

---

# Design Documentation

Matthew Arbesfeld, Matthew Tancik, Anubhav Jain

---

The changes in this revision include a larger description of all of our datatypes, a clearer snapshot diagram and more details added to the Concurrency Strategy and to the Testing Strategy.

## Datatypes

### Packet

Packet is an abstract class that implements the Serializable interface, which is used to represent all the types of data that is sent between the client and the server.

**process(PacketHandler handler)** - Abstract method to handle processing of packets

Many subclasses of Packet:

#### **PacketBoardIdentifierList(BoardIdentifier[] boards)**

A Packet that represents all the Boards currently known to the server. Takes a list of boards as input.

**boards()** – returns a deep copy of the boards stored in the packet

#### **PacketBoardModel(BoardModel boardModel)**

A packet that represents the current boardModel

**boardModel()** – returns the boardModel stored in the packet

#### **PacketBoardUsers(Identifiable[] Users)**

A packet that represents all the users associated with a boardModel

**boardUsers()** – returns the users stored in the packet

#### **PacketClientReady()**

A packet that represents a client is ready to accept drawing commands

#### **PacketDrawCommand(DrawCommand drawCommand)**

A packet that represents a draw command on a board

**drawCommand()** – returns the DrawCommand associated with the packet

#### **PacketExitBoard()**

A packet that represents a client quitting a board

#### **PacketJoinBoard(BoardIdentifier boardName)**

A packet that represents a client joining a board

**boardName()** – returns the boardName that was joined by the client

**PacketMessage(String text)**

A packet that represents a message, used for chat client

**Text()** – returns the text stored in the message

**PacketNewBoard(BoardIdentifier boardName, int width, int height)**

A packet that represents all the information from client of a new board

**boardName()** – returns the boardName

**width()** – returns the width of the board

**height()** – returns the height of the board

**PacketNewClient(ClientIdentifier senderName)**

A Packet that represents new client information

**senderName()** – returns ClientIdentifier senderName

**PacketLayerAdjustment(LayerProperties properties, LayerAdjustment adjustment)**

A packet that represents an adjustment in the layer, whether that be a change in opacity, visibility or moving the layer up or down

**layer()** – return LayerIdentifier layerName

**layerProperties()** – return LayerProperties properties

**PacketNewLayer(LayerIdentifier layerName)**

A packet that represents a new layer being created.

**layerName()** – return LayerIdentifier layerName

**PacketHandler** – interface to handle different types of Packets

**PacketType** – enum for all the different packets

New\_Client, New\_Board, Client\_Ready, Join\_Board, Exit\_Board, Board\_Model, Board\_Users, Board\_Identifier\_List, Draw\_Command

**Identifiable** – interface to create different identifier objects for Users, Boards, and Layers

**Identifier(int id, String name)**

Abstract class that represents a way of identifying boards and users. Implements Identifiable

**Id()** – returns the integer id

**Name()** – returns the String name

**identifier()** – returns this Identifier

**hashCode()** – returns a hash value of the identifier

**equals(Object obj)** – returns a Boolean of whether this Identifier is equal to another object

**toString()** – returns a String representation of the Identifier

**ClientIdentifier** – subclass of Identifier, representing a client

**BoardIdentifier** – subclass of Identifier, representing a board

**LayerIdentifier** – subclass of Identifier, representing a layer

**BoardModel(BoardIdentifier boardName, CanvasController canvas)**

**BoardModel(BoardIdentifier boardName, CanvasController canvas, Identifiable[] initUsers)**

BoardModel is the model used to represent a board, with a CanvasController representing the canvas on top of which we draw and a synchronized set <Identifiable> of users. It implements Identifiable, Drawable, and Serializable

**addUser(Identifiable user)** – adds the user to the synchronizedSet of users

**containsUser(Identifiable user)** – Boolean value if the board contains the user

**removeUser(Identifiable user)** – removes the user from the synchronizedSet of users

**users()** – returns a deep copy array of Identifiable[] users

**drawPixel(Pixel pixel)** – draws pixel on the canvas

**drawLine(Pixel pixel)** – draws line on the canvas

**drawFill(Pixel pixel)** – draws a fill command on the canvas

**width()** – returns the canvas' width

**height()** – returns the canvas' height

**canvas()** – returns the Drawable canvas

**getBoardName()** – returns the boardName

**getPixelColor(LayerIdentifier id, Pixel pixel)** – draws pixel on the canvas

**addLayer(LayerIdentifier layer)** – adds a layer to the canvas

**adjustLayer(LayerProperties properties, LayerAdjustment adjustment)** – adjusts the layer with the appropriate adjustment

**setDrawingControllerDefault()** – sets the default drawing controller to be

**UserTableModel** – a TableModel to represent the users connected to the board and display them

**getColumnCount()** – returns the number of columns

**getRowCount()** – returns the number of rows

**getColumnName(int col)** – returns the column name at index col

**getValueAt(int row, int col)** – returns the value at location (row, col)

**setValueAt(Object value, int row, int col)** – updates the value at (row,col) with the new value

**updateData(Object[][] newData)** – updates the data in the model

## **SocketHandler**

SocketHandler is an abstract class that implements PacketHandler, Runnable, Identifiable and Serializable. It handles all the connections between the server and client, and contains an input and output ObjectOutputStream to handle the serialization and deserialization of objects sent over the network.

**SocketHandler(Socket socket)** – constructor which just takes a socket as an input

**run()** – runs handleConnection() and upon completion, closes the socket for this SocketHandler and closes the Input/Output object streams

**connectionClosed()** – closes the connection

**sendPacket(Packet packet)** – writes the packet to the objectOutputStream

**identifier()** – return the identifier

**ServerSocketState** – enum that has values of INITIALIZING, IDLE, PLAYING which represent the different states a ServerSocket is in

### **ServerSocketHandler(Socket socket, Server server)**

ServerSocketHandler creates a new socket when the client joins the server and it handles communication with just one client, and talks to the Server to receive information about the board.

**receivedNewClientPacket(PacketNewClient packet)** – handles receiving a new client packet

**receivedNewBoardPacket(PacketNewBoard packet)** – handles receiving a new board packet

**receivedJoinBoardPacket(PacketJoinBoard packet)** – handles receiving a join board packet

**receivedExitBoardPacket(PacketExitBoard packet)** – handles receiving an exit board packet

**notifyBoardUsersChanged()** – broadcasts a packet to all users letting them know of a change in the user list

**receivedClientReadyPacket( PacketClientReady packet)** – handles receiving a client ready packet

**receivedDrawCommandPacket(PacketDrawCommand packet)** – handles receiving a draw command packet

**receivedMessagePacket(PacketMessage packet)** – handles receiving a message packet

**receivedLayerAdjustmentPacket(PacketLayerAdjustment packet)** – handles receiving a layer adjustment packet

**receivedNewLayerPacket(PacketNewLayer packet)** - handles receiving a new layer packet

**receivedBoardModelPacket(PacketBoardModel packet)** – handles receiving a board model packet

**receivedBoardIdentifierListPacket(PacketBoardIdentifierList packet)** – handles receiving a Board Identifier List packet

**receivedBoardUsersPacket(PacketBoardUsers packet)** – handles receiving a board users packet

**broadcastPacketToBoard(Packet packet)** – broadcasts the packet to all the clients of the current board

### **Server** – represents the main class of our server and implements Identifiable

**Server(int port)** – we create a server using just the port it's running on. It stores the boards on the server and a list of all the clients

**Serve()** – serves each new connection to a new thread

**removeClient(ServerSocketHandler handler)** – removes the handler from our list of clients

**addBoard(BoardIdentifier boardName, BoardModel model)** – adds the board to the list of boards stored on the server

**broadcastPacketToAllClients(Packet packet)** – broadcasts a packet to all the clients

**constructBoardIdentifierListPacket()** – creates a PacketBoardIdentifierList

**notifyBoardListChanged()** – broadcasts packet to all clients with the new board list

**newBoard(PacketNewBoardPacket packet)** – creates a new BoardModel from a packet

**joinBoard(PacketJoinBoard packet)** – adds the client to the board

## **Vector2** – represents a 2D vector

**Vector2(double x, double y)** – create a vector point at location (x,y)

**x()** – returns x

**y()** – return y

**abs()** – returns length of the vector

**normalized()** – creates a normalized form of the vector

**clone()** – returns a deep copy of the object

**hashCode()** – computes a hash code value for the vector

**equals(Object obj)** – returns Boolean value of whether current object is equal to obj

## **Layer**

Layer represents a layer in the GUI and it extends DrawableBase

**Layer(int width, int height, LayerIdentifier layerIdentifier, int level)**

**layerProperties()** – return layerProperties

**setLevel(int level)** – sets level to the new level

**level()** – returns level

**fillWithColor(Color color)** – makes the drawing buffer entirely one color and is a synchronized method

**drawPixel, drawLine, drawFill** – synchronized methods to handle each of the draw commands

**getPixelColor(LayerIdentifier id, Pixel pixel)** – synchronized method that returns the color of pixel at pixel location

**paintOnGraphics(Graphics g)** – paints graphics onto image

**isVisible()** – synchronized method which says if the layer is visible or not

**opacity()** – synchronized method which returns the opacity of the layer

**writeObject()** and **readObject()** – serialize and deserialize the information to recreate layers after they've sent/received

**image()** – returns the image on the layer

## **ClientController** – extends SocketHandler and represents the controller for the client

**ClientController(String userName, String hostName, int portNumber)** – constructor which creates a new Socket with the hostName and portNumber and then starts clientState at IDLE. It also announces itself by sending a NewClientPacket

**receivedNewClientPacket(PacketNewClient packet)** – handles receiving a new client packet

**receivedNewBoardPacket(PacketNewBoard packet)** – handles receiving a new board packet

**receivedJoinBoardPacket(PacketJoinBoard packet)** – handles receiving a join board packet

**receivedExitBoardPacket(PacketExitBoard packet)** – handles receiving an exit board packet

**notifyBoardUsersChanged()** – broadcasts a packet to all users letting them know of a change in the user list

**receivedClientReadyPacket(PacketClientReady packet)** – handles receiving a client ready packet

**receivedDrawCommandPacket(PacketDrawCommand packet)** – handles receiving a draw command packet

**receivedMessagePacket(PacketMessage packet)** – handles receiving a message packet

**receivedLayerAdjustmentPacket(PacketLayerAdjustment packet)** – handles receiving a layer adjustment packet

**receivedNewLayerPacket(PacketNewLayer packet)** - handles receiving a new layer packet

**receivedBoardModelPacket(PacketBoardModel packet)** – handles receiving a board model packet

**receivedBoardIdentifierListPacket(PacketBoardIdentifierList packet)** – handles receiving a Board Identifier List packet

**receivedBoardUsersPacket(PacketBoardUsers packet)** – handles receiving a board users packet

**setGUILayers()** – updates the layers of the GUI

**setView(ClientGUI view)** – updates the view of the GUI

**generateNewBoard(BoardIdentifier boardName, int width, int height)** – create a new board and send a NewBoardPacket

**connectToBoard(BoardIdentifier boardName)** – connects to the new board

**disconnectFromCurrentBoard()** – disconnects from the current board

**sendDrawCommand(DrawCommand drawCommand)** - send a draw command to the server

**sendMessage(String text)** – send a message to the server

**setStrokeColor(Color strokeColor)** – update the stroke color

**setStrokeWidth(int strokeWidth)** – update the stroke width

**setStrokeType(StrokeType newStrokeType)** – update the stroke type

**setEraserOn(Boolean eraserOn)** – update if the eraser is on or off

**setFillOn(Boolean fillOn)** – update the fill Boolean whether it's on or off

**setSymetry(int symmetry)** – update the symmetry value

**addLayer(LayerIdentifier identifier)** – send a NewLayerPacket

**adjustLayer(LayerProperties layerProperties, LayerAdjustment adjustment)** – send a new LayerAdjustment Packet

## **Protocol**

We used a client/server architecture as our network architecture to handle data transfer between the clients and our server. We used an object stream to pass over packets, so we technically don't have a grammar for communication. We use serializable objects and an ObjectOutputStream / ObjectInputStream for communication.

Protocol from client to server:

The client sends data in packets letting the server know what is happening on the client side. This includes:

- new client packet, meaning that a new client has joined the server
- new board packet, meaning that a client has created a new board
- join board packet, meaning that a client has joined a board

- exit board packet, meaning that a client has left a board
- draw command packet, meaning that a client has drawn on their board
- client ready packet, meaning that a client is ready to receive draw commands
- message packet, which holds chat information
- new layer packet, which represents a new layer being sent to the server
- layer adjustment packet, which lets the server know about the new adjustment

Protocol from server to client:

The server sends data back to the client in packets as well, giving it updates to the new state that it should reflect as other users work on the same board. These packets include:

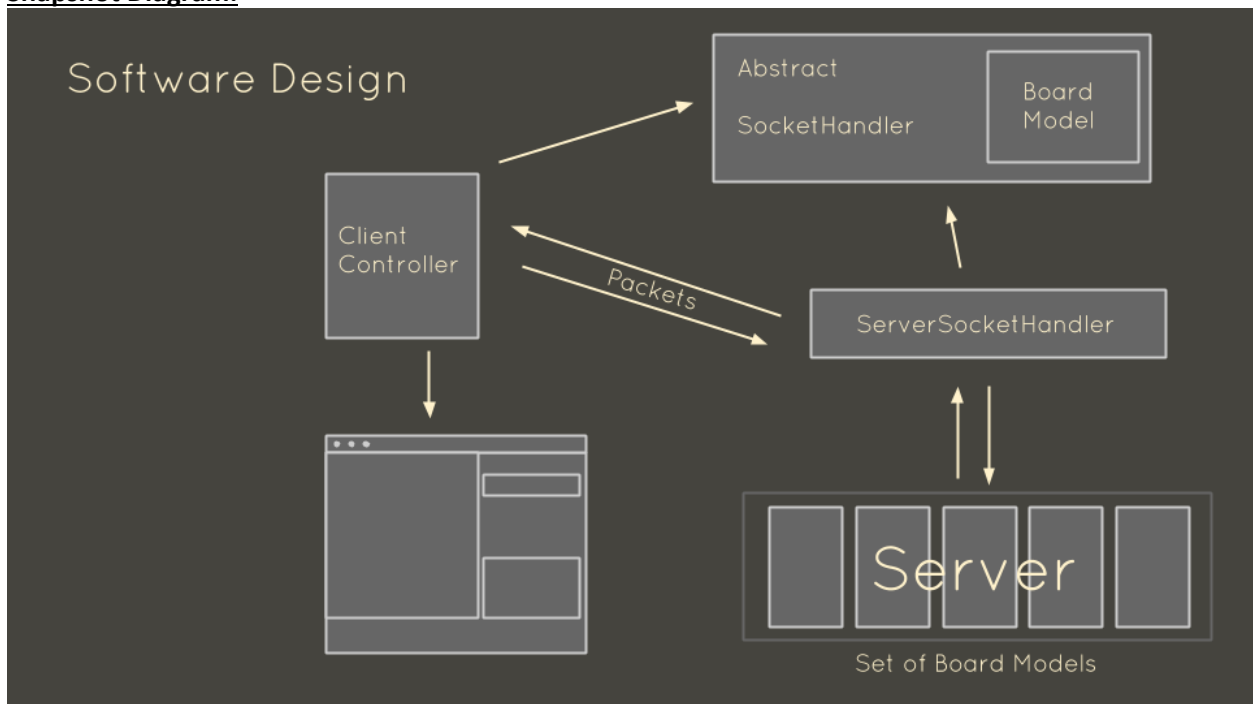
- join board packet, meaning that a client has joined a board
- exit board packet, meaning that a client has left the board
- draw command packet, meaning that a client has drawn on the board and the board should be updated
- board model packet, which sends the current board to the client so it can update itself to the appropriate point
- board users packet, which sends the current users connected to the board to the client
- board identifier packet, which sends the current boards on the server to the client
- message packet, which holds a new chat message to be added to the chat menu
- new layer packet, which represents a new layer being added to the clients
- layer adjustment packet, which lets all clients know about the new adjustment

## **Concurrency Strategy**

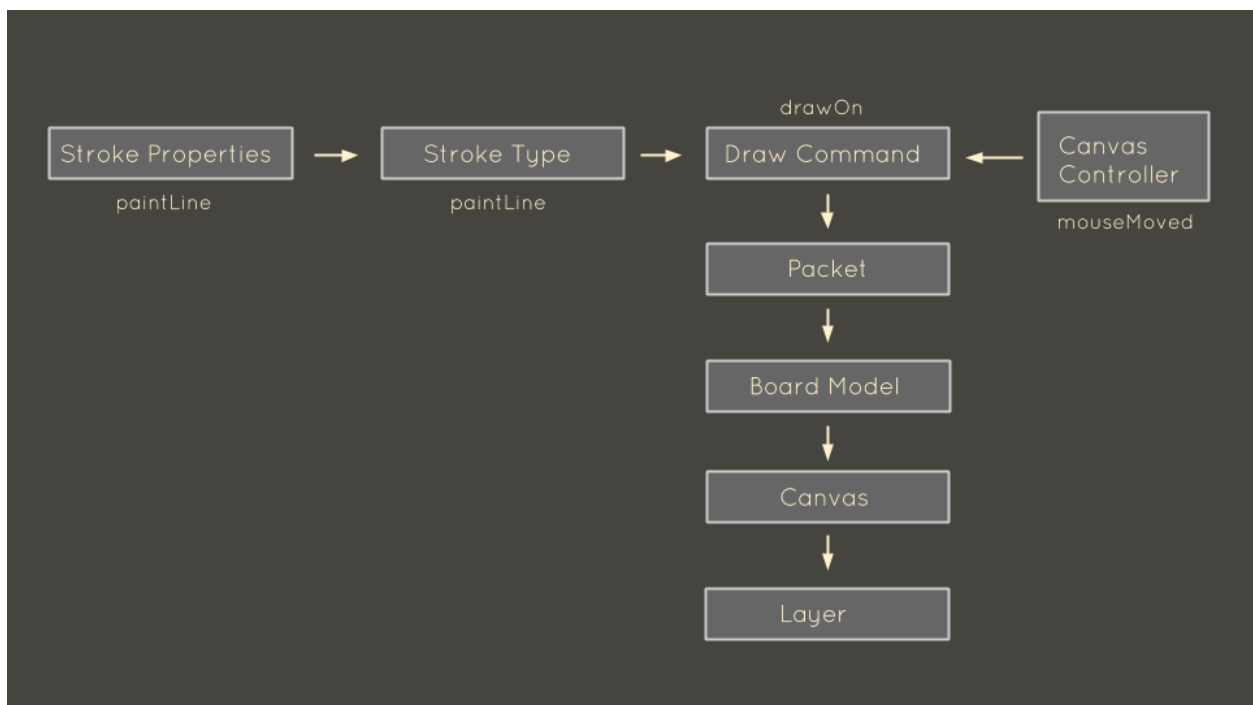
### **Specifications for our design:**

We currently don't have limitations on the number of clients that can be on the server and the number of boards the server can hold and these will be based off of the limitation of the server, which we have yet to test. The client only stores a list of boards and when it switches boards, the client discards its current board and replaces that data with the board data that is sent over the server. The server stores boards until it is shutdown. The client only works on one board at a time.

### Snapshot Diagram:



Above is a snapshot diagram which demonstrates our software design. Below is a diagram which shows the different classes interacting with each other as we draw on the screen.



There are two potential concurrency issues in our design. First, we have multiple "ServerSocketHandlers" that are simultaneously mutating data in the server. We deal with this concurrency by making the member variables in the Server class thread safe, and also by



making all the Server methods that are accessed from the threads synchronized on these variables. Moreover, any method of the ServerSocketHandler that is accessed via the Server thread, such as "sendPacket" must be synchronized.

The second concurrency issue is on the client side, when the client receives packets on a separate thread from the UI thread. We identify all of the methods where this is the case, and make them synchronized. For example, we may receive a "drawPixel" command from the server that is concurrent with our own "drawPixel" command. We ensure that "drawPixel" is synchronized so that there is no concurrency issue. All the commands to draw to the images on the layers are synchronized, as well as the methods to receive information back from the images, meaning that our drawing abilities are safe from concurrency issues.

## **Testing Strategy**

We plan on testing the following objects we create: Packets, BoardModel, Identifiers, Client, ClientController, ClientState, DrawCommand, DrawCommandPixel, Drawable, Canvas2d, DrawableCanvas2d, DrawableBase, Pixel, Server, ServerSocketHandler, ServerSocketState, SocketHandler, StrokeTypes, and StrokeProperties

### **GUI Testing Strategy:**

Manual testing, see ClientGUI.java file.

### **Packet Testing Strategy:**

We will create all the different packets, check that they properly store the information we pass into them, and then check that serializing / deserializing the objects means they still store the correct data.

### **BoardModel Testing Strategy:**

We will create a board model from the two different constructors, make sure to check the edge cases (0 users, null canvas, etc) and then test each of the methods in Board Model.

### **Identifiers Testing Strategy:**

We will create all the different types of identifiers and make sure to check the edge cases (not providing ids / names) and then test the methods defined by the interface.

### **Client Testing Strategy:**

Test the creation of a client controller and then check that the client gui runs if the popup gui verifies we've connected to the server.

### **ClientController Testing Strategy:**

We test the creation of the client controller with a variety of ports and user/host names and then test each of the receivedPacket methods and make sure they're working properly by checking that they change the appropriate attributes within the controller.

### **DrawCommandPixel Testing Strategy:**

We try to draw pixels and make sure that they are actually drawn on the canvas.

**Canvas2d Testing Strategy:**

We test the creation of a canvas and then test drawing pixels, saving images, and painting components.

**DrawableCanvas2d Testing Strategy:**

We test the user responses manually by checking that the gui clicks are actually being read and that they are creating a corresponding pixel drawing on the screen.

**DrawableBase Testing Strategy:**

We test the creation of instances of DrawableBases and then check that we're able to get the same values back by testing the getter methods.

**Pixel Testing Strategy:**

We test the creation of pixels using both constructors and test all the edge cases, and then test each method individually.

**Server Testing Strategy:**

We test the server class by creating a new server, adding clients and removing them and checking that they are actually added/removed, and then testing adding and removing boards. We also test that calling Identifier returns an UnsupportedOperationException and that we can handle multiple clients.

**ServerSocketHandler Testing Strategy:**

We test server socket handler by testing the constructor with edge cases, and then test each method and its effects on an instance of the model.

**StrokeProperties Testing Strategy:**

We test constructing a stroke property with edge case values and then test setting the stroke values and making sure they get set properly. We also test the paint point and make sure it draws the current point on our canvas visually.