

Design Patterns

- Tested, proven and documented solutions for recurring design problems in given contexts.
- Each design pattern is structured as
 - Pattern name
 - Intent
 - Motivation
 - Applicability
 - Class structure
 - Participants
 - ...etc.

Design Patterns

1

Resources

- *Design Patterns: Elements of Reusable Object-Oriented Software*
 - Eric Gamma et al., Addison-Wesley
- *Head First Design Patterns*
 - Elizabeth Freeman et al., O'Reilly
- *Game Programming Patterns*
 - Robert Nystrom, Genever Benning
 - <http://gameprogrammingpatterns.com/introduction.html>
- Web
 - [http://en.wikipedia.org/wiki/Design_patterns_\(computer_science\)](http://en.wikipedia.org/wiki/Design_patterns_(computer_science))
 - http://sourcemaking.com/design_patterns

Benefits of Design Patterns

- Useful information source to learn and practice good designs
- Useful as a communication tool among developers
 - e.g., Recursion, collection (array, stack, queue, etc.), sorting, buffers, infinite loops

Recap: Brief History to OOD

- In good, old days... programs had no structures.
 - One dimensional code.
 - From the first line to the last line on a line-by-line basis.
 - “Go to” statements to control program flows.
 - Produced a lot of “spaghetti” code
 - » “Go to” statements considered harmful.
 - No notion of structures (or modularity)
 - Modularity: Making a chunk of code (module) self-contained and independent from the other code
 - Improve reusability and maintainability
 - » Higher reusability → higher productivity, less production costs
 - » Higher maintainability → higher productivity and quality, less maintenance costs

5

Modules in SD and OOD

- Modules in Structured Design (SD)
 - Structure = a set of variables (data fields)
 - Function = a block of code
- Modules in OOD
 - Class = a set of data fields and functions
 - Interface = a set of abstract functions
- Key design questions/challenges:
 - how to define modules
 - how to separate a module from others
 - how to let modules interact with each other

6

SD v.s. OOD

- OOD
 - Intends coarse-grained modularity
 - The size of each code chuck is often bigger.
 - Extensibility in mind in addition to reusability and maintainability
 - How easy (cost effective) to add and revise existing modules (classes and interfaces) to accommodate new/modified requirements.
 - How to make software more flexible/robust against changes in the future.
 - How to gain reusability, maintainability and extensibility?
 - Design patterns show good examples.

7

Suggested Read

- Chapter 1 (Introduction) of *Game Programming Patterns*
 - <http://gameprogrammingpatterns.com/introduction.html>

8

Static Factory Method

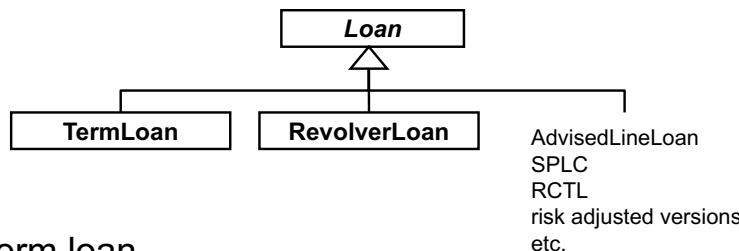
- Intent
 - Define a “communicable” method to instantiate a class
 - Constructors are the methods to instantiate a class.
 - Static factory methods are more “communicable” (or easier to use) than constructors.
- Benefits
 - Static factory methods have names (!).
 - Improve code maintainability.
 - The name can explicitly tell what object to be returned.
 - Client code gets easier to understand.

9

10

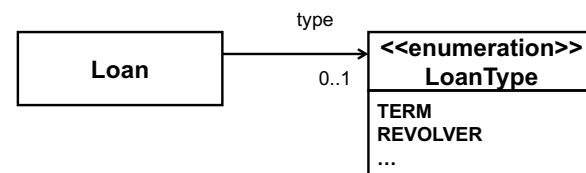
Static Factory Method

Recap: This Design is not Good.



- Term loan
 - Must be fully paid by its maturity date.
- Revolver (e.g. credit card)
 - With a spending limit and expiry date
- Dynamic class change problem
 - A revolver can transform into a term loan when the revolver expires.

Enumeration-based Design



- A class inheritance should not be used here.

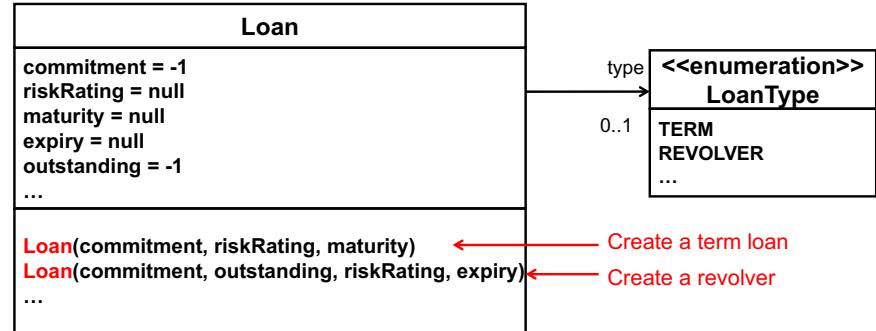
11

12



- Different loans need different sets of data to be set up.
 - A term loan needs commitment, risk rating and maturity date.
 - A revolver needs commitment, outstanding debt, risk rating and expiry date.
- The constructor is hard to use and error-prone.
- Client code is hard to understand.
 - `Loan l1 = new Loan(100, -1, 0.9, new Date(...), null);
l1.setLoanType(LoanType.TERM);`
 - `Loan l2 = new Loan(100, 0, 0.7, null, new Date(...));
l2.setLoanType(LoanType.REVOLVER);`

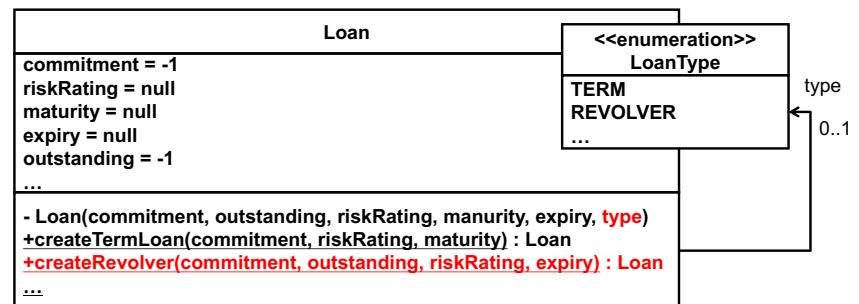
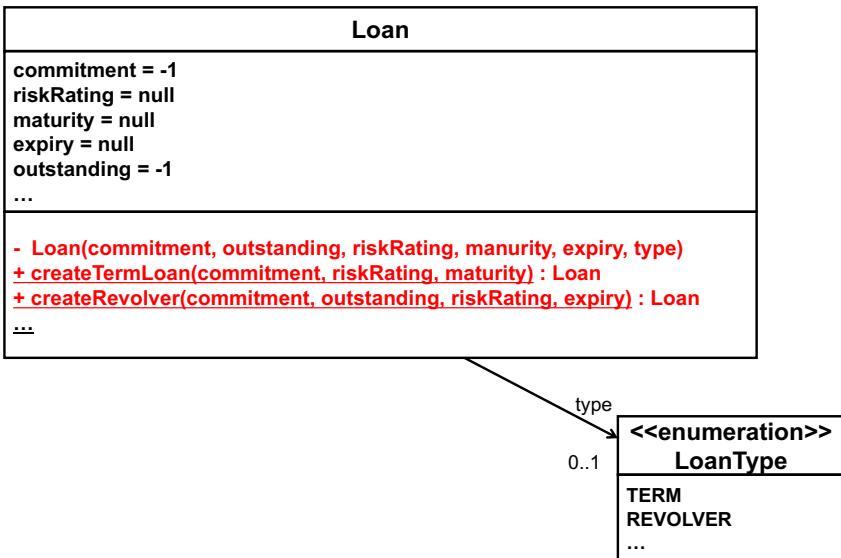
13



- Multiple constructors to create different types of loans.
- They are less error-prone, but still hard to use.
- Client code is still hard to understand.
 - `Loan l1 = new Loan(100, 0.9, new Date(...));
l1.setLoanType(LoanType.TERM);`
 - `Loan l2 = new Loan(100, 0, 0.7, new Date(...));
l2.setLoanType(LoanType.REVOLVER);`

14

Static Factory Methods

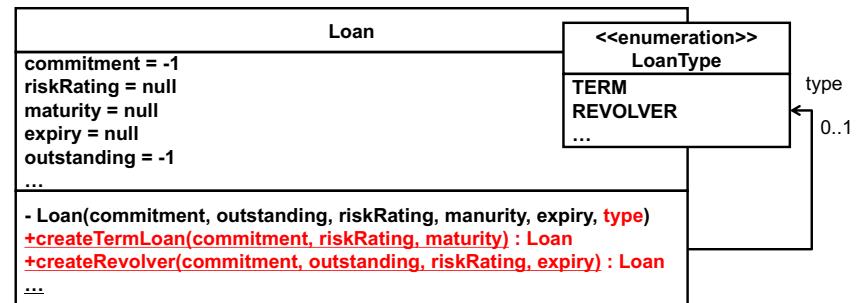


16

A Potential Issue

- Benefits of *Static Factory Method*

- Static factory methods have names (!).
- Improve code maintainability.
 - The name can explicitly tell what object to be returned.
 - Client code gets easier to understand.
- `Loan l1 = new Loan(100, 0.9, new Date(...));`
- `Loan l2 = Loan.createTermLoan(100, 0.9, new Date(...));`

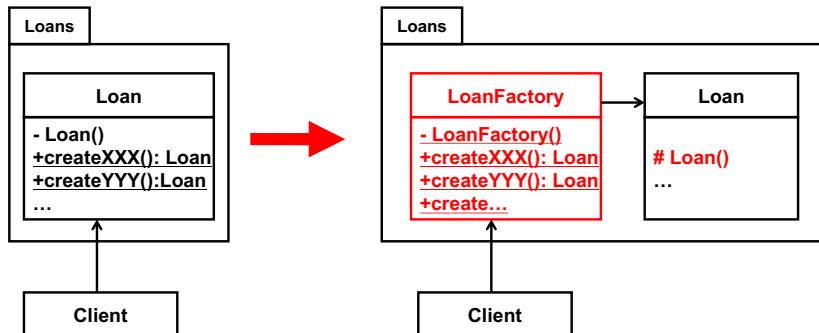


- Too many (static) factory methods in a class may obscure its primary responsibility/functionality.
 - They may dominate the class's public methods.
 - Loan may no longer strongly communicate it's primary (i.e., loan-related) responsibility/functionality.

17

18

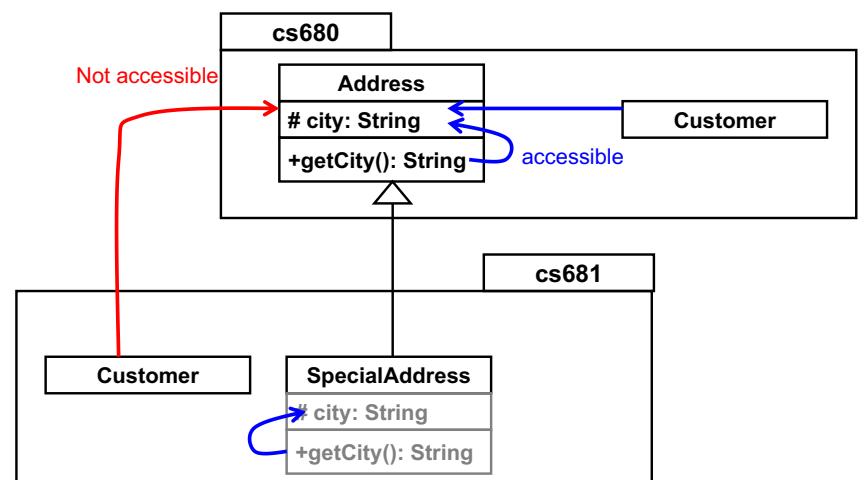
Alternative: Factory Class



- Factory class
 - A class that encapsulates static factory methods and isolate instantiation logic from other classes.
- Loan can now directly/strongly communicate its primary responsibility/functionality.

19

Recap: “Protected” Visibility



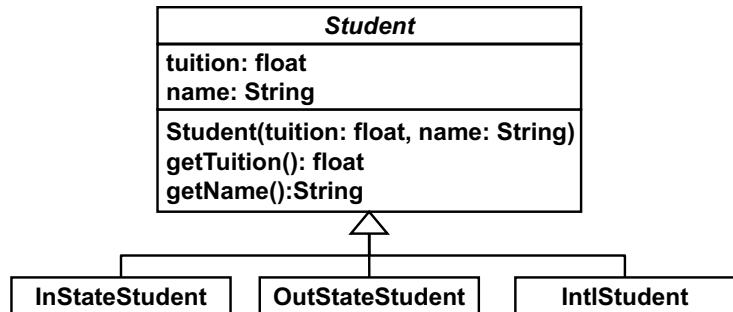
20

Design Tradeoffs

- Class inheritance
 - Pros: Straightforward.
 - Cons: Dynamic class change is hard (virtually impossible) to implement.
- Static factory method
 - Pros: Can avoid dynamic class change problem.
 - Cons: Potentially too many factory methods in a single class
- Factory class
 - Pros: Separates a class's primary logic and its instantiation logic
 - Cons: Non-factory classes in the same package can call protected constructors.
 - Could violate the encapsulation principle.
 - Consider an inner class (e.g., Loan as an inner class of LoanFactory)

21

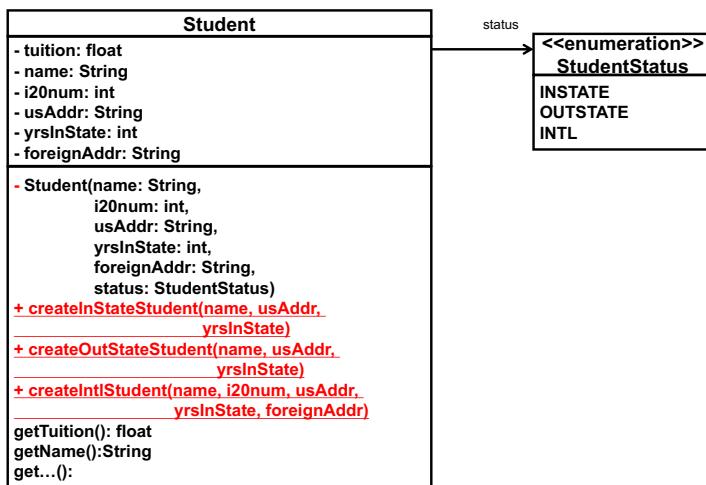
Recap: This Design is not Good.



- Alternative designs
 - Use an enumeration
 - Use *Static Factory Method* and an enumeration

22

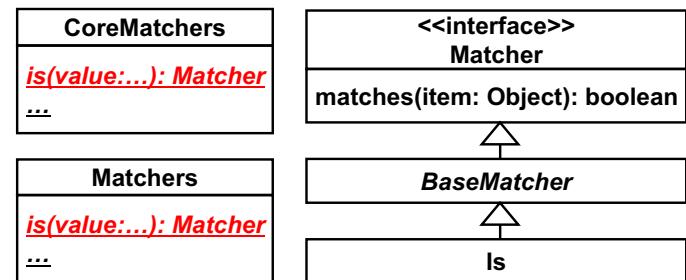
Design Improvement w/ Static Factory Method



23

Another Example: Matchers in hamcrest-all.jar

- org.hamcrest.CoreMatchers
- org.hamcrest.Matchers
 - Contains static methods, each returning a matcher object that performs matching logic.
 - Matchers is a superset of CoreMatchers.
- is() is a static factory method that creates an instance of Is.



25

Suggested Read

- Chapter 2 (Creating and destroying Objects) of *Effective Java*
 - Joshua Bloch, Addison
 - <http://bit.ly/2ydblp8>

Singleton Design Pattern

26

27

Singleton

- Intent
 - Guarantee that a class has only one instance.
- ```
public class Singleton{
 private Singleton() {};
 private static Singleton instance = null;

 public static Singleton getInstance() {
 if(instance==null)
 instance = new Singleton ();
 return instance;
 }
}
```
- You should not define public constructors.
- If you do not define constructors...

- ```
Singleton instance = Singleton.getInstance();
instance.hashCode();
Singleton instance = Singleton.getInstance();
instance.hashCode();
```
- hashCode() returns a unique ID for each instance.
 - Different instances of a class have different IDs.
- *Singleton* is an application of *Static Factory Method*.
 - getInstance() is a static factory method.
 - *Singleton* focuses on a requirement to have a class keep only one instance.

28

29

What Can be a Singleton?

- Object pools
- Logger
- Plugin manager
- Access counter
- Game loop
- ..., etc.

30

Alternative Implementation with Null Checking API in Java 7

- java.util.Objects, extending java.lang.Object
 - A utility class (i.e., a set of static methods) for the instances of java.lang.Object and its subclasses.
 - ```
- class Foo{
 private String str;
 public Foo() (String str){
 this.str = Objects.requireNonNull(str);
 }
}
```
  - `requireNonNull()` throws a `NullPointerException` if `str==null`. Otherwise, it simply returns `str`.
  - ```
- class Foo{
    private String str;
    public Foo() (String str){
        this.str = Objects.requireNonNull(
            str, "str must be non-null!!!!");
    }
}
```
 - `requireNonNull()` can accept an error message, which is to be contained in a `NullPointerException`.

34

Singleton Implementation with Objects.requireNonNull()

- Traditional null checking
 - ```
- if(str == null)
 throw new NullPointerException();
this.str = str;
```
- With `Objects.requireNonNull()`
  - `this.str = Objects.requireNonNull(str);`
- Can eliminate an explicit conditional statement and make code simpler.

35

```
• public class SingletonNullCheckingJava7{
 private SingletonNullCheckingJava7() {};
 private static SingletonNullCheckingJava7 instance = null;

 public static SingletonNullCheckingJava7 getInstance(){
 try{
 return Objects.requireNonNull(instance);
 }
 catch(NullPointerException ex){
 instance = new SingletonNullCheckingJava7();
 return instance;
 }
 }

• public class Singleton{
 private Singleton() {};
 private static Singleton instance = null;

 public static Singleton getInstance(){
 if(instance==null)
 instance = new Singleton ();
 return instance;
 }
}
```

36

## State

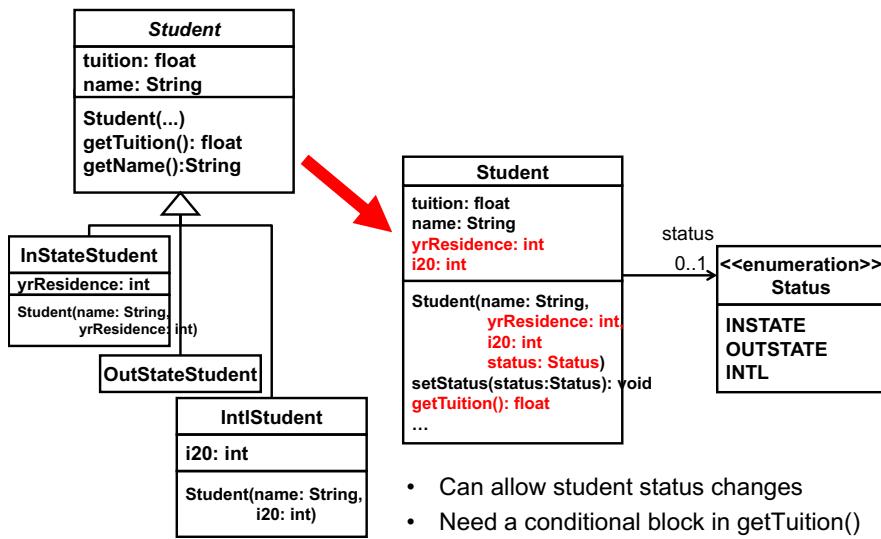
- Intent
  - Allow an object to change its behavior according to its state.

## State Design Pattern

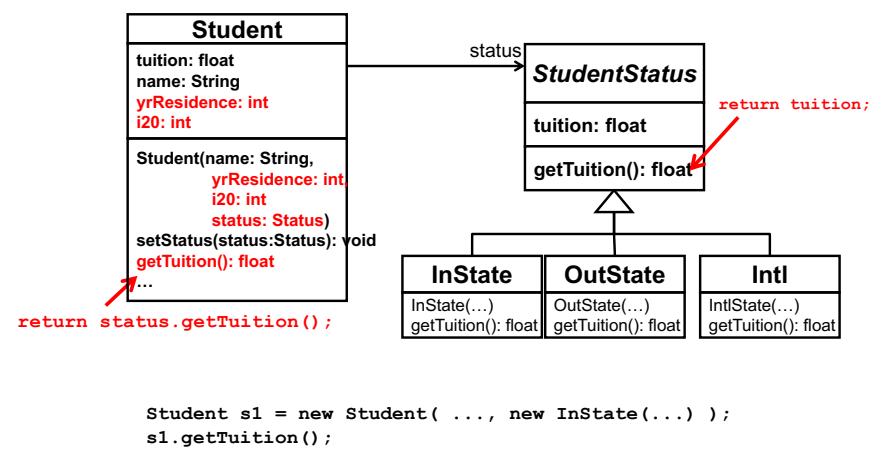
41

42

## Eliminating Class Inheritance



## Design Improvement with State



```

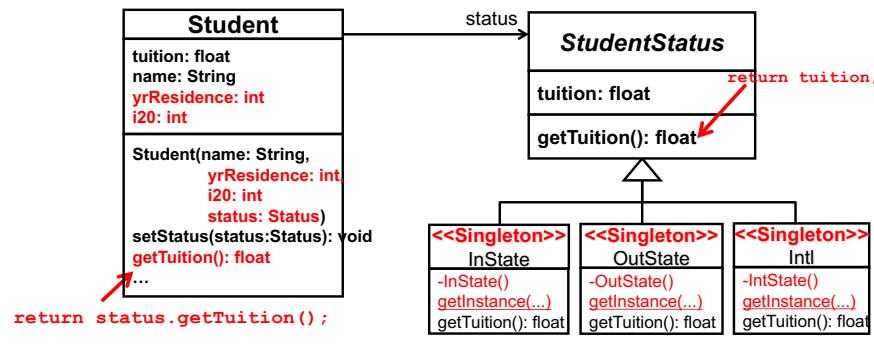
Student s1 = new Student(..., new InState(...));
s1.getTuition();

```

43

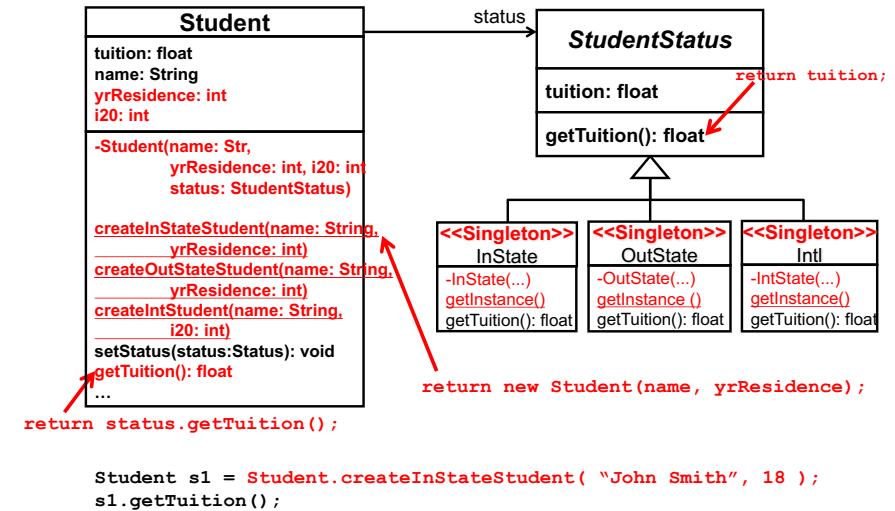
44

## State Classes as Singleton



45

## Adding Static Factory Methods



46

## Another Example: DVD Player

- Imagine you implement a firmware of DVD players
  - Focus on a player's **behaviors** upon **events**.
  - Events
    - The “Open/Close” button is pushed.
    - The “Play” button is pushed.
    - The “Stop” button is pushed.
  - The player behaves upon a certain event differently depending on its current state.
    - State-dependent behaviors



47

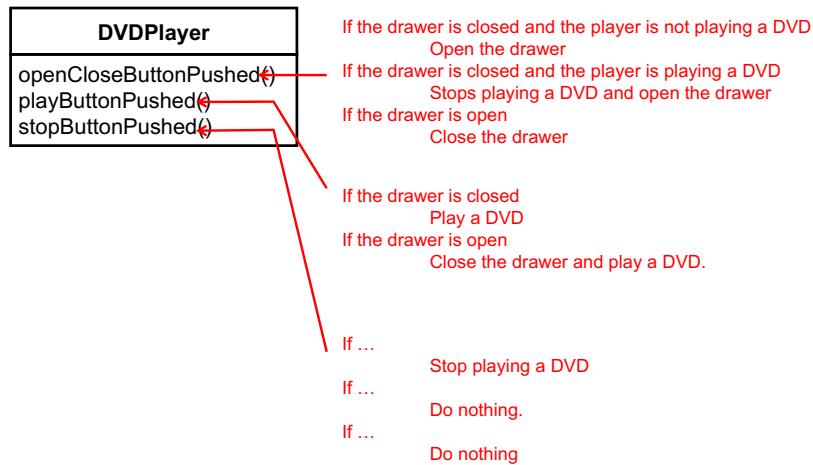
## State-Dependent Behaviors

- When the “open/close” button pushed,
  - Opens the drawer
    - If the drawer is closed and the player is not playing a DVD.
    - Stops playing a DVD and opens the drawer
      - if the drawer is closed and the player is playing a DVD.
    - Closes the drawer
      - if the drawer is open.
- When the “play” button pushed,
  - Plays a DVD
    - If the drawer is closed.
      - Displays an error message if the drawer is empty.
    - Closes the drawer and plays a DVD
      - If the drawer is open.
        - Displays an error message if the drawer is empty.
- When the “stop” button pushed
  - Stops playing a DVD
    - If the drawer is closed and the player is playing a DVD
  - Does nothing.
    - If the drawer is closed and the player is not playing a DVD.
  - Does nothing
    - If the drawer is open.

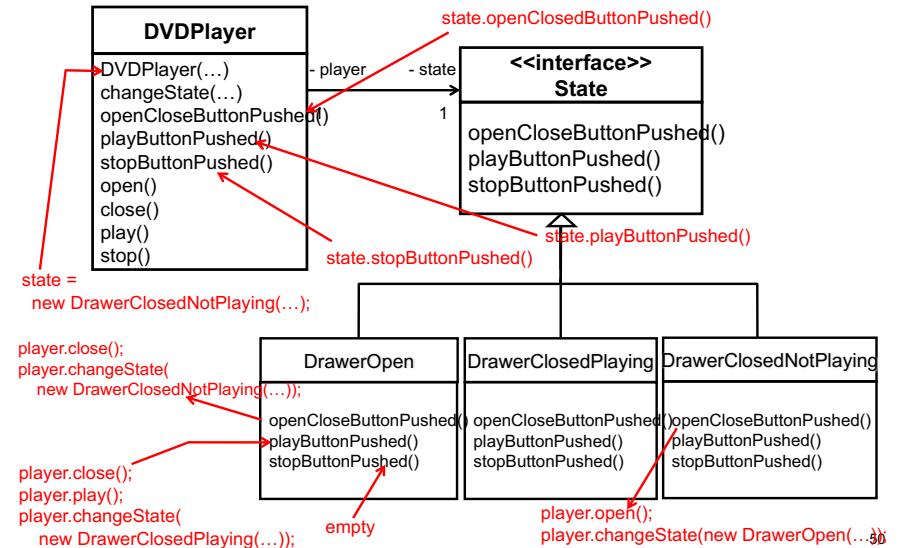


48

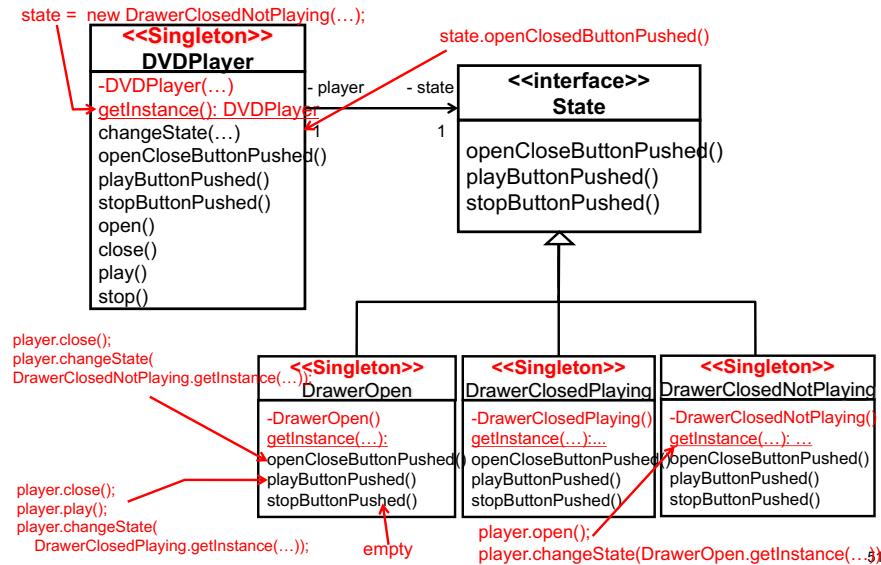
# Defining States as Classes



49



## State Classes as Singleton



- **Intent**
  - Allow an object to change its behavior according to its state.
  - Can implement state-dependent behaviors without conditionals.

## State

52

## HW 2

- Implement the design in Slide 51.
- Write and run at least one test case for every single method.
- **FIRM** Deadline: October 26 (Thu) midnight

53

- Use <batchtest> to have Ant search test classes in your project directory and run all of them (DVDPlayerTest, etc.).
  - <junit ...>
    - ...
    - <batchtest ...>
    - ...
    - </batchtest>
    - ...
    - </junit>
  - c.f. JUnit documentation

54

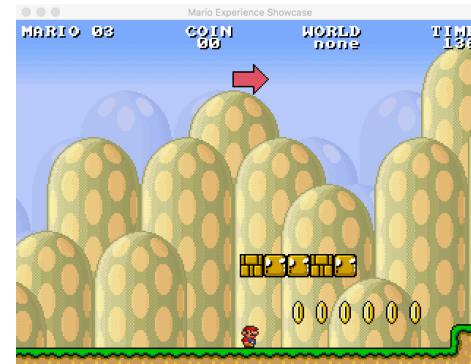
## HW Submission

- Submit me an archive file (.rar, .tar.gz, .7z, etc.) that contains
  - build.xml
  - “src” sub-directory
  - “test/src” sub-directory
- DO NOT send me binary files (.class and .jar files)
- Avoid a .zip file. (Gmail often removes attached .zip files.)
- Send the archive file to **umasscs680@gmail.com** from **your “primary” email address**
- Or, place it somewhere online (e.g. at G Drive) and email me a link to it.

55

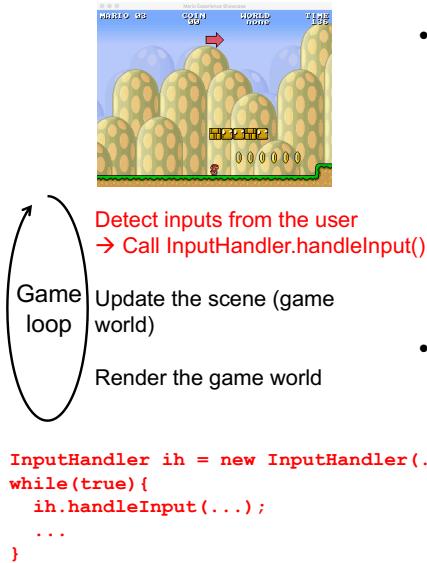
## One More Example: A Simple 2D Game

- Imagine a simple 2D game like Super Mario
  - <http://www.marioai.org/>
  - <http://www.platformersai.com>



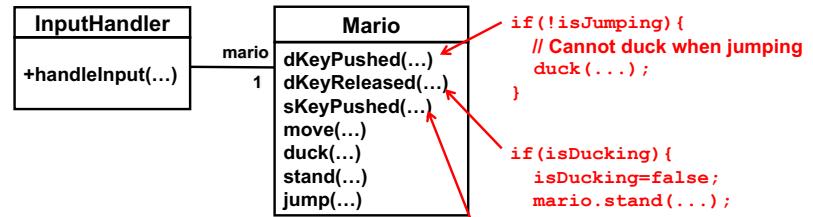
56

# Handling User Inputs



- 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)

57



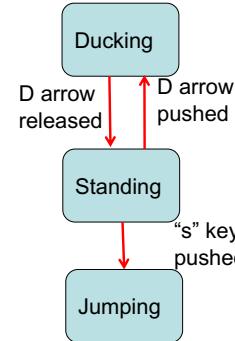
```

if(!isJumping) {
 // Cannot duck when jumping
 duck(...);
}

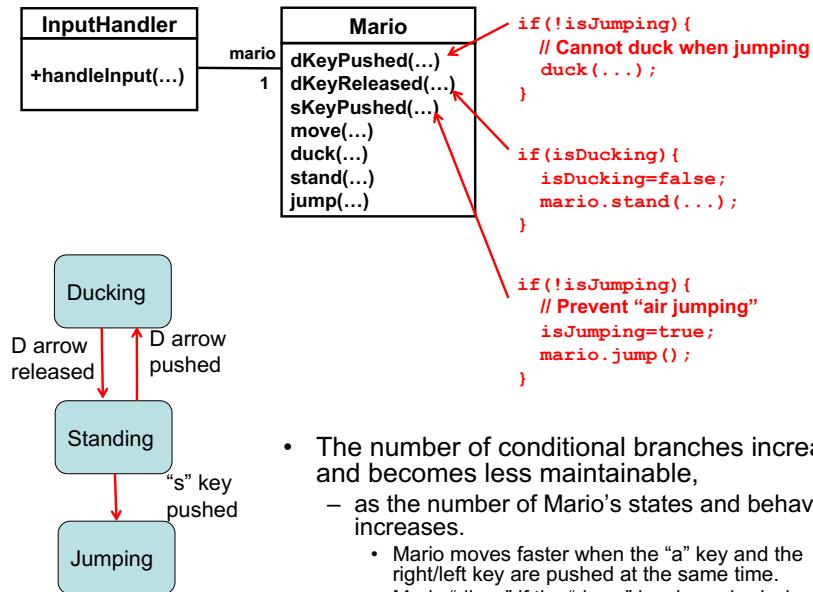
if(isDucking) {
 isDucking=false;
 mario.stand(...);
}

if(!isJumping) {
 // Prevent “air jumping”
 isJumping=true;
 mario.jump();
}

```

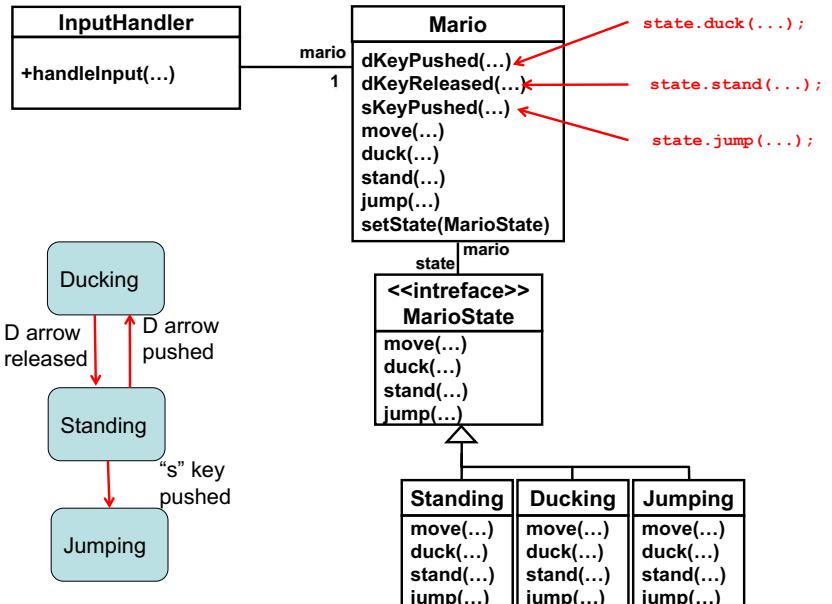


58



- 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.

59



```

state.duck(...);
state.stand(...);
state.jump(...);
setState(MarioState)

```

## HW 3

- Explain how each state class's methods should be implemented.
- **FIRM** Deadline: October 26 (Thu) midnight