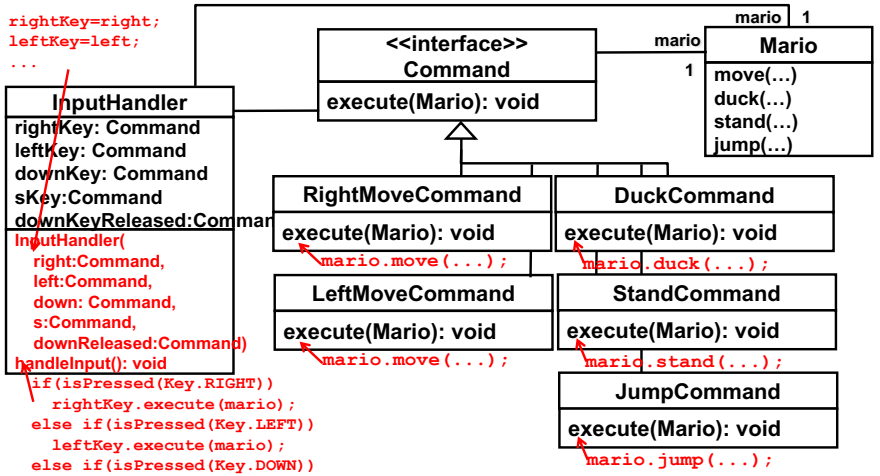
Update the scene (game world)

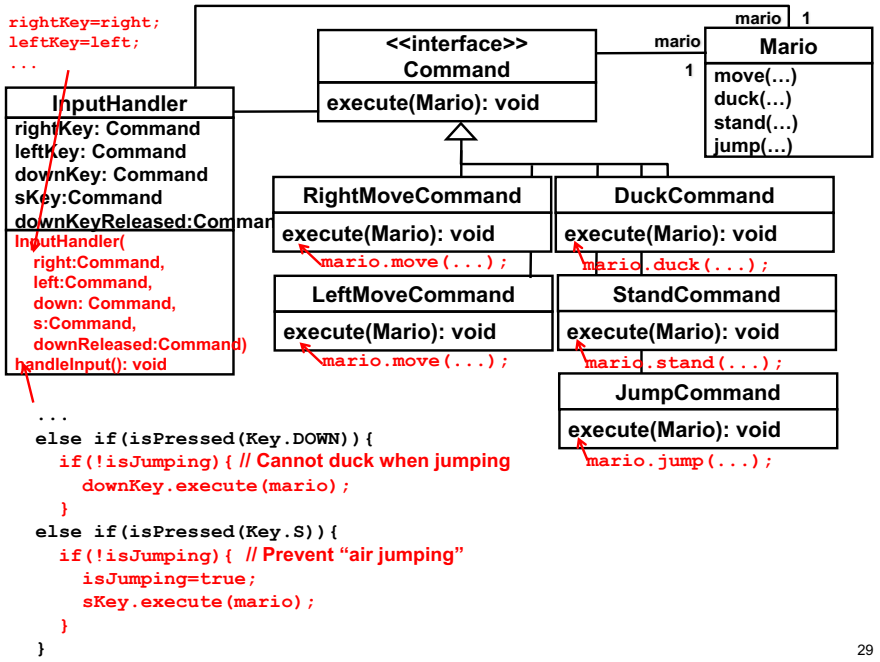
Render the game world

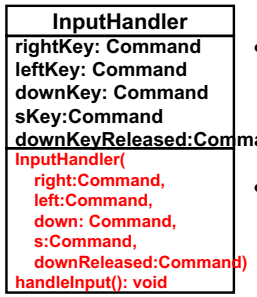
```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput();
  ...
}
```

- 5 types of inputs
  - The user pushes the right arrow, left arrow, down arrow and "s" keys.
    - R arrow to move right
    - L arrow to move left
    - D arrow to duck
    - "s" to jump
  - The user releases the D arrow to stand up.
- InputHandler
  - handleInput()
    - identifies a keyboard input since the last game loop iteration (i.e. since the last frame).
    - 60 frames/s (FPS): One input per frame (i.e. during 1.6 msec)

Mario's Actions as Command Classes








```
...
else if(isPressed(Key.DOWN)){
  // Cannot duck when jumping. Can jump only on the ground.
  if(!isJumping){
    ducking=true;
    downKey.execute(mario);
  }
else if(isPressed(Key.S)){
  // Prevent "air jumping." Cannot jump when ducking.
  if(!isJumping && !isducking){
    isJumping=true;
    sKey.execute(mario);
  }
else if(isReleased(Key.DOWN)){
  if(isDucking){
    isDucking=false;
    sKeyRelased.execute(mario);
  }
...
```

- The number of conditional branches increases and becomes less maintainable,
  - as the number of Mario's states and behaviors increases.
    - Mario moves faster when the "a" key and the right/left key are pushed at the same time.
    - Mario "dives" if the "down" key is pushed when jumping.

Define Mario's States as State Classes