Fach:	Verteilte Informationsverarbeitung
Studienheft:	VII01-03
Projekt:	Client-Server-Malprogramm
Dokumenttitel:	Phase 1: Analyse und Definition
Dokumentart:	Dokumentation
Erstellt von:	Martin Blankenstein
Erstelldatum:	26.08.2010

Inhaltsverzeichnis:

1 Projektstruktur

Bevor die erste Codezeile geschrieben wird, muss erst noch die Projektstruktur geklärt werden. Im Entwurf sind vier große EBCs entstanden:

- PaintTogetherStartSelector
- PaintTogetherServer
- PaintTogetherClient
- PaintTogetherCommunicater

Jede dieser EBCs besteht zum Teil wieder aus einzelnen kleineren EBCs, was aber für die Projektstruktur erstmal ignoriert wird. Für jede der vier EBCs benötigen wir eine eigene Projektmappe.

Anmerkung zur Projektstruktur bei EBCs:
Jede EBC besteht immer aus drei bis vier einzelnen Projekten. Die ersten drei Projekte sind dabei bei jeder EBC vorhanden und bestehen alle aus einer nicht startbaren Dll.

*Das erste Projekt besteht aus den Nachrichten, dieses Projekt heißt immer **"EBCName".Messages**. Hier sind nicht nur die Nachrichten der EBC sondern auch die Nachrichten der internen EBCs definiert.*

*Das zweite Projekt besteht aus den Interfaces der EBC und den Schnittstellenbeschreibungen der internen EBCs. Dieses Projekt heißt **"EBCName".Contracts**. Die Interfaces benutzten dabei die Nachrichten aus dem Nachrichtenprojekt.*

*Das dritte und letzte immer vorhandene Projekt beinhaltet die Implementierungen der in dem Contract-Projekt definierten Schnittstellen. Hier sind also die eigentlichen EBCs enthalten. Der Name des Projektes lautet nur **"EBCName"**.*

*Als viertes Projekt definiere ich bei bestimmten EBCs noch ein ausführbares Projekt (Konsolenanwendung oder WindowsForms-Anwendung). In dem vorliegendem Fall werde ich für die beiden EBCs PaintTogetherStartSelector, und PaintTogetherClient eine WindowsForms-Startanwendung definieren, die ausführbar ist und die einzelnen entworfenen Haupt-EBCs aus dem Implementierungsprojekt verdrahtet und dann startet. Für den PaintTogetherServer benötigen wir ebenfalls ein Startprojekt, allerdings eine Konsolenanwendung, damit dort die Lognachrichten erscheinen. Einzig die EBC PaintTogetherCommunicater besteht aus nur drei Projekten, da diese nur vom Server und Client verwendet wird - sie wird nie selbstständig als Anwendung gestartet. Der Name des Ausführungsprojektes ist jeweils **"EBCName".Run**.*

*Bei allen meinen Projekten definiere ich zusätzlich noch ein fünftes Projekt mit dem Namen **"EBCName".Test**. Dieses Projekt besteht nur aus nUnit-Testklassen die die Grundfunktionalitäten der einzelnen EBCs automatisiert testen sollen.*

Ich verwende schon während der Implementierung automatische Tests. Dies kann als Beginn der Testphase noch während der Implementierung betrachtet werden. Somit laufen die beiden Phasen Implementierung und Test in diesem Projekt ein Stück parallel ab. Dies halte ich unabhängig der Größe eines Projektes aber für sinnvoll und sogar unablässig. Es spielt in erster Linie dabei keine Rolle, ob die Erstellung der Testfälle vor der Entwicklung einer Funktionalität oder im unmittelbarem Anschluss daran stattfindet. Die beiden Phasen Implementierung und Test sind dabei immernoch voneinander getrennt, laufen aber parallel.

Persönliche Anmerkung:
In meiner bisherigen Arbeit habe ich sehr gute Erfahrungen mit der Testgetriebenen Entwicklung bzw. mit der sofortigen Erstellung automatischer Tests nach der Implementierung von testbaren Funktionen gemacht. Die Erstellung der Testfälle oder auch Testcases, erfolgt aus persönlicher Überzeugung, könnte aber mit Blick auf die Aufgabenstellung weggelassen werden. Meine Testphase wird trotz der automatischen Tests nach Fertigstellung der Software manuelle Tests der ClientServerAnwendung beinhalten, da die automatisierten Testfälle lediglich einzelne Funktionen, aber nicht das gesamte Softwaresystem testen.

Sichtbarkeit von Klassen in einer EBC-Projektmappe:
Damit nach außen später nur die Haupt-EBCs sichtbar ist, lege ich fest, dass alle internen EBCs, deren Interfaces sowie die internen Nachrichten als "internal" definiert werden. Zusätzlich sollen die "internal" definierten Klassen des Nachrichtenprojektes für das Schnittstellen- und das Implementierungsprojekt sichtbar sein (Einstellung in der Assembly-Info) sowie die internen Interfaces für das Implementierungsprojekt. Bei Verwendung der fertigen Dlls durch andere, werden dann nur die öffentlichen Klassen verfügbar sein, was hilft den Überblick zu bewahren.

2 Codeverwaltung

Wie in der ersten Phase, der Analyse und Definition angekündigt, werde ich die Software mit Unterstützung einer Quellcodeverwaltung entwickeln. Hierfür werde ich den kostenlosen Dienst von Google, "GoogleCode" verwenden. Hier die Adresse meines Projektes bei GoogleCode:

http://code.google.com/p/painttogether/

Notiz:
Leider konnte am 09.08.10 bis zum 10.08.10 um 16:00 Uhr nur lesend auf googleCode zugegriffen werden, weshalb mein erster CheckIn nicht nur aus den leeren Projektordnern besteht, sondern auch schon den Beginn der Implementierung der PaintTogetherCommunicater-EBC enthalten wird.

3 Verwendete Entwicklungswerkzeuge

Als Entwicklungsumgebung verwende ich Microsoft **Visual Studio** Team System 2008. Als integriertes Werkzeug in VS verwende ich das open source Addin **AnknSVN**, um die Synchronisierung mit dem SVN-Repository zu vereinfachen. Für die Erstellung von Testcases verwende ich **NUnit** und für deren einfache Ausführung das kostenpflichtige Visual Studio Addin **ReSharper** von JetBrains.

Visual Studio (2010):	http://www.microsoft.com/visualstudio
AnknSVN:	http://ankhsvn.open.collab.net/
TortoiseSVN:	http://tortoisesvn.tigris.org/
ReSharper:	http://www.jetbrains.com/resharper/
NUnit:	http://www.nunit.org/

4 Vorgehen bei der Implementierung

Wie im Abschnitt Codeverwaltung schon angedeutet werde ich bei der Implementierung zuerst die vier Projektordner (noch ohne Projekte) anlegen. Anschließend beginne ich mit der PaintTogetherCommunicater-EBC. Diese wird später vom PaintTogetherServer und dem PaintTogetherClient benötigt und bildet das funktionale Kernstück der gesamten Software "PaintTogether", denn hier sind alle Funktionen für die Client-Server-Kommunikation enthalten.

Zuerst bestimme ich die Verzeichnisstruktur für die gesamte Anwendung:

- trunk/
  - ☐ Docs/ - beinhaltet alle bei dem Projekt entstandenden Dokumente
  - ☐ PaintTogetherClient/ - Projektverzeichnis für den Client
  - ☐ PaintTogetherCommunicater/ - Projektverzeichnis für den Communicater
  - ☐ PaintTogetherServer/ - Projektverzeichnis für den Server
  - ☐ PaintTogetherStartSelector/ - Projektverzeichnis für die Startanwendung

Für die einzelnen EBC-Ordner definiere ich ebenfalls eine einheitliche Struktur:

- EBC-Name/
  - ☐ EBC-Name/ - enthält das Implementierungsprojekt
  - ☐ EBC-Name.Contracts/ - enthält das Schnittstellenprojekt
  - ☐ EBC-Name.Messages/ - enthält das Projekt mit den Nachrichten
  - ☐ EBC-Name.Test/ - enthält die automatischen UNit-Tests für diese EBC
  - ☐ Docs/ - optional, enthält Skizzen und Dokumente die nur für diese EBC gelten
  - ☐ EBC-Name.Run/ - optional, enthält ein ausführbares Projekt zum Start der EBC
  - ☐ EBC-Name.sln - die Projektmappendatei
  - ☐

4.1 Implementierung der PaintTogetherCommunicater-EBC

4.1.1 Verzeichnisstruktur aufbauen

Nach der initialen Erstellung des PaintTogetherCommunicater Verzeichnisses, sieht dieses wie folgt aus:

- PaintTogetherCommunicater/
  - ☐ PaintTogetherCommunicater/
    - ☐Properties/
      - AssemblyInfo.cs
    - ☐PaintTogetherCommunicater.csproj
  - ☐ PaintTogetherCommunicater.Contracts/
    - ☐Properties/
      - AssemblyInfo.cs
    - ☐PaintTogetherCommunicater.Contracts.csproj
  - ☐ PaintTogetherCommunicater.Messages/
    - ☐Properties/
      - AssemblyInfo.cs
    - ☐PaintTogetherCommunicater.Messages.csproj
  - ☐ PaintTogetherCommunicater.Test/
    - ☐Properties/
      - AssemblyInfo.cs
    - ☐PaintTogetherCommunicater.Test.csproj
  - ☐ PaintTogetherCommunicater.sln

4.1.2 Nachrichtenklassen anlagen

Nun beginne ich mit der eigentlichen Implementierung am PaintTogetherCommunicater. Zuerst erstelle ich alle Nachrichtenobjekte im Messages-Projekt. Dabei liegen die Nachrichtenklassen der PaintTogetherCommunicater-EBC direkt in dem Messages-Verzeichnis, alle anderen Nachrichtenklassen liegen dann jeweils in einem Unterverzeichnis, welches nach der internen EBC benannt ist, die diese Nachrichten benötigt.

Für die PaintTogetherCommunicater-EBC schreibe ich die Struktur der erstellten Nachrichten jetzt nocheinmal mit einer kurzen Beschreibung auf. Dabei ist immer auf den Entwurf zu achten, die hier erstellten Nachrichten sind eine reine Umsetzung des Entwurfs. Einzige Ausnahme stellen alle unter dem Unterverzeichnis “ClientServerCommunication” zusammengefasste Nachrichtenklassen dar. Hierbei handelt es sich nicht um Nachrichten für den direkten Austausch über Pins, sondern um den Nachrichteninhalt von Nachrichten die über Pins verschickt werden. Diese Nachrichteninhaltsklassen bilden alle zwischen Server und Client auszutauschenden Inhalte ab.

- PaintTogetherCommunicater.Messages/

- ☐ Properties/
  - ☐AssemblyInfo.cs
- ☐ PaintTogetherCommunicater.Messages.csproj
- ☐ ClientServerCommunication/ - enthält die austauschbaren, inhaltlichen Nachrichtentypen
- ☐ PTMessageDecoder/ - Nachrichten des PTMessageDecoder
  - ☐DecodeRequest.cs - Bytes -> Nachrichtenobjekt
  - ☐EncodeRequest.cs - Nachrichtenobjekt -> Bytes
- ☐ PTMessageXmlSerializer/ - Nachrichten des PTMessageXmlSerializers
  - ☐ToMessageRequest.cs - XML -> Nachrichtenobjekt
  - ☐ToXmlRequest.cs - Nachrichtenobjekt -> XML
- ☐ ConLostMessage.cs - Verbindung zu überwachten Verbindung verloren
- ☐ NewMessageReceivedMessage.cs - Neue Nachricht über Socketverbindung eingegangen
- ☐ SendMessageMessage.cs - Aufforderung zum Senden einer Nachricht über einen Socket
- ☐ StartReceivingMessage.cs - Socket auf eingehende Nachrichten überwachen
- ☐ StopReceivingMessage.cs - Überwachung eines Sockets auf eingehende Nachrichten beenden

Benennung:
Im Entwurf hatten die Nachrichten kürzere Namen, die ich hier verlängert habe um Konflikte mit anderen Namespaces zu vermeiden. Ebenso habe ich die EBCs “Decoder”, “Receiver” und “Sender” nach “PtMessageDecoder”, “PtMessageReceiver” und “PtMessageSender” umbenannt.

Überlegungen beim Erstellen der Nachrichtenklassen:
Beim Erstellen der Nachrichtenklassen ist mir aufgefallen, dass beim Entwurf nur jeweils einmal Senden und Empfangen vorgesehen ist. Es gibt also nicht für “Ermittle aktuellen Malbereich” oder “Male Punkt” unterschiedliche Pins, alles läuft bei der EBC über den “SendMessage”-Inputpin. Deshalb habe ich für alle über den Communicater versendbaren und empfangbaren Inhalte ein Interface erstellt. Es liegt zusammen mit allen inhaltlichen Nachrichten im oben schon aufgezähltem Unterverzeichnis “ClientServerCommunication”. Die schon oben angegebennen Nachrichtenklassen verwenden jetzt dieses Interface. Der Client bzw. Server muss dann ermitteln um welche Art von Nachricht es sich bei der empfangenen Nachricht handeln und dann mit dieser arbeiten. Trotz des Interfaces für die Nachrichteninhalte, ist eine eigene Implementierung außerhalb des Communicater nicht sinnvoll, da auch die Serialisierung für jedes Nachrichtenobjekt ersteinmal implementiert werden muss.

Das Interface für die inhaltlichen Nachrichten ist “IServerClientMessage”, von diesem gibt es verschiedene Implementierungen (abgeleitet aus den Nachrichten die beim Entwurf aufgezeigt wurden). Die Implementierungen des Interface habe ich in zwei Teile aufgeteilt,

Server und Client. Für jeden Teil gibt es einen Unterordner. Unter Client liegen die Nachrichten die vom Client gesendet und vom Server empfangen werden. Unter Server liegen die Nachrichten-Implementierungen die der Server an die Clients sendet und die die Clients dann verarbeiten.

Folgende Nachrichtenklassen habe ich herausgearbeitet:

- ClientServerCommunication/
  - ☐ IServerClientMessage.cs - Interface für Nachrichten
  - ☐ Client/ - Nachrichten die die Clients an den Server senden
    - ☐ConnectScm.cs - Verbindungsstartnachricht mit Farbe und Alias
    - ☐PaintScm.cs - Bemalter Punkt inkl. Farbe
  - ☐ Server/ - Nachrichten die der Server an die Clients sendet
    - ☐AllConnectionsScm.cs - Alle aktuell am Server verbundenen Alias + Farbe
    - ☐ConnectedScm.cs - Initiale Antwort auf “ConnectScm”-Nachricht vom Client mit Alias des Servers
    - ☐ConnectionLostScm.cs - Information über einen Alias + Farbe der nicht mehr an der Malerei beteiligt ist
    - ☐NewConnectionScm.cs - Information über einen Alias + Farbe eines neuen Beteiligten an der Malerei
    - ☐PaintContentScm.cs - Der aktuelle Malbereich als Bitmap
    - ☐PaintedScm.cs - Information über einen bemalten Punkt inkl. Farbe

Wie die Nachrichten und in welcher Reihenfolge sie später am Client und Server gesendet und empfangen werden, ist hier noch nicht wichtig. Beim Erstellen der Nachrichtenklassen kommt es erst einmal darauf an alle möglichen Information abzubilden, die später zwischen Client und Server ausgetauscht werden können.

4.1.3 Contracts anlagen

Bevor die Schnittstellen erstellt werden können, müssen die internen Klassen aus dem Messagesprojekt für die anderen Projekte sichtbar gemacht werden. Diese Einstellung kann man in der AssemblyInfo.cs des Messagesprojektes vornehmen. Ich habe einfach die drei Einträge

```
[assembly: InternalsVisibleTo("PaintTogetherCommunicater")]
[assembly: InternalsVisibleTo("PaintTogetherCommunicater.Contracts")]
[assembly: InternalsVisibleTo("PaintTogetherCommunicater.Test")]
```

an das Ende der AssemblyInfo geschrieben. Bei “InternalsVisibleTo” kann man einen Namen einer anderen Assembly angeben, die dann auch auf die internen Klassen, Eigenschaften, Methoden usw. zugreifen kann.

Das erste anzulegende Interface ist das Hauptinterface der EBC. Es muss alle In- und Outputpins entsprechend des Entwurfs besitzen.

*Anmerkung zur Umsetzung eines Pins in C# und von mir für das Projekt festgelegte Benennung: In C# werden die Inputpins durch Methoden dargestellt. Diese Methoden heißen "**Process**MessageName". Wenn der Methodenname auf **Request** endet, dann handelt es sich um einen Inputpin, der die Resulteigenschaft an dem Nachrichtenobjekt füllt. Alle Input-Methoden haben die Signatur:*

*(public) void Process**MessageName**(Nachrichtentyp name)*

Es handelt sich also immer um eine Methode ohne Rückgabewert und mit nur einem Übergabeparameter.

*Outputpins werden durch Events realisiert. Später werden dann jeweils ein Output-Event mit einer Input-Methode verbunden. Der Name des Events beginnt immer mit "**On**", die Signatur ist immer die folgende:*

*(public) event Action<Nachrichtentyp> On**MessageName**;*

*Diese Vorgabe verwende ich jetzt bei der Contracterstellung und kann im Quellcode nachvollzogen werden. Events die Outputpins mit Ergebnis darstellen beginnen mit **OnRequest**.*

Das nach den Vorgaben für Benennung und den Vorgaben aus dem Entwurf entstandene Interface heißt "IPaintTogetherCommunicater". Es besitzt jetzt die im Entwurf beschrieben Input-Pins als Eingangsmethoden und die beschriebenen Output-Pins als Events.

Nach Erstellung des Platineninterfaces folgt die Erstellung der Schnittstellen für die einzelnen internen EBCs. Diese sind ebenfalls aus dem Entwurf abgeleitet und heißen:

- PtMessageReceiver (im Entwurf nur Receiver genannt)
- PtMessageSender (im Entwurf nur Sender genannt)
- PtMessageDecoder (im Entwurf nur Decoder genannt)
- PtMessageXmlSerializer (Name aus Entwurf übernommen)

Anders als für die Nachrichtenklassen, erstelle ich für die vier Interfaces keine weiteren Unterordner, da die im Contracts-Projekt dann insgesamt existierenden fünf Klassen sehr übersichtlich sind. Die vier internen EBC-Schnittstellen werden wieder als internal markiert, damit sie später nicht von außer verwendet werden.

4.4.4 Implementierung der ersten internen EBC

Bei der Implementierung der in den Contracts erstellten Interfaces, kann man mit jeder der vier internen EBCs beginnen. Denn jede EBC ist für sich abgeschlossen und selbständig. Für die erste Implementierung wähle ich den PtMessageSender. Die Aufgabe der EBC ist der Versandt einer

Nachricht über eine verbundene Socketverbindung. Zu Beginn lege ich erst eine leere Implementierung des Interfaces IPtMessageSender mit dem Namen PtMessageSender im Projekt "PatintTogetherCommunicater" an. (vorher wieder in der AssemblyInfo.cs im Contracts-Projekt die Sichtbarkeit der internen Klassen erweitert)

Nach der leeren Implementierung erstelle ich zuerst ein paar einfache Testcases im Test-Projekt.

Anmerkung zur Benennung von Testfällen:
In meinem gesamten Projekt verwende ich ausschließlich englische Namen und Bezeichnungen. Bei den Testcases mache ich aber ein Ausnahme, hier besteht jeder Testfall aus einem deutschen Wort oder Satz der aussagt was getestet wird. Ich finde das die Lesbarkeit der Testberichte dadurch deutlich steigt. Später sieht man schnell bei fehlgeschlagenen Testcases wo das eigentliche Problem liegt.

Anmerkung zur Strukturierung von Testfällen:
*Für eine zu testende Klasse erstelle ich im Test-Projekt auf gleicher Ebene einen Ordner der "**Klassenname**CS" heißt. Dort gibt es dann für jede zu testende Methode eine eigene Klasse. Diese erhält den Namen "**Methodename**Test.cs". Eine Testklasse testet also eine einzige Methode. Eine Ausnahme stellen Testklassen da, die "IntegrationsTest.cs" heißen. Sie testen das Zusammenspiel mehrerer Funktionen bzw. Komponenten zu testen.*

Ich habe einen Testcase mit genauer Kommentierung als Beispiel erstellt. In der Klasse "PaintTogetherCommunicater.Test/PtMessageSenderCS/ProcessSendMessageTest.cs" gibt es den Testfall "Einfache_Nachricht_senden". Hier wird eine Nachricht zusammengebaut und über die Inputmethode "ProcessSendMessage" gesendet. Anschließend wird geschaut ob der mit dem SenderSocket verbundene Socket auch die richtigen Daten empfangen hat. Bei den weiteren Testcases verzichte ich auf zusätzliche Kommentierungen. (Testcases sind immer nach dem gleichen AAA-Prinzip aufgebaut - Arrange, Act, Assert).

Nach der Erstellung des Testcases schlägt dieser zuerst fehl, weil die Funktionalität noch nicht implementiert wurde. Jetzt erst folgt die Implementierung der "getesteten" Inputmethode.

Bei der Implementierung der Inputmethode habe ich noch eine private, statische Eigenschaft "Log" für das Logging über log4net hinzugefügt. Dies werde ich bei allen weiteren Implementierungen (wo ich Logging für wichtig erachte) so machen. Da bei der Implementierung der Inputmethode auch gleich der Outputpin bzw. das Event verwendet wurde, ist der PtMessageSender nun vorerst fertig und entspricht den Vorgaben aus dem Enwurf. Verarbeitung des Inputs und Erstellung des Outputs funktionieren jetzt.

Quellcodeauszug aus der Inputmethode - ProcessSendMessage:

Da der Input aus dem Socket und einem inhaltlichen Nachrichtenobjekt besteht, aber über einen Sockets Bytes verschickt werden, muss das Nachrichtenobjekt zuerst in Bytes umgewandelt werden. Für die Umwandlung ist der PtMessageDecoder zuständig, also wird zuerst der Auftrag zur Kodierung der Nachricht gestellt:

```
// Kodierungsauftrag erstellen und "abschicken"
var encodeRequest = new EncodeRequest { Message = message.Message };
OnRequestEncode(encodeRequest);
```

Die Bytes die jetzt am encodeRequest in der Result-Eigenschaft gesetzt sein müssen, werden jetzt über die Socketverbindung versendet. Den Sendevorgang habe ich in die Methode SendContent ausgelagert:

```
// Nun müsste die Resulteigenschaft am request mit der
// kodiertern Nachricht gefüllt sein. Dieses Ergebnis jetzt
// über den Socket verschicken
SendContent(message.SoketConnection, encodeRequest.Result);
```

Quellcodeauszug aus der Sendemethode - SendContent:

Bei der Übertragung von Bytestroms habe ich während der Implementierung des MessageReceivers herausgefunden, dass eine Trennung der Nachrichten durch bestimmte Marken vorgenommen werden müssen. Im Receiver habe ich deshalb die beiden Byte-Blöcke “StartBlock” und “EndBlock” definiert. (Ich muss hier leider der Implementierung des Receivers vorgreifen, da ich zuerst nur die kodierten Bytes gesendet habe, was allerdings beim Empfangen nicht ausgereicht hat) Den zu versendenen Inhalt umrahme ich vor dem Senden deshalb mit dem Start- und Endbyteblock:

```
var bytes = new List<byte>();
bytes.AddRange(PtMessageReceiver.StartBlock);
bytes.AddRange(content);
bytes.AddRange(PtMessageReceiver.EndBlock);
socket.Send(bytes.ToArray());
```

4.4.5 Implementierung der weiteren internen EBCs

Bei der Implementierung der letzten drei internen EBCs werde ich wie bei dem PtMessageSender vorgehen und nur die Besonderheiten zusätzlich schriftlich festhalten.

Als Besonderheit hat sich gleich zu Beginn der PtMessageReceiver herausgestellt. Obwohl ich mit den Socket-Verbindungen bereits zu Beginn der Arbeit experimentiert habe, um herauszufinden in wie weit es die Architektur beeinflusst, sind einige Probleme aufgetaucht. Ich habe wichtige Erkenntnisse zu den Sockets, die ich verbindungorientiert über TCP als Streamverbindung benutzte, gewonnen. So werden beim Empfangen der gesendeten Nachrichten

diese aneinandergereiht, ohne ein Trennzeichen. Dieses Problem habe ich gelöst, indem beim Senden eine Startbytefolge vor und eine Endbytefolge nach dem eigentlichen Inhalt gesendet wird. Beim Empfangen werden dann zwischen einem StartByteBlock und einem EndByteBlock die Bytes herausgetrennt und für die Nachrichtensignalisierung verwendet. Dafür musste ich auch das Senden im PtMessageSender leicht anpassen. (wie schon bei der Implementierung des PtMessageSender geschildert)

Ein weiteres Problem gab es bei der Übertragung mehrerer Nachrichten in einer Folge. Ein angelegter Testcase hat jeweils nur ca. 93 % der gesendeten Nachrichten erfolgreich empfangen. Bei 1000 Nachrichten gingen immer ca. 70 Nachrichten verloren. Die Ursache war, dass ich den Rückgabewert der Receive-Methode beim Senden über eine Socketverbindung nicht ausgewertet habe. In meine Nachrichtendaten haben sich somit Nullfolgen eingeschlichen. In so einem Fall konnte dann die Nachricht nicht mehr erfolgreich dekodiert werden.

4.4.6 Implementierung der Platine PaintTogetherCommunicater

Jetzt sind die internen EBCs vorerst abgeschlossen und nun fehlt noch die Implementierung der Platine “PaintTogetherCommunicater”. Die Entwicklung von Platinen unterscheidet sich stark zu der von Bausteinen. Eine Platine hat selbst annähernd keine Funktionalität, dafür muss sie aber viele verschiedene Bausteine/Platinen miteinander verbinden.

Anmerkung zu Platinen:
Eine Platine kümmert sich nur um die Verdrahtung verschiedener in ihr zusammengefasster EBCs. Hier werden die Input- und Outputpins der Platine mit den Pins der Bausteine verdrahtet. Zusätzlich werden noch bei Bedarf Pins zwischen den Bausteinen verknüpft.

Die gesamte Verknüpfung wird hier im Konstruktor durchgeführt.

4.2 Implementierung des PaintTogetherServers

Nach der Fertigstellung des Communicater entwickle ich jetzt zuerst die Server-EBCs.

Bei dem PaintTogetherServer handelt es sich nicht wie bei dem Communicater um eine EBC, er besteht aus drei größeren EBCs. Zuerst lege ich aber wieder die Projektstruktur entsprechend meiner Bestimmungen an und erhalte hier fünf Projekte (PaintTogetherServer , PaintTogetherServer.Test, PaintTogetherServer.Messages, PaintTogetherServer.Contracts, PaintTogetherServer.Run). Bei den im PaintTogetherServer benötigten libs, kommen jetzt auch die Dlls des Communicaters hinzu.

Nach der Erstellung der drei Haupt-EBCs (Portal, Server, Adapter und deren interner EBCs) erstelle ich die ausführende Konsolenanwendung. In dem Run-Projekt befindet sich neben der

üblichen Program.cs nun auch noch eine Server.cs. Diese kann man ebenfalls als eine EBC-Platine betrachten, da hier jetzt die drei EBC “raufgelötet” und verdrahtet werden. Die Server.cs kümmert sich nur um die Verknüpfung ihrer drei Bausteine. In der Program.cs werden die Parameter wie nach Vorgabe aus Phase 1 :Anforderung/Definition behandelt und ausgewertet.

4.3 Implementierung des PainTogetherClients

Mit dem Portal des PaintTogetherClients gibt es die erste WindowsForm, die es zu entwickeln gilt. Vorher lege ich aber auch hier erst die Projektmappe und Projekt-Ordner an. Auch der PaintTogetherClient besteht wie der Server aus fünf Projekten (PaintTogetherClient, PaintTogetherClient.Test, PaintTogetherClient.Messages, PaintTogetherClient.Contracts, PaintTogetherClient.Run).

Bei der Implemtentierung der Contracts bin ich auf eine Unstimmigkeit im Entwurf gestoßen. Hier wurde als initiale Startnachricht für den Client, nur Servername und Port verlangt. Allerdings muss auch der Alias sowie die gewünschte Malfarbe mit übergeben werden. Der Server benötigt diese Informationen für die Beteiligtenliste und der Client um zu wissen, mit welcher Farbe der Anwender eigentlich malt. Ich habe den Entwurf an der entsprechenden Stelle geändert (Entwurf: ClientServerAdapter).

Bei der Implementierung des PtClientStarter ist mir aufgefallen, dass in dem Entwurf gar nicht die initiale Connected-Nachricht vom Server verarbeitet und an den ClientStarter weiter geleitet wird. Ich habe im Quellcode deshalb den neuen Inputpin “ProcessConnectedMessage” am PtClientStarter (ebenfalls an der Platine PtClientCore) und dem entsprechend am PtServerConnectionMananger den Outputin “OnConnected” (ebenfalls an der Platine PtClientServerAdapter). Mit dieser weiteren Nachricht erhält der Client den Aliasnamen des Servers als Information. Dieser wird in der ClientGUI laut Entwurf für den Titel des Malfensters benötigt. Damit die neuen Pins erstellt werden können, habe ich auch die neue Nachrichtenklasse “ConnectedMessage” eingeführt (enthält nur den Alias).

Funktionsweise der ClientStarter-EBC. Nach dem erfolgreichem Aufbau der Serververbindung wird erst noch auf den Eingang von zwei Nachrichten des Servers gewartet. Als erstes wird auf die Verbindungsnachrichtbestätigung mit dem Alias des Server gewartet und dann auf den initialen Malbereich. Erst wenn dieses beiden Nachrichten empfangen wurden, dann löst die EBC die Initialisierungs-Outputs für das Portal und die Malbereichsverwaltung auf.

Die Implementierung des “PtClientAdapterStarters” erfolgt der Verbindungsaufbau zum Server. Dabei muss die Art der Socketverbindung angegeben werden. Ich habe hier wie im Server als Verbindungstyp “TCP” und als Sockettyp “Stream” angegeben.

Nach Fertigstellung der beiden Platinen (inkl. der internen EBCs) “Adapter” und “Core” erstelle ich die Portal-EBC. Hier handelt es sich um die erste WindowsForm. Sie implementiert jetzt die Vorgaben des PortalInterfaces. Da nicht sicher ist, dass das Nachrichtenempfangen im Hauptthread stattfindet, müssen alle Funktionen die an den GUI-Komponenten etwas verändern, vorher in den Hauptthread überführt werden. Das mache ich mit der durch das Framework bereitgestellten Klasse “SynchronisationContext”. Die Verwendung ist an den Inputpin-Methoden des PtClientPortals gut zu sehen. Die GUI baue ich mit dem WindowsFormDesinger nach den Vorgaben aus dem Entwurf zusammen.

Nach der Fertigstellung der Portal-EBC muss im Run-Projekt noch die Parameterauswertung für den Clientstart geschrieben werden. Wie auch schon im Run-Projekt des Servers werde ich hier eine EBC-Platine erstellen. Die Platine heißt “Client” und besteht aus den drei EBCs (PtClientPortal, PtClientCore, PtClientAdapter).

4.4 Implementierung des StartSelector

Die Implementierung des StartSelectors werde ich vorerst nicht durchführen, da Server und Client auch ohne StartSelector funktionieren. Die Implementierung werde ich erst durchführen wenn Server und Client weitesgehend funktionieren, weshalb ich an dieser Stelle in die vierte Phase wechsle, in die Testphase.

<Erste Tests und Fehlerbehebungen durchgeführt, siehe Dokumentation der Testphase>

Nach dem ersten erfolgreichem Kurztest von Server und Client erstelle ich die Startanwendung nach dem gleichem Verfahren wie bereits die anderen drei Projekte (PT-Client, PT-Server, PT-Communicater).

Zuerst lege ich wieder die Projektemappe mit den Projekten an. Hier sind es wieder fünf Projekte (PT-StartSelector, PT-StartSelector.Messages, PT-StartSelector.Contrcats, PT-StartSelector.Test, PT-StartSelector.Run).

Anschließend erstelle ich wieder alle Nachrichtenklassen im Messages-Projekt, entsprechend der Kommunikation des Entwurfs. Nach Erstellung der Nachrichtklassen folgt wieder die Definition der Interfaces für die vier Haupt-EBCs (das Portal, den StartSelector und die beiden Adapter). Im Entwurf sind bisher keine internen EBC aufgelistet worden, ggf. entstehen bei der Implementierung noch welche.

Bei der Implementierung der EBC StartSelector benötigte ich für die Prüfung der Serverdaten eine Funktion zum Ermitteln der IP eines Servernamens und zur Prüfung ob diese IP vorhanden und erreichbar ist. Genau diese Funktion hatte ich schon im PaintTogetherClient und dort in der

EBC PtClientAdapterStarter implementiert. Da beim “Clean Code Development“ und generell das Kopieren größerer Quellcodemengen unschön ist, habe ich im Pt-Client eine Klasse “PtNetworkUtils.cs” erstellt und dort die Funktionen hineingezogen. Der StartSelector referenziert jetzt einfach die PaintTogetherClient.dll und hat so auf die PtNetworkUtils-klasse Zugriff. (Alternativ hätte ich die Utilsklasse auch als eigenes EBC in ein eigenes Projekt namens PtNetworkUtils aufsetzen können. Diese EBC hätte ich dann einfach im Client und im StartSelector verwenden können, genau wie die Communicater-EBC. Der Aufwand eines zusätzlichen EBC-Projektes welches wieder aus vier Unterprojekten besteht war mir für diese eine Funktion aber zu groß)

Bei der Erstellung des Portals benötige ich mehrmals die Prüfung, ob der eingegebene Alias gültig ist sowie ob Port, Breite und Höhe richtig eingegeben wurden. Dazu erstelle ich eine ValidateInputUtils-klasse, die mir diese Angaben überprüft. Da ich bei den für den Alias gültigen Zeichen mit einer Regular Expression am einfachsten die Eingabe prüfen kann und diese Prüfung auch noch beim Client und Server einbauen möchte, macht es Sinn diese RegEx nur an einer Stelle zu definieren. Da der Server davon ausgeht, dass die Clientverbindungen schon über einen gültigen Alias verfügen, gehört die RegEx nicht ins Server-Haupt-Projekt. Die Prüfung wird bei den Startparameterauswertungen am Server und Client sowie im StartSelector durchgeführt. Beim StartSelector hat die RegEx somit nichts zu suchen, denn Server und Client benötigen diese Anwendung nicht. Da der Server auch ohne den Client arbeiten kann, erstelle ich die RegEx in dem Server-Nebenprojekt “PaintTogetherServer.Run”. Dort kommt die RegEx als Konstante an die StartServerParams, die von dem “PaintTogetherClient.Run-Projekt” und dem StartSelector referenziert werden kann.

Die Implementierung ist hiermit fertig. Es folgen für ggf. noch Fehlerbeseitigungen für Fehler die während der Testphase gefunden wurden.