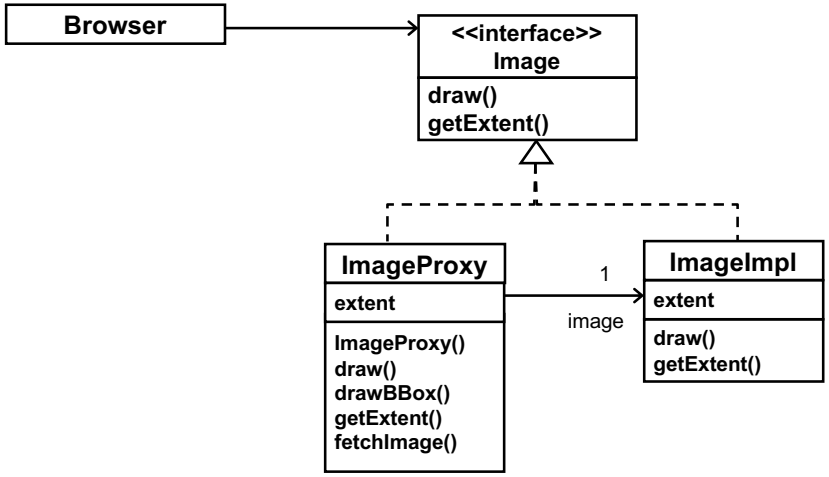
Proxy Design Pattern

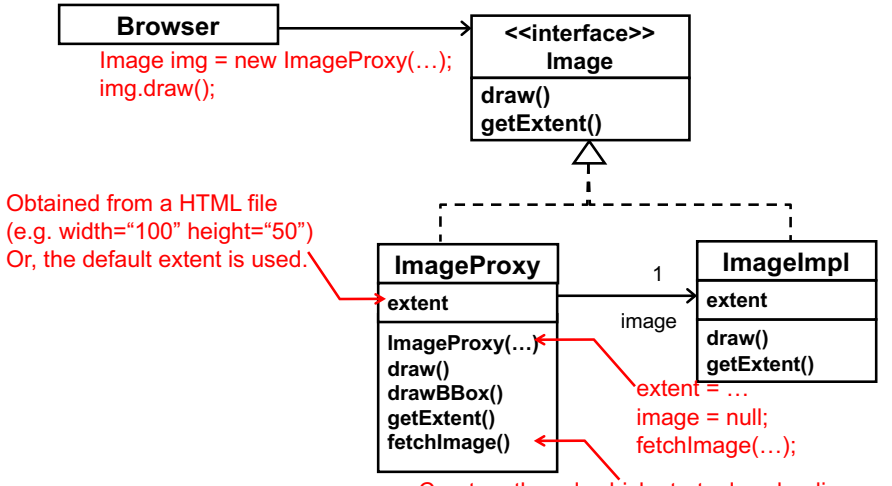
Proxy Design Pattern

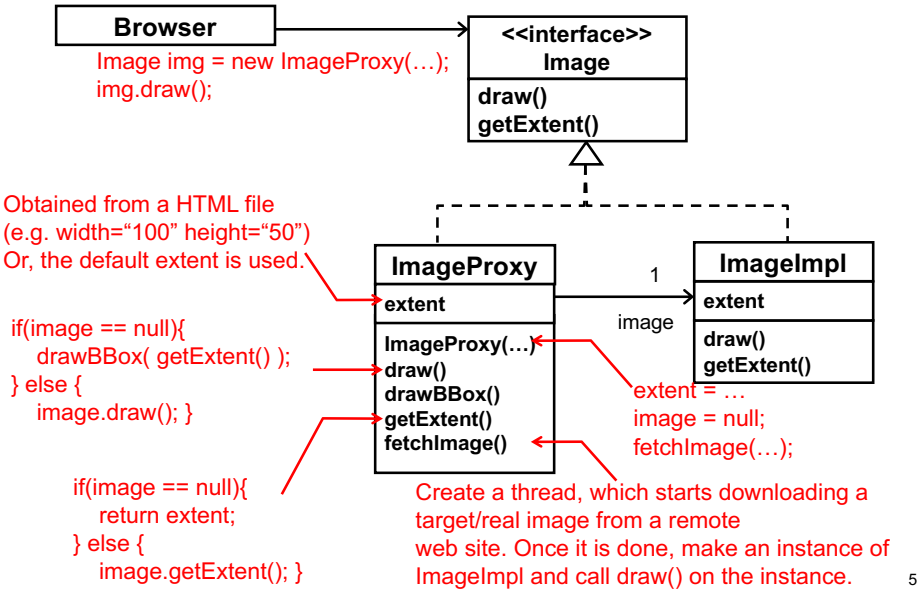
- Intent
  - Provide a surrogate or placeholder for another object to control access to it.
- An example: Handling images in a web browser
  - When an HTML file contains images,
    - A bounding box (placeholder) is displayed first for each image
      - Until the image is downloaded and ready to be displayed.
        - » Most users are not patient enough to keep watching blank browser windows until all text and images are downloaded and displayed.
    - Whenever the image is downloaded, the bounding box is replaced with the real image.

An Example of *Proxy*



Proxy Class

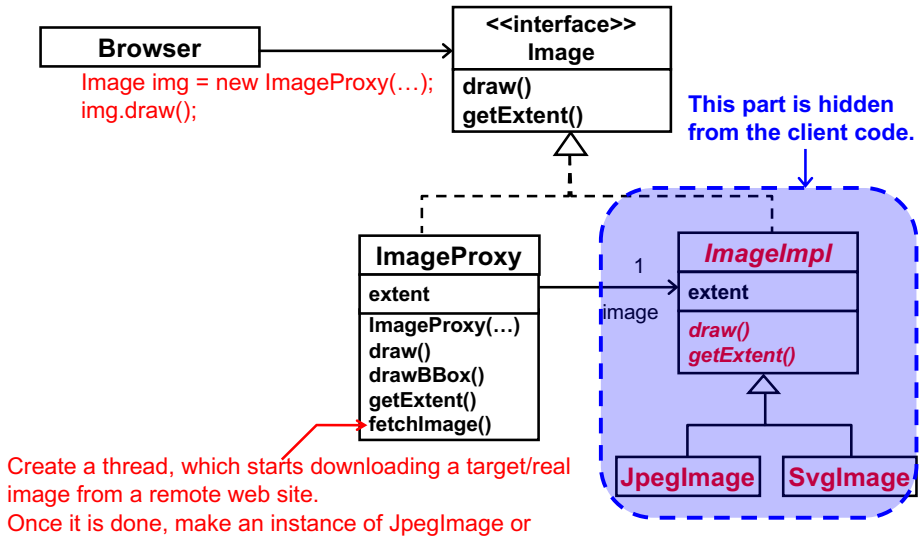




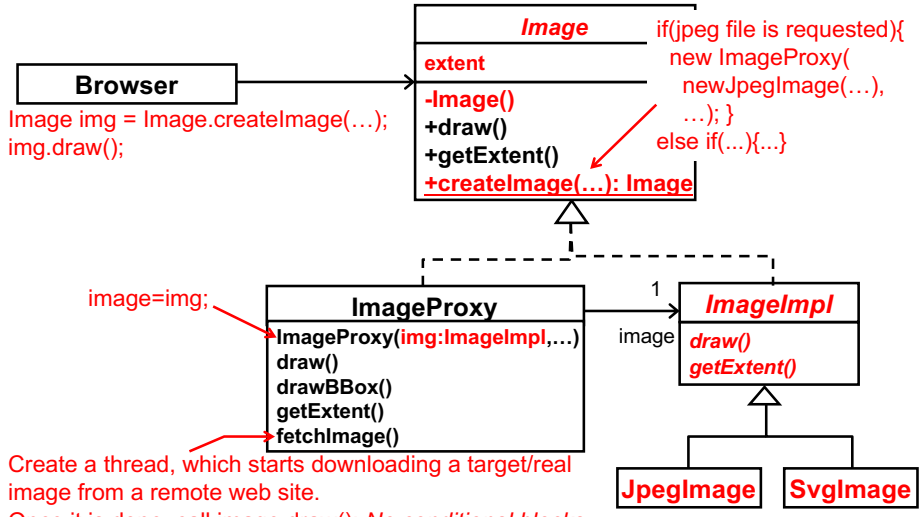
What's the Point?

- Loosely couple bounding box placement and image rendering.
- Why is that important?
  - Changes are expected for
    - Image formats that the browser supports.
    - Rendering algorithms
  - Bounding box placement is independent from those changes.
  - Separate *what can change often* from *what wouldn't* to improve maintainability.

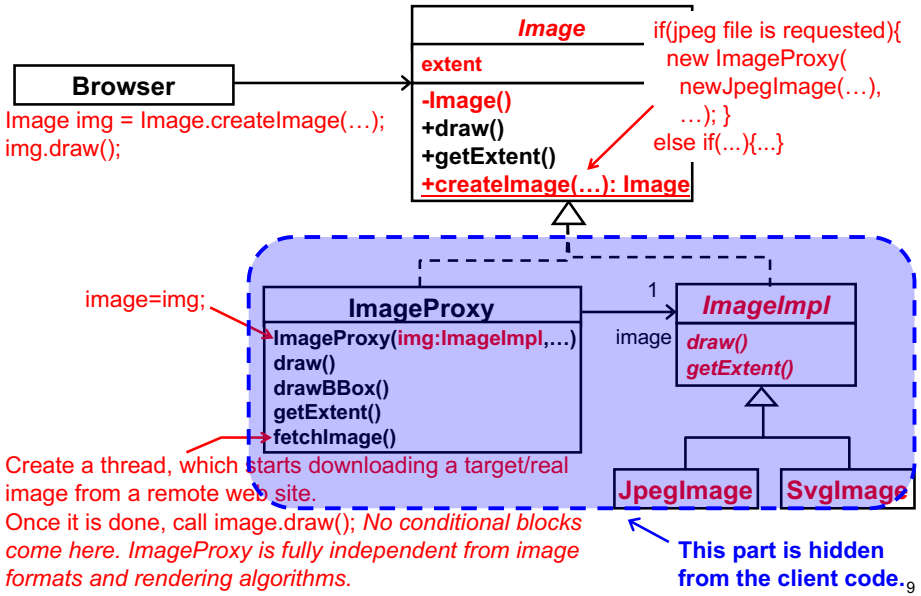
Supporting Multiple Image Formats



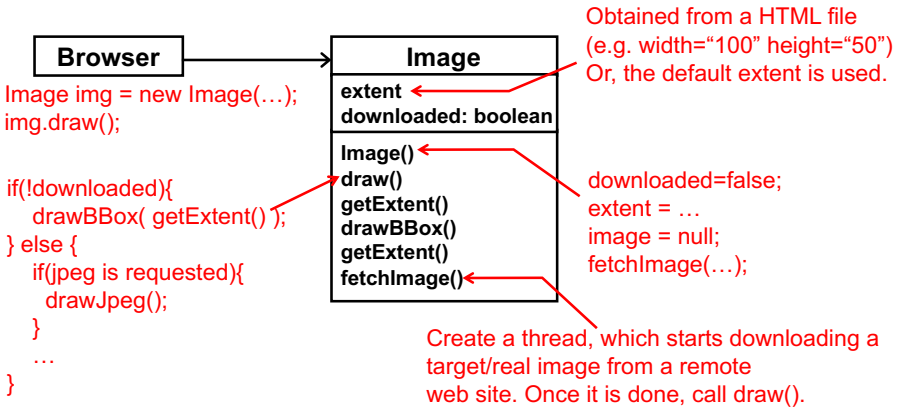
One Step Further with *Factory Method*



One Step Further with *Factory Method*



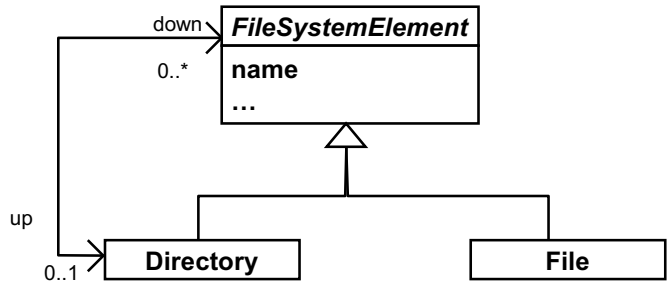
What if Everything is Integrated into a Single Class?



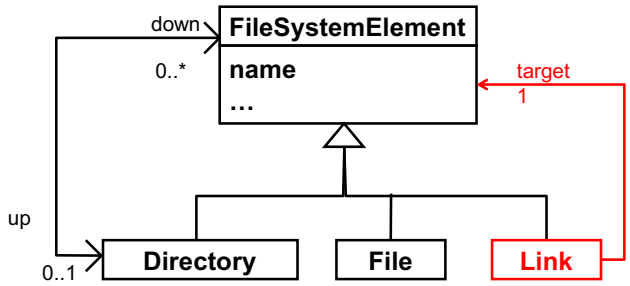
Bounding box placement, choices of image formats and image rendering are all mixed up in a single class, which will become fat and spaghetti (i.e. unmaintainable) soon.

Better design strategy: Separation of concerns

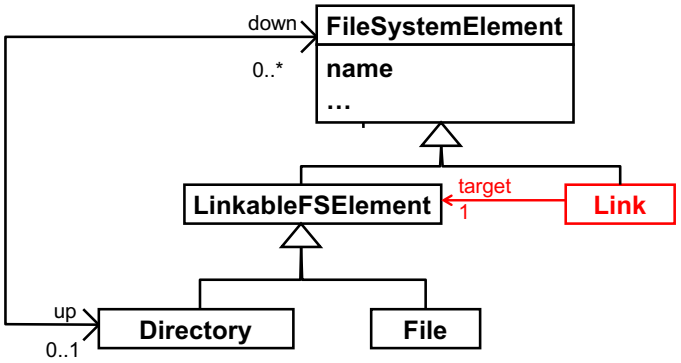
Add Proxy to your File System



- Add a symbolic link feature
  - a.k.a. alias (Mac), shortcut (Windows)
- A link acts as a proxy of a directory or file.
- Use the Proxy design pattern.

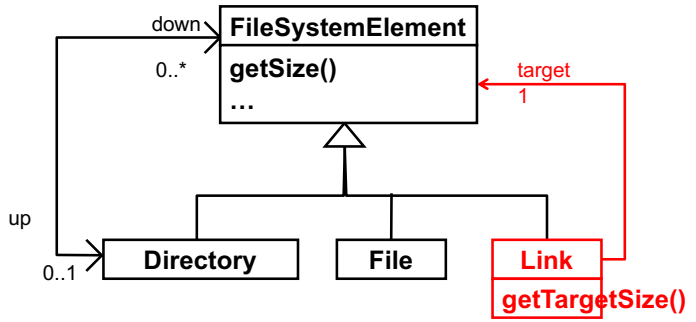


- A link acts as a proxy of a directory or file.
  - A link can act as a proxy of a link too.

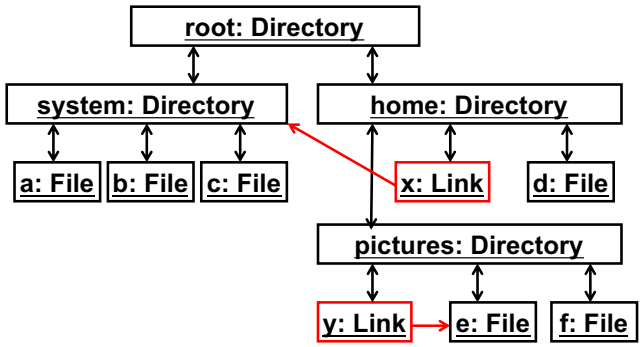


- A link acts as a proxy of a directory or file.
  - A link can act as a proxy of a link too.

HW13: Add Link to your HW 12 Code



- Link.getSize returns 0.
- If a Link refers to a file, getTargetSize() returns the size of that file.
- If a Link refers to a directory, getTargetSize() returns the total amount of disk consumption by the directory's all child nodes.
- If a Link refers to another link, getTargetSize() goes through a chain of links until it reaches a file or directory.



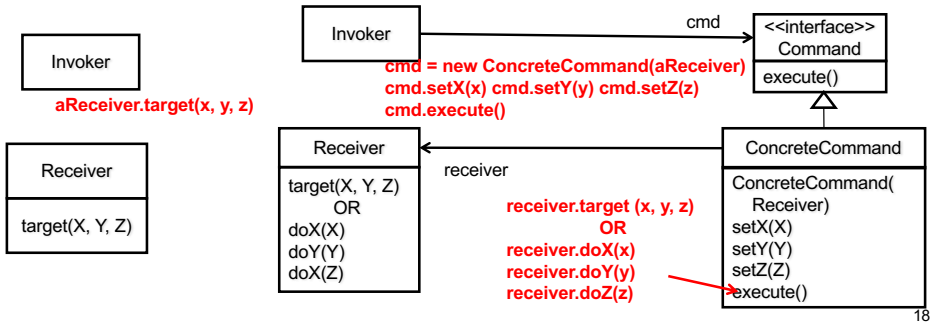
- Make this tree structure in your test cases.
  - Assign values to data fields (size, owner, etc.) as you want.
  - Call getSize() on x, y and the root directory.
  - Call getTargetSize() on x and y.
  - Call showAllElements() to print out this tree structure.
    - You can define your own textual format.

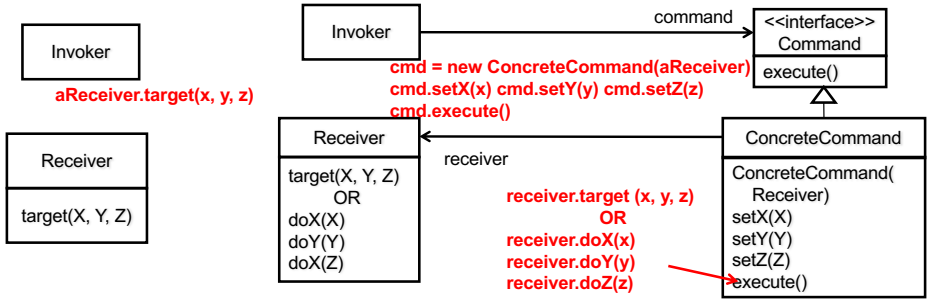
Command Design Patten

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.

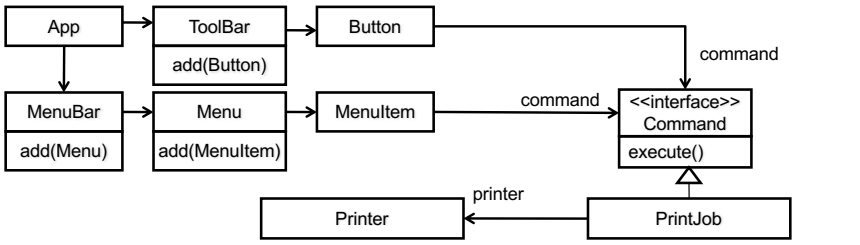
Simple method call **Command design pattern**

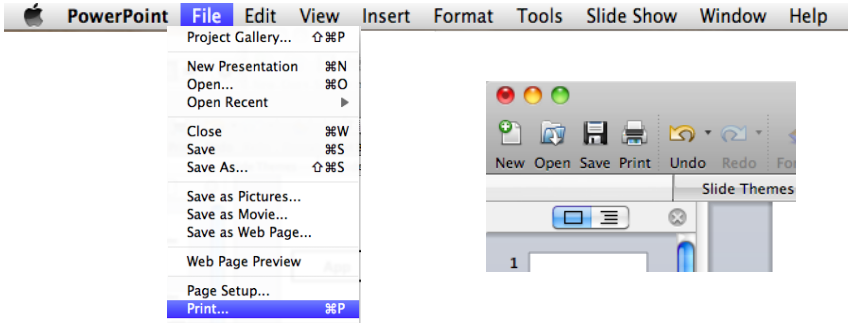


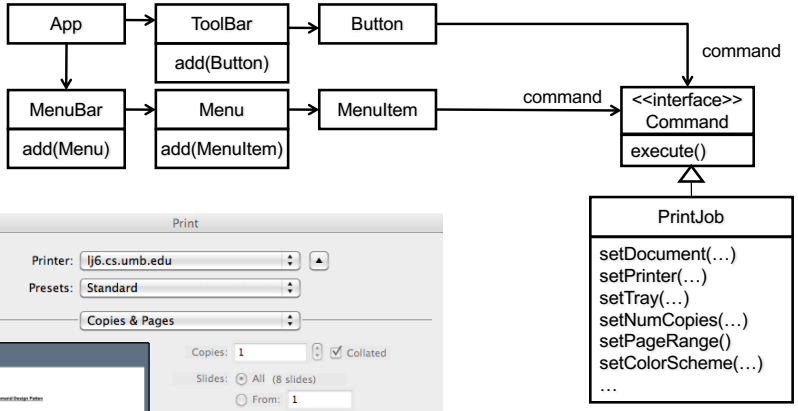


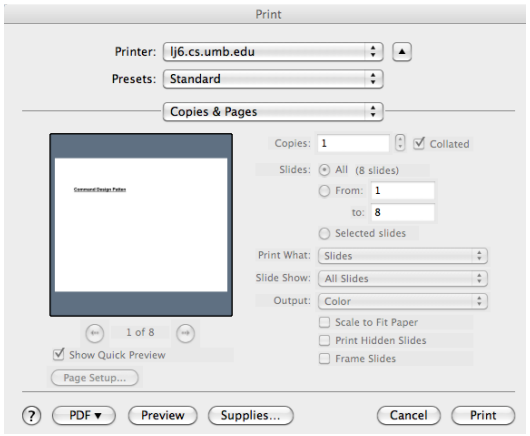
- Benefits
  - Loosely couple an invoker and a receiver
    - An invoker doesn't have to know how to perform a command.
    - Invokers can be intact when receivers are changed.
  - Make it easy to add new commands
    - No need to change invokers and receivers.

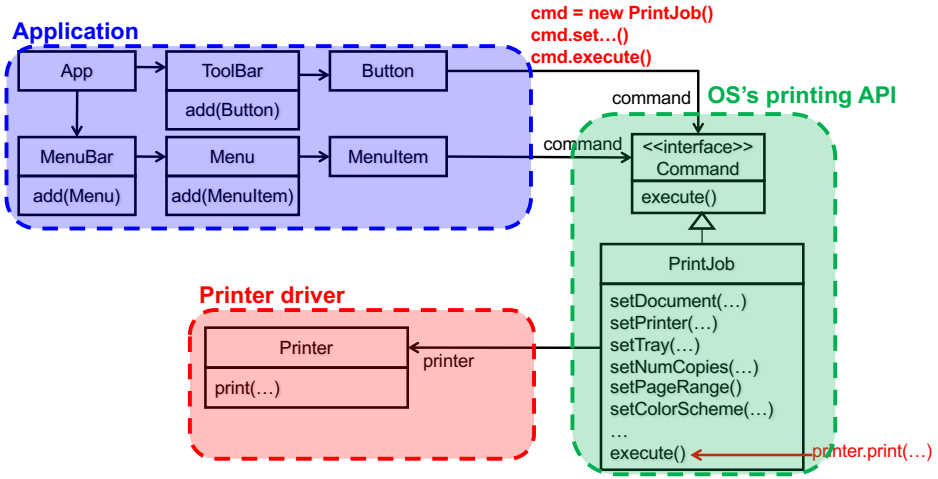
An Example: Print Command









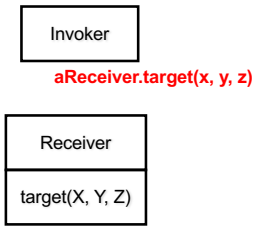


- Loosely couple invokers (a button and a menu item) and a printer
  - Invokers don't have to know how to perform a command.
    - They don't have to know the underlying printing facility (printer drivers, etc.)
  - Invokers can be intact when the printer (its driver) is changed.
- Make it easy to add 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

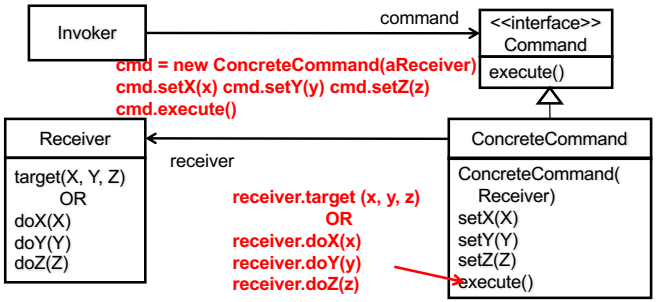
When to Use *Command*?

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

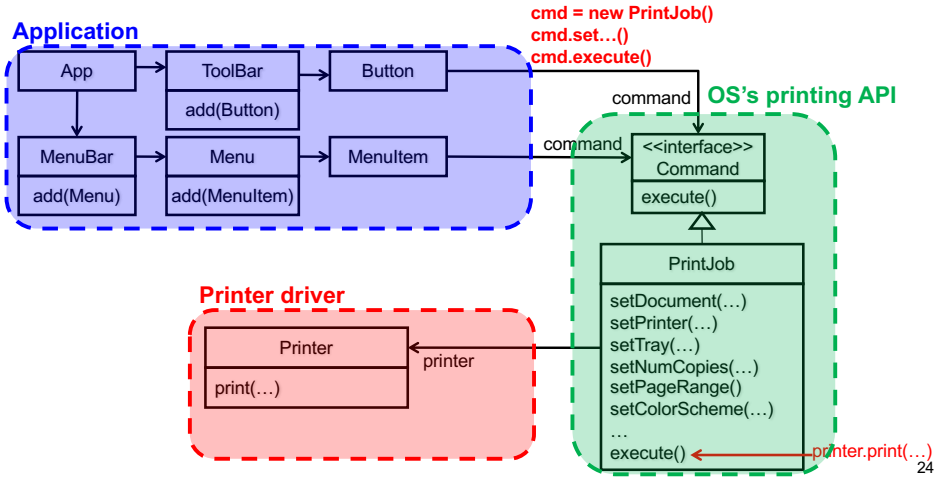
Simple method call



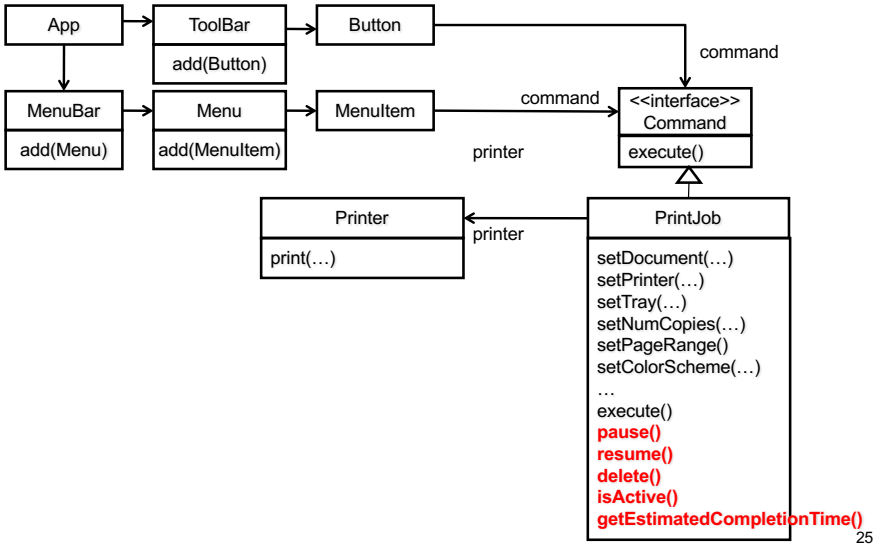
Command design pattern



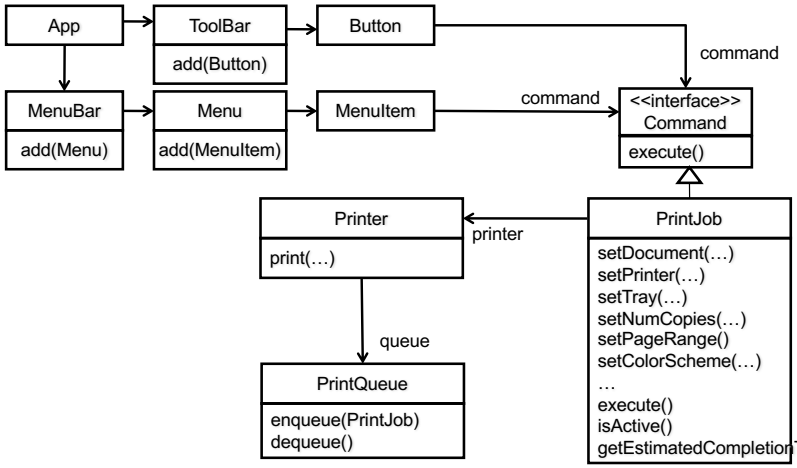
- When you want invokers and receivers to be loosely-coupled.



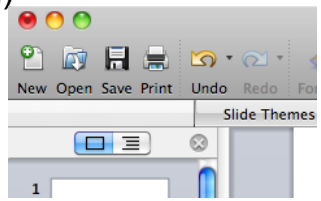
- When a command has many relevant information/parameters.
- When you have many invokers for each command.
- When you want to perform some operations on a command.

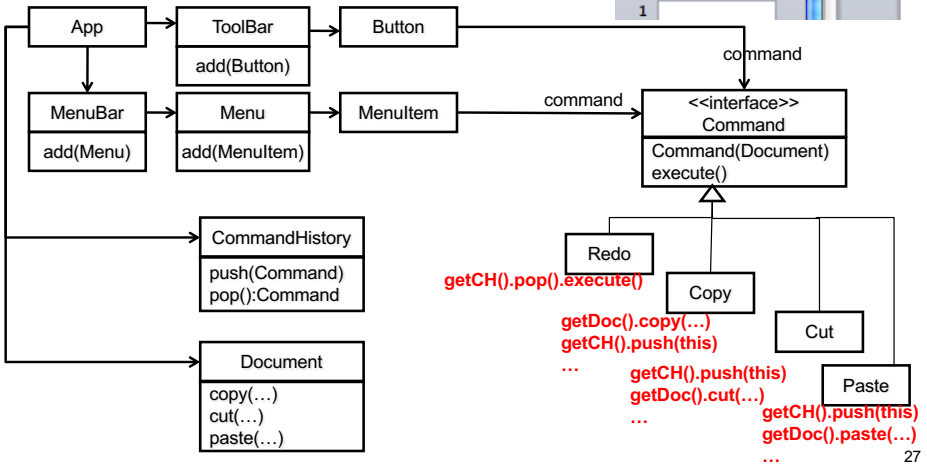


- When you would like to manage (or keep track of) multiple commands

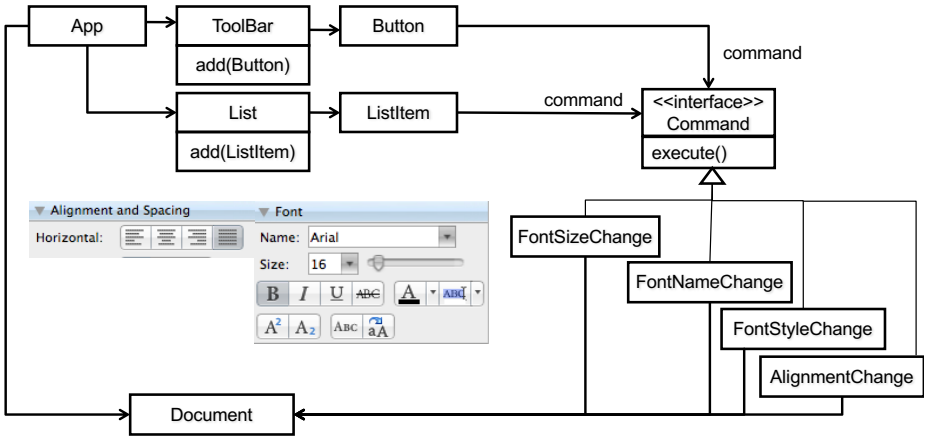


- When you would like to record (or log) command history
  - e.g., for implementing "undo" and "redo" function.



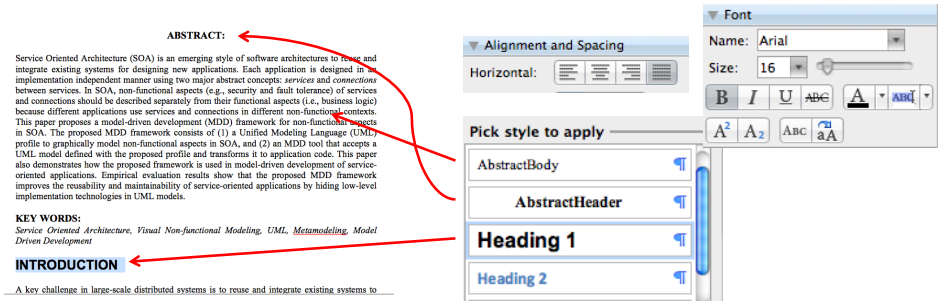


Another Example

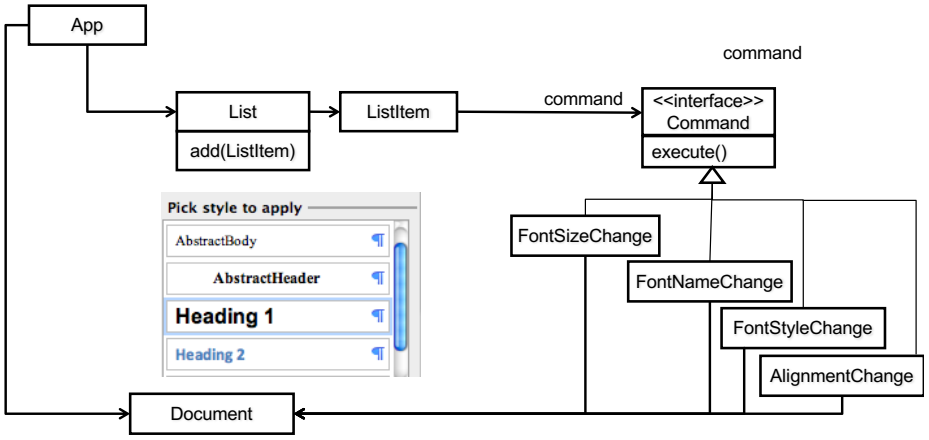


- These 4 command classes correspond to the buttons for changing font size, font name font style and alignment.

A Sequence of Commands (Composite Command)

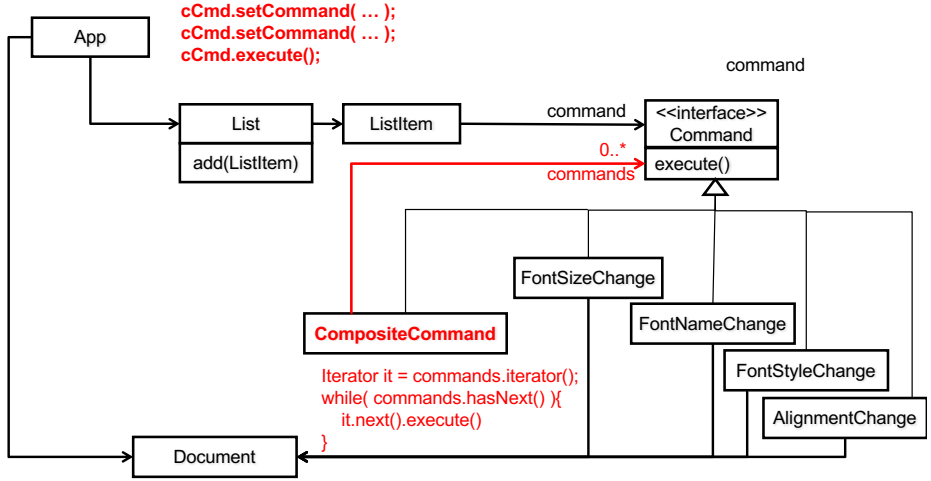


- Clicking "Abstract Header" style = performing the following 4 commands
  - Font size = 12pt
  - Font name = Times New Roman
  - Font style = bold
  - Alignment = center
- Clicking "Heading 1" style = performing the following 5 commands
  - Font size = 16pt
  - Font name = Arial
  - Font style = bold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

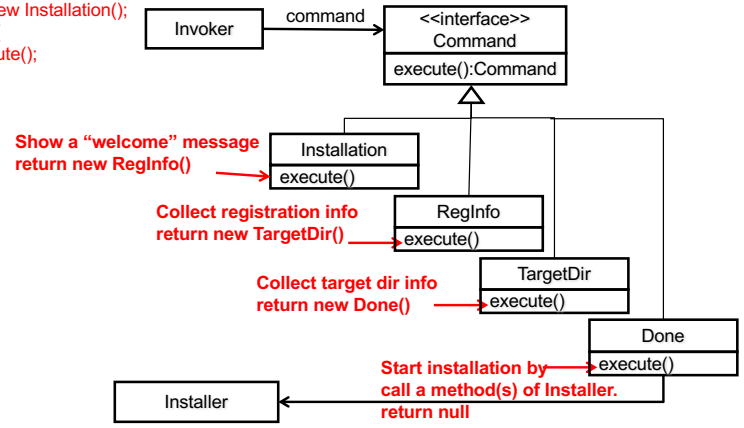
HW14

- Complete the pseudo code by
  - Replacing natural language descriptions with pseudo code
  - Replacing "…" with pseudo code
- You are free to add methods to command classes.
- Do not send me the entire pseudo code
  - I don't need pseudo code for the classes that appear in Slide 31.
  - I need pseudo code for *client code* that uses the classes in Slide 31.

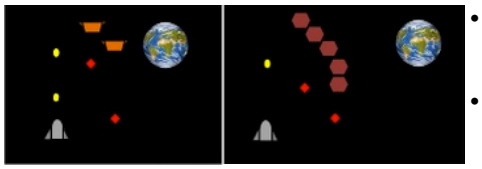
Wizards

- 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
  - Performs a single action/command only when the "finish" button is clicked on the last dialog.

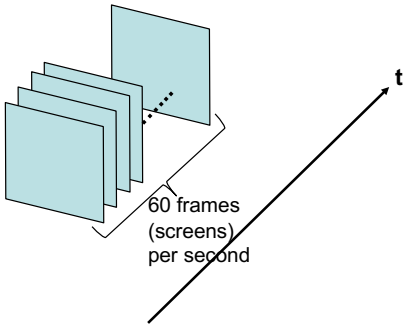
```
Command next = new Installation();
while (next != null) {
    next = next.execute();
}
```

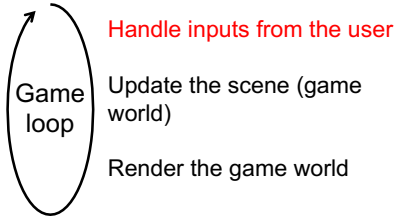


Imagine a Simple 2D Game

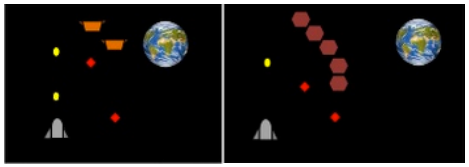


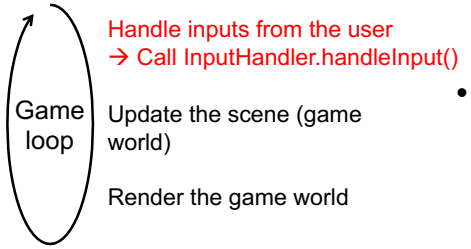
- The game loop is iterated repeatedly.
- Each iteration takes care of rendering one frame (or screen).
- Typical frame rate (FPS)
  - 60 iterations per second
  - 1.6 msec (1/60 sec) per frame





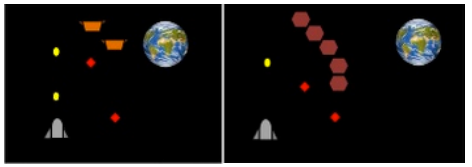
Handling User Inputs



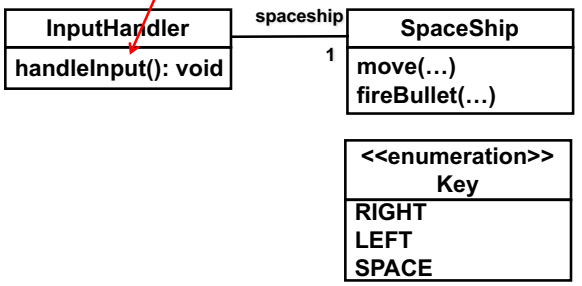


- 3 types of inputs
  - The user can push the right arrow, left arrow and space keys.
    - R arrow to move right
    - L arrow to move left
    - Space to fire a bullet
- InputHandler
  - Collects user inputs and respond to them.
  - handleInput()
    - identifies a keyboard input since the last game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

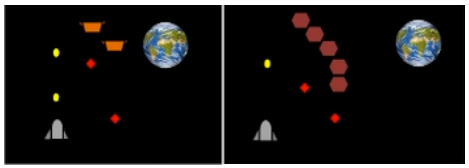
```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput();
  ...
}
```


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



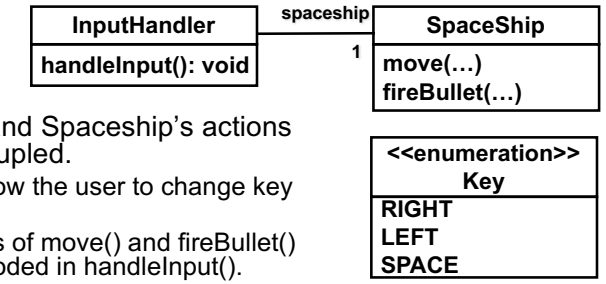
Not That Good… Why?



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

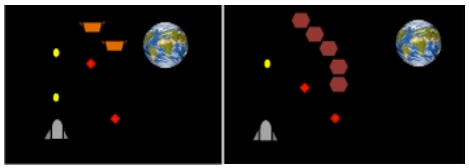
Handle inputs from the user
→ Call InputHandler.handleInput()

Game loop



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

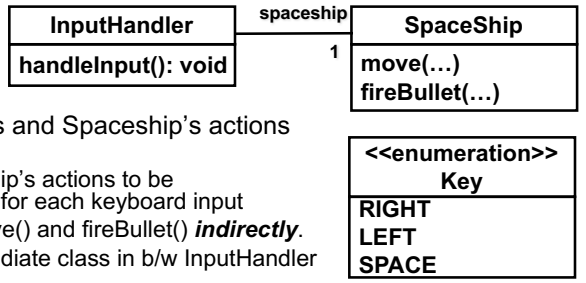
A Design Revision Needed



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

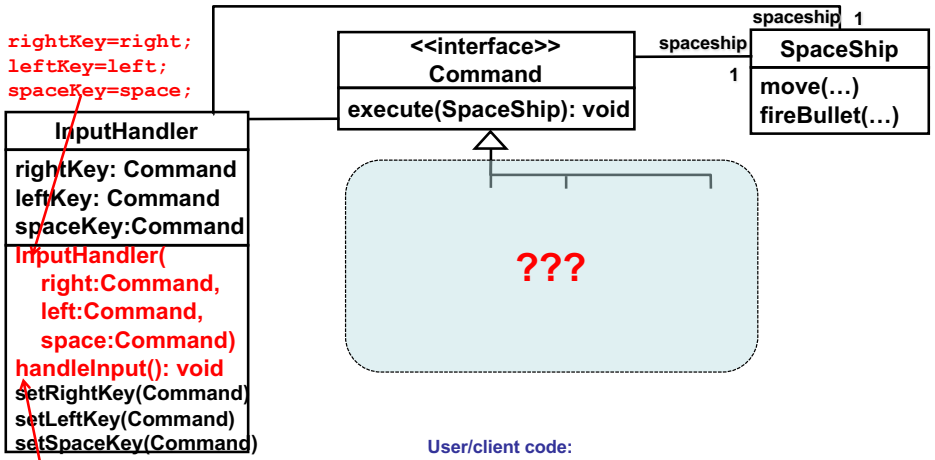
Handle inputs from the user
→ Call InputHandler.handleInput()

Game loop



- 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




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

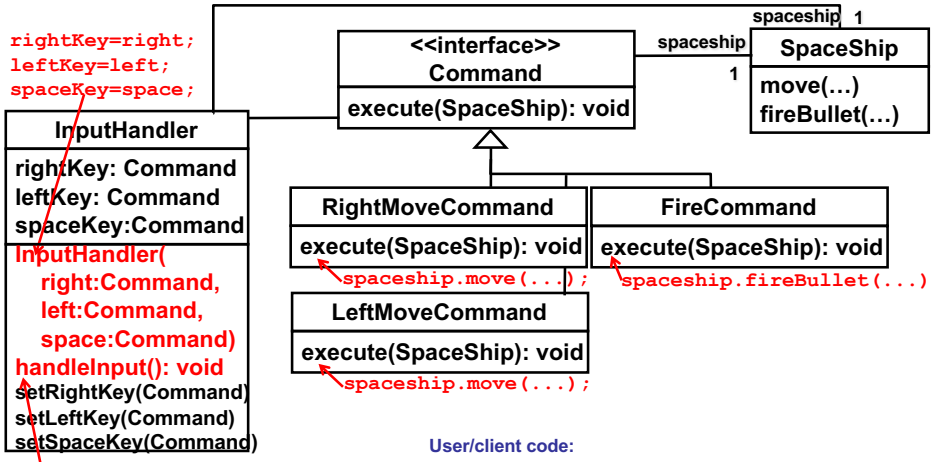
User/client code:

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


Quiz

- Complete InputHandler's handleInput() by replacing ??? with pseudo code.
- Define command classes that implements Command.
  - Complete the UML diagram by adding classes in the blue region.
  - Show how execute() should look like in each command class.

SpaceShip's Actions as Command Classes



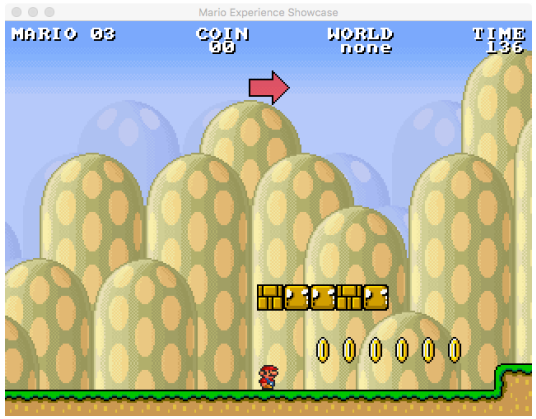

```
if(isPressed(Key.RIGHT))
  rightKey.execute(spaceship);
else if(isPressed(Key.LEFT))
  leftKey.execute(spaceship);
else if(isPressed(Key.SPACE))
  spaceKey.execute(spaceship);
```

User/client code:

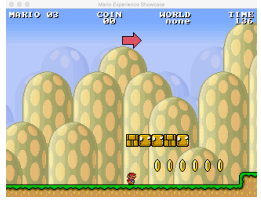
```
InputHandler ih =
  new InputHandler(new RightMoveCommand(...),
                   new LeftMoveCommand(...),
                   new FireCommand(...));
while(true){
  ih.handleInput();
  ... }
```


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()

Game loop

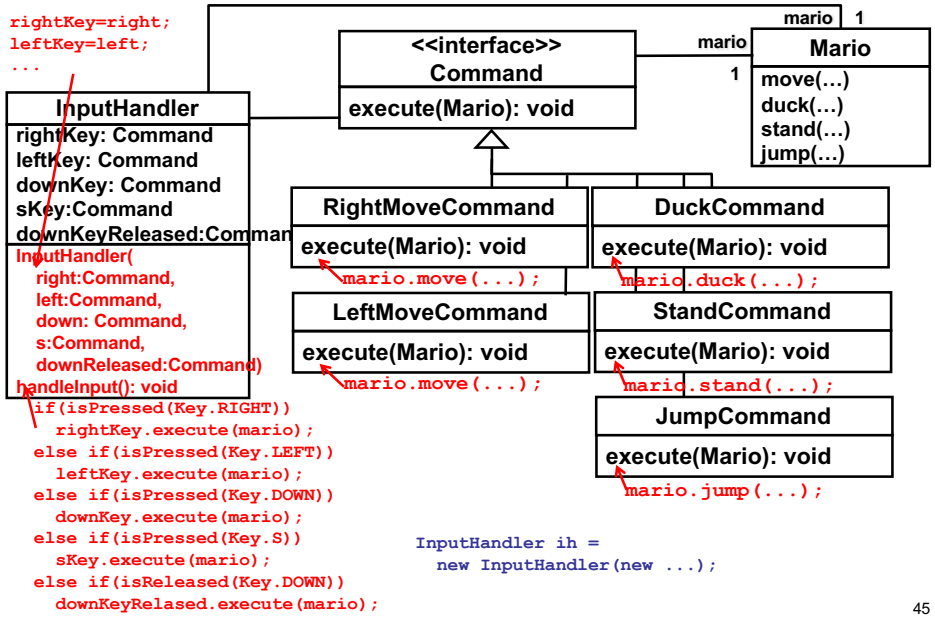
Update the scene (game world)

Render the game world

- 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)

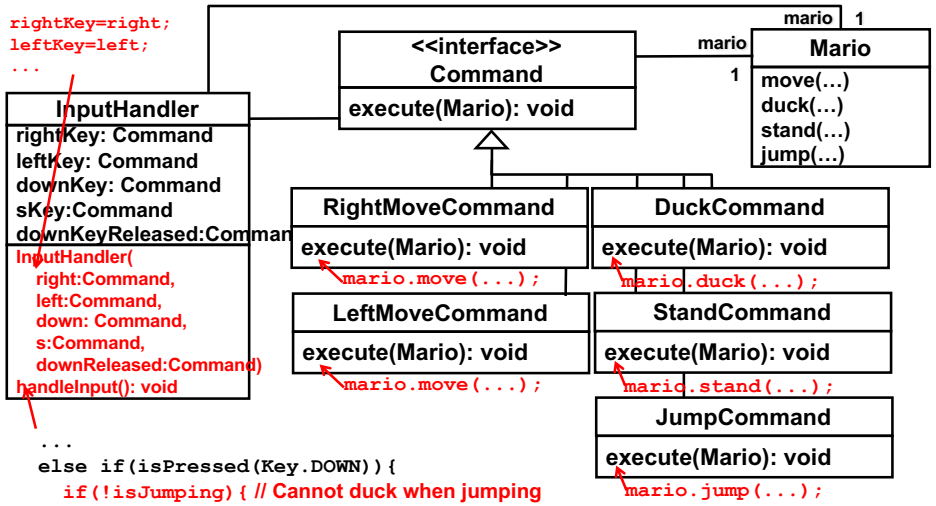
```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput();
  ...
}
```


Mario's Actions as Command Classes

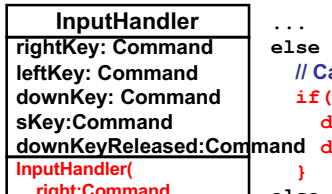



```
if(isPressed(Key.RIGHT))
  rightKey.execute(mario);
else if(isPressed(Key.LEFT))
  leftKey.execute(mario);
else if(isPressed(Key.DOWN))
  downKey.execute(mario);
else if(isPressed(Key.S))
  sKey.execute(mario);
else if(isReleased(Key.DOWN))
  downKeyRelased.execute(mario);
```

```
InputHandler ih =
  new InputHandler(new ...);
```



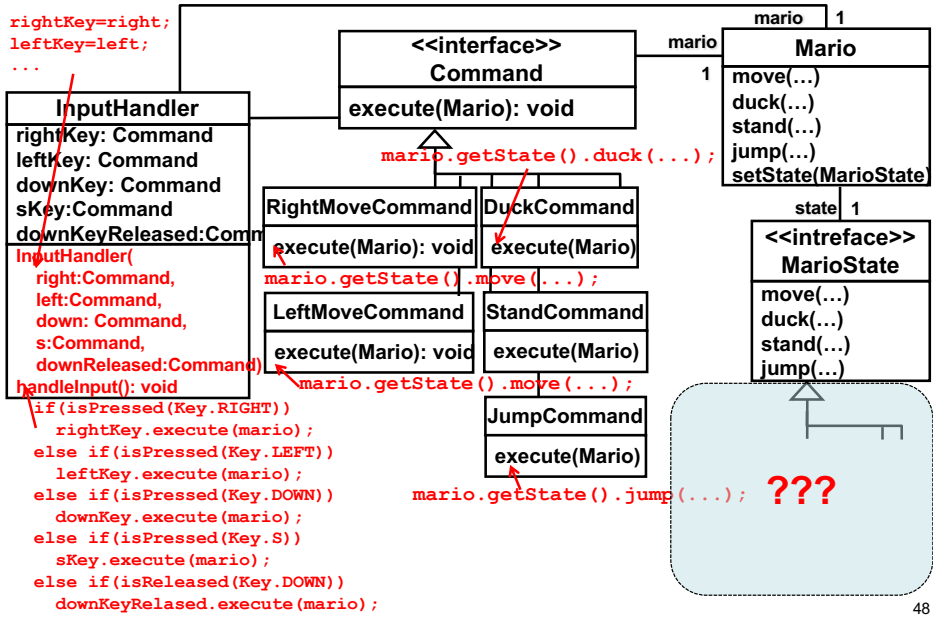

```
...
else if(isPressed(Key.DOWN)){
  if(!isJumping){ // Cannot duck when jumping
    downKey.execute(mario);
  }
else if(isPressed(Key.S)){
  if(!isJumping){  // Prevent "air jumping"
    isJumping=true;
    sKey.execute(mario);
  }
}
```


```
...
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);
  }
...
```

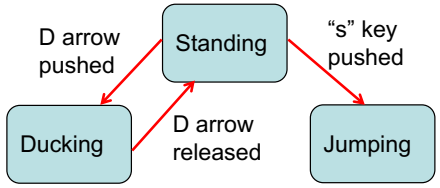
- The number of conditional branches increases and becomes less maintainable,
  - as the number of Mario's states and behaviors increases.
    - Mario moves faster when the "a" key and the right/left key are pushed at the same time.
    - Mario "dives" if the "down" key is pushed when jumping.

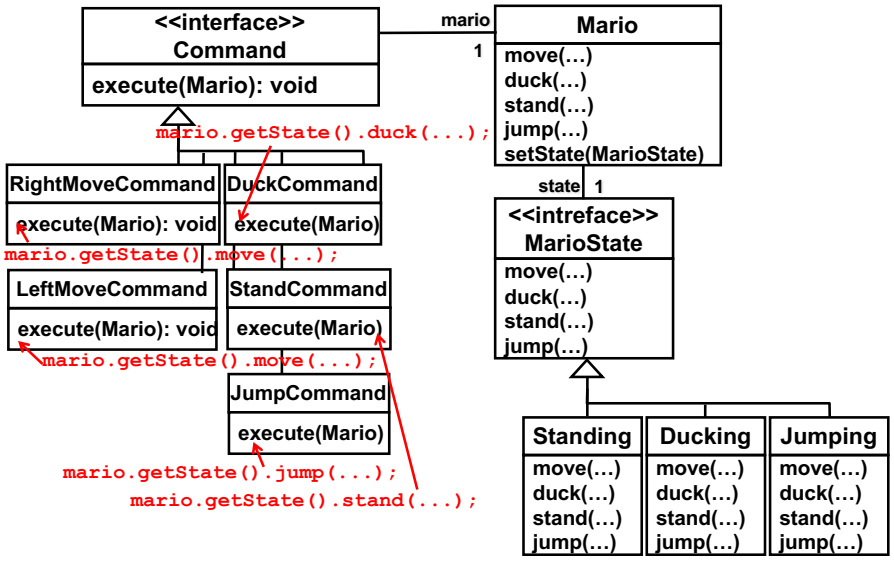
Define Mario's States as State Classes

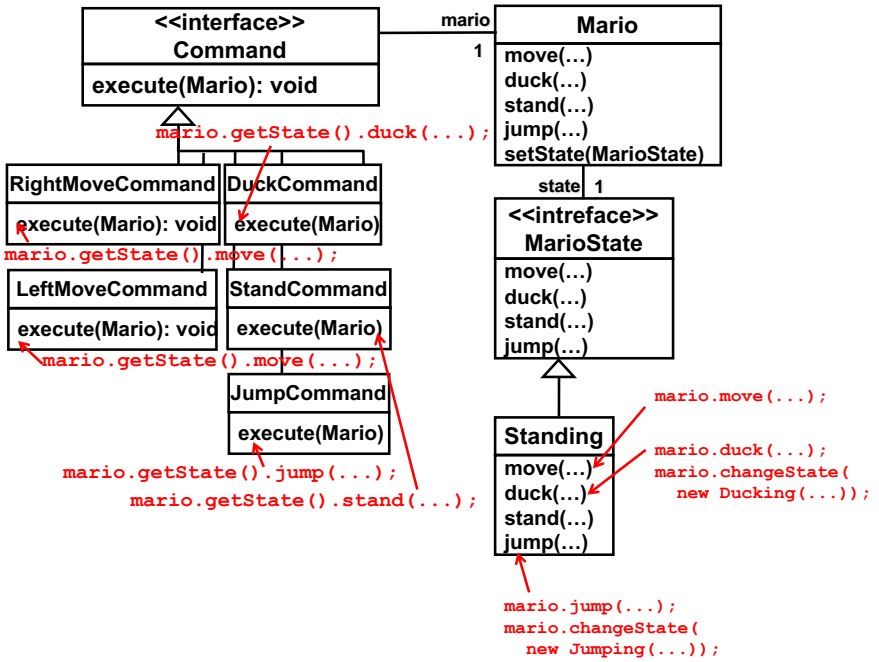


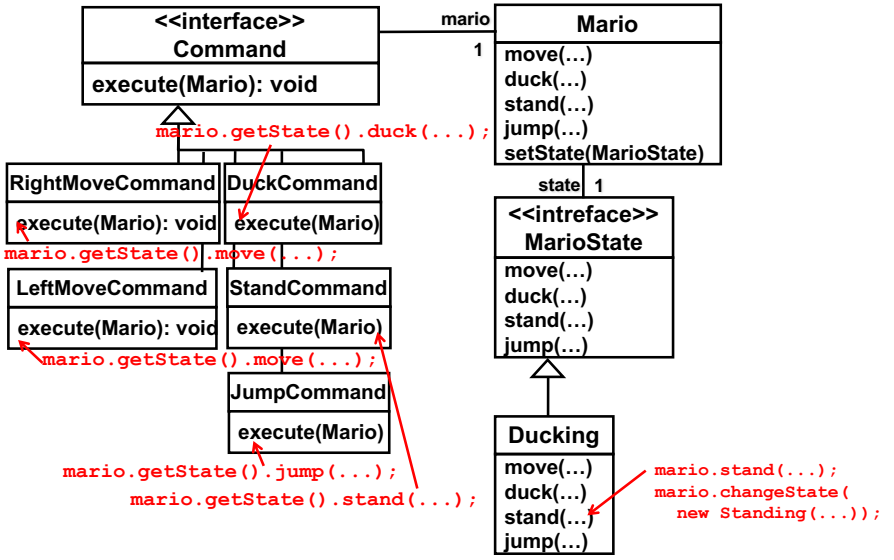
Quiz

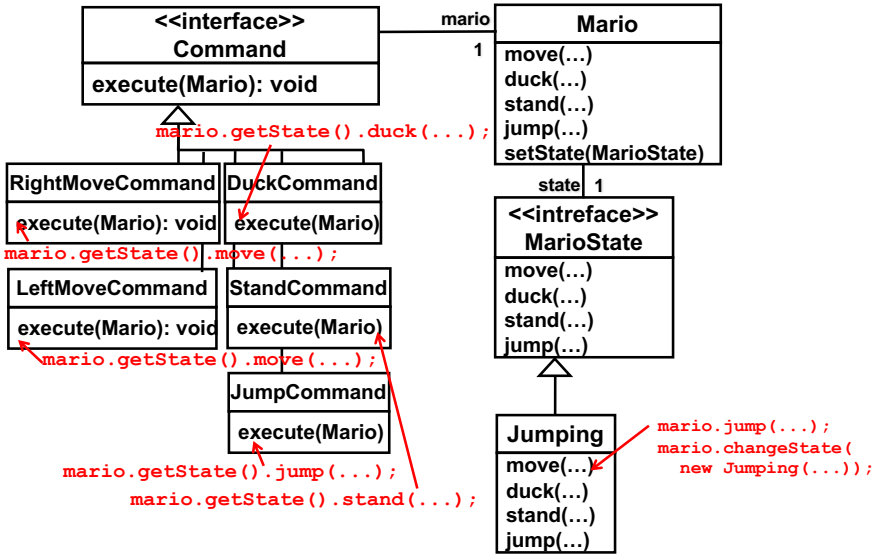
- Define state classes that implements MarioState.
  - Just consider 3 states: standing, ducking and jumping
  - Complete the UML diagram by adding classes in the blue region.
  - Show how 4 methods should like in each state class.




<<interface>>
Command
execute(Mario): void
mario
1
Mario
move(...)
duck(...)
stand(...)
jump(...)
setState(MarioState)
state
1
<<intreface>>
MarioState
move(...)
duck(...)
stand(...)
jump(...)
RightMoveCommand
execute(Mario): void
DuckCommand
execute(Mario)
LeftMoveCommand
execute(Mario): void
StandCommand
execute(Mario)
JumpCommand
execute(Mario)
Standing
move(...)
duck(...)
stand(...)
jump(...)
Ducking
move(...)
duck(...)
stand(...)
jump(...)
Jumping
move(...)
duck(...)
stand(...)
jump(...)
mario.getState().duck(...);
mario.getState().move(...);
mario.getState().move(...);
mario.getState().jump(...);
mario.getState().stand(...);


<<interface>>
Command
execute(Mario): void
mario
1
Mario
move(...)
duck(...)
stand(...)
jump(...)
setState(MarioState)
state
1
<<intreface>>
MarioState
move(...)
duck(...)
stand(...)
jump(...)
RightMoveCommand
execute(Mario): void
DuckCommand
execute(Mario)
LeftMoveCommand
execute(Mario): void
StandCommand
execute(Mario)
JumpCommand
execute(Mario)
Standing
move(...)
duck(...)
stand(...)
jump(...)
mario.getState().duck(...);
mario.getState().move(...);
mario.getState().move(...);
mario.getState().jump(...);
mario.getState().stand(...);
mario.move(...);
mario.duck(...);
mario.changeState(
new Ducking(...));
mario.jump(...);
mario.changeState(
new Jumping(...));


<<interface>>
Command
execute(Mario): void
mario
1
Mario
move(...)
duck(...)
stand(...)
jump(...)
setState(MarioState)
state
1
<<intreface>>
MarioState
move(...)
duck(...)
stand(...)
jump(...)
RightMoveCommand
execute(Mario): void
DuckCommand
execute(Mario)
LeftMoveCommand
execute(Mario): void
StandCommand
execute(Mario)
JumpCommand
execute(Mario)
Ducking
move(...)
duck(...)
stand(...)
jump(...)
mario.getState().duck(...);
mario.getState().move(...);
mario.getState().move(...);
mario.getState().jump(...);
mario.getState().stand(...);
mario.stand(...);
mario.changeState(
new Standing(...));


<<interface>>
Command
execute(Mario): void
mario
1
Mario
move(...)
duck(...)
stand(...)
jump(...)
setState(MarioState)
state
1
<<intreface>>
MarioState
move(...)
duck(...)
stand(...)
jump(...)
RightMoveCommand
execute(Mario): void
DuckCommand
execute(Mario)
LeftMoveCommand
execute(Mario): void
StandCommand
execute(Mario)
JumpCommand
execute(Mario)
Jumping
move(...)
duck(...)
stand(...)
jump(...)
mario.getState().duck(...);
mario.getState().move(...);
mario.getState().move(...);
mario.getState().jump(...);
mario.getState().stand(...);
mario.jump(...);
mario.changeState(
new Jumping(...));