4 Quality Control

4.1 Comments

After a class is written and submitted to be included in the library, each function will be sent through an intensive code review.

(1) Every code segment must have a description of its basic function.
If the behaviour is not selfexplanatory, additional information must be included in the comments.
(2) Every function must have a list of all interface tags showing their valid value range.
(3) Whenever a new version of a function is accepted in the library, it must be noted in the history comments.

```
//checks for correct input value on "Cards"
//breaks down card code 0-51 into values 0-12, colors 0-3
//counts values and colors, stores maximum counts
void SCRanking::AnalyseInputData(int Cards[], int CardCountInput)
{
    //IN:  Cards[7]              = 0-51 (local)
    //     CardCountInput        = 5-7
    //     SortGreater           = 0-1  (class global)
    //OUT: CardCount             = 5-7  (class global)
    //     CardsSorted[7]        = 0-51
    //     HandType              = 0
    //     HandPointer           = 0
    //     FlushDrawPointer      = 0
    //     StraightDrawpointer   = 0
    //     CardsValue[7]         = 0-51
    //     CountValues[13]       = 0-4
    //     CardsColors[7]        = 0-3
    //     CountColors[13]       = 0-7
    //     CountValuesMax        = 1-4
    //     CountValuesMaxPos     = 0-12
    //     CountValuesSecPos     = 0-12
    //     CountColorsMax        = 2-7
    //     CountColorsMaxPos     = 0-12
    //     CardsCleaned[7]       = 0-12

    //v003, 2015.09.15 M.Buchty
    //      cleaned and switched to use help class
    //v002, 2014.12.02 M.Buchty
    //      version used to create SCLookUpTables
    //v001, 2014.11.08 M.Buchty
    //      first published version
```

4.2 Interface Check

Interfaces must be monitored during runtime.
This should happen according to the definition in the comments.

(1) Every single input tag must be checked (in each function) so that the actual value is within the valid range.
(2) Critical output tags should be checked for the valid range after they are written (debugging is easier if faults are detected on the spot instead of waiting until they are caught by another functions input check).

This is highly redundant if values are used in multiple functions, but it is a very effective way of detecting errors.

```
// ------------------------------------------------------------------
//check and sort input cards

//copy number of cards in global variable
CardCount = CardCountInput;

if (SCHelp::CheckDoubleCards(Cards,CardCount))
    { SCHelp::ErrorMessage("SCRanking", "AnalyseInputData", "CheckDouble"); }

//copy values to new array as the original array should not be changed (by sorting)
for (int i=0; i < CardCount; i++)
{
    //check if card-array contains valid entries only
    if (Cards[i] < 0 or Cards[i] > 51)
    {
        SCHelp::ErrorMessage("SCRanking", "AnalyseInputData", "Cards");
    }
...
```

4.3 Sub Class Check

Parallel program worlds must be checked if they remain synchronized.

(1) Values that can coexist as local tags in separate classes (e.g. the player AI that are independent programs with only a few interfaces to the dealer) should be double checked either periodically (e.g. stack compare at beginning of each round) or when relevant (e.g. hand value at show down)

```
//player sais he has different money than shown in stack
if (Player_KI->CheckStack(ID, Player_Data[i].Stack) == 0)
{
    Msg.append("Wrong Stack");
    SCHelp::ErrorMessage("SCGameEngine", "Dealer_CheckData", Msg);
}

// ------------------------------------------------------------------

//if player claims winnings, double check his data
Cards[5] = Player_Data[Pos].Cards[0];
Cards[6] = Player_Data[Pos].Cards[1];
Hand.GetHandValue(Cards, 7);

if (FinalistsHand[Pos] != Hand.HandValue)
{
    //player is cheating (program error)
    std::string Msg = "ID " + std::to_string(ID) + " - Player Hand mismatch";
    SCHelp::ErrorMessage("SCGameEngine", "Dealer_Showdown", Msg);

    Player_Data[Pos].IsActive    = false;
    Player_Data[Pos].LastAction  = "Cheater";
    Player_Data[Pos].CardsShow   = 0.25;
    Player_Data[Pos].LastActionX = false;
    continue;
}
```

4.4 Test Cases

The most time consuming part in the code review process is creating test cases for every single behaviour of each function, setting up the input tags of the function to predefined values and generating error messages if the resulting output is different than the expected value.

```
Player.Hand->HandValue = 2000000;
R = Player.ShowCards(1000000);
if (R != 2000000) {SCHelp::ErrorMessage("SCPlay_HumanTest", "ShowCards", "Show Higher");}
R = Player.ShowCards(2000000);
if (R != 2000000) {SCHelp::ErrorMessage("SCPlay_HumanTest", "ShowCards", "Show Equal");}
R = Player.ShowCards(3000000);
if (R != -1)      {SCHelp::ErrorMessage("SCPlay_HumanTest", "ShowCards", "Fold");}

S = I.Convert_LongToString(100000);
if (S != "1.00k") {SCHelp::ErrorMessage("SCInterfaceTest", "Convert", "100k");}
S = I.Convert_LongToString(110000);
if (S != "1.10k") {SCHelp::ErrorMessage("SCInterfaceTest", "Convert", "110k");}
S = I.Convert_LongToString(111000);
if (S != "1.11k") {SCHelp::ErrorMessage("SCInterfaceTest", "Convert", "111k");}
S = I.Convert_LongToString(111100);
if (S != "1.11k") {SCHelp::ErrorMessage("SCInterfaceTest", "Convert", "111k.1");}
S = I.Convert_LongToString(999999);
if (S != "9.99k") {SCHelp::ErrorMessage("SCInterfaceTest", "Convert", "999k.9");}
```

The source code for test cases can easily be 10x bigger than the code oft he function that is tested.

If a function passes the testcases without error messages (neither from test case results nor internal flags), it is passed to the library of the poker app for runtime tests.

4.5 Runtime Tests

Every function has to pass two stages after being activated in the library:

(1) Automatic Runs
These runs trigger the functions to test an excessively amount of times to see if internal flags appear.

Example 1: Every possible combination of 5, 6 and 7 cards is passed to the hand ranking function.
Example 2: The game engine plays a few million rounds against itself.

(2) Human Player Trials
These runs aim to check for strange behaviour that can not be checked by automated programs (concept errors).

Example 1: Wrong play of an AI like going all-in when having a Four-of-a-Kind disregarding that it is lying in the common cards (and the own kicker is weak).
Example 2: Continuing to raise, even after all other players are all-in.