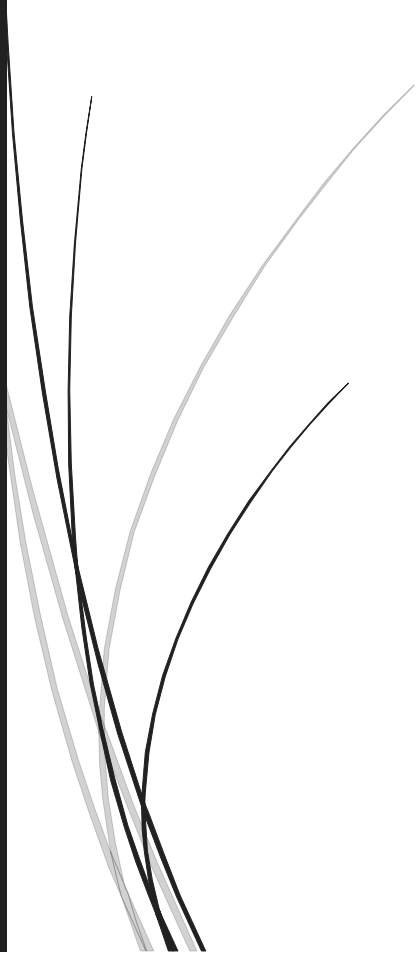
Zachery Utt's Project Portfolio

All source code can be viewed publicly on GitHub.



Exam Software for the Ministry of Education in Saudi Arabia

Language Selection: C#

Project Brief: After administering a city-wide English examination for the students of Jeddah, the Ministry of Education needed custom software to score custom answer sheets quickly and accurately- and scalable to function on thousands of scanned sheets.

My Contribution: I was tasked with designing the software to scan and score each answer sheet. After scoring, the program uses identifiers on the exam (a student ID bar code) to pass the exam information to the Ministry of Education's student database system. I collaborated with other engineers on how to export data from my program with existing student information systems.

My Solution: My program uses the OpenCV library to identify circles within the image of the sheet (Image 1). It then aggregates concentric circles into single objects and uses a custom algorithm to look for a grid pattern based on the center points of these circle objects and identify sections of the exam [shown in different colors] (Image 2). Circle objects that are darker (have a higher average pixel color) are identified as the answer choice selected by the student (Image 3). The software is multithreaded and optimized to work on hundreds of sheets in linear time.

المملكة العربية السعودية
وزارة التعليم
الادارة العامة للتعليم بمحافظة جدة
مدرسة الامام الذهبي المتوسطة

وزارة التعليم
Ministry of Education

المادة	اللغة الانجليزية
الصف	3
الزمن	ساعتان
رقم اللجنة	1

اسم الطالب : احمد سعد معوض المرواني الجهني

001-1262050

1262050

Student Condition
Absent
Cheat

Multiple Choice

1	A	B	C	D
2	A	B	C	D
3	A	B	C	D
4	A	B	C	D
5	A	B	C	D
6	A	B	C	D
7	A	B	C	D
8	A	B	C	D
9	A	B	C	D
10	A	B	C	D
11	A	B	C	D
12	A	B	C	D
13	A	B	C	D
14	A	B	C	D
15	A	B	C	D
16	A	B	C	D
17	A	B	C	D
18	A	B	C	D
19	A	B	C	D
20	A	B	C	D
21	A	B	C	D
22	A	B	C	D
23	A	B	C	D
24	A	B	C	D
25	A	B	C	D
26	A	B	C	D
27	A	B	C	D
28	A	B	C	D
29	A	B	C	D
30	A	B	C	D

Pairing Questions

1	A	B	C	D	E	F	G	H	I	J
2	A	B	C	D	E	F	G	H	I	J
3	A	B	C	D	E	F	G	H	I	J
4	A	B	C	D	E	F	G	H	I	J
5	A	B	C	D	E	F	G	H	I	J
6	A	B	C	D	E	F	G	H	I	J
7	A	B	C	D	E	F	G	H	I	J
8	A	B	C	D	E	F	G	H	I	J
9	A	B	C	D	E	F	G	H	I	J
10	A	B	C	D	E	F	G	H	I	J

True / False

1	T	F
2	T	F
3	T	F
4	T	F
5	T	F
6	T	F
7	T	F
8	T	F
9	T	F
10	T	F

Completion

1	0	1
2	0	1
3	0	1
4	0	1
5	0	1
6	0	1
7	0	1
8	0	1
9	0	1
10	0	1

Subjective

1	0	1	2	3	4	5	6
2	0	1	2	3	4	5	6
3	0	1	2	3	4	5	6
4	0	1	2	3	4	5	6
5	0	1	2	3	4	5	6

المملكة العربية السعودية
وزارة التعليم
الادارة العامة للتعليم بمحافظة جدة
مدرسة الامام الذهبي المتوسطة

وزارة التعليم
Ministry of Education

المادة: اللغة الانجليزية
الصف: 1
الزمن: ساعتان
رقم اللجنة: 105

اسم الطالب : عمر محمد علي المعبدي

السجل المدني: 1132115757

رقم الجلوس: 0725012 8

Student Condition
Absent
Cheat

Multiple Choices

1	A	B	**C**	D	11	A	B	C	**D**	21	**A**	B	C	D
2	A	**B**	C	D	12	A	B	**C**	D	22	A	**B**	C	D
3	A	B	C	**D**	13	A	B	C	**D**	23	**A**	B	C	D
4	**A**	B	C	D	14	A	B	**C**	D	24	A	B	C	**D**
5	A	B	**C**	D	15	**A**	B	C	D	25	A	B	C	D
6	A	**B**	C	D	16	A	**B**	C	D	26	A	B	C	D
7	A	B	C	**D**	17	A	B	C	**D**	27	A	B	C	D
8	**A**	B	C	D	18	A	**B**	C	D	28	A	B	C	D
9	A	B	**C**	D	19	A	**B**	C	D	29	A	B	C	D
10	**A**	B	C	D	20	A	**B**	C	D	30	A	B	C	D

Pairing Questions

1	A	B	**C**	D	E	F	G	H	I	J
2	A	**B**	C	D	E	F	G	H	I	J
3	A	B	C	**D**	E	F	G	H	I	J
4	**A**	B	C	D	E	F	G	H	I	J
5	A	B	C	D	**E**	F	G	H	I	J
6	A	B	C	D	E	F	G	H	I	J
7	A	B	C	D	E	F	G	H	I	J
8	A	B	C	D	E	F	G	H	I	J
9	A	B	C	D	E	F	G	H	I	J
10	A	B	C	D	E	F	G	H	I	J

True / False

1	**T**	F
2	T	**F**
3	**T**	F
4	T	**F**
5	**T**	F
6	T	F
7	T	F
8	T	F
9	T	F
10	T	F

Completion

1	0	**1**	6	0	**1**
2	0	**1**	7	0	**1**
3	0	**1**	8	0	**1**
4	0	**1**	9	0	**1**
5	0	**1**	10	0	**1**

Subjective

1	0	1	2	3	4	5	**6**
2	0	1	2	3	4	5	**6**
3	0	1	2	3	4	5	**6**
4	0	1	2	3	4	5	**6**
5	0	1	2	3	4	5	**6**

المملكة العربية السعودية
وزارة التعليم
الادارة العامة للتعليم بمحافظة جدة
مدرسة الامام الذهبي المتوسطة

وزارة التعليم
Ministry of Education

المادة: اللغة الانجليزية
الصف: 1
الزمن: ساعتان
رقم اللجنة: 105

اسم الطالب :

السجل المدني

رقم الجلوس

Student Condition
Absent
Cheat

Multiple Choices

1	A	B	C	D	11	A	B	C	D	21	A	B	C	D
2	A	B	C	D	12	A	B	C	D	22	A	B	C	D
3	A	B	C	D	13	A	B	C	D	23	A	B	C	D
4	A	B	C	D	14	A	B	C	D	24	A	B	C	D
5	A	B	C	D	15	A	B	C	D	25	A	B	C	D
6	A	B	C	D	16	A	B	C	D	26	A	B	C	D
7	A	B	C	D	17	A	B	C	D	27	A	B	C	D
8	A	B	C	D	18	A	B	C	D	28	A	B	C	D
9	A	B	C	D	19	A	B	C	D	29	A	B	C	D
10	A	B	C	D	20	A	B	C	D	30	A	B	C	D

Pairing Questions

1	A	B	C	D	E	F	G	H	I	J
2	A	B	C	D	E	F	G	H	I	J
3	A	B	C	D	E	F	G	H	I	J
4	A	B	C	D	E	F	G	H	I	J
5	A	B	C	D	E	F	G	H	I	J
6	A	B	C	D	E	F	G	H	I	J
7	A	B	C	D	E	F	G	H	I	J
8	A	B	C	D	E	F	G	H	I	J
9	A	B	C	D	E	F	G	H	I	J
10	A	B	C	D	E	F	G	H	I	J

True / False

1	T	F
2	T	F
3	T	F
4	T	F
5	T	F
6	T	F
7	T	F
8	T	F
9	T	F
10	T	F

Completion

1	0	1	6	0	1
2	0	1	7	0	1
3	0	1	8	0	1
4	0	1	9	0	1
5	0	1	10	0	1

Subjective

1	0	1	2	3	4	5	6
2	0	1	2	3	4	5	6
3	0	1	2	3	4	5	6
4	0	1	2	3	4	5	6
5	0	1	2	3	4	5	6

Education Technology Software

Language Selection: SQL, Java, React

Project Brief: As users move through an online engineering course meant to teach 3D CAD modeling skills to students in underserved schools, assessment data must be aggregated and processed to provide individualized student learning plans and tools for instructors.

My Contribution: In addition to designing the data collection and database systems, I also designed an instructor dashboard to give students and teachers insights onto areas of weak student performance.

My Solution: In addition to focusing on assessment averages, my data algorithm attempts to quantify student engagement by tracking response time in answering questions, time spent without user activity, clicks recorded, and video replay counts. This factor is aggregated into a single multiplier and applied to assessment scores to track student progress.

ACU Inventor Study Guide

Instructor Dashboard

Home

Demo Data

NEW FOLDER

EDIT USER

DOWNLOAD DATA

Name ▼	Course Progress	Last Login	Total Activity
► Aziz, Sherif		10/3/2019	02:39:30
► Barger, Debra		9/20/2019	02:20:30
► Bell Wade, Brittnaie		9/20/2019	00:18:00
► Dixon, Tishanna		9/30/2019	03:08:30
► Dtorres, Sylvia		9/20/2019	00:10:30
► Fox, Matthew		9/20/2019	02:14:00
► Hoayun, John		9/19/2019	01:33:30
► Howes, Elizabeth		9/20/2019	03:30:00
► Hutton, Josh		10/1/2019	02:29:30
► Jane Pack, Mary		9/20/2019	03:10:00

© Certiprep Guides, LLC | Made in Orlando

certiprep guides

Home Pricing certifyHQ Login Sign Up

ACU Inventor Stud

Hon

Demo Data

NEW FC

EDIT U

DOWNLOA

ructor Das

ctivity

0

0

0

0

0

0

0

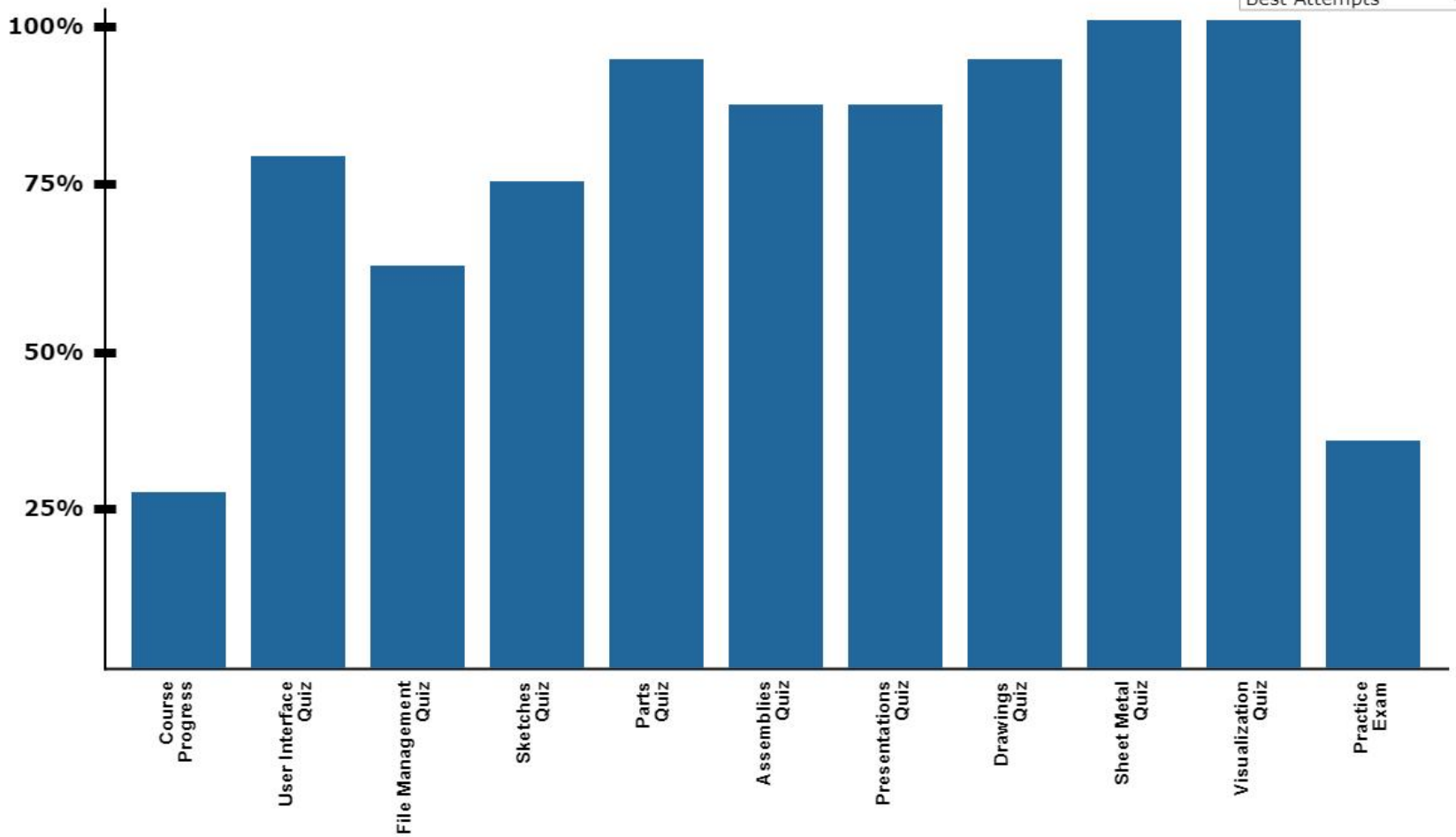
0

0

Metrics

X

Best Attempts



© Certiprep Guides, LLC | Made in Orlando

Campaign Website

Language Selection: SQL, PHP, React, Python

Project Brief: Campaign websites serve as a metaphor for the candidate themselves; websites that are generic, difficult to use, undermaintained, lack translations represent a lack of polish and effort on the part of the candidate to connect with voters in the community. Further, they help gather valuable analytics including when people enter the site, their device specifications, and their location within the district that track campaign efficacy.

My Contribution: I worked as the chief programmer with campaign staff to engage voters through the site. In addition to providing a nice skin for end users, I designed the data analytics system that tracks and notifies the campaign team when user participation and frequency become statistically significant. Further, I developed the data analytic tracking system, and implemented key security provisions to prevent an attack.


Phil Dodds
US HOUSE - FL District 3
#philwill invest in rurual education
Name
Email Address
Take Action

Our Values

No Corporate PAC Money

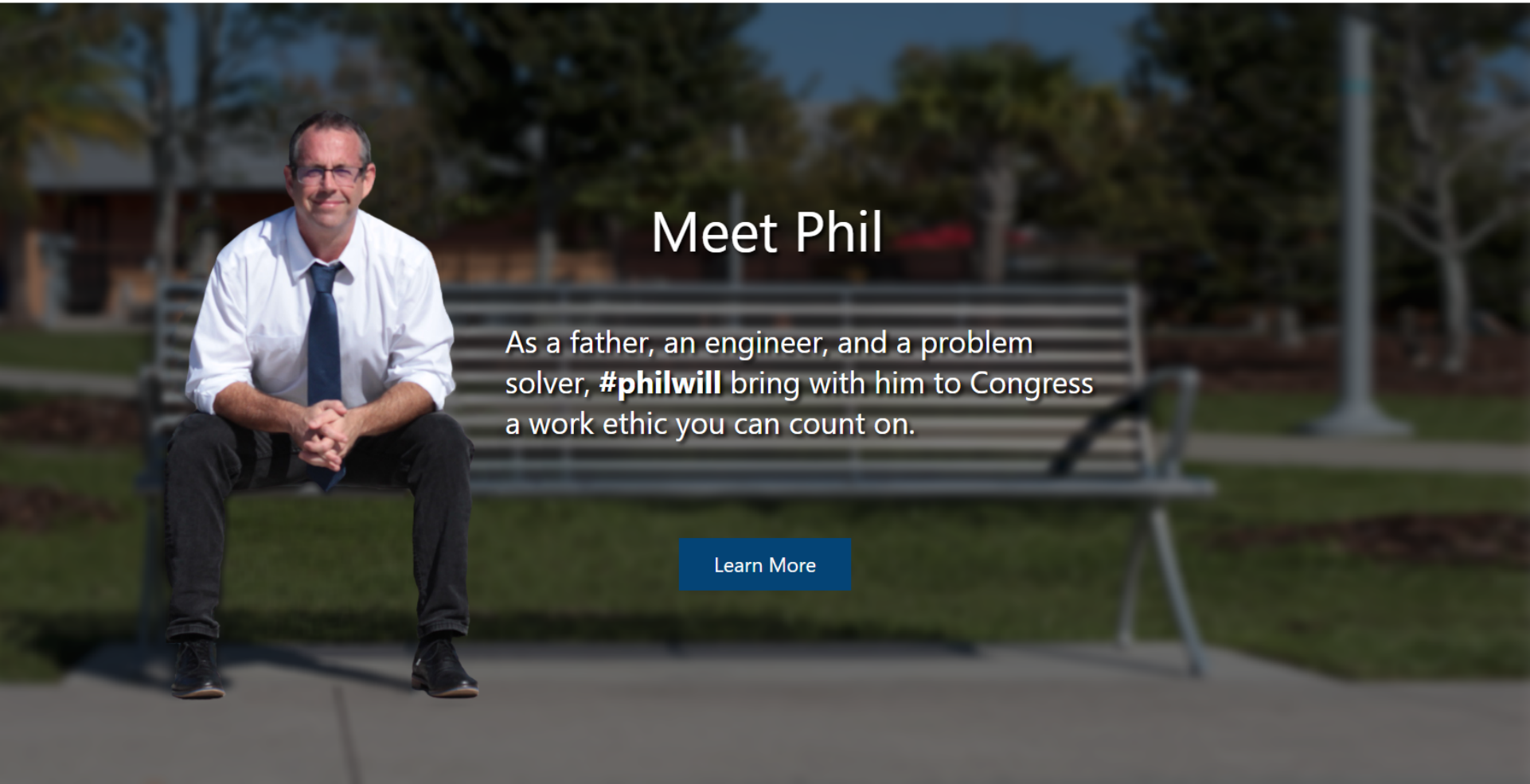
Phil's first priority is **ending corruption**. Phil won't take a nickle from any corporation or lobbyist, and will work to make laws to fight corruption.

Finding Solutions that Work

As an engineer, Phil's campaign is about **bringing people together** to find solutions instead of furthering extreme partisan talking points.

Fighting For What Matters

Phil is in this race to **lower the cost of healthcare**, **end climate change**, and **restore ethics to government**.



Get Involved

Event Name: Meet 'n Greet
Date: 10/22/2019
Time: 6:00 PM - 8:00 PM

Sign Up

Event Name: Meet 'n Greet
Date: 10/28/2019
Time: 6:00 PM - 8:00 PM

Sign Up

Event Name: Meet 'n Greet
Date: 11/02/2019
Time: 6:00 PM - 8:00 PM

Sign Up

Contribute

$2 $5 $10 $15 $25 $50

Phil Dodds
US HOUSE - FL District 3

Stay in Touch

9200 NW 39th Ave Suite 130-209 Gainesville, Fl 32606

Paid for by Voters for Phil Dodds

Cyber Security Research

Project Brief: One sign that a user may be a bot instead of a human is their click patterns. To look for ways to understand how suspicious click patterns can be detected and flagged, researchers needed a tool to simulate bot clicks at a certain click rate. Researchers wanted to control the spread of the random distribution to write better algorithms to detect bots by comparing a distribution to an associated random distribution.

My Contribution: I designed a "random" click speed distribution at a specified average and spread. I published the algorithm that I wrote for other developers to use for this niche application.

My Solution: The program splits a second (1000 milliseconds) into the A sections (where A represents the number of clicks per second), with each section getting a completely random number from 0 to 1000, representing the time to wait between clicks. A scalar multiple is applied to each section to get the total wait time to 1000 milliseconds. A scalar is subtracted from each term such that the average wait time is A). Each term is corrected by a such that an increase in one term towards the average is countered by a decrease in another, to control the degree of spread.

Figure 2: Click Distribution with average at 10 clicks / second at high random spread

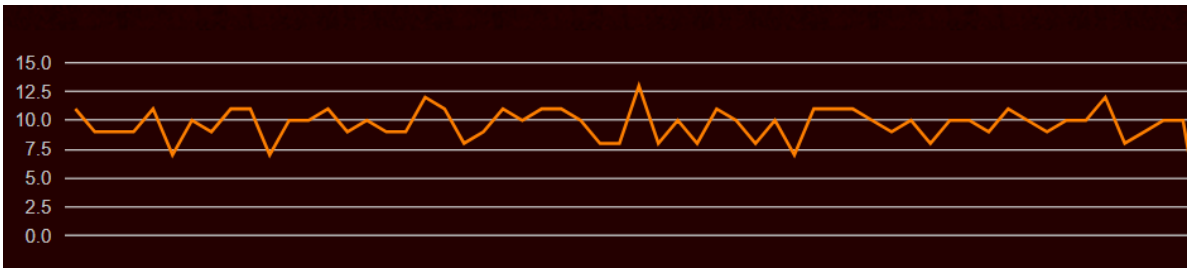


Figure 3: Click distribution with average at 10 clicks / second at medium random spread

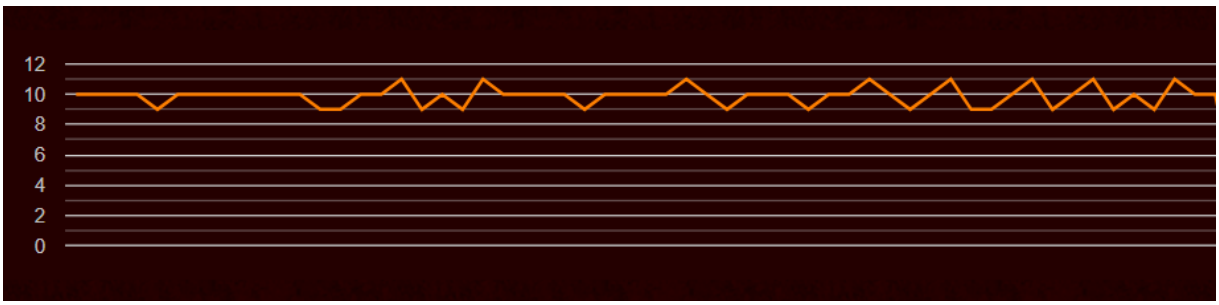
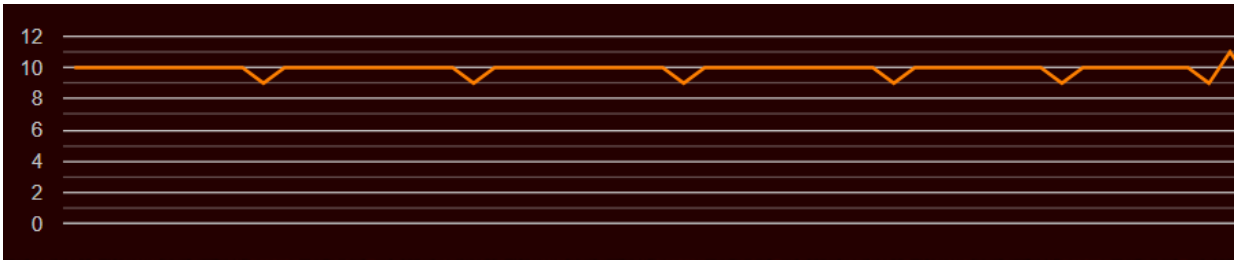


Figure 4: Click distribution with average at 10 clicks / second at low random spread



Tabletop Framework

Project Brief: Amidst the COVID-19 stay at home orders, I worked with a handful of other developers to create an open source table-top game framework to develop board, card, and role-playing games quickly and easily for use during quarantine.

My Contribution: I developed the networking, graphics, and security systems behind the framework. I interwove a strict security protocol to prevent memory editing attacks, as well as a graphics system to support cross-compatibility and cross-language compatibility.

My Solution: I outlined my solution to the project within the following pages. Further, I implemented a sample chess application on the framework.

Network and UI Architecture

The Authoritarian Server Dynamic

When designing a multiplayer environment, it is irresponsible to ignore the reality of memory editing exploitations. As a developer, it is easy to make security decisions during the design process based on the demands of the application. However, as framework designers, we do not have the luxury of accurately performing a risk tradeoff. While it might be innocuous to exploit a friendly game of chess, a virtual casino environment in which real money is wagered presents a far stricter need for adequate security. Thus, we've decided to protect our tabletop simulator from local exploitation through the use of an **authoritarian server**. This model of system architecture requires all handling of game state and rules be performed by the server alone. Clients send their actions to the server and wait for a response to render to the screen.

Given that our framework is "twice removed", the authoritarian server offers many advantages to developers who will eventually use the framework. By placing all game decisions, state, and processing within the server, this design limits the amount of moving parts during development and makes debugging much easier. In distribution, client applications are lower in size because they lack the majority of the logic components to govern gameplay.

Because the vast majority of development occurs on the server side, the developer will do the bulk of development on a single platform, rather than constantly worrying about client's cross-platform compatibility. This is a useful feature of the authoritarian server: the client is reduced to rendering and networking, which are the 2 components most likely to change from platform to platform. By, in effect, reducing the client to an adapter for graphics and networking, our design parallels the existing constraints of multiplatform design. This makes our platform lightweight and extendable to many devices.

From the user's perspective, the authoritarian server offers many gameplay benefits. If users get disconnected, the game is still playable for everyone else, and is recoverable if the player returns. Features like chat can be integrated into the gameplay experience seamlessly. RAM and CPU requirements are not as stringent.. This isn't to say that the client is completely removed

from any calculations or behavior. The client would still be responsible for processing textures, interpolating animations, and physics operations that reduce traffic to the server and offer no vulnerabilities.

The Authoritarian Server Implementation: Client

The client's roles within the authoritarian server model can essentially be boiled down into the view and controller of the MVC model. These responsibilities are defined as follows:

1. Render entities provided by the server and maintain the user's perspective at the tabletop
2. Collect input from the user that could affect the state of the game
3. Update the server with events as the user provides input

First and foremost, it is the client's job to display the game play environment in a way that is meaningful to the end user and conveys the active state of game play. This task itself requires two main subcomponents: physical objects to render and a perspective that represents a realistic transformation of how the user would see the objects from their position. Within the framework, these subcomponents are represented by the **Entity** and **Camera** class.

While the Client is responsible for using the **Entity** and **Camera** classes, it should not be producing these objects. They are created by the server and passed to the Client when the game commences.

Entity objects represent physical objects during gameplay, including dice, poker chips, cards, pieces, etc. **Entity** objects are composed of a **Sprite**, a **PositionVector**, and a unique identifier that will allow the server to track its presence and update all clients if a change occurs.

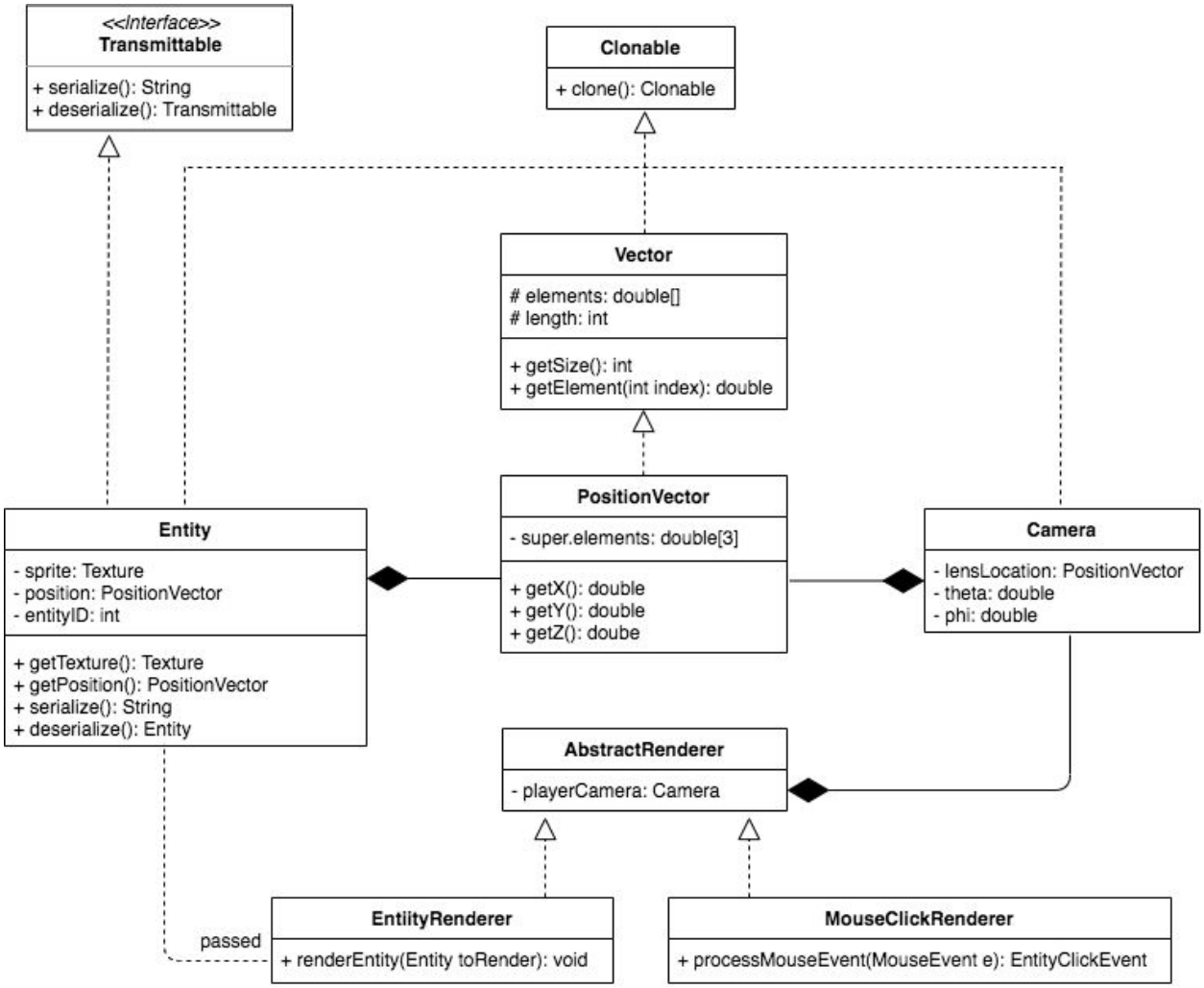
Entity Renderer objects are responsible for translating both the position and texture from a two dimensional **Entity** object into a three dimensional rasterized pixel representation to display to the screen using the perspective of the **Camera**. **Entity** objects must be rendered in a particular order (from farthest to closest) to allow objects to realistically hide behind other objects.

However, given that the **Entity Renderer** functions by traversing and translating **Entity** objects, based on perspective, from farthest to closest, into pixels, the **MouseClickRenderer** can use the

same mechanism in reverse to process a **MouseEvent**. By traversing from closest to farthest, and using the same perspective, the **MouseClickRenderer** simply must render **Entity** objects, until a particular **Entity** renders to the clicked coordinate.This is why both the **MouseClickRenderer** and the **Entity Renderer** inherit from the **AbstractRenderer** superclass.

PositionVector objects represent a series of 3 values corresponding to the X,Y, and Z position of a point in 3-dimensional space. If the developer would prefer to use a different, non-cartesian coordinate system, **PositionVector** can be subclassed to support these coordinates and can convert to cartesian coordinates through the **PositionVector** superclass.

Camera objects represent the viewing perspective of the client. Cameras are composed of a **PositionVector** defining its location, as well as values for θ and Φ to describe the rotation and tilt of the perspective, where each parameter is described by the spherical coordinate system. If a game does not require each player to have their own viewing perspective, **Camera** objects can all be duplicates of one another.



As shown in the diagram, the majority of classes the client interacts with follow the Prototype design pattern and implement the **Cloneable()** method. This is because many of these objects are stored and originate from the server and are handed to the client when the game commences to allow the server to control and reproduce the state of gameplay. All **Entity** objects are stored within the **EntityHolder** class. Each **Entity** object is a duplicate of an identical object stored within a larger **EntityHolder** on the server. In fact, **Entity** objects are completely immutable to the client. If a client wishes to alter the appearance or position of an **Entity**, the client should create an **Event** to the server and obtain a new **Entity** instance with the desired characteristics, if appropriate. The ultimate purpose of the immutability and object duplication interface is to allow the server to control the state of gameplay. While we will not go as far as to

seal the **Entity** class to prevent developers from undoing these protections through subclassing, we strongly believe that these principles should be adhered to.

On the other hand, **Camera** objects are mutable by the client. If the developer chooses to allow players to sweep about the table or the ability to zoom in and out about the table, the developer should subclass the **Camera** class. The server will provide the client with a default **Camera** object when the game commences.

There is one large exception to the premise that clients should not create their own behavioral objects. It is critical to the authoritarian model that the client collect user input, package it, and send it to the server in a way that is parsable and can affect the state of the game. If this functionality is not present, the simulator is, in effect, playing a movie rather than simulating an active game. Thus, the client must be able to dynamically produce **Event** objects to transmit via socket to the server, and respond when the server provides new **Response** instructions on how to continue gameplay.

A considerable amount of time will be spent by the final game developers on subclassing the **AbstractEvent** and **AbstractResponse** classes to match the needs of their games. Each of these subclasses must be implemented on the **EventVisitor** on the server side and **Response Visitor** on the client side to be processed.

If a state-changing concrete **Event** is sent to the server, a concrete **Response** class is returned through the socket to all relevant clients. Once deserialized, the client must perform the appropriate actions to represent to the user the new state of gameplay. The client processes incoming deserialized **Response** objects through the **ResponseVisitor**. The **ResponseVisitor** class should be subclassed by the developer with any custom concrete **Response** objects, providing the client with details on how to represent the new game state to the user.

The Authoritarian Server Implementation: Server

Despite the ominous name, the role of the server in the authoritarian server model can be broken up into 5 pieces

1. Accept new clients to enter the game
2. Provide the client with the gameplay **Entities** when the game commences
3. Maintain the state of the game and all **Entities**
4. Query the clients with necessary GUI prompts
5. Accept **Events** and return **Responses**, where appropriate

The server tracks the state of all entity objects by holding a master **EntityHolder** object that contains within it all player's **Entity** objects. As a player joins the game, the server invokes the **clone()** method of all the **Entity** objects that are pertinent to them, and transmits them through the **EntityResponse** message to the client. **Entity** objects shared between players - or all players - are not cloned for each player. The server simply uses the **Transmittable** abstraction to serialize the shared object, and sends the same string to all players through the **EntityResponse** message.

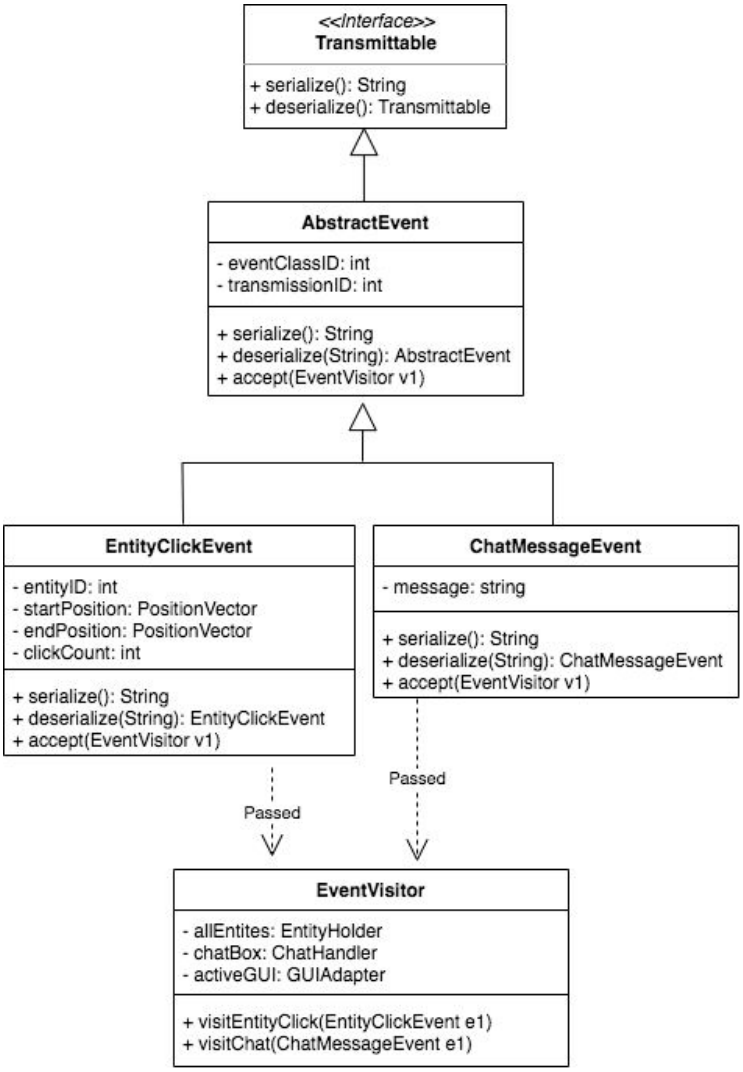
While the prototype interface typically dictates that all fields in a clone should be equivalent, it is critical that **Entity** all clones receive a unique entityID member, or else the server will be unable to track which **Entity** object a change occurred on. The **EntityHolder** will throw an exception if two unique instantiations share the same entityID.

As with any game framework, GUI elements are critical to the user experience. To make the framework easily extendable to as many platforms as possible, the server will not dictate how to create and represent a GUI, only when a GUI should be prompted and when it should be removed, through a **GUIResponse** message. It will be up to the client to produce a platform specific GUI, to collect input from the user, and to transmit this data back to the server through custom **Event** subclasses. If the developer would like the client to construct a GUI, which has components based on the state of the server or of any other player, **GUIResponse** should be subclassed to send this information to the client.

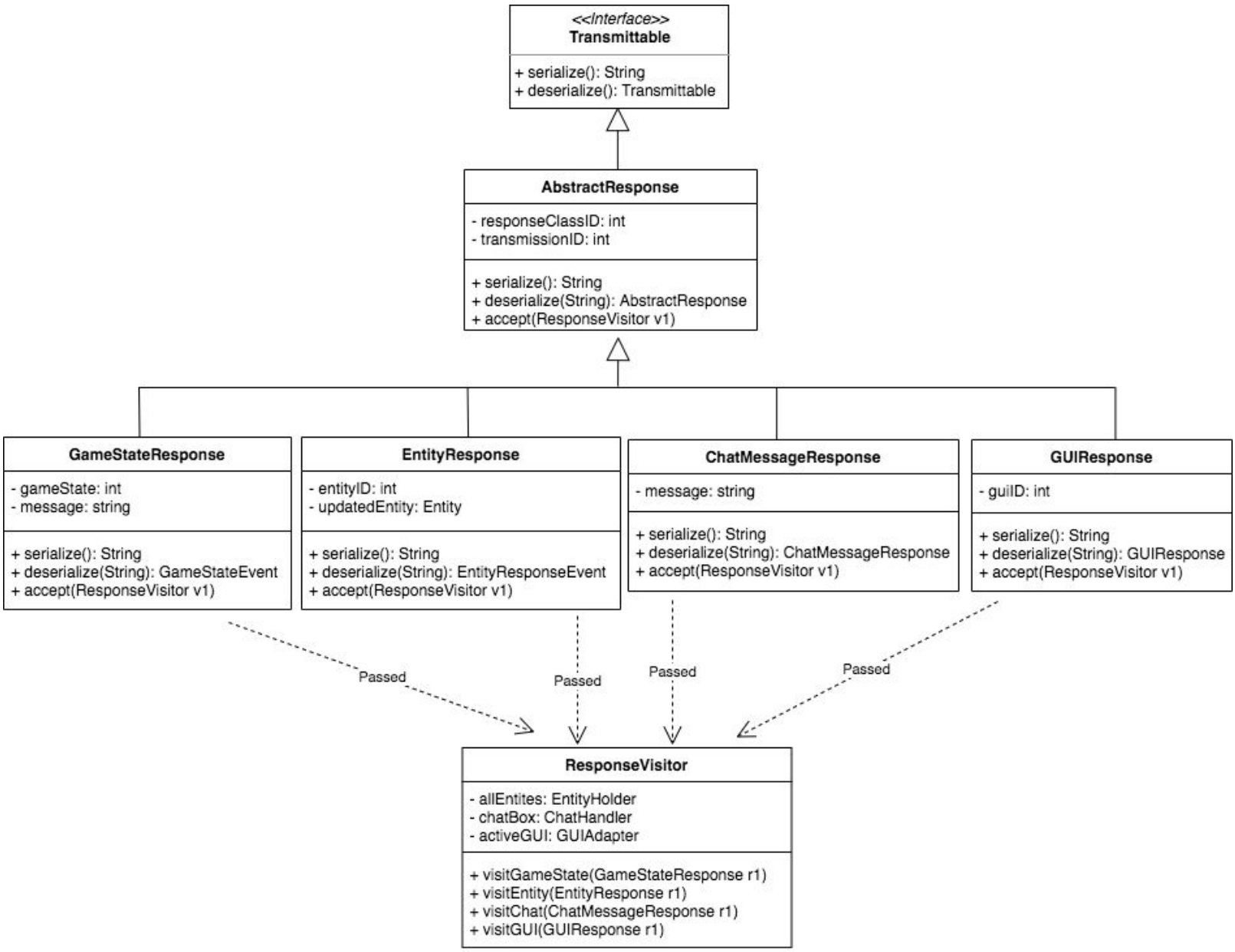
The specific mechanism of **Event** processing, as it relates to the rules and logic of gameplay, will be discussed later in the document.

The Authoritarian Server Implementation: Networking

Every concrete **Event** created and sent to the server is serialized by the client, and is deserialized, upon receipt, by the server. Each concrete **Event** subclass must have a unique eventClassID field to inform the server on which **Event** subclass to deserialize to. The transmissionID field refers to a unique integer dispatched from the client that, when a concrete **Response** is received and deserialized, will allow the client to pair the **Event** to the **Response**. Once the **Event** is received and deserialized on the server, the accept() method will be invoked and the **EventVisitor** object will perform the correct operation to process the **Event**, and send back a **Response** when all relevant processing and state changes have occured. Note that the **EventVisitor** is exclusive to the server.

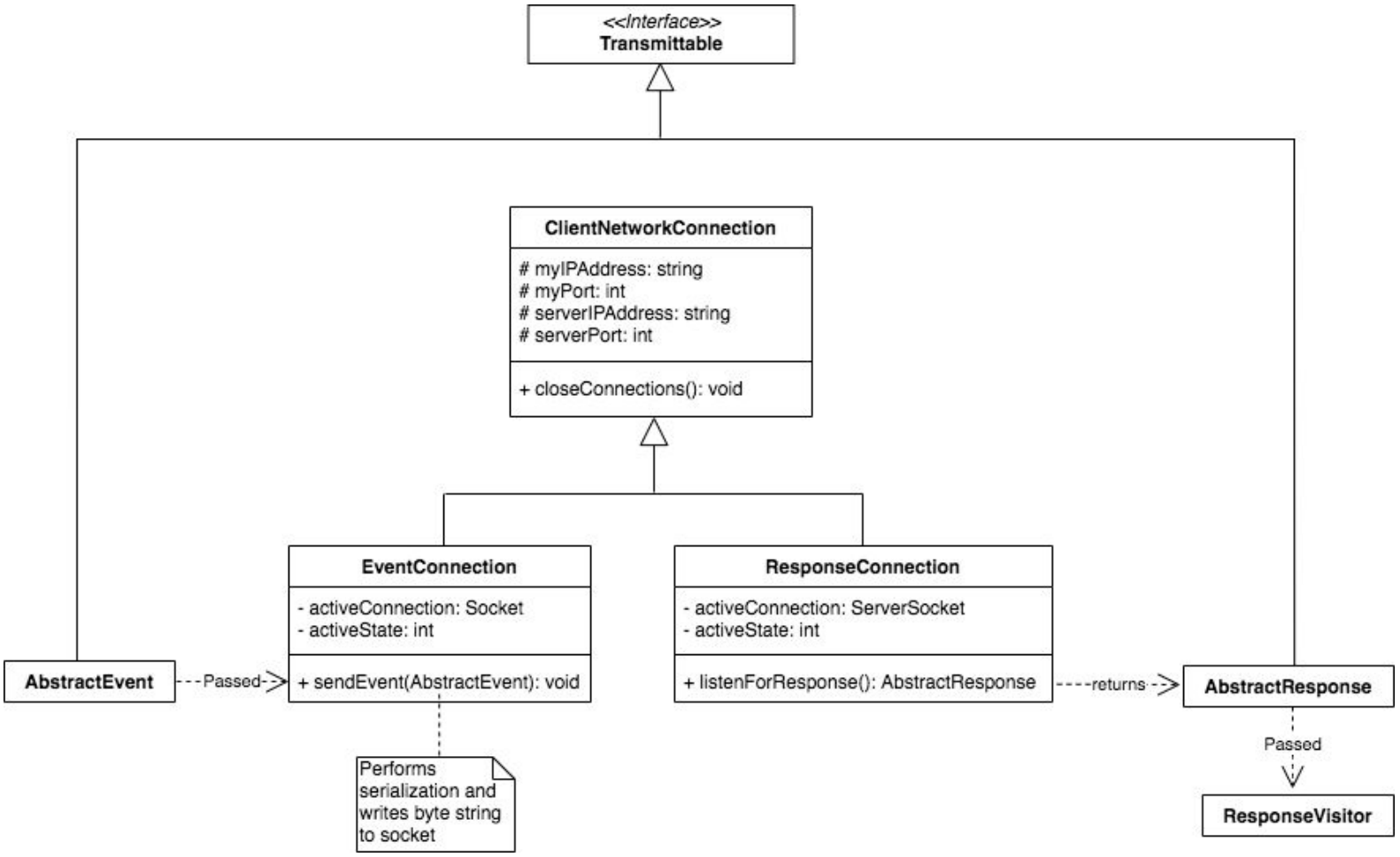


Likewise, the concrete **Response** classes have a similar format and are processed in a similar fashion. Note that the **ResponseVisitor** is exclusive to the client.

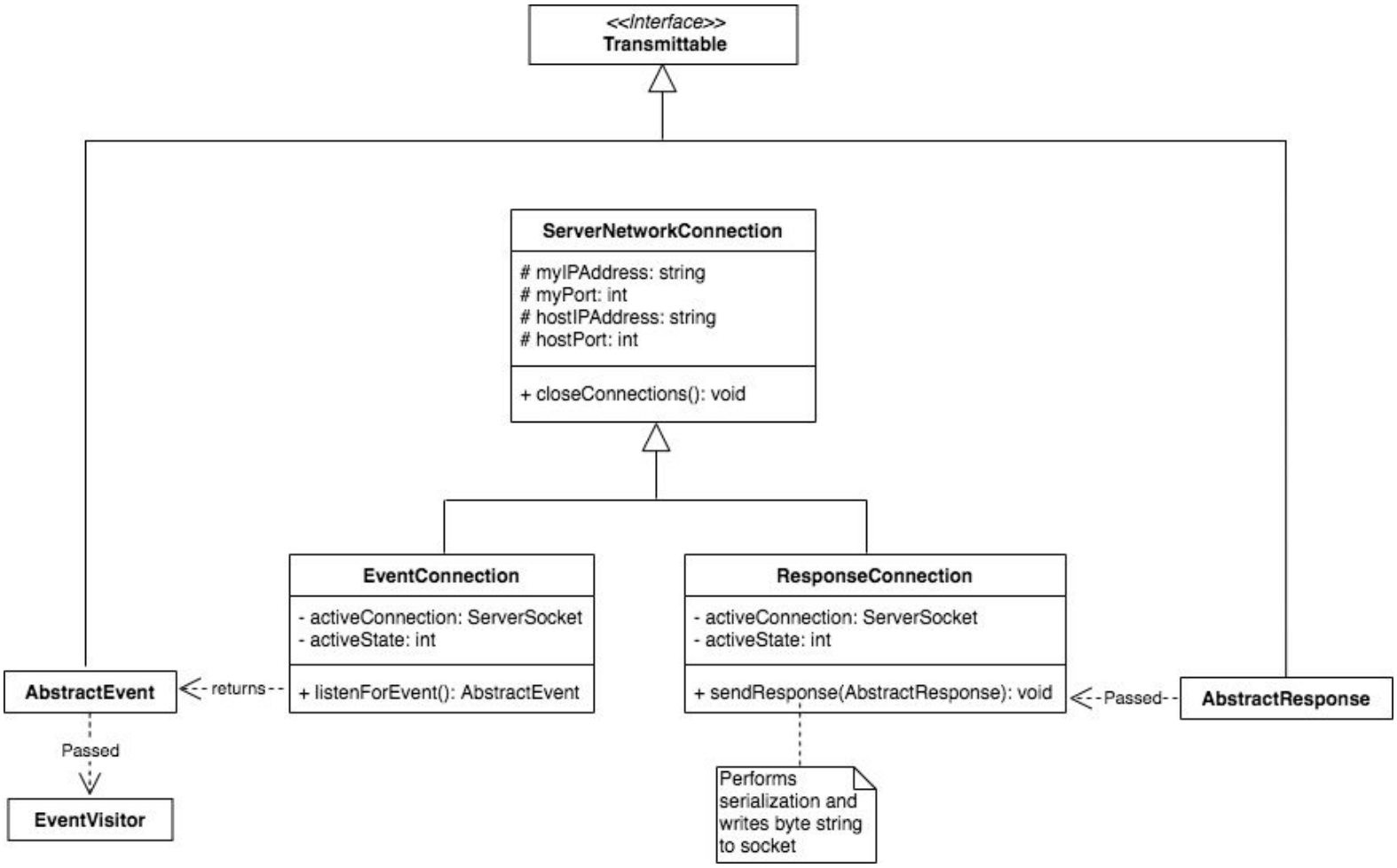


The **Event** and **Response** classes are designed to be complementary to produce as coherent an interface as possible for the game developers. The main purpose for the **Transmittable** abstraction is to allow the server to keep an itemized list of all network interactions for debugging purposes.

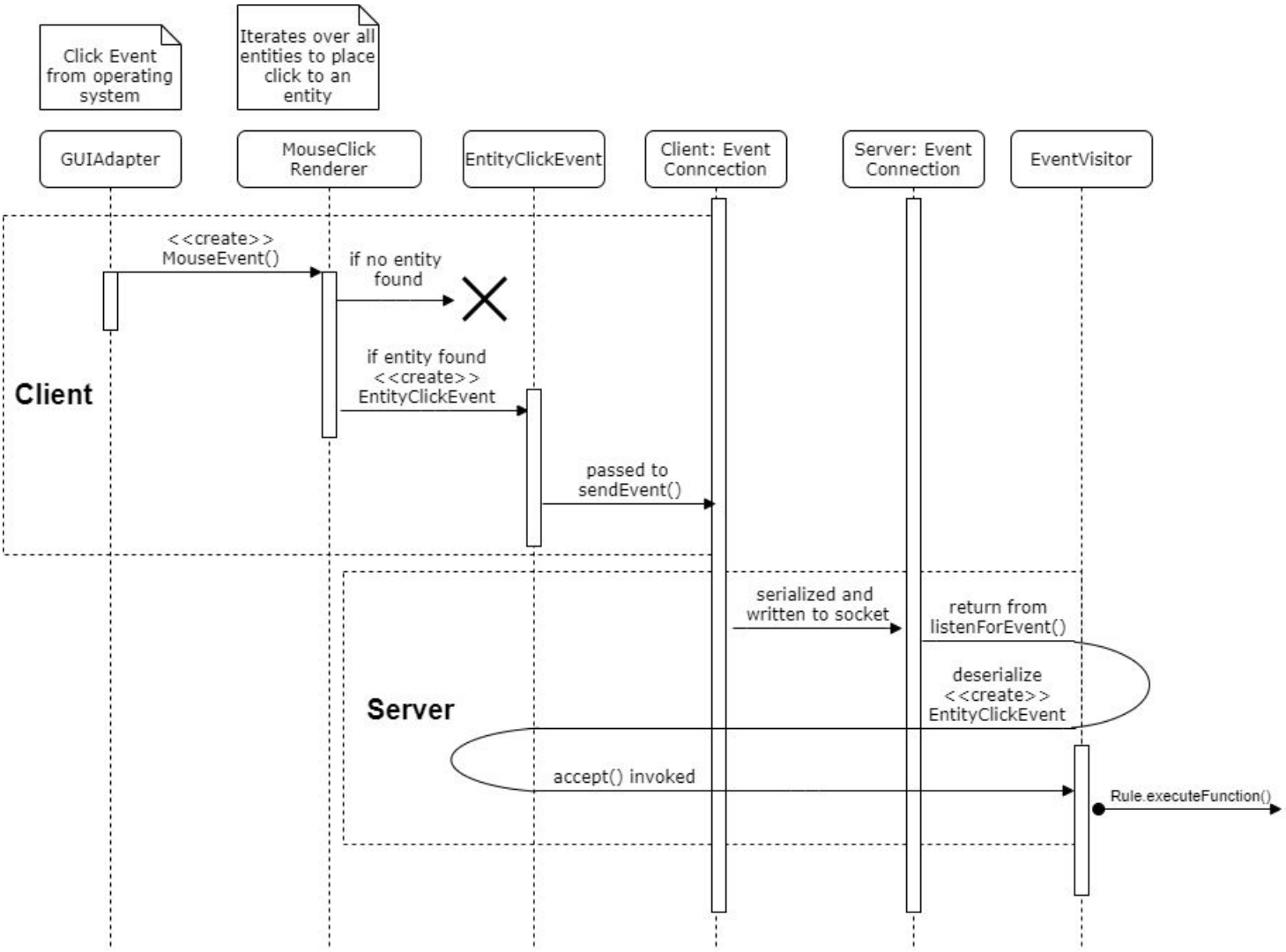
From a networking perspective, the client must include 2 active, asynchronous sockets to the server: an **EventConnection** and a **ResponseConnection**. The **EventConnection** is responsible for opening a socket, serializing, and writing an **AbstractEvent** to transmit to the server. The **ResponseConnection** is responsible for opening a server socket, accepting an input string, and serializing the string to the proper **AbstractResponse** subclass dictated by the responseClassID member of the **AbstractResponse**. Once the object is created, the client invokes the accept() member on the concrete **Response** with a reference to the **ResponseVisitor** to process the transmitted instructions and take the appropriate action.

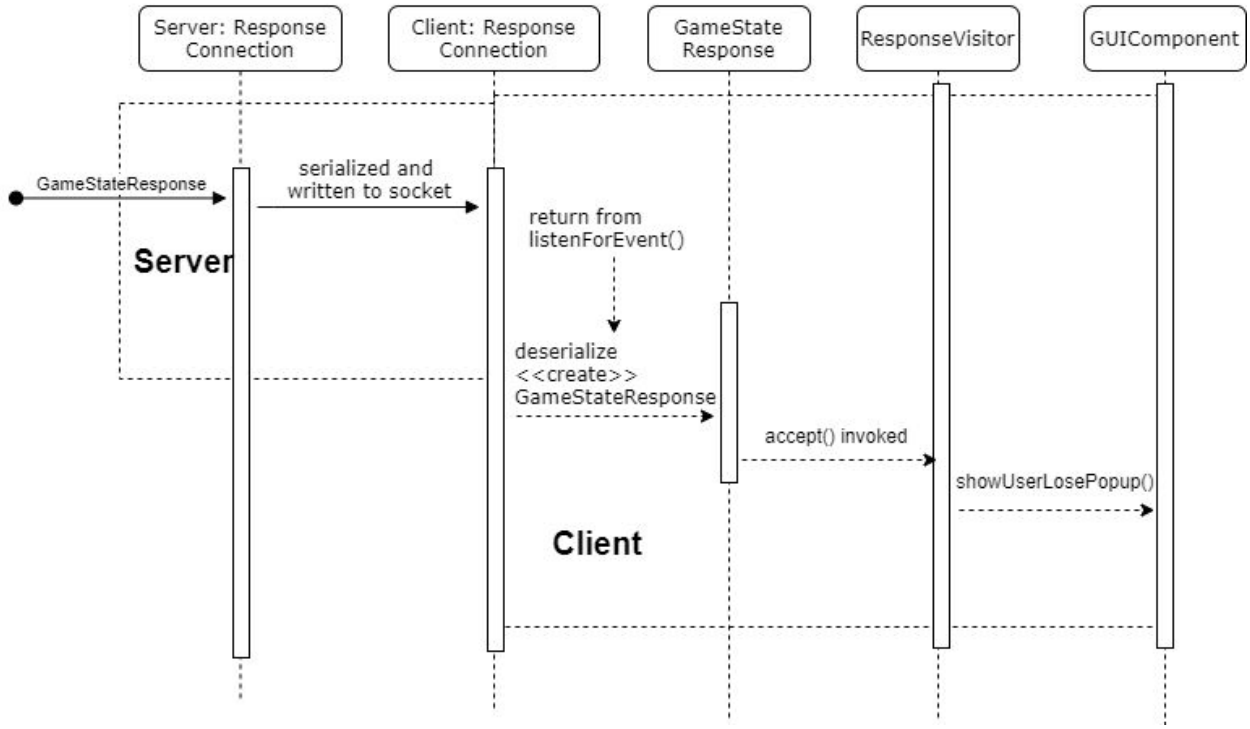


From the perspective of the server, the paradigm is effectively the same, just flipped. The symmetry of the design between server and client is no accident. In addition to reducing the number of protocols developers have to learn and implement, if developers wish to extend this design to include protocols of their own, they can reuse a considerable amount of code, as the protocol allows for transparency behind the client-server relationship.



Perhaps the most complicated yet crucial feature for the tabletop simulator is the ability for the client to interact with game **Entity** objects. Without any definition for the rules and state governing **Entity** objects, the client must update the server with the user input it collected, and wait for a response as for what that input actually meant. The first half of this information exchange is outlined below:





At this point in the exchange, the client has provided the server with all the relevant information and must wait for a response on if / how the **Event** changed the state of the game. This process occurs asynchronously, as the server may be getting and responding to **Event** objects from other clients simultaneously that nullify the effect of the transmitted **Event**.

As shown in the diagram, once processed, the server determined that the **Event** transmitted actually changed the state of gameplay. As such, it issued a new **GameStateResponse** object to inform the client that all players have lost. With this information, the client displays the preprogrammed showUserLosePopup() GUI defined in the **GUIComponent** subclass.

Functionally, the **GUIComponent** and **GUIAdapter** are abstract classes that represent an “insert implementation here” post-it note to final developers. The **GUIAdapter** contains purely virtual basic graphics methods, like drawRectangle(int x, int y, int length, int height), allowing the renderer to perform necessary operations assuming the platform dependent work is implemented. The **GUIComponent** abstract class simply builds compound graphics by defining a sequence of smaller sub graphics to be performed by the **GUIAdapter**.

Component Subsystem

At its most basic level, chess features a rectangular board with 2 sets of pieces on it. Each piece type, or **Component**, will be subclassed into the 6 categories of pieces: king, rook, bishop, queen, knight, and pawn. The **Component** object serves as the link between the **Entity** objects that the user interacts with on the screen, and the logical representation that represents the piece's state and valid moves to the rule's subsystem. Each piece will adhere to the **ChessComponent** interface defined below. It is particularly critical that the **ChessComponent** abstraction exists in chess because the game specifies that if a pawn is promoted to the farthest file, the player may choose a piece to replace the pawn. Thus, polymorphism within **ChessComponent** will allow for this functionality.

```
public abstract class ChessComponent {

     vector2D[] moveableLocations;
     vector2D[] attackableLocations;
     Entity clientRepresentation;
     boolean isWhite;
     boolean isKing; //Useful for discovering checks

}
```

The moveableLocations field refers to an array of two dimensional vectors representing the X and Y coordinate transformation relative to the current tile that describes the general locations of tiles the piece can move to. For instance, a vector containing <0,1> would represent a movement of the piece up one file. This does not mean that this particular move is valid, just that it is possibly valid. The rule's subsystem later will verify if it is valid or not.

The schematic shows the displacement vectors in the `moveableLocations` field. For every piece except the pawn, the `attackableLocations` field is equivalent. Vector addition can be used to produce a list of position vectors, rather than displacement vectors

Likewise, the attackableLocations field refers to an array of two dimensional vectors representing the X and Y coordinate transformation relative to the current tile that describe the general location of tiles that a piece can attack. For instance an array of vectors containing <1,1>, <-1,1> would describe an attack area diagonal of the current piece.

Board Subsystem

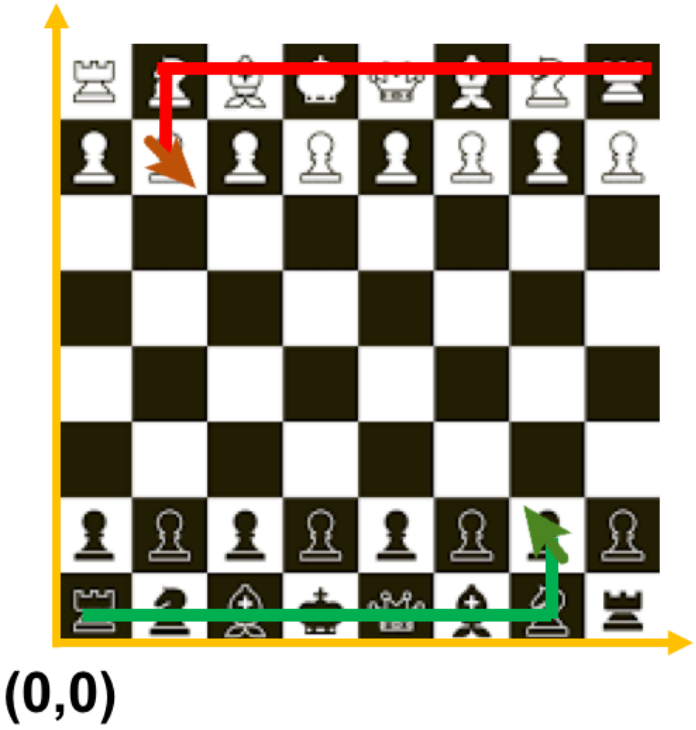
Shifting gears, the chess board is a crucial feature of the tabletop. As a physical representation crucial to the client's gameplay environment, the **ChessBoard** class will inherit from the **Entity** class, allowing it to be rendered and clicked. The **ChessBoard** class will contain a two dimensional array of **Tile** elements. Because the **ChessBoard** is static, a two dimensional array allows for easy random access to **Tile** elements and a predefined rectangular structure.

The **ChessBoard** class will serve as a medium to help other subsystems access **Components** by location. The **Tile** class serves as effectively a pointer to a **Component.**

```
class Tile {
     ChessComponent containedComponent;
     int x,y,width,height;
     boolean containsPoint(int x, int y);
}
```

Whereas the **ChessBoard** mostly serves a coordinate system to map clicks and vectors to tiles, and from tiles to components. It is important to note that the white and black pieces face opposite directions, and so the coordinate mapping system must reverse the vector of one color. This phenomenon is described below:

Thus, even though both pawns would contain the same moveableLocations vector, the Y component of the vector must be inverted for the white player



The same principle occurs when processing clicks. The white user's raw coordinates will not map directly to the black user's coordinates, even when describing the same square. The Y coordinate must be subtracted from the height of the board.

When an **EntityClickEvent** is created by the client, the click is mapped from a 3D perspective onto the 2D sprite. It is the **Board**'s responsibility to process what this 2D coordinate means within the context of the 2D sprite and the user's perspective.

```
public class ChessBoard extends Entity {

     Tile[][] board;

     Tile mapClickToComponent(EntityClickEvent e1, bool isWhite)
     {
         vector2D corrected = e1.getEndPosition();
         if (isWhite) {
              corrected.y=super.getHeight() - corrected.Y;
          }

          for (int i=0; i < board.length; i++)
          {
              for (int q=0; q < board[0].length; q++)
              {
                   if (board[i][q].containsPoint(corrected.x,corrected.y)
                        return board[i][q];
              }
          }
         return null;
     }

 Tile goToTileFromTile(Tile origin, vector2D displacement, boolean isWhite)
     {
         vector2D yTransform = displacement;
         if(isWhite) {
             yTransform.y = yTransform.y * -1;
         }
         return board[yTransform.x + origin.x][yTransform.y + origin.y];
     }

EntityResponse sendBoardState() {
            return new EntityResponse(1, this);

      }

   ....
}
```

The pieceUpdated() method should be invoked whenever a piece is successfully moved, attacks, or is attacked. It's **EntityResponse** should be handed to the **ResponseConnection** to be written and transmitted to all clients over the socket connection.