— Amit's Codebase

All my codes

Archive

Design

Coming up with software design

February 14, 2014

Design

1 Comment

Disclaimer: This article is not a tutorial or *design-for-dummies* kind of article.

You are given the task of designing a software / feature, how would you go about designing it?

1. Clarify usecases

This is the most important and hence the first step. If you miss to come up with the exact asks (aka requirements), you may end up wasting time and money designing something which is far far away from what is expected. You may also miss out on some very important functionalities which may be required, which in turn may result in overhaul of an almost final design. So feel free to ask and cross question the customer (she may be product manager, interviewer, client, etc.).

3. Document the usecases and setup a timelines

Documentation helps in tracking the requirements and commitments. Despite all clarifications, accept the fact that the requirements may change *OR* new ones may come up. Hence, it is a good idea to keep some buffer time to accommodate these changes (if possible and reasonable).

Remember – that which has no deadline, is never completed.

3. UML design

So you have investigated all the usecases, documented them, shared timelines. Now you get to do the actual designing. Do not think in terms of code / pseudocode at this stage!

> To begin with, think about the real life problem that you are trying to solve through software. This real life problem will have certain entities (aka objects), incorporate their representation into your UML design first of all. This helps as these are the most obvious parts of the design and hence are easy to translate.
> Next, mark all relationships (such as composition, aggregation, generalization, etc.) between the most obvious objects in your design.
> Criticize your design. Seriously! Try to look for potential issues in the design as if you are your nemesis. As you spitefully review your design, you may realize some potential abstractions that you missed out on. Put them in place. Now!
> Introduce interfaces where required. Remember – interfaces are contracts of functionality. If you feel that more implementations of a component may come up in future, ensure that you expose an interface for it. Is a class taking up a role? Expose interface for that role. This ensures that if another class takes up this role (aka implements this interface) in future, the developer is not forced to introduce dependency of the new class in client.

That's it. Now have the design reviewed, make changes as per design review discussion and start coding

Object oriented design for chess game

February 4, 2014

Design

Object Oriented design represents the real world objects in code.
Think of how you play chess:

> A Board '***is***' collection of blocks (ΣBlock)
> A Player '***is***' collection of pieces (Player = ΣPiece)
> A Game has :
> one board

1 Comment

one black player
one white player

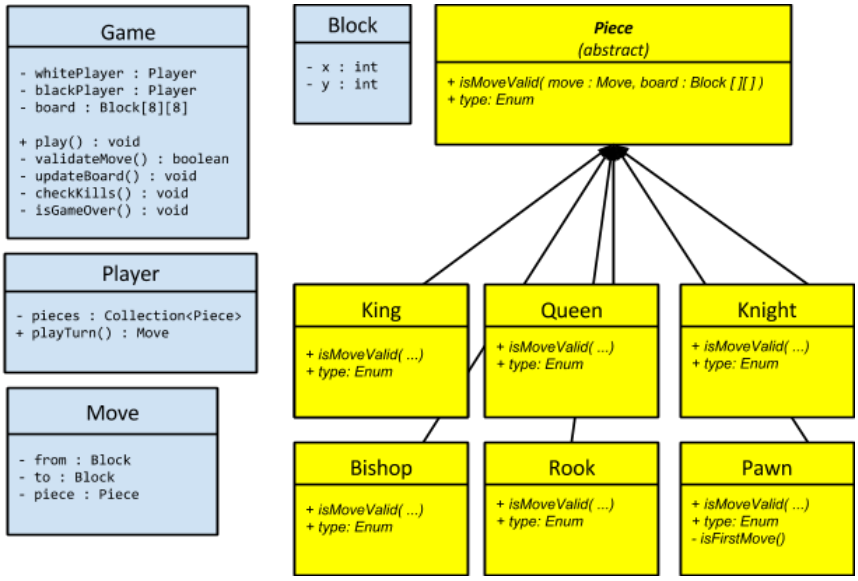
When making a move a Player chooses a Piece Depending on type of piece, the Player checks valid moves.Which means that validity of move is :

tightly coupled to type of piece
is a property of piece

When thinking about valid moves, the player 'is aware of'

the board
location of all the pieces on the board

Last point implies that the move validation routine should have access location of all the pieces on the board
On the board, block '***contains***' a piece



Gameplay

1. A Player gets a turn [`Game.play()`]
2. Player makes a plays a turn and returns the played 'move' to Game [`Player.playTurn()`] (selected piece, from block and to block are encapsulated in a data-type 'Move')
3. Game delegates the move validation processing to the moved Piece instance, passing destination-block and board instances. (`Piece.isValidMove(...)`: boolean)(Board instance is required to check is a piece lies in path of the move)
4. Piece calculates validity of the move and returns a boolean to Game
5. If move is valid, GameController
   A. updates location of Piece on board (`Game.updateBoard(...)`)
   B. checks if move has performed any kills (`Game.checkKills(...)`)
   C. checks if any of game terminations conditions are valid (`Game.isGameOver(...)`)

Design the Nokia Snake game

January 26, 2014

Design, Programming Problems

1 Comment

```
******************************
* Design the Nokia Snake game *
******************************

Classes
=======

    Class: SnakeGame
    Member Vars:
        * arena : Arena
        * snake : Queue
```

My Personal Blog
Programming Problems

Search

Blog Log !

February 2014 (4)
January 2014 (7)
December 2013 (1)
December 2012 (1)
August 2012 (1)
April 2012 (3)
March 2012 (1)
May 2011 (1)
March 2011 (1)
February 2011 (1)
January 2011 (1)
July 2010 (1)
June 2009 (6)
April 2009 (1)
October 2008 (1)
September 2008 (1)
August 2008 (2)
July 2008 (1)
April 2008 (1)
December 2007 (1)
September 2007 (2)
April 2007 (2)
September 2006 (1)
August 2006 (4)

```
        * current : CoOrdinate
        * food : CoOrdinate
    Methods:
        * init()
        * move()
        * grow()
        * generateFood()
        * isFoodEaten()
        * isCrash()
        * isBite()
        * terminate()
    ----------------
    Class: Arena
    Member Vars:
        * minX
        * maxX
        * minY
        * maxY
    ----------------
    Class: CoOrdinate
    Member Vars:
        * x
        * y
    ----------------

Pseudo Code:
===========
Class SnakeGame

    var Arena arena      // definition of arena
    var Queue snake      // body of snake
    var CoOrdinate cur   // current co-ordinates of head
    var CoOrdinate food  // food

    func init():
        arena = Arena(0,0,X,Y)
        snake = new Queue()
        head = CoOrdinate(arena.minX + 1, arena.minY + 1)
        snake.add(curr)
        food = CoOrdinate(arena.maxX/2, arena.maxY/2)

    func move(dir):
        snake.pop() // remove tail pixel
        switch(dir) // take action based on direction
            case up:
                head.y++
                grow()
                if (isFoodEaten())
                    head.y++
                    grow()
            case down:
                head.y--
                grow()
                if (isFoodEaten())
                    head.y--
                    grow()
            case left:
                head.x--
                grow()
                if (isFoodEaten())
                    head.x--
                    grow()
            case right:
                head.x++
                grow()
                if (isFoodEaten())
                    head.x++
                    grow()
```

```
    generateFood()

func grow():
    if(isBite() || isCrash()) { terminate() }
    snake.add(head)  // add head pixel

func isFoodEaten():
    return (head.x == food.x) && (head.y == food.y)

func generateFood():
    // TODO: check that food is not
    //       generated on snake
    x = random(arena.minX, arena.maxY)
    y = random(arena.minY, arena.maxY)
    food.x = x
    food.y = y

func isCrash():
    return    curr.x == arena.minX
           || curr.x == arena.maxX
           || curr.y == arena.minY
           || curr.y == arena.maxY

func isBite():
    for(CoOrdinate e : snake)
        if(e.x == head.x && e.y == head.y) {return true}
    return false

func terminate():
    print("Oops! Bang!! Bye!!!")
```

Designing A Connection Pool

January 24, 2014

Design, Java

Leave a comment

Disclaimer

If you want to use a production quality connection pool, I would recommend you search and use some existing *open source* connection pool (such as Apache dbcp, c3p0, etc).
If you need a functionality, which is not provided by any existing opensource implementation, try to implement it
If you wanna design a connection pool for improving your knowledge, understanding and taking one step closer to mastering 'the craft', carry on

Underlying Principal

Connection creation is a costly process as you have to communicate via network. Connection pooling is an optimization technique in which the application creates and pools a given number of connections at initialization time. A typical flow is as follows:

1. The connection pool creates connections at initialization time (or maybe lazily, on demand)
2. Upon creation, connections are added to a 'pool of connections'
3. To use the connection
    A. Application borrows a connection from the connection pool. (The connection is now marked as 'active' or 'borrowed' in the connection pool)
    B. Application performs required operations using the connection (aka database call)
    C. Application surrenders the connection to the connection pool. (The connection is now marked as 'idle' or 'available' in the connection pool)

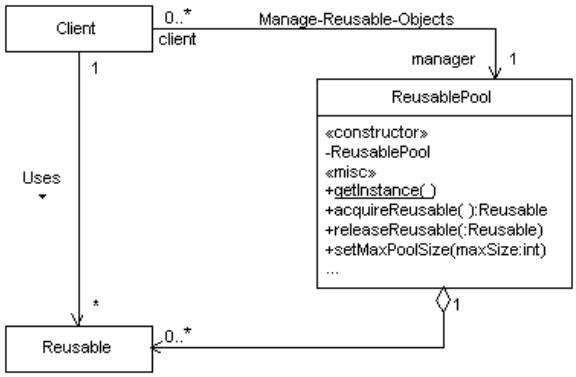
Gotchas

1. The operation of giving a connection upon request should be thread safe. It means that the connection pool logic should ensure that no two requesting threads should be granted the same connection
2. Client threads (i.e., threads which have borrowed a connection from connection pool) should not be able to close the connection. A good idea is to wrap the concrete connection in a wrapper class which implements the same interface as in concrete implementation. The close() function of the wrapper class should simply return the connection to the connection-pool. [If this point did not make much sense, I recommend you read the 'Proxy Design Pattern'.]
3. If the pool has no idle connection. then the requesting thread should be blocked till a

connection becomes available.

Design pattern used

Object Pool Pattern
The UML of object pool pattern is given below



The Code

Github: MinimalConnectionPool

Related links

http://www.javaworld.com/article/2076690/java-concurrency/build-your-own-objectpool-in-java-to-boost-app-speed.html
http://en.wikipedia.org/wiki/Object_pool_pattern
http://codereview.stackexchange.com/questions/4090/connection-pool-in-java
http://codereview.stackexchange.com/questions/40005/code-review-for-connection-pool