

---

# DELPHI

# PROGRAMMIERPRAKTIKUM

# WS 24/25

---

Piraten Karpern Programmierhandbuch

16. MÄRZ 2025  
MORTEN SCHOBERT  
ITAS 106089

## Inhalt

Entwicklungskonfiguration .....	4
Betriebssystem .....	4
Compiler.....	4
Problemanalyse.....	5
Das Computer game “Piraten kapern” .....	5
Spielmechanik.....	5
Spielende .....	5
Weitere Anforderungen.....	5
Grundfunktionen.....	5
Benutzeroberfläche.....	5
Spielstand-Management .....	6
Ereignislog.....	6
Realisationsanalyse .....	7
Herausforderungen .....	7
Realisierungskonzept .....	8
grundsätzliche Datentypen.....	8
naheliegende Entitäten.....	8
Struktur des Programms .....	9
Kernkomponenten .....	9
Action-Refresh Konzept.....	9
Punkteberechnung.....	10
Laden/Speichern .....	11
Ereignislog.....	11
Implementierung .....	12
Objektorienterter Ansatz .....	12
Testbarkeit.....	12
Grundsätzliches.....	12
Spielsituationen erstellen .....	12

# Piraten Karpern Programmierhandbuch

Cheat-Modus.....	13
Ersatz für Dynamische Datenstrukturen.....	13
Actions .....	14
fmGame.Refresh.....	14
Animationen .....	15
Bilder.....	15
Memory-Management.....	15
Memory-Management Schwierigkeiten.....	16
In.CreateInstance und DeclInstance .....	18
Logging.....	18
Datenstrukturen .....	19
Unit Base.....	19
Unit uLNgame.....	19
Unit uLNgame.Cubes.....	20
Unit uLNgame.Player .....	20
Unit uLNgame.internal .....	20
Unit uLNgame.IO .....	20
Unit uLNgame.Cards.....	20
Unit uAction .....	21
Unit uLogging .....	21
Unit uFmGame.....	21
Unit uFmGame.utils .....	22
Unit uImageUtils.....	22
Unit uFmSelectPlayers .....	22
Unit uFmCheatThrow .....	22
Programmorganisationenplan: .....	22
Test-Strategy .....	24
Das Thema Testbarkeit. ....	24
Unit-Tests .....	24

## Piraten Karpern Programmierhandbuch

GUI-Tests.....	25
----------------	----

## Entwicklungskonfiguration

### Betriebssystem

Windows 11, version 23H2

### Compiler

Embarcadero® Delphi 12 Version 29.0.52631.8427

## Problemanalyse

### Das Computer game “Piraten kapern”

Es wird noch einmal die Aufgabe beschrieben, die umgesetzt werden muss.

Das game basiert auf den Regeln des Originals "Piraten kapern". Wichtig für das Verständnis ist das Benutzerhandbuch. Es wird vorausgesetzt, dass dieses gelesen wurde.

Anmerkung: Es wird hier bewusst von game gesprochen, wenn das Spiel gemeint ist. Das ist kein Rechtschreibfehler 😊

### Spielmechanik

Die Spieler würfeln mit acht Würfeln und können nach jedem Wurf entscheiden, welche Würfel sie behalten und welche erneut geworfen werden.

- Würfel mit Totenkopf sind bis zum Ende des Zugs gesperrt.
- Drei Totenköpfe beenden die Runde mit 0 Punkten.
- Punkte ergeben sich aus den Würfelseiten, deren Kombinationen und der aktuellen Karte.
- Spezielle Karten beeinflussen das Spielgeschehen und die Punkteberechnung
- Eine "Totenkopfinsel"-Mechanik bestrafst die Mitspieler mit Punktabzug.

### Spielende

- Das Spiel endet, wenn ein Spieler 6000 Punkte erreicht.
- Eine "Piratenmagie"-Kombination (9 gleiche Symbole) führt zum sofortigen Gewinn.

## Weitere Anforderungen

### Grundfunktionen

- 2 bis 4 Spieler mit individuellen Namen
- Neustart eines Spiels soll jederzeit möglich sein
- Zufällige Mischung des Kartenstapels (35 Karten mit vorgegebener Verteilung der game-Karten)

### Benutzeroberfläche

- Darstellung der aktuellen Punkte und des aktiven Spielers

## Piraten Kartern Programmierhandbuch

- Anzeige des Würfeltischs mit zufälliger Rotation der Würfel
- Hervorhebung von Totenkopf-Würfeln: Totenköpfe mit roten Augen entweder wenn Spielzug verloren wurde oder im Totenkopfinsel Modus.
- Darstellung des aktuellen Spielgeschehens. D.h. Zugewinn, Verlust, Totenkopfinsel, Piratenschiff-Modus, u.a. durch Hintergrundbilder des Würfeltisches (Säbel im Piratenschiffmodus / Totenkopf im Totenkopfinselmodus)
- Bereich für "geschützte" Würfel in Rasterdarstellung
- Interaktive Würfelbewegung (Linksklick zum Sichern/Rückführen, Rechtsklick für Schatzinsel)
- Visuelle Darstellung besonderer Spielsituationen (z.B. Totenkopfinsel/Piratenschiff Modus)
- Bedienung über Buttons (Würfeln, Werten, Neues Spiel, Laden, Speichern, Log ein-/ausblenden, Beenden)
- Bestätigungsdialoge vor spielstand-kritischen Aktionen

## Spielstand-Management

- Speichern und Laden von Spielständen als Textdatei (.pk)
- Speicherung nur möglich, wenn ein Spieler neu am Zug ist
- Sicherstellung der Datenintegrität beim Laden
- Struktur der Spielstand-Datei:
  - Zeile 1: Anzahl der Spieler
  - Zeile 2: Aktueller Spieler
  - Zeilen 3+: Spielername und Punktzahl
  - Reihenfolge des Kartenstapels

## Ereignislog

- Vollständiges Spielprotokoll zur Nachverfolgung des Verlaufs
- Log wird bei neuem Spiel gelöscht
- Anzeige in der Spieloberfläche ein- und ausblendbar

## Realisationsanalyse

Im Folgenden wird immer von Methode gesprochen, wenn entweder Prozedur oder Funktion gemeint ist.

### Herausforderungen

Es gibt folgende Bereiche, die eine deutliche Komplexität aufweisen

Die Punkteberechnung ist abhängig vom Spielgeschehen, d.h. der Karten und der einzelnen Würfe, sowie der gesicherten Würfel. Folgende Situation beeinflussen die konkret:

- alle Karten außer die Wächterin zählen.
- Totenkopfinsel: Nur Totenköpfe zählen, und zwar nur bei den anderen Mitspielern,
- Bei Piratenschiffkarte kann man auch Punkte abgezogen bekommen.
- Wenn alle 8 Würfel punkten, dann gibt das einen Bonus

Der Verlauf des Spielzugs

- Wenn beim ersten Mal 4 Totenköpfe gewürfelt werden, oder die Piratenschiffkarten auf dem Tisch liegen, dann kann nicht freiwillig beendet bzw. erst wenn die Bedingung Anzahl Säbel erfüllt ist
- Piratenmagie: Es liegen 9 Diamanten oder Goldmünzen (incl. Karte) auf dem Tisch, dann ist das Spiel sofort gewonnen

Optische Darstellung des Spielgeschehens

- Rotation von Würfeln, die Würfel sollen nicht überlappend dargestellt werden.
- Je nach Spielgeschehen, soll optisch die Situation hervorgehoben werden. Im Zugmode PiratenSchiff erscheint als Hintergrund des Würfeltisches ein grosser Säbel, Im Zugmode Totenkopfinsel werden alle nicht Totenköpfe grau dargestellt, Totenköpfe haben „rote“ Augen und wenn der Zug beendet ist, verlieren sie die roten Augen. Als Hintergrundbild erscheint ein Totenkopf Wird der Zug verloren wegen zu vieler Totenköpfe, werden diese mit roten Augen angezeigt

Extra Anforderung:

## Piraten Kartern Programmierhandbuch

- Im Original-Spiel sind die Würfel, die man berechnet haben möchte zur Seite gelegt (gesichert) und im Falle der Schatzinselkarte werden die Würfel auf der Karte „Sicher“ gesichert und nur die restlichen Würfel bis auf Totenköpfe können gewürfelt werden. Im game aber soll es im Fall der Schatzinselkarte möglich sein, die Würfel „normal“ sichern können und auch „Sicher“ sicher „auf der der Schatzinsel“
- Es soll ein MemoryManagement implementiert werden, dass eventuelle Speierlecks aufdeckt. Das soll auch für die Benutzung von Klassen gelten.

## Realisierungskonzept

### grundsätzliche Datentypen

Anhand der Spielregeln erkennt man folgende Datentypen/Strukturen und Entitäten.

- Aktionen (Würfeln, Würfel sichern, Würfel befreien, Spielzug beenden, Totenkopf würfeln, Spiel Starten, Spiel laden, Spiel Sichern) geben muss.
- ZugModus (normal, Piratenkarte, Totenkopfinsel)
- ZugStatus (ZugBeendet, im Zug)
- GameStatus (game läuft, game beendet, game wartend)
- GewinnArt (normal, PiratenMagie)
- Würfel-Status (Frei,gesichert,SicherGesichert)

naheliegende Entitäten.

### *Das inGame*

Status des game (Aktiv, gewonnen, pausierend)  
GewinnArt (Normal, GewinnMagie)  
Aktuelle Spielkarte  
Kartenstapel  
Aktueller Spieler  
Spieler,Punkte  
ZugStatus (aktiv, beendet, beendigungNotwendig)  
Zugmodus (TreasureIsland, Normal, oder Saber)  
SpielWürfel

### *Die Spieler*

Die aktiven Spieler (2..4 eindeutige Spielernamen)  
Deren Punktestand

## Piraten Karpern Programmierhandbuch

### *Die SpielWürfel*

Würfelseiten (0..7)

WürfelStatus (0..7)

### *Die uiGame*

Darstellung der Karte, des Würfeltischs, des gesicherten Bereichs

Darstellungen der aktuellen Karte, der Würfel und deren Platzierung

Darstellung der Spieler und deren Punkte

Aktionsschnittstellen (Buttons)

## Struktur des Programms

Kern der Struktur sollte ein Komponentenmodell aufbauend auf diesen Entitäten sein. So ein Modell hat folgende Vorteile:

- Diese Komponenten sollen möglichst eigenständig funktionieren
- Implementierungsdetails werden gekapselt.
- Testbarkeit von unabhängigem Code ist deutlich einfacher-
- nur ausgewählte Funktionen der Komponenten werden für die Kommunikation untereinander verwendet.

### Kernkomponenten

Kern des Programms ist das inGame und die uiGame ,

Das inGame verwaltet den Status und die Modi und führt die Berechnungen aus, es bietet die notwendigen Methoden um die Aktionen des game auszuführen.

Die uiGame ist die Oberfläche auf der das game stattfindet. Es spiegelt das inGame wieder und initiiert die Aktionen des Spiels in inGame

### Action-Refresh Konzept

Kern der Kommunikation in diesem Programm ist die Kommunikation zwischen uiGame and inGame. Diese Kommunikation muss alle Spielaktionen beinhalten und auf alle Änderungen in inGame reagieren.

Die Idee ist, dass uiGame „blöd“ ist, und nicht viel über die Details von inGame wissen soll.

Also sollen alle Aktionen die in uiGame initiiert werden, durch Konstanten (TActions) und nur eine einheitliche DoAction Methode verwendet werden. Zu jeder Aktion gibt es eine entsprechende Methode von inGame.

Die Aktualisierung des Spieloberfläche geschieht über eine Refresh Methode, die wird direct nach doAction ausgeführt. So kann uiGame sicherstellen, dass der Status von inGamme in der GUI dargestellt wird.

Beispiel-Code, der den Ansatz erklären soll:

```
procedure TfrmGame.doAction(action: TAction; id:  
TCubeld);  
var  
    ids: TCubeldSet;  
begin  
    ids := [id];  
    uAction.doAction(ingame, mmoLog, action, ids);  
    refresh(action, ids);  
end;
```

Diese Aktionen und die Statii des inGame bestimmen dann die Kommunikation zwischen der UIGame und der dahinter liegenden inGame-Komponente. Dadurch bleibt alles recht übersichtlich, lesbar und nachvollziehbar. Die Steuerung des Geschehens sowie der Darstellung wird auf wenige Bedingungen reduziert.

## Punkteberechnung

Aufgrund der Komplexität der Punkteberechnung soll diese besondere Beachtung erhalten und so implementiert werden, dass sie möglichst unabhängig und gut getestet werden kann

Um Klarheit zu schaffen, sollen für die

- Kombination-Punkte Beziehung (COMBINATATION\_POINTS),
- für die Seite-Punkt (SIDE\_POINTS) Beziehung,
- Karte-Würfel (ADD\_CUBES\_CARDS) Beziehung

entsprechende Datenstrukturen angelegt werden

Die Punkteberechnungsformeln sollen auf Kombinationen dieser Arrays basieren.

Diese Formeln verwenden ansonsten nur die Würfelinformatioinen und den Zugstatus. Weitere Informationen des game sind nicht nötig.

Die Punkteberechnung soll aufgrund ihrer Komplexität in eine eigene Unit ausgelagert werden.

## Laden/Speichern

Weiter empfiehlt es sich, auf die Funktionen des Laden und speichern auszulagern. Eine geeignete Datenstruktur sollte gewählt werden, so dass die game-Logik möglichst unabhängig von speziellen Dateiformaten etc... bleibt

## Ereignislog

Das EreignisLog soll möglichst entkoppelt von der Benutzeroberfläche funktionieren. Jede Aktion die ausgeführt wird soll in das Log schreiben.

## Implementierung

### Objektorienterter Ansatz

Da das Programm mit Delphi entwickelt wird, bietet sich ein objektorienter Ansatz an. Er ist ein naheliegender Ansatz für die Umsetzung einer auf Komponenten basierender Architektur ist. Vererbung ist nicht nötig.

Anmerkung: Nach Rücksprache wurde dieser Ansatz erlaubt,

Diese Objektorientierung ermöglicht auf eine einfache Art eine saubere Kapselung (private vs public) von Datenstrukturen, welche sich anhand der Entitäten / Komponenten ergeben. Auch sind die Methoden für die Kommunikation zwischen diesen Komponenten hier einfach und verständlich.

Anmerkung: Es würde auch klassisch möglich sein. Aber eine saubere Kapselung ist aufwendiger zu implementieren. Und es wäre auch nicht der durch Delphi vorgegebene übliche Weg. Aber um zu demonstrieren, wie eine klassische Implementierung aussehen würde, wurde speziell die Komponente TPlayers auf diesem Weg umgesetzt. Siehe hierzu die unit uIngame.players.

Es existiert auch noch die ursprüngliche TPlayers Klasse in der unit. **uArchiv.players**.

### Testbarkeit

#### Grundsätzliches

Es sollte möglich sein, Methoden und Ihre Ergebnisse über spezielle Testprogramme (unit-tests) zu testen. Aber nicht alles lässt sich so testen, bzw. nur sehr schwer. UI Tests sollen weiterhin manuell durchgeführt werden.

#### Spielsituationen erstellen

Um Testbarkeit zu ermöglichen, muss es möglich sein, spezielle Spielsituationen zu erstellen. Da man das nicht dem Zufall überlassen kann, muss die „Zufälligkeit“ des Spiels umgangen werden. Es muss dafür möglich sein, Karten und Würfelergebnisse dem InGame vorzugeben.

Die Karten vorzugeben ist einfach, da über die Funktion Laden eines Games „frisierte“ Kartenstapel vorgegeben werden können.

Aber für das Würfeln muss dem inGame mitgeteilt werden, dass sie vorgebene Würfelergebnisse benutzen soll. (Siehe Details hierzu unter Cheat-Modus)

## Cheat-Modus

Für die GUI Tests, welche manuell durchgeführt werden, muss ein Cheat-Modus (Tastenkombination Alt+C) implementiert werden. Dieser ruft beim Würfeln ein frmCheatThrow-Formular auf, indem die nötigen Würfelergebnisse eingestellt werden können.

## Ersatz für Dynamische Datenstrukturen

Es gibt häufig die Notwendigkeit im Code Mengen von Würfeln oder Würfelseiten zuzugreifen (TCubelds, TCubeSides, TCubeStates), oder diese als Filterkriterien zu verwenden. Das ist sehr gut möglich über die MengenTypen von Delphi. Anderseits gibt es Situationen, in den Arraystrukturen auf Basis dieser Mengen als Übergabe-Parameter benutzt werden sollen. Das ermöglicht eine bessere Entkopplung

Da keine Delphi-spezifischen dynamischen Arrays verwendet werden durften, stellte sich die Frage, ob man diese über dynamisch allokiertes Memory umsetzen sollte. Das würde die Sache sehr verkomplizieren. Und diese Strukturen wurden nur für die Übergabe zwischen den einzelnen Funktionen verwendet.

Also machte ein einfacher Ansatz Sinn. Man definiert in diesem Fall einen Record-Typ, der das Array und die Menge verbindet. Nur die Array-Einträge sind gültig bzw. änderbar, welche auch in der Menge vorhanden sind. Alle anderen Einträge sind bedeutungslos bzw. nur zur Information.

Als Beispiel soll die Struktur TPlacements dienen.

```
TPlacements = record
    cubelds: TCubeldSet;
    rects: array [TCubeld] of TRect;
end;
```

Diese wird für die Berechnung neuer Positionen von gewürfelten Würfeln verwendet. Obwohl alle 8 Array-Elemente vorhanden sind, sind nur die relevant und korrekt, welche auch über die zu wüfelenden Cubelds definiert sind.

Weitere Strukturen:

```
TCubeldSides = record
    ids: TCubelds;
    sides: TCubeSideArray;
end;
```

## Piraten Kartern Programmierhandbuch

Diese wird vor allem für die Übergabe von Würfeln im Cheat-mode mit frmCheatThrwo verwendet. Hier werden alle sides mit den bestehenden Würfelseiten gesetzt, aber die Ids bestimmen, welche durch den Cheat änderbar sind.

```
TPlayersRec = record  
  allStats: TAllStats;  
  numberOfActivePlayers: TActivePlayers;  
  currentPlayerId: TPlayerId  
end;
```

Dies ist die Struktur, welche intern von Players verwendet wird. TAllStats ist ein statisches Array für die maximale Anzahl Player. Aber durch numberOfActivePlayer ist sichergestellt, dass nur auf die relevanten Einträge zugegriffen wird.

## Actions

Das game hat Aktionen, die vom Spieler ausgeführt werden dürfen:

```
TAction = (gaSaveCube, gaSaveSavedCube, gaFreeCube, gaFreeSkull, gaThrow,  
gaEndMove, gaNextMove, gaWaitForGame, gaSaveGame, gaStartGame, gaLoadGame);
```

Die UI (TFrmGame) soll möglichst wenig wissen müssen, um die Aktionen des Spiels ausführen zu können.

Deshalb wird immer dort, wo eine Spielaktion nötig ist „einfach eine Funktion „doAction(action TAction) aufgerufen.

Bsp: Wenn btnThrow angeklickt wird, dann wird die Methode unten ausgeführt.

```
procedure TfrmGame.btnThrowClick(Sender: TObject);  
begin  
  doAction(gaThrow);  
end;
```

Dadurch wird die UI sehr übersichtlich. DoAction führt dann die notwendigen Details dieser Aktion durch und ruft am Ende ein UI.refresh auf.

## fmGame.Refresh

Refresh ist die Methode, die den Spielstand des game auf der UI darstellt. Es soll vermieden werden, dass verschiedene Eventhandler/doActions nur Teile des Refresh

## Piraten Karpern Programmierhandbuch

übernehmen und damit dieses Refresh verteilt in mehreren Code-Abschnitten implementiert ist. Das erhöht die Verständlichkeit des Codes.

## Animationen

Auch wenn keine wirklichen Animationen im uiGame vorhanden sind, so werden dennoch gewisse Aktionen zeitverzögert dargestellt. Dadurch wirken Aktionen auf dem Bildschirm natürlicher und der player hat die Zeit die Situation zu verstehen. Das ist so üblich im „Gaming“-Umfeld. Das soll auch in diesem game zum Einsatz kommen. Da wäre

- das Anzeigen des Beschreibungstextes der Karte, welches dadurch wie ein Umdrehen der Karte wirkt
- das Ende des Zugs, welcher kurz innehält, dann die Punkte aktualisiert, bzw. die Totenköpfe mit roten Augen anzeigt und dann nochmal innehält ,um den nächsten Zug einzuleiten

## Bilder

Bilder der Würfel, Würfelseiten, und Karten werden als einfache TImage Komponenten in der Form gespeichert. Da das Tag-eines images nicht verwendet werden darf, wurde mehrere internes Array of TImage ein direkter Zugriff auf die relevanten images realisiert.

Anmerkung. JPG Bilder lassen sich nicht transparent darstellen. Deshalb wurden diese auf PNG umgewandelt.

**WICHTIG: Die Rotation der Bilder wird über eine externe Unit ultimageutils umgesetzt. Diese verwendet eine moderne Windows Schnittstelle GDI zum Rotieren von Images. Das erzeugt deutlich bessere Bilder.**

## Memory-Management.

Es soll jedesmal wenn dynamisch Speicher erzeugt wird, diese +1 gezählt werden, und wenn er wieder freigegeben wird, -1 gezählt werden. Hinzu kommt, dass auch das erzeugen von Objekten entsprechend gemanagt werden sollen.

Die vorgegebene Idee, in Units einen globalen Zähler in der Implementierung zu haben, welcher bei Finalization = 0 sein soll, ist unglücklich, da

- Man immer wo Memory allociert bzw. deallocated ein inc bzw dec als 2. Befehl ergänzen muss. Das ist selbst fehleranfällig.
- Wenn man in einer Unit Memory-allociert und in einer anderen Unit diesen deallocated, dann kommen die Unit-spezifischen Zähler durcheinander. Also bliebe am Ende doch nur ein einziger Zähler übrig.

Also entstand die Idee, eine zentrale -CreateMem/ReleaseInstance Functionen zu entwickeln. Die Standardfunktionen sollten nicht mehr verwendet werden.

Der Ansatz ist recht einfach für Memory. Aber für Objekte war gar nicht klar, ob das gehen könnte. Und ob es nicht sinnvoller wäre, hier einfach einen Zähler bei create und destroy von Objekten einzubauen.

Anmerkung an dieser Stelle: Assert in unit-Finilazation zu verwenden funktionierte nicht. Anstelle desssen wurde Windows.showMessage verwendet

Das hat sich in der Realisierung doch als komplizierter als erwartet rausgestellt. Mehr davon im nächsten Kapitel.

## Memory-Management Schwierigkeiten

**Wichtig:** Es wurde jetzt Hilfe von Außen hinzugezogen, da das dann doch die eigene Kompetenz überschritt. Andererseits sollte auch nicht in der Mitte aufgegeben werden.

Es stellt sich bei Objekten die Frage, ob man diese Zähllogik in die Objekte mitaufnimmt. Also in den Constructor und Destructor integriert, oder eben, ob man von aussen ein ObjectCreate und InstanceRelease ausführt. So wie auch die Idee für CreateMemory and releaseMemory.

Solange die Objecte selbst erzeugt sind, bietet es sich an, dass die Objecte selbst ihr Memory-Management verwalten. Also increment im Constructor und decrement im Destructor. Man müsste sich dann darum nicht mehr kümmern.

Das ist zwar nicht schön, denn man würde das Memory Management und die Objekte miteinander koppeln, aber es wäre einfacher umzusetzen.

Aber es gibt im Programm auch noch andere Objekte welche erzeugt werden. Das sind Delphi Objekte. Hier sind das im Wesentlichen Objekte, welche von TComponent abstammen. Es sind Dialoge und Formulare.

Also wieder die Überlegung, so ein CreateObject/ReleaseInstance zu bauen. Immerhin ist Delphi eine moderne Sprache und bietet sehr viele Möglichkeiten. Es stellt sich heraus, dass man nicht nur Objekte als Parameter übergeben kann, sondern auch Klassen. Diese kann man wie genauso im Code benutzen. Das sieht dann wie folgt aus:

```
function createObject(classID: TClass): TObject;
var    obj: TObject;
begin
  obj := classID.create;
  incInstance(obj);
```

```
createObject := obj;  
end;  
  
var obj : TIngame;  
begin  
  obj := createObject(TIngame) as TIngame;  
end.
```

Aber die Tücke liegt im Detail.

Probleme bereitet die Tatsache, dass TForm oder TDialog, welches im Programm benutzte Komponenten sind, alle von TComponent abstammen, d.h. sie haben nur einen create(owner) und nicht einen Standard create Konstrutor, d.h. ein create ohne Parameter ist dort nicht möglich.

Leider ließ sich das nur lösen, indem man eine createComponent Function einführt, die für TComponentClass gedacht ist. Also alle Klassen, welche von TComponent abstammen.

```
function      createComponent(classID: TComponentClass): TComponent;  
var obj: TComponent;  
begin  
  obj := classID.create(nil);  
  incInstance(obj);  
  createComponent := obj;  
end;  
  
var obj : TIngame;  
begin  
  obj := createObject(TIngame) as TIngame;  
end.
```

Aber das war noch nicht alles.

Leider ist das Klassenmodell von Delphi nicht konsistent und TObject.create, die Mutter aller Klassen hat einen Constructor der nicht virtuell ist, und damit können alle anderen Klassen dieses create nicht mit override überschreiben, d.h., der code obj := classID.create funktioniert nicht korrekt und ruft nur den TObject.create auf.

Leider initialisieren sich die Klassen TInGame, TGameCubes des Programms im constructor create.

Es gäbe die Möglichkeit die eigenen Klassen von einer neuen BasisKlasse TVObject mit einem virtuellen create abzuleiten. Aber weitere Vererbung sollte nicht verwendet werden.

Nach Begutachtung von TObject, (Anm: man kann ja den Code von Delphi selbst begutachten), kam als möglicher Kandidat die Methode AfterConstruction infrage. Diese ist virtuell und kann damit überschrieben werden. Also wurden die Klassen im Programm entsprechend angepasst und der Code ist nun lauffähig.

### InInstance und DeInInstance

Sind die Funktionen, die den internen Zähler einen hoch bzw. runterzählen. Diese haben als Übergabe-Parameter die ClassId, denn sie sollen für die benutzen Klassen, jeweils eigene Zähler einsetzen. Dabei wird intern ein Array verwendet, welche für jede classId ein eigenes Element besitzt. Auf diese Weise kann genau identifiziert werden, welche Klassen von den Memory-Leaks betroffen sind.

### Logging

Die Entkopplung von der UI wird dadurch implementiert, dass man eine TMemo Komponente als Schreibziel des Logging nutzt. D.h. das Logging erhält als Parameter eine TMemo Variable.

Zu jede einzelnen Aktion wird eine eigene Logging Methode erstellt.

Die Aktionensverarbeitung in uAction stellt sicher, dass nach jeder Aktion diese Logging Methode aufgerufen wird.

## Datenstrukturen

Hier soll ein Überblick über die Realisierung und deren Datenstrukturen gegeben werden. Es geht nicht um Vollständigkeit, da der aktuelle Code im Detail doch aussagekräftiger ist als eine umständliche textuelle Umschreibung. Und zum anderen ist der Code nochmal dokumentiert.

### Unit Base.

Hier wurden alle notwendigen game übergreifenden Informationen über Typen und Konstanten definiert.

Diese Unit muss in allen anderen Units hinterlegt werden. Hervorzuheben sein hier

TCubeSide	Die Würfelseiten
TCubeld	Die ids der einzelnen Würfel
TCubestate	Würfel befinden sich immer in einem Status. ctFree: Liegen auf dem Würfeltisch und sind keine Totenköpfe ctSaveSaved: Wird für Schatzinselmode verwendet ctSaved: Speichert diese Würfel ctKilled: sind immer Totenköpfe.
TGameCard	Die Spielkarten
TPlayerId	Die Spieler Id
TMoveMode;	Steuert das Spielgeschehen: Wurde schon erklärt (mmNormal, mmSkullIsland, mmPirateShip)

### Unit ulngame

Hier wird die eigentliche Spiellogik implementiert. Kern ist die Klasse TInGame

TActions	Der Kern der Kommunikation zwischen InGame und der UI. gaSaveCube, gaSaveSavedCube, gaFreeCube, gaFreeSkull, gaThrow, gaEndMove, gaNextMove, gaWaitForGame, gaSaveGame, gaStartGame, gaLoadGame);
TMoveStatus	msInMove: Man befindet sich im Spielzug msEndMoveRequired. Der Spielzug muss beendet werden msNotInMove: Der Spielzug ist abgeschlossen
TIngame	Die Klasse, die die Spiellogik implementiert. Diese Funktionen sind hervorzuheben: GetPossibleActions: steuert, welche Aktionen möglich sind. CalculatePoints: lässt die Punkte berechnet Throw: Die das Würfeln durchführt

## ulngame nutzt

- uBase,
- ulngame.io für das laden und speichern des spiels
- ulngame.internal für die Berechnung der Punkte
- ulngame.Players für die Players.
- ulngame.Cubes für die Cubes

## Unit ulngame.Cubes

Diese Unit ist für das Managen der Würfel zuständig

TGameCubes class    Diese Klasse managed die Cubes, das Würfeln und auch den Status der einzelnen Cubes

Sie nutzt uBase.

## Unit ulngame.Player

Diese Unit stellt die Player-Logik bereit. Diese ist nicht in Form einer Klasse implementiert. Nach außen gibt es keine Datenstruktur. Diese ist komplett gekapselt.

Sie nutzt uBase,

## Unit ulngame.internal

Ist für die Berechnung der Punkte zuständig.

Sie nutzt uBase

## Unit ulngame.IO

Ist für das Laden und Speichern des Spiels zuständig.

TGameStructure    Diese Struktur ist wesentlich für den Austausch von InGame und den IO Funktionen

TIngameIOError =    Relevant für das Fehlermanagement beim Laden und Sichern von den Dateien

Sie nutzt uBase.

## Unit ulngame.Cards

Diese unit stellt einen gemischten Cardstack zur Verfügung.

TCardStack = array [TCardId] of TGameCard;    Das ist der Cardstack, der von Ingame verwendet wird

Sie nutzt uBase,

## Unit uAction

Diese Unit hat als einzige Aufgabe die Aktionen die die game UI d.h. frmGame ausführt , auszulagern und zu verarbeiten und relevante Details zu implementieren. Sie kümmert sich auch um das Logging.

Es werden genutzt

- uBase
- uFmSelectPlayers zum Erfassen der Spieler beim Start eines neuen Games
- uFmCheatThrow zum Cheaten des Throw
- uLogging
- uInGame

## Unit uLogging

Diese Unit beinhaltet die Logging-Methoden, die jeder Action entsprechen.

Es wird uBase benutzt.

## Unit ufmGame

Ist die Unit für die game GUI. Hier wird das Spiel dargestellt und dem Benutzer die Möglichkeit gegeben Aktionen auszuführen.

TFrmGame      Die Hauptfunktionen sind  
                  Do"Action" welches die Aktionen ausführen, die durch  
                  Ingame.possibleAction möglich sind  
                  Refresh stellt sicher, dass die UI immer nach einer Aktion das richtige  
                  darstellt

DoLog speichert die ausgeführte Aktion, oder eventuelle Fehler, die aufgetreten sind

Sie nutzt

- uBase
- UIngame als Schnittstelle für die GameLogik
- uAction, welche die tatsächliche Action bearbeitet und mit ingame kommuniziert
- uFmGame.utils lagert gewisse Utility-Funktionen aus
- UIImageUtils ist eine Fremdunit, die es ermöglicht Bilder zu rotieren

## Unit uFmGame.utils

Diese Unit lagert Funktionen für das platzieren und anzeigen der Würfel aus.

## Unit ulmageUtils

Diese Unit stelle eine image-Rotate-Funktion (RotateBitmapInPlace) zur Verfügung

## Unit ufmSelectPlayers

Die Unit ist für die GUI des Auswählens der Spieler da.

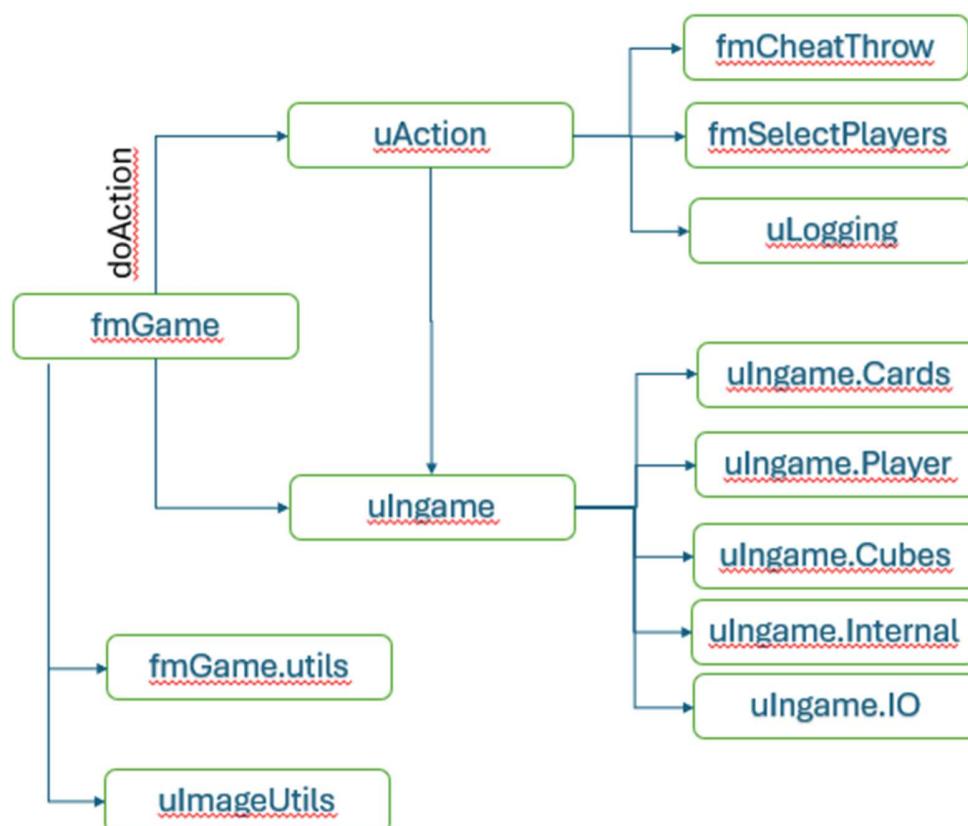
Sie nutzt uBase

## Unit ufmCheatThrow

Die Unit ist für die GUI des Cheat-Throw da.

Sie nutzt uBase

## Programmorganisationsplan:



## Piraten Karpern Programmierhandbuch

Hier wird nochmal graphisch dargestellt, wie die units sich gegenseitig aufrufen bzw. verwenden. Diese Beziehung wurde ja schon unter den Datenstrukturen beschrieben. uBase wird hier nicht dargestellt, da sie Basis jeder Unit ist

## Test-Strategy

### Das Thema Testbarkeit.

Umfangreiche Tests sind notwendig, um die Komplexität des Spiels zu überprüfen.

Programmiercode ist nur dann gut testbar, wenn die einzelnen Komponenten möglichst getrennt voneinander funktionieren und nicht voneinander abhängig sind.

Unit-Tests sind der übliche Weg einzelne Funktionen des Programms zu testen. Delphi bietet mit DUnitX ein Testframework dafür.

Die GUI, dessen Test unbedingt erforderlich ist, soll aber „manuell“ getestet werden. Eine Unit-Test Umsetzung für die GUI erschien zu schwierig. Auch wenn es prinzipiell möglich erschien.

### Unit-Tests

Es gibt im Quellcode-Ordner ein Projekt unitTests.dpr.

Es sollen Unit-Tests für die Hauptfunktionen des Games umgesetzt werden.

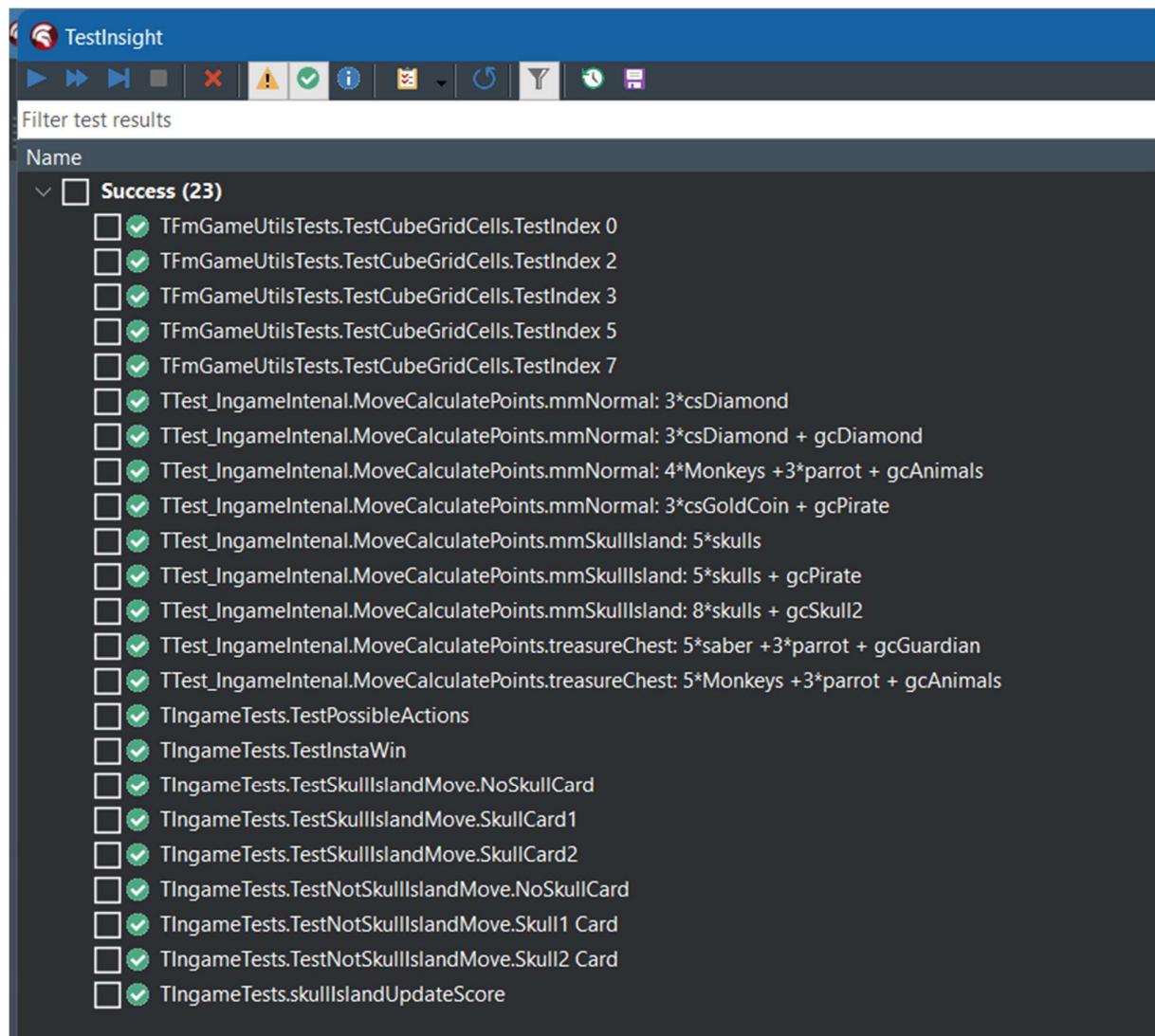
Für `ingame.getPossibleActions`, da diese das Spielgeschehen steuert.

Für die Punkteberechnung, da es unzählige Möglichkeiten der Punkteberechnung gibt und diese sonst kaum überprüft werden könnten.

Es gibt im Verzeichnis ein Projekt unitTests.

Unit/Klasse	Funktion
<code>tulngame.Internal</code> <code>TIngameInternalTests</code>	Testet aller möglichen Szenarien der Punkte Berechnung und deren Ergebnissen.
<code>tulngame</code> <code>TIngameTests</code>	Testet die Korrektheit der <code>ingame.getPossibleActions</code> Funktion.
<code>tFmGame.utilts</code> <code>TFmGameUtilsTests</code>	Testet das Platzieren der Würfel in Raster des gesicherten und Sichergesicherten Bereiches.

### Units Test-Case Resultate



## GUI-Tests

Das GUI-Verhalten muss manuell getestet werden. Es gibt einen Cheat-Modus über „Alt+C“ aufzurufen. Über diesen lässt sich das Würfelverhalten steuern.

Um die Karten zu steuern, müssen speziell frisierte Dateien geladen werden.

Und dann steht dem manuellen Testen nichts mehr im Weg.

Besondere Aufmerksamkeit im Test sollen die Spielsituationen:

- Totenkopfinsel-Modus,
- Piratenschiff-Modus,
- „Wächterin-Karte,
- Schatzinsel-Karte,

## Piraten Karpern Programmierhandbuch

- ZugEnde,
- „Spielende-Piratenmagie“ erfahren.

Entsprechende Dateien werden erstellt.

Testdatei	Spielsituation
4player_close_to_win_normal_cardstack.pk	Test des Gewinns.
Pirateship_mode_pirateship2_card.pk Pirateship_mode_pirateship3_card.pk Pirateship_mode_pirateship4_card.pk	Test des Piratenschiff Modus
Treasureisland_card.pk	Test der Schatz Insel Karte derer <ul style="list-style-type: none"><li>• besondere Sichersichern Möglichkeit.</li><li>• Punkte Berechnung.</li><li>• </li></ul>
Test_corupted_number_of_players.pk	Test des Spiel-Ladens und dessen Fehlermeldungen.
Two_player_standard_beginning.pk	Test des Spiel-Ladens (unfrisierte Datei).
Skullisland_mode_skull1_card.pk Skullisland_mode_skull2_card.pk	Test der Totenkopf Karte Und deren Einfluss <ul style="list-style-type: none"><li>• schneller in den Totenkopfinsel-Modus zu gelangen</li><li>• Punkte Berechnung</li><li>• Vorzeitiges Ende.</li></ul>