CORE5

INITIAL DESIGN REPORT

Alper DAŞ
Josif GRABOCKA
Mustafa KAYNAK
Semih KEKÜL
M. Sabri KAYA
1. INTRODUCTION

1.1. Project Scope

1.1.1. Project Description

Project Core5 is a Massive Multiplayer Online Role Playing Game project. The scenario, defined for the game, affects the intent of the game. The game will proceed according to our scenario which will be described in the next parts of this document.

1.1.2. Project Features

Basically, from the design point of view our project will be composed of:

Network: to make multi-playing possible we are going to design the network of the game. The server side will be performed on Linux operating system and the client side will be performed on Windows operating system. Since the big and challenging part of our Project is this network topic, we will implement a complete model of a multi-client interacting server. On the other hand, to make the game more realistic in terms of response refresh times, synchronization is also a big issue that will be considered in particular.

3D Graphics: to build a powerful interface which will help both understandability and functionality modern games make use of complex graphic components. In our Project, we will pay big attention to developing rich user interfaces and attractive graphics by making use of Microsoft DirectX.

Artificial intelligence: to make the game more realistic, the living creatures will be created. In order to develop an attractive scenario those creatures should have the ability to think. Thinking will of course be implemented by our AI engine. But in order not to increase the complexity and avoid redundant processing time, the AI engine should provide an efficient algorithm and design.

Game Physics: in the light of being realistic and attractive, multiplayer game should provide well calculated and smooth movements. Beside that, it should check whether a specific move violates the physical rules of the game world. For example: collision detection, speed calculations, flight trajectories etc.
**Sound System:** to attract the player and make them feel excited and embedded in the scenario, rich sounds and themes will be provided. Each movement and character action will be associated with a specific sound. Furthermore, the depth of the sound will be calculated in order to give a 3D sound surrounding environment. Sound system will be accomplished with the help of OpenAL.

### 1.2. Design Constraints and Limitations

#### 1.2.1. Performance

We plan to distribute our game to a large audience varying from modern architecture to old systems. So performance is a vital task for providing a smooth execution even in the old architectures having limited resources. Hereafter it is the developer’s mission to provide an efficient and well-structured design. Optionally the developers may consider valuable the possibility of offering a number of different releases, designed to be executed on different target systems distinguished by their resource richness (DDRAM, Video Ram, CPU, etc.).

#### 1.2.2. Time issue

The game is due the last days of the spring semester 06/07, so we have an important time limitation. For this reason we are forced to keep the level of details at a certain threshold. However we will make it possible to provide as much details as it is required for a normal implementation of the proposed scenario.

### 1.3. Design Goals/Objectives

#### 1.3.1. Usability:

The user should be able to control the game without constraint. Since visual and audible aids are important for the player to comprehend the game, these aids should be attractive and understandable. Menus should be designed as clear as possible in order to prevent user lost in menus. Game flow and game scenario should be smooth in order to make the game adaptable.
1.3.2. Reliability:

Breakdowns should be removed. The game is reliable as far as it is error free so almost all the bugs should be debugged. Since multithreading will be used, all the possible deadlocks should be determined and then prevented. Network connection should not be dropped unexpectedly. Uncontrolled cheating will be prevented in order to provide justice.

1.3.3. Security:

The network security should be accomplished faultless. In order to ensure the money transfer safely, it is necessary to contact with professional and reliable corporations.

1.3.4. Supportability:

The game will be up to change and the updates will be distributed via internet. Production defects will be compensated.
2. DESIGN

2.1. Dataflow diagrams
CLIENT DFD1

Display: 
- Keyboard & Mouse Inputs
- Formatted Inputs
- Received Data
- Sent Data

Audio: 
- Keyboard
- Mouse

SERVER

INPUT

Client DFD1
Server DFD1

- Bot's Nearby
- Bots' Action Position
- Bots' Attributes
- Bot ID
- Updated Bot Statuses and Best Moves
- Position of Last Action

AI-1-1

- Request Position of Last Action
- Position of Bot

AI-1-2

- Move Validity Confirmation

AI-2-1

- Bot's Nearby
- Bots' Action Position
- Bots' Attributes
- Bot ID
- Updated Bot Statuses and Best Moves
- Position of Last Action

AI-2-2

- Request Position of Last Action
- Position of Bot

AI

- Move Validity Confirmation

Client
GAME DATA

CLIENT SOUND DFD 2

- SET SOURCE VALUES: Request for setting source values
- PARSE FILE: Sound File Names
- LOAD SOUND FILES: Source values OK
- SET LISTENER VALUES: Listener Values OK
- VALIDATE PLAYING: Audio output
- INITIALIZE: Request for setting listener values
- Properties of objects that make sound
- CHARACTER PROPERTIES
- Speaker
- Game Engine
- File containing list of sound files
GAME ENGINE

INITIALIZE

collision detection

NET FORCE CALCULATION

WORLD ITERATION CALCULATION

GAME DATA
2.2. State Diagram

CLIENT NETWORK STATE DIAGRAM

GAME ENGINE

RECEIVE FROM GAME ENGINE

Action Class

CLOSE CONNECTION & RETURN KILL COMMAND

Action Class

If not close message

If close message

SEND DATA TO SERVER

SERVER

RECEIVED MESSAGE

VALIDATION

If valid

If invalid

SEND TO GAME ENGINE

SERVER

RECEIVED MESSAGE
2.3. ER Diagram

2.4. Sequence diagrams
2.4.1. Login sequential diagram

```
2.4.2. Trade sequential diagram

[reply==TRUE]cGameState::setState (SELECPROFILE)
```
2.4.3. Item picking sequential diagram
2.4.4. Attack sequence diagram
2.5. Class diagrams
class cCharacter
{
    unsigned long hitpoint;
    unsigned long manapoint;
    unsigned short level;
    unsigned long experience;
    cInventory inventory;
    cCharacter party;
    float strength;
    float agility;
    float intelligence;
    float wisdom;
    float charisma;
    float endurance;
    float constitution;
    float class;
    float skillList;
    float equippedList;
    float fame;
    float stamina;
    float class;
    float skillList;
    float equippedList;
    float fame;
    float stamina;
    void sendTradeRequest();
    void setHitPoint(float hitPoint);
    void setManaPoint(float manaPoint);
    void equipItem();
    unsigned long getHitPoint();
    unsigned long getManaPoint();
    void set*Value(float *value);
    float get*Value();
}

cInventory
{
    uint capacity;
    uint weight;
    uint totalValue;
    void setCapacity(uint capacity);
    uint getCapacity();
    void setWeight();
    uint getWeight();
    void addItem(cItem item);
}

cNPC
{
    cAIObject ai;
    void ai_state();
    void party();
}

cPC
{
    cNPC ai;
    cPC party;
}
The game will have a background music. Moreover, a simulated sound world be implemented.

Design Decisions:
- OpenAL API will be used.
cPhysicsEngine calculates the next step of every object in the physical world. The simulation is a combination of two sequential operations: collision detection and net force calculation. In final step engine sets new physical values of objects.

Design Decisions:
- ODE will be used for simulating...
3. USE CASES

Admin use case

- Kick Player
- Communicate
- Transport Player
- Ban Player
- Monitor
- View world
User use case

Movement
- Turn
- Walk
- Run
- Jump
- Teleport
- Strafe

Attack
- Hit
- Spell

User
- Loot
- Join Party
- Create Party
- Use Inventory
- Grab
- Die
- Respawn
- Feed
- Trade
- Speech
4. NETWORK DESIGN

4.1. SERVER

4.1.1. Server Functionality Scenario:

Server functions in a multi-threaded fashion, making use of POSIX “p thread” library. There will be primarily those kinds of threads:

- **Connection Listener**
- **Query Listener**
- **Request Handlers for each client**

1. **Connection Listener Daemon Thread:**
   This thread, (actually working like a synchronized daemon), will handle new connections. New connections will be caught by UNIX sockets' listen routine. After catching the new connection this thread will read all identification information sent by the client. By identification information we mean name, password, etc. After checking the validity of this account the login status of this client will be updated. In case the login information is not valid the connection listener thread will send a negative reply to the client, otherwise will send a confirmation code. Next this thread will fork off a new request handler thread for the new connection, (see 1.3). This thread will keep on repeating the connection listening routines until explicitly stopped by the server.

1.2) **Query Listener Daemon Thread:**
   This thread, also working in a daemon fashion, will handle clients' requests. The thread will iterate to listen for incoming requests from currently active clients. The requests, (any type of action inquiry), will trig the execution to the corresponding thread (there will be one thread for each open connection). So, this thread will wake up the request handler thread and will sleep until the request is handled. This thread will keep on this executing cycle until stopped by the server.

1.3) **Request Handler Threads:**
These threads will handle each client's sent request by communicating with the game engine. After sending the requested action to the game engine, the threads will wait for the engines' response to this action. When the response is ready the request handler will sent the responses to each related client. Afterward the threads will sleep on their semaphores, i.e., will wait until another request has come.

An pictorial description of the 'new connection' and 'new request' functionalities for the server's scenario is shown below:

4.1.2. Algorithmic Description of the thread handlers

2.1) Connection Listener Daemon Thread:

Repeat from 1 to MAX_CONNECTION_NO

1. Listen for the new connection
2. Read the login info from the connection
3. Validate the login info from the server's internal database
4. if( login info is not valid )
   {}

4.1. Send a negative confirmation to the client

else
{

4.1. Send a positive confirmation to the client

4.2. Increment the no of connection counter by one via a special method making use of mutexes. This issue is of vital importance since all threads will make use of this counter, consequently there won’t exists some possibility for the race condition to cause any data hazards.

4.3. Set the socket flags non-blocking to avoid blocking the thread in case data is missing on that socket. This aspect is important in the query handler thread which continuously probes for requests from the connected socks.

4.4. Fork a new thread giving its identification arguments consisting of the connection socket id and the client name.

5. Sleep to yield the scheduling to other threads

2.2) Query Listener Daemon Thread:

1. read current no of connections (via a mutex protected method)

repeat from 1 to CURRENT_NO_OF_CONNECTIONS
{

2. read from the sock(at iteration index), whether there is any request coming on that channel.

If( read is successful )
3. wake up the request handler thread $(iteration index)$, by issuing a signal call to the semaphore it is waiting.

4. If the query listener thread has been running more than a specified amount of time stop it, to give a look at new incoming connection. In other words, to yield the execution to the connection listener thread.

2.3) Request Handler Threads:

1. Read the requested action from the socket associated with the current request handler thread.
2. Send the action to the game engine and then wait until responses are ready.
3. After reading the resulting replies from the game engine, send the replies to each affected client via each client's socket.
4. wait on the appropriate semaphore
4.1.3. Server’s Data Flow Diagram

Server’s data flow diagram is primarily characterized by alternating between reading clients requests and servicing them via communicating with the game. In the honor of this task, we implement reading by making use of the socket reading routines in the connection listener thread. Hereafter the query listener iteratively probes for a request from the active connections. In case it catches a request it signals a semaphore and switches the context to the request handler thread. The request thread reads the request, sends it to the game engine, waits for replies and transmits the replies to affected clients.
4.1.4. Server’s State Transition Diagram

Control flow in the server flows like the following: Server instance forks two threads namely the connection listener and the query listener. Only one of them will execute, of course, so we made use of a concurrency node to demonstrate that. It doesn’t matter in fact which one of them executes, since they alternate by issuing sleep routines. After that the query listener probes until catches a request. As it catches the request it passes the control to the appropriate request handler thread by signaling the appropriate semaphore. The request handler satisfies the request by calling the game engines, sends the replies to affected clients and waits on its semaphore. The control then returns to the query listener keeping on repeating this interaction. Once in a defined time slice the query listener gives execution to the connection listener to avoid starvation.
4.2. Client

4.2.1. Client Functionality Scenario:

Client functions which is in a multi-threaded fashion, is composed of two threads, namely:

- Sender
- Receiver

Before starting the explanation of these threads, firstly the connection process at the client-side will be explained at this part.

Connection Process

Since we use connection-based method, connection between client and server must be established before communication. Firstly, we create a socket in order to handle the communication(sending and/or receiving the messages, the actions) between the client and server. We used stream-based full-duplex communication(SOCK_STREAM) as the type of the socket, adress format is host and port number (AF_INET) as the domain argument while creating the socket. Then, the connection process starts. We get the IP address of the server from its name. Then we try to connect with the created socket ID, given host adress and given host port. After the establishing of connection, first the name and password of the user is sent to the server, one time at each connection. These information are checked at the server side and after the validation, the connection is established with the given user name. The online list which is held at the server-side is updated.

Sender

This thread will handle the action and message sending process. Sending process of actions and messages is repeated in an infinite loop in the “sending function” during the connection, until the user leaves the game which closes the socket of the client. This infinite sending action is controlled with the help the multi-threading. Sending thread is killed and waked up repeatedly. When the client closes the connection the online list which is held at the server side is updated.
**Receiver**

This thread will handle the action and message receiving process. Receiving process of actions and messages is repeated in an infinite loop in the “receiving function” during the connection, until the user leaves the game which closes the socket of the client. This infinite receiving action is controlled with the help the multi-threading. Receiving thread is killed and waked up repeatedly. When the client closes the connection the online list which is held at the server side is updated.

**Algorithmic Description**

**Getting connection info**

1. open the file which has the host info and is given by us
2. if the file cannot be opened successfully throw error
3. if succeeded read the information from the file namely, host adress and host port.
4. close the file

**Creating the socket**

1. use host and port number (AF_INET) adress format as the domain argument of the socket
2. use stream-based full-duplex communication(SOCK_STREAM) as the type argument of the socket
3. call the socket() function
4. if the socket cannot be created throw error
5. if successful get a file descriptor (socket ID) and save this info

**Connecting to the server**

1. get the IP adress of the host from the name
2. put this info into the “struct hostent*” which holds info about the machine
3. fill struct sockaddr_in which is the internet socket address stuct for the internet family

```c
{
    sin_family = AF_INET;  /* adress format is host and port number */
    sin_port = ;            /* is the port number */
    sin_addr = ;            /* the IP adress */
}
```
4. call connect() function with the socketID, the filled adress struct and its size
5. if unseccessful throw error
6. if connection is successful send the name and password of the user
7. if server sends the validation, connection is established
   else inform the user and ask for the correct name and password 3 times
   if incorrect name and password entered 3 times close the socket

Sending function

1. while (true) /* infinite loop */
   {
       if (the new action object is created or changed
           || the user typed a chat message)
           clean the buffer
           copy the object to the buffer
           send the buffer with: the socket ID, pointer of buffer,
           the size of the buffer and the flag (0)
   }

Receiving function

1. while (true) /* infinite loop */
   {
       clean the buffer
       receive the message from the server and put it into the buffer
   }
5. ARTIFICIAL INTELLIGENCE DESIGN

5.1. Artificial Intelligence Functionality Scenario

Artificial intelligence is one of the major components of this project, as well as the most difficult one to implement precisely. In this phase of the design we will make use of the classing game playing algorithms such as alpha-beta pruning making use of heuristics functions evaluation.

In the game scenario we mentioned about non-human creatures as a part of the characters’ set. The behavior of all these creatures will be controlled by our artificial intelligence engine. The functionality potential of the creatures varies with respect to their types and their tendency to attacking and/or working. For example a dragon’s primary tendency is toward attacking and causing damage to opposite characters, while a rabbit has greater attitude towards gathering food and escaping from potential damage risks.

The design will be composed of:

- AI Engine
- Alpha-Beta Engine
- Heuristic Evaluation

**AI Engine:**

This is the primary engine of our AI component. The game engine will call the AI engine after each action to give opportunity to AI bots to respond to those actions, (bot = creature throughout this document). The engine will look for affected bots around the last action positions. In case there is any, it will search for best moves of each bots. To accomplish this task the AI engine calls the Alpha-Beta engine giving the bot id as its parameter. Finally it updates the bots status and returns the bots’ moves to the game engine.
Alpha-Beta Engine:

The alpha-beta engine is the “brain” of our AI system. It is provided a bot and it finds the best move for that bot. To accomplish that task it search for opposite players nearby the bot, and finds the best moves to be applied toward those players. Finding the best moves is completed by 2-ply alpha-beta pruning search, making use of heuristic evaluation in each base step. In the end it returns the move providing maximum gain among best moves.

Heuristic Evaluation:

The heuristic evaluation serves to evaluate the current status of the bot issuing an action and the surrounding players. After the evaluation it will return a value ranging from MAX_PROFIT to MIN_PROFIT. During the evaluation process it will make use of some heuristic functions which shapes the bot’s attitudes to searching for best move, for example: force a rabbit to basically gather food.

Proposal (to be implemented in future phases):

Even though it won’t be implemented in the present phase, the game shall provide the capability of machine learning. Our bots should learn from their experiences: successes and fails. The machine learning will be implemented by making use of neural networks. So the game bots will train the neural network for a considerably long time, (otherwise it will not show accurate results), then will search for an estimation of the present world condition through the neural network.

5.2. AI Algorithmic description:

AI Engine:

1. Request position of the last action from the game engine.
2. Get all bots nearby last action.
3. For each bot in 2. do
   {
      3.1) Calculate best move by the Alpha–Beta engine.
      3.2) Apply the best move to the bots’ database.
   }
**Alpha-Beta Engine:**

1. Get all players nearby the bot provided as argument.
2. For each player in 1. do :
   
   2.1. Calculate best move toward the player indexes by the iteration counter, (obviously one of the players in 1), by making use of Heuristic Evaluation.

}  
3. Apply the move having greatest value, (profit), among ones calculated in ‘2.1’.

**Heuristic Evaluations:**

1. Evaluate world status of the bot under consideration by making use of basically these Heuristic functions:

   - Maximum total damage to opposite players/teams (let it be h₁)
   - Escape in case of death risk (let it be h₂)
   - Search for valuable objects and steal them (let it be h₃)

Note: The final heuristic function will be a linear combination of the above functions:

\[ H_f = c_1 \cdot h_1 + c_2 \cdot h_2 + c_3 \cdot h_3 \]

where \( c_i \)-s, \( i=1..3 \), are the coefficients describing the priority of each heuristic.

**5.3. AI’s Data Flow Diagram**

AI engine gathers actions positions from the game engine, and then gets the nearby bots from the world’s database. Afterwards it gives the id of one of the bots to the alpha-beta engine and waits for the best move of this bots. Alpha-beta engine, on the other part, finds the best move by interacting with the Heuristic Evaluator and then checks if the move is legal by communicating with the Physics Engine. Lastly, the Heuristic Evaluator calculates the condition of the bot and its players around by checking from the world’s database the bot’s attributes.
5.4. AI’s State Transition Diagram
The control passes from the game engine to the AI engine after each action requested by the clients. The AI engine will pass the control to the alpha-beta engine for each bot found on a specific (namely circular) area around the last action requested. The alpha-beta engine will pass controls to the Heuristic Evaluator for each base step in the ply-search algorithm.

6. CODING STANDARDS
6.1. Source code organization

6.1.1. Files and project organization

File names could be a mix of upper and lower case characters. The content of the file should be clearly seen by these names.

For the files which contain class definitions or implementations; the name should be same as the class name. Moreover, these files should contain exactly one class, except inner classes and private classes.

In order to avoid name conflicts the name of the headers should contain prefixes. For instance: If two separate module (main window, database) have GUI classes than they should be named as CMainWindowGUI.h and CDataBaseGUI.h.

6.1.2. Header Files

Include Statements

Include statements should not contain the exact path. For example this statement is wrong:

#include "/Utilities/GUI/Include/CMainWindowGUI.h"

but

this statement is true:

#include "CMainWindowGUI.h"

However, in this cases make files should be configured.

Moreover,

#include ".../Include/CMainWindowGUI.h" is a right statement in case the include files gathered in an exact relative location.

Multiple Inclusion of a header file

In order to avoid multiple inclusion.
In windows:

```c
#pragma once
#ifndef C_MAIN_GUI_H
#define C_MAIN_GUI_H
... Rest of Header File ...
#endif //C_MAIN_GUI_H
```

Moreover, only depended headers should be include by header files.

In Unix:

```c
#ifndef C_MAIN_GUI_H
#define C_MAIN_GUI_H
... Rest of Header File ...
#endif //C_MAIN_GUI_H
```

### 6.2. Naming Conventions

#### 6.2.1. Function Names

Function names should tell what they do. These names should be a combination of words which have an uppercase first letter; except the first word. For example:

- `getSize()`, `setFirstMember()`, `print()`.

Function names should have a `verb + object` structure: `printReport()`, `calcLimitOfMove()`.

The description of the return value can be used for function names: `currentColor()`, `nextEmptyCell()`, `screenReady()`.

Use more specific verbs instead of elastic verbs: `processOutput()` is wrong however, `formatAndPrintOutputToScreen()` is right.

The properties of the classes should not be directly accessed instead use functions such that:

- `getColorValue()`, `setColorValue()`. For boolean use “is” instead of get:
  - `getFileReady()` vs `isFileReady()`.

Function arguments should have “a_” prefix: `a_intArraySize;`
6.2.2. Class Names

These names are combination of strings which all words start with capitalized letters. Also, these names should have a capitalized “C” for the first letter: CNetworkClient.

6.2.3. Variable Names

Variable names should tell the content of the variable. These names should be a combination of words which have an uppercase first letter; except the first word.

The variables that are member of classes should have “m_” prefix.

Constant variables should have the #define statement structure: MAX_SIZE, DATE_FILE_NAME.

Pointers should have “p_” intermediate structure: an integer pointer; p_nSizeOfFile;

A table for variables:

<table>
<thead>
<tr>
<th>Type</th>
<th>Member</th>
<th>Local</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>m_nName</td>
<td>nName</td>
<td>a_nName</td>
</tr>
<tr>
<td>char</td>
<td>m_cName</td>
<td>cName</td>
<td>a_cName</td>
</tr>
<tr>
<td>string</td>
<td>m_strName</td>
<td>strName</td>
<td>a_strName</td>
</tr>
<tr>
<td>boolean</td>
<td>m_bName</td>
<td>bName</td>
<td>a_bName</td>
</tr>
<tr>
<td>short</td>
<td>m_sName</td>
<td>sName</td>
<td>a_sName</td>
</tr>
<tr>
<td>long</td>
<td>m_lName</td>
<td>lName</td>
<td>a_lName</td>
</tr>
<tr>
<td>pointer(integer)</td>
<td>m_p_nName</td>
<td>p_nName</td>
<td>a_p_nName</td>
</tr>
<tr>
<td>Structure Circle</td>
<td>m_Circle</td>
<td>Circle</td>
<td>a_Circle</td>
</tr>
</tbody>
</table>

6.3. Source Documentation

Inline comments should have “//” style and they should have the same indent as the code they describe.

6.3.1. Module Comments and Revision history

Module comments should be placed on top of every file and should have the structure:
6.3.2. Commenting Functions

/*
 ** FILE: filename.cpp
 **
 ** DESCRIPTION:
 ** A general description of the module’s role in the
 ** overall software architecture, What services it
 ** provides and how it interacts with other components.
 **
 ** AUTHOR:
 ** Your name here
 **
 ** CREATION DATE:
 ** 14/03/1998
 **
 ** NOTES:
 ** Other relevant information
 **
 ** HISTORY:
 ** Dec 2, 06- M. Taylor - Creation
 ** Dec 2, 06- J. Brander - Insert validation for
 ** m_nArraySize to detect
 ** buffer overflow
 */
/*
**Function Name:getSizeOfFile
**
**Arguments:a_strFileName
**
**Globals: none
**
**Return Value: nFileName
**

**Description: This function gives the size of the argumented file function.
*/

6.4. Programming Conventions

6.4.1. Layout Styles
void checkSomething(int a_nFirstParameter, string a_strSecondParameter)
{
    char cCurrentChar;
    doSomething();
    while (condition)
    {
        doSomething();
        doSomethingElse();
        if (condition)
        {
            doSomething();
        }
        else
        {
            doSomething();
        }
    } 
    switch (condition)
    {
        case CASE_1:
            doSomething();
            break;
        case CASE_2:
            {
                doSomething();
                break;
            }
        default:
            doSomething();
    }
}

6.4.2. Complicated Expressions

Separate conditions are on separate lines.
Wrong:

```java
if (('?0' <= inputChar && inputChar <= '?9') || ('?a' <= inputChar && inputChar <= '?z') || ('?A' <= inputChar && inputChar <= '?Z'))
{
    doSomething(inputChar);
}
```

Right:

```java
if ( ('?0' <= inputChar && inputChar <= '?9') ||
     ('?a' <= inputChar && inputChar <= '?z') ||
     ('?A' <= inputChar && inputChar <= '?Z') )
{
    doSomething(inputCharput);
}
```

6.4.3. Large Function Calls

The method used in large functions calls is to align parameters to the end of the function name each on separate line:

```java
drawLine( Window.north,
    Window.south,
    Window.east,
    Window.west,
    currentWidth,
    currentHeight);
```

6.4.4. Error Handling

In failures objects should pass to error state not an undefined state. For instance: an error of opening a file shouldn't put the object to an locked state instead object should do less things than usual.