Zeus Basic Library

Zeus-Framework

Autor	Benjamin Hadorn
E-Mail	b_hadorn@bluewin.ch
Ablage/Website	http://www.xatlantis.ch/zeusbase.html
Datum	13.09.08
Version	0.6.3

Inhaltsverzeichnis

1 Einleitung

Die Basis Bibliothek des Zeus-Frameworks bietet eine umfangreiche Sammlung an Klassen für allgemeine Softwareentwicklung in C++ an. Dieses Dokument beschäftigt sich ausschliesslich mit den Klassen der Basis-Bibliothek und beinhaltet zudem die Bedienanleitung zur Zeus-Framework Applikation und dem RMI für C++.

2 Module Object Model Spezifikation 2.0

Die Module Object Model Spezifikation dient der allgemeinen Vereinbarung und der Definition von Objekten in der Framework-Umgebung. Das Zeus-Framework ist eine Referenzimplementation dieser Spezifikation.

Das Module Object Model (MOM)[1]definiert den Aufbau eines dynamisch und modularen Frameworks. Das MOM besteht primär aus einem Baum von Softwarekomponenten (Objekten), die dynamisch geladen werden können.

2.1 Eigenschaften des MOM

Das MOM hat durch seine einheitliche Entkopplung von Komponenten folgende Eigenschaften:

- Hohe Wiederverwendbarkeit der Komponenten. Das ermöglicht hervorragendes Software-Recycling.
- Dezentrale Entwicklung von Softwarekomponenten. Durch klare Schnittstellen können komplexe Systeme dezentral entwickelt werden.
- Stufenweises Entwickeln. Das System muss nicht schon beim Projektstart ins Detail geplant werden. Komponenten können auch später spezifiziert, entwickelt und integriert werden.
- Erweiterbarkeit der Software. Durch sich ändernde Kundenwünsche kann die Software auch später noch einfach erweitert werden.
- Durch die flexible Konfiguration kann die Software nach Kundenwunsch angepasst werden. Es können beliebige Komponenten in den Objektbaum eingehängt oder entfernt werden.
- Gute Wartbarkeit von komplexen Systemen. Einzelne Komponenten können ausgewechselt oder neu geschrieben werden, ohne dass das gesamte System neu bearbeitet werden muss.

Das Modell hat auch seine Schwachpunkte, die aber durch klare Richtlinien bei der Entwicklung wieder wett gemacht werden können:

- Die Vernetzung der Komponenten ist nicht begrenzt. Legt ein Projekt-Team die

1 MOM 1 wurde bei Studer.com definiert. [STWIN]

Vernetzung nicht fest, kann ein System entstehen, welches sehr viele und unüberblickbare Abhängigkeiten besitzt.

2.2 Aufbau und Definition

Die MOM Spezifikation wird in 2 Anforderungskategorien unterteilt:

1. Die Basis-Anforderungen definieren die MUSS-Kriterien

2. Die erweiterten Anforderungen definieren Zusätze, welche implementiert werden können.

Jedes Objekt ist durch eine einheitliche Schnittstelle von den anderen Objekten entkoppelt. Es gibt 2 Arten von Objekt-Typen:

- *X-Objekt*: Das X-Objekt ist ein einfaches Objekt, welches eine Softwarekomponente repräsentiert. Das X-Objekt implementiert die Basis-Anforderungen der MOM Spezifikation.

- *Modul*: Das Modul ist eine Komponente welche eine ganze Software-Bibliothek repräsentiert. Das Modul implementiert die Erweiterungen der MOM Spezifikation. Zudem kann das Modul eine Session von einer Bibliothek erzeugen (Kontext).

2.2.1 Basis Anforderungen

Das MOM beschreibt sich folgendermassen:

- X-Objekte können Endknoten sein, welche im System eine Dienstleistung oder eine bestimmte Aufgabe erfüllen.

- X-Objekte können Knoten sein, welche andere X-Objekte zu logischen Einheiten gruppieren.

- X-Objekte können eingefroren oder freigeschaltet werden.

    - Beim Einfrieren der X-Objekte werden alle Referenzen freigegeben, die von anderen X-Objekten stammen. Eingefrorene X-Objekte reagieren nicht mehr.

    - Beim Freischalten können die benötigten Referenzen von anderen X-Objekten angefordert und gebraucht werden.

- Das MOM definiert den Datenaustausch zwischen X-Objekten durch vordefinierte Schnittstellen.

- Ein X-Objekt kann seine Kinderobjekte verzögert erstellen. Die Kinderobjekte werden erst erstellt, wenn jemand aus dem System ein solches Kinderobjekt referenzieren möchte.

2.2.2 Erweiterungen

Die Erweiterungen definieren spezielle X-Objekte, die *Module*:

- Module können aktiviert oder deaktiviert werden. In einem System ist immer nur ein Modul aktiv (Siehe Abbildung 1). Die Aktivierung wird vor allem bei graphischen Anwendungen verwendet. Das aktive Modul erhält hier den Eingabefokus.

- Module laden ihre Ressourcen (Shared Libraries, Dokumente oder stellen Verbindungen zu anderen Systemen her). Hat ein Modul, welches aktiviert oder von dem Objekte angefordert wurde, seine Ressourcen noch nicht geladen, erfolgt nun der Ladeprozess. Beim Deaktivieren bleiben die geladenen Ressourcen meistens erhalten.

- Manager sind Module die andere Module verwalten und zu logischen Gruppen zusammenfassen.

- Werden in einem Objektbaum Module und Manager eingesetzt, ist es wichtig, dass jedes Modul und jeder Manager ein Kindobjekt von einem Manager ist. Der Wurzelknoten muss also ein Manager sein. Nur so ist der Aktivierungsprozess nach Definition gewährleistet.

Jeder Manager und jedes Modul kann beliebig viele X-Objekte als Kinderobjekte besitzen.

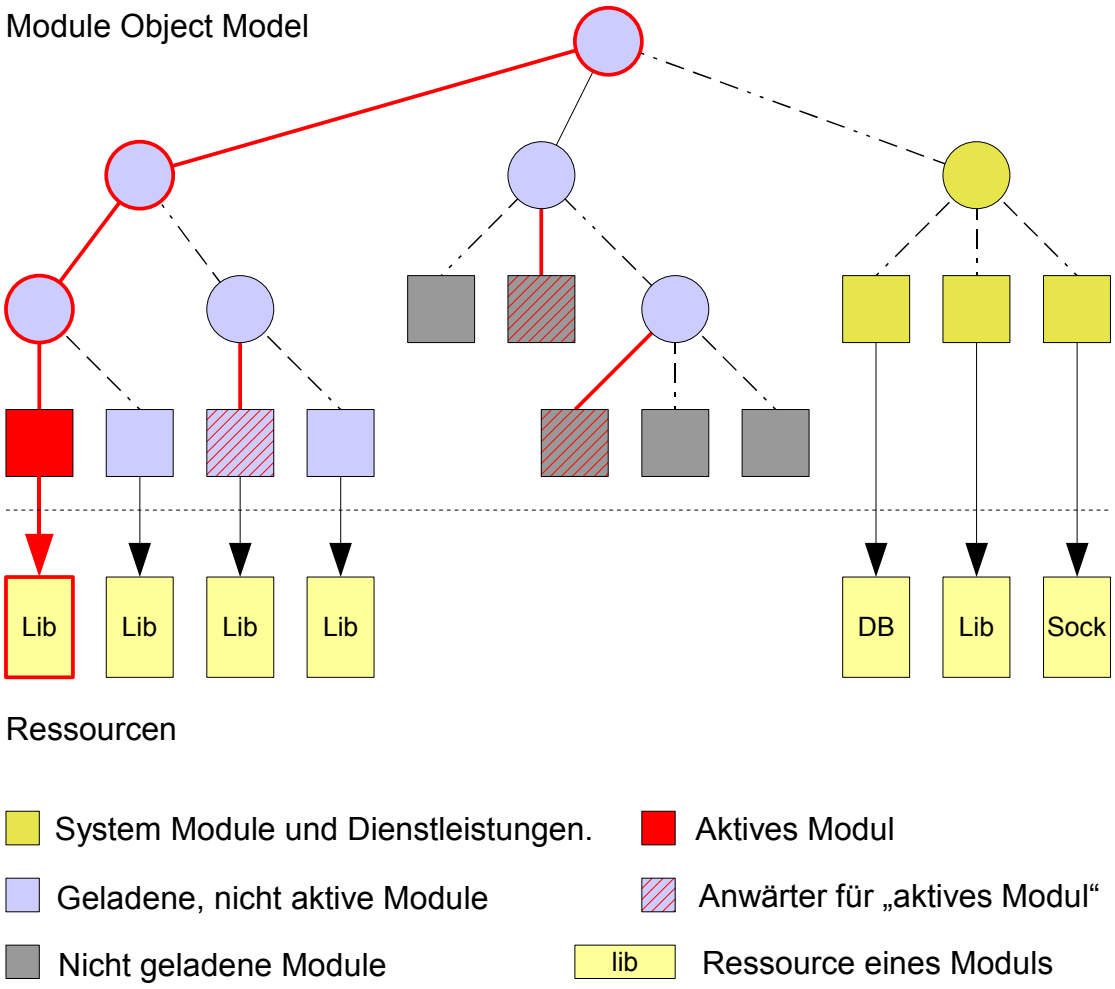


Abbildung 1: Diese Darstellung verdeutlicht den Aufbau und die Funktionsweise des MOM mit Modulen. Kreise symbolisieren die Manager, die Quadrate die Module. Das aktive Modul (rot) ist durch einen Aktivierungspfad gekennzeichnet. Die Systemmodule werden immer geladen. Sie stehen dem System generell zur Verfügung.

3 Das Klassen-Framework

Zur Entwicklung das Frameworks wurde ein eigenes Klassen-Framework aufgebaut. Es ermöglicht das Entwickeln und Betreiben von Applikationen in Linux und Windows. Das Klassen-Framework umfasst:

- Verschiedene Datentypen, wie Listen, Zeichenketten, Variant etc.
- System API
  - Threading
  - Networking
  - Messaging
  - Kommunikationskanäle
- Die MOM 2 Implementation
- Serialisierung von Objekten
  - Streaming
  - Objektfabriken
- Remote Method Invocation für C++
- Diverse Hilfsklassen zur Entwicklung von Applikationen

Das Framework kann beliebig erweitert werden. Es ist möglich ein eigenes Framework zu entwickeln, welches alle Eigenschaften des Zeus-Frameworks erbt. Die Klassen wurden in einer Bibliothek zusammengefasst, welche als Klassen-Bibliothek (`ZeusBase.dll` oder libZeusBase`.so`) zur Verfügung steht.

3.1 Vorschriften bei der Entwicklung mit Zeus

Das Zeus-Framework schreibt gewisse Vorgehensweisen bei der Entwicklung von Applikationen vor. Diese Vorschriften basieren einerseits auf Erfahrung und sind anderseits notwendig, um gewisse Plattformen zu unterstützen.

Es wird jedem Entwicklungsteam empfohlen, diese Vorschriften zu beachten und weitere Einschränkungen vorzunehmen.

3.1.1 Schnittstellen

Schnittstellen sind ein wesentliches Konzept im Zeus-Framework. Jeder komplexere Datentyp (wie Zeichenketten, Listen etc.) braucht eine Schnittstelle. Auch jedes Objekt implementiert mindestens die Basisschnittstellen `IZUnknown`.oder `IValueType` Das hat folgende Gründe:

- Wenn die Objekte über Schnittstellen kommunizieren, sind die Abhängigkeiten relativ klein.
    - Die Wartung und Erweiterbarkeit von Software wird wesentlich besser.
    - Das System wird übersichtlicher.
    - Es vereinfacht das Zusammenarbeiten im Projektteam.
- Objekte können irgendwo in einem Speicher sein, sie können sich also auch in einen lokalen Heap befinden.
    - Das geschieht zum Beispiel, wenn eine Borland C++ Applikation eine Bibliothek lädt, die in Visual C++ entwickelt wurde. Die Applikation und die Bibliothek besitzen beide unterschiedliche Heaps. Die Verwaltung der Objekte darf nur vom entsprechenden Heap Organizer durchgeführt werden.
    - Manipulationen auf dem falschen Heap (wie `realloc`, `new` und `delete`) führen zu Problemen. Deshalb ist ein spätes Binden (late binding) notwendig, damit die Aktion im richtigen Heap ausgeführt wird.
- Eine gute Architektur bedient sich der Möglichkeit von Schnittstellen.
    - Trennen von Definition und Implementation.
    - Abstraktes Denken verringert Fehler im Design und Code.

Bei der Entwicklung von Schnittstellen in C++ ist auf folgendes zu achten:

- Alle Methoden sind abstrakt definiert.
- Schnittstellen dürfen keine Attribute besitzen. (Auch keine statische Attribute. Diese sind eh nicht global, sondern pro Modul einmal vorhanden. Besser Singletons verwenden!).
- Alle Übergabeparameter sind primitive Datentypen oder Schnittstellen. Es dürfen keine konkreten Objekte übergeben werden. Die Schnittstellen werden als Referenzen oder Zeiger übergeben.
- Wird von einer Schnittstelle eine andere Schnittstelle verlangt, muss der

Referenzzähler um eins erhöht werden. Diese Schnittstelle darf nicht direkt als Rückgabewert zurückgegeben werden, da in C++ die Rückgabe ins Leere gehen kann (verursacht ein Ressourcen Leck).

Der letzte Punkt ist sehr wichtig. Alle angeforderten Schnittstellen müssen wieder freigegeben werden. Beispiel eines Schnittstellen-Benutzers:

Rückgabe von Schnittstellen

```
TString strName = L"./CellSystem1/MyCell";
IXObject* pMyCell = NULL;
if (pRoot->getObject(strName, pMyCell) == RET_NOERROR)
{
  ...
  pMyCell->release();
}
```

Tabelle 1: Bei der Rückgabe einer Schnittstelle wird der interne Referenzzähler automatisch erhöht. Deshalb muss ein `release()` aufgerufen werden. Wird dies nicht gemacht, bleibt das Objekt im Speicher und wird nie mehr gelöscht. Das könnte ein Systemabsturz auslösen.

Das Freigeben der Schnittstelle kann mittels Auto-Pointer eleganter gelöst werden:

Rückgabe von Schnittstellen

```
TString strName = L"./CellSystem1/MyCell";
TAutoPtr<IXObject> ptrMyCell = NULL;
if (pRoot->getObject(strName, ptrMyCell) == RET_NOERROR)
{
  ...
}
```

Tabelle 2: Mit der Klasse TAutoPtr wird eine Schnittstellenreferenz verwaltet. Da der Auto-Pointer ein Objekt auf dem Stack ist, wird nach Verlassen des Scope automatisch die Schnittstelle freigegeben.

Beispiel einer Implementation einer solchen Methode:

Rückgabe von Schnittstellen

```
Retval MQUALIFIER Ttest::getObject(const IString& rName, IXObject*& rpObject)
{
  Retval retValue = RET_INVALID_PARAMETER;
  IXObject* pObjectToFind = this;

  //Try to find object
  ...
  if (pObjectToFind != NULL)
  {
    //Return object
    rpObject = pObjectToFind;
    rpObject->addRef();
    retValue = RET_NOERROR;
  }
  return retValue;
}
```

Tabelle 3: Wird bei solchen Methoden ein `RET_NOERROR` zurückgegeben, muss der interne Referenzzähler um eins erhöht werden. Bei jedem anderen Rückgabewert dürfen keine Referenzzähler erhöht werden.

3.2 Objekt Typen

Das Zeus-Framework unterscheidet prinzipiell zwischen 2 Arten von Objekten, den *shared objects* und den Wert-Typen (*value types*)

3.2.1 Shared Objects

Zu den *shared objects* gehören folgende Klassen:

- Klassen die aus Bibliotheken exportiert werden
- Komplexere Klassen die sich nicht als Wert-Typ eignen
- Servant-Klassen des Remote Method Invocation RMI
- Singletons

Die *shared objects* implementieren immer die Basisschnittstelle `IZUnknown` und sind meistens von der Klasse `TZObject` abgeleitet, welche die Referenzenverwaltung bereits implementiert.

3.2.1.1 Die Basisschnittstelle IZUnknown

Die Basisschnittstelle für globale und *shared objects* heisst `IZUnknown` und regelt die folgenden Zugriffe:

- Verwaltung von Referenzen. Die Verwaltung regelt, wann ein konkretes Objekt gelöscht wird. Ein direktes `delete` führt zu Fehler und sollte vermieden werden.

- Casting zu anderen Schnittstellen. Da eine Art Mehrfachvererbung gebraucht wird, können durch direktes casting falsche Zeiger entstehen. Das geschieht vor allem beim Casten von einer beliebigen Schnittstelle zur anderen. Eine Ausnahme ist das Casting auf eine Basisklasse.

Diese Basis-Schnittstelle ist zu der COM Schnittstelle IUnknown kompatibel und kann deshalb auch als solche eingesetzt werden.

Die Methoden der Basisschnittstelle sind definiert als:

API	`zeusbase/System/Interfaces/IZUnknown.hpp`
	`Retval MQUALIFIER askForInterface(const InterfaceID& rInterfaceID, IZUnknown*& rpIface);` Diese Methode dient dem Abfragen und ggf. Casten auf eine andere Schnittstelle.
	`void MQUALIFIER addRef() const;` Diese Methode erhöht den internen Referenzzähler um eins.
	`void MQUALIFIER release() const;` Diese Methode verringert den internen Referenzzähler um eins. Fällt der Referenzzähler auf Null, wird das Objekt gelöscht.

Jede Schnittstelle hat eine eindeutige Schnittstellen-ID. Diese ID ist eine eindeutige 128 Bit Nummer (GUID). Eine Schnittstellen-ID wird immer in der HPP-Datei der Schnittstelle definiert. Damit jede ID eindeutig bleibt, wird empfohlen die ID vom System generieren zu lassen.

Wichtig:
Die Schnittstellen Identifikationsnummern müssen eindeutig sein, damit das Casting auch richtig und zuverlässig funktioniert.

Die Rückgabewerte, wie zum Beispiel `RET_NOERROR`, sind in der Datei `./zeusbase/Config/RetvalDefines.hpp` definiert.

Das folgende Beispiel zeigt ein Casting von einer Basisschnittstelle zu einer spezialisierten Schnittstelle.

Casting mit Schnittstellen

```
IXRootObject* pRoot = NULL;
if (pObject->askForInterface(INTERFACE_IXRootObject, ICAST(pRoot)) ==
RET_NOERROR)
{
  ...
  pRoot->release();
}
```

Tabelle 4: Beim erfolgreichen Casting wird `RET_NOERROR` zurückgegeben. Der interne Referenzzähler wird automatisch erhöht, deshalb muss danach ein `release()` erfolgen.

3.2.1.2 Die Basisklasse TZObject

Die Klassen der *shared objects* sind von `TZObject` abgeleitet. Diese Klasse implementiert die Methoden der Basisschnittstelle `IZUnknown`.

Will eine solche Klasse weitere Schnittstellen implementieren, geschieht dies in C++ durch die Mehrfachvererbung. Durch vorgefertigte Makros wird die Speicherverwaltung automatisch in die Klasse eingefügt. Die Makros sind in der Datei `./zeusbase/System/Interfaces/IZUnknownImplHelper.hpp` definiert.

API	`MEMORY_MANAGER_DECL;`
	Dieses Makro deklariert die Speicherverwaltung im Header der Klassendefinition.

API	`MEMORY_MANAGER_IMPL(classid);`
	Dieses Makro startet die Implementation der Speicherverwaltung. Es gibt eine Variante mit Locks, um eine thread-sichere Speicherverwaltung zu implementieren.

API	`INTERFACE_CAST(interface_type, interface_id);`
	Dieses Makro definiert ein speziellen Cast. Es muss der Typ der Schnittstelle angegeben werden und die eindeutige Nummer der Schnittstelle (ID).

API	`MEMORY_MANAGER_IMPL_END;`
	Beendet die Implementation der Speicherverwaltung. Es gibt die Variante der Delegation aller Aufrufe an eine spezielle Vaterklasse. Die Vaterklasse `TZObject` wird standardmässig gebraucht.

Folgende Bespiele illustrieren das Definieren einer solchen Klasse:

Klassendefinition in Zeus

```
#include <zeusbase/System/ZObject.h>
#include <ITest.hpp>

class TTest : public TZObject, public ITest
{
  public:
    TTest();

    //Methods of ITest
    virtual Retval MQUALIFIER getObject(IString* pName,
                                        IXObject*& rpObject);
    ...
    //Methods of IZUnknown
    MEMORY_MANAGER_DECL

  protected:
    virtual ~TTest();
    ...
  private:
    ...
};
```

Tabelle 5: Deklaration des Memory Managements. Bei jeder Implementation von Schnittstellen muss das Memory Management definiert und implementiert werden.

Implementation einer Klasse

```
TTest::TTest() : public TZObject()
{...}

...

//Implementation der Speicherverwaltung
MEMORY_MANAGER_IMPL(TTest);
  INTERFACE_CAST(ITest, INTERFACE_ITest);
MEMORY_MANAGER_IMPL_END;
```

Tabelle 6: Bei der Implementation der Speicherverwaltung können Makros verwendet werden.

Es gibt zusätzlich folgende Makros für Inline-Implementationen. Bei diesen Makros entfällt die Deklaration mittels `MEMORY_MANAGER_DECL`.

API	`MEMORY_MANAGER_INLINEIMPL();`
	Dieses Makro startet die Inline-Implementation der Speicherverwaltung.

Inline Implementation

```
#include <zeusbase/System/ZObject.h>
#include <ITest.hpp>

class TTest : public TZObject, public ITest
{
  public:
    TTest();

    //Methods of ITest
    virtual Retval MQUALIFIER getObject(const IString& rName,
                                        IXObject*& rpObject);
    ...
    //Methods of IZUnknown
    MEMORY_MANAGER_INLINEIMPL();
      INTERFACE_CAST(ITest, INTERFACE_ITest);
    MEMORY_MANAGER_IMPL_END;

  protected:
    virtual ~TTest();
    ...
  private:
    ...
};
```

Tabelle 7: Inline Implementation des Memory Managements.

3.2.2 Wert-Typen

Zu den Wert-Typen (*value types*) gehören die Klassen, die

- nicht in die Kategorie der *shared objects* gehören (Siehe vorhergehendes Kapitel).
- als Schnittstellenparameter verwendet werden sollen
- keine primitiven Datentypen sind

3.2.2.1 Die Basisschnittstelle IValueType

Diese Schnittstelle ist die Basisschnittstelle aller Wert Typen. Sie dient vor allem für Datentypen wie Zeichenketten, Listen, Maps, Sets, etc. Die konkreten Klassen dieser Datentypen sind nicht von `TZObject` abgeleitet. Meistens werden sie auf dem Stack erzeugt, sie können aber wie andere C++ Klassen mit `new` erzeugt und mit `delete` wieder freigegeben werden.

3.2.3 Übersicht

Folgende Tabelle gibt einen kurzen Überblick über die value types und shared object-Klassen:

API `Vergleichstabelle`

Klasse	Value	Shared
Primitive Datentypen	X	
Hilfsklassen für primitive Datentypen, wie TByte, etc	X	
TString	X	
Listen wie TSingleLinkedList, etc	X	
Maps wie TMap	X	
Sets wie TSet	X	
TFile, TDirectory	X	
TCalendar	X	
Streams wie TFileInputStream		X
TVariant		X
alle anderen Zeus-Framework Klassen		X

3.2.4 Programmierung

Dieses Kapitel soll kurz illustrieren, wie *shared objects* und wie *value types* verwendet werden.

3.2.4.1 Programmieren mit *shared objects*

Bei dieser Illustration wurde bewusst auf Auto-Pointer verzichtet, um zu zeigen, wie mit dem Referenzzähler gearbeitet wird.

<table>
<tr><th></th><th>shared object-Klassen</th></tr>
<tr><td>Rückgabe</td><td>Der Aufrufer deklariert einen Zeiger mit NULL. Der der Aufgerufene weist dem Zeiger die Adresse des Objekts zu und erhöht den Referenzzähler.

Aufrufer:

<pre>IMyObject* pObject = NULL;
if (rProvider.getObject(pObject) == RET_NOERROR)
{
 //do some thing with pObject...
 pObject->release();
}</pre>
Aufgerufener:

<pre>Retval TProvider::getObject(IMyObject*& rpObject)
{
  Retval retValue = RET_REQUEST_FAILED;
  if (m_pObject != NULL)
  {
    rpObject = m_pObject;
    rpObject->addRef();
    retValue = RET_NOERROR;
  }
  return retValue;
}</pre></td></tr>
<tr><td>Übergabe</td><td>Bei Muss-Parameter wird eine Referenz verlangt. Der Aufrufer ist gezwungen ein gültiges Objekt, oder eine gültige Schnittstelle zu übergeben.

Aufrufer:

<pre>if (pMyObject != NULL)
{
 rProvider.setObject(*pMyObject);
}</pre>
Aufgerufener:

<pre>void TProvider::setObject(const IMyObject& rValue)
{
 //Release old instance first ...
 m_pObject = &rValue;
 m_pObject->addRef();
}</pre>
Bei optionalen Parameter kann auch ein Zeiger des Objekts, oder der Schnittstelle übergeben werden.

<pre>void TProvider::setObject(const IMyObject* pValue)
{
  //Release old instance first ...
  if (pValue != NULL)
  {
    m_pObject = pValue;
    m_pObject->addRef();
  }
}</pre></td></tr>
</table>

Durch die Verwendung der Auto-Pointer-Klasse `TAutoPtr<T>` können Zeiger der *shared objects* ihre Gültigkeit auch auf einen Scope beschränken.

Verwendung von TAutoPtr<T>

```
//-------------------------------------------------------
//Im Header
#include <zeusbase/System/FileInputStream.h>

int main()
{
  TAutoPtr<TFileInputStream> Stream = new T
    FileInputStream(L"test.xml");

  if (!Stream.available())
  {
    return 1;
  }
  else
  {
    ...
  }

  //close must not be called, because destroying the Stream object
  // will do that, leaving the scope

  return 0;
}
```

Tabelle 8: Verwendung der TAutoPtr<T> Klasse mit shared objects. Die Auto-Pointer Klasse verwaltet den Zeiger. Dadurch wird der Gültigkeitsbereich auf einen Scope reduziert.

3.2.4.2 Programmieren mit *value types*

Folgende Illustration zeigt die Verwendung von Wert-Typen im Zusammenhang mit Schnittstellen-Methoden. Anders als bei *shared objects* wird der Inhalt des Typs kopiert.

Beim Übergeben und Speichern von Wert-Typen-Zeiger muss der C++ Programmierer selber sicherstellen, dass der Zeiger gültig bleibt.

	Value Types

Rückgabe	Der Wert-Typ wird vom Aufrufer erzeugt, und beim Aufgerufenen mit dem Inhalt abgefüllt. Der Inhalt wird kopiert.
	Aufrufer:
	`TString strData;`
	`rProvider.getString(`**`strData`**`);`
	Aufgerufener:
	`void MQUALIFIER TProvider::getString(`**`IString& rValue`**`)`
	`{`
	`  rValue.assign(L"Rückgabe");`
	`}`

Übergabe	Der Wert-Typ wird vom Aufrufer erzeugt. Der Aufgerufene kopiert den Inhalt in ein eigenes Objekt.
	Aufrufer:
	`TString strData = L"Test String";`
	`rProvider.setString(`**`strData`**`);`
	Aufgerufener:
	`void MQUALIFIER TProvider::setString(`**`const IString& rValue`**`)`
	`{`
	`  m_strValue = rValue;`
	`}`

Wert-Typen können auch mit `new()` erstellt werden. Anders als bei den *shared objects*, muss der Speicher mit dem Operator `delete()` freigegeben werden. Dabei muss der Programmierer selber die Zeiger verwalten und sicherstellen, dass nicht auf freigegebene Zeiger zugegriffen wird. Eine kleine Hilfe bietet die Klasse `TPtr`, welche einen Zeiger verwaltet.

Verwendung der Klasse TPtr<T>

```
{
  TPtr<TString> ptrString = new TString(L"Test");

...

} //string will be deleted here
```

Tabelle 9: Die Klasse TPtr hilft beim Verwalten eines C++ Zeigers.

3.3 Framework Einstellungen

Das Zeus-Framework ist portabel auf verschiedene Plattformen, wie Linux und Windows. Zudem kann es zusammen mit anderen Plattformen wie Borland C++Builder oder QT gekoppelt werden. Dieses Kapitel befasst sich mit den verschiedenen Möglichkeiten.

3.3.1 Error Level

Der Error Level definiert die Rückgabe von Fehler. Nicht jede Methode kann ihre Fehler direkt als `unsigned long` Rückgabewert zurückgeben. Vielmehr möchte man vielleicht ein `double` oder eine Zeichenkette direkt zurückgeben, damit dieser weiter verwendet werden kann:

```
Logger(L"Test").printfln(LOGMODE_ERROR, "Data=%s", strData.c_str());
```

ErrorLevel 0: Die Rückgabe der Fehler wird nicht erzwungen. Im Zeus-Framework ist standardmässig Error Level auf 0 (Null).

ErrorLevel 1: Die Rückgabe der Fehler wird erzwungen. Der Aufrufer § muss ein Zeiger auf eine Variable der Methode übergeben, damit der Fehler zurückgegeben werden kann. Der Aufrufer kann aber auch NULL übergeben, wenn er am Rückgabewert nicht interessiert ist.

Verwendung ErrorLevel 0

```
#include <zeusbase/System/LoggerManager.h>
#include <zeusbase/System/String.h>
#include <stdlib.h>

int main()
{
  ...

  //Ohne Fehlerrückgabe
  TString strText(L"Any text goes here");
  Logger(L"Test").printfln(LOGMODE_INFO, "This is a test [%s]",
    strText.c_str());

  //Mit Fehlerrückgabe
  TString strText2(L"Any other text goes here");
  bool bError = false;
  Logger(L"Test").printfln(LOGMODE_INFO, "This is a test [%s]",
    strText.c_str(&bError));
  if ( bError )
  {
    Logger(L"Test").printfln(LOGMODE_INFO,
      "Error converting string");
  }
  return 0;
}
```

Tabelle 10: Verwendung mit Error Level 0

Verwendung ErrorLevel 1

```
#ifdef ZEUS_ERRORLEVEL
  #undef ZEUS_ERRORLEVEL
  #define ZEUS_ERRORLEVEL=1
#endif

#include <zeusbase/System/LoggerManager.h>
#include <zeusbase/System/String.h>
#include <stdlib.h>


int main()
{
  ...

  //Ohne Fehlerrückgabe
  TString strText(L"Any text goes here");
  Logger(L"Test").printfln(LOGMODE_INFO, "This is a test [%s]",
    strText.c_str(NULL));

  //Mit Fehlerrückgabe
  TString strText2(L"Any other text goes here");
  bool bError = false;
  Logger(L"Test").printfln(LOGMODE_INFO, "This is a test [%s]",
    strText.c_str(&bError));
  if ( bError )
  {
    Logger(L"Test").printfln(LOGMODE_INFO,
      "Error converting string");
  }
  return 0;
}
```

Tabelle 11: Verwendung mit Error Level 1

3.3.2 Exportierte Klassen (zeusbase_class)

Beim Kompilieren des Zeus-Frameworks muss darauf geachtet werden, dass die Definition `ZEUSBASE_EXPORT` in der Projektdatei oder der Make-Datei enthalten ist. Mittels dieser Definition wird der Export der Symbole gesteuert.

Alle Bibliotheken und Anwendungen, welche das Framework verwenden, sollten die Definition `ZEUSBASE_EXPORT` nicht gesetzt haben.

Soll eine Klasse exportiert werden, muss anstelle der `class`-Anweisung eine `zeusbase_class`-Anweisung stehen. Beim Kompilieren des Zeus-Frameworks werden alle Klassensymbole exportiert. Beim Verwenden der Klassen in anderen Bibliotheken und Anwendungen werden die Klassen hiermit importiert.

Verwendung zeusbase_class

```
//Nur für Klassen des Zeus-Frameworks verwenden
zeusbase_class TCell
 : public TXObject,
   public ICellComm,
   public ICell
{
  public:
    TCell(IXMLNode& rNode);
  ...
};
```

Tabelle 12: Verwendung von zeusbase_class für weitere Zeus-Framework Klassen.

3.3.3 Namespace Zeus

Das Zeus-Framework wurde in einem Namensraum entwickelt. Damit der Namensraum auch geändert werden kann (durch Versionierung), sind für dessen Gebrauch Makros definiert.

API NAMESPACE_Zeus

Dieses Makro definiert den Namen des Namensraums. Er kann direkt als solcher im Code eingesetzt werden

Beispiel: **`void getText(NAMESPACE_Zeus::TString& rText) const;`**

API BEGIN_NAMESPACE_Zeus

In den Header-Dateien wird hiermit der Namensraum für eine Klasse geöffnet. Das ist gleichbedeutend wie **`namespace NAMESPACE_Zeus {`**

API END_NAMESPACE_Zeus

In den Header-Dateien wird hiermit der Namensraum für eine Klasse beendet.

API USING_NAMESPACE_Zeus

In der Implementation kann vor der Verwendung von Zeus-Klassen dieses Makro eingesetzt werden, damit bei der Deklaration von Variablen die Angabe des Namensraum entfällt.

Verwendung Namensraum Zeus

```
//-------------------------------------------------------
//Im Header
#include <zeusbase/System/XObject.h>

BEGIN_NAMESPACE_Zeus

class TTest : public TXObject
{
  public:
    TTest(IXMLNode& rNode);
  ...
};

END_NAMESPACE_Zeus



//-------------------------------------------------------
//In der Implementation
#include <Test.h>

USING_NAMESPACE_Zeus

TTest:: TTest(IXMLNode& rNode) : TXObject(rNode)
{}
```

Tabelle 13: Verwendung vom Zeus Namensraum. Auch aussen stehende Klassen können in diesem Namensraum entwickelt werden.

3.4 Datentypen des Zeus-Frameworks

Das Framework bietet verschiedenste Datentypen an, wie zum Bsp. Zeichenketten und Listen. Dieses Kapitel zeigt deren Verwendung und Spezialitäten.

3.4.1 Primitive Datentypen

Folgende primitive Datentypen werden im Zeus-Framework verwendet und vollständig unterstützt:

API `zeusbase/Config/PlatformDefines.hpp`

Zeus-Framework	**C++**	**Beschreibung**
`Int8`	char	Ganzzahliger Wert mit Vorzeichen (8 Bit)
`Uint8`	unsigned char	Ganzzahliger Wert ohne Vorzeichen (8 Bit)
`Int16`	short	Ganzzahliger Wert mit Vorzeichen (16 Bit)
`Uint16`	unsigned short	Ganzzahliger Wert ohne Vorzeichen (16 Bit)
`Int32`	long	Ganzzahliger Wert mit Vorzeichen (32 Bit)
`Uint32`	unsigned long	Ganzzahliger Wert ohne Vorzeichen (32 Bit)
`Int64`	Plattform abhängig	Ganzzahliger Wert mit Vorzeichen (64 Bit)
`Uint64`	Plattform abhängig	Ganzzahliger Wert ohne Vorzeichen (64 Bit)
`Float32`	float	Reeller Zahlenwert (32 Bit)
`Float64`	double	Reeller Zahlenwert (64 Bit)
`Float80`	Plattform abhängig	Reeller Zahlenwert (80 Bit)
--	`wchar_t`	Wide Character für UNICODE

Diese Standardtypen werden durch Datentypen erweitert, welche sich der Prozessorarchitektur anpassen. Dies sind insbesondere Zahlentypen für ganze und reelle Zahlen.

API `zeusbase/Config/PlatformDefines.hpp`

`Int`	Ganzzahliger Wert mit Vorzeichen. Je nach Einstellung 32 oder 64 Bit. **Vorsicht: Wird nicht auf int gesetzt.**
`Uint`	Ganzzahliger Wert ohne Vorzeichen. Je nach Einstellung 32 oder 64 Bit **Vorsicht: Wird nicht auf unsigned int gesetzt.**
`Float`	Reelle Zahl. Je nach Einstellung 32 oder 64 Bit

Die Datentypen `int`, `unsigned int` sollten nicht in Schnittstellen verwendet werden, da ihre Grösse je nach Plattform und Compiler unterschiedlich ist (32bit oder 64bit). Bei

Remote-Anwendungen kann es so zu Überläufen oder Fehlinterpretationen kommen. Intern können diese Datentypen jedoch ohne weiteres verwendet werden.

Weiter sind logische Datentypen sind definiert. Sie dienen der Lesbarkeit von Programmcode und sollten sich unabhängig der oben genannten Datentypen verhalten:

API	`zeusbase/Config/PlatformDefines.hpp`	
	`Retval`	Dieser Datentyp kapselt die Fehlerrückgaben der Methoden und Funktionen. Standardmässig ist dieser auf Uint gesetzt, kann aber je nach Anwendung auch anders definiert werden.
	`Timeval`	Dieser Datentyp kapselt eine Zeitangabe (in Sekunden)
	`InterfaceID`	Die Interface ID ist normalerweise als 128bit Wert definiert (TypGUID)
	`TypGUID`	Global unique identifier ist ein 128bit Wert

3.4.2 Hilfsklassen für primitive Datentypen

Für gewisse primitive Datentypen existieren Hilfsklassen für erweiterten Einsatz.

API		
	`TByte`	Diese Klasse nimmt ein Byte (char) auf. Sie kann zum Beispiel das Byte in Hex-Format ausgeben (Zeichenkette) `zeusbase/System/Byte.h`
	`TCharacter`	Diese Klasse stellt verschiedene statische Methoden zur Verfügung, um ein Zeichen zu klassifizieren. Die Klasse ist sehr praktisch für Scanner-Bau `zeusbase/System/Character.h`
	`TFloat`	Diese Klasse nimmt einen Float-Wert auf. Verschiedene Methoden wie Vergleichen von Float-Werten sind implementiert (Nicht unproblematisch, wenn Float-Variablen direkt verglichen werden). `zeusbase/System/Float.h`
	`TInt`	Kapselt einen einen Integerwert `zeusbase/System/Int.h`
	`TGUIDWrapper`	Kapselt einen TypGUID und stellt Methoden zum Vergleichen und Darstellen zur Verfügung `zeusbase/System/GUIDWrapper.h`

3.4.2.1 TByte Klasse

Die `TByte`-Klasse dient primär der Darstellung von Bytes als Zeichenkette. Die Methode `convertToBinary()` wandelt ein Byte in eine Zeichenkette um, welcher das Byte binär darstellt.

Verwendung von TByte

```
//-------------------------------------------------------
//Im Header
#include <zeusbase/System/Byte.h>
#include <stdio.h>

int main()
{
  TByte ByteA(88);
  TString strByteA = ByteA.convertToBinary();
  printf("Byte %d is in binary %s",
    ByteA.getValue(),
    strByteA.c_str());

  return 0;
}
```

Tabelle 14: Verwendung der TByte Klasse.

3.4.2.2 TCharacter Klasse

Die `TCharacter` Klasse dient der Klassifikation von Zeichen. Dies ist sehr praktisch, um effiziente Scanner zu bauen. Die Methoden sind vorwiegend statisch, um das Erzeugen von Objekten zu vermeiden. Folgende statische Methoden sind von Interesse:

API	`zeusbase/System/Character.h`
	`static bool isAlpha(char cValue);` Prüft, ob das Zeichen ein Buchstabe `[a..zA..Z]` ist.
	`static bool isDigit(char cValue);` Prüft, ob das Zeichen eine Ziffer ist `[0..9]`.
	`static bool isHexDigit(char cValue);` Prüft, ob das Zeichen eine hexadezimale Ziffer ist `[0..9A..F]`.
	`static bool isAlphaNum(char cValue);` Prüft, ob das Zeichen eine Ziffer oder ein Buchstabe ist.

API	zeusbase/System/Character.h
	`static bool isWhiteSpace(char cValue);` Prüft, ob das Zeichen ein Leerzeichen, vertikaler oder horizontaler Tabulator, neue Zeile ist.

3.4.2.3 TFloat-Klasse

In C++ gibt es gewisse Einschränkungen, bzw. Probleme mit dem Umgang von `double`-Werten. Die `TFloat`-Klasse implementiert Methoden zum verbesserten Zugriff auf `double`-Werte.

Wichtige Methoden sind:

API	zeusbase/System/Float.h
	`bool equals(Float fValue, Float fPrecision) const;` Diese Methode vergleicht zwei `Float`-Werte miteinander. In C++ ist der direkte Vergleich mit `==operator()` problematisch. Es werden auch die Rundungsfehler auf den letzten Kommastellen berücksichtigt. Mit equals() Methode kann die gewünschte Präzision angegeben werden.
	`Float round() const;` Rundet einen Float auf einen ganzzahligen Wert. Existiert auch als Integer Variante.
	`Float roundEx(Float fPrecision) const;` Rundet ein Float auf n Kommastellen genau. Zum Beispiel mit f`Precision=0.01` auf einen Hundertstel, mit f`Precision=0.001` auf einen Tausendstel.
	`Float ceil() const;` Rundet ein Float auf die nächst tiefere ganze Zahl. Existiert auch als Integer Variante.
	`Float floor() const;` Rundet ein Float auf die nächst höhere ganze Zahl. Existiert auch als Integer Variante.
	`TString format(Float fRoundPrecision = 0, Uint uiTailingZeros = 0, Uint uiLeadingZeros = 1) const;` Formatiert einen Float als String. Mit LeadingZeros and TailingZeros kann die Darstellung der Nullen gesteuert werden
	`Float getFactorial() const;` Gibt die Fakultät der Zahl zurück.
	`bool isNaN() const;` Prüft ob der Float keine Nummer ist

API	zeusbase/System/Float.h

```
bool isInfinity() const;
bool isPosInfinity() const;
bool isNegInfinity() const;
```

Prüft, ob der Wert unendlich ist (negativ oder positiv)

Wichtig: Vergleichen von `float`-Variablen sollte NIE direkt mit dem `==operator()` geschehen. C++ vergleicht die Daten Byte um Byte. Bei Float-Berechnungen kann es zu Rundungsfehler kommen. Obwohl mathematisch gesehen der Vergleich stimmt, kann es sein, dass auf dem Computer dieser Vergleich fehlschlägt. Deshalb ist es besser die `TFloat`-Klasse zu verwenden. Die `float`-Werte werden miteinander verglichen indem die Differenz ausgerechnet wird. Die Differenz wird mit der angegeben Präzision verglichen. Ist die Differenz grösser, so sind die Float-Werte unterschiedlich, sonst sind die Werte gleich.

Verwendung von TFloat

```
//-------------------------------------------------------
//Im Header
#include <zeusbase/System/Float.h>
#include <stdio.h>

int main()
{
  TFloat FloatA = 0.0234212;

  //True
  if (FloatA .equals(0.234, 0.001))
  {
    printf("Float %f auf 2 Kommastellen gerundet= %f",
      *FloatA,
      FloatA.roundEx(0.01));
  }
  return 0;
}
```

Tabelle 15: Verwendung der TFloat Klasse.

Beim Runden von Zahlen auf die n-te Kommastelle wird folgendermassen vorgegangen:

1. Ist die Zahl auf der Kommastelle n+1 grösser oder gleich 5 wird aufgerundet
2. Ist die Zahl auf der Kommastelle n+1 kleiner als 5 wird abgerundet.

3.4.2.4 TGUIDWrapper Klasse

Die GUID-Wrapper Klasse dient der Kapselung einer GUID (128bit Wert). Dabei werden

plattformabhängige API Funktionen verwendet, um GUID zu erstellen.

API	`zeusbase/System/GUIWrapper.h`
	`static TString toString(const InterfaceID& rID);` Eine InterfaceID als String formatieren.
	`static bool isEqual(const TypGUID& rID1, const TypGUID& rID2);` Prüft, ob zwei GUID gleich sind
	`static TGUIDWrapper createGUID();` Erstellt eine neue GUID mit Hilfe der API Funktionen

	`static bool isEqualInterface(const InterfaceID& rID1,`
	`const InterfaceID& rID2);`
	
	Prüft, ob zwei Interface'IDs gleich sind.

	`static bool isIZUnknown(const InterfaceID& rID1);` Prüft ob die InterfaceID derjenigen von IZUnknown entspricht.

3.4.3 Zeichenketten Datentyp

Das Zeus-Framework besitzt eine UNICODE fähige Zeichenkette. Dieser nimmt Konvertierungen von char-Arrays, long-Werten und double-Werten vor.

API	
`TString`	Unicode fähige Zeichenkette, welche die Schnittstelle `IString` implementiert. `zeusbase/System/String.h`

Der `TString` ist eine vollwertige Zeichenketten-Implementation welche die Schnittstelle `IString` implementiert. Dabei handelt es sich immer um eine Zeichenkette mit 16bit Zeichen (`wchar_t`). Die Zeichenkette ist immer NULL-terminiert, dass heisst, das letzte Zeichen ist immer 0x0000.

Wichtige Methoden der Schnittstelle `IString` sind:

API	`zeusbase/System/Interfaces/IString.hpp`
	`wchar_t* MQUALIFIER c_bstr() const;` Diese Methode gibt das `wchar_t`-Array zurück.

API	zeusbase/System/Interfaces/IString.hpp

```
char* MQUALIFIER c_str(BOOL_ERRORRETVAL(pError)) const;
```

Diese Methode konvertiert die Zeichenkette in ein char-Array. Diese Methode gibt einen Fehler zurück, wenn eine Unicode Zeichenkette konvertiert wird und Daten verloren gehen (Siehe Error Level).

```
Float MQUALIFIER toFloat(Float fPredef = 0.0) const;
```

Konvertiert eine Zeichenkette in einen Float-Wert

```
Int MQUALIFIER toInt(Int iPredef = 0) const;
```

Konvertiert eine Zeichenkette in einen Integer-Wert.

```
Uint MQUALIFIER toUint(Uint uiPredef = 0, bool* pbError = NULL) const;
```

Konvertiert eine Zeichenkette in einen unsigned int.

```
bool MQUALIFIER toBool(bool bPredef = false, bool* pbError = NULL) const;
```

Konvertiert eine Zeichenkette in einen boolschen Wert.

```
Timeval MQUALIFIER toTimeval(Timeval tmPredef = 0,
                             bool* pbError = NULL) const;
```

Konvertiert eine Zeichenkette in einen Zeitwert (64bit)

```
bool MQUALIFIER equalsStr(const IString* pInstr) const;
```

Vergleicht 2 Zeichenketten miteinander.

```
void MQUALIFIER concatStr(const IString* pInstr);
```

Fügt 2 Zeichenketten zusammen.

```
void MQUALIFIER assignStr(const IString* pInstr);
```

Ordnet dem Objekt eine neue Zeichenkette zu. Anders als in JAVA kann hier das `TString`-Objekt seinen Inhalt während der Laufzeit beliebig oft ändern.

```
Int MQUALIFIER getSize()const;
```

Gibt die Grösse der Zeichenkette zurück. Von der Grösse ist das letzte Zeichen 0x0000 ausgenommen.

```
wchar_t MQUALIFIER getChar(Int iIndex)const;
```

Gibt ein einzelnes Zeichen aus der Zeichenkette zurück.

```
Int MQUALIFIER getPos(const IString* pSubstring)const;
```

Diese Methode prüft, ob eine Zeichenkette in dem TString enthalten ist und gibt den Index zurück an dem diese Zeichenkette startet. Wird -1 zurückgeben, wurde die Zeichenkette nicht gefunden.

```
bool MQUALIFIER endsWithStr(const IString& rString)const;
```

Prüft, ob eine Zeichenkette mit gewissen Zeichen endet.

```
bool MQUALIFIER startsWithStr(const IString& rString)const;
```

Prüft, ob eine Zeichenkette mit gewissen Zeichen startet.

Weitete Methoden sind nur in der Implementation zugänglich:

API	`zeusbase/System/String.h`
	`TString getSubString(Int iStart, Int iEnd) const;` Diese Methode gibt eine Teilzeichenkette zurück, die an Position `[iStart]` startet und an der Position `[iEnd]` endet.
	`TString deleteSubString(Int iStart, Int iEnd) const;` Löscht eine Teilzeichenkette aus der Originalzeichenkette und gibt das Resultat als neue Zeichenkette zurück. Das Original wird nicht verändert.

	`TString remove(const IString& rToRemove,`
	`Int* pReplaces = NULL) const;`
	
	Löscht alle Teilzeichenketten und gibt das Resultat zurück.

	`TString removeFirst(const TString& rToRemove) const;` Löscht nur die erste Teilzeichenkette und gibt das Resultat zurück.

	`TString replace(const IString& rToRemove, const IString& rReplace,`
	`Int* pReplaces = NULL) const;`
	
	Ersetzt alle Vorkommnisse von rToRemove in der Zeichenkette und gibt das Resultat zurück.

	`TString replaceFirst(const IString& rToRemove,`
	`const IString& rReplace) const;`
	
	Ersetzt nur das erste Vorkommnis rToRemove in der Zeichenkette.

	`TString toUpperCase() const;` Konvertiert den Inhalt in Grossbuchstaben.
	`TString toLowerCase() const;` Konvertiert den Inhalt in Kleinbuchstaben.
	`TString trim() const;` Entfernt alle White-Spaces (Leerzeichen, \n, \t) am Anfang und Ende der Zeichenkette. Mit `trimLeft()` oder `trimRight()` kann gezielt nur jeden des Anfangs, bzw. Endes entfernt werden.

Die TString-Klasse unterstützt die Zeichenketten-Klassen `std::string` oder `std::wstring` der Standard-Bibliothek. Die Unterstützung kann mit der Projektdefinition `USE_STL_BINDINGS` eingeschaltet werden.

Weitere Zeichenketten werden im so genannten Transcoder unterstützt.

API	
`TSTLTranscoder`	Transcoder für Zeichenketten aus der Standard-Bibliothek (stl) `zeusbase/System/Platforms/Transcoder.hpp`

API	
`TQTTranscoder`	Transcoder für Zeichenketten aus der QT Bibliothek `zeusbase/System/Platforms/Transcoder.hpp`
`TBCBTranscoder`	Transcoder für Zeichenketten aus der Borland C++ Builder Bibliothek (VCL) `zeusbase/System/Platforms/Transcoder.hpp`

Verwendung von TString

```
//-------------------------------------------------------
//Im Header
#include <zeusbase/System/String.h>
#include <stdio.h>

USING_NAMESPACE_Zeus

void main()
{

  TString strData(L"This is a string test:");
  strData += 0.234; //adds a Float32 value

  printf("Out: %s", strData.c_str(NULL));
}
```

Tabelle 16: Beispiel zur Verwendung von Zeichenketten. Zeichenketten zusammensetzen ist mit TString um ein vielfaches einfacher, als die Verwendung von sprintf zum Beispiel.

3.4.3.1 Ermitteln von Teilzeichenketten (SubStrings)

Zum ermitteln von Teilzeichenketten gibt es eine Fülle von Methoden und Varianten. Einige wichtige sind hier dokumentiert.

Die Basis-Methoden ist `getSubString()`. Der Startindex gibt das erste Zeichen an, von dem die Teilzeichenkette gelesen wird. Der Endindex ist optional.

- Wird kein Endindex angegeben, wird die Teilzeichenkette von Startindex bis ans Ende der Zeichenkette zurückgegeben.
- Wird ein Endindex angegeben, wird die Teilzeichenkette von Startindex bis Endindex zurückgegeben.

Die Methode `deleteSubString()` verhält sich analog.

Folgende Arten zum Ermitteln und Löschen von Teilzeichenketten stehen zur Verfügung:

API	`zeusbase/System/String.h`
	`TString getLeft(Int iCharCount) const;` Gibt eine Teilzeichenkette zurück, die links beginnt und `iCharCount`-Zeichen lang ist
	`TString getRight(Int iCharCount) const;` Gibt eine Teilzeichenkette zurück, die rechts beginnt und `iCharCount`-Zeichen lang ist.
	`TString deleteLeft(Int iCharCount) const;` Löscht eine Teilzeichenkette, die links beginnt und `iCharCount`-Zeichen lang ist.
	`TString deleteRight(Int iCharCount) const;` Löscht eine Teilzeichenkette, die rechts beginnt und `iCharCount`-Zeichen lang ist.

3.4.3.2 Transcode

Die Klasse `TString` stellt verschiedene Transcoding Methoden zur Verfügung, deren Hauptaufgabe es ist, einer Zeichenkette in binäre Form umzuwandeln und umgekehrt.

API	`zeusbase/System/String.h`
	`static TString transcode(const IByteArray& list);` Wandelt einen Byte Array in eine Zeichenkette um.
	`static TByteArray transcode(const TString& rValue, bool bAsWideChar);` Wandelt eine Zeichenkette in ein Byte Array um. Mit dem Flag `bAsWideChar` wird die Zeichenkette als Wide Character Array umgewandelt.

Beim Umwandeln einer Zeichenkette in ein Byte Array kann durch die Angabe `bAsWideChar` angegeben werden, dass ein Wide Character in einzelne Bytes umgewandelt und so ins Byte Array gespeichert wird. Wird das Flag nicht gesetzt können Daten verloren gehen.

Ein umgewandeltes Wide Array wird durch den Unicode Header im Byte Array gekennzeichnet (Erstes Zeichen = 0xFF, zweites Zeichen = 0xFE).

3.4.4 Listen und Arrays

Listen sind ein elementarer Bestandteil des Zeus-Frameworks. Es gibt sie in verschiedenen Ausprägungen, wie zum Beispiel Linked-Lists, Arrays oder Maps. Diese Klassen sind ausschliesslich Templates.

API	
`TSingleLinkedList`	Template-Klasse, welche Objekte in einer verketteten Liste verbindet. Implementiert die Schnittstelle `IList`. `zeusbase/System/SingleLinkedList.hpp`
`TArrayList`	Eine Template-Klasse, welche ein Array verwaltet. Implementiert die Schnittstelle `IList`. `zeusbase/System/ArrayList.hpp`
`TMap`	Eine Template-Klasse, welche eine Map (Binary Search Tree) implementiert. `zeusbase/System/Map.hpp`
`TStack`	Eine Template-Klasse, die einen Stack implementiert. `zeusbase/System/Stack.hpp`
`TQueue`	Eine Template-Klasse, die eine Queue implementiert. `zeusbase/System/Queue.hpp`
`TSet`	Eine Template-Klasse, die eine Menge implementiert `zeusbase/System/Set.hpp`

Es gibt verschiedene Spezialisierungen dieser Template-Klassen für Datentypen:

API	
`TByteArray`	Diese Klasse verwaltet ein char-Array und wird für Streaming-Anwendungen gebraucht. `zeusbase/System/ByteArray.hpp`
`TStringList`	Eine Liste für Zeichenketten vom Typ `TString`. Diese Liste implementiert die Schnittstelle `IList<IString*>`. `zeusbase/System/StringList.h`
`TStringMap`	Ein binärer Suchbaum, welcher als Schlüssel eine Zeichenkette akzeptiert. `zeusbase/System/StringMap.hpp`
`TManagedMap`	Map welche Objekte akzeptiert. Die Objektreferenzen werden von der Map direkt verwaltet.

API	
	`zeusbase/System/ManagedMap.hpp`
`TManagedList`	Liste welche-Objekte akzeptiert. Die Objektreferenzen werden von der Liste direkt verwaltet. `zeusbase/System/ManagedList.hpp`
`TManagedQueue`	Queue welche Objekte akzeptiert. Die Objektreferenzen werden von der Queue direkt verwaltet. `zeusbase/System/ManagedQueue.hpp`
`TManagedStack`	Stack welche Objekte akzeptiert. Die Objektreferenzen werden von dem Stack direkt verwaltet. `zeusbase/System/ManagedStack.hpp`

Listen, Maps etc., die zur Verwaltung von Objekten dienen, besitzen folgendes besonderes Verhalten:

1. Beim Hinzufügen von Objekten wird ein Zeiger oder eine Referenz aufgenommen. Soll zusätzlich ein `addRef()` ausgeführt werden, muss das Flag `bAllocPointer` auf `true` gesetzt werden.

2. Beim Entfernen von Objekten wird automatisch ein `release()` aufgerufen.

Verwendung von Objekt-Listen

```
//-------------------------------------------------------
//Im Header
#include <zeusbase/System/ZObjectList.hpp>
#include <stdio.h>

USING_NAMESPACE_Zeus

void main()
{
  TManagedList<TMyObject> lstObjects;

  //Hinzufügen von Objekten ohne addRef()
  lstObjects.add(new TMyObject());

  //Hinzufügen von Objekten. addRef() wird ausgeführt.
  TMyObject* pObject = new TMyObject();
  lstObjects.add(pObject, true);
  pObject->release();

  //Besser mit TAutoPtr<T>
  TAutoPtr<TMyObject> Object = new TMyObject();
  lstObjects.add(Object);


  lstObjects.clear(); //Alle Objekte werden automatisch gelöscht.
}
```

Tabelle 17: Die Managed-Listen sind in der Lage auf dem Heap allozierte Objekte zu verwalten.

3.4.4.1 Iteratoren

Um der Inhalt einer Liste zu lesen ist es sinnvoll einen Iterator zu verwenden. Im Zeus-Framework werden vorzugsweise normale Vorwärts-Iteratoren verwendet.

Pro Liste können mehrere Iteratoren erstellt und verwendet werden. Diese sind weitgehend von einander unabhängig. Beim Verändern des Listeninhalts, werden die Iteratoren ebenfalls aktualisiert.

Verwendung von Iteratoren

```
//-------------------------------------------------------
//Im Header
#include <zeusbase/System/SingleLinkedList.hpp>
#include <stdio.h>

USING_NAMESPACE_Zeus

void main()
{
  TSingleLinkedList<Int> lstNumbers;
  ...
  TIterator<long> It = lstNumbers.getIterator();
  while(It.hasNextItem())
  {
    foo(It.getNextItem());
  }
}
```

Tabelle 18: Beispiel eines Iterators. Mit der TIterator-Klasse werden die Instanzen automatisch verwaltet. Ein releaseIterator() auf der Liste braucht es in diesem Fall nicht.

Wichtig:
Mit getIterator() erzeugte Iteratoren müssen mit release() wieder freigegeben werden, wenn sie nicht mit TIterator oder äquivalenten .Smart-Pointer-Klassen verwaltet werden.

Speziell bei verknüpften Listen (wie TSingleLinkedList) führt die Verwendung der Iteratoren zu einer Verbesserung der Laufzeitgeschwindigkeit. Die Zeitkomplexität beim traversieren der Liste wird von $O(n^2)$ (mit Klammeroperator oder `getItem()`-Methode) auf O(n) reduziert.

3.4.5 Pair-Datentyp

Manchmal muss ein Daten-Paar verwendet werden. Damit nicht jedes mal eine Struktur definiert werden muss, bietet das Zeus-Framework den Datentypen `TPair<T1, T2>` an. Der Datentyp wurde als Template implementiert und besteht aus zwei Datentypen T1 und T2.

API	
`IPair<T1, T2>`	Schnittstelle eines Wertepaares `zeusbase/System/Interfaces/IPair.hpp`
`TPair<T1, T2>`	Implementation des Wertepaares als Template. `zeusbase/System/Pair.hpp`

API	
`IProperty`	Schnittstelle für ein Property Wertepaar aus 2 Strings `zeusbase/System/Interfaces/IPropety.hpp`
`TProperty`	Spezielle Implementation für ein Wertepaar aus 2 Strings `zeusbase/System/Propety.h`

3.4.6 Kalender Typ

Das Zeus-Framework bietet eine Kalender Klasse, mit welcher Zeit und Datum gelesen und formatiert werden kann.

Der Kalender Typ ist ein Wert-Typ.

API	
`TCalendar`	Klasse für Kalenderfunktionen `zeusbase/System/Calendar.h`

Folgende Methoden stehen zur Verfügung:

API `zeusbase/System/Calendar.h`

```
Uint getSecond() const;
Uint getMinute() const;
Uint getHour() const;
Uint getDayOfWeek() const;
Uint getDayOfMonth() const;
Uint getDayOfYear() const;
Uint getMonth() const;
Uint getYear() const;
```

Ermitteln der einzelnen Zeitkomponenten.

```
TString formatDateTime() const;
TString formatDateTime(TLocale& rLocale) const;
TString formatDate(const TString& rDelimiter) const;
TString formatTime(const TString& rDelimiter) const;
TString formatDateTime(const IString& rFormat,
                       Int iExpectedLength = 200) const;
```

Formatiert die Zeitangabe in eine Zeichenkette.

```
Timeval getTimeValue();
```

Ermitteln des Zeitwerts.

API	zeusbase/System/Calendar.h

	`static TString formatDateTimeForLogger();`
	`static TString formatDateTimeForLogger(const TCalendar& rCal);`
	
	Formatiert die Zeitangabe für Logger im Format `YYYY-MM-DD hh:mm:ss`

3.4.7 Zeiger Typen

Das Zeus-Framework bietet verschiedene Zeigertypen an, die C++ Zeiger verwalten können.

API	
`TAutoPtr<T>`	Auto-Zeiger zum Verwalten von Schnittstellen und Shared Objects. Die Klasse oder Schnittstelle T muss addRef() und release() implementieren. `zeusbase/System/AutoPtr.hpp`
`TPtr<T>`	Auto-Zeiger zum Verwalten eines beliebigen C++ Objekts, welches mit new alloziert wurde und mit delete freigegeben wird.
`TArrayPtr<T>`	Auto-Zeiger zum Verwalten von

3.4.7.1 Klasse TAutoPtr

Die Auto-Pointer Klasse TAutoPtr verwaltet Shared-Objects (Siehe Kapitel zu Objekt-Typen). Die Klasse des verwalteten Zeigers muss die Methoden `release()` und `addRef()` implementieren und vorzugsweise von die Schnittstelle IZUnknown implementieren.

API	zeusbase/System/AutoPtr.hpp
	`void assign(const T* pInterface);` Zuweisen eines Zeigers. Der Referenzzähler wird erhöht.
	`void attach(const T* pInterface);` Zuweisen eines Zeigers. Der Zeiger wird adoptiert und der Referenzzähler wird **nicht** erhöht.
	`void release();` Gibt den Zeiger frei. Der Referenzzähler wird verringert.

API	zeusbase/System/AutoPtr.hpp
	`void detach();` Der Zeiger wird entkoppelt, das heisst der Inhalt des Auto-Pointers wird leer, ohne dass der Referenzzähler verringert wird
	`IZUnknown*& getInterfaceReference();` Gibt den Inhalt des Auto-Pointers als IZUnknown*& zurück. Diese Methode wird vor allem für askForInterface() verwendet.
	`T*& getPointerReference();` Gibt den Inhalt des Auto-Pointer zum Zuweisen zurück.
	`T* getPointer();` Gibt den Pointer zurück.

Verwendung von TAutoPtr

```
//-------------------------------------------------------
//Im Header
#include <zeusbase/System/AutoPtr.hpp>


void foo(IUnknown& rPtr)
{
  TAutoPtr<ITestIface> ptrTest;
  ...
  if (rPtr.askForInterface(INTERFACE_ITestIface,
                           ptrTest.getInterfaceReference()) == RET_NOERROR)
  {
    ...
  }
}
```

Tabelle 19: Verwendung von askForInterface() mit Auto-Pointer.

3.4.7.2 Klasse TArrayPtr

Die Klasse `TArrayPtr` verwaltet Objekte die mit dem new[] Operator erzeugt wurden. Anders als die `TArrayList<T>` Klasse dient diese Klasse rein der Verwaltung des Zeigers und nicht den Inhalt.

Die Klasse besitzt einen Referenzzähler, welcher beim Kopieren von des Arrays erhöht und beim Freigeben wieder verringert wird.

API	zeusbase/System/ArrayPtr.hpp
	`void assign(const T* pInterface, Int iSize = -1);`

API	`zeusbase/System/ArrayPtr.hpp`
	Zuweisen eines Array-Zeigers. Die Grösse des Arrays kann optionell angeben werden.
	`void clean();` Inhalt des Arrays mit NULL-Zeichen initialisieren.
	`Int getSize() const;` Gibt die optionelle Grösse des Arrays zurück.
	`void release();` Verringert den Referenzzähler und gibt das Array frei.

3.4.8 Variant-Datentyp

Der Variant Datentyp dient zur Kapselung von verschieden Datentypen in einem gemeinsamen Container.

API	
`TZVariant`	Ein Variant zum Kapseln von anderen Datentypen und Objekten. `zeusbase/System/ZVariant.h`

Der Variant kann folgende Datentypen aufnehmen:

- Int-Werte (Int8, Int16, Int32, Int64)
- Float-Werte (Float32, Float64)
- `TString`-Objekte
- `TByteArray`-Objekte
- Objekte, welche die Schnittstelle `ISerializable` implementieren.
- Eine Liste von Objekten, welche die Schnittstelle `ISerializable` implementieren.
- Ein Variant kann auch leer sein

Dieser Datentyp ist besonders geeignet, um primitive Datentypen und Objekte zu serialisieren. Mehr dazu siehe Serialisierung von Objekten.

3.4.9 Streams

Die Streaming-Klassen diesen zum Lesen oder Schreiben von Daten. Es gibt zwei Kategorien von Streams, die Input-Streams und die Output-Streams.

3.4.9.1 Input-Stream

Die Input-Streams dienen dem Lesen von Daten. Der Ursprung der Daten ist je nach Input-Stream-Klasse verschieden.

API	
`TByteArrayInputStream`	Ein Byte-Array dient als Ursprung der Daten `zeusbase/System/ByteArrayInputStream.h`
`TFileInputStream`	Eine Datei dient als Ursprung der Daten `zeusbase/System/FileInputStream.h`

Diese Klassen implementieren die Schnittstelle `IInputStream`. Wichtige Methoden sind:

API `zeusbase/System/Interfaces/IInputStream.hpp`
`Retval MQUALIFIER read(char& rBuffer, Int iBufferSize, Int& rValidSize) const` Liest ein n-Bytes aus dem Stream. Der Parameter `rValidSize` gibt an, wie viele Bytes effektiv gelesen wurden.
`bool MQUALIFIER isEmpty() const` Gibt an, ob noch Daten vorhanden sind, die nicht gelesen wurden.

`Int8 MQUALIFIER readInt8() const`
`bool MQUALIFIER readBool() const`
 	
List ein Byte aus dem Stream und gibt es zurück.
 	
Vorsicht: ist der Stream leer, gibt diese Methode immer 0 zurück. 0 kann aber auch ein gültiger Wert sein, wenn der Stream noch nicht leer ist. Durch das Verwenden von `isEmpty()` können Fehlinterpretationen vermieden werden.

`void MQUALIFIER close()` Schliesst ein Stream. Dies ist vor allem bei File-Streams wichtig.

3.4.9.2 Output-Stream

Die Output-Streams dienen dem Schreiben von Daten. Das Ziel ist je nach Output-Stream-Klasse verschieden.

API	
`TByteArrayOutputStream`	Ein Byte-Array dient als Ziel der Daten `zeusbase/System/ByteArrayOutputStream.h`
`TFileOutputStream`	Eine Datei dient als Ziel der Daten `zeusbase/System/FileOutputStream.h`

Diese Klassen implementieren die Schnittstelle `IOutputStream`. Wichtige Methoden sind:

API `zeusbase/System/Interfaces/IOutputStream.hpp`
`Retval MQUALIFIER write(const char* pBuffer, Int iBufferSize)` Schreibt n-Bytes in den Stream.

`Retval MQUALIFIER writeInt8(Int8 cData)`
`Retval MQUALIFIER writeBool(bool bData)`
 	
Schreibt ein Byte in den Stream.

`void MQUALIFIER flush()` Sendet den Stream-Buffer an das Device.
`void MQUALIFIER close()` Schliesst ein Stream. Dabei wird zuerst noch die Methode `flush()` ausgeführt. Die Methode `close()` ist vor allem bei File-Streams wichtig.

3.4.9.3 Stream-Filter

Zum Lesen oder Schreiben von Streams können Filter eingesetzt werden.

API	
`TFilterOutputStream`	Basisklasse für Output-Stream Filter `zeusbase/System/FilterOutputStream.h`

API	
TFilterInputStream	Basisklasse für Input-Stream Filter zeusbase/System/FilterInputStream.h
TBase64OutputStream	Filter zum decodieren von Base64 verschlüsselten Daten zeusbase/System/Base64OutputStream.h
TBase64InputStream	Filter zum codieren von Daten nach Base64 Verfahren zeusbase/System/Base64InputStream.h
TTextOutputStream	Filter zum Schreiben von Text. Dabei werden verschiedene Text-Codierungen wie UNICODE, UTF-8 usw. unterstützt. zeusbase/System/TextOutputStream.h
TTextInputStream	Filter zum Lesen von Text. Durch automatisches Erkennen der Text Codierung, können verschiedene Formate gelesen werden. zeusbase/System/TextInputStream.h
TCryptedOutputStream	Filter zum Verschlüsseln von Daten. Dabei wird der XTEA verwendet. zeusbase/Security/CryptedOutputStream.h
TCryptedInputStream	Filter zum Lesen von verschlüsselten Daten. Dabei wird der XTEA verwendet. zeusbase/Security/CryptedInputStream.h
TZippedOutputStream	Filter zum Komprimieren von Daten. Dabei wird die ZLib-Bibliothek verwendet zeusbase/System/ZippedInputStream.h
TZippedInputStream	Filter zum Lesen von komprimierten Daten zeusbase/System/ZippedInputStream.h

Die Filter-Klassen können beliebige Ketten bilden. Die Daten werden in jedem Glied dieser Kette verändert. Am Schluss der Kette stehen beim Schreiben eine Output-Stream-Klasse, die die Daten auf ein Device schreibt. Beim Lesen steht am Anfang der Kette eine Input-Stream-Klasse, welche Daten von einem Device liest (Siehe Abbildung).

Die Kette wird gebildet, indem der Filterklasse beim Konstruktor das Folgeobjekt bei Output-Streams oder bei Input-Streams der Vorgänger angegeben wird.

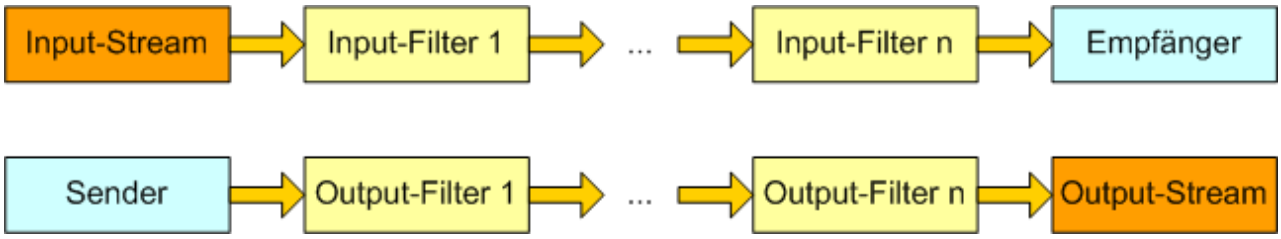


Abbildung 2: Filterklassen können beliebig zusammengesetzt werden. Die Ketten werden immer von links nach rechts abgearbeitet.

Beispiel zum Kopieren von Dateien

```
//-------------------------------------------------
// Dieses Beispiel kopiert eine Datei in eine base64 codierte Textdatei
#include <zeusbase/System/FileInputStream.h>
#include <zeusbase/System/FileOutputStream.h>
#include <zeusbase/System/Base64OutputStream.h>

TString strSourceFile = L"MyFile.jpg";
TString strTargetFile = L"MyFile.base64";

TAutoPtr<TFileInputStream> SourceFile
  = new TFileInputStream(strSourceFile);
TAutoPtr<TFileOutputStream> TargetFile
  = new TFileOutputStream(strTargetFile);

TAutoPtr<TBase64OutputStream> Decoder
  = new TBase64OutputStream(TargetFile);

Int iValidSize = 0;
bool bError = false;

do
{
  char cBuffer[1000];
  if (SourceFile->read(cBuffer, 1000, iValidSize) == RET_NOERROR)
  {
    //Der Decoder wandelt die bytes in Base64 Code und sendet
    // diesen an das Target-File
    bError = (Decoder->write(cBuffer, iValidSize) == RET_NOERROR);
  }
  else
  {
     bError = true; //Kann Source-File nicht lesen
  }
} while(iValidSize > 0 || bError);

//Decoder muss zuerst seine Daten abarbeiten (flushen)
Decoder->flush();

//Speicher  wird beim Verlassen des Scopes freigeben.
// Close kann aufgerufen werden, wird aber ebenfalls beim
// Verlassen des Scops aufgerufen.
SourceFile->close();
TargetFile->close();
```

Tabelle 20: Dieses Beispiel erläutert, wie Streams gebraucht werden können. Der Inhalt einer Datei wird in eine Base64-codierte Datei kopiert.

3.5 Networking und Internet

Das Zeus-Framework stellt diverse Klassen für die einfache Verwendung von Netzwerk zur Verfügung. Die Klassensammlung befindet sich im Verzeichnis `./zeusbase/Net/`.

3.5.1 Netzwerkklassen

Diese Gruppe von Klassen dient der direkten Programmierung von Netzwerk-Funktionen.

API	
`TIPAddress`	Datentyp zum Kapseln einer IP Adresse mit Port. Diese Klasse ist abstrakt. Es sind die konkreten Klassen `TIPv4Address` und `TIPv6Address` zu verwenden `zeusbase/Net/IPAddress.h`
`TIPv4Address`	Kapselt eine IP Adresse der Version 4 `zeusbase/Net/IPv4Address.h`
`TIPv6Address`	Kapselt eine IP Adresse der Version 6 `zeusbase/Net/IPv6Address.h`
`TSocket`	Diese Klasse implementiert einen Client Socket um TCP/IP Verbindungen aufzubauen `zeusbase/Net/Socket.h`
`TServerSocket`	Der Server kann mit dieser Klasse implementiert werden `zeusbase/Net/ServerSocket.h`
`TNetworkInterface`	Ermittelt die Netzwerkschnittstellen mit den konfigurierten TCP/IP Adressen `zeusbase/Net/NetworkInterface.h`

3.5.1.1 Klasse TSocket

Die Klasse `TSocket` kann eine Verbindung zu einem Server herstellen. Diese Verbindung kommt mit TCP/IP zustande. Um Sockets zu gebrauchen wird folgendermassen vorgegangen:

1. Initialisieren der Verbindung
2. Lesen bzw. Schreiben der Daten
3. Schliessen der Verbindung

Die Angaben mit wem die Verbindung zustande kommen soll, wird im Konstruktor angegeben.

API	`zeusbase/Net/Socket.h`
	`Retval connect();` Mit dem Server verbinden.
	`Retval disconnect();` Vom Server trennen.
	`void setBlockable(bool nMode);` Wird true übergeben, ist der Thread beim Aufruf von read() blockiert, bis alle Daten gelesen wurden. Wird false übergeben, gibt die read()-Methode einen Fehler, wenn keine Daten bereitstehen.
	`Retval MQUALIFIER read(char* pBuffer, Int iBufferSize, Int& rValidSize) const;` Lesen der Daten vom Server. Siehe IInputStream für weitere Methoden zum Lesen von Daten.
	`Retval MQUALIFIER write(const char* pBuffer, Int iBufferSize);` Senden der Daten. Siehe IOutputStream für weitere Methoden zum Schreiben von Daten.

Client Implementation

```
#include <zeusbase/Net/Socket.h>

void sendData(const IString& rAddress, Uint uiPort,
     IByteArray& rData)
{
  TAutoPtr<TSocket> Socket = new TSocket(rAddress, uiPort);

  if (Socket->connect() == RET_NOERROR)
  {
    Socket->write(rData.getArray(), rData.getCount());
    Socket->close();
  }
```

Tabelle 21: Benutzung der Socket-Klasse.

3.5.1.2 Klasse TServerSocket

Zur Implementation eines Servers wird die Klasse `TServerSocket` verwendet. Sie ist das Gegenstück zum `TSocket`. Zu den bereits beschrieben Methoden von `TSocket`, kommen weitere serverspezifische Methoden hinzu:

API	`zeusbase/Net/ServerSocket.h`
	`Retval bind(bool bDynamicBinding = false);` Bindet den Server Socket zu einer Adresse mit Port. Wird dynamische Bindung verlangt, wird der Port zufällig gewählt. Das wird vor allem bei Remote Objects verwendet.
	`Retval accept(TSocket*& rpClient)` Diese Methode gibt ein Client Socket zurück, wenn sich jemand auf den Server verbunden hat. Wurde vorgängig setBlockable() mit true aufgerufen, so wird dieser Aufruf blockiert, bis ein Client sich anmeldet.
	`Retval acceptTO(TSocket*& rpClient, Float fTimeout);` Diese Methode wartet eine bestimmte Zeit (timeout), wenn vorgängig setBlockable() mit false aufgerufen wurde,

Server Implementation

```
#include <zeusbase/System/Thread.h>
#include <zeusbase/System/SingleLinkedList.hpp>
#include <zeusbase/Net/ServerSocket.h>


USING_NAMESPACE_Zeus

/*****************************************************************/
/*! Der Server Thread verwaltet alle Verbindingen der Clients */
class TServerThread : public TThread
{
  public:

    /*! Erstellen des Servers*/
    TServerThread(const IString& rAddress, Uint uiPort)
     : TThread(), m_rServerSocket(*new TServerSocket(rAddress,
                                                     uiPort))
    {
      m_rServerSocket.setReUsable();
      m_rServerSocket.setBlockable(false);
      m_bHasError = m_rServerSocket.bind(false);
    }

    /*! auf Klienten warten*/
    virtual void execute()
    {
      while(!this->isInterrupted())
      {
        TSocket* pClient = NULL;
        if (m_pServerSocket->accept(pClient) == RET_NOERROR)
        {
          TClientThread* pClientThread = new TClientThread(*this,
            rClient);
          pClientThread->start();
          m_lstClients.add(pClientThread);
          pClient->release();
        }

        //TODO: Alle Klienten, die nicht mehr verbunden sind
        // aus der Liste entfernen.
      }
      //Kill all client threads
      while(m_lstClients.getCount() > 0)
      {
        m_lstClients[0]->kill(); m_lstClients[0]->release();
        m_lstClients.deleteItem(0);
      }
    }

  protected:
    ///List of all available clients
    TSingleLinkedList<TThread*> m_lstClients;

    virtual ~TServerThread()
    {
      m_rServerSocket.release();
    }

  private:
    ///Server socket
```

Server Implementation

```
    TServerSocket& m_rServerSocket;
    ///Error flag
    bool m_bHasError;
};

/*****************************************************************/
/*! Der Client Thread nimmt die Anfragen der Clients entgegen und
    beantwortet diese. */
class TClientThread : public TThread
{
  public :
    TClientThread(TServerThread& rParent, TSocket& rSocket)
     : TThread(),
       m_rParent(rParent),
       m_rClientSocket(rSocket)
    {
      m_rClientSocket.addRef();
      m_rParent.addRef();
      m_bExit = false;
    }

    /*! This kills the client thread */
    virtual void kill()
    {
      if (!this->isDead())
      {
        m_bExit = true;
        TThread::kill();
      }
    }

    /*! Thread execution */
    virtual void execute()
    {
      m_rClientSocket.setBlockable(false);
      while(!m_bExit)
      {
        //Daten der Clients empfangen und auswerten
      }
      //Abmelden beim Parent (Server Thread)

      //Schliessen der Verbindung
      m_rClientSocket.close();
    }

  protected:
    virtual ~TClientThread()
    {
      m_rClientSocket.release();
      m_rParent.release();
    }

  private:
    //Server thread
    TServerThread& m_rParent;
    ///Client thread
    TSocket& m_rClientSocket;
    bool m_bExit;
};
```

Tabelle 22: Beispiel eines verbindungsorientierten parallelen Servers. Der ServerThread akzeptiert die

Clients und erzeugt einen Thread (ClientThread), welcher die Anfragen des Clients beantwortet. Nach dem Beenden des ClientThreads muss die Verbindung geschlossen werden.

3.5.2 Internet-Klassen

Diese Gruppe befasst sich im den Internet-Standards wie Unified Resource Identifier URI oder dem HTTP Protokoll.

API	
`TURI`	Mit Hilfe dieser Klasse können URIs analysiert und gelesen werden. `zeusbase/Net/URI.h`
`THTTPRequest`	Ein HTTP Request-Paket. Die Klasse kann einerseits aus einer Zeichenkette einen Request parsen, oder einen neuen Request erstellen. Voll funktionsfähig für HTTP 1.0 `zeusbase/Net/Protocols/HTTPRequest.h`
`THTTPResponse`	Ein HTTP Response-Paket. Voll funktionsfähig für HTTP 1.0 `zeusbase/Net/Protocols/HTTPResponse.h`

3.5.3 Cell Communication Transfer Protocol

Zellen müssen untereinander kommunizieren können. Zellen die im gleichen Prozess arbeiten, können direkt über den Shared Memory kommunizieren. Befinden sich die Zellen oder Zellensysteme auf entfernten Computersystemen, müssen sie über ein Netzwerk kommunizieren können.

Mit dem Cell Communication Transfer Protocol CCTP können binäre Datenpakete ausgetauscht werden. Das Protokoll definiert einen Header von 12 Bytes für die Response- und Request-Pakete. Das Protokoll wird vor allem für die Kommunikation zwischen zwei Zellen verwendet.

Die Request-Pakete werden zum Senden der Datenpakete verwendet. Der Request dient der Anfrage beim entfernten System, ob sie die Meldung akzeptiert. Das entfernte System antwortet mit einer Response über aufgetretene Fehler oder Erfolge.

API	
`TCCTPRequest`	Request Objekt für das binäre Cell Communication Transfer Protocol `zeusbase/Net/Protoclos/CCTPRequest.h`
`TCCTPResponse`	Response Objekt für das CCTP `zeusbase/Net/Protoclos/CCTPResponse.h`

Der Body des Protokolls kann ein beliebiger binärer Datenblock sein, wie zum Bsp. Bilder, serialisierbare Objekte usw.

3.5.3.1 Request Pakete

Das Request Paket besteht aus folgenden Felder:

- Version: Dieses Feld dient der Protokollversion
- Methode: Definiert die Art und Weise wie der Request verarbeitet werden soll.
- Identität der Meldung: Diese ID kann wichtig sein, wenn Meldungen wiederholt werden müssen.
- Länge des Body: Anzahl Bytes des folgenden Datenblocks

Byte 1	Byte 2	Byte 3	Byte 4
Version	Methode	Reserviert	
Identität der Meldung (ID)			
Länge des Body in Bytes			
Body			

Tabelle 23: Format des Request-Pakets

3.5.3.2 Response Pakete

Das Response Paket besteht aus folgenden Teilen:

- Version: Protokollversion
- Zustand: Antwort, bzw Fehlercode (Analog zu HTTP)
- Länge des Body: Anzahl Bytes des folgenden Datenblocks

Byte 1	Byte 2	Byte 3	Byte 4
Version	Reserviert		
Zustand der Antwort			
Länge des Body in Bytes			
Body			

Tabelle 24: Format des Response-Pakets

3.6 Threading

Fürs Threading unter Linux[2] und Windows wurden Klassen erstellt, die Funktionen und Datentypen der Betriebssystem-API kapseln. Es wurde speziell darauf geachtet, dass das Verhalten auf beiden Systemen gleich ist.

Das Threading-Modell des Zeus-Frameworks ist die Basis für die bereits erwähnte Zellenarchitektur. Das Modell sieht für jeden Thread eine Meldungsqueue vor, welche vor allem für Synchronisation zwischen zwei Threads gedacht ist. Damit die Queues von anderen Threads gefunden werden können, wurde ein Singleton entworfen, der `ThreadManager`. Dieser verwaltet alle Queues der existierenden Threads (Siehe Abbildung).

Wichtige Hinweise zur Programmierung mit Threads:

2 für Linux werden POSIX Threads eingesetzt [PTHREAD] und [POSIXTH]

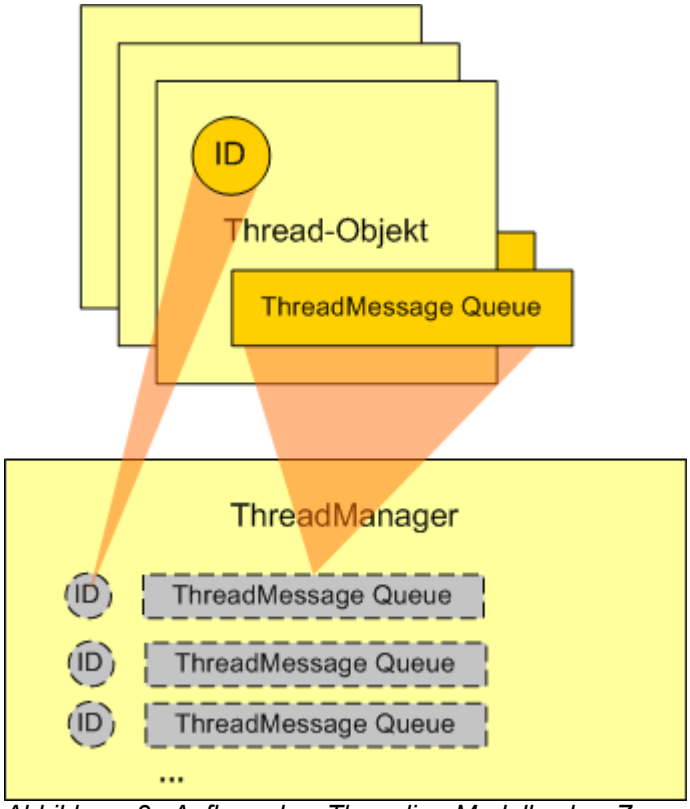


Abbildung 3: Aufbau des Threading-Modells des Zeus-Frameworks. Das Singleton `ThreadManager` verwaltet die Queues der Threads.

- Es ist darauf zu achten, dass ein Thread **NIE** völlig blockiert wird.
  - **Linux:** non-blockable Flags setzen.
  - **Windows:** API Funktionen mit alertable Flags verwenden (Beispiel SleepEx)
- Beim Endlos Loop ist das Interrupted-Flag des Threads zu prüfen
  - Innerhalb der Thread Klasse kann direkt auf `isInterrupted()` zugegriffen werden:

```
while(this->isInterrupted())
{
  ...
}
```

  - Ausserhalb des Thread-Objekts muss mindestens die `ThreadID` bekannt sein. Mit dem `ThreadManager` kann geprüft werden, ob dieser Thread unterbrochen wurde.

```
while(!ThreadManager.isThreadInterrupted(uiThreadID))
{
  ...
```

```
        }
```

Folgende Klassen sind fürs Threading implementiert:

API	
`TThread`	Ein Thread kann mit Hilfe dieser Klasse implementiert werden. `zeusbase/System/Thread.h`

`TAbstractMainThread`	Der MainThread der Applikation wird durch diese Klasse gekapselt. Sie ist abstrakt und muss je nach Entwicklungsumgebung abgeleitet werden. Folgende Klassen wurden bereits entwickelt:
	
	● `TConsoleMainThread`: Für Konsolen-Applikationen
	
	● `TBorlandMainThread`: Für Borland Applikationen
	
	`zeusbase/System/AbstractMainThread.h`
	`zeusbase/System/ConsoleMainThread.h`
	`zeusbase/System/Platforms/BorlandMainThread.h`

`TThreadManager`	Verwaltung aller Threads einer Applikation `zeusbase/System/ThreadManager.h`
`TCriticalSection`	Kritische Abschnitte und Locks können mit dieser Klasse umgesetzt werden. `zeusbase/System/CriticalSection.h`
`TEvent`	Diese Klasse dient dem Signalisieren von Threads. `zeusbase/System/Event.h`
`TTime`	Klasse für Zeit-Funktionen, wie `sleep()` und `getTimeStamp()`. `zeusbase/System/Time.h`
`TAtomicValueType<T>`	Eine template Klasse für threadsichere Datentypen. Jede Operation kann nicht von anderen Threads unterbrochen werden. `zeusbase/System/AtomicValueType.hpp`
`TAtomicInt`	Repräsentiert eine ganze Zahl als threadsicherer Datentyp `zeusbase/System/AtomicInt.hpp`
`TAtomicFloat`	Repräsentiert eine reelle Zahl als threadsicherer Datentyp `zeusbase/System/AtomicInt.hpp`
`TAtomicCounter`	Eine einfache Implementation für eine threadsichere Zählvariable `zeusbase/System/AtomicCounter.hpp`

3.6.1 Synchronisieren von Threads

Unter Synchronisieren von Threads verstehen wir, dass ein Thread A auf Thread B wartet, bis dieser seine definierte Arbeit beendet hat. Im Zeus-Framework wird diese Anforderung noch erweitert. An Hand eines Beispiels wird die Funktionsweise verdeutlicht (Siehe Abbildung 4):

- Thread A stellt Daten zur Verarbeitung durch Thread B bereit.
- Thread B wird benachrichtigt. Thread A wartet bis Thread B mit der Verarbeitung fertig ist.
- Thread B empfängt die Nachricht (Synchronisationsmeldungen) und führt die `process()` Methode aus (Siehe `TSynchronizeObject`). Durch diesen Aufruf werden die Daten (bereitgestellt durch Thread A) verarbeitet.
- Thread B wertet nach der Verarbeitung den Rückgabewert aus und benachrichtigt Thread A (noch schlafend).
- Thread A erwacht und führt seine Tätigkeit weiter.

Der Aufruf zur Synchronisation wird im Thread A durchgeführt. Dazu wurden folgende Makros angefertigt.

API	`SYNCHRONIZE_METHOD(uiThreadID, clss, mMethod)`
	Dieses Makro synchronisiert Thread A mit einem beliebigen Thread mit der ID `uiThreadID`. `clss`: Angabe der Klasse in deren sich die Synchronisationsmethode befindet `mMethod`: Name der Synchronisationsmethode

API	`SYNCHRONIZE_MT_METHOD(clss, mMethod)`
	Dieses Makro synchronisiert Thread A mit dem Main Thread. `clss`: Angabe der Klasse in deren sich die Synchronisationsmethode befindet `mMethod`: Name der Synchronisationsmethode

Die Synchronisation von Threads erfolgt durch deren Queues. Damit die Synchronisation funktioniert, müssen die Threads ihre Queue abarbeiten. Dies geschieht für den Main Thread durch die abgeleiteten Klassen von `TAbstractMainThread`. Für alle anderen Threads muss der Entwickler selber sorgen, die Queues richtig abzuarbeiten.

Das folgende Beispiel zeigt die Verwendung der Synchronisation-Makros

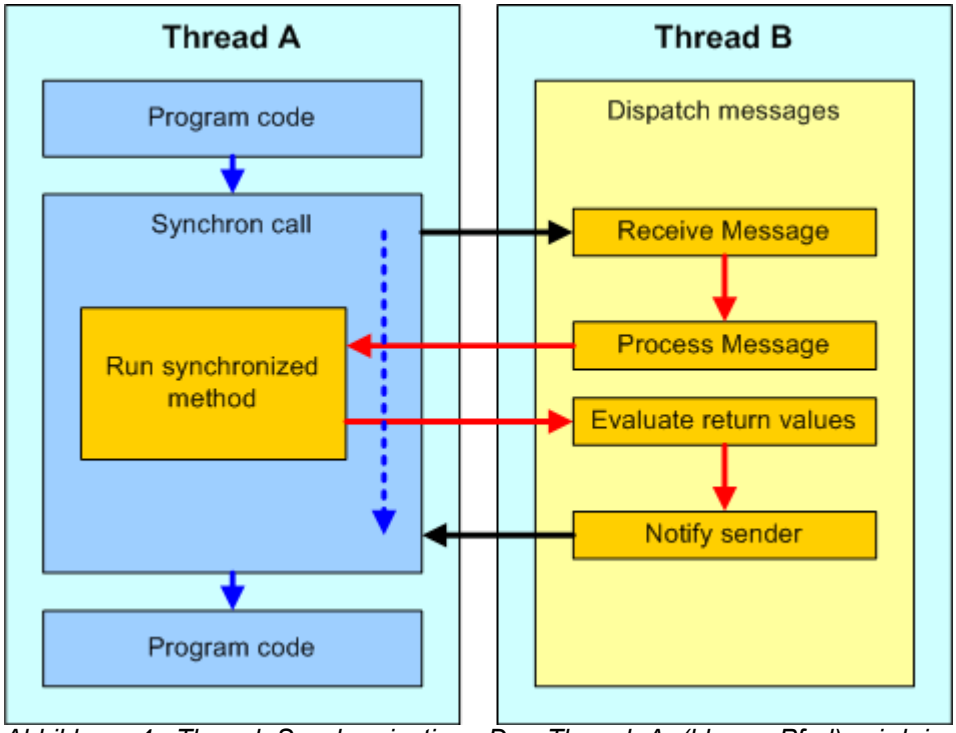


Abbildung 4: Thread Synchronisation. Der Thread A (blauer Pfad) wird im "Synchron call" suspendiert und wartet auf die Benachrichtigung des Thread B (roter Pfad).

Synchronisation von Threads

```
//---------------------------------------------------------------
//Method of thread A
void TMyClass::runA()
{
  //Stores all data for synchronisation in variables
  m_strSynchData = getData();

  //Call synchronized method
  SYNCHRONIZE_MT_METHOD(TMyClass, doSynchCallMT);
}


//---------------------------------------------------------------
//This method will be called from the main thread
void TMyClass::doSynchCallMT()
{
  printf("Data from Thread A %s", m_strSynchData.c_str(NULL));
}
```

Tabelle 25: Implementation zum Synchronisieren von Threads.

3.6.2 Asynchrone Kommunikation

Zwei Threads können auch asynchron miteinander Kommunizieren. Durch das Singleton `ThreadManager` können Objekte direkt in die Queue von Threads abgelegt werden. Dabei wartet der Thread A nicht auf die Verarbeitung der Daten, sondern führt seine Arbeit unmittelbar fort.

Die Daten, welche asynchron Verarbeitet werden, müssen in einer speziellen Klasse gekapselt werden. Die Klasse muss von `TSynchronizeObject` abgeleitet werden und dessen Methode `process()` überschreiben. Die Methode `process()` wird vom Thread B aufgerufen und verarbeitet die Daten.

Das Objekt wird dem `ThreadManager` übergeben. Dabei wird das Flag `bWaitForCompletion` nicht gesetzt.

3.6.3 Die Klasse TThread

Eine Applikation, die mehrere Abläufe parallel ausführen muss, braucht die Klasse `TThread`. Durch das Erzeugen eines Objekts von diesem Typ wird das Betriebssystem noch nicht angewiesen, ein neuen Thread zu erstellen. Erst beim Aufruf von `start()` wird ein Betriebssystem-Objekt erstellt. das ermöglicht ein einfaches Starten und Stoppen von Threads durch das gleiche `TThread`-Objekt.

Wird der Thread nicht mehr gebraucht, kann er mit `signalizeStop()` gestoppt werden. Diese Methode setzt das Interrupted-Flag des Threads. Wenn dieser in einem Loop seine Arbeit betätigt, sollte er in jedem Fall dieses Flag prüfen. Ist dies nicht der Fall, kann sich der Thread nicht selber beenden. Die Methode wartet 5 Sekunden auf die Terminierung des Threads. Diese Zeit kann mit dem Aufruf `signalizeStop(dTimeOut)` beliebig angepasst werden. In jedem Fall sollten mindestens 0.1 Sekunden gewartet werden.

Mit der Methode `kill()` wird der Thread definitiv beendet. Ohne Zeitangabe führt dieser Aufruf zur sofortigen Terminierung. Mit Zeitangabe wird zuerst ein `signalizeStop(dTimeOut)` ausgeführt. Der Aufruf `kill()` kann beim sofortigen Terminieren, sowie bei Threads, die sich nicht selber terminieren, Ressourcen Lecks verursachen.

API	zeusbase/System/Thread.h
	`bool start();` startet einen Thread. Die `execute()` Methode wird vom Thread abgearbeitet. Wird false zurückgegeben, konnte kein Thread-Objekt beim OS erstellt werden.
	`bool signalizeStop(Float64 dTimeOut=5.0);` Stoppt einen Thread. Das Interrupted-Flag wird gesetzt. Das Time-Out gibt die Zeit an, für welche auf die Terminierung des Threads gewartet wird.
	`void kill(Float64 dTimeOut=0);` stoppt den Thread definitiv. Mit dem Time-Out kann noch auf die Terminierung des Threads gewartet werden.
	`void suspend();` stoppt einen Thread. Mit der Methode resume() kann der Thread wieder gestartet werden.
	`void resume();` startet einen suspendierten Thread (gestoppt durch die Methode suspend()).
	`bool yield();` Der Thread gibt den Time Slice des Prozessors frei. Damit erhalten andere Threads die Möglichkeit, den Prozessor zu verwenden.

	`void setPriority(EPriority ePriority);`
	
	Setzt die Thread-Priorität. Folgende Typen sind definiert:
	`[etNormal, etLow, etHigh, etLower, etHigher, etTimeCritical]`

	`Uint MQUALIFIER getThreadID() const;` Gibt die ThreadID zurück. Diese ID ist erst gültig, **nachdem der Thread gestartet wurde**. Mit dieser ID kann der Thread beim `ThreadManager` ermittelt werden.

	`virtual Retval postObject(ISynchronizeObject& rObject,`
	`bool bWaitForCompletion);`
	
	Sendet ein Objekt zum Synchronisieren. Ist der Thread nicht gestartet, muss darauf geachtet werden, dass entweder zuerst noch ein `start()` ausgeführt wird, oder das Flag `bWaitForCompletion` nicht gesetzt ist, **sonst könnte es zu Deadlocks führen.**

Folgende Methoden können vollständig überschrieben werden, dass heisst die Basisklasse bracht nicht aufgerufen zu werden (Overwriting):

API	zeusbase/System/Thread.h
	`virtual void execute();` Diese Methode muss von abgeleiteten Klassen überschrieben werden. Durch den Aufruf von start() wird der Thread diese Methode ausführen. Rufen sie execute() nicht direkt auf.
	`virtual void onStarted();` Diese Methode wird vom Thread ausgeführt, bevor er die `execute()`- Methode ausführt.
	`virtual void onTerminated();` Diese Methode wird vom Thread ausgeführt, nachdem er die `execute()`- Methode verlassen hat.

Implementation eines Threads

```
#include <zeusbase/System/Thread.h>


USING_NAMESPACE_Zeus

class TMyThread : public TThread
{
  public:
    TMyThread() : TThread()
    {
    }

  protected:
    virtual void execute()
    {
      while (!this->isInterrupted())
      {
        //Do something here
      }
    }
}
```

Tabelle 26: Dieses Beispiel zeigt, wie einfach ein Thread implementiert wird. Es ist jedoch darauf hinzuweisen, dass die sichere Verwendung von Threads nicht so trivial ist.

3.6.4 Main Thread Klassen

Der Thread der Applikation (Main Thread) wurde mit der speziellen Klasse `TAbstractMainThread` implementiert. Diese kapselt den Thread nur (ohne einen

neuen zu erstellen) und stellt eine Meldungsqueue zur Verfügung. Damit diese Meldungsqueue auf den verschiedenen Plattformen, wie Borland, QT oder MFC verarbeitet wird, wurden spezialisierte Klassen erstellt.

Die allgemeine Verwendung dieser Klassen ist für jede Plattform gleich.

1. Registrieren der Queue des Main Threads
2. Initialisieren verschiedener Objekte der Applikation
3. Starten der Applikation. Hier werden die Synchronisationsmeldungen verarbeitet.
4. Entfernen der Queue des Main Threads aus der Thread Verwaltung (`ThreadManager`)
5. Entfernen der Objekte

Punkt 4 und 5 können auch getauscht werden. Wichtig ist, dass der LibraryManager die CodeModule erst nach dem Entfernen der Main Thread Queue entlädt.

API	`zeusbase/System/AbstractMainThread.h`
	`void registerThread(TThreadManager& rManager, bool bUseInternalDispatcher);` Registriert die Queue des Main Threads beim angegebenen `ThreadManager`. Das Flag `bUseInternalDispatcher` wird für spezifische Klassen gebraucht, um das Verarbeiten der Synchronisationsmeldungen extern zu regeln (`=false`).
	`void unregisterThread(TThreadManager& rManager)` Entfernt die Queue von der Thread Verwaltung.
	`bool MQUALIFIER isInterrupted() const;` Prüfen, ob das Interrupted-Flag gesetzt wurde.
	`Uint MQUALIFIER getThreadID() const;` Gibt die ID des Main Threads zurück.

3.6.4.1 Hinweis zu TAbstractFrameLoader und TAbstractMainThread:

Wird mit der Klasse `TAbstractFrameLoader` oder von ihr abgeleiteten Klassen

gearbeitet, übernimmt die Frame Loader-Klasse das Entfernen der Code-Module (Destruktor). Deshalb ist es wichtig, dass diese Instanz erst nach dem Entfernen der Queue freigegeben wird.

Tipp: Am einfachsten ist es, beim Implementieren des eigenen Frame-Loaders, die Queue des Main Threads im Konstruktor zu registrieren und im Destruktor zu Entfernen. Siehe Klasse `CCMFrameLoader.cpp`.

3.6.4.2 Klasse TConsoleMainThread

Die `TConsoleMainThread` ist eine plattformunabhängige Klasse zur Kapselung des Main Threads einer Konsolen-Applikation. Durch den Aufruf von `start()`, wird ein Endlos-Loop gestartet, in dem alle Synchronisationsmeldungen verarbeitet werden. Die Applikation reagiert auf CTRL + C (unter Linux und Windows). Dadurch kann der Endlos-Loop verlassen werden, damit die Applikation ordnungsgemäss beenden kann.

API	`zeusbase/System/ConsoleMainThread.h`
	`void start();` Startet die Applikationsschleife (Loop). Hiermit werden alle Meldungen der Queue verarbeitet, bis CTRL + C betätigt wurde.
	`void initControlHandler();` Initialisiert den ControlHandler für CTRL + C Ereignisse.
	`static void terminate();` Der Aufruf dieser Methode veranlasst den Main Thread, den Loop zu verlassen. Die `start()` Methode wird verlassen
	`static bool isTerminated();` Prüfen, ob das Terminated-Flag gesetzt wurde. Gleichwertig wie `isInterrupted()`.

Eine Konsolen-Applikation

```
#include <zeusbase/System/ConsoleMainThread.h>
#include <zeusbase/System/ThreadManager.h>


USING_NAMESPACE_Zeus

int main(int argc, char* argv[])
{
  //Initialises the control handler
  ConsoleMainThread.initControlHandler();

  //Initialises the multithreading synchronisation
  ConsoleMainThread.registerThread(ThreadManager, true);

  //Do initialisation here
  ...

  //Starts the application loop
  ConsoleMainThread.start();


  ConsoleMainThread.unregisterThread(ThreadManager);

  //Do cleaning here
  ...
}
```

Tabelle 27: Implementation einer Konsolen-Applikation mit Verarbeitung von Synchronisationsmeldungen.

3.6.4.3 Klasse TBorlandMainThread

Diese Klasse wurde für Borland C++ Builder 6.0 entwickelt und kann für andere Borland Plattformen angepasst werden.

Die Plattform von Borland beinhaltet eine Klasse `TApplication` für GUI Anwendungen, welche den Main Thread Von dieser Klasse existiert ein globales Objekt `Application`. Durch den Aufruf `Application->run()` wird die GUI Anwendung gestartet. Alle Windows-Meldungen werden nun durch Windows eigene Queues verarbeitet.

Damit wir nun Synchronisationsmeldungen verarbeiten können müssen wir folgende Schritte unternehmen:

1. das Singleton-Objekt vom Typ `TBorlandMainThread` initialisieren

2. Queue registrieren. Ist das Flag `bUseInternalDispatcher` der Methode `registerThread()` gesetzt, werden die Synchronisationsmeldungen automatisch verarbeitet. Wird das Flag nicht gesetzt, müssen die Meldungen

eigens verarbeitet werden.

3. Aufruf `Application->run()`

4. Entfernen der Queue.

Damit in einem Windows-System die Meldungen richtig verarbeitet werden, wird die Windows-Message Queue verwendet. Die Queue des Main Thread Objekts sendet eine Windowsmessage. Der Windowsmessage Dispatcher verarbeitet diese und ruft seinerseits die Verarbeitungsmethode des Main Thread Objekts auf (Siehe Abbildung).

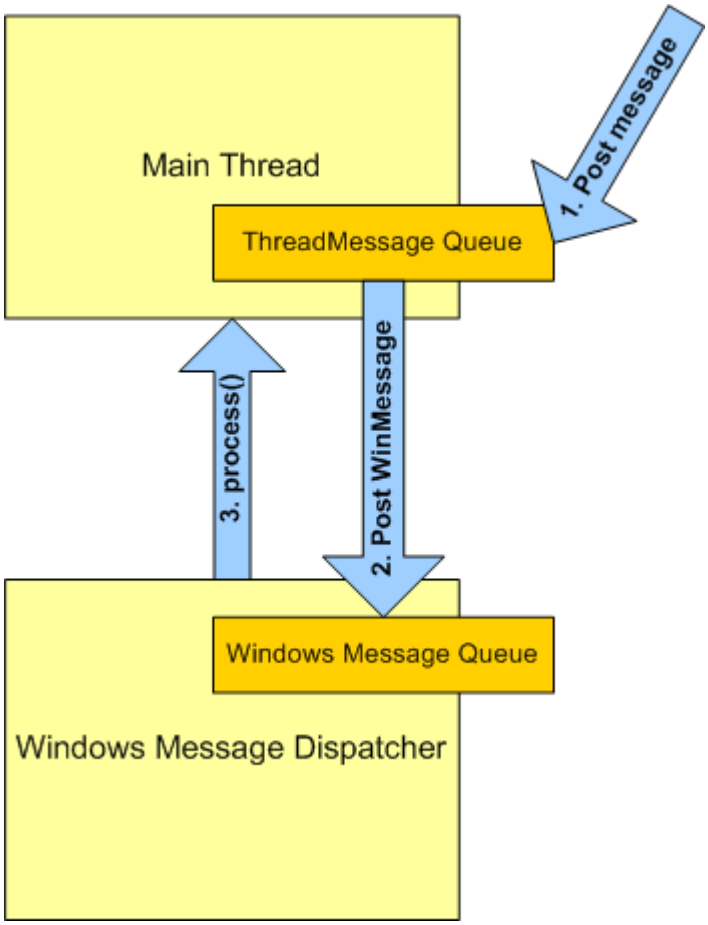


Abbildung 5: Schematische Darstellung zur Verarbeitung von Synchronisationsmeldungen für den Main Thread bei Windows Applikationen.

Die folgenden 2 Beispiele zeigen die Implementierung einmal mit internem Dispatcher und einmal mit eigenem Dispatcher.

Eine Borland GUI-Applikation

```
#include <zeusbase/System/ThreadManager.h>
#include <zeusbase/System/Platforms/BorlandMainThread.hpp>

...

USING_NAMESPACE_Zeus

//Initialisation of the instance
TBorlandMainThread TBorlandMainThread::m_Instance;


WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  //Initialises the multithreading synchronisation
  BorlandMainThread.registerThread(ThreadManager,
                                                   true);
  //Do initialisation here
  ...

  //Starts the application loop
  Application->Run();


  BorlandMainThread.unregisterThread(ThreadManager);

  //Do cleaning here
  ...
}
```

Tabelle 28: Implementation einer Borland GUI-Applikation mit interner Verarbeitung von Synchronisationsmeldungen.

Das folgende Beispiel zeigt den Programmcode zur eigenen Verarbeitung der Synchronisationsmeldungen.

Eine Borland GUI-Applikation
Eigenes Verarbeiten

```
//Zuweisen der Methode zum verarbeiten von Windowsmeldungen
{
  ...
  Application->OnMessage = performAppMessage;
  ...
}

//Methode empfängt alle Windowsmeldungen dieser Applikation
void __fastcall TMyClass::performAppMessage(tagMSG& Msg,
                                            bool& bHandled)
{
  //Verarbeiten
  BorlandMainThread.performAppMessage(Msg, bHandled);
  ...
}
```

Tabelle 29: Implementation einer Borland GUI-Applikation mit eigener Verarbeitung von Synchronisationsmeldungen.

3.6.4.4 Klasse TQTMainThread

Noch nicht vorhanden.

3.6.4.5 Klasse TMFCMainThread

Noch nicht vorhanden.

3.6.5 Der ThreadManager

Der `ThreadManager` dient der Verwaltung von Threads, deren Interrupted-Flags und Queues. Diese Verwaltung ist speziell für Multithreading entwickelt, das heisst, sie von verschiedenen Threads gleichzeitig verwendet werden.

Für folgende Hauptfunktionen wurde die Verwaltung konzipiert:

- Ermitteln der Main Thread ID

- Ermitteln der Thread ID des aktuellen Threads.
- Prüfen von Interrupted-Flags von registrierten Threads
- Unterbrechen, bzw. Beenden von registrierten Threads
- Senden von Synchronisationsmeldungen an registrierte Threads

3.6.6 Die Klasse TCriticalSection

Um kritische Bereiche in einem Programm zu schützen, wurde die Critical Section entwickelt. Durch diesen Mechanismus können

- Bereiche im Programm vor gleichzeitigem Zugriff von mehreren Threads geschützt werden
- Variablen und nicht Multithreading fähige Objekte geschützt werden

Die Klasse TCriticalSection erlaubt verschiedene Modi (voll unterstützt nur im Linux). Die Modi müssen beim Erstellen des Objekts bekannt sein.

- **ltFast**: Schnellere Verarbeitung durch das Betriebssystem.
- **ltRecursive**: Der gleiche Thread kann den Abschnitt mehrmals passieren (Reentrant mit gleichem Thread)
- **ltError**: Für Fehlerprüfung (nur Linux)

API	`zeusbase/System/CriticalSection.h`
	`void MQUALIFIER enter();` Sperrt den Bereich. Ist der Bereich bereits durch einen anderen Thread gesperrt, wird dieser Thread warten, bis der Bereich wieder freigegeben wird.
	`void MQUALIFIER leave();` Den Bereich wieder freigegeben.
	`bool isLocked();` Prüft ob der Bereich bereits besetzt ist.

Verwendung einer Critical Section

```
#include <>


class  TMyClass: public TZObject
{
  public:

    //Constructor
    TMyClass()
     : TZObject(),
       m_rLock(*new TCriticalSection(TCriticalSection::ltRecursive))
    {

    }

    //Destructor
    virtual ~TMyClass()
    {
      m_rLock.release();
    }

    //multi threading access
    void  doIt()
    {
      m_rLock.enter();

      //Access to objects and variables
      ...

      m_rLock.leave();
    }

  private:
    //Lock to protect other data
    TCriticalSection& m_rLock;
}
```

Tabelle 30: Verwendung von kritischen Bereichen mit Zeus-Framework.

3.6.7 Die Klasse TMutex

Die Klasse TMutex entspricht grösstenteils der TCriticalSection. Bei Windows-Plattformen wird im Gegensatz zu der TCriticalSection bereits beim Initialisieren ein Kernel Objekt erzeugt.

3.6.8 Die Klasse TEvent

Damit verschiedene Threads miteinander kommunizieren können und ein ewiges Abfragen von Zuständen effizienter gelöst werden kann, wurde das Ereignis implementiert. Es ermöglicht die Thread-Synchronisation. Ein Ereignis hat zwei Zugriffspunkte

- der Empfänger, meistens als wartendes Element
- der Sender oder Auslöser des Ereignisses.

Durch Ereignisse kann der Empfänger zum Beispiel auf die Verarbeitung der Daten warten. Sind die Daten verarbeitet, wird das Ereignis gesetzt, der Empfänger wird benachrichtigt. Nun kann er die Daten abholen.

Es gibt zwei Modi:

- **etBlocked**: Der wartende Thread ist blockiert. Er kann nur durch das Setzen des Ereignis oder durch signalizeStop() aufgeweckt werden
- **etActive**: Der wartende Thread ist zwar blockiert, er verarbeitet aber die Daten aus seiner Message-Queue (nur für Windows implementiert).

API	`zeusbase/System/Event.h`
	`void set();` Setzt das Ereignis.
	`void reset();` Setzt das Ereignis zurück.
	`bool wait(Float64 dSeconds = 10.0);` Wartet auf ein Ereignis mit einem TimeOut. wird kein TimeOut angegeben, wird max. 10 Sekunden gewartet. Die Rückgabe gibt an, ob abgebrochen wurden (`=false`) oder das Ereignis eintraf (`=true`).
	`bool waitInfinite();` Wartet auf ein Ereignis ohne TimeOut. Rückgabe siehe `wait()` Methode.

3.6.9 Die Klasse TTime

Die Klasse `TTime` wird im Threading genutzt, um einen Thread für definierte Zeit zu suspendieren (sleep).

Weitere Funktionen sind

- Generieren von Zufallszahlen
- Ermitteln von Zeitstempeln

Für Datums- und Uhrzeit-Funktionen wird auf die Klasse `TCalendar` verwiesen.

API	`zeusbase/System/Time.h`
	`static Float64 getTimeStamp();` Gibt einen Zeitstempel zurück. Der Zeitstempel ist in Sekunden.
	`static void sleep(Float64 dDwell);` Der aktuelle Thread geht für dDwell-Sekunden schlaffen.
	`static void initRandom();` Initialisiert die Random-Funktion für den Prozess

	`static Uint32 random(Uint32 ulFrom, Uint32 ulTo);`
	`static Float randomFloat(Float dFrom, Float dTo);`
	`static Int randomInt(Int iFrom, Int iTo);`
	
	Gibt einen zufälligen Wert zurück. Es gibt verschiedene Werte-Typen, wie Integer, Float und unsigned Integers.

3.7 Singletons

Für Objekte die nur einmal im System vorkommen sollen, bietet das Framework die Möglichkeit so genannte Singletons zu verwalten. Dabei stellt sich folgende Problematik:

Wie können wir sicherstellen, dass ein solches Objekt nur einmal vorhanden ist, egal wie viele Bibliotheken geladen sind, egal ob diese Bibliotheken einen eigenen Heap verwenden, eigene statische Initialisierungen vornehmen?

Diese Aufgabe übernimmt die Framework-Klasse TSingletonManager. Diese Klasse verwaltet alle Singletons und kann diese Singletons an Bibliotheken weitergeben (Siehe Entwickeln von Code-Modulen). Dabei werden folgende Bereiche unterteilt:

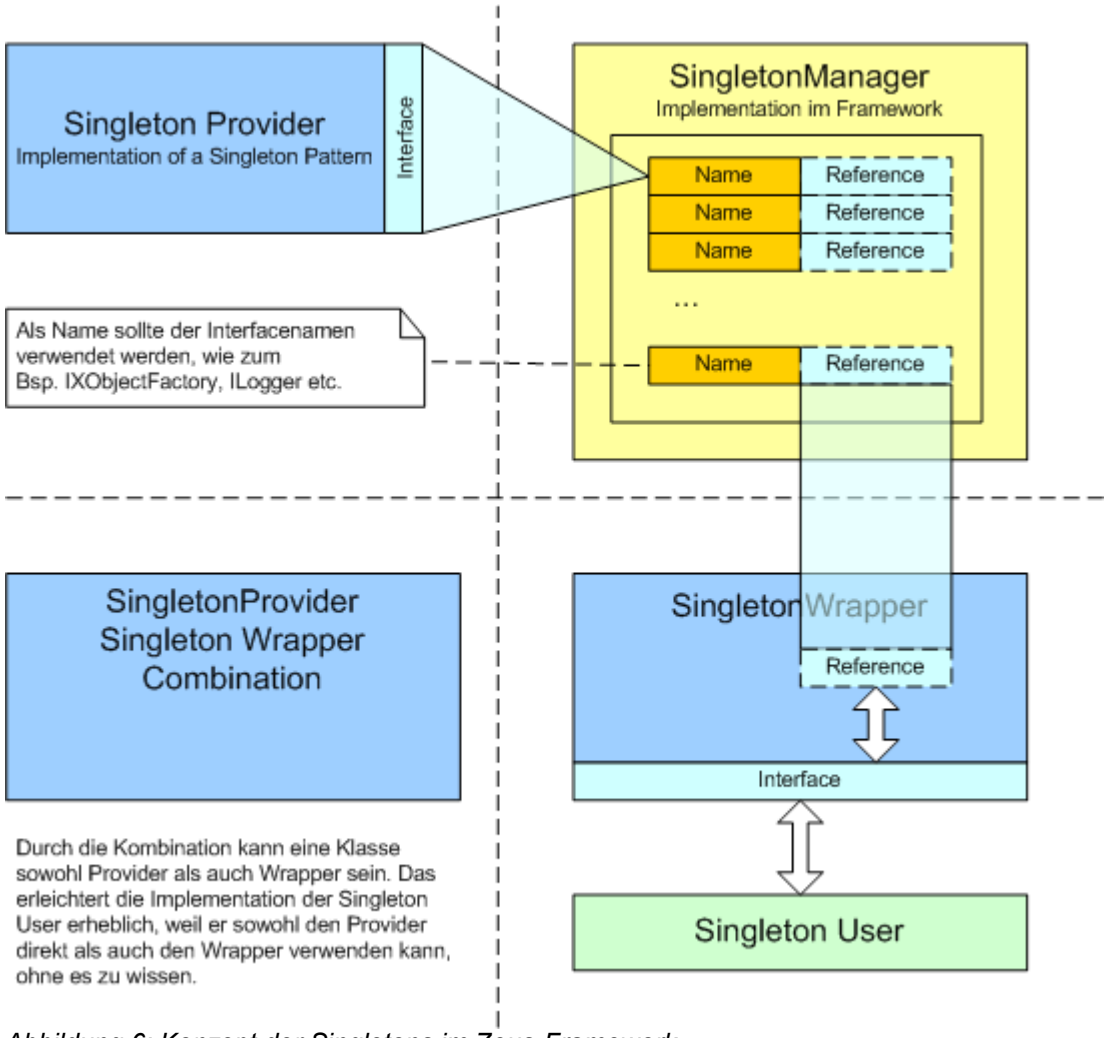


Abbildung 6: Konzept der Singletons im Zeus-Framework

- Der **Singleton-Provider** ist ein Singleton-Objekt, welches die Funktionalität zur Verfügung stellt

- Der **Singleton-Wrapper** ist ein Singleton-Objekt, welches die Schnittstelle eines Singletons oder eines beliebigen Objekts kapselt.

- Der **Singleton-Manager** verwaltet systemweit alle Singletons. Durch ihn können Bibliotheken mit einem Heap und Speicherverwaltung auf die original Singletons zugreifen.

- Durch die **Kombination** von Singleton-Provider und Singleton-Wrapper können Klassen universell im Framework und in Bibliotheken eingesetzt werden. Fast alle Singleton-Klassen des Zeus-Frameworks sind in Kombination dieser beiden Typen entwickelt.

Folgende Singletons stehen im Zeus-Framework zur Verfügung:

API	
`LibraryManager`	Der Library-Manager dient der Verwaltung von geladenen Bibliotheken. `zeusbase/System/LibraryManager.h`
`Logger(name)`	Singleton zum Loggen von Meldungen und Fehler. `zeusbase/System/LoggerManager.h`
`SingletonManager`	Ein Singleton zur Verwaltung von anderen Singletons. `zeusbase/System/SingletonManager.h`
`MyXObjectFactory`	Fabrik zum Laden von X-Objektbäumen. (Siehe MOM 2 Implementation) `zeusbase/System/XObjectFactory.h`
`ZObjectFactory`	Fabrik zum Erzeugen von beliebigen Objekten aus einer Bibliothek oder zum Erzeugen von Objekten aus einem Objektstream. `zeusbase/System/ZObjectFactory.h`
`Naming`	Naming-Service zum Registrieren von Remote Objekten (Siehe Kapitel Remote Method Invocation mit C++) `zeusbase/Remote/Naming.h`
`SecurityManager`	Der Sicherheit-Manager verwaltet die Sicherheitsbestimmungen eines Systems. (Siehe das Kapitel Das Sicherheitskonzept) `zeusbase/Security/SecurityManager.h`
`SettingsManager`	Dieses Singleton verwaltet Applikationseinstellungen und Applikationsdaten. Die Einstellungen stammen aus einer Property-Datei, die Applikationsdaten aus einer XML Datei (siehe `AbstractFrameLoader`). `zeusbase/System/SettingsManager.h`
`ThreadManager`	Der `ThreadManager` dient der Verwaltung der vorhandenen Threads. Insbesondere werden das Interrupted-Flag und die Thread Queue von jedem Thread verwaltet. (Siehe Kapitel Threading).

3.7.1 Verwalten von Bibliotheken mit dem Library-Manager

Der Library-Manager dient zum Verwalten von Bibliotheken (DLL's) während der Laufzeit. Der Library-Manager enthält eine Liste der geladenen Bibliotheken.

3.7.1.1 Wichtige Methoden

Folgende Liste enthält die wichtigsten Methoden des Library-Managers:

API `zeusbase/System/LibraryManager.h`

```
retval MQUALIFIER createObject(const IString& rCodeModule,
                               const IString& rClassName,
                               IZUnknown*& rpIface);
```

Versucht ein Objekt der Bibliothek zu erstellen. Das Objekt muss im Minimum die Schnittstelle IZUnknown implementieren.

Es können keine X-Objekte erzeugt werden!!! X-Objekte brauchen ein XML Knoten zum initialisieren. (Siehe Kapitel MOM 2 Implementation).

Erstellen eines Objekts aus einer Bibliothek

```
//-------------------------------------------------
// Bibliothek direkt im ModulePath
#include <zeusbase/System/LibraryManager.h>
#include <IMyInterface.hpp>

IMyInterface* pObject = NULL;
if (LibraryManager.createObject(TString(L"MyLibrary"),
                                TString(L"IMyInterface"),
                                ICAST(pObject))
                                ==RET_NOERROR)
{
  ...
  pObject->release();
}
```

Tabelle 31: Erstellen eines Objekts mit Hilfe des Library-Managers. Dabei wird die Bibliothek `MyLibrary.dll` auf Windows bzw. `MyLibrary.so` auf Linux Systemen geladen und die exportierte Funktion `createIMyInterface()` aufgerufen.

Zu beachten ist, dass nur der Name der Bibliothek (`codemodule`) angegeben werden muss. Die Endung [.dll] oder [.so] wird je nach System angehängt. Die Bibliotheken müssen sich direkt im Code-Module-Pfad befinden oder in einem Unterverzeichnis. Bei der Verwendung von Unterverzeichnissen, muss der Namen des Unterverzeichnisses in der Angabe des codemodule enthalten sein.

Erstellen eines Objekts aus einer Bibliothek

```
//-------------------------------------------------
// Bibliothek befindet sich in einem Unterverzeichnis
#include <zeusbase/System/LibraryManager.h>
#include <IMyInterface.hpp>

IMyInterface* pObject = NULL;
if (LibraryManager.createObject(TString(L"SubDir/MyLibrary"),
                                TString(L"IMyInterface"),
                                ICAST(pObject))
                                ==RET_NOERROR)
{
  ...
  pObject->release();
}
```

Tabelle 32: Die Bibliothek `MyLibrary.dll` auf Windows bzw. `MyLibrary.so` auf Linux Systemen wird aus dem Verzeichnis `$(ModulePath)/SubDir/` geladen .

3.7.2 Loggen mit dem Logger-Manager

Der Logger-Manager dient der Ausgabe von Meldungen auf eine Konsole oder in eine Log-Datei. Der Logger-Manager braucht verschiedene Einstellungen, die mit dem Framework normalerweise in der `*.properties`-Datei enthalten sind.

Zur Entwicklungszeit braucht der Log-Aufruf folgende Parameter:

API	`zeusbase/System/LoggerManager.h`
	`void printfln(Uint uiMode, char* pTxt,...);` Schreibt eine Log-Ausgabe. Der Parameter `uiMode` gibt den Ausgabemodus an (Siehe unten). Der Parameter `pTxt` enthält Text mit Formatierungen (siehe `printf` von `libc`).

Es gibt 5 Ausgabemodi:

- Fataler Fehler (`LOGMODE_FATAL`)
- Fehler (`LOGMODE_ERROR`)
- Warnung (`LOGMODE_WARN`)
- Information (`LOGMODE_INFO`)
- Debug Ausgabe (`LOGMODE_DEBUG`)

Zur Formatierung der Ausgabe gelten jene Richtlinien des sprintf der Standard C-Bibliothek.

Ein Logger-Objekt besitzt zudem einen Namen. Je nach Logging-Dienst kann dieser Name hierarchisch orientiert sein (wie zum Beispiel bei LOG4CXX).

Im Zeus-Framework braucht ein neuer Logger nicht speziell erzeugt werden. Die Angabe des Namens im Makro `Logger(name)` erledigt dies automatisch.

Loggen

```
#include <zeusbase/System/LoggerManager.h>
TString strData ="Meldung A";

//Ausgabe durch den Root-Logger (ohne Logger-Namen)
RootLogger.printfln(LOGMODE_ERROR, "Ausgabe einer Meldung [%s]",
                                    m_strData.c_str(NULL));

//Ausgabe durch einen eigenen Logger
Logger(L"MyModule").printfln(LOGMODE_DEBUG, "Debug Ausgabe");
```

Tabelle Anwendungsbeispiel eines Loggers

3.7.3 Lesen von Einstellungen mit dem SettingsManager

Die Einstellungen können mit dem `SettingsManager` Singleton gelesen und auch verändert werden.

In der aktuellen Version werden nur die veränderten Einstellungen der User-Daten zurück in die Datei geschrieben.

3.7.3.1 Die Propertiy-Schnittstelle

Der `SettingsManager` verfügt über folgende Methoden:

API `stwbase/SettingsManager.h`

```
virtual Retval MQUALIFIER addProperty(const IString& rName,
                                      const IString& rValue);
```

Fügt eine neue Eigenschaft hinzu. Wenn sie bereits existiert, wird die alte überschrieben.

API	stwbase/SettingsManager.h

```
bool MQUALIFIER existsProperty(const IString& rName) const;
```

Prüft, ob eine Eigenschaft existiert.

```
virtual Retval MQUALIFIER getPropertyValue(const IString& rName,
                                           IString& rValue)const ;
```

Ermittelt den Wert einer Eigenschaft.

```
virtual Retval MQUALIFIER getPropertyPathValue(const IString& rName,
                                          IString& rValue)const;
```

Ermittelt den Wert einer Eigenschaft als Pfad (Abschliessendes \\ garantiert).

```
virtual Retval MQUALIFIER getPropertyFileValue(const IString& rName,
                                          IString& rValue)const;
```

Ermittelt den Wert einer Eigenschaft als Dateinamen.

Der Namen der Eigenschaft kann aus der Properties-Datei entnommen werden.

Lesen von Properties

```
#include <zeusbase/System/SettingsManager.h>

TString strLibs;

SettingsManager.getPropertyValue(TString(L"zeus.LibraryManager"),
                                 strLibs);
```

Tabelle Anwendungsbeispiel zum Lesen von Properties

3.7.3.2 Die User Daten Schnittstelle

Durch diese Schnittstelle können Applikationsdaten oder Userdaten gelesen, verändert und gespeichert werden. Diese Daten sind in Form eines XML Dokuments gespeichert.

Folgende Methoden stehen zur Verfügung:

API	stwbase/SettingsManager.h

```
void setUserXMLFile(const IString& rFileName);

void setUserXMLFile(TXMLFile& rFile);
```

Laded eine neue Datei mit Userdaten.

API	stwbase/SettingsManager.h
	`Retval UserData.commit();` Speichert die Daten zurück in das XML Dokument

	`Retval UserData.readInt(const IString& rKey, Int& rValue, Int iDefault = 0);`
	
	`Retval UserData.readFloat(const IString& rKey, Float& rValue, Float fDefault = 0.0);`
	
	`Retval UserData.readString(const IString& rKey, IString& rValue, TString strDefault=L"");`
	
	Lesen der Daten. Der Schlüssel dient zur Adressierung der Daten. Können Daten nicht gefunden werden, wird der Default-Wert und RET_REQUEST_FAILED zurückgeben.

	`Retval UserData.writeInt(const IString& rKey, Int iValue);`
	
	`Retval UserData.writeFloat(const IString& rKey, Float fValue);`
	
	`Retval UserData.writeString(const IString& rKey, const IString& rValue);`
	
	Schreiben von Daten. Das XML Dokument wird nicht automatisch gespeichert. Es ist hier die Methode commit() zu verwenden.

	`void UserData.attach(INotifyObserver& rObserver);` Fügt einen Observer hinzu, welcher über Änderungen benachrichtigt werden kann.
	`void UserData.detach(INotifyObserver& rObserver);` Entfernt einen Observer. Für Module (Bibliotheken) ist es wichtig, dass Observer vor dem Entfernten der Bibliothek aus dem Speicher freigegeben werden!!
	`void UserData.notifyObservers();` Benachrichtigt alle Observer

Die Userdaten-Schnittstelle ist in ein separates Kindobjekt `UserData` ausgelagert.

Lesen von User Daten

```
#include <zeusbase/System/SettingsManager.h>

SettingsManager.setUserXMLFile(TString(L"./User.xml"));
...

SettingsManager.UserData.writeInt(TString(L"Test.Data"), 34);
...

SettingsManager.UserData.commit();
```

Tabelle Anwendungsbeispiel zum Lesen von UserDaten

3.8 Klassen für XML-Handling

Das Zeus-Framework stellt eine umfassende Funktionalität mit XML[3] Daten zur Verfügung. Lesen, Schreiben und Traversieren von XML Knoten sind die Grundfunktionen. Die XML-Funktionalität ist im Zeus-Framework nicht direkt implementiert. Die Bibliothek `XML_Service.dll` bzw `XML_Service.so` muss im Verzeichnis `./modules` vorhanden sein. Diese Bibliothek dient als Wrapper von XML Implementationen wie `MSXML` oder Apache `Xerces/Xalan`.

Das Zeus-Framework definiert folgende Schnittstellen.

API	
`TXMLFile`	Klasse zur Verwaltung einer XML Datei. Dabei können sowohl einfache XML Dateien, als auch komplexere X-Objekt-Dateien verwaltet und manipuliert werden. `zeusbase/System/XMLFile.h`
`TXMLStream`	Klasse zur Verwaltung eines XML Streams. Funktionen wie bei TXMLFile. `zeusbase/System/XMLStream.h`
`IXMLParser`	Schnittstelle zum Parsen eines XML Dokuments. `zeusbase/System/Interfaces/IXMLParser.hpp`
`IXMLDocument`	Verwaltungsklasse eines XML Dokuments. `zeusbase/System/Interfaces/IXMLDocument.hpp`
`IXMLNode`	Ein XML Knoten, welcher Attribute und Text beinhalten kann. `zeusbase/System/Interfaces/IXMLNode.hpp`
`IXMLNodeList`	Eine Liste von XML Knoten. `zeusbase/System/Interfaces/IXMLNodeList.hpp`
`IXPathResults`	Liste von XML Knoten, die durch X-Pfad gefunden wurden. `zeusbase/System/Interfaces/IXPathResults.hpp`
`IXMLErrorReporter`	Fehlerreporter zum Empfangen von Parser und Validator Fehler. `zeusbase/System/Interfaces/IXMLErrorReporter.hpp`
`IXSLProcessor`	XSLT Prozessor zum Umwandeln von XML Daten. `zeusbase/System/Interfaces/IXSLProcessor.hpp`

3 In dieser Version von Zeus-Framework wird der Parser von Apache verwendet. [XML02]

3.8.1 Verwalten der XML-Dateien mit TXMLFile

Die Klasse `TXMLFile` bietet eine umfangreiche Schnittstelle zum Verwalten und Verändern von XML-Dateien.

API	`zeusbase/System/XMLFile.h`
	`Retval loadXML(NAMESPACE_Zeus::IXMLDocument*& rpDocument);` Laden eines XML Baums aus der Datei.
	`Retval loadXObjects(IXObject*& rpObject,  bool dDoUnfreeze = true, bool bKeepXObjectAfterClosing = false);` Laden des X-Objektbaums aus der Datei.
	`void MQUALIFIER addErrorReporter(IXMLErrorReporter& rReporter);` Fügt ein XML Fehlerreporter hinzu. Beim Einlesen und Erstellen des XML Baums können Fehler auftreten. Diese Fehler werden durch diesen Reporter bekannt gegeben.

	`Retval createXMLFile(const IString& rMainNodeName);`
	`Retval createXMLFileFromStream(const IString& rStream);`
	`...`
	
	Erstellen einer neuen XML Struktur. Es gibt weitere Methoden zum Erstellen von X-Objektbaumen.

	`Retval selectNode(const IString& rXPath, IXMLNode*& rpNode);`
	`Retval selectNode(const wchar_t* pXPath, IXMLNode*& rpNode);`
	
	Selektieren eines XML Knotens mittels X-Pfad. es gibt weitere Methoden zum Selektieren von mehreren XML-Knöten gleichzeitig (XPathResults) und von X-Objekten.

	`Retval writeNodeValue(const IString& rXPath, const IString& rValue);`
	`Retval writeNodeValue(const wchar_t* pXPath, const wchar_t* pValue);`
	
	Schreiben eines Knoten-Wertes, durch Selektieren des Knotens mittels X-Pfad. Damit können sowohl XML Attribute als auch Inhalte von anderen Knoten-Typen geschrieben werden.

	`Retval save();`
	`Retval saveAs(const IString& rFileName, bool bKeepOrginalFileName);`
	
	Speichern der XML Daten. Mit `bKeepOrginalFileName=true` beim saveAs() bleibt der Namen der XMLDatei erhalten. Es wird lediglich eine Kopie angefertigt.

	`void closeXML();` Die XML Struktur wird freigegeben.

Verwendung der XML-Datei

```
#include <zeusbase/System/XMLFile.h>

TString strFileName(L"./Testfile.xml");

TAutoPtr<TXMLFile> ptrFile = new TXMLFile(strFileName);
TAutoPtr<IXMLNode> ptrNode;
if (ptrFile->selectNode(L"./MainNode/Data[@ID='3']",
                        ptrNode) == RET_NOERROR)
{
  ...
}
```

Tabelle 33: Lesen aus einer XML Datei.

3.8.2 Verwalten der XML-Streams mit TXMLStream

DIe TXMLStream Klasse besitzt prinzipiell die gleiche Klassenschnittstelle wie das TXMLFile.

Verwendung der XML-Streams

```
#include <zeusbase/System/XMLFile.h>

TString strStream;

//get the stream

TAutoPtr<TXMLStream> ptrStream = new TXMLStream(strStream);
TAutoPtr<IXMLNode> ptrNode;
if (ptrStream->selectNode(L"./MainNode/Data[@ID='3']",
                          ptrNode) == RET_NOERROR)
{
  ...
}
```

Tabelle 34: Lesen aus einem XML Stream.

3.8.3 Parsen von XML durch XML_Service

Etwas umständlicher als mit der `TXMLFile`-Klasse kann eine XML Datei auch direkt durch die Service-Bibliothek `XML_Service` verwaltet werden.

Damit XML verwendet werden kann, muss bei der Bibliothek `XML_Service` die exportierte Funktion `createIXMLParser()` aufgerufen werden. Sie gibt ein Parser-Objekt zurück, welches Streams oder Dateien in einen DOM Baum wandelt.

Die exportierte Methode kann mit dem Library-Manager aufgerufen werden (siehe unten und im Kapitel Singletons).

Laden des XMLParsers

```
#include <zeusbase/System/LibraryManager.h>
#include <zeusbase/System/Interfaces/IXMLParser.hpp>

TAutoPtr<IXMLParser> ptrParser;
if (LibraryManager.createObject(TString(L"XML_Service"),
                                TString(L"IXMLParser"),
                                ptrParser.getInterfaceReference())
                                ==RET_NOERROR)
{
  ...
}
```

Tabelle 35: Erstellen eines XML-Parsers mit dem Library-Manager. Da die XML Parser Implementation sich nicht direkt im Zeus-Framework befindet, sondern in der XML_Service Bibliothek, muss eine exportierte Methode zum Erzeugen des Parser-Objekts aufgerufen werden.

Die Schnittstelle IXMLParser stellt folgende Methoden zur Verfügung:

API	`zeusbase/System/Interfaces/IXMLParser.hpp`
	`Retval MQUALIFIER loadFromFile(const IString& rFileName, IXMLDocument** rpDocu);` Laden eines XML Baums aus einer Datei.
	`Retval MQUALIFIER parseStream(const IString& rStream, IXMLDocument** ppDocu)` Laden eines XML Baums aus einem Text-Stream (String).
	`void MQUALIFIER addErrorReporter(IXMLErrorReporter& rReporter);` Fügt ein XML Fehlerreporter hinzu. Beim Einlesen und Erstellen des XML Baums können Fehler auftreten. Diese Fehler werden durch diesen Reporter bekannt gegeben.

3.8.4 Transformieren vom XML Daten

Damit der XSL-Transformator verwendet werden kann, muss bei der Bibliothek `XML_Service` die exportierte Funktion `createIXSLProcessor()` aufgerufen werden. Sie gibt ein XSL Prozessor-Objekt zurück.

Die exportierte Methode kann mit dem Library-Manager aufgerufen werden (siehe Kapitel Singletons).

Laden des XSLProcessor

```
#include <zeusbase/System/LibraryManager.h>
#include <zeusbase/System/Interfaces/IXSLProcessor.hpp>

TAutoPtr<IXSLProcessor> ptrProc;
if (LibraryManager.createObject(TString(L"XML_Service"),
                                TString(L"IXSLProcessor"),
                                ptrProc.getInterfaceReference())
                                ==RET_NOERROR)
{
  ...
}
```

Tabelle 36: Erstellen eines XSL-Prozessors mit dem Library-Manager.

Die Schnittstelle IXSLProcessor stellt folgende Methoden zur Verfügung:

API `zeusbase/System/Interfaces/IXSLProcessor.hpp`

```
Retval MQUALIFIER transformFile(const IString& rFileName,
                                const IString& rRuleFile,
                                IByteArray& rOutput)
```

Eine XML-Datei nach einem Stylesheet (rRuleFile) transformieren. Diese Methode gibt die transformierten Daten als Byte-Array zurück.

```
Retval MQUALIFIER transformBuffer(const IString& rBuffer,
                                  const IString& rRuleFile,
                                  IByteArray& rOutput)
```

Funktioniert wie [`transformFile`]. Transformiert wird ein XML Stream (`rBuffer`).

```
Retval MQUALIFIER transformBuffer2(const IString& rBuffer,
                                   const IString& rRuleBuffer,
                                   IByteArray& rOutput)
```

Die Transformation findet nur mit Buffer statt.

3.8.5 Das XML Dokument

Die Schnittstelle IXMLDocument repräsentiert ein geladenes XML Dokument.

API `zeusbase/System/Interfaces/IXMLDocument.hpp`

```
Retval MQUALIFIER getMainNode(IXMLNode*& rpMain);
```

Ermittelt den Hauptknoten des XML Baums.

```
Retval MQUALIFIER save();
```

Speichert den XML Baum zurück in die Datei.

API	`zeusbase/System/Interfaces/IXMLDocument.hpp`

```
Retval MQUALIFIER saveAs(const IString& rFileName);
```

Speichert den XML Baum zurück in eine Datei mit dem Namen [`rFileName`].

Wichtig:
Das XML-Dokument-Objekt sollte erst am Ende der XML Modifikation freigegeben werden. Existieren noch XML-Knoten im System, kann das Freigeben zu Problemen führen (nachgewiesen bei Apache Xerces-Parser).

3.9 MOM 2 Implementation

Dieses Kapitel befasst sich mit den Klassen und der Funktionsweise des Module Object Models. Die Definition finden sie im Kapitel Module Object Model Spezifikation 2.0.

3.9.1 Erstellen des X-Objektbaums

Die MOM Struktur wird in dieser Implementation als XML-Baum eingelesen und dynamisch zu einem Objektbaum aufgebaut. Dabei entspricht der XML Baum dem zu erstellenden Objektbaum.

Das Einlesen des XML's und erstellen des Objektbaums wird durch die X-Objekt-Factory wahrgenommen (siehe Kapitel zu Singletons).

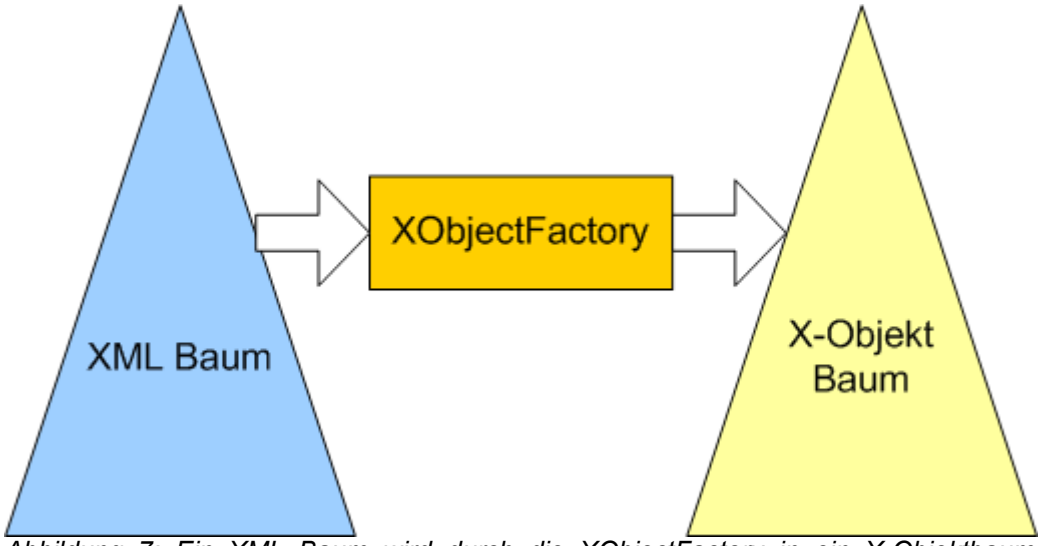


Abbildung 7: Ein XML Baum wird durch die XObjectFactory in ein X-Objektbaum umgewandelt.

Sie versucht aus dem XML Baum von jedem Knoten ein X-Objekt zu erzeugen und so den Objektbaum aufzubauen. Jeder XML Knoten muss deshalb spezifizieren, welche Klasse instanziert werden soll, damit die Fabrik das entsprechende Objekt erzeugen und am richtigen Ort im Baum platzieren kann. Die XML Attribute sind bei den jeweiligen Klassen beschrieben.

Folgende Graphik zeigt den Ablauf beim Erstellen des Objektbaums.

1. Analysieren der XML Daten
2. Wenn nötig ein Code Module in den Speicher laden
3. Erzeugen einfügen des X-Objekts in den X-Objektbaum.

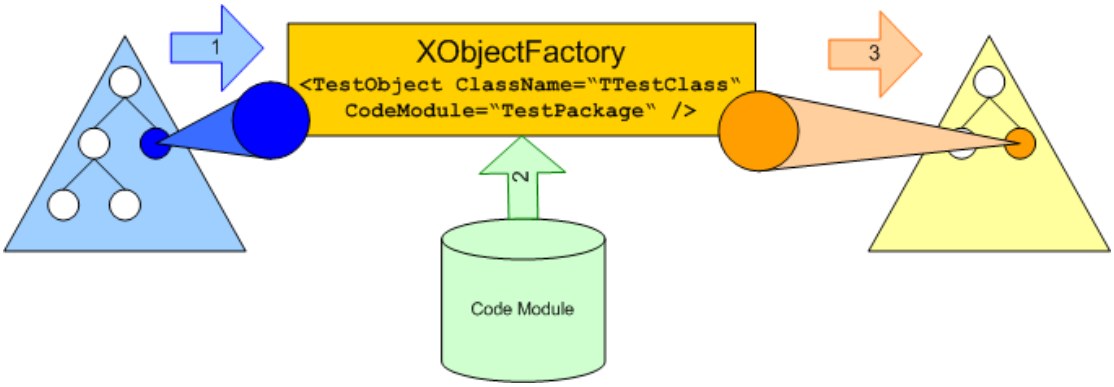


Abbildung 8: Schematische Darstellung der Funktionsweise einer XObjectFactory

Die XObjectFactory stellt folgende Methoden zur Verfügung:

API	zeusbase/System/XObjectFactory.h

	`Retval MQUALIFIER createObjectFromStream(`
	`  const IString& rStream,`
	`  IXObject*& rpObj,`
	`  bool bOnlyRoot = true,`
	`  bool bDoUnfreeze = true);`
	
	Erstellt ein X-Objektbaum aus einem XML Stream. Ist das `bDoUnfreeze`-Flag gesetzt, wird der Baum aktiviert (unfrozen).

	`Retval MQUALIFIER createObjectFromFile(`
	`  const IString& rFileName,`
	`  IXObject*& rpObj,`
	`  bool bOnlyRoot = true,`
	`  bool bDoUnfreeze = true);`
	
	Erstellt ein X-Objectbaum aus einer XML Datei.

	`Retval MQUALIFIER addNewChildObjects(`
	`  IXMLNode& rNewNode,`
	`  IXObject& rParent,`
	`  IXObject*& rpChild,`
	`  bool bDoUnfreeze = true,`
	`  IXObject* pRightSibling = NULL);`
	
	Erstellt aus einem XML Knoten (`rNewNode`) einen X-Objektbaum. Dieser Objektbaum wird dem Vaterobjekt (`rParent`) als Kindobjekt hinzugefügt. Mit dem Parameter `pRightSibling` kann die Einfügeposition definiert werden.

	`virtual void MQUALIFIER setErrorObjectExpandProperty(bool bValue);` Wird die Eigenschaft gesetzt, werden alle Objekte deren Klassen nicht gefunden werden konnten, als Fehler-Objekt erzeugt. Standardeinstellung ist deaktiviert.

Erstellen eines X-Objektbaums

```
#include <zeusbase/System/XObjectFactory.h>

TString strFile = L"./Test.xml";
TAutoPtr<IXObject> ptrRootObject;
if (MyXObjectFactory.createObjectFromFile(strFile, ptrRootObject)
    == RET_NOERROR)
{
  //Work with the root x-object
  //...

}
```

Tabelle 37: Erstellen eines X-Objektbaums mit der XObjectFactory.

3.9.2 X-Objekt Klassen

Als X-Objekt wird jedes Objekt bezeichnet, welches durch die XObjectFactory erzeugt werden kann. Die X-Objekte sind immer XML basierend und werden ausschliesslich von der XObjectFactory durch eine XML Datei oder einen XML-Stream erzeugt.

Ein XML Knoten kann Kinderknoten besitzen. Die Kinderknoten können auch wieder X-Objekte darstellen. Diese X-Objekte werden automatisch dem übergeordneten X-Objekt als Kinderobjekte zugewiesen.

Folgende Schnittstellen und Grundtypen sind definiert:

API	
`IXObject`	Basisschnittstelle eines X-Objekts `zeusbase/System/Interfaces/IXObject.hpp`
`IXLoaderObject`	Schnittstelle für ein ladbares X-Objekt `zeusbase/System/Interfaces/IXLoaderObject.hpp`
`TXObject`	Implementation eines X-Objekts. Andere X-Objekte können von dieser Klasse erben. `zeusbase/System/XObject.h`
`TXRootObject`	Implementation eines Wurzelobjekts. Andere Wurzelobjekte können von dieser Klasse erben. `zeusbase/System/XRootObject.h`
`TXLoaderObject`	Implementation eines ladbares X-Objekts. `zeusbase/System/XLoaderObject.h`
`IXErrorObject`	Schnittstelle für Fehler-Objekte im X-Objektbaum `zeusbase/System/Interfaces/IXErrorObject.hpp`

Diese Klassen implementieren die Basisanforderungen der MOM Spezifikation. Sie dienen weiter als Persistenz-Layer. Beim Starten der Applikation können X-Objekte ihre Einstellungen (Attribute und Variabeln) aus der XML Struktur lesen und beim Beenden können sie diese wieder in die XML Struktur speichern. Diese XML Struktur kann in einer Datenbank (XML fähige Datenbank) oder als XML-Datei gespeichert werden.

Weitere Informationen zur XML Definition von X-Objekten siehe Kapitel Definition von X-Objekten in XML.

3.9.3 Die TXObject Klasse

Das `TXObject` ist die Basisklasse aller X-Objekte. Sie verwaltet eine Liste von

Kinderobjekten, welche ebenfalls vom Typ `TXObject` sind. Durch diese rekursive Formulierung wird der Objektbaum definiert.

3.9.3.1 Variablen und Attribute

Die Attribute und Variablen eines X-Objekts können durch entsprechende Methoden gelesen und geschrieben werden (Siehe Tabelle). Zwischen Attributen und Variablen eines X-Objekt gibt es folgende Unterschiede:

- Typisierung
  - Attribute sind untypisiert und werden als Zeichenkette gespeichert. Es sind nur einzeilige Texte zugelassen. Eine Typen-Konvertierung erfolgt durch die TXObject-Klasse
  - Variablen sind typisiert. Es gibt die Typen string, int, uint, float, ref und xml.
- XML Repräsentation
  - Attribute werden auch im XML als XML Attribut zum Objekt gespeichert.
  - Variablen werden als Unterknoten mit dem Namen `XMember` gespeichert.
- Neue Instanzen
  - Nicht vorhandene Attribute werden beim Setzen dessen Werts automatisch erzeugt.
  - Es können nur existierende Variablen gelesen und geschrieben werden.
- Zugriffszeit
  - Der Zugriff auf die Attribute ist direkter und dadurch ein wenig effizienter
  - Der Zugriff auf Variablen-Inhalt dauert etwas länger, da im XML Baum 2 Stufen tiefer navigiert wird.

Methoden für den Attribut- und Variable-Zugriff:

API	zeusbase/System/Interfaces/IXObject.hpp

```
virtual Retval MQUALIFIER readIntAttribute(const IString& rName, Int& rValue,
Int iDefault = 0) const;

virtual Retval MQUALIFIER readUintAttribute(const IString& rName, Uint&
rValue, Uint uiDefault = 0) const;

virtual Retval MQUALIFIER readFloatAttribute(const IString& rName, Float&
rValue, Float fDefault = 0) const;

virtual Retval MQUALIFIER readStringAttribute(const IString& rName, IString&
rValue) const;
```

Lesen eines Attributs als Integer/ unsigned Integer/ reellen oder Zeichenketten-Wert

```
virtual Retval MQUALIFIER storeIntAttribute(const IString& rName, const Int&
rValue);

virtual Retval MQUALIFIER storeUintAttribute(const IString& rName, const
Uint& rValue);

virtual Retval MQUALIFIER storeFloatAttribute(const IString& rName, const
Float& rValue);

virtual Retval MQUALIFIER storeStringAttribute(const IString& rName, const
IString& rValue);
```

Schreiben eines Attributs als Integer/ unsigned Integer/ reellen oder Zeichenketten-Wert

```
virtual Retval MQUALIFIER readMemberVariable(const IString& rName, IZVariant&
rValue) const;
```

Lesen einer Variable. Die Variable ist typisiert und wird als Variant zurückgegeben.

```
virtual Retval MQUALIFIER storeMemberVariable(const IString& rName, const
IZVariant& rValue);
```

Schreiben einer Variable. Die Variable muss typisiert als Variant übergeben werden.

3.9.3.2 Objekt-Zustände

Die `TXObjekt` Klasse implementiert zwei Zustände:

- eingefroren (frozen)
- aktiv (unfrozen)

Diese Zustände können durch zwei Methoden gesteuert werden:

API	zeusbase/System/Interfaces/IXObject.hpp
	`bool MQUALIFIER unfreeze();` Das X-Objekt und seine Kinderobjekte werden zum Leben erweckt.
	`bool MQUALIFIER freeze();` Das X-Objekt und seine Kinderobjekte werden eingefroren.

Das Konzept der X-Objekte sieht vor, dass jede abgeleitete Klasse diese Methoden überschreibt. Bei `unfreeze()` können diese Klassen andere X-Objekte referenzieren. Bei `freeze()` hingegen sollten die Klassen alle externen Referenzen (auch andere X-Objekte) wieder freigeben.

3.9.3.3 Navigation im Objektbaum

`TXObject` bietet eine Fülle von Methoden, die zum Abfragen und Verändern des Objektbaums dienen. Diese Methoden sind ausnahmslos in der Schnittstelle definiert:

API	zeusbase/System/Interfaces/IXObject.hpp
	`void MQUALIFIER getName(IString& rName)const;` Ermittelt den Objektnamen. Dieser Name entspricht dem XML Knotennamen.
	`void MQUALIFIER getClassName(IString& rName)const;` Gibt den Klassennamen zurück. (Attribute `ClassName`)
	`Uint MQUALIFIER getID()const;` Diese Methode gibt das Attribut ID zurück.
	`Retval MQUALIFIER addChild(IXObject& rChild);` Fügt zur Laufzeit ein X-Objekt hinzu. Der entsprechende XML Knoten wird in den XML Baum eingefügt.
	`Retval MQUALIFIER removeChild(IXObject& rChild);` Entfernt ein X-Objekt aus dem X-Objektbaum. Der XML Knoten wird ebenfalls aus dem XML Baum entfernt.
	`Int MQUALIFIER getChildCount()const;` Gibt die Anzahl Kinderobjekte zurück.
	`Retval MQUALIFIER getChild(Int iIndex, IXObject*& rpChild);` Gibt ein indexiertes Kindobjekt zurück.
	`Retval MQUALIFIER getParent(IXObject*& rpParent)const;` Gibt das Vaterobjekt zurück.

API	zeusbase/System/Interfaces/IXObject.hpp
	`Retval MQUALIFIER getObject(const IString& rPath, IXObject*& rpObj);` Gibt ein X-Objekt zurück. Das Objekt wird nach dem Konzept des X-Objektpfad gesucht.
	`Retval MQUALIFIER getObjects(const IString& path, IXObjectCollection*& rpObjects);` Gibt eine Menge von X-Objekten zurück. Die Objekte wurden nach dem Konzept des X-Objektpfad gesucht.
	`Retval MQUALIFIER getRootObject(IXObject*& rpObj);` Diese Methode gibt das Wurzelobjekt zurück.

3.9.3.4 Callbacks

Hinzu kommen weitere Methoden, die in der TXObject Klasse zum Überschreiben definiert wurden:

API	zeusbase/System/XObject.h
	`virtual void onChildAdded(IXObject& rChild);` Call back Methode wird aufgerufen, wenn ein Kindobjekt hinzugefügt wird. Das Hinzufügen kann durch die XObjectFactory oder durch den Aufruf von addChild() erfolgen.
	`virtual void onChildDeleted(IXObject& rChild);` Call back Methode wird aufgerufen, wenn ein Kindobjekt entfernt wird. Das Entfernen kann durch die XObjectFactory oder den Aufruf von removeChild() erfolgen.

3.9.3.5 Definierte XML Attribute

Bei der XML Konfiguration sind folgende Attribute massgebend:

API	
`ClassName`	Name der Klasse (Muss)
`CodeModule`	Name des Code Moduls (Bibliothek in welcher die Klasse implementiert ist). Leer bedeutet, dass die Klasse im Framework definiert ist. (Muss)
`ID`	ID des Objekts (Optional)

3.9.4 Die TXRootObject Klasse

Die `TXRootObject` Klasse erweitert die allgemeine `TXObject` Klasse. Sie definiert das Wurzelobjekt eines X-Objektbaums und verwaltet das geladene XML Dokument. Folgende Methoden sind hier zusätzlich definiert:

API	`zeusbase/System/XRootObject.h`
	`Retval save();` Speichert den gesamten XML Baum in die Originaldatei.
	`Retval saveAs(const TString& rFileName);` Speichert den gesamten XML Baum in einen neue Datei.

3.9.5 Die TXLoaderObject Klasse

Diese Klasse definiert ein Objekt, welches folgende Funktionalität besitzt:

- Kann verhindern, dass beim Aufbauen des Objektbaums ein Unterbaum erstellt wird. Dieser Unterbaum wird später erst bei Gebrauch erstellt.
- Kann ein Unterbaum aus einem anderen XML Dokument laden. Dies ermöglicht eine Modularisierung der XML Dokumente

Objekte dieses Typs besitzen zwei weitere Zustände:

- nicht geladen. Noch keine Kinderobjekte erstellt.
- geladen. Kinderobjekte sind erstellt.

Diese neuen Zustände können durch folgende Methoden gesteuert werden:

API	`zeusbase/System/Interfaces/IXLoaderObject.hpp`
	`Retval MQUALIFIER load();` unmittelbares Laden und Erstellen der Kinderobjekte. Diese Methode wird implizit bei dem vermeintlichen Ermitteln von Kinderobjekten aufgerufen, sofern der Zustand [`not loaded`] ist.
	`Retval MQUALIFIER unload();` Entladen aller Kinderobjekte.

3.9.5.1 Definierte XML Attribute

Bei der XML Konfiguration sind folgende Attribute zusätzlich erforderlich:

API	
`XMLRef`	Name der externen XML Datei (Teilbaum). Die XML Datei muss sich im Pfad der Module befinden.
`CreateChildren`	Hat dieses Attribut den Wert „0“, werden die Kinderobjekte beim Laden des X-Objektbaums noch nicht erstellt.

3.9.6 Implementation eines X-Objekts

Damit die X-Objekte von der XObjectFactory erzeugt werden können, müssen diese registriert werden. Ein X-Objekt kann auf zwei Arten implementiert werden:

- In einem Code Module
- Direkt in der Applikation

Im Zeus-Framework 0.3 ist die Registrierung der beiden Implementationen verschieden.

3.9.6.1 X-Objekt Implementation in einem Code-Module

Dieser Abschnitt befasst sich mit X-Objekt Implementationen, die in einem Code-Module (Bibliothek) erfolgt.

Die X-Objekte können in so genannten Code-Modulen (Bibliotheken) entwickelt und für eine Anwendung bereitgestellt werden. Damit die Objektfabrik die Klasse eines X-Objekts auch findet und ein Objekt instanziern kann, muss die Klasse entweder manuell registriert werden (siehe nächstes Kapitel), oder sie wird durch eine Fabrikfunktion in einem Code Modul exportiert. Informationen zur Programmierung von externen Code Modulen sind im Kapitel [Entwickeln von Code-Modulen] zu finden.

Jede Bibliothek kann beliebige von `TXObject` abgeleitete Klassen exportieren. Der Export geschieht durch vorgefertigte Makros. Die Makros sind in der Datei `zeusbase/MOM/AbstractModuleSession.h` definiert.

API `MExportXRootObjectFactory(classid);`
Exportiert ein von `TXRootObject` abgeleitete Klasse.

API	`MExportXObjectFactory(classid);`
	Exportiert ein von `TXObject` abgeleitete Klasse.

Für X-Objekte in einem speziellen Namensraum (Namespace), wie zum Beispiel bei Inneren Klassen, gibt es folgende Makros:

API	`MExportXRootObjectFactoryNS(__namespace, classid);`
	Exportiert ein von `TXRootObject` abgeleitete Klasse, die aus einem speziellen Namensraum erzeugt werden muss.

API	`MExportXObjectFactory(__namespace, classid);`
	Exportiert ein von `TXObject` abgeleitete Klasse, die aus einem speziellen Namensraum erzeugt werden muss.

Implementation in einem CodeModule

```
//----------------------------------------
//Header
#include <zeusbase/System/XObject.h>

BEGIN_NAMESPACE_Zeus

class TXExample : public TXObject
{
  public:
    TXExample(IXMLNode& rNode);
...

  protected:
    virtual ~TXExample();

  private:
...
};

END_NAMESPACE_Zeus

//----------------------------------------
//Implementation
#include <XExample.h>

USING_NAMESPACE_Zeus

TXExample::TXExample(IXMLNode& rNode) : TXObject(rNode)
{
  ...
}

TXExample::~TXExample()
{
  ....
}


//Export
MExportXObjectFactory(TXExample);
```

Tabelle 38: Header Datei für das X-Objekt TInputCell.

3.9.6.2 X-Objekt Implementationen in einer Applikation

Zum Erzeugen von X-Objekten innerhalb einer Applikation (ohne Code-Module), muss in der Version 0.3.x folgendermassen vorgegangen werden.

Das X-Objekt muss vor dessen Verwendung manuell bei der Objektfabrik registriert werden. Zur Registrierung wurden ebenfalls Makros angefertigt. Die Makros sind in der Datei `zeusbase/System/Interfaces/IXObjectImplHelper.hpp` definiert.

API	`REG_SUB_BEGIN(classid);`
	Dieses Makro startet die Definition der X-Objekt Registrierung. Als Parameter muss die ID der Klasse angegeben werden.

API	`REG_SUB_ADD(classid, classname);`
	Dieses Makro dient dem Zuordnen eines Klassennamens zu der ID der Klasse. Ein X-Objekt kann auch mehrere Klassennamen besitzen (im XML), welche die gleiche ID referenzieren.

API	`REG_SUB_END;`
	Dieses Makro beendet die Definition.

API	`REG_ROOT_BEGIN(classid);`
	Dieses Makro startet die Definition der X-Wurzelobjekt Registrierung. Als Parameter muss die ID der Klasse angegeben werden.

API	`REG_ROOT_ADD(classid, classname);`
	Dieses Makro dient dem Zuordnen eines Klassennamens zu der ID der Klasse. Ein X-Objekt kann auch mehrere Klassennamen besitzen (im XML), welche die gleiche ID referenzieren.

API	`REG_ROOT_END;`
	Dieses Makro beendet die Definition.

Das direkte Registrieren bei der Objektfabrik wird durch ein entsprechendes Makro getätigt. Dieses Makro muss vor dem Laden des Objektbaums ausgeführt werden.

API	`XOBJECTFACTORY_REGISTER_ROOT(classid);`
	Dieses Makro registriert ein Wurzelobjekt bei der Objektfabrik.

API	`XOBJECTFACTORY_REGISTER_SUB(classid);`
	Dieses Makro registriert ein X-Objekt bei der Objektfabrik.

Es gibt auch Makros zum Abmelden von X-Objekten

API	`XOBJECTFACTORY_UNREGISTER_ROOT(classid);`
	Dieses Makro meldet ein Wurzelobjekt bei der Objektfabrik ab.

API	`XOBJECTFACTORY_UNREGISTER_SUB(classid);`
	Dieses Makro meldet ein X-Objekt bei der Objektfabrik ab.

Das folgende Beispiel zeigt eine Definition der X-Objektklasse `TInputCell`. Eine Wurzelobjekt wird mit den entsprechenden Makros analog definiert.

InputCell.h

```
#include <zeusbase/CellModel/AbstractCell.h>
#include <zeusbase/System/XObjectFactory.h>

BEGIN_NAMESPACE_Zeus

class TInputCell : public TAbstractCell
{
  public:
    TInputCell(IXMLNode& rNode);

    REG_SUB_BEGIN(TInputCell)
    REG_SUB_ADD(TInputCell, L"TInputCell");
    REG_SUB_END
...

  protected:
    virtual ~TInputCell();

  private:
...
};

END_NAMESPACE_Zeus
```

Tabelle 39: Header Datei für das X-Objekt TInputCell. Diese Klasse kann manuell bei der Objektfabrik registriert werden.

Manuelles Registrieren

```
#include <InputCell.h>


USING_NAMESPACE_Zeus

int main(int argc, char* argv[])
{
  //Dieser Aufruf registriert die Klassen TXObject und
  // TXRootObject
  XOBJECTFACTORY_REGISTER_ROOT(TXObjectFactory);

  //registrieren benutzerspezifischer Klassen
  XOBJECTFACTORY_REGISTER_SUB(TInputCell);

  ...

  return 0;
}
```

Tabelle 40: Manuelles Registrieren der X-Objekt-Klassen muss vor deren Verwendung geschehen.

3.9.7 Module Object Model Klassen

Die erweiterten Eigenschaften des MOM 2 wurden durch die Klassen `TModule` und `TSystemManager` implementiert.

API	
`IModule`	Basisschnittstelle aller Module und Manager. `zeusbase/MOM/Interfaces/IModule.hpp`
`TModule`	Implementation des Moduls und des Managers. Das Modul kann auch die Aufgabe des Managers übernehmen. `zeusbase/MOM/Module.h`
`TSystemManager`	Implementation eines Wurzelobjekts in Form eines Moduls. `zeusbase/MOM/SystemManager.h`

3.9.7.1 Die TModule Klasse

Die Basisklasse des Moduls heisst `TModule`. Sie implementiert die Schnittstelle `IModule`, welches das Aktivieren und deaktivieren von Modulen spezifiziert, und leitet sich vom `TXLoaderObject` ab.

Um die zwei neuen Zustände anzusteuern, gibt es folgende Methoden:

API	zeusbase/MOM/Interfaces/IModule.hpp
	`virtual void MQUALIFIER activate();` Aktiviert ein Modul.
	`virtual void MQUALIFIER deactivate();` Deaktiviert das Modul.

Wichtig:
Damit das Aktivieren, bzw. Deaktivieren von Module restlos funktioniert, muss darauf geachtet werden, dass die Objekthierarchie geschlossen aus Objekten besteht, welche den Typ `IModule` vererben. Alle anderen Objekte können nicht aktiviert, bzw. deaktiviert werden.

Weitere Methoden dienen der Implementation von Modul-Sessionen:

API	zeusbase/MOM/Interfaces/IModule.hpp
	`Retval MQUALIFIER getZeusAPI(IZeusAPI*& rpApi);` Gibt die Framework Schnittstelle eines Moduls zurück.
	`Retval MQUALIFIER getModuleSession(IModuleSession*& rpSession);` Gibt die Session des Moduls zurück (Schnittstelle der Modul-Bibliothek).

3.9.7.2 Die TSystemManager Klasse

Die Klasse `TSystemManager` ist vom Typ `TXRootObject` abgeleitet und implementiert ebenfalls die Schnittstelle `IModule`. Der System-Manager ist das Wurzelobjekt im Modul-Objektbaum.

3.9.8 Der Objektpfad

Der Objektpfad dient dem Suchen und Ermitteln von X-Objekten in einem geladenen Objektbaum. Der Pfad ist aufgebaut wie ein Verzeichnispfad bei Unix. Er enthält folgende Elemente:

API	
/	Repräsentiert das Wurzelobjekt

API	
`./`	Referenziert das aktuelle Objekt (this)
`../`	Referenziert das Vaterobjekt
`./MyChild`	Referenziert alle Kinderobjekte mit dem Namen `MyChild`
`./MyChild[2]`	Referenziert das dritte Kindobjekt mit dem Namen `MyChild`
`//MyObject`	Sucht rekursiv alle X-Objekte mit dem Namen `MyObject`

Diese Elemente des Pfads können nach belieben zusammengesetzt werden.

3.9.9 Übersicht der X-Objekte

Die folgende Abbildung zeigt die Vererbung der X-Objekte im Zeus-Framework.

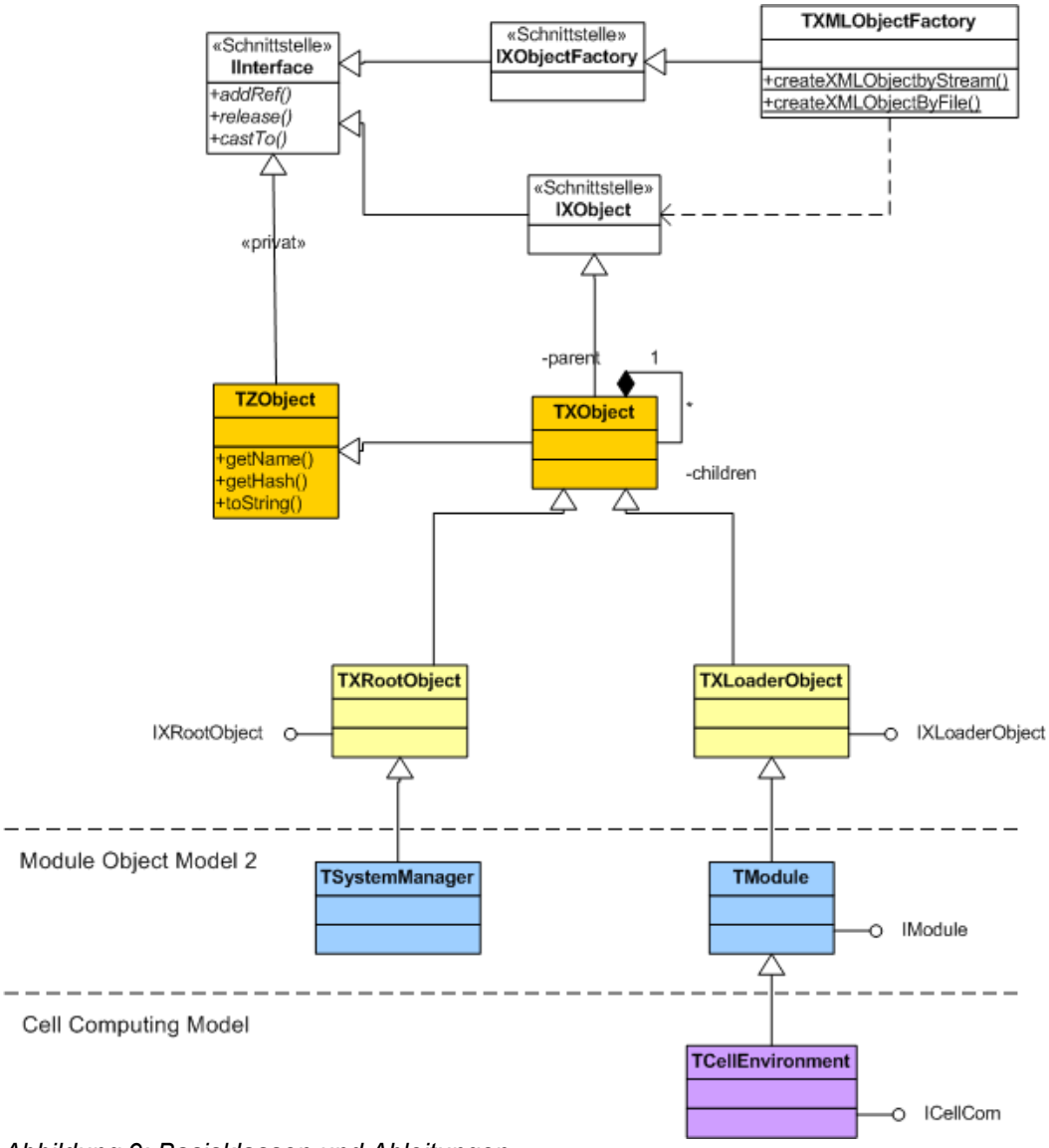


Abbildung 9: Basisklassen und Ableitungen

3.9.10 X-Objektbäume synchronisieren

Das X-Objekt Modell beinhaltet einen Synchronisationsprozess (Merging) zum Abgleichen von X-Objektbäumen. Dieser Prozess wird zum Beispiel gebraucht, um aus zwei Konfigurationen die nötigen Schritte zu berechnen, damit eine Konfiguration in die zweite überführt werden kann. Mit anderen Worten, der erste Baum dient als Basis, der zweite Baum wird anhand dieser Basis so angepasst, dass er am Ende identisch ist.

Der Prozess erfolgt in 2 Schritten:

1. Erstellen eines Differenz-Baums
2. Abgleichen und bereinigen der Differenzen

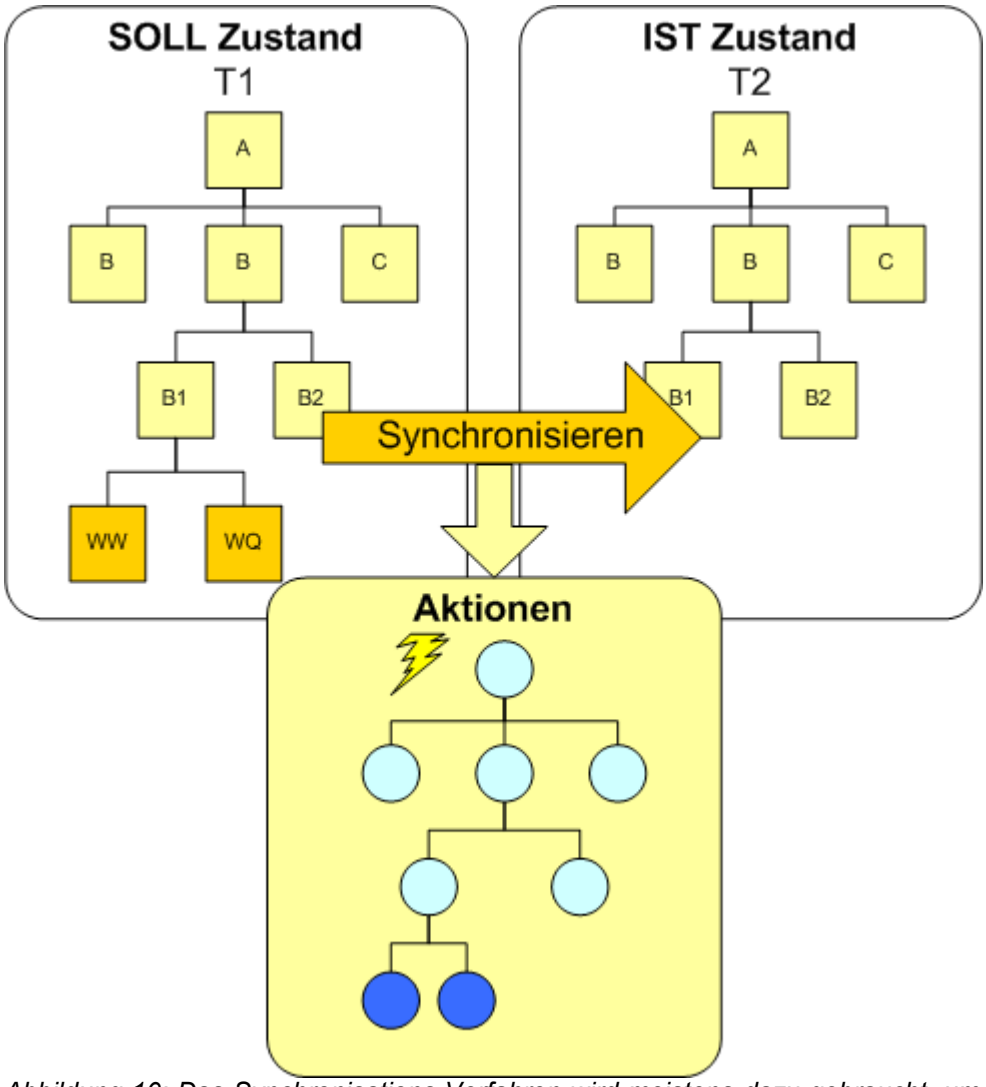


Abbildung 10: Das Synchronisations-Verfahren wird meistens dazu gebraucht, um einen SOLL-Zustand in den IST-Zustand zu versetzen.

Der Prozess startet mit zwei erstellten X-Objektbäumen `T1` und `T2`. Dabei werden die einzelnen Objekte und deren Kinderobjekte miteinander verglichen. Wird ein Unterschied festgestellt, wird eine entsprechende Aktion erstellt, durch welche der Unterschied korrigiert werden kann. Da die Basis-Struktur ein Baum ist, entsteht beim Vergleichen der Objekte erneut ein Baum, der Differenzbaum aus Aktionen.

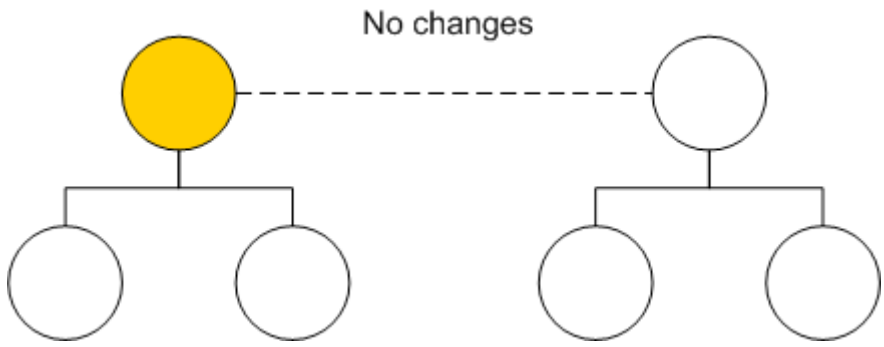
Folgende Klassen stehen zur Verfügung:

API	
`TXObjectTreeSynchronizer`	Diese Klasse dient als erster Schritt im Synchronisations-Prozess und erstellt den Differenz-Baum. `zeusbase/System/XObjectTreeSynchronizer.h`
`IXSynchronAction`	Schnittstelle einer Aktion `zeusbase/System/Interfaces/ IXSynchronAction.hpp`
`TXSynchronAction`	Diese Klasse implementiert eine Aktion. `zeusbase/System/XSynchronAction.h`

3.9.10.1 Aktionen

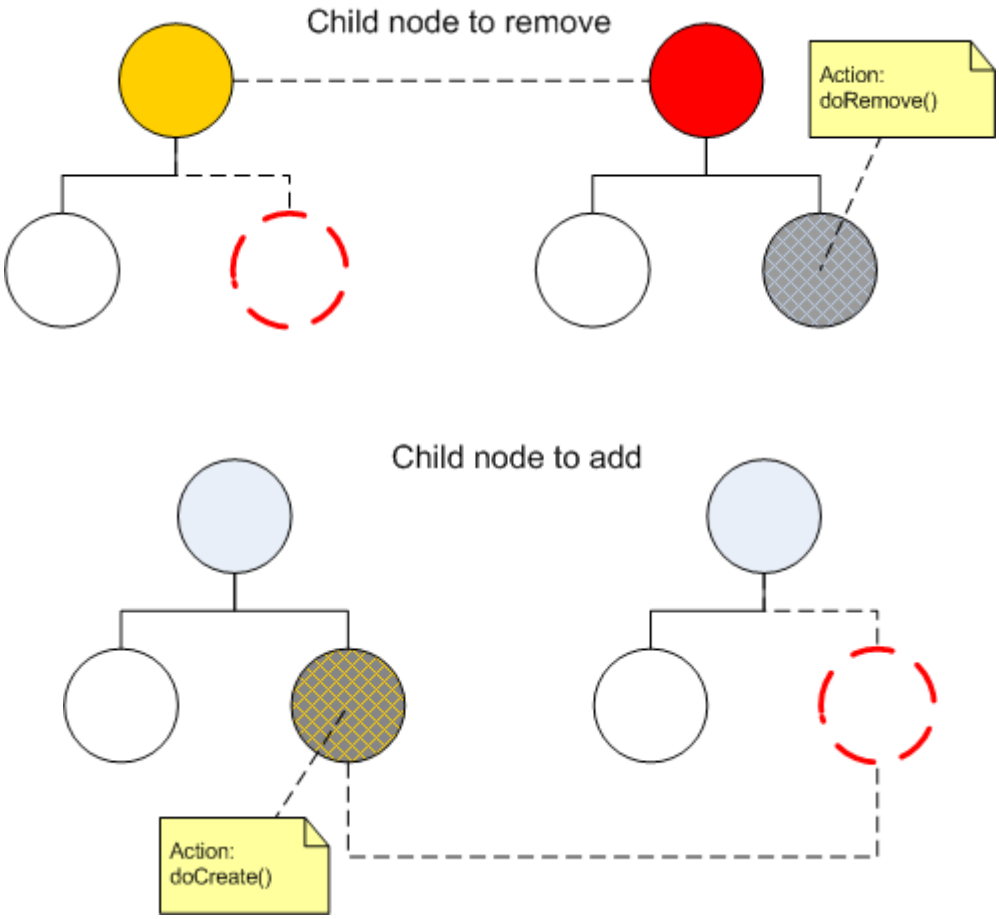
Der Differenzbaum besteht aus Aktionen zu jedem X-Objekt Paar $(T1::O_x, T2::O_{x'})$. Je nach Art des Abgleichs entstehen folgende Aktionen:

- Sind die zwei Objekte O_x und $O_{x'}$ gleich, wird eine Aktion erzeugt, die keine Veränderung am zweiten Baum `T2` vornimmt.

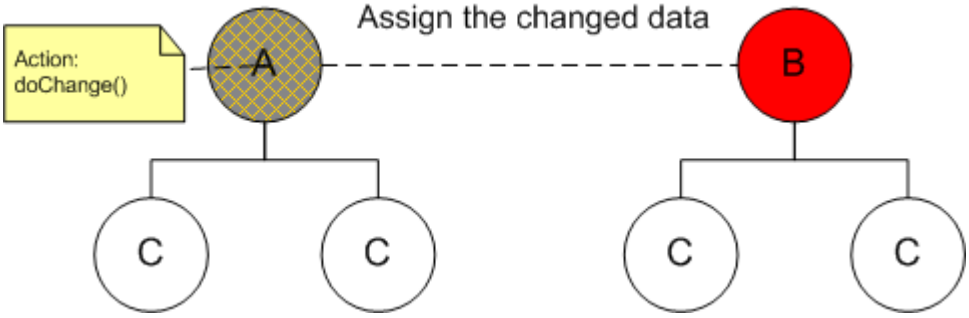


- Existiert ein Objekt $O_{x'}$ im Baum `T2`, welcher im Basis Baum `T1` nicht vorkommt, wird eine Aktion erzeugt, welche diesen Knoten entfernt. Dabei ist das X-Objekt-Paar unvollständig $(\varepsilon, O_{x'})$

- Existiert ein Objekt O_x im Basis Baum `T1`, welcher im Baum `T2` nicht existiert, wird eine Aktion erzeugt, welche den fehlenden Knoten hinzufügt. Dabei ist das X-Objekt-Paar unvollständig (O_x, ε)



- Sind die Objekte O_x und $O_{x'}$ unterschiedlich, wird eine Aktion erzeugt, welche die Unterschiede bereinigt.



Nach dem die Aktionen erstellt wurden, können diese ausgeführt werden. Am Ende erhalten wir `T1 = T2`.

Die Aktionen werden in der Klasse TXSynchronAction abgebildet.

API	zeusbase/System/XSynchronAction.h
	`virtual ENodeActionType MQUALIFIER getType() const;` Gibt den Typ der Aktion zurück.
	`virtual Retval MQUALIFIER getNode1(IXObject*& pNode) const;` Gibt das X-Objekt des Baums T1 zurück.
	`virtual Retval MQUALIFIER getNode2(IXObject*& pNode) const;` Gibt das X-Objekt des Baums T2 zurück.

	`virtual void MQUALIFIER disable();`
	`virtual void MQUALIFIER enable();`
	
	Deaktivieren bzw. aktivieren einer Aktion. Standardeinstellung ist aktiviert.

	`virtual Retval MQUALIFIER executeAction(long lCycleCount=0, bool bUnfreeze = false, bool bCleanTree = false, IXSynchronActionListener* pListener = NULL);` Ausführen einer Aktion. Anstelle dieser Methode kann auch .die Methode `executeRootAction()` der Klasse TXObjectTreeSynchronizer ausgeführt werden.
	`virtual Retval MQUALIFIER getErrorCode() const;` Gibt den Fehler-Code nach dem 2ten Schritt der Synchronisation zurück.
	`virtual Retval MQUALIFIER getRootAction(IXSynchronAction*& rpRoot) const;` Gibt das Wurzel-Objekt des Differenz-Baums zurück.

3.9.10.2 Objekt-Baum Synchronisation mit TXObjectTreeSynchronizer

Die Klasse TXObjectTreeSynchronizer implementiert den ersten Schritt des Prozesses.

API	zeusbase/System/XObjectTreeSynchronizer.h
	`Retval synchronizeTree(IXObject& rSource, IXObject& rTarget);` Erstellt den Differenz-Baum aus dem SOLL-Zustand `rSource` und dem IST-Zustand `rTraget`.
	`Retval getRootAction(IXSynchronAction*& rpObject);` Gibt die Wurzel des Differenz-Baums zurück. Diese Aktion beinhaltet die Differenzen der Wurzel Objekte der X-Objekt-Bäume.
	`Retval getSubAction(IXObject* pRefObject, IXSynchronAction*& rpObject);` Git eine Aktion zu einem bestimmten X-Objekt des Baums T1 zurück.
	`Retval executeRootAction(IXSynchronActionListener* pListener = NULL);` Führt die Wurzel-Aktion des Differenz-Baums aus. Dadurch werden alle Aktionen rekursiv aufgerufen und ausgeführt.

API	zeusbase/System/XObjectTreeSynchronizer.h
	`Retval executeSubAction(IXObject* pRefObject, IXSynchronActionListener* pListener = NULL);` Führt eine Aktion aus, welche das X-Objekt `pRefObject` des Baums T1 beinhaltet.
	`void setUnfreezeFlag(bool bUnfreeze);` Wird das Flag gesetzt, werden nach der Synchronisation alle Objekte freigeschaltet. Standardeinstellung ist ausgeschaltet (false).
	`void setForceChangeActions(bool bForceChanges);` Die Synchronisation wird erzwungen, wenn dieses Flag gesetzt wird. Es gibt keine Aktionen, die am Baum T2 keine Änderungen vornehmen. Standardeinstellung ist ausgeschaltet (false).
	`void setAutoCheck(bool bValue);` Die Attribute, Variablen und XML-Unterknoten werden automatisch auf Unterschiede geprüft. Die Prüfung erfolgt rekursiv. Standardeinstellung ist ausgeschaltet (false).
	`void setNumberOfCycles(Int iCycleCount);` Der Synchronisations-Prozess (2ter Schritt) erfolgt in mehreren Zyklen. Dabei ist der erste Zyklus immer das Bereinigen der XML-Struktur. Weitere kundenspezifische Zyklen können definiert werden. Standardeinstellung ist 1.

3.9.10.3 Beispiele

Folgendes Beispiel synchronisiert 2 X-Objekt-Bäume mit der TXObjectTreeSynchronizer-Klasse.

Synchronisieren der X-Objekte

```
#include <zeusbase/System/XObjectTreeSynchronizer.h>

TAutoPtr<IXObject> ptrRoot1;
TAutoPtr<IXObject> ptrRoot2;

//getting root objects
...

TAutoPtr<TXObjectTreeSynchronizer> ptrSynchronizer = new
  TXObjectTreeSynchronizer();

//Creates the action tree (1st step of the process)
if (ptrSynchronizer->synchronizeTree(*ptrRoot1,
                                     *ptrRoot2) == RET_NOERROR)
{
  TAutoPtr<IXSynchronAction> ptrActionTree;
  if (ptrSynchronizer->getRootAction(ptrActionTree) == RET_NOERROR)
  {

    //Executes the action tree (2nd step of the process)
    if (ptrActionTree->executeAction() == RET_NOERROR)
    {
      //Trees are now synchronized
    }
  }
}
```

Tabelle 41: Das Beispiel zeigt ein einfaches Synchronisieren zweier X-Objekt-Bäume.

3.10 Das Sicherheitskonzept

In diesem Kapitel wird das Sicherheitskonzept des Zeus-Frameworks erläutert. Nicht umgesetzte Mechanismen oder Aspekte sind klar deklariert.

Das Sicherheitskonzept umfasst das Laden von den Code-Modulen.

3.10.1 Sicheres Laden von Code-Modulen

Das Konzept des sicheren Ladens von Code-Modulen wurde in der aktuellen Version des Zeus-Frameworks umgesetzt. Die Eigenschaft kann durch die Einstellung `zeus.LibraryManager.Security` in der Konfiguration ein oder ausgeschaltet werden.

Ob ein Code-Module geladen werden kann oder nicht, liegt an seiner Signatur. Diese Signatur wird vor jedem Laden erstellt und mit einem Eintrag aus einer Datenbank verglichen.

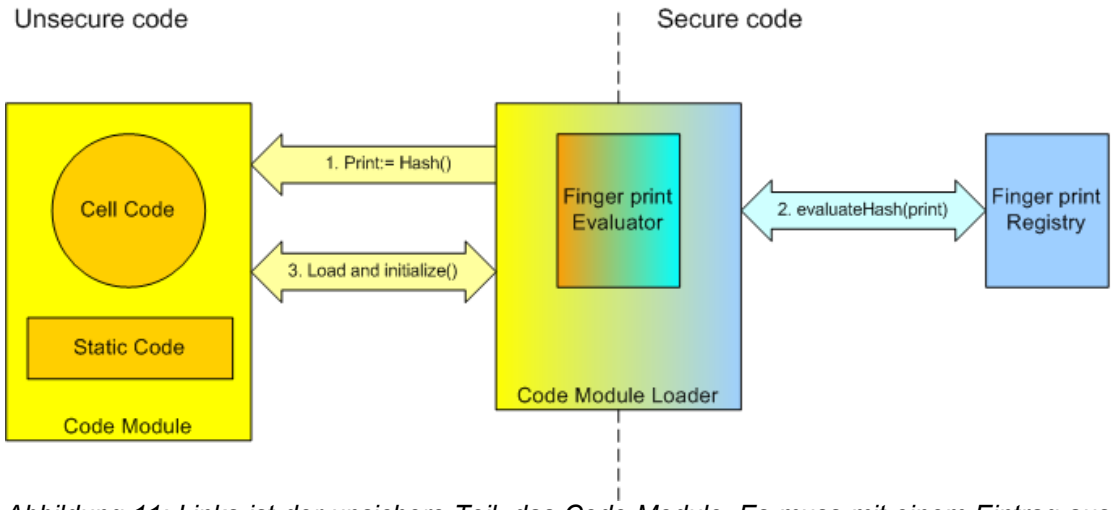


Abbildung 11: Links ist der unsichere Teil, das Code-Module. Es muss mit einem Eintrag aus der Datenbank verglichen werden, bevor es geladen wird.

Das aktuelle Zeus-Framework generiert den Fingerabdruck mit dem Whirlpool Algorithmus. Die Einträge in der Datenbank (Datei `security.db`) sind noch unverschlüsselt.

➔ Extension:

Die Datenbank könnte mit einem Kryptographie Algorithmus verschlüsselt werden.

Die Registrierung von Code-Modulen ist im Anhang beschrieben (Benutzerhandbuch).

3.10.2 Verschlüsselung

Das Zeus-Framework bietet eine einfache und sehr effiziente Block-Verschlüsselung an. Durch das Verwenden der Input und Output-Streams können somit Dateien und Übertragungen via Netzwerk verschlüsselt und auf der anderen Seite wieder entschlüsselt werden.

Folgende Klassen werden für die Verschlüsselung verwendet

API	
TAbstractCrypter	Basisklasse für alle kryptologischen Klassen, die zur Verschlüsselung verwendet werden. zeusbase/Security/AbstractCrypter.h
TSimpleDES	Beispiel Implementation von DES. Diese Klasse sollte nur zu illustrationszwecken verwendet werden. zeusbase/Security/SimpleDES.h
TBlockCipherXTEA	Block Verschlüsselung mit dem XTEA. zeusbase/Security/BlockCipherXTEA.h
TCryptedOutputStream	Filter zum Verschlüsseln von Daten. Dabei wird der XTEA verwendet. zeusbase/Security/CryptedOutputStream.h
TCryptedInputStream	Filter zum Lesen von verschlüsselten Daten. Dabei wird der XTEA verwendet. zeusbase/Security/CryptedInputStream.h

Folgendes Beispiel zeigt eine mögliche Verschlüsselung von Dateien auf:

Verschlüsseln von Dateien

```
#include <zeusbase/System/FileOutputStream.h>
#include <zeusbase/Security/CryptedOutputStream.h>

TByteArray aKey;

//read the key
...


TString strFile = L"./Test.data";
TAutoPtr<TFileOutputStream> ptrOutFile = new
   TFileOutputStream(strFile, true);

TAutoPtr<TCryptedOutputStream> ptrOutSec = new
   TCryptedOutputStream(ptrOutFile, etXTEA);

ptrOutSec->setKey(aKey);

ptrOutSec->write(pBuffer, iBufferSize);

ptrOutSec->flush();
ptrOutSec->close();
```

Tabelle 42: Dateien werden verschlüsselt, indem ein CryptedOutputStream-Filter dazwischen geschaltet wird. Bevor der Daten-Puffer geschrieben werden kann, muss ein Schlüssel gesetzt werden.

3.10.3 Secure-Hash und Finger Print

Mit der Klasse TSecureHash können Daten zum Identifikationsnachweis verschlüsselt werden. Die Funktion ist eine Ein-Weg-Funktion.

Die TFingerPrint-Klasse kapselt den Secure-Hash und kann in eine Datenbank serialisiert werden. Sie wird durch den SecurityManager verwendet, um Module laden zu können.

API	
`TSecureHash`	Secure Hash Klasse zum Ein-Weg-Verschlüsseln von Daten. `zeusbase/Security/SecureHash.h`
`TFingerPrint`	Fingerabdruck von Daten können mit dieser Klasse serialisiert und in eine Datenbank gespeichert werden. `zeusbase/Security/FingerPrint.h`

Folgendes Beispiel zeigt die Verwendung der TSecureHash-Klasse:

Secure Hash

```
#include <zeusbase/Security/SecureHash.h>

TString strData = L"This is some text";

TSecureHash Hash(strData);

TByteArray aHashData;
Hash.getHash(aHashData);

...
```

Tabelle 43: Die Secure Hash Klasse generiert einen eindeutigen Hash-Wert zum Identifikationsnachweis der Original-Daten.

3.11 Messaging

Zum asynchronen Kommunizieren zwischen entfernten Systemen wurde ein Meldungssystem entwickelt. Die Meldungen können durch so genannte Kommunikationskanäle (Pipes) abgesetzt werden. Ein Kanal ist eine Verbindung von einem System zu einem anderen. Das andere Ende des Kanals kann auf dem gleichen PC oder auf einem entfernten Rechner liegen.

3.11.1 Meldungsklassen

Die Meldungsklassen sind alle von der Schnittstelle `ISerializable` abgeleitet und können so serialisiert übers Netzwerk gesendet werden. Dadurch lassen sich die Systemgrenzen einfach überwinden.

API	
`IMessage`	Basisschnittstelle für Meldungen. `zeusbase/Messaging/Interfaces/IMessage.hpp`
`IAttributeMessage`	Schnittstelle für attributbasierende Meldungen `zeusbase/Messaging/Interfaces/IAttributeMessage.hpp`
`IBinaryMessage`	Schnittstelle für Meldungen mit binärem Format. `zeusbase/Messaging/Interfaces/IBinaryMessage.hpp`
`IXMLMessage`	Schnittstelle für Meldungen mit XML Inhalt. `zeusbase/Messaging/Interfaces/IXMLMessage.hpp`
`IObjectMessage`	Schnittstelle für Meldungen, die ein Objekt beinhalten. `zeusbase/Messaging/Interfaces/IObjectMessage.hpp`
`ISystemMessage`	Schnittstelle für Meldungen, die Systemdaten übermitteln. `zeusbase/Messaging/Interfaces/ISystemMessage.hpp`
`TAttributeMessage`	Klasse welche eine Attribut-Meldung repräsentiert. `zeusbase/Messaging/AttributeMessage.h`
`TBinaryMessage`	Klasse welche ein binäre Meldung repräsentiert. `zeusbase/Messaging/BinaryMessage.h`
`TXMLMessage`	Klasse welche eine XML Meldung repräsentiert. `zeusbase/Messaging/XMLMessage.h`
`TObjectMessage`	Klasse welche eine Objekt Meldung repräsentiert. `zeusbase/Messaging/ObjectMessage.h`
`TSystemMessage`	Klasse welche eine Systemmeldung repräsentiert. `zeusbase/Messaging/SystemMessage.h`

Mit diesen Klassen können nun verschiedenartige Meldungen realisiert werden:

- Kontrollmeldungen, wie Ping, Status Informationen etc.

- Die Arbeitspakete, welche Daten mit Prozessbeschreibung enthalten.
- Objekte (Meldungszellen, Agenten)

3.11.1.1 Die Kontrollmeldungen

Die Kontrollmeldungen dienen dem Steuern und Überwachen von Systemen. Sie kann zur Prüfung von Qualität und Lebenserhaltung verwendet werden.

Die berühmteste Meldung ist das Ping, welches prüft, ob der Kommunikationspartner noch lebt.

3.11.1.2 Die Arbeitspakete

Die Arbeitspakete enthalten Daten. Diese Daten können binär oder als Text übergeben werden.

➔ Extension: Arbeitspakete mit Workflow

Der Workflow ist in Form eines Petrinetzes angeben und wird jedem Arbeitspaket mitgegeben. In diesem Petrinetz wird gespeichert, wo sich das Paket zur Zeit befindet, ob die Zelle auf weitere Arbeitspakete warten muss und an welche Zelle(n) das bearbeitete Paket gesendet werden soll.

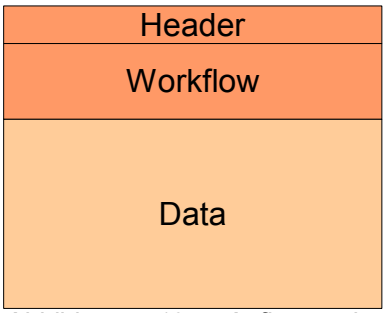


Abbildung 12: Aufbau der Arbeitspakete

3.11.2 Kommunikationskanäle

Will ein System mit einem anderen System kommunizieren, muss zuerst ein Kommunikationskanal erstellt werden. Durch diesen Kanal werden dann Arbeitspakete und Kommandos gesendet.

Die Kommunikationskanäle können als lokale Objekte oder als Remote-Objekte erzeugt werden. Remote-Pipes müssen in einer globalen Registrierungsstelle (Naming-Service) registriert werden, damit externe Systeme diese Pipe ermitteln kann.

Zum Bau einer Kommunikationsplattform dienen folgende Klassen:

API	
`ICommPipe`	Schnittstelle eines Kommunikationskanals. Durch diese Schnittstelle können Meldungen abgesetzt werden `zeusbase/CellModel/Interfaces/ICommPipe.hpp`
`IOwnCommPipe`	Dieses Schnittstelle eines Kommunikationskanals ist nur für den Besitzer eines Kanals verfügbar und dient dem Lesen von Meldungen. `zeusbase/CellModel/Interfaces/IOwnCommPipe.hpp`
`TCommPipe`	Kommunikationskanal zum Senden und Empfangen von Meldungen. `zeusbase/CellModel/CommPipe.h`
`TMappedCommPipe`	Kommunikationskanal zum Senden und Empfangen von Meldungen. Jede Meldung hat eine ID. Diese ID kann nur einmal in dieser Pipe vorhanden sein. Kommt eine neue Meldung mit einer ID die bereits in der Pipe vorhanden ist, wird die bereits bestehende Meldung durch die neue Meldung ersetzt.

3.11.2.1 Lokale Kommunikationskanäle

Lokale Kommunikationskanäle sind intern im Framework implementiert. Es gibt unterschiedliche Arten der Kommunikation:

- Queuing: Die Arbeitspakete werden in eine Queue abgelegt. Für Queues gibt es keine Grenze der Anzahl Einträge.

- Queuing with Replace: Werden viele gleiche Pakete übertragen, interessieren aber nur die aktuellsten Pakete, kann das Queuing with Replace angewandt werden. Jedes Arbeitspaket besitzt eine ID. Wird die gleiche ID nochmals verwendet, und ist das vorhergehende Arbeitspaket noch in der Queue, wird dieses durch das aktuelle Arbeitspaket ersetzt. Der Vorteil ist, dass die Grösse der Queue klein gehalten werden kann. Dieses Verfahren wird vor allem bei der Signal-Visualisierung verwendet, um schnell auf Änderungen reagieren zu können

und das Schleppen von Werten zu umgehen (Stau in der Queue). Dieses Verfahren steht in der aktuellen Version noch nicht zur Verfügung.

Ein lokaler Kommunikationskanal wird durch folgende Kenngrössen beschrieben:

- Name des Systems
- Name des Kommunikationskanals (Pipe)

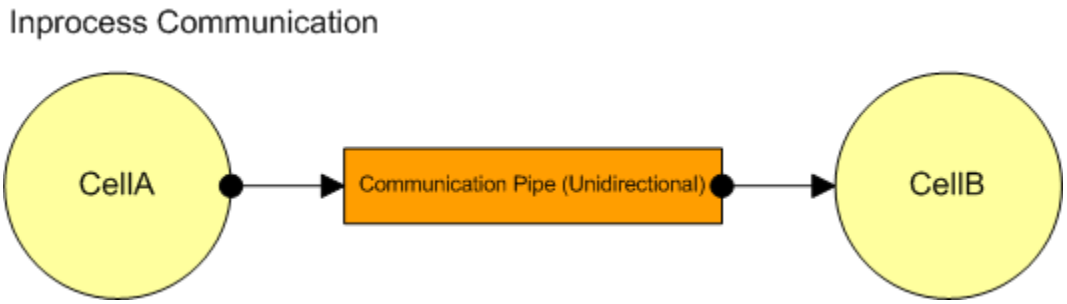


Abbildung 13: Inprozess-Kommunikationskanäle sind eine direkte unidirektionale Verbindung zwischen 2 Zellen. DIe Zelle dient hier als System.

3.11.2.2 Entfernte Kommunikationskanäle

Damit Systeme über ein Netzwerk miteinander kommunizieren können, braucht es eine Registrierung. Folgende Daten der Zellen werden dort verwaltet:

- Name des Systems
- Name des Kommunikationskanals (Pipe)
- Adresse des Host (Namen des Computers oder IP Adresse) (TCP/IP)
- Port des Socket (TCP/IP)

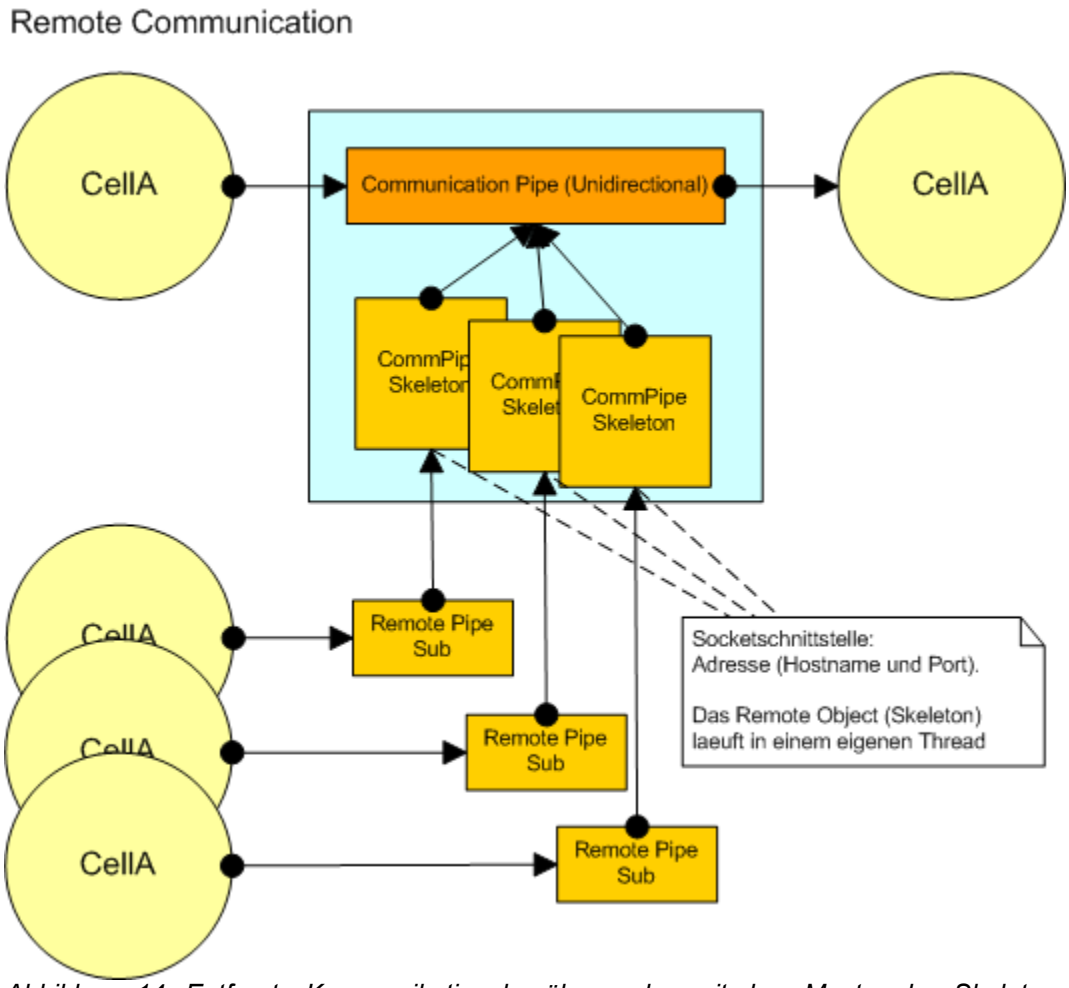


Abbildung 14: Entfernte Kommunikationskanäle werden mit dem Muster des Skeleton-Stub implementiert.

Die Registrierungsstelle ist eine eigene Applikation, welche für die laufenden Zeus-Applikationen zugänglich ist. Beim Starten einer Zelle werden ihre Kommunikationskanäle automatisch bei der Registrierungsstelle registriert.

Als Kommunikationsprotokoll wird das Cell Communication Transfer Protocol CCTP verwendet. Es handelt sich um ein binäres Protokoll, welches einfach über Sockets angewendet werden kann.

3.12 Serialisierung von Objekten

Die Serialisierung von Objekten ist eine Technik, um Objekte in einen Byte Stream zu verwandeln und daraus wieder ein Objekt zu erstellen. Diese Technik wird vor allem bei komplexeren verteilten Applikationen verwendet, um Datenobjekte übers Netzwerk zu senden. Das Speichern von Objektbäumen in einer Datei ist eine andere Anwendung dieses Verfahren.

3.12.1 Serialisierungs-Protokoll

Bei dem Verfahren ist es wichtig, dass sich die serialisierten Objekte nach dem Erzeugen wieder gleich verhalten. Dabei werden ausschliesslich die Attribute eines Objekts im Byte Stream gespeichert. Zur Serialisierung von Objekten wurde ein einfaches binäres Format definiert. Es besteht aus einem Kopfteil und aus den Daten der Objektattribute.

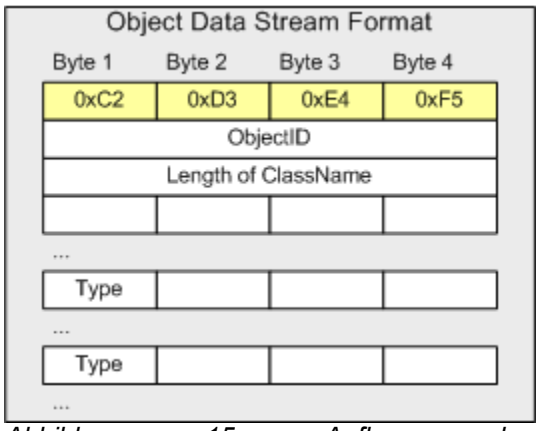


Abbildung 15: Aufbau des Serialiserungsprotokoll

Der Kopfteil wird durch eine 4 Byte lange Kennung initialisiert. Damit kann ein Stream als Objekt-Stream identifiziert werden. Weitere 4 Bytes dienen der Objekt ID. Am Schluss des Kopfteils wird die Länge des Klassennamens und der Klassennamen selbst angegeben. Dieser Namen dient der Objektfabrik die richtige Klasse zu instanzieren.

Das Zeus-Framework unterstützt folgende Attributtypen:

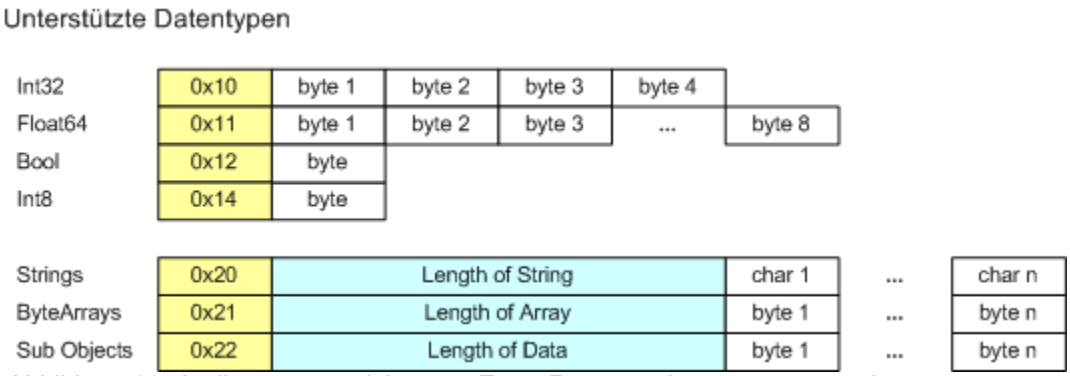


Abbildung 16: Attributtypen welche vom Zeus-Framework unterstützt werden.

3.12.2 Die Objektfabrik

Damit aus einem Byte Stream wieder die richtigen Objekte erzeugt werden, braucht es eine Objektfabrik. Sie prüft das Protokoll des Streams und liest den Header aus. Durch den Klassennamen wird über eine interne Liste nach der Fabrikfunktion gesucht und diese schlussendlich aufgerufen.

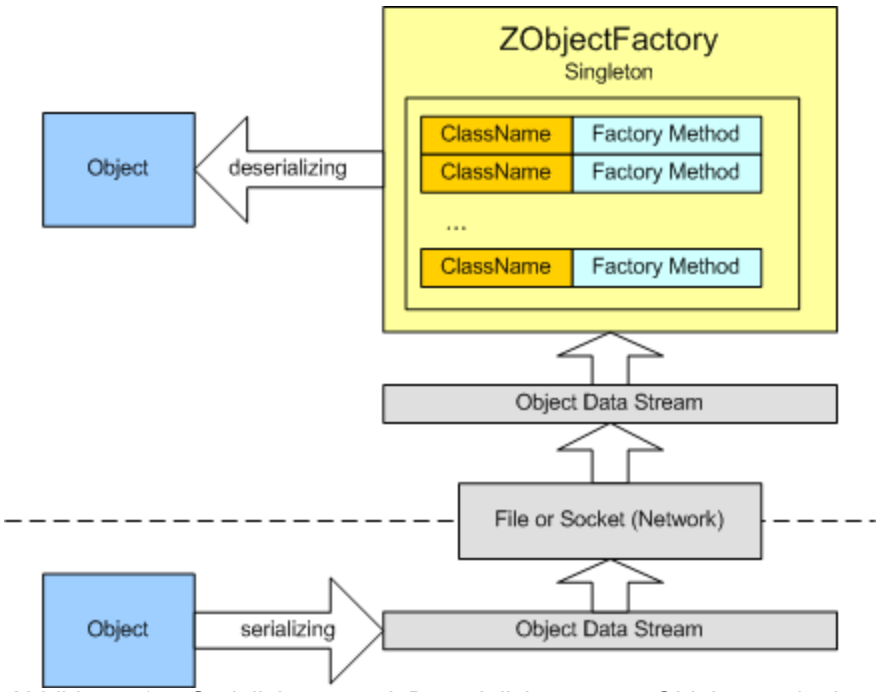


Abbildung 17: Serialisieren und Deserialisieren von Objekten mit dem Zeus-Framework

3.12.3 Serialisierbare Objekte implementieren

Alle serialisierbaren Objekte müssen bei der Objektfabrik registriert sein, damit die richtige Klasse instanziert werden kann. Die Registrierung erfolgt beim Programmstart, kann aber auch während der Programmlaufzeit geschehen. Sie muss aber zwingend **vor** dem Eintreffen des Byte Streams erfolgen.

Folgende Klassen dienen zum Serialisieren bzw Deserialisieren von Objekten.

API	
`ISerializable`	Schnittstelle für die serialisierbaren Objekte. `zeusbase/System/Interfaces/ISerializable.hpp`
`TZObjectFactory`	Fabrik zum Erstellen von Objekten aus einem Byte Stream. `zeusbase/System/ZObjectFactory.h`
`TSerializer`	Hilfsimplementation zum Serialisieren der Attribute. `zeusbase/System/Serializer.hpp`

3.12.3.1 Schnittstelle ISerializable

Jedes serialisierbare Objekt muss die Schnittstelle ISerializable implementieren.

API `zeusbase/System/Interfaces/ISerializable.hpp`
`Retval MQUALIFIER serialize(IByteArray& rStream) const;` Erzeugt aus einem Objekt einen Stream (Byte Array).
`Uint MQUALIFIER getObjectID() const;` Gibt die ID eines Objekts zurück. Jedes serialisierbare Objekt braucht eine eindeutige ID

Diese Methoden werden durch vorgefertigte Makros in der Klasse selbst implementiert. (Siehe Makros)

3.12.3.2 Makros

Die Entwicklungsumgebung des Zeus-Frameworks stellt verschiedene Makros zur Verfügung, um möglichst einfach serialisierbare Objekte zu implementieren. Im Header

der Klasse muss das Include `<zeusbase/System/Serializer.hpp>` stehen.

Folgendes Makro dient dem Registrieren der Klasse:

API	`OBJECTFACTORY_REGISTER_CLASS(classid)`
	Registriert eine Klasse bei der ZObjectFactory. Durch diese Registrierung kann ein Objekt einer bestimmten Klasse aus einem Stream erzeugt werden.

Makro zum Registrieren

```
#include <SerialTestObject.h>

void main()
{
  ...
  OBJECTFACTORY_REGISTER_CLASS(TSerialTestObject);
  ...
}
```

Tabelle 44: Registrieren von serialisierbaren Objekten, hier der Klasse `TSerialTestObject`.

Für die Programmierung von serialisierbaren Objekten stehen folgende Makros zur Verfügung:

API	`SERIAL_START(classid, classname)`
	Im Header der serialisierbaren Klasse muss ein Bereich definiert werden, welcher alle Attribute spezifiziert, die im Stream vorkommen. Beim Start muss die Klasse und der Klassennamen (als Zeichenkette) angegeben werden.

Gefolgt vom Startteil können folgende Makros zum Streamen von Attributen angefügt werden

API	`SERIAL_BOOL(attribute)`
	Serialisieren von bool-Werten.

API	`SERIAL_INT32(attribute)`
	Serialisieren eines Int32 Werts. Weitere Makros sind `SERIAL_INT8`, `SERIAL_INT16` und `SERIAL_INT64`, `SERIAL_INT32_CONVERT`.

API	`SERIAL_FLOAT64(attribute)`
	Serialisieren von Float-Werten. Weitere Makros sind `SERIAL_FLOAT32`

API	`SERIAL_STRING(attribute)`
	Serialisieren eines Strings. Weitere Makros sind `SERIAL_STRING_SET_GET` zum Aufrufen von get- und set-Methoden.

API	`SERIAL_BYTEARRAY(attribute)`
	Serialisieren eines Byte-Arrays. Weitere Makros sind `SERIAL_BYTEARRAY_PTR`

API	`SERIAL_FLOAT64LIST(attribute)`
	Serialisieren eines Float-Arrays.

API	`SERIAL_OBJECT(interface_id, attribute)`
	Serialisieren eines serialisierbaren Objekts. Dabei muss noch die Schnittstellen-ID angegeben werden, damit der richtige Cast-auf das Objekt erfolgt.

API	`SERIAL_OBJECTLIST(interface_id, subclass, attribute)`
	Serialisieren einer Liste von serialisierbaren Objekten. Dabei muss noch die Schnittstellen-ID angegeben werden, damit der richtige Cast-auf das Objekt erfolgt. Die Angabe subclass ist notwendig zum Abfüllen der Liste. Es entspricht der Angabe im Template. `List<subclass>`

API	`SERIAL_END`
	Abschliessen der Serialisierung des Objekts.

Header eines serialisierbaren Objekts

```
#include <zeusbase/System/Serializer.hpp>

class TSerialTestObject : public TZObject, public ISerialTestObject
{
  public:
    TSerialTestObject();
...
    //Serializing macros
    SERIAL_START(TSerialTestObject, L"TSerialTestObject");
      SERIAL_INT32(m_lValue)
      SERIAL_FLOAT64(m_dValue)
      SERIAL_STRING(m_strValue)
      SERIAL_OBJECT(INTERFACE_ISerialTestSubObject, m_pObject)
    SERIAL_END
...
  private:
    ///Long value
    Int32 m_lValue;
    ///Double value
    Float64 m_dValue;
    ///String value
    TString m_strValue;
    ///An other serializable Object
    ISerialTestSubObject* m_pObject;
    ///Lock (not a serialisable object)
    TCriticalSection& m_rLock;
};
```

Tabelle 45: Deklaration der Klasse TSerialTestObject als serialisierbares Objekt

In der Implementation der Klasse braucht es einen speziellen Konstruktor, welcher ein neues Objekt aus dem Stream erzeugen kann.

API	`SERIAL_CONSTRUCTOR(classid)`
	Konstruktor des serialisierbaren Objekts. Für beinhaltet alle Initialisierungen. Für komplexere Objekte sind die 2 folgenden Makros zu verwenden.

API	`SERIAL_CONSTRUCTOR_START(classid)`
	Methodenrumpf des Konstruktors. Dieses Makro wird mit `SERIAL_CONSTRUCTOR_INIT` verwendet, wenn eigene Objekte und Referenzen erzeugt werden müssen.

API	`SERIAL_CONSTRUCTOR_INIT`
	Aufruf zum Initialisieren des Objekts aus dem Stream

Implementation eines serialisierbaren Objekts

```
USING_NAMESPACE_Zeus


SERIAL_CONSTRUCTOR_START(TSerialTestObject),
  m_rLock(*new TCriticalSection())
{
  //some other intialisation goes here
  SERIAL_CONSTRUCTOR_INIT;
}
```

Tabelle 46: Implementation der Klasse TSerialTestObject als serialisierbares Objekt

3.12.4 Einschränkungen

Die Serialisierung bietet etliche Möglichkeiten Objekte und deren Attribute zu serialisieren. Es gibt jedoch Einschränkungen in der Verwendung und der Implementation der Objekte.

1. Ein serialisierbares Objekt kann keine Attribute von Objektreferenzen besitzen, die serialisiert werden sollen.

2. Verweise in Form von Zeiger oder Referenzen auf andere Objekte werden nicht serialisiert. Es können nur SubObjekte serialisiert werden.

3. Zur Zeit ist es nur möglich, Klassen aus der Applikation zu serialisieren. Klassen, die in den Code-Modulen implementiert wurden, können nicht serialisiert werden, ausser die Klasse wird beim Laden des CodeModuls mit dem Makro registriert.

Makro zum Registrieren einer Klasse im CodeModule

```
#include <SerialTestObject.h>
#include <zeusbase/System/XObjectFactory.h>
#include <zeusbase/System/LibraryManager.h>
#include <zeusbase/MOM/AbstractModuleSession.h>

MregisterLibrary_Start(L"MyTestModule");
  OBJECTFACTORY_REGISTER_CLASS(TSerialTestObject);
MregisterLibrary_End;
```

Tabelle 47: Registrieren von serialisierbaren Objekten, hier der Klasse `TSerialTestObject`.

3.13 Remote Method Invocation mit C++

Durch eine Implementation von Remote Method Invocation (RMI) kann das Zeus-Framework für verteilte Applikationen eingesetzt werden. Diese Implementation ist an Java RMI angelehnt. Dabei werden Methodenaufrufe von Objekten übers Netzwerk getätigt, ohne dass der Programmierer sich um Protokolle, Sockets usw. kümmern muss.

3.13.1 Funktionsweise

RMI für C++ ist sehr einfach implementiert. Mittels Remote Interface, eine Beschreibung der Schnittstelle des Objekts, werden Skeleton und Stub-Dateien generiert.

Der Stub nimmt die Methodenaufrufe des Clients entgegen und wandelt diese in einen Byte-Stream um (serialisieren). Der Byte-Stream wird über eine Socketverbindung zum Server gesendet.

Der Server besteht aus dem Servant-Objekt und einem Skeleton. Das Skeleton empfängt einen Byte-Stream vom Socket und versucht diesen in einen Methodenaufruf umzuwandeln. Bei erfolgreicher Umwandlung wird die verlangte Methode beim Servant ausgeführt.

Alle Rückgabewerte und Rückgabeparameter werden vom Skeleton empfangen und wieder als Byte-Stream über den Socket zurückgesendet.

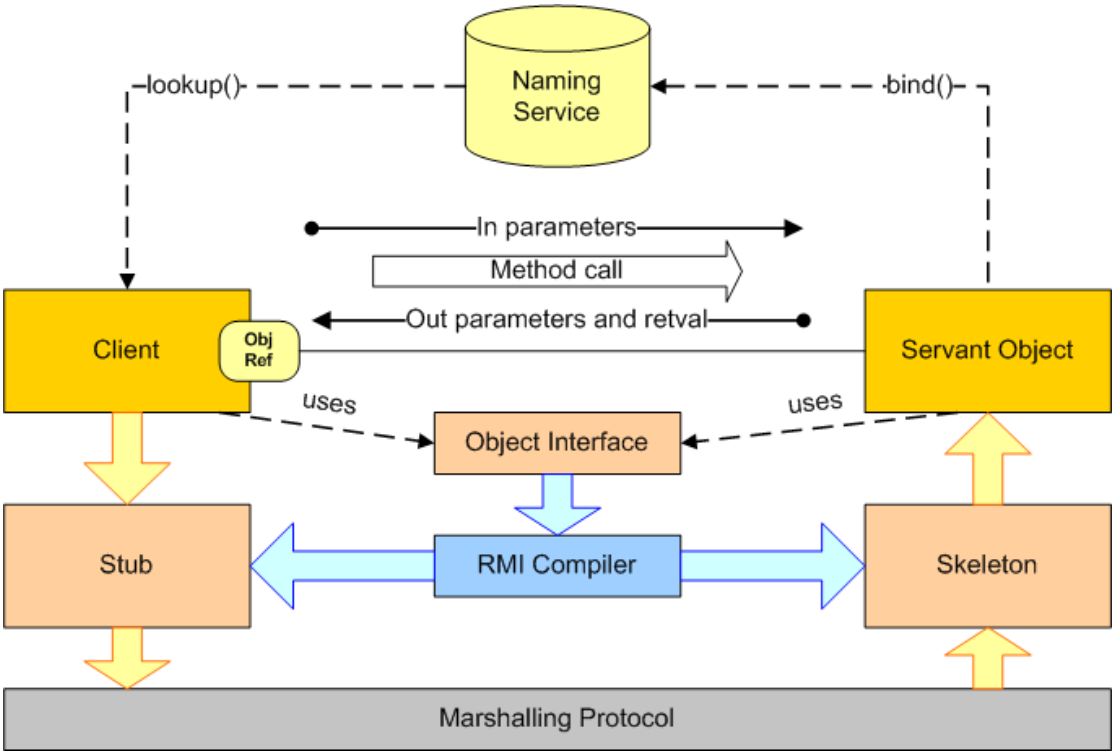


Abbildung 18: Zeigt das Schichtenmodell der RMI Implementation von Zeus-Framework

3.13.2 Parameter Typen

Durch das RMI können neben den primitiven Datentypen auch Objekte übergeben werden. Dabei gibt es zwei Arten von Objekten:

- Serialisierbare Objekte: Interface ISerializable
- Remote Objects: Intefrace IRemoteObject

Die serialisierbaren Objekte werden serialisiert und als Byte-Stream gesendet. Das Bedeutet, dass Client nicht das gleiche Objekt referenziert wie der Servant, sondern eine Kopie mit identischem Inhalt erhält.

Von den Remote Objects wird nicht der Inhalt übergeben, sondern eine Referenz (Remote Object Reference). Diese Referenz besteht aus folgenden Elementen:

- Adresse: IP Adresse des Servants
- Port: Port des Servant Sockets

- ClassName: Name der Klassen
- CodeModule: Name des Code Moduls

Wird ein Parameter vom Typ IRemoteObject übergeben, wird auf der Empfängerseite ein Stub erstellt, welcher die Verbindung zu dem Remote Object herstellen kann.

3.13.3 Klassen des RMI's

Folgende Klassen und Objekte sind von Interesse:

API	
`Naming`	Objekt des Naming-Service. Durch dieses Objekt können Remote Objects nachgeschlagen werden, oder es können eigene Remote Objects registriert werden. `zeusbase/Remote/Naming.h`
`IRemoteObject`	Alle Remote Objects müssen diese Schnittstelle implementieren. Die Hilfsklasse `TAbstractRemoteObject` implementiert die wichtigsten Methoden. `zeusbase/Remote/Interfaces/IRemoteObject.hpp`
`TAbstractSkel`	Hilfsklasse für die Implementation des Skeleton. `zeusbase/Remote/AbstractSkel.h`
`TAbstractStub`	Hilfsklasse für die Implementation der Stub-Klasse. `zeusbase/Remote/AbstractStub.h`
`ISerializable`	Schnittstelle für die serialisierbaren Objekte. `zeusbase/System/Interfaces/ISerializable.hpp`

3.13.4 Verwenden von RMI

Der Ablauf zum Verwenden von RMI ist wie folgt:

1. Erstellen eines Remote-Interface.
2. Mit dem RMI Compiler [rmicc] werden aus dem Remote Interface die Stub und Skeleton Dateien erzeugt

3. Schreiben des Servants (Objekt, welches das Remote Interface implementiert und die Arbeit ausführt)

4. Schreiben des Clients, welcher das Remote Interface verwendet.

5. Starten der Applikationen

   5.1. Starten des NameServers (Server Dienst für NamingSevice)

   5.2.Starten der Server Applikation (Servant Object)

   5.3.Starten der Client Applikation

3.13.4.1 Erstellen des Remote Interface

Das Remote Interface wird in eine `.hpp` Datei geschrieben. Dabei gelten gewisse Vorschriften:

- Anstelle der `class`-Anweisung wird die Anweisung `remote_interface` verwendet, damit der RMI Compiler dieses Interface als Remote Interface akzeptiert.

- Das Interface muss von `IZUnknown` abgeleitet sein.

- Jede Methode muss ein Retval-Rückgabetyp besitzen. Da keine Exceptions ausgeworfen werden können, werden RMI-Fehler per Rückgabewert zurückgeben.

- Übergabeparameter sind primitive Datentypen, `IString`, `IStringList`, `IRemoteObject`, `IRemoteObjectList` oder `ISerializable`-Typen. Andere Typen verlangen eine Erweiterung des [rmicc].

- Als primitive Datentypen sind keine plattformspezifische Typen wie `Int`, `Float`, etc. zu verwenden. Die Datentypen müssen auf allen Plattformen die gleiche Grösse besitzen (`Float64`, `Int32` etc).

Ein Übergabeparameter kann in folgende Gruppen eingeteilt werden: In-, Out- oder InOut-Parameter. Im RMI für C++ wird je nach Signatur und `const`-Qualifikation entschieden, in welche Gruppe ein Parameter gehört.

- In-Parameter

  - sind alle Übergaben per Value (primitive Datentypen)

  - alle Übergaben mit der Signatur `const classname& xy` oder `const classname* xy`

- Out-Parameter
  - alle Übergaben mit der Signatur classname*& xy
- InOut-Parameter
  - alle Übergaben mit der Signatur `classname& xy` oder `classname* xy`

Folgendes Beispiel zeigt eine Definition eines Remote Interfaces.

Definition eines Remote Interfaces

```
#ifndef IRemoteTestObjectHPP
#define IRemoteTestObjectHPP

#include <zeusbase/System/Interfaces/IZUnknown.hpp>
#include <zeusbase/System/Interfaces/IString.hpp>

#define INTERFACE_IRemoteTestObject      0x00FFFFFF

BEGIN_NAMESPACE_Zeus

/*************************************************************/
/*! interface definition of the remote test object
*/
/*************************************************************/
remote_interface IRemoteTestObject : public IZUnknown
{
  public:
    /*************************************************************/
    /*! Returns the double value
    */
    /*************************************************************/
    virtual Retval MQUALIFIER getFloatValue(Float64& rValue)
                              const=0;

    /*************************************************************/
    /*! Sets the double value
    */
    /*************************************************************/
    virtual Retval MQUALIFIER setFloatValue(Float64 dValue)=0;
};

END_NAMESPACE_Zeus

#endif
```

Tabelle 48: Definition eines Remote Interfaces. Eine Methode dient dem Ermitteln eines Float64-Werts, die andere um diesen Wert beim Remote Object zu ändern.

3.13.4.2 Erstellen des Stubs und Skeleton

Mit dem RMI Compiler [rmicc] können aus dem Remote Interface die Stub und Skeleton-Dateien erstellt werden.

Erstellen des Stub und Skeletons

```
rmicc ./IRemoteTestObject.hpp
```

Tabelle 49: Erstellen der Stub- und der Skeleton-Dateien.

Die erstellten Dateien müssen nun in die verschieden Projekte verteilt werden:

- Servant: Der Servant benötigt das Remote Interface und die Skeleton-Dateien
- Client: Der Client benötigt das Remote Interface und die Stub-Dateien.

Die Dateien müssen im entsprechenden Projekt aufgenommen und einkompiliert werden.

Weitere Möglichkeiten der Applikation [rmicc] sind im Kapitel [rmicc]- RMI Compiler für C++ beschrieben.

3.13.4.3 Implementieren des Servants

Der Servant, auch RemoteObject genannt, implementiert das Remote Interface. Diese Arbeit muss vom Entwickler gemacht werden.

Auf den Servant wird von aussen durch einen Netzwerk-Client zugegriffen. Dabei können auch mehrere Zugriffe parallel stattfinden. Deshalb ist es sehr wichtig, den Servant für Multithreading zu implementieren und die nötigen Schutzmechanismen einzusetzen.

Der Servant wird von der Basisklasse `TAbstractRemoteObject` abgeleitet. Die Methoden werden mit dem Makro `REMOTE_OBJECT_DECL` definiert.

Header des Servants

```
#include <interfaces/IRemoteTestObject.hpp>
#include <zeusbase/Remote/AbstractRemoteObject.h>

BEGIN_NAMESPACE_Zeus

class TRemoteTestObject : public TAbstractRemoteObject,
                          public IRemoteTestObject
{
  public:
    TRemoteTestObject();

    //Methods of IRemoteTestObject
    virtual Retval MQUALIFIER getFloatValue(Float64& rValue) const;
    virtual Retval MQUALIFIER setFloatValue(Float64 dValue);

    //Methods of IRemoteObject
    REMOTE_OBJECT_DECL

    //Methods of IZUnknown
    MEMORY_MANAGER_DECL

  protected:
    virtual~ TRemoteTestObject();

  private:
    ///Float64 value
    Float64 m_dValue;
    ///Lock
    TCriticalSection& m_rLock;
};
END_NAMESPACE_Zeus
```

Tabelle 50: Deklaration eines Servants (Remote Objects).

Im Implementationsteil wird das Makro `REMOTE_OBJECT_IMPL` verwendet.

Implementation des Servants

```
#include "RemoteTestObject.h"
#include "RemoteTestObject_Skel.h"

USING_NAMESPACE_Zeus

//Macro for Remote object
REMOTE_OBJECT_IMPL(TRemoteTestObject,
                   L"TRemoteTestObject",
                   L"",
                   TRemoteTestObject_Skel);

TRemoteTestObject::TRemoteTestObject()
 : TAbstractRemoteObject(),
   m_rLock(*new TCriticalSection())
{
  m_dValue = 0.0;
}

TRemoteTestObject::~TRemoteTestObject()
{ m_rLock.release(); }

Retval MQUALIFIER TRemoteTestObject::getFloatValue(Float64& rValue) const
{
  m_rLock.enter();
  rValue =  m_dValue;
  m_rLock.leave();

  return RET_NOERROR;
}

Retval MQUALIFIER TRemoteTestObject::setFloatValue(Float64 dValue)
{
  m_rLock.enter();
  m_dValue = dValue;
  m_rLock.leave();

  return RET_NOERROR;
}

//Macro for memory management (addRef, release and casting)
MEMORY_MANAGER_IMPL(TRemoteTestObject);
  INTERFACE_CAST(IRemoteTestObject, INTERFACE_IRemoteTestObject);
MEMORY_MANAGER_IMPL_PARENT_END(TAbstractRemoteObject);
```

Tabelle 51: Implementation eines Servants (Remote Objects).

Das Servant Objekt muss gestartet werden. Dabei erzeugt es einen Server-Socket. Zudem kann es bei einem Naming-Service registriert werden, damit andere Anwendungen (Clients) den Servant finden können. Mit dem Naming-Service kann durch das Singletons `Naming` kommuniziert werden.

Starten des Servants

```
#include <zeusbase/Remote/Naming.h>
#include <zeusbase/ZeusBase.h>
#include "RemoteTestObject.h"

int main(int iArgc, char* paArgs[])
{
  //Registers all classes for class factories
  TZeusBase::registerClasses();

  //Connect to naming server
  Naming.connect(L"124.33.21.5", 7000);

  ...
  //Create remote object and register object
  TAutoPtr<TRemoteTestObject> Servant = new TRemoteTestObject();
  Servant->connect(L"124.33.21.2", 6000);
  Naming.bind(TString(L"MyObject"), *Servant);
  ...
  //loop here (wait for clients and handle requests
  ...
  Naming.unbind(TString(L"MyObject"));

  Servant->disconnect();
  Naming.disconnect();

  return 0;
}
```

Tabelle 52: Verwenden des Naming-Service mit Remote Objects. Das Singleton Naming muss zuerst auf den Server verbunden werden.

3.13.4.4 Implementation des Clients

Der Client braucht den Stub des Servant. Bevor eine Verbindung mit dem Servant hergestellt werden kann, muss der Stub registriert werden. Dazu kann das Makro `OBJECTFACTORY_REGISTER_CLASS()` verwendet werden. Es ist sinnvoll alle Klassen der Zeusbase-Bibliothek ebenfalls zu registrieren.

```
TZeusBase::registerClasses();
OBJECTFACTORY_REGISTER_CLASS(<xy_Stub>);
```

Damit die IP-Adresse und der Port des Servants gefunden werden können, muss mit dem Naming-Server verbunden werden.

```
Naming.connect(L"124.33.21.5", 7000);
```

Mit der Methode Naming.lookup() kann der Servant mit den registrierten Namen gesucht werden. Bei erfolgreicher Rückgabe wird jetzt automatisch der Stub erstellt und die Schnittstelle der Servants zurückgegeben. Dabei sollten direkte casts vermieden und anstelle die Methode askForInterface() verwendet werden.

Das folgende Beispiel zeigt, wie die Implementation des Clients aussehen könnten:

Client Anwendung

```
#include <zeusbase/Remote/Naming.h>
#include <zeusbase/ZeusBase.h>
#include "IRemoteTestObject.hpp"
#include "RemoteTestObject_Stub.h" //Used for class registration

int main(int iArgc, char* paArgs[])
{
  //Registers all classes for class factories
  TZeusBase::registerClasses();

  //Needed to register the stub class
  OBJECTFACTORY_REGISTER_CLASS(TRemoteTestObject_Stub);

  //Connect to naming server
  Naming.connect(L"124.33.21.5", 7000);
  ...
  //Lookup for remote object
  IRemoteObject* pRemoteObject = NULL;
  if (Naming.lookup(TString(L"MyObject"),
                    pRemoteObject) == RET_NOERROR)
  {
    IRemoteTestObject* pMyObject = NULL;
    if (pRemoteObject->askForInterface(INTERFACE_IRemoteTestObject,
                                       ICAST(pMyObject)) == RET_NOERROR)
    {
      //Use myobject here
      pMyObject->setFloatValue(12.005);
      ...
      pMyObject->release();
    }
    pRemoteObject->release();
  }

  Naming.disconnect();
  return 0;
}
```

Tabelle 53: Implementation eines Clients.

3.13.4.5 Naming-Service

Der Naming-Service ist primär als lokales Singleton `Naming` implementiert. Es dient dem Registrieren von Objekten mit Namen und Kategorien.

Damit Remote Objects von anderen Anwendungen ermittelt werden können, muss der Naming-Service in dem Sub-Netz als Server verfügbar sein. Der Server wird durch die Anweisung `[nameserver]` gestartet.

Die Client- und die Server-Anwendungen müssen ihre lokale Naming-Services auf den Server verbinden (Siehe Beispiele oben).

Weitere Informationen zur Applikation `[nameserver]` stehen im Kapitel [[nameserver]-Globaler Namesdienst](#).

4 Definition von X-Objekten in XML

Dieser Abschnitt befasst sich mit der statischen Strukturdefinition von X-Objekten in einem Objektbaum. Dieser Objektbaum ist in XML abgebildet, da sich diese Beschreibungssprache optimal für solche Strukturen eignet.

Die X-Objekt Klassen sind im Kapitel X-Objekt Klassen definiert.

4.1 Allgemeine XML Spezifikation

Die Definition in XML ist sehr offen spezifiziert. Ein XML Knoten, welcher ein X-Objekt repräsentiert, kennzeichnet sich lediglich durch folgende Attribute:

- Knotennamen: Gibt dem Objekt einen Namen.
- Attribut `ClassName`: Name der Klasse von welcher das Objekt erzeugt werden soll.
- Attribut `CodeModule`: Name des Code-Moduls, in welchem die Klasse implementiert ist.

➔ Es können beliebige weitere Attribute definiert werden, welche Daten für die konkreten Objekte enthalten. Der XML Knoten kann auch Kinderknoten haben, die keine X-Objekte repräsentieren.

Der Wurzelknoten des XML Knotens muss immer ein Objekt erzeugen, welches die Schnittstelle `IXRootObject` implementiert.

4.1.1 Objekte

Die Spezifikation schreibt nicht vor, dass sich Objekte im XML eindeutig von einander unterschieden. Je nach Anwendung ist es aber erwünscht, dass jedes Objekt eindeutig identifizierbar ist. Das XML Objekt wird durch den Namen des Objekts (Knotennamen) und dem Attribute ID zum Unikat.

Objekt: `<Name><ID>`

Beispiel von XML Objekt Tree

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<RootObject ID="1" ClassName="TXRootObject">
  <SubObject1 ID="1" ClassName="TXObject1"/>
  <SubObject1 ID="2" ClassName="TXObject1"/>
  <SubObject2 ID="1" ClassName="TXObject2"/>
</RootObject>
```

Tabelle 54: Beispiel einer XML Objekt Konfiguration mit eindeutig identifizierbaren Objekten.

4.1.2 Attribute und Variablen

Das Objekt kann Attribute enthalten und Variablen besitzen (Member). Attribute beschreiben das Objekt näher. Die Variablen beinhalten Daten des Objekts.

Die Attribute werden im XML auch als Attribut des Knotens gespeichert. Sie sind untypisiert und werden als Zeichenkette gelesen. Dabei können alle primitiven Werte einfach und schnell abgebildet werden. Attribute zu verwenden bietet den Vorteil, dass das Abbilden wenig Speicher verwendet und der Zugriff schnell ist. Nachteilig ist, dass sie untypisiert sind.

Objekt-Attribute

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<RootObject ID="1"
            ClassName="TXRootObject"
            Label="Begruessung"
            Text="Hallo Welt!">

  <SubObject1 ID="1" ClassName="TXObject1" Status="OK"/>
  <SubObject1 ID="2" ClassName="TXObject1" Status="NA"/>
  <SubObject2 ID="1" ClassName="TXObject2"/>

</RootObject>
```

Tabelle 55: Beispiel einer XML Objekt Konfiguration mit Attributen, die auch als Attribute im XML Objekt eingefügt wurden.

Die Variablen werden im XML als Unterknoten `XMember` repräsentiert. Sie besitzen immer

- einen Namen
- einen Typ (Type= string | int | uint | float | boolean

- den Wert

Der Namen und der Typ werden als XML Attribut abgebildet, während der Wert der Variable als Textknoten abgebildet wird. Das bietet den Vorteil, dass auch komplexere Daten wie zum Beispiel mehrzeiliger Text abgebildet werden kann. Ein weiterer Vorteil gegenüber den Attributen bietet die Typisierung der Daten. Ein Nachteil ist die Zugriffszeit, die etwas höher liegt als beim Lesen und Schreiben der Objekt-Attribute.

Zweites Schema

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<RootObject ID="1" ClassName="TXRootObject">
  <XMember Name="Label" Type="string">Begruessung</XMember>
  <XMember Name="Text" Type="string">Hallo Welt!</XMember>

  <SubObject1 ID="1" ClassName="TXObject1">
    <XMember Name="Status" Type="string">OK</XMember>
  </SubObject1>

  <SubObject1 ID="2" ClassName="TXObject1"/>
    <XMember Name="Status" Type="string">NA</XMember>
  </SubObject1>

  <SubObject2 ID="1" ClassName="TXObject2"/>

</RootObject>
```

Tabelle 56: Beispiel einer XML Objekt Konfiguration mit Variablen, die als Unterknoten zum XML Objekt eingefügt wurden.

4.2 Erweiterte Definition zum ClassLoader

Der XML Objektbaum kann Objekte besitzen, die zur Laufzeit noch nicht bekannt sind. Dabei handelt es sich um so genannte externe Objekte, welche in einem Code-Modul implementiert sind. Diese Code-Module müssen während dem Aufbauen des Baums geladen werden. Beim XML Objekt wird das Attribut `CodeModule="path/filename"` angegeben. Das Attribut setzt sich aus einem relativen Pfad und dem Namen einer Bibliothek (ohne Endung) zusammen. Der Klassenlader (ClassLoader) fügt automatisch die korrekte Endung an (unter Linux mit Endung *.so, unter Windows mit Endung *.dll). Der Basispfad aller Module ist standardmässig auf `./modules` gesetzt.

Weiter können externe XML Referenzen angegeben werden. Diese werden während dem Erstellen der Objekthierarchie geladen. Darin können sich weitere Teilbäume der Objekthierarchie befinden. Die XML Referenzen müssen als Dateinamen mit relativem Pfad angegeben werden. Der Basis-pfad ist auch hier auf `./modules` gesetzt.: Zum Beispiel: `XMLRef="path/filename.xml"`. Diese Angabe kann nicht bei allen

Objekttypen gemacht werden. Das Objekt muss die Schnittstelle `IXLoaderObject` implementieren.

Bei den ladbaren Objekten (`IXLoaderObject`) kann das Attribut `CreateChildren="x"` gesetzt werden, um die Objekthierarchie gestaffelt aufzubauen. Mit dem Wert `x=1` werden alle Kinderobjekte sofort erstellt. Beim Wert `x=0` werden die Kinderobjekte erst erstellt, wenn sie jemand braucht.

Durch das Einsetzen der drei Erweiterungen, können Teilbäume ausgelagert und erst bei Bedarf geladen werden. Zudem können Objekte in Code-Modulen implementiert werden.

Die 3 Erweiterungen

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<RootObject ID="1" ClassName="TXRootObject">
  <ModuleA ID="1"
           ClassName="TModule"
           CodeModule="ModuleA"
           CreateChildren="1">

    <ModuleALogic ClassName="TLogicA">

  </ModuleA>

  <ModuleB ID="1"
           ClassName="TModule"
           CodeModule="test/ModuleB"
           XMLRef="test/ModuleB.xml"
           CreateChildren="1"/>

</RootObject>
```

Tabelle 57: Beispiel welches die 3 Erweiterungen nutzt. Zu beachten ist, dass hier das Attributschema 3 verwendet wird.

5 Entwickeln von Code-Modulen

Damit Bibliotheken (Code-Module) im Zeus-Framework gebraucht werden können, müssen zwei Funktionen exportiert werden.

API	`MODULEXPORT_PRE Retval MODULEXPORT_POST registerLibrary(NAMESPACE_Zeus::ISingletonManager* manager)`
	Diese Funktion wird beim Laden der Bibliothek aufgerufen. Die Bibliothek kann durch den Singleton-Manager alle System-Singletons ermitteln und seine statischen Singleton-Referenzen auf die System-Singletons setzen.

API	`MODULEXPORT_PRE Retval MODULEXPORT_POST unregisterLibrary(NAMESPACE_Zeus::ISingletonManager* )\`
	Diese Funktion wird kurz vor dem Entladen der Bibliothek aufgerufen.

Bei den meisten Bibliotheken wird der Code zur Registrierung immer gleich aussehen. Deshalb kann hier das Makro verwendet werden. Es initialisiert die wichtigsten Singletons und erzeugt einen Logger zur Ausgabe von Informationen (in eine Datei oder in eine Konsole).

API	`MregisterLibrary(libname)`
	Makro zum Registrieren des Code-Moduls beim Zeus-Framework

API	`MunregisterLibrary()`
	Makro zum Abmelden des Code-Moduls beim Zeus-Framework

Initialisieren der Bibliothek

```
MregisterLibrary(L"MeineBibliothek");
MunregisterLibrary();
```

Soll in der Register-Methode oder Unregister-Methode weiteren Code ausgeführt werden, brauchen wir folgende Makros.

API	`MregisterLibrary_Start(libname)`
	Startet das Makro zum Registrieren des Code-Moduls beim Zeus-Framework

API	`MregisterLibrary_End`
	Beendet das Makro zum Registrieren des Code-Moduls beim Zeus-Framework

API	`MunregisterLibrary_Start()`
	Startet das Makro zum Abmelden des Code-Moduls beim Zeus-Framework

API	`MunregisterLibrary_End`
	Beendet das Makro zum Abmelden des Code-Moduls beim Zeus-Framework

Das folgende Beispiel illustriert, wie die Register Methoden verwendet werden können.

Beispiel einer Erweiterung

```
#include <zeusbase/MOM/AbstractModuleSession.h>
#include<TextManager.h>

//Register
MregisterLibrary_Start("MyLibrary")
{
  TString strStwName = L"ITextManager";
  ITextManager* pTextManager = NULL;
  if (manager.getSingleton(strStwName, ICAST(pTextManager))
      == RET_NOERROR)
  {
    TTextManager::getInstance().
      setDelegationInterface(pTextManager);
    pTextManager->release();
  }
}
MregisterLibrary_End;


//Unregister
MunregisterLibrary_Start("MyLibrary")
{
  TTextManager::getInstance().releaseDelegationInterface();
}
MunregisterLibrary_End;
```

Tabelle 58: Dieses Beispiel zeigt, wie die Register und Unregister-Methoden erweitert werden können. Ein Singleton TTextManager wird beim Registrieren der Bibliothek delegiert. Smot kann dieses Singleton nun im Code Module verwendet werden.

Es gibt 2 Arten wie ein Code-Modul gebraucht werden kann:

- Das Code-Modul wird nur zur Erzeugung von Objekten gebraucht
- Das Code-Modul wird durch ein X-Objekt vom Typ `TModule` geladen und eine Session wird erstellt.

Beide Varianten sind auch in Kombination einsetzbar. Am häufigsten wird jedoch die

erste Variante gewählt. Hier eine kurze Erläuterung der Funktionalität und der Unterschiede

5.1 Code Modul zur Erzeugung von Objekten

Die Bibliothek `xy` wird geladen, wenn im Objektbaum ein Objekt erzeugt werden soll, welches das Attribut `CodeModule="xy"` gesetzt hat. Danach wird versucht das Objekt durch eine exportierte Funktion der Bibliothek zu erzeugen. Das Objekt wird direkt zurückgegeben und kann nun in den Objektbaum eingehängt werden.

Die exportierten Funktionen heissen immer **`create`**`TMyClass`, wobei `TMyClass` für einen Klassennamen steht. Der gleiche Klassennamen muss im XML im Attribut `ClassName` angegeben werden.

Um eigene X-Objekt Klassen einfach zu exportieren gibt es im C-Header `zeusbase/MOM/AbstractModuleSession.h` Makros.

Fabrikfunktion der Bibliothek

```
#include <InputCell.h>
#include <zeusbase/MOM/AbstractModuleSession.h>

USING_NAMESPACE_Zeus

MExportXObjectFactory(TInputCell);
```

Tabelle 59: Beispiel zeigt das Exportieren einer Fabrikfunktion zum Erzeugen des X-Objekts `TInputCell`

5.2 Code Modul zum Erstellen einer Session

Ladet ein X-Objekt vom Typ `TModul` eine externe Ressource in Form einer Bibliothek, wird eine Modul-Session erzeugt. Die Session dient fortan zur Kommunikation zwischen dem X-Objekt und der Bibliothek. In einer Session können beliebige Objekte von der Bibliothek erzeugt werden. Diese Möglichkeit dient all jenen Bibliotheken die keine X-Objekte implementieren wollen.

Zur bidirektionalen Kommunikation zwischen der Bibliothek und dem Modul werden 2 Schnittstellen ausgetauscht:

API	
`IModuleSession`	Durch diese Schnittstelle kann auf die Session einer Bibliothek zugegriffen werden. `zeusbase/MOM/Interfaces/IModuleSession.hpp`
`IZeusAPI`	Schnittstelle für den Zugriff auf den X-Objektbaum `zeusbase/MOM/Interfaces/IZeusAPI.hpp`
`TAbstractModuleSession`	Diese Klasse ist eine Hilfsimplementierung um einfach und schnell eigene Modul-Sessionen zu entwickeln. `zeusbase/MOM/AbstractModuleSession.h`

Die Schnittstelle der Ressource wird `IModuleSession` genannt. Durch diese kann das Framework mit der gewünschten Bibliothek kommunizieren und Objekte erzeugen, bzw. anfordern.

Die Schnittstelle des Frameworks ist das `IZeusAPI`. Durch diese Schnittstelle kann die Bibliothek auf das Framework oder auf eine andere Bibliothek des Frameworks zugreifen.

API `zeusbase/MOM/Interfaces/IZeusAPI.hpp`
`Retval MQUALIFIER createObjectOfModule(const IString& rModuleName, const InterfaceID& rInterfaceID, IZUnknown*& rpIface);` Erzeugt oder gibt ein existierendes Objekt einer Modul Session zurück.
`Retval MQUALIFIER executeCommand(Uint uiMode, const IString& rstrTarget, const IString& rstrMainData, const IString& rstrAddData, const IString& rstrStreamData, Uint uiSecurity, Uint uiExecutionMode);` Führt einen Befehl vom Typ `TSimpleCommand` im Framework aus.
`Retval MQUALIFIER getCastedObject(const IString& rPath, const InterfaceID& rIfaceID, IZUnknown*& rpObj);` Gibt ein bereits gecastetes Objekt aus dem X-Objektbaum zurück.

`Retval MQUALIFIER getObject(const IString& rPath, IXObject*& rpObj);`
`Retval MQUALIFIER getObjects(const IString& rstrPath, IXObjectCollection*& rpCollection);`
 	
Gibt ein oder mehrere Objekte aus dem X-Objektbaum zurück.

API	zeusbase/MOM/Interfaces/IZeusAPI.hpp

```
Retval MQUALIFIER getPropertyValue(const IString& rName,
                                   IString& rValue) const;
Retval MQUALIFIER setPropertyValue(const IString& rName, const IString&
rValue);
```

Liest bzw. schreibt eine Einstellung. Die Einstellungen werden dem Property-File entnommen. Diese Einstellungen werden nicht gespeichert.

```
Retval MQUALIFIER getUserValue(const IString& rName, IString& rValue) const;
Retval MQUALIFIER setUserValue(const IString& rName, const IString& rValue);
```

List bzw. schreibt einen Benutzerwert. Die Einstellungen werden der Benutzer-XML Datei entnommen. Diese Einstellungen können gespeichert werden.

```
void MQUALIFIER logMessage(Uint uiMode, const IString& rstrMessage);
```

Schreibt eine Log-Meldung.

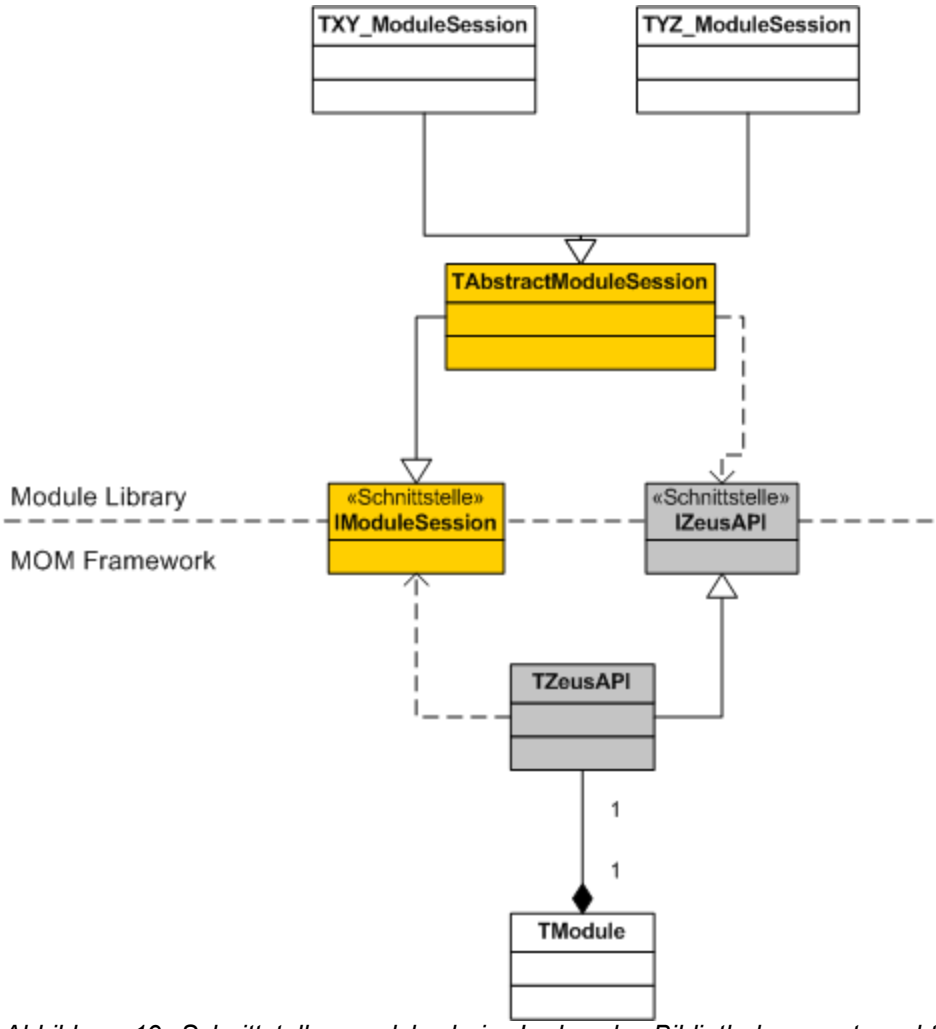


Abbildung 19: Schnittstellen, welche beim Laden der Bibliothek ausgetauscht werden.

Wenn mehrere X-Objekte vom Typ `TModule` die gleiche Bibliothek laden möchten, wird pro X-Objekt eine neue Session erzeugt. Die Bibliothek wird aber nur einmal geladen.

➔ Werden die Bibliotheken erst bei Gebrauch geladen, kann einerseits Speicher und anderseits Zeit beim Hochfahren eingespart werden. Das Attribut im entsprechenden XML Knoten muss mit `CreateChildren="0"` eingestellt werden

Um die Modul-Session oder eigene Klassen einfach zu exportieren gibt es im C-Header `zeusbase/MOM/AbstractModuleSession.h` Makros.

Fabrikfunktion der Bibliothek

```
#include <TestModuleSession.h>

USING_NAMESPACE_Zeus

MExportObjectFactory(IModuleSession, TTestModuleSession);
```

Tabelle 60: Beispiel zeigt das Exportieren einer Fabrikfunktion zum Erzeugen von normalen Objekten, wie zum Beispiel eine Modul-Session

5.3 Beispiel einer Code-Modul Implementation

Im folgenden Beispiel wird gezeigt, wie ein Code-Modul zu entwickeln ist. Das Code-Modul beinhaltet ein X-Objekt, welches exportiert wird.

Das Beispiel setzt sich aus 3 Dateien zusammen:

- `ExampleModule.cpp` : Exportierte Funktionen der Bibliothek
- `XObjectExample.h`: Definition des X-Objekts
- `XObjectExample.cpp`: Implementation des X-Objekts

Alle exportieren Funktionen

```
#include <stdio.h>
#include <XObjectExample.h>
#include <zeusbase/System/XObjectFactory.h>
#include <zeusbase/System/LibraryManager.h>
#include <zeusbase/MOM/AbstractModuleSession.h>

USING_NAMESPACE_Zeus

////////////////////////////////////////////
// Exported Factories
MExportXObjectFactory(TXObjectExample );
////////////////////////////////////////////

MregisterLibrary(L"ExampleModule");
MunregisterLibrary();
```

Tabelle 61: Dieser Code steht in der CPP Datei ExampleModule.cpp

Das `TXObjectExample` ist ein normales X-Objekt. Welche Möglichkeiten und welche Makros noch zur Verfügung stehen um X-Objekte zu erstellen, ist im Kapitel [MOM 2] auf

Seite 87 beschrieben.

Definition des X-Objekts

```
class TXObjectExample : public TXObject
{
  public:
    TXObjectExample(IXMLNode& rNode);

    //Methods of IXObject
    virtual bool MQUALIFIER freeze();
    virtual bool MQUALIFIER unfreeze();
...
  protected:
    virtual ~TXObjectExample();
...
};
```

Tabelle 62: Definition des X-Objekts. Die Makros dienen der Registrierung bei der Objektfabrik.

Implementation des X-Objekts

```
TXObjectExample::TXObjectExample(IXMLNode& rNode) : TXObject(rNode)
{...}

TXObjectExample::~TXObjectExample()
{...}

bool MQUALIFIER TXObjectExample::freeze()
{
  bool bRetval = TXObject::freeze();
  if (bRetval) {...}
  return bRetval;
}

bool MQUALIFIER TXObjectExample::unfreeze()
{
  bool bRetval = TXObject::unfreeze();
  if (bRetval) {...}
  return bRetval;
}
```

Tabelle 63: Implementation des X-Objekts.

Die Methoden `freeze()` und `unfreeze()` sind im Kapitel [MOM 2] Seite 87 erklärt.

6 Das Applikationen-Framework

Dieses Kapitel beschreibt die Anwendungen des Zeus-Frameworks und deren Verwendung.

6.1 Basis Applikation [zeus]

Die Applikation `[zeus]` wird als dynamische Applikation für unterschiedlichste Arbeiten und Tools verwendet. Durch eine XML-Konfiguration kann die Applikation beliebige Objekthierarchien aufbauen. Das ermöglicht eine hohe Dynamik und Wiederverwendbarkeit von Softwarekomponenten.

Diese Applikation ist eine reine Konsolen-Applikation und kann sehr einfach zum Erstellen von Clustering und Server-Diensten verwendet werden.

6.1.1 Konfiguration

Die Applikation `[zeus]` wird durch 2 Dateien konfiguriert. Die erste Datei ist die Konfigurationsdatei des Frameworks. Sie beinhaltet Angaben und Einstellungen zum Framework und kann durch kundenspezifische Einstellungen erweitert werden. Standardmässig heisst die Datei `zeus.properties`.

API	
`zeus.LibraryManager`	Dient als Angabe, welche Systemkomponenten (Services) direkt vom Framework geladen und verwaltet werden sollen. Diese Systemkomponenten werden hier als eindeutige Namen angegeben. Die Angabe ist eine Aufzählung (kommagetrennt).
`zeus.LibraryManager.<Name>`	Die einzelnen Systemkomponenten werden dann mit dieser Einstellung konfiguriert. Dabei wird der Namen der Bibliothek (Code Modul) in der Einstellung angegeben, welche vom Framework geladen werden soll (Platzhalter `<Name1>`). Rechts muss dann ein Pfad zugewiesen werden, welcher dem Bibliothekspfad entspricht.
`zeus.LibraryManager.Security`	Ist diese Einstellung =1, werden alle Code-Module mit einem Sicherheitsdienst verifiziert. Nur registrierte Module werden zugelassen.
`zeus.LoggerManager`	Diese Angabe setzt den gewünschten Logging-Dienst.
`zeus.LoggerManager.ConfigFile`	Der Logging-Dienst braucht auch Einstellungen. Hier wird angegeben, welche Einstellungen verwendet werden sollen. Es kann ein Dateinamen angegeben werden, oder mit der Angabe `BasicConfiguration` wird eine Standardkonfiguration geladen.

API	
`zeus.SecurityManager. FileName`	Setzt die Datei der registrierten Modulen für den Security- Manager.
`zeus.SettingsManager. UserDataFile`	Optional. Diese Einstellung dient zum Laden eines XML Dokuments für Userdaten.

Ein Beispiel soll die Anwendung der Konfiguration verdeutlichen:

Beispiel von zeus.properties

```
#-------------------------------------------------------------
# ZEUS CONFIGURATION FILE
#-------------------------------------------------------------
# This file is used to configure the zeus framework
#-------------------------------------------------------------


#-------------------------------------------------------------
# List of all system services. The services are
# comma separated
zeus.LibraryManager=XML_Service,LOG_Service
zeus.LibraryManager.Security=0

# Linux config
zeus.LibraryManager.XML_Service=./modules/XML_Service.so
zeus.LibraryManager.LOG_Service=./modules/LOG_Service.so

zeus.LoggerManager=LOG_Service
zeus.LoggerManager.ConfigFile=zeus.log.properties
```

Tabelle 64: Beispiel einer Framework Konfiguration

Die zweite Datei dient dem Erstellen des Module Object Model (MOM), der eigentlichen Objekthierarchie. Sie wird als XML-Datei geschrieben (Siehe Definition von X-Objekten in XML).

Der Aufbau ist je nach Anwendung sehr verschieden. Zum Beispiel die Testanwendung zum Zeus-Framework:

Beispiel Testapplikation

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<zeus ClassName="TSystemManager">
  <LocalPipeRegistry ClassName="TLocalPipeRegistry"
    RegistryPort="7278"
    RegistryAddress="127.0.0.1"/>
  <Services ClassName="TModule" CreateChildren="0">
    <XML_Service ClassName="TModule"/>
    <LOG_Service ClassName="TModule"/>
  </Services>
  <CellSystem1 ClassName="TModule" CreateChildren="1">
    <WorkCell ClassName="TCellEnvironment"
       CreateChildren="1"
       ModuleLibrary="WorkCellModule"
       Security="1">
      <InputPipe Name="Pipe1"/>
      <InputPipe Name="Pipe2"/>
      <Cell ClassName="TCell" CreateChildren="1">
        <Kernel ClassName="TWorkCell" CodeModule="WorkCellModule"/>
      </Cell>
      <Survey ClassName="TCellSurvey"/>
    </WorkCell>
  </CellSystem1>
</zeus>
```

Tabelle 65: Beispiel einer MOM Konfiguration. Die Testapplikation erzeugt eine Arbeitszelle, die WorkCell.

6.1.2 Kommandozeilen Parameter

Die Zeus-Framework Applikation kann durch Kommandozeilen-Parameter zusätzlich beeinflusst werden. Folgende Parameter sind zulässig:

API	
`-conf=<file>`	Durch diese Angabe wird nicht die MOM-Konfiguration `zeus.xml` gelesen, sondern die Datei `<file>`.
`-prop=<file>`	Durch diese Angabe wird nicht die Framework-Konfiguration `zeus.properties` gelesen, sondern die Datei `<file>`.
`-reg`	Registriert ein Code-Modul bei dem Sicherheitsdienst. Dieses Kommando braucht noch die Angaben `-file` und `-module`.
`-unreg`	Entfernt ein Code-Module von der Datenbank des Sicherheitsdiensts. Dieses Kommando braucht die Angabe `-module`.
`-file=<file>`	Mit dieser Angabe kann ein Code-Modul mit Pfad angegeben werden.
`-module=<name>`	Gibt den Namen an eines Code-Moduls. Dieser ist muss dem Dateinamen des Code-Moduls entsprechen (ohne Endung)

API	
`-list`	Zeigt alle registrierten Code-Module an.
`--version`	Zeigt die Version der Applikation an
`--help` `-?`	Zeigt die Beschreibung der Kommandos in der Konsole an.

Weiter können beliebige Parameter angegeben werden. Alle Parameter, wie auch die Framework-Konfigurationen stehen den Code-Modulen zur Verfügung (Siehe SettingsManager). Alle Einstellungen der Framework-Konfiguration können durch Übergabeparameter überschrieben werden.

Übergabeparameter überschreiben

```
> zeus -conf=test.properties zeus.LoggerManager.ConfigFile=test
```

Tabelle 66: Startet die zeus-Applikation mit der Framework-Konfiguration „test.properties“. Die Einstellung „zeus.LoggerManager.ConfigFile“ wird aber beim Starten überschrieben.

6.2 [rmicc]- RMI Compiler für C++

Die Applikation [rmicc] wird gebraucht um Stub- und Skeleton-Dateien aus einem Remote-Interface zu generieren.

- Input: Remote Interface Definition (zum Bsp. IMessagePoos.hpp)
- Output:
  - Skeleton Klasse (zum Bsp. MessagePool_Skel.h und MessagePool_Skel.cpp)
  - Stub Klasse (zum Bsp. MessagePool_Stub.h und MessagePool_Stub.cpp)

Der Servant muss manuell implementiert werden (Siehe Verwenden von RMI). [rmicc] generiert noch keine Vorimplementationen für den Servant.

Folgende Optionen können verwendet werden:

API	
`--check_only`	Prüft ob das angegebene Remote Interface ok ist. Es werden keine Stub- und Skeleton Dateien generiert.
`--output=<path>`	Ausgabepfad der generierten Dateien
`--iinclude=<path>`	Include-Pfad des Remote Interface. Der Pfad wird als `#include <[path][interfacefile]>` in die Dateien genertiert.

API	
`--stubinclude=<path>`	Include Pfad für Stub-Klasse. Der Pfad wird als `#include <[path][stubheader]>` in die Dateien genertiert.
`--skelinclude=<path>`	Include Pfad für Skeleton-Klasse. Der Pfad wird als `#include <[path][skeletonheader]>` in die Dateien genertiert.
`--interfaces=<list>`	Liste der bekannten Remote Interfaces.
`--help`	Ausgabe der Hilfe

Beispiele von rmicc

```
> rmicc --check-only ./IMessagePool.hpp
> rmicc ./IMessagePool.hpp
> rmicc --iinclude=zeusbase/Messaging/Interfaces ./IMessagePool.hpp
```

Tabelle 67: Zeigt eingige Beispiele der [rmicc] Anwendung.

6.3 [nameserver]- Globaler Namesdienst

Um eine Applikation auf verschiedene Computer zu verteilen, braucht es einen Dienst, bei welchem sich die Remote Objects der Applikationen registrieren können. Ein solcher Dienst ist der Naming-Service.

Der Naming-Service dient der Registrierung von Objekten, die auf entfernten Computersystemen leben. Dieser Dienst ist eine sehr wichtige Komponente, wenn mit verteilten Applikationen gearbeitet werden möchte. Durch das Registrieren sind diese Objekte für andere Anwendungen zugänglich und können mit dem RMI für C++ über Netzwerk verwendet werden.

Die Applikation `[nameserver]` wurde auf der Basis der Applikation `[zeus]` erstellt. Der Naming-Service ist im Code-Module `libNameServer.so` implementiert. Die Konfiguration des Dienst, muss in der Datei `nameserver.properties` vorgenommen werden. Hier muss die IP-Adresse und der TCP-Port eingetragen werden.

Durch die Skript-Datei `nameserver` kann der Dienst gestartet werden.

API	
`server.Address`	Adresse des Servers
`server.Port`	Port des Servers

Konfiguration des Naming-Service

```
#-------------------------------------------------------
# NAMESERVER CONFIGURATION FILE
#-------------------------------------------------------
# This file is used to configure the name server
#-------------------------------------------------------


#-------------------------------------------------------
# List of all system services. The services are
# comma separated
zeus.LibraryManager=XML_Service,LOG_Service

# Windows config
zeus.LibraryManager.XML_Service=./modules/XML_Service.dll
zeus.LibraryManager.LOG_Service=./modules/LOG_Service.dll
zeus.SecurityManager.FileName=./Security.db

server.Address=127.0.0.1
server.Port=7278
```

Tabelle 68: Konfiguration des Naming-Service.

Die Adresse der Netzwerkschnittstelle kann auch direkt über die Konsole konfiguriert werden:

Starten des Naming-Service

```
> nameserver "server.Address=10.2.0.178" "server.Port=70"
```

Tabelle 69: Starten des Naming-Service über die Konsole mit anderer Adresse und Port.

6.4 [messageserver]- Zeus Message Service ZMS

Der Message-Server dient zum Verteilen von Meldungen. Dies wird vor allem bei Clustering Applikationen verwendet. Applikationen kommunizieren über Meldungen miteinander in einer Peer-To-Peer Topologie.

Die Applikation `[messageserver]` wurde auf der Basis der Applikation `[zeus]` erstellt. Der Message-Server wird in der Datei `messageserver.properties` konfiguriert.

API	
`server.Address`	Adresse des Servers
`server.Port`	Port des Servers

API	
`naming.Address`	Adresse des Naming-Service
`naming.Port`	Port des Naming-Service

Konfiguration des Message-Servers

```
#-------------------------------------------------------
# MESSAGESERVER CONFIGURATION FILE
#-------------------------------------------------------
# This file is used to configure the message server
#-------------------------------------------------------


#-------------------------------------------------------
# List of all system services. The services are
# comma separated
zeus.LibraryManager=XML_Service,LOG_Service

# Windows config
zeus.LibraryManager.XML_Service=./modules/XML_Service.dll
zeus.LibraryManager.LOG_Service=./modules/LOG_Service.dll
zeus.SecurityManager.FileName=./Security.db
#zeus.LoggerManager=LOG_Service1
#zeus.LoggerManager.Console=1
#zeus.LoggerManager.ConfigFile=nameserver.log.properties

server.Address=127.0.0.1
server.Port=7881

naming.Address=127.0.0.1
naming.Port=7278
```

Tabelle 70: Konfiguration des Message-Servers. Neben der Adresse des Servers muss auch die Adresse des Naming-Service angegeben werden.

6.5 [webserver]- Einfacher WebServer

Das Zeus-Framework beinhaltet einen kleinen Webserver, welcher HTTP 1.0 unterstützt. Zur Zeit dient er lediglich als Beispielapplikation.

Die Applikation `[webserver]` wurde auf der Basis der Applikation `[zeus]` erstellt. In der Datei `webserver.properties` wird der Webserver konfiguriert.

API	
`server.Address`	Adresse des Servers

API	
`server.Port`	Port des Servers
`server.DocumentRoot`	Root-Verzeichnis des Web-Servers

Folgendes Beispiel zeigt eine mögliche Konfiguration:

Konfiguration des WebServer

```
#------------------------------------------------------------
# WEBSERVER CONFIGURATION FILE
#------------------------------------------------------------
# This file is used to configure the web server
#------------------------------------------------------------


#------------------------------------------------------------
# List of all system services. The services are
# comma separated
zeus.LibraryManager=XML_Service,LOG_Service

# Windows config
zeus.LibraryManager.XML_Service=./modules/XML_Service.dll
zeus.LibraryManager.LOG_Service=./modules/LOG_Service.dll
zeus.SecurityManager.FileName=./Security.db
#zeus.LoggerManager=LOG_Service1
#zeus.LoggerManager.Console=1
#zeus.LoggerManager.ConfigFile=nameserver.log.properties

server.DocumentRoot=D:\Zeus-Framework\Doc\api\zeusbase\
server.Address=127.0.0.1
server.Port=8080
```

Tabelle 71: Konfiguration des Webservers.

7 Erweitern des Zeus-Frameworks

Dieses Kapitel befasst sich mit der Erweiterung des Zeus-Frameworks. Das kann sinnvoll sein, wenn ein spezielles Framework auf Basis des Zeus-Frameworks entstehen soll. Folgende Punkte werden behandelt:

- Wie kann das Zeus-Framework als Basis verwendet werden
- Welche Klassen dienen als Basis und können erweitert werden
- Welche Einstellungen werden von Zeus-Framework benötigt

7.1 Der Frameloader

Damit die Framework-Applikation den X-Objektbaum laden und verwalten kann, braucht es einen Frameloader. Das Zeus-Framework stellt eine Hilfsklasse, den `TAbstractFrameLoader` zur Verfügung. Diese Hilfsimplementation bietet folgende Methoden:

API	`zeusbase/AbstractFrameLoader.h`
	`void initialize()` Das Framework wird initialisiert. Dabei werden alle Einstellungen (properties) gelesen.
	`bool canStartup()` Prüft, ob das Framework gestartet werden darf.
	`Retval startup()` Startet das Framework auf. Jetzt werden alle X-Objekte aus dem XML erstellt und der Objektbaum aufgebaut.
	`Int run()` Mit dieser Methode wird die Applikation gestartet. Bei Konsolen Anwendungen wird die TConsoleMainThread Klasse verwendet. Bei CTRL+C wird die Applikation gestoppt.
	`Retval shutdown()` Der X-Objektbaum wird heruntergefahren und die Objekte freigegeben.

Der TAbstractFrameLoader kann von Framework-Entwicklern abgeleitet werden.

Die Klasse verlangt beim Erstellen einen Namensraum für die Einstellungen und die Argumente des Programms.

7.1.1 Erweitern der Initial-Methoden

Die `initialize()`-Methode arbeitet verschiedene Initial-Methoden ab. Jede dieser Methoden kann von Framework-Entwicklern überschrieben und mit eigenem Code erweitert werden.

API	`zeusbase/AbstractFrameLoader.h`
	`void initSystem()` Initialisiert die Attribute `m_strPropertyFileName` und `m_strConfigFileName`.
	`Retval initProperties()` Lädt die Einstellungen aus der Datei (`m_strPropertyFileName`) und initialisiert das Singleton `SettingsManager`.
	`Retval initSecurity()` Initialisiert die Sicherheitseinstellungen des Frameworks. Die Einstellung `<ns>.SecurityManager.FileName` wird gelesen. Weiter werden alle Applikationsargumente dem SettingsManager übergeben, damit alle Module (Bibliotheken) darauf zugreifen können. am Schluss werden noch die Userdaten initialisiert, wenn der Property-Eintrag `<ns>.SettingsManager.UserDataFile=[filename]` konfiguriert ist.
	`Retval initSystemLibraries()` Die System-Bibliotheken werden hier geladen. Folgende Einstellungen werden verwendet: – `<ns>.LibraryManager`: Liste aller System-Bibliotheken – `<ns>.LoggerManager`: Name der Logging-Bibliothek – `<ns>.LoggerManager.ConfigFile`: Konfigurationsdatei des Loggers – `<ns>.LibraryManager.Security`: Sicherheitsdienst aktivieren
	`Retval shutdown()` Der X-Objektbaum wird heruntergefahren und die Objekte freigegeben.

7.1.2 Registrieren der X-Objekte und Singletons

Der TAbstractFrameLoader registriert bereits alle X-Objekte des Zeus-Frameworks. Kommen in einem erweiterten Framework weitere X-Objekte dazu, müssen diese ebenfalls registriert werden, falls die Implementation sich direkt in der Applikation befindet (Siehe Kapitel Implementation eines X-Objekts).

Wichtig ist, dass die Registrierung im Konstruktor als erstes erfolgt.

registrieren der X-Objekts

```
#include <MyFrameworkLoader.h>
#include <MySystemManager.h>
#include <TestObject.h>
#include <TestModule.h>


//TAbstractFrameLoader registriert bereits alle
// ZeusBase Framework X-Objekte
TMyFrameworkLoader::TMyFrameworkLoader(int argc, char *argv[])
 : TAbstractFrameLoader(TString(L"my"), argc, argv)
{
  registerClasses();
}

//Registriert die eigenen X-Objekte
void TMyFrameworkLoader::registerClasses()
{
  XOBJECTFACTORY_REGISTER_ROOT(TMySystemManager);
  XOBJECTFACTORY_REGISTER_SUB(TTestObject);
  XOBJECTFACTORY_REGISTER_SUB(TTestModule);
  ...
}
```

Tabelle 72: Registrieren der eigenen X-Objekten mit TAbstractFrameLoader.

Beim Registrieren der X-Objekte können auch weitere wichtige Registrierungen vorgenommen werden, wie zum Beispiel das Registrieren von Singletons (Siehe Kapitel Singletons).

registrieren der Singletons

```
#include <PropertyManager.h>

...

//Registriert die eigenen X-Objekte und Singletons
void TMyFrameworkLoader::registerClasses()
{
  XOBJECTFACTORY_REGISTER_ROOT(TMySystemManager);
  XOBJECTFACTORY_REGISTER_SUB(TTestObject);
  XOBJECTFACTORY_REGISTER_SUB(TTestModule);
  ...
  TString strName = L"IPropertyManager";
  TSingletonManager::getInstance().addSingleton(
    strName,
    (IPropertyManager&)TPropertyManager::getInstance());
}
```

Tabelle 73: Registrieren der eigenen Singletons mit TAbstractFrameLoader.

7.1.3 Wichtige Attribute der TAbstractFrameLoader-Klasse

Die `TAbstractFrameLoader`-Klasse verfügt über diverse Attribute, die für abgeleitete Klassen zugänglich sind:

API	`zeusbase/AbstractFrameLoader.h`
	`TDirectory* m_pWorkingDir;` Arbeitsverzeichnis der Applikation.
	`TDirectory* m_pBaseDirectory;` Verzeichnis, in welchem sich die Applikation befindet.
	`IXObject* m_pRootObject;` Wurzel Objekt des X-Objektbaums. Diese Variable ist erst gültig, wenn der X-Objektbaum geladen wurde
	`TPropertyFile* m_pSystemSettings;` Property Datei mit allen Einstellungen der Applikation. Diese Variable ist erst nach dem Aufruf von `initProperties()` gültig.
	`TString m_strPropertyFileName;` Name und Pfad der Property-Datei. Wird mit `initSystem()` gesetzt.

API	zeusbase/AbstractFrameLoader.h

```
TString m_strConfigFileName;
```

Dateinamen der XML Konfiguration. Die X-Objekte werden aus dieser Datei gelesen und erstellt.

A Installation des Framework Zeus

Das Zeus-Framework umfasst diverse Bibliotheken, Code-Module und Programme. In diesem Kapitel werden die Dateien aufgelistet und wohin sie installiert werden.

A.1 Installation unter Linux

Es sind folgende Schritte auszuführen um das Zeus-Framework unter Linux zu installieren:

1. Herunterladen der Source Dateien (http://www.xatlantis.ch/downloads.html)
2. Extrahieren des Archivs in ein lokales Verzeichnis
3. Ins Verzeichnis `[./ZeusFramework]` wechseln
4. Erstellen der Binaries
5. Installieren der Binaries

Installieren des Zeus-Frameworks

```
> gzip -d ./ZeusFramework_x_x_x.tar.gz
> tar -xf ./ZeusFramework_x_x_x.tar
> cd ZeusFramework
> ./configure
> make
> make install
```

Tabelle 74: Installieren des Zeus-Frameworks unter Linux

A.1.1 Basis Dateien des Zeus-Frameworks

Folgende Tabelle enthält die Basis Dateien des Zeus-Frameworks

Name	Ort	Beschreibung
`libZeusBase.so`	`/usr/local/lib`	Basis Bibliothek des Zeus-Frameworks
`libZeusMath.so`	`/usr/local/lib`	Mathematische Bibliothek des Zeus-Frameworks

Name	Ort	Beschreibung
libZeusCell.so	/usr/local/lib	Cell Computing Network Bibliothek
libXML_Service.so	/usr/local/bin/modules	XML Bindings von Xerces XML Parser
libLOG_Service.so	/usr/local/bin/modules	Logging Bindings von Log4CXX
rmicc	/usr/local/bin	Compiler für RMI für C++
zeus	/usr/local/bin	Framework Plattform
zeus.properties	/usr/local/etc/ccm	Standardeinstellungen der Framework Plattform
zeus.log.properties	/usr/local/etc/ccm	Standardeinstellungen für Logger

A.1.2 Naming Service

Dateien des Naming-Service

Name	Ort	Beschreibung
libNameServer.so	/usr/local/bin/modules	Code Module welches den Naming-Service implementiert.
nameserver	/usr/local/bin	Script zum Starten des Naming-Service
nameserver.xml	/usr/local/etc/ccm	X-Objekt Konfiguration
nameserver.properties	/usr/local/etc/ccm	Konfiguration des Servers
nameserver.log. properties	/usr/local/etc/ccm	Konfiguration des Loggers

A.1.3 Message-Server

Dateien des Message-Servers:

Name	Ort	Beschreibung
libMessageServer.so	/usr/local/bin/modules	Code Module welches den Message-Server implementiert.
messageserver	/usr/local/bin	Script zum Starten des Message-Servers
messageserver.xml	/usr/local/etc/ccm	X-Objekt Konfiguration
messageserver.	/usr/local/etc/ccm	Konfiguration des Servers

Name	Ort	Beschreibung
`properties`		
`messageserver.log. properties`	`/usr/local/etc/ccm`	Konfiguration des Loggers

A.1.4 Webserver

Dateien des Webservers:

Name	Ort	Beschreibung
`libWebServer.so`	`/usr/local/bin/modules`	Code Module welches den Webserver implementiert.
`webserver`	`/usr/local/bin`	Script zum Starten des Webservers
`webserver.xml`	`/usr/local/etc/ccm`	X-Objekt Konfiguration
`webserver.properties`	`/usr/local/etc/ccm`	Konfiguration des Servers
`webserver.log. properties`	`/usr/local/etc/ccm`	Konfiguration des Loggers

A.2 Installation unter Windows

Es sind folgende Schritte auszuführen um das Zeus-Framework unter Windows zu installieren:

1. Herunterladen der Source Dateien (http://www.xatlantis.ch/downloads.html)
2. Extrahieren des Archivs in ein lokales Verzeichnis
3. Erstellen der Binaries (mit Borland Builder oder VC++)

A.2.1 Basis Dateien des Zeus-Frameworks

Folgende Tabelle enthält die Basis Dateien des Zeus-Frameworks

Name	Ort	Beschreibung
zeusbase.dll	./bin	Basis Bibliothek des Zeus-Frameworks
zeusmath.dll	./bin	Mathematische Bibliothek des Zeus-Frameworks
zeuscell.dll	./bin	Cell Computing Network Bibliothek
XML_Service.dll	./bin/modules	XML Bindings von Xerces XML Parser
LOG_Service.dll	./bin/modules	Logging Bindings von Log4CXX
rmicc.exe	./bin	Compiler für RMI für C++
zeus.exe	./bin	Framework Plattform
zeus.properties	./bin	Standardeinstellungen der Framework Plattform
zeus.log.properties	./bin	Standardeinstellungen für Logger

A.2.2 Naming Service

Dateien des Naming-Service

Name	Ort	Beschreibung
NameServer.dll	./bin/modules	Code Module welches den Naming-Service implementiert.
nameserver.bat	./bin	Script zum Starten des Naming-Service
nameserver.xml	./bin	X-Objekt Konfiguration
nameserver.properties	./bin	Konfiguration des Servers
nameserver.log. properties	./bin	Konfiguration des Loggers

A.2.3 Message-Server

Dateien des Message-Servers:

Name	Ort	Beschreibung
MessageServer.dll	./bin/modules	Code Module welches den Message-Server implementiert.

Name	Ort	Beschreibung
`messageserver.bat`	`./bin`	Script zum Starten des Message-Servers
`messageserver.xml`	`./bin`	X-Objekt Konfiguration
`messageserver.properties`	`./bin`	Konfiguration des Servers
`messageserver.log.properties`	`./bin`	Konfiguration des Loggers

A.2.4 Webserver

Dateien des Webservers:

Name	Ort	Beschreibung
`WebServer.dll`	`./bin/modules`	Code Module welches den Webserver implementiert.
`webserver.bat`	`./bin`	Script zum Starten des Webservers
`webserver.xml`	`./bin`	X-Objekt Konfiguration
`webserver.properties`	`./bin`	Konfiguration des Servers
`webserver.log.properties`	`./bin`	Konfiguration des Loggers

B Versionsgeschichte

Änderungen am Dokument sind hier festgehalten

Version	*Beschreibung*	*Datum*
0.1	Projektarbeit Cell Computing Model	26.06.05
0.2	Refactoring des Frameworks	09.09.05
0.3	Neues Dokument für Zeus-Framework	23.12.05
0.3.3	Anpassung an das Framework 0.3.3	01.09.06
0.4.1	Anpassung an das Framework 0.4.1	22.03.07
0.4.2	Dokumentation des RMI und der Applikationen des Zeus-Frameworks	09.04.07
0.5.0	Neue Klassen für Networking	08.09.07
0.6.0	- Attributschema entfernt - Neue X-Objekt Methoden für Variablen - Krypto-Klassen für Stream und Hash - XML File Klasse - X-Objektbaum Synchronisation	27.12.07
0.6.2	- detailliertere Dokumentation von Klassen - GUID als InterfaceID	07.05.2008
0.6.3	Renamed document to ZeusBase.pdf - Array-Pointer Klasse	01.06.2008
0.6.4	Auslagern der allgemeinen Framework- Erklärung in ZeusFramework.pdf	13.09.2008

C Literaturverzeichnis

[STWIN]: Benjamin Hadorn, Programmieranleitung zum StuderWINFrame, 2003

[PTHREAD]: Xavier Leroy, Manual Linux Threads,

[POSIXTH]: Ulrich Drepper, Ingo Molar, The Native POSIX Thread Library for Linux, 2005, 2005

[XML02]: Apache, ,

D Index

M

N

O

P

R

S

T