Code Inspection Report

for

Virtual Video Modeling on the Social Skills of Adults with Autism

Version 1.0 approved

For Sarah K. Howorth

University of Maine

March 23, 2024

Prepared by JamTech:

Tristan Cilley, Allison Lupien, Nick Sarno,

Jacob Michaud, Maha Fazli

Virtual Video Modeling on the Social Skills of Adults with Autism

Code Inspection Report

Table of Contents

Page

1. Introduction

This is a project for Dr. Sarah Howorth on virtual video modeling of the social skills of adults with autism. Dr. Howorth is the director of the PEERS® Lab at UMaine and our primary client for this capstone project. The PEERS® curriculum, a formal process for teaching social skills, was written by Dr. Elizabeth Laugeson. Our client's project proposal outlined a request to make the curriculum available in a virtual reality environment.

The JamTech team consists of five students currently enrolled in a two-semester (Fall 2023 - Spring 2024) computer science capstone course at the University of Maine. Each member of the team has volunteered for a specific team role which allows us to delegate assignments to everyone on the team.

1.1 Purpose of This Document

The document is a record of all aspects of JamTech's code review process. Each section of this document corresponds to a specific component of the review process. Section 1 includes JamTech's adopted coding conventions and a checklist of code defects which violate these conventions. Section 2 contains JamTech's personal impressions of the code review process as well as records of the individual inspection meetings. Section 3 is primarily a detailed list of every module that was critically reviewed. Lastly, section 4 contains a table in which every defect was enumerated, categorized, and documented. The intended audience of this document is our client, Dr. Sarah Howorth, and our capstone professor Dr. Laura Gurney.

1.2 References

- *C# at Google Style Guide*. styleguide. (n.d.). https://google.github.io/styleguide/csharp-style.html
- PEERS®. (2021). PEERS® (version 1.1.0) [Mobile app]. Apple Store OR Google Play. https://play.google.com/store/apps/details?id=com.peersclinic.peers&hl=en_US&gl=US
- PEERS (2023) UCLA PEERS® Clinic, Semel Institute for Neuroscience and Human Behavior. https://www.semel.ucla.edu/peers
- Tristan Cilley, Allison Lupien, Nick Sarno, Jacob Michaud, Maha Fazli. (2023). GitHub repository, https://github.com/VoloVita/SeniorCapstone/tree/main/Deliverables
- Tristan Cilley, Allison Lupien, Nick Sarno, Jacob Michaud, Maha Fazli. (2023). System Design Document (SDD), version 1.0

- Tristan Cilley, Allison Lupien, Nick Sarno, Jacob Michaud, Maha Fazli. (2023). Systems Requirement Specification (SRS), version 2.0

1.3 Coding and Commenting Conventions

For coding and commenting conventions, we have decided to follow the guidelines set by Google's style guide for C# code. This guide was developed internally by Google and has since become Google's default guide. These guidelines include naming conventions, file naming conventions, whitespace rules, and overall organizational conventions.

Link to Source: https://google.github.io/styleguide/csharp-style.html

The naming convention for classes, methods, enumerations, public fields, public properties, and namespaces is PascalCase, where the first letter of each word is capitalized. We will be using camelCase for local variables and parameters, where the first letter of each word, except the first word, is capitalized. _camelCase with an underscore at the beginning will be used for private, protected, internal, and protected internal fields and properties. For all of these previously mentioned naming conventions, a 'word' is defined as anything written without internal spacing.

Names of all interfaces will begin with 'I'. Files, filenames and directory names will use PascalCase, and when possible, file names should match the name of the main class within the file. Our files should generally be limited to one core class per file. Whitespace will limit us to one statement per line, and we will intent with tabs, which are five spaces wide, except when in yaml files. Additionally, all trailing whitespace is trimmed to keep the code concise. The overall organization of the guide states that namespace 'using' declarations always go at the top of the file. Class members also have a specific order which they must follow, beginning with nested classes, enums, delegates, and events; static, const, and read-only fields; fields and properties; constructors and finalizers; and finally methods.

1.4 Defect Checklist

In this section, we've provided an organized table of all defects found in our code through the inspection process.

Table 1. Defect Checklist: *A list of common defects found in code.*

#	Defect	Category	Coding convention	Produces Errors?
1	improper usage of Update()	Logic Error	Update is called every frame, this is not suitable for all operations and can cause error	maybe
2	Improper usage of Start()	Logic Error	Start is called upon program startup and is not suitable for all operations and can cause errors.	maybe
3	Improper _camelCase	Naming Convention	Applies to private, protected internal and protected internal fields and properties	no
4	Improper PascalCase	Naming Conventions	Applies to: classes, methods, enumerations, public fields, public properties, namespaces	n/a
5	Improper camelCase	Naming Conventions	Applies to local variables and parameters	no
6	Non-Descriptive Naming	Naming Conventions	Chosen names should be useful and descriptive	no
7	Improper Acronyms	Naming Conventions	All character in acronyms should all be capitalized	no
8	Improper File names	Naming Conventions	Filenames and directory names are PascalCase	no
9	File Structure	Organizational	In general, prefer one core class per file.	no
10	Lack of documentation	Organizational	Every non-standard method should have documentation	no
11	Class Variables not defined at the top of class definition	Organizational	variables should be defined at the top of	no
12	Public and Private Variables Mixed	Organizational	Variables should remain private unless they are needed to be public, and that	no

			declaration should remain consistent	
13	Incorrect public/private class definition	Security Oversight	Only certain needed information should be public whereas almost all other variables should	no
14	Personal Identifiable Information not obfuscated	Security Oversight	All sensitive data should not be kept in plain text	no
15	Improper whitespace	Whitespace Conventions	One statement per line, indentation with tab (5 spaces)	no
16	Trailing whitespace	Whitespace Conventions	Avoid excessively spaced out formatting.	no

2. Code Inspection Process

The goal of a code inspection is to review the code of the project, making sure everything is clean, well organized, functional, and bug free. This process focuses on locating bugs and defects within the code so that they can be fixed, and analyzing the internal issues within the code. Our code development is currently still in progress. Within these code inspections, we analyzed our scripts for convention issues, errors, and security oversights.

2.1 Description

Our team began every meeting with a friendly and casual atmosphere as we noted the date, time, and location. After everyone was settled and ready to focus, we started with a brief overview of the agenda for our code inspection meeting. This agenda began by reviewing the defect checklist we constructed. We also identified the modules that would be inspected during that meeting, and which sections of code we would be talking about in detail.

After the overview, we would assign roles and responsibilities for the meetings. These roles included presenters and notetakers. One person in the team was a presenter for each code inspection review, and one person was a notetaker. Everyone else listened and offered input and feedback throughout the course of the meeting.

Code inspection began with the presenters introducing their code, walking the rest of the team through it and ensuring everyone understood the content and functionality. This presentation of code would be followed by a Q&A and group feedback. The process of recording action items would follow, where we discussed as a team what the defects were and how to address them. The notetaker for the meeting would note all defects discussed, and as a group we would decide how to avoid each of them going forward. At the end of the code inspection review, we would summarize all modules analyzed throughout the meeting and make sure there were no outstanding questions.

2.2 Impressions of the Process

Overall, our team has a positive impression of the code review process. It's a great method for making sure the whole team has seen every aspect of the project, and it's good practice to see code written by other members of the team with enough understanding to be able to pick it apart. The code review process encourages better documentation of code and it ensures that our code is more consistent across the project, and that the code itself contains fewer errors. This makes our code structure and formatting more consistent. However, the process was boring as there were some files that yielded little-to-no defects or errors after review. The process of checking for every possible defect felt tedious. This does not, of course, negate the necessity of critical analysis.

Additionally, resolving a defect often led to other errors throughout the file and the project. For instance, if we decide as a team to change the name of a specific variable to meet conventional standards, we then have to go through every bit of code to update all references to that variable, and then re-run our testing to ensure that nothing else was broken.

If there was one thing we could do differently as a team, we agree that we would have set convention standards before beginning the coding process specifically for the reason mentioned previously. Defining naming conventions before programming reduces the possibility of having to go back and make changes later. We believe this would have saved us a significant amount of time. In terms of our units, our best modular unit would be Jsonconfig.cs. This unit is small and simple, and therefore yielded few errors. On the other hand, our worst unit was by far lesson1test.cs due to its large size and complexity. We found the most errors in this unit, and it's the most likely to continue having errors in the future as development continues.

2.3 Inspection Meetings

Our group has had two code inspection meetings for the purpose of reviewing and discussing our code in depth. Most of our code is C# scripts, but a lot of development involves standard components of the Unity Editor, which means that not everything can be reviewed as code.

We spent our meetings reviewing our scripts and discussing all possible defects and errors within them. All members were present for the code inspection review and all members participated in reviewing the presented scripts.

Table 2. Inspection Meetings Held: *A brief description of each code inspection held, the location, time, the participants and their roles as well as the code units reviewed.*

Date	Location	Start	End	Performed By & Role	Participant/s	Particular code unit covered
2/28/24	Library Study Room - in person	5:30pm	6:30pm	Jacob - presenter Maha - notetaker	Tristan Cilley, Jacob Michaud, Maha Fazli, Allison Lupien, Nick Sarno	Player.cs Jsonconfig.cs PlayerData.cs
3/4/24	Library Study Room - in person	5:00pm	6:30pm	Jacob - presenter Nick - notetaker	Tristan Cilley, Jacob Michaud, Maha Fazli, Allison Lupien, Nick Sarno	Lesson1test.cs LessonData.cs SaveSystem.cs

3. Modules Inspected

This section contains a detailed description of each module that was evaluated during our code review process. Every description contains an SDD reference which explains whether the class is new or a modified version of a class outlined in the SDD.

Module Name: Player.cs
Description: A custom script for saving and loading player data.
Context: This script defines the Player class. It contains two public methods which are used for saving or loading the state of the player.
SDD Comparison: This class most closely represents the PlayerObject class defined in figure 2.2 of our SDD. In addition to storing the player's position, this class will be expanded to save and load additional detail from PlayerData.cs

Module Name: Jsonconfig.cs
Description: This module handles the loading of lesson data from a JSON file.
Context: Jsonconfig.cs loads the lesson data from a JSON file located at the path within the project's asset directory.
SDD Comparison: In the SDD we planned to use a database to store lesson content. We are using JSON files in the place of having the Lesson Content Database. The video content is

now stored in our Assets folder and the JSON file contains the file paths for each lesson's videos.

Module Name: PlayerData.cs
Description: This is a custom script that holds player data such as the lesson the player is on and the player's position in the scene.
Context: The PlayerData class holds the players state information and the Player class can access this information as needed.
SDD Comparison: This class contains information from the CurriculumView class seen in figure 2.2 from the SDD. Mainly the lesson variable is synonymous with the userCurriculumProgress variable. It has the additional variable for the player's position. It has the potential to hold additional player data as development progresses that may include data from the Lesson and Quiz classes from the SDD.

Module Name: lesson1test.cs
Description: This is a custom script that allows for dynamically loading and allocating data into the lesson panel within the scene.
Context: The lesson1test class loads the JSON file containing the lesson data based on an integer value passed into the main function of the class: lessonLoad(). Supporting functions include one for loading video sprites into the scene from JSON: LoadSpriteFromFile(), and for changing the video displayed in the video view: ChangeVideo().
SDD Comparison: This class is not specifically mentioned in the SDD, but does contain the functionality mentioned in the LessonView scene. This class takes the methods, (selectlesson(), selectvideo(), and getlessonprogress()) mentioned from the lessonview scene and combines them into a fully functional all-in-one implementation.

Module Name: LessonData.cs
Description: Two classes detailing specific aspects of each lesson.
Context: This module assists with importing lesson data from JSON files into the framework of the lesson view.
SDD Comparison: This class was originally listed as LessonInfo in our SDD. This script deals with linking the lesson data to the actual lessons, but whereas in our SDD we planned to pull the information from a database, LessonData pulls from JSON files containing the lesson information.

Module Name: SaveSystem.cs
Description: A class which provides methods for saving and loading the player's state
Context: This class contains two functions; SavePlayer() writes all the player's data to a file after serializing it, PlayerData() deserializes the contents of the save file and loads it into the current session.

SDD Comparison: Neither this class nor its component functions were outlined in the SDD. This is a novel class for our project. Figure 2.2 of our SDD includes a class for players which contains a field for the player's position in the scene. Our current implementation saves the player's position but its functionality has been expanded to include other aspects of the current state.

4. Defects

This section contains a table of all defects discovered in the code review process. Every defect has a general and specific categorization as well as references to its exact location in the code.

(Table on following page)

Table 3. Found Defects: *A list of all defects found during the code review meetings including the module or file they appear in, the kind of defect it is, what the defect is, the component it applies to, and the line of code it is found.*

Module Name	Defect Category	Defect Specific	Component Name:	Line #
Player.cs	Naming Conventions	Non-Descriptive Naming	variable: lesson	**7**
Player.cs	Organizational	Lack of documentation	function: SavePlayer()	17
Player.cs	Organizational	Lack of documentation	function: LoadPlayer()	23
Jsonconfig.cs	Logic Error	Improper usage of Start()	function: Start()	12-14
Jsonconfig.cs	Organizational	Lack of documentation	function: LoadLessonData()	16
PlayerData.cs	Naming Conventions	Non-Descriptive Naming	variable: lesson	9
PlayerData.cs	Naming Conventions	Non-Descriptive Naming	variable: position	10
PlayerData.cs	Organizational	Lack of documentation	function: PlayerData()	13-20
lesson1test.cs	Naming Conventions	improper naming PascalCase	variable: testobject variable: lessonPannelcontent variable: header variable: description variable: non_example_content variable: example_content variable: moreinfo public class: lessonload(int i)	11 12 13 14 15 16 18 48

Module Name	Defect Category	Defect Specific	Component Name	Line #
lesson1test.cs	Naming Conventions	Non-Descriptive Naming	variable: videop variable: butt variable: badVid	18 98 and 128 96
lesson1test.cs	Naming Conventions	improper camelCase	variable: videoreset variable: videoreset2 variable: badthumb variable: goodthumb variable: goodvideocount variable: badvideocount	53 54 59 60 61 62
lesson1test.cs	Naming Conventions	improper _camelCase	variable: lessonData	17
lesson1test.cs	Naming Conventions	improper filenames	file: lesson1test.cs	9
lesson1test.cs	Organizational	Public and Private Variables Mixed	all public variables	
LessonData.cs	Naming Conventions	improper camelCase	variable: non_example_videos variable: more_info variable: example_videos variable: quiz_url	12 14 15 17
LessonData.cs	Organizational	File Structure	public class: lessonData public class: Lesson	4 10

Module Name	**Defect Category**	**Defect Specific**	**Component Name**	**Line #**
LessonData.cs	Organizational	Public and Private Variables Mixed	variable: lesson1 variable: lesson2 variable: title variable: non_example_videos variable: description variable: more_info variable: example_videos variable: exercises variable: quiz_url	5 6 11 12 13 14 15 16 17
LessonData.cs	Naming Conventions	Improper PascalCase	public class: lessonData	4
SaveSystem.cs	Organizational	Lack of Documentation	public class: SavePlayer()	**7**
SaveSystem.cs	Organizational	Lack of Documentation	public class: PlayerData()	19
SaveSystem.cs	Whitespace Conventions	Trailing Whitespace	public class: PlayerData()	29

Appendix A – Team Review Sign-off

This is the team review sign off meaning that all current team members of JamTech (Tristan Cilley, Allison Lupien, Nick Sarno, Jacob Michaud and Maha Fazli) have fully reviewed and read the code inspection report and do agree with the content and format included in the document.

By typing one's name under the signature column and giving the date, the individual signs this document.

Name	Signature	Date
Allison Lupien	Allison Lupien	03/23/24
Comments:		
Jacob Michaud	Jacob Michaud	03/23/24
Comments:		
Maha Fazli	Maha Fazli	03/23/24
Comments:		
Nick Sarno	Nick Sarno	03/23/24
Comments:		
Tristan Cilley	Tristan Cilley	03/23/24
Comments:		

Appendix B – Document Contributions

This is the current contribution of each team member towards the CIR.

Name	% of contribution
Allison Lupien	20% [Coding and Commenting Conventions, References, Defects Checklist, PlayerData.cs, editing]
Jacob Michaud	20% [Coding and Commenting Conventions, Defects Checklist, Lesson1test.cs, editing]
Maha Fazli	20% [Jsonconfig.js, Defects Checklist, editing, formatting]
Nick Sarno	20% [Impressions of the Process, Inspection Meetings, LessonData.cs, editing, proofreading, defects checklist]
Tristan Cilley	20% [Player.cs, SaveSytem.cs, proofreading, editing, intro paragraphs]