Software Testing Part 2

test some code

Objectives

- Explain the purpose for a testing framework.
- Explain the purpose for the JUnit framework.
- Implement rudimentary test cases using an automated test framework.
- Explain the key aspects of a quality bug report
- Explain how bugs can be reported using a bug tracking system.

Not covered most likely.

Developing Test Conditions

- Description of the test
  - What are you trying to test for
- What initial conditions may need to be present before running the test?
- What are the input values for the test?
  - What parameter(s) will be passed into the routine
- What is the expected output from running the test
  - Will an exception be generated?
  - Will the code run properly?

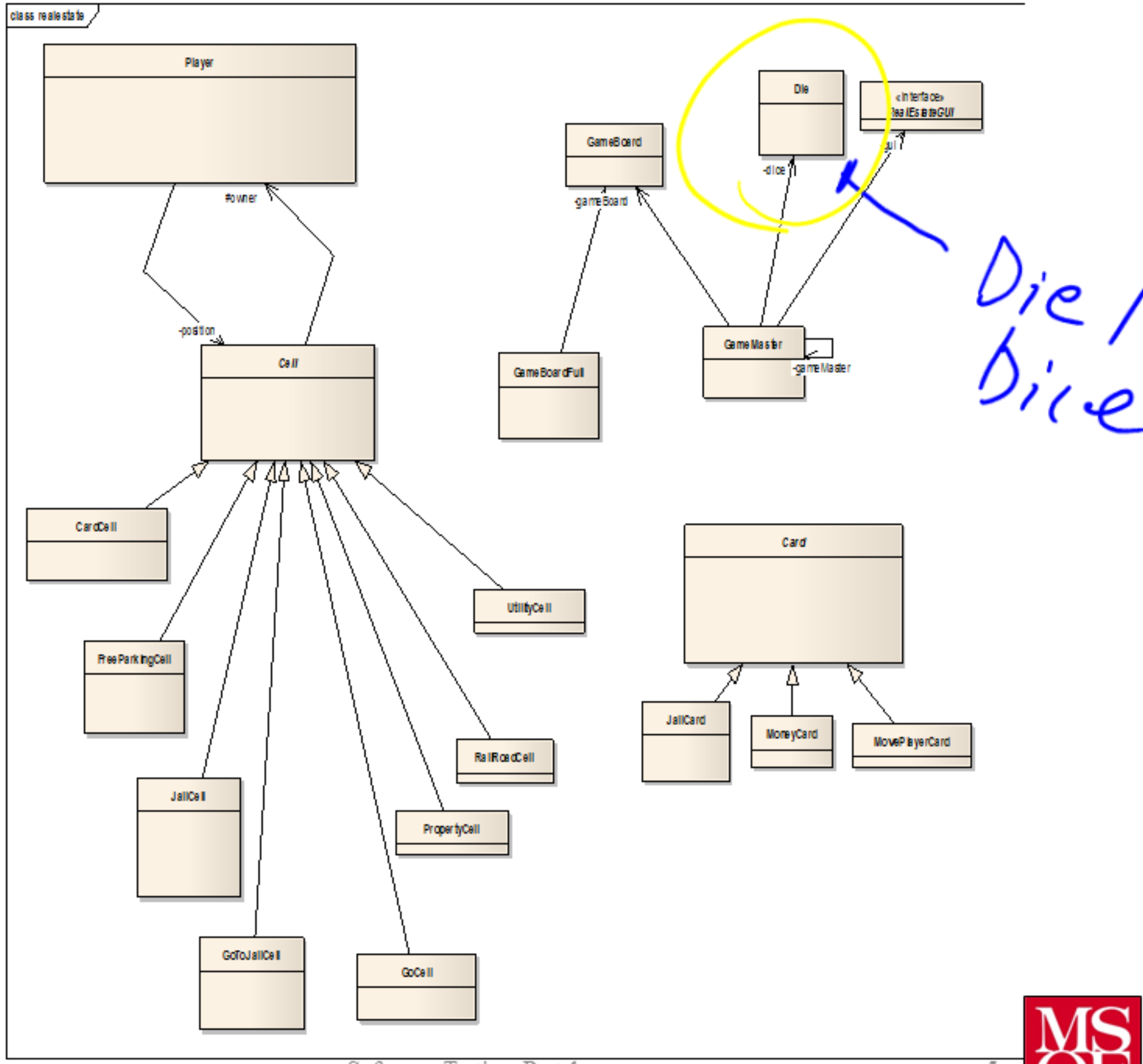
An Example

- The triangle Problem
  - A program accepts three integers, a, b, and c, as input representing the sides of a triangle
  - The output of the program is the type of triangle determined by the three sides
    - Isosceles, Scalene, Equilateral, or not a Triangle
  - Sides are between 1 and 200 units in length

- Write a set of test cases and expected results to fully test this program

Test Case #	Description of Test	Input Values	Expected Result
1	Equilateral Valid Triangle	1,1,1	Eq. Triangle
2	Side too long	1,1,200	Not a Triangle
3	Sides equal to others	1,2,3	Not a Triangle
	Zero Side	0,1,2	Not a triangle
		-1,2,3	Not a triangle
		2,2,3	ISOL Triangle
		2,3,4	Scalene
			⋮

Real Estate Example



Lets start thinking about how we would test this…

- We'll start by playing the game a bit…

In Class Exercise

- Go to the class website and download the jar file for the game real estate. Run the jar on your computer and test the program

What is wrong?

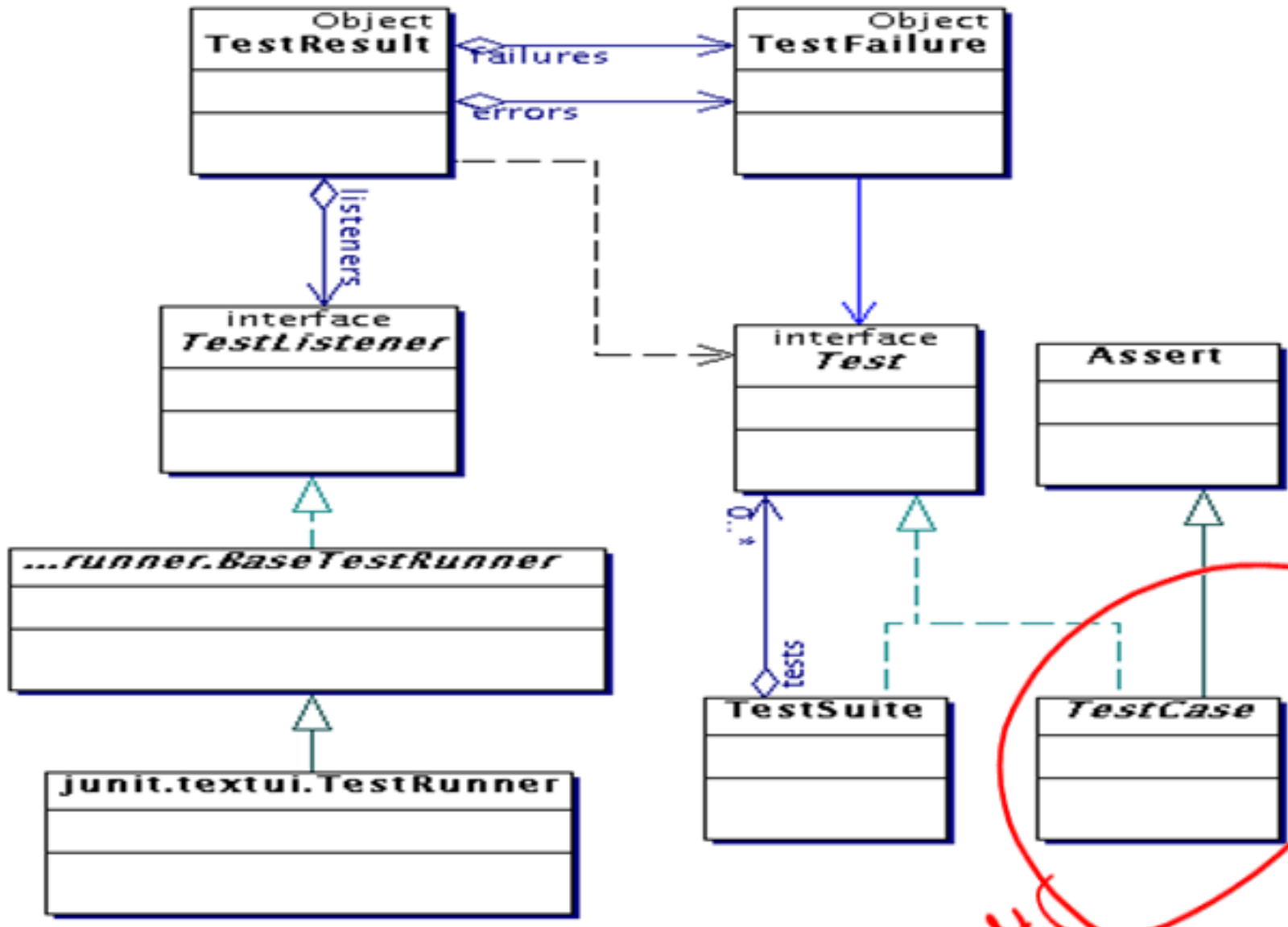
What is JUNit

- JUnit is an open source Java testing framework used to write and run repeatable tests.
- It is an instance of the xUnit architecture for unit testing frameworks.
- JUnit features include:
  - Assertions for testing expected results
  - Test fixtures for sharing common test data
  - Test suites for easily organizing and running tests
  - Graphical and textual test runners

Junit Design Objectives

- A simple framework that encourages developers to write unit tests.
  - Minimalist framework – essential features, easier to learn, more likely to be used, flexible
  - Test Cases & Test Results are objects
  - Patterns – high "density" of patterns around key abstractions : mature framework

The Framework of JUnit



Used for writing test

JUnit Terminology

- A test fixture sets up the data (both objects and primitives) that are needed to run tests
  - Example: If you are testing code that updates an employee record, you need an employee record to test it on
- A unit test is a test of a *single* class
- A test case tests the response of a single method to a particular set of inputs
- A test suite is a collection of test cases
- A test runner is software that runs tests and reports results
- An integration test is a test of how well classes work together
  - JUnit provides some limited support for integration tests

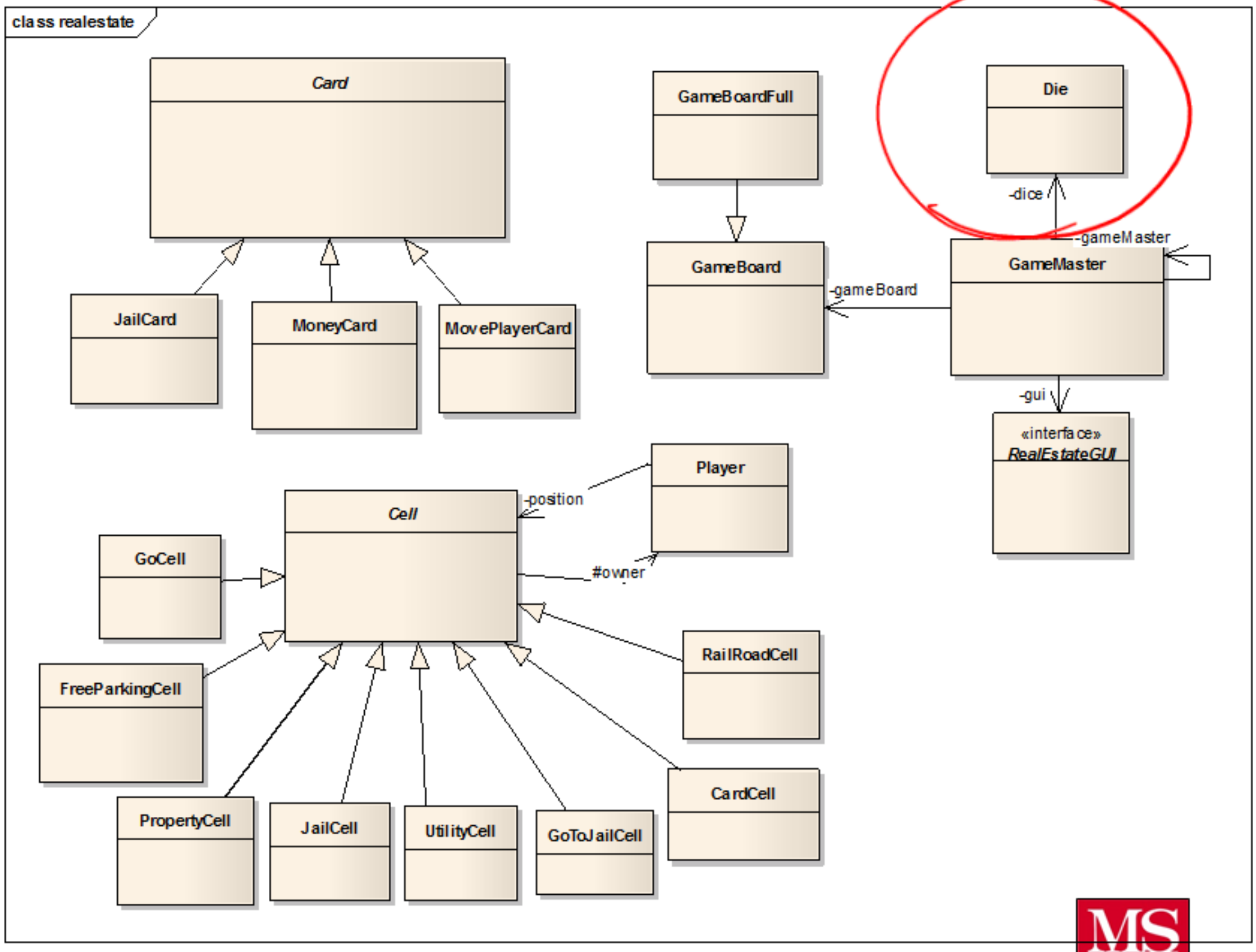
Assert

- assertEquals(expected, actual)
- assertEquals(message, expected, actual)
- assertEquals(expected, actual, delta)
- assertEquals(message, expected, actual, delta)
- assertFalse(condition)
- assertFalse(message, condition)
- Assert(Not)Null(object)
- Assert(Not)Null(message, object)
- Assert(Not)Same(expected, actual)
- Assert(Not)Same(message, expected, actual)
- assertTrue(condition)
- assertTrue(message, condition)

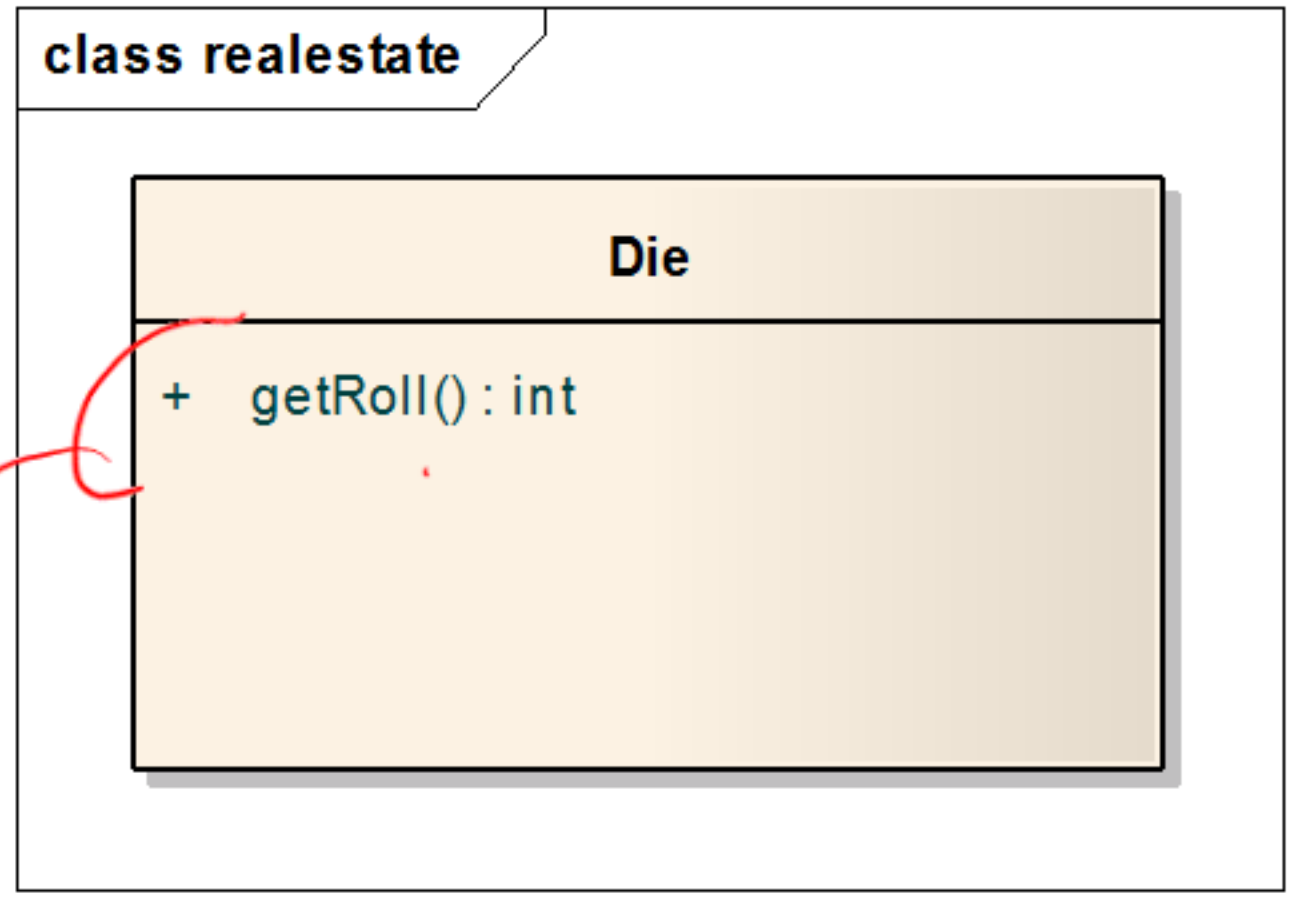
Other methods

- fail — Forces a test to fail if a problem occurs.
- setUp — Sets up a test environment
- tearDown — Cleans up afterwards

Lets do some testing...



Lets do some testing

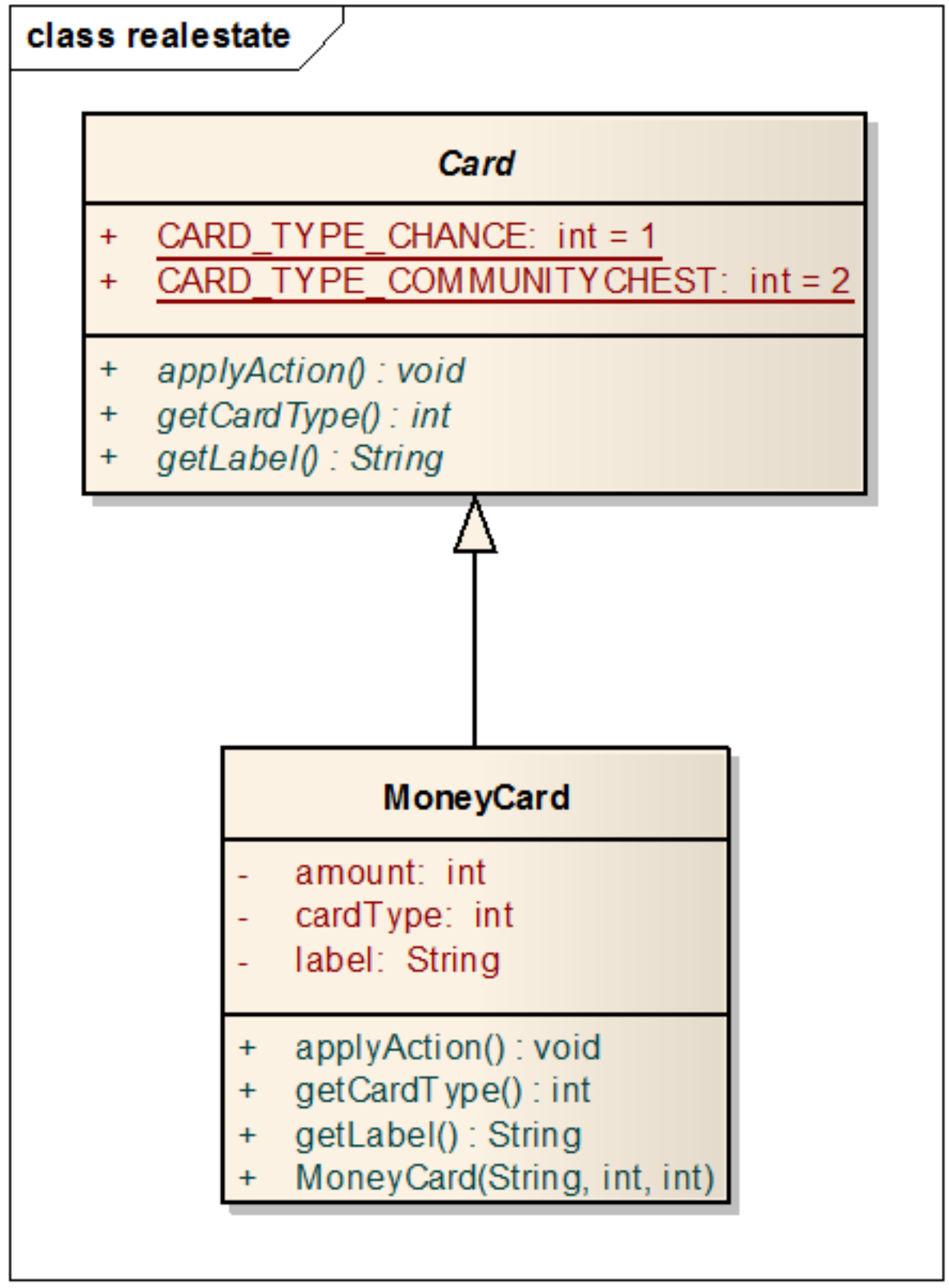


Always return a #
between 1 and 6 sides
on die

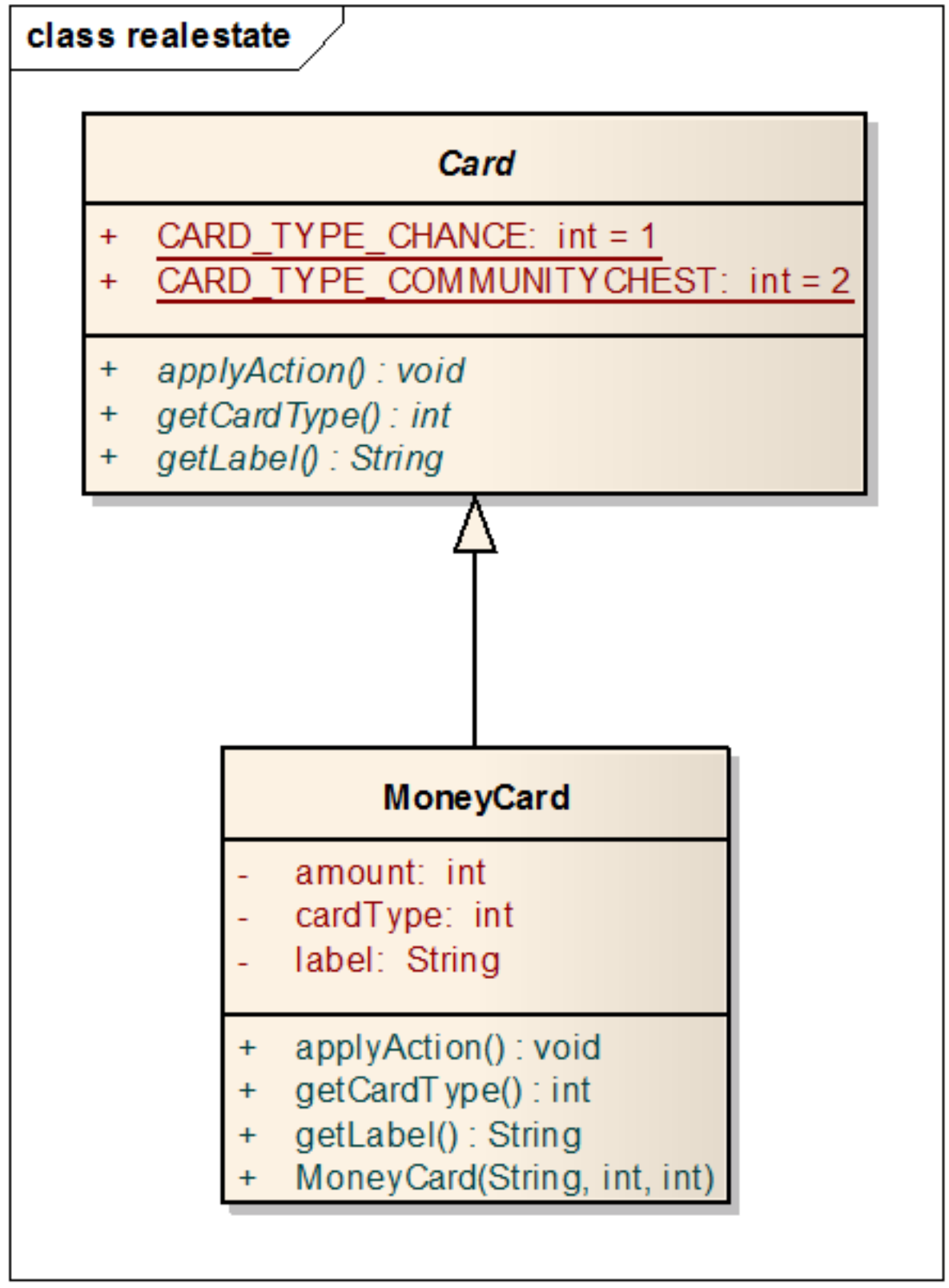
Game Master


class realestate
GameMaster
-gameMaster
- dice: Die ([])
- gameBoard: GameBoard
- gameMaster: GameMaster
- gui: RealEstateGUI
- initAmountOfMoney: int
+ MAX_PLAYER: int = 8 {readOnly}
- players: ArrayList = new ArrayList()
- testMode: boolean
- turn: int = 0
- utilDiceRoll: int
+ btnBuyHouseClicked() : void
+ btnDrawCardClicked() : Card
+ btnEndTurnClicked() : void
+ btnGetOutOfJailClicked() : void
+ btnPurchasePropertyClicked() : void
+ btnRollDiceClicked() : void
+ btnTradeClicked() : void
+ checkForWinner() : void
+ completeTrade(TradeDeal) : void
+ drawCCCard() : Card
+ drawChanceCard() : Card
+ GameMaster()
+ getCurrentPlayer() : Player
+ getCurrentPlayerIndex() : int
+ getGameBoard() : GameBoard
+ getGUI() : RealEstateGUI
+ getInitAmountOfMoney() : int
+ getNumberOfPlayers() : int
+ getNumberOfSellers() : int
+ getPlayer(int) : Player
+ getPlayerIndex(Player) : int
+ getSellerList() : ArrayList
+ getTurn() : int
+ getUtilDiceRoll() : int
+ instance() : GameMaster
+ movePlayer(int, int) : void
+ movePlayer(Player, int) : void
+ playerMoved(Player) : void
+ reset() : void
+ rollDice() : int[]
+ sendToJail(Player) : void
- setAllButtonEnabled(boolean) : void
+ setGameBoard(GameBoard) : void
+ setGUI(RealEstateGUI) : void
+ setInitAmountOfMoney(int) : void
+ setNumberOfPlayers(int) : void
+ setTestMode(boolean) : void
+ setUtilDiceRoll(int) : void
+ startGame() : void
+ switchTurn() : void
+ updateGUI() : void
+ utilRollDice() : void

MoneyCard


class realestate
Card
+ CARD_TYPE_CHANCE: int = 1
+ CARD_TYPE_COMMUNITYCHEST: int = 2
+ applyAction() : void
+ getCardType() : int
+ getLabel() : String
MoneyCard
- amount: int
- cardType: int
- label: String
+ applyAction() : void
+ getCardType() : int
+ getLabel() : String
+ MoneyCard(String, int, int)

MoneyCard


class realestate
Card
+ CARD_TYPE_CHANCE: int = 1
+ CARD_TYPE_COMMUNITYCHEST: int = 2
+ applyAction() : void
+ getCardType() : int
+ getLabel() : String
MoneyCard
- amount: int
- cardType: int
- label: String
+ applyAction() : void
+ getCardType() : int
+ getLabel() : String
+ MoneyCard(String, int, int)

Limitations Of JUnit

- JUnit is designed to call methods and compare the results they return against expected results
  - This works great for methods that *just* return results, but many methods have side effects
    - To test methods that do output, you have to capture the output
    - To test methods that change the state of the object, you have to have code that checks the state
  - It isn't easy to see how to unit test GUI code
- JUnit encourages a "functional" style, where most methods are called to compute a value, rather than to have side effects
  - This can actually be a good thing
  - Methods that *just* return results, without side effects (such as printing), are simpler, more general, and easier to reuse

What Is a Bug Report and Why Do We Write Them?

- Do We Write Them?
  - Bug report is a technical document
- Describes failure mode in system under test (SUT)
  - The only tangible "product" of testing
- Not a management problem escalation tool
  - "Build not delivered on time" is not a bug report summary
  - "Build 781 fails to install" is a bug report summary
- Written to increase product quality
  - Documents a specific quality problem quality of SUT
  - Communicates to developers

Bug Reports