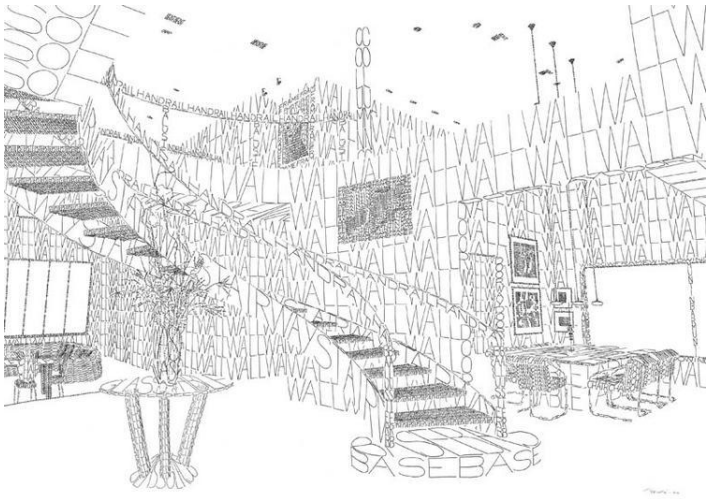
UNIVERSITÄT LEIPZIG

SOFTWARE DESIGN PATTERNS
SOMMERSEMESTER 2009

Muster für Ereignisbearbeitung – Proactor, Asynchronous Completion Token, Acceptor-Connector

Autor:
Martin CZYGAN

Betreuer:
Frank SCHUMACHER
Axel NGONGA
Martin GEBAUER



4. August 2009

Inhaltsverzeichnis

Teil I
Theorie

1 Design Patterns

1.1 Entwicklung und Einteilung

Entwurfsmuster stellen eine Menge an Regeln auf, die beschreiben, wie eine bestimmte Aufgabe im Bereich der Softwareentwicklung gelöst werden kann [Pree, 1994]. Die Forschung um und die Nutzung von Entwurfsmusters für (objekt-orientierte) Software begann um 1980. Ein bekanntes, zuerst in Smalltalk implementiertes Muster, ist das Model-View-Controller Framework gewesen [Cooper, 1998]. 1995 erschien das einflussreiche [Cooper, 1998] Buch *Design Patterns for Object Oriented Software Development* [Gamma et al., 1995], welches 23 Muster beschrieb, die unterteilt wurden in *Erzeugungsmuster* (Creational Patterns), *Strukturelle Muster* (Structural Patterns) und *Verhaltensmuster* (Behavioural Patterns). Eine weitere Einteilung der Muster ist möglich: So zählen etwa Ausnahmebehandlung [Multiple, 2009] oder Vererbung zu *fundamentalen Mustern*; Scheduler zu *Nebenläufigkeitsmustern*, ... Grand [Grand, 2009] stellt 18 Kategorien auf, welche insgesamt 130 Muster enthalten.
Entwurfsmuster haben eine besondere Bedeutung bei der Entwicklung von Frameworks, welche erst eine Ausschöpfung von objekt-orientiertem Design und objekt-orientierter Programmierung ermöglichen [Pree and Sikora, 1997].
Design Pattern sind ein weitläufig erschlossenes Froschungsfeld wie 49900 Einträge in der Publikationsdatenbank von Google[1] zeigen.

1.2 Beschreibung von Entwurfsmustern

Entwurfsmuster können neben der Definition von Pree [Pree, 1994] noch wie folgt beschrieben werden:

- Ein Muster präsentiert eine Lösung für ein *wiederkehrend*[2] auftretendes Architekturproblem, welches in bestimmten Planungsphasen auftritt [Buschmann et al., 1996].
- Muster identifizieren und spezifizieren Abstraktionen, welche oberhalb von Klassen, Instanzen oder Komponenten liegen [Gamma et al., 1993].

Da im Verlauf der Arbeit drei Muster vorgestellt werden, wird im Folgenden die Schablone für die Beschreibung von Entwurfsmustern **Kontext** ↔ **Problem** ← **Lösung** in Stichworten skizziert.

- **Kontext** – Beschreibung der Situation, in der das Problem auftritt; kann vage sein; dient zur Unterstützung bei der Patternwahl.
- **Problem** – Was ist das genau Designproblem, das gelöst werden muß? Welche Kräfte (*forces*) wirken? Kräfte = *Requirements*, *Constraints*, *Properties*; Fragen nach Effizienz, Skalierbarkeit, Komplexität, Restriktionen, Protokollen, Eigenschaften

[1] http://scholar.google.de/scholar?q="design+patterns"
[2] Hervorhebung durch Autor.

- **Lösung** – Lösungsschema (UML, ...); Ausgleich zugeordneter Kräfte; Struktur und Laufzeitverhalten;

2 Ereignisbearbeitung

Anwendungsfälle zur Ereignisbearbeitung treten bevorzugt in Netzwerken auf, in denen verschiedenen Komponenten und Systeme miteinander parallel kommunizieren. Die folgenden Muster beschreiben Szenarien im Netzwerk in unterschiedlichen Komponenten. Das **Proactor/Reactor** Muster fokussiert auf eine *einzelne Komponente*, welche parallele Anfragen performant verarbeiten muß. Das **Asychronous Completion Token** Muster beschreibt einen Nachrichtenaustausch zwischen *zwei Komponenten*, welcher sich *identifizieren* läßt, und der eine robuste Implementierung asynchroner Serviceabwicklung darstellt. Das **Acceptor-Connector** Muster schließlich hat eine skalierbare Implementierung einer *Menge von Services* zwischen einer *Menge von Komponenten* zum Ziel, wobei der Hauptgedanke in einer Trennung von Verbindung und Service, von Vorbereitung und Durchführung, liegt.
Die Pattern stammen alle aus den Jahren 1996–1997 und wurden von Schmidt [Pyarali et al., 1997][Schmidt, 1996b][Schmidt, 1996a] und dessen Koautoren auf der *Pattern Language of Programs* Konferenz[3] vorgestellt.
In den vorgestellten Mustern findet man oft eine Dispatcher-Komponente. Eingebettet in einem generellen *Handler* läßt sich diese Notation Ende der 1970er Jahre bei Yourdon [Yourdon and Constantine, 1979] finden, und Ferg [Ferg, 2006] bemerkt, daß die Autoren dies (Abb. 1) ein Muster oder *Design Pattern* genannt hätten, falls der Begriff schon existiert hätte.

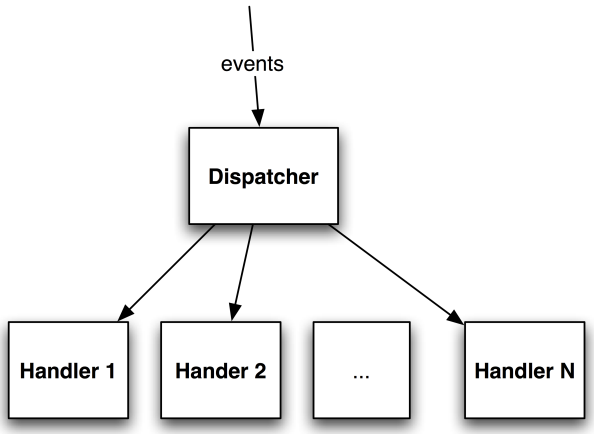


Abbildung 1: Handler *Pattern*

[3] http://www.hillside.net/plop

3 Proactor

Das Proactor-Muster erlaubt es ereignisorientierten Applikationen, Service-Anfragen zu verteilen (to demultiplex) und zu verarbeiten (to dispatch). Dabei werden die Vorteile von Nebenläufigkeit genutzt, wobei die Implementierung von Asynchronität an vorhandene Infrastruktur delegiert wird. Das Proactor Muster entkoppelt I/O Nebenläufigkeit von Prozess Nebenläufigkeit, welche in einer $n : m$–Beziehung stehen.
Ein typisches Beispiel sind Web-Server, welche eine Menge von Client-Anfragen gleichzeitig bearbeiten[4].

3.1 Kontext

Eine ereignisgesteuerte Applikation, welche mehrere Serviceanfrage empfängt und asynchron verarbeitet.

3.2 Problem

3.2.1 I/O Kategorien

Es gibt drei Arten von I/O [Libman and Gilbourd, 2005]: *Geblockte*, *nicht-geblockt synchrone* und *nicht-geblockt asynchrone*. *Geblockte I/O* übergibt die Kontrolle an den Aufrufer erst, wenn die Anfrage vollständig abgearbeitet ist. Eine Weiterführung oder Wiederverwendung des Threads des Aufrufenden und eine dementsprechende Nutzung vorhandener Resourcen ist nicht möglich. *Nicht-geblockt synchrone* I/O übergibt die Kontrolle sofort an den Aufrufer, wobei einer von zwei Fällen eintreten kann: *a*) ist die Operation bereits erfolgreich vollzogen, werden die Daten an den Aufrufer zurückgegeben; *b*) ist die Operation noch nicht abgeschlossen, wird ein Fehler gemeldet[5]. Bei *Nicht-geblockt asynchroner I/O* wird die Kontrolle – wie bei der synchronen Variante – sofort an den Aufrufer zurückgegeben, während intern Resourcen oder Threads zur Verfügung gestellt werden, um die Anfrage zu bearbeiten, über deren Vollendung der Aufrufende jedoch mittels eines vereinbarten Mechanismus benachrichtigt wird, z.B. mittels Callbacks. Von den drei vorgestellten I/O Arten ist die nicht-geblockt asynchrone die performanteste und skalierbarste [Libman and Gilbourd, 2005].
Die verschiedenen I/O Ansätze in Codebeispielen [Matusiak, 2007]:

Listing 1: Geblockte synchrone I/O

```
1    lines = read("info.txt");
2    // thread blocks until I/O call completes
3    // moved to I/O wait list, not runnable
4    // * wait until call completes *
5    // thread status changed back to runnable
6    print(lines);
```

Listing 2: Nicht-geblockte synchrone I/O

```
1    myhandle = from_filename("info.txt");
2    while (!lines) {
```

[4] Weitere Beispiele für Implementierungen sind: Completion Ports in Windows NT, POSIX AIO, ACE Framework

[5] z.B. `EWOULBLOCK/EAGAIN`

```
3           result = select(myhandle, timeout);
4           // thread switched to I/O bound
5           // * wait until call completes *
6           // thread switched back to cpu bound
7           if (result) {
8               lines = read(handle);
9               // syscall has confirmed handle is ready, I/O call
10              // is synchronous and non-blocking
11          }
12      }
13      print(lines);
```

Listing 3: Nicht-geblockt asynchrone I/O

```
1       int main {
2           read("info.txt");
3           // call is asynchronous, thread still cpu bound
4           // I/O operation executes in kernel thread,
5           // not application thread
6       }
7       void handle_read(string lines) {
8           print(lines);
9       }
```

3.2.2 Beobachtungen

Die Performance von Webservern wird durch Einsatz asynchroner Verarbeitung von Anfragen verbessert.

3.2.3 Kräfte

Vier Kräft lassen sich identifizieren [Pyarali et al., 1997]:

- **Skalierbarkeit (Concurrency)** Es werden viele Anfrage parallel and die Applikation gestellt. Diese muß in der Lage sein lange laufende Prozesse auszuführen, ohne daß die Performance von parallelel Prozessen beeinträchtigt wird.
- **Effizienz (Efficiency)** Unnötige kostenintensive Context-Wechsel, Synchronisationen (und CPU-Nutzung in allgemeinen) sollten in Hinblick auf die Datendurchsatzrate und Latenzzeit, vermieden werden.
- **Änderungsanforderungen (Adaptability)** Neue oder veränderte Servicemodule sollten in das bestehende System mit minimalem Aufwand integrierbar sein.
- **Information Hiding (Programming Simplicity)** Der Applikationscode soll keine Kenntnisse über die konkrete Implementierung des Multithreading und der Synchronisation voraussetzen.

3.2.4 Probleme mit konvetionellen Ansätzen zu Nebenläufigkeit

Intuitiv können Webserver parallele Anfragen mittels *synchroner Threads* verarbeiten, welche geradelinig in viele Sprachen mittels entrsprechender Threading-Bibliotheken zu implementieren sind. Folgende Nachteile ergeben sich aus diesem Modell [Pyarali et al., 1997]: *a*) Das Threading ist direkt an die Anzahl der Clients gekoppelt, obwohl eine resourcen-orientierte Aufteilung sinnvoller wäre; *b*) Probleme bei der Synchronisation, z.B. Caches und Log-Dateien; *c*)

Steigende Laufzeitkosten durch Threading (Context-Wechsel, Synchronisation, Datentransfer); *d*) Theading wird nicht auf allen Platformen unterstützt. Ein weiteres Muster für den Anwendungsfall stellt der *Reactor* dar [Pyarali et al., 1997], auf welchen hier nur verwiesen sei.

3.3 Lösung

3.3.1 Ablauf

1. Ein **Handler** initiiert eine asynchrone Lese-operation (falls das Betriebssystem dies unterstützt) und registriert Interesse in Fertigstellungsereignissen.
2. Der Ereignis-**Demultiplexer** wartet auf die Beendigung der Operation.
3. Das Betriebssystem verarbeitet die Anfrage in einem parallelen Thread, hinterlegt die Daten in einem Buffer und benachrichtigt den Demultiplexer
4. Der Demultiplexer ruft den entsprechenden Handler auf
5. Der Handler liest die Daten vom Buffer

3.3.2 Komponenten und Interarktion

Abbildung 2 zeigt das Komponenten-, Abbildung 3 das Interaktionsdiagramm.

3.3.3 Reactor vs. Proactor

Reactor und Proactor sind sich ähnlich[6]: Sie stellen einen einzelnen Ausführungsthread dar und lassen mehrere I/O Operationen parallel ablaufen. Unterschiede: Während der Reactor I/O Operationen im eigenen Thread ablaufen läßt, delegiert der Proactor diese an das Betriebssystem; der Proactor initiiert I/O Operationen und wartet auf Callbacks, der Reactor erwartet lediglich Events [Matusiak, 2007].

3.3.4 Web Server Benchmark

Ein Performancevergleich (Web Server, 5K, 50K, 500K und 5M Datei Download) von *Thread/Request*, *Thread Pool* und *Proactor Pattern* zeigt eine eindeutigen Vorteil für das Proactor Pattern sowohl beim Durchsatz als auch bei der Latenzzeit für Dateien ab 5M. Für Dateigrößen von 5K zeigt das *Thread Pool* Modell, für 50K das Proactor Pattern bei Nebenläugigkeit von mindestens 5 Anfragen beste Ergebnisse [Matusiak, 2007].

[6] TProactor (Terabit Solutions) is a proactor emulated on top of a reactor. (no performance degradation) [Matusiak, 2007]

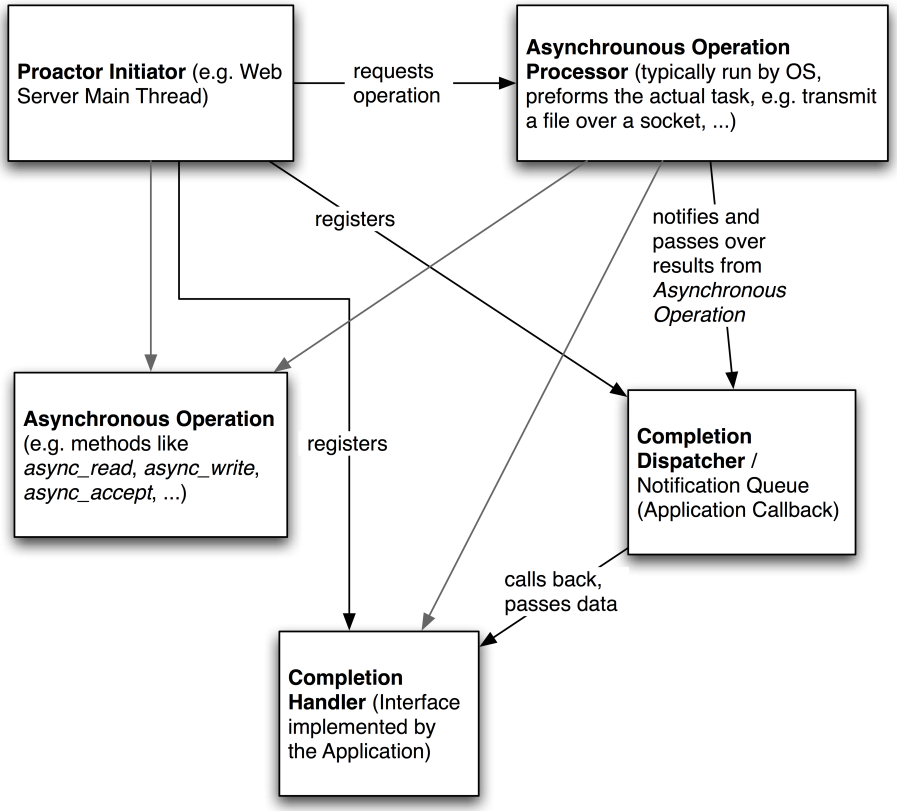


Abbildung 2: Proactor, Komponenten

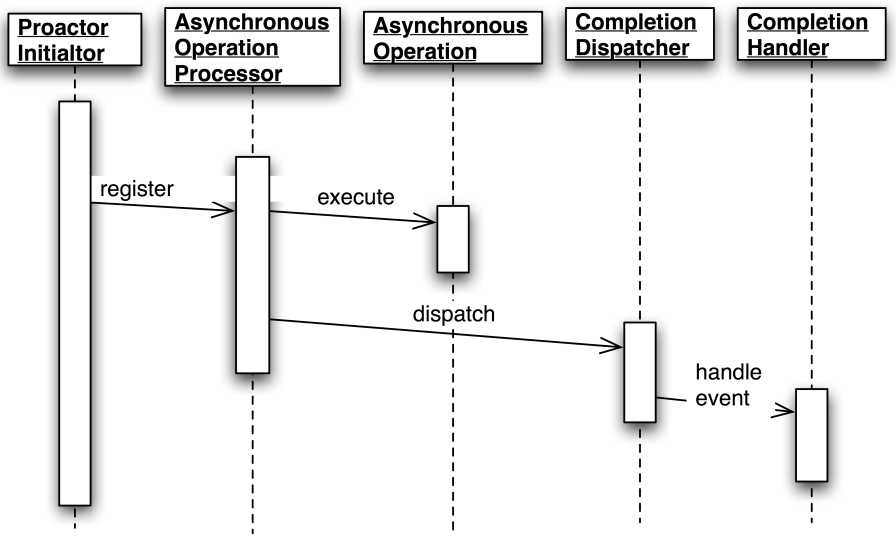


Abbildung 3: Proactor, Interaktion

4 Asychronous Completion Token

Das Asychronous Completion Token (ACT) [Schmidt, 1996b] Muster bietet eine effiziente Methode um von einem Client initiierte asynchrone Operation im Client selbst zu verarbeiten. Das Muster wird auch als *Active Multiplexing* bezeichnet.

4.1 Szenario

Ein verteiltes System von Komponenten, die von ein oder mehrere Verwaltungsmodule besitzt, welche verschiedene Statusnachrichten von einer Menge von Komponenten erhält. Ein Verwaltungsmodul (Management-Applikation) meldet bei *verschiedenen Komponenten* Interesse an *verschiedenen Ereignissen* an. Diese werden von den Komponenten asynchron beantwortet. Die Management-Applikation muß nach Art des Antwort die Eingabe weiterverarbeiten (z.B. nach Art der Statusmeldung, nach Quelle, ...).
Ein Verarbeitungsschema in der Management-Applikation, in der für jede Registrierung ein Thread erstellt wird, hat den Nachteil mangelnder Skalierbarkeit und Performanz [Schmidt, 1996b].

4.2 Kontext

Eine verteilte Umgebung in der Applikationen asynchron Services aufrufen und Antworten sukzessiv verarbeiten.

4.3 Problem

Das Problem, bereits im Abschnitt *Szenario* illustriert, ist das zuordnen von n Service-Antworten zu einer Menge M von Aktionen innerhalb einer Applikation.

4.3.1 Kräfte

Zwei Kräfte lassen sich identifizieren [Schmidt, 1996b]:

- **Effizienz** Die Verarbeitungs nach dem Eintreffen der Antwort soll minimal sein. Die Anzahl der zwischen den Komponenten übermittelten Nachrichten sollen minimal sein.
- **Separation of Concerns**[7] Der Server übernimmt die korrekte Emission von Event, der Client die korrekte Auswertung.

4.4 Lösung

Generiere für applikationsspezifischen Zustand und erforderliche Antwort ein *eindeutiges Token* und assoziiere dieses mit den mit anderen Komponenten ausgetauschten Nachrichten.

[7] oder: *Divide and Conquer*, *Teile und Herrsche*, *Divide et Impera*

4.4.1 Struktur

Beteiligte Designelemente (Abb. 4) Client, Server, Token und deren Beziehungen (Abb. 5)

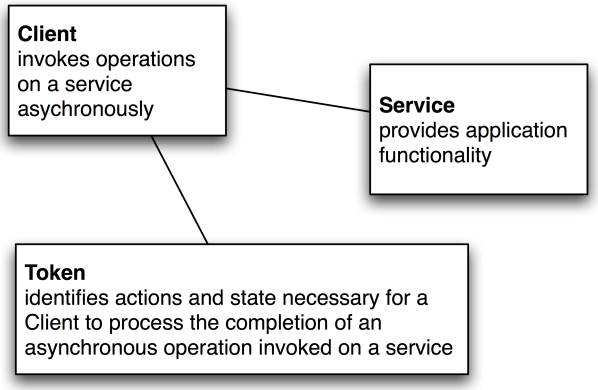


Abbildung 4: Asychronous Completion Token, Struktur

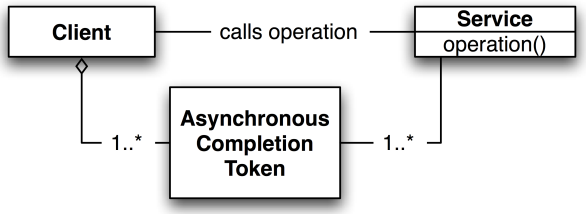


Abbildung 5: Asychronous Completion Token, Beziehung der Komponenten

4.4.2 Dynamische Aspekte

Verhalten der Komponenten im Muster:

- Vor jeder asynchronen Operation wird ein ACT erstellt, zugeschnittenn auf die entsprechende Operation.
- Das Token wird in der Anfrage an den Service mitübermittelt.
- Aufgrund der Asynchronität der Operation wird der Client nicht geblockt und kann weitere Aufgaben ausführen.
- Bei Empfang der Nachricht vom Service wird das Token zurückgesendet und ausgewertet (bzw. stellt erforderliche Zustände wieder her, damit die Antwort vom Client ausgewertet werden kann).

Das Token muß für den Client auswertbar, für den Service opak sein [Schmidt, 1996b].

4.5 Vorteile und Nachteile

Schmidt [Schmidt, 1996b] listet folgende Vorteile des ACT Musters auf:

- **Einfacher Client Datenstrukturen** Da der Client beliebige Informationen im Token hinterlegen kann, benötigt der Client für den Dispatch keine komplexen Datenstrukturen.
- **Effiziente Zustandsaquirierung** und **Speichereffizienz** Das Token selbst kann beispielsweise Pointer auf bereits vorhandene Datenstrukturen setzen.
- **Flexibilität** Eine Pflicht, bestimmte Interfaces zu unterstützen um einen Service zu nutzen, besteht nicht.
- **Nebenläufigkeitunabhängigkeit** Das Nebenläufigkeitsmodell des Client spielt im ACT Muster keine Rolle (Client kann Single-Threaded oder Multi-Threaded sein).

Ebenda [Schmidt, 1996b] finden sich auch folgende Probleme:

- **Speichereffizienz** ACTs mit Zeigern auf Client-Datenstrukturen werden unter Umständen nicht deallokiert.
- **Remapping** Direkte Zeiger in persistenten Clients verlieren nach einem Neustart ihre Gültigkeit, und auch nach Speicher Remappings[8]. Eine weitere Stufe der Indirektion (indirection) umgeht dieses Problem[9].

4.6 Diskussion zur Implementierung

Schmidt [Schmidt, 1996b] listet als Schritte der Implementierung auf: *a*) Design der ACT Repräsentation (diese kann zeigerbasiert[10], Objektreferenz-basiert[11] oder indexbasiert[12] implementiert werden), *b*) Art und Weise der Übermittlung an den Service[13], *c*) Speicherung des ACT im Service und *d*) Feststellung des Multiplex-Mechanismus im Client[14].

4.7 Einsatz des Patterns

Als konkrete Implementierung werden aufgezählt [Schmidt, 1996b]: Windows NT Completion Ports [Russinovich, 2006], POSIX AIO [Jones, 2006], CORBA [Schmidt, 2006] Demultiplexing, EMIS (Electronic Medical Imaging System, *Siemens*) Netzwerk Management und *FedEx* Inventar Tracking.

[8] Z.B. unter Linux *mremap(2)*

[9] "Any problem in computer science can be solved with another level of indirection", Butler W. Lampson

[10] empfohlen auf homogenen Plattformen [Schmidt, 1996b]

[11] diskutiert im Zusammenhang mit CORBA-Schnittstelle

[12] Indizes führen eine zusätzliche Schicht der Indirektion ein, welche u.a. Persistenz der ACT unterstützt

[13] hier wird zwischen impliziter expliziter Übergabe unterschieden; bei der expliziten Übermittlung enthält die Servicesignatur einen eigenen ACT Parameter, bei der impliziten Übermittlung wird das ACT transparent in einer allgemeinen Kontextbeschreibung eingebettet

[14] Der Autor [Schmidt, 1996b] beschreibt *Callbacks*, synchron via *Event Loops* oder *Reactor* bzw. asynchron über Signal-Handler (POSIX)

5 Acceptor-Connector

Das Acceptor-Connector Muster ermöglicht eine konzeptuelle Trennung von *Initialisierung* und *Nutzung* eines Service in einer verteilten Umgebung [Schmidt, 1996a].

5.1 Szenario

Als Beispiel für ein Acceptor-Connector Szenario beschreibt Schmidt [Schmidt, 1996a] eine typisches Netzwerk mit Clients (Rechner), externen Systemen (Satelitten) und einem Gateway dazwischen. Clients können *aktiv* Verbindungen mit *verschiedenen Protokollen* aufbauen, sie können *passiv* auf Nachrichten warten. Die *Rollen* von Client und Server/Gateway sind dementsprechend austauschbar.

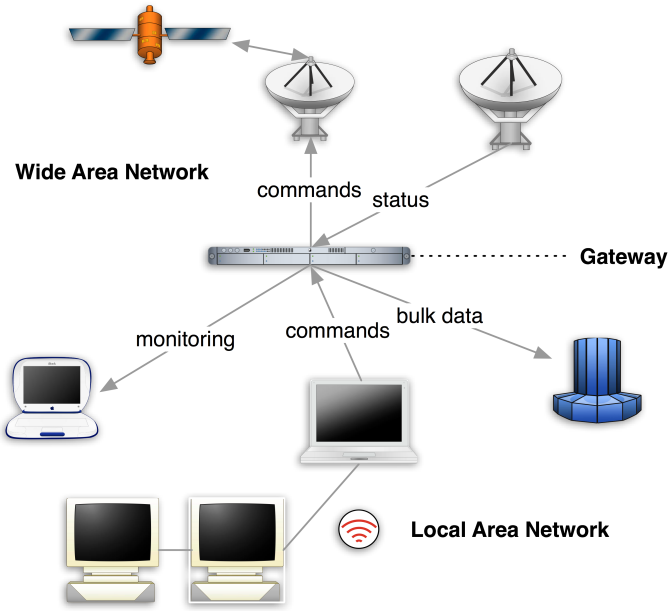


Abbildung 6: Acceptor-Connector, Szenario

5.2 Kontext

Eine Client/Server Umgebung, in der verbindungsorientierte Protokolle zur Kommunikation eingesetzt werden [Schmidt, 1996a].

5.3 Problem

Bei dem *Verbindungsaufbau* spielt die Verbindungsrolle (*Connection Role*), bei der Verbindungsnutzung die Kommunikationsrolle (*Communication Role*) die größere Rolle. Verbindungsaufbau ist asymmetrisch (eine Komponente wartet, die andere initiiert); die Rollen bei der *Verbindungsnutzung* können jedoch

anders liegen. Die Datenübertragung hängt wesentlich vom Übertragungsprotokoll ab[15]. Die Protokolle zum Verbindungsaufbau (Connection Management Protocols) können unterschiedlich sein, bzw. unterschiedliche Technologien oder Interfaces zur Verfügung stellen, während die Nachrichtenübertragung von diesen Einzelheiten abstrahiert[16].
Protokolle des Verbindungsaufbaus werden seltener geändert als Applikationsprotokolle. Daher sollten diese Bereiche (durch das Muster) getrennt behandelt werden Schmidt [1996a].

5.3.1 Kräfte

- **Änderbarkeit/Erweiterbarkeit** Neue Initialisierungs- und Übertragungsprotokolle sollten leicht hinzufügbar sein; vorhandere Komponenten sollen weiter verwendet werden können
- **Separation of Concerns** Das Muster sollte klar *Connection Roles* (aktiv, passiv) und *Communication Roles* (Client, Server) unterschieden.
- **Plattform-Unabhängigkeit** Die Notwendigkeit der Unterstützung heterogener Netze ist realistisch und sollte möglich und wenig aufwendig sein.
- **Information Hiding** Verbindungsendpunkte sollten transparent Einhaltung von Bedingungen prüfen[17]. Dazu ist das Aufteilen der Verantwortlichkeiten je nach Verbindungsphase (Aufbau, Übertragung) nötig.
- **Skalierbarkeit** Vorhandene Systemresourcen (asynchroner Verbindungsaufbau) sollte unterstützt werden, um maximale Skalierbarkeit und minimale Latenzzeiten zu ermöglichen.

5.4 Lösung

Für jeden angebotenen Service implementiere Komponenten nach dem Acceptor-Connector Muster. Implementiere zwei Fabriken, Acceptor und Connector. Der *Acceptor* erstellt einen Transportendpunkt, der *passiv* auf einem spezifischen Adresse auf Verbindungen *wartet*. Der *Connector initiiert aktiv* eine Verbindung zu einem anderen Transportendpunkt. *Beide* initialisieren einen Service Handler, die die weiteren Aktionen im Rahmen der Datenübertragung ausführen Schmidt [1996a].

5.4.1 Struktur

Das Muster separiert drei Komponenten: Akzeptoren (*Acceptors*), Konnektoren (*Connectors*) und *Service Handler*[18]. Abb. 7 zeigt das Struktur, Abb. 8 die Kollaboration im Acceptor in der Übersicht.

[15] Typische Klassen von Übertragungsprotokollen sind Peer-to-Peer, Request-Response und Streaming.
[16] Z.B. in Form uniformer `send/receive` Schnittstellen.
[17] Z.B. Verhinderung des Schreibens auf einen `READONLY`-Socket.
[18] Im folgenden werden der Kohärenz halber nur die englischen Bezeichnungen der Komponenten verwendet.

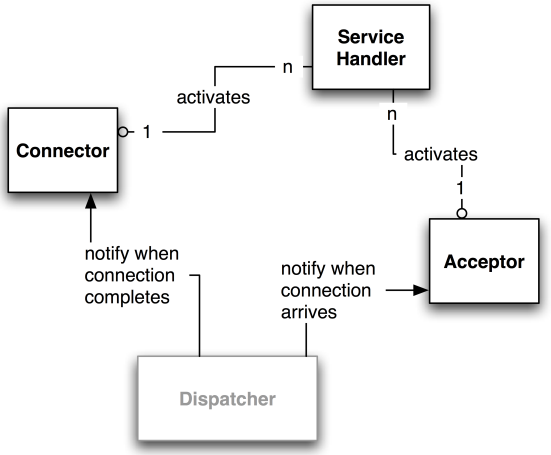


Abbildung 7: Acceptor-Connector, Struktur

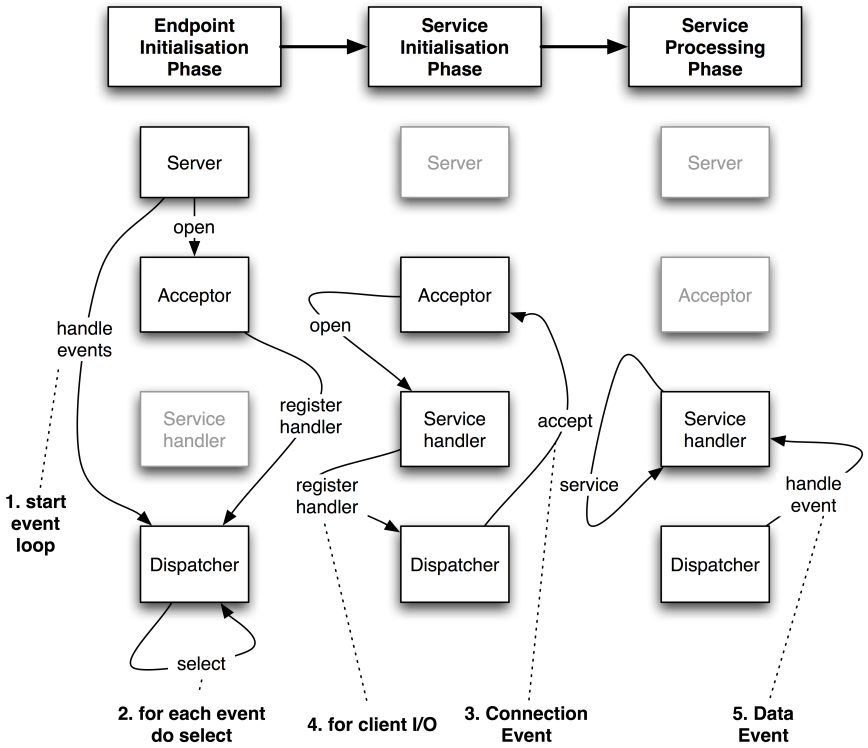


Abbildung 8: Acceptor-Connector, Kollaboration im Acceptor

5.4.2 Ablauf im Acceptor

- **Endpunktinitialisierung** – Eine Applikation ruft die `open` Method des `Acceptors` auf, woraufhin eine passiver Transportendpunkt (mit Adresse und Port) initiiert wird; das Acceptor-Objekt registriert sich im `Dispatcher`, so daß dieser via `Callback` bei eingehenden Verbindungen benachrichtigt werden kann; der `Dispatcher` eröffnet einen *Event-Loop*.
- **Serviceinitialisierung** – Bei eintreffenden Verbindungen ruft der `Dispatcher` die `accept` Methode des `Acceptor` auf; diese Methode stellt die für den `Service Handler` nötigen Resourcen zusammen, liest aus dem bereits initiierten Transportendpunkt und verbindet diese mit dem Service; der `Service Handler` wird ebenfalls bei dem `Dispatcher` registriert
- **Serviceabwicklung** – Die Verbindung wurde aufgebaut, der `Service Handler` registriert; Schicht 7 [Zimmermann, 1980] Protokolle wie HTTP, FTP, SSH können zwischen den Peers ausgeführt werden; ist die Übertragung beendet, kann die Verbindung geschlossen, die Resourcen des `Service Handlers` freigegeben werden.

5.4.3 Ablauf im Connector

Der Prozess im Connector kann *synchron* oder *asynchron* ablaufen, wobei sich der *synchrone* Ansatz für Fälle eignet, in denen die Netzwerklatenzzeit gering ist, pro Service Handler ein Thread zu Verfügung steht oder der Client ohne einen geordneten Aufbau von Services keine nützlichen Aufgaben ausführen kann, während der *asynchron* Ansatz eignet, falls eine hohe Anzahl von Verbindungen mit hohen Latenzzeiten (z.B. im WAN) zu verarbeiten sind, die Plattform kein Multithreading unterstützt oder der Client weitere Aufgaben neben dem Verbindungsaufbau auszuführen hat.
Die Kollaboration im *synchronen* Fall:

- **Verbindungsaufbau** – Die Applikation ruft die `connect` Methode des `Connectors` auf; der Kontrollthread wird bis zum vollständigen Verbindungsaufbau geblockt.
- **Serviceinitialisierung** – Aus der `connect` Methode wird wird die `complete` Methode des `Service Handlers` aufgerufen, welche servicespezifische Initialisierung vornimmt.
- **Serviceabwicklung** – Ähnlich wie im Acceptor-Fall. Die `Service Handler` führen das jeweilige Protokoll auf Applikationsschicht aus.

Die Kollaboration im *asynchronen* Fall:

- **Verbindungsaufbau** – Wie im synchronen Fall etabliert der `Connector` aktiv eine Verbindung zum Peer; der Kontrollthread wird nicht geblockt.
- **Serviceinitialisierung** – Bei Beendingung des Aufbaus der Verbindung ruft der `Dispatcher` in Form eines Callbacks die `complete` Methode des `Connector` auf, welche servicespezifische Initialisierung vornimmt.
- **Serviceabwicklung** – Ähnlich dem synchronen Fall. Die `Service Handler` führen das jeweilige Protokoll auf Applikationsschicht aus.

5.5 Vorteile und Nachteile

Als Vorteile zählt Schmidt Schmidt [1996a] die Wiederverwendbarkeit, Portabilität und Erweiterbarkeit der Service Module auf, welche durch die Trennung von Verbindungsaufbau und Servicehandling ermöglicht wird; weiterhin Robustheit und effiziente Nutzung der Netzwerken inheränten Parallelität.
Als Nachteile werden die zusätzliche Indirektion und Komplexität genannt, die sie durch einen nicht-direkten Zugriff auf Netzwerkschnittstellen ergeben.

5.6 Einsatz des Patterns

Als Einsatz des Patterns nennt Schmidt Schmidt [1996a] den unix network superserver *inetd*, CORBA ORB, Webbrowser (speziell HTML-Renderer, die asynchron im Seitentext eingebettete Resourcen herunterladen), Ericson EOS Call Center Management und das ACE Framework[19].

[19] http://www.cs.wustl.edu/~schmidt/ACE.html

Teil II
Implementierung

6 Proactor

Das Proactor Muster wurde mit Hilfe von Threads implementiert, also ohne wahre Asynchronizität, da es um die Veranschaulichung der Klassen ging und Java NIO nur einen Low-Level Zugriff auf asynchrone Sockets erlaubt[20].
Die Beispielapplikation ist ein simpler Server, der als Requests Dateinamen annimmt und deren Inhalt vom Filesystem liest und zurückgibt.

7 Beispiel

Zunächst muß der Server gestartet werden.

Listing 4: ProactorServer

```
$ java -jar ProactorServer.jar
Aug 4, 2009 6:17:29 PM proactor.server.Server main
INFO: [main] Listening on port 43210 ...
```

Danach können Anfragen an den Server gestellt werden.

Listing 5: ProactorClient

```
$ java -jar ProactorClient.jar /etc/passwd
##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode.  At other times this information is provided by
# Open Directory.
#
# This file will not be consulted for authentication unless the BSD local node
# is enabled via /Applications/Utilities/Directory Utility.app
#
# See the DirectoryService(8) man page for additional information about
# Open Directory.
##
...
```

Die Abläufe im Proactor werden in der Konsole geloggt:

Listing 6: Logging

```
..
INFO: [main] Connected to /127.0.0.1:62929
Aug 4, 2009 6:19:11 PM proactor.server.Server loop
INFO: [main] Waiting ...
Aug 4, 2009 6:19:11 PM proactor.server.Server$RequestProcessor run
INFO: Processing request: /etc/passwd
```

[20] Für eine echte Java Implementierung von Proactor sei hier auf den TProactor http://www.terabit.com.au/solutions.php verwiesen.

```
Aug 4, 2009 6:19:11 PM proactor.server.MockAsynchronousOperationProcessor register
INFO: [Thread-4] New operation requested. Will mock asynchronous operation with via thread.
Aug 4, 2009 6:19:11 PM proactor.server.MockAsynchronousOperationProcessor register
INFO: [Thread-4] Thread started: Thread-5
Aug 4, 2009 6:19:11 PM proactor.server.Server$RequestProcessor run
INFO: New (mock) async operation registered and initiated. Ready to accept new connections.
Aug 4, 2009 6:19:11 PM proactor.server.MockAsynchronousOperationProcessor$Worker run
INFO: [Thread-5] Executing mock async operation...
Aug 4, 2009 6:19:11 PM proactor.server.FileReadOperation execute
INFO: [Thread-5] Starting mocked async operation. Attempting to read file: /etc/passwd
Aug 4, 2009 6:19:11 PM proactor.server.FileReadOperation execute
INFO: [Thread-5] Result ready. Length: 2888
Aug 4, 2009 6:19:11 PM proactor.server.SimpleCompletionDispatcher dispatch
INFO: [Thread-5] Dispatching result
Aug 4, 2009 6:19:11 PM proactor.server.SimpleCompletionHandler setData
INFO: [Thread-5] Received data of length: 2888
Aug 4, 2009 6:19:11 PM proactor.server.SimpleCompletionHandler handleEvent
INFO: [Thread-5] Sending data.
Aug 4, 2009 6:19:11 PM proactor.server.SimpleCompletionHandler handleEvent
INFO: Thread-5
Aug 4, 2009 6:19:11 PM proactor.server.SimpleCompletionHandler handleEvent
INFO: [Thread-5] Sending EOM Token: Goodbye.
...
```

7.1 Interfaces

1. `AsynchronousOperation`, kapselt eine (mock-)asynchrone Operation, stellt die Methoden `execute` und `getResults` zur Verfügung.
2. `AsynchronousOperationProcessor`, kapselt das Handling für asynchrone Operationen, implementiert eine Methode `register(AsynchronousOperation, CompletionHandler, CompletionDispatcher)`
3. `CompletionDispatcher`, verteilt Objekte an `CompletionHandler`, implementiert `dispatch(Object, CompletionHandler)`
4. `CompletionHandler`, kapselt das Handling am Ende der Operation, Methoden: `setData(Object)`, `handleEvent`
5. `ProactorInitiator`, Interface für Terminologie

7.2 Request Cycle

Die Klasse `Server` wartet auf synchon-blockend auf Verbindungen und lagert die Verarbeitung in den `RequestProcessor`-Thread aus:

Listing 7: Server

```
while ((socket = ss.accept()) != null) {
        ...
        Thread t = new Thread(new RequestProcessor(socket));
        ...
```

Der `RequestProcessor` initiiert die Proactor Komponenten. In diesem Fall wird als Request ein String verarbeitet, der den Inhalt der Datei die diesem Namen vom Filesystem liest (falls sie existiert) und zurückgibt. Die `AsynchronousOperation` in diesem Fall ist eine `FileReadOperation`.

Listing 8: RequestProcessor

```
1
2           AsynchronousOperation operation = new FileReadOperation(requestedFile);
3           CompletionHandler handler = new SimpleCompletionHandler(socket);
4           processor.register(operation, handler, dispatcher);
```

Die `register` Methode des `AsynchronousOperationProcessor` initiert die Operation. Diese führt die Operation aus und ruft anschließend den Dispatcher auf:

Listing 9: AsynchronousOperationProcessor

```
1           operation.execute();
2           dispatcher.dispatch(operation.getResult(), handler);
```

Der `Dispatcher` übergibt das Ergebnisobject dem Handler und ruft die Methode `handleEvent` des CompletionHandlers auf. Der `CompletionHandler` übernimmt in diesem Fall (`SimpleCompletionHandler`) lediglich die Ausgabe des Ergebnisobjektes auf den `Socket` über einen `PrintStream`.

7.3 Nachtrag zu NIO

Laut einer Benchmark-Studie ist Java NIO unter Linux mit NPTL (Native POSIX Thread Library) bis zu 30% langsamer als threadbasierte Ansätze. Einen Vorteil von NIO stellt das Debugging dar. Wenige Threads sind einfacher zu debuggen als Tausende Threads[21].

8 Asynchronous Completion Token

Die Implementierung von ACT ist bewußt einfach gehalten. Das Beispiel implementiert einen einfachen Server und Client die über ObjectStreams (`ObjectOutputStream`, `ObjectInputStream`) kommunizieren.

8.1 Ablauf

Der Client erstellt ein `Packet`, welches einen Integer enthält. Außerdem wird ein Token vom Type String an jedes Packet angehangen.

Listing 10: setToken

```
1           Packet p = new Packet().setNumber(number).setToken(action);
```

Das Token bezeichnet die Operation, die bei Eintreffen des Packets ausgeführt werden soll. In diesem Fall ist es eine Zeichenkette, die direkt im Code geprüft wird (`square` quadriert, `cube` kubiert die im Packet gefundene Zahl).

[21] In tests, NPTL succeeded in starting 100,000 threads on a IA-32 in two seconds.

Listing 11: getToken

```
1
2           if (rp.getToken().equals("square")) {
3                   int result = square(rp.getNumber());
4           }
5
6           if (rp.getToken().equals("cube")) {
7                   int result = cube(rp.getNumber());
8           }
9
10          logger.info(String.valueOf(result));
```

Der Server empfängt lediglich ein Objekt (`Packet`) und sendet es zurück.

9 Acceptor-Connector

9.1 Bemerkung

Acceptor-Connector ist ein komplexes Pattern mit einem speziellen Use-case, realisiert z.B. im UNIX superserver `inetd`. Das Hauptgewicht liegt in der Trennung von Connection und Communication Role. Zur Komplexität von Acceptor-Connector kommt die Java NIO eigene Kompexität, resultierend hauptsächlich in den elementaren Datenstrukturen, z.B. ByteBuffer, auf denen höhere Protokolle (Satellitensteuerung [Schmidt, 1996a]) nur mit Zwischenschichten zu realisieren, deren Implementierung nicht direkt zum Verständnis des Musters beiträgt.
Statt einer eigenen Implementierung möchte ich einen Superserver auf Basis des Apache MINA[22] Frameworks vorstellen.

9.2 Apache MINA

Apache MINA ist ein Java NIO basiertes Framework für Netzwerkapplikationen. Es separiert die Transportschicht von der Applikationsschicht und ermöglicht die Implementierung von Client- und Serverapplikationen.

9.3 Simple Superserver

Das Beispiel implementiert einen einfachen Superserver, der verschiedene Protokolle auf verschiedenen Ports startet. Als Protokolle wurden implementiert:

- **Echo**, Echo Server, an RFC 862 [23] angelehnt.
- **Reverse**, Sendet empfangenen Text ungekehrt zurück
- **PortCheck**, Sendet Port-Status eines Hosts zurück, konfigurierbar

Der Superserver registert eine Reihe von Services[24]:

[22] http://mina.apache.org/
[23] http://www.faqs.org/rfcs/rfc862.html
[24] Gleiche Services auf mehreren Ports sind möglich.

Listing 12: SuperServer

```
1         static ArrayList<String> services = new ArrayList();
2         static {
3                 services.add("ac.echo.EchoService:10000");
4                 // services.add("ac.echo.EchoService:20000");
5                 // services.add("ac.echo.EchoService:30000");
6                 services.add("ac.port.PortCheckService:10001");
7                 services.add("ac.reverse.ReverseService:10002");
8         }
```

Diese werden der Reihe nach gestartet:

Listing 13: Logging

```
1         Aug 4, 2009 5:45:08 PM ac.echo.EchoService create
2         INFO: ac.echo.EchoService started on port 10000
3         Aug 4, 2009 5:45:08 PM ac.ping.PingService create
4         INFO: ac.ping.PingService started on port 10001
5         Aug 4, 2009 5:45:08 PM ac.reverse.ReverseService create
6         INFO: ac.reverse.ReverseService started on port 10002
```

Am Beispiel des Echoservers sollen die Komponenten eines Services verdeutlicht werden. Zunächst wird ein Acceptor erstellt

Listing 14: EchoService

```
1         SocketAcceptor sa = new NioSocketAcceptor();
2         sa.setHandler(new EchoProtocolHandler());
3         sa.bind(new InetSocketAddress(port));
```

Der Acceptor/Connector kapselt den Transport und kann durch andere Implementierungen[25] transparent für die darüberliegenden Schichten ersetzt werden. Der Handler übernimmt die applikationsspezifischen Aufgaben und implementiert das `IoHandler` Interface bzw. erweitert den `IoHandlerAdapter`. Das Interface definiert Callbacks für den Nachrichtenaustausch (`messageSent`, `messageReceived`), für das Session Handling (`sessionClosed`, `sessionCreated`, `sessionIdle`, `sessionOpened`) und Ausnahmebearbeitung (`exceptionCaught`). Die Session stellt dabei die Verbindung zwischen zwei Peers dar.
Beispiel EchoProtocolHandler:

Listing 15: EchoProtocolHandler

```
1     public void messageReceived(IoSession session, Object message) throws Exception {
2                 ...
3                 // End sessions from within session via 'exit'
4                 if (message.toString().equals("exit")) {
5                         // Say goodbye before closing the session ...
6                         WriteFuture future = session.write("Exiting...");
7                         future.addListener(new ExitListener());
8                 } else {
9                         // Write the received data back to remote peer
10                        session.write(message);
11                }
12        }
```

[25] z.B. `AprSocketAcceptor`, `SerialConnector`, ...

9.4 Beispielsessions

9.4.1 Server

Listing 16: Server

```
$ java -jar AcceptorConnector.jar
Aug 4, 2009 7:26:04 PM ac.echo.EchoService create
INFO: ac.echo.EchoService started on port 10000
Aug 4, 2009 7:26:04 PM ac.port.PortCheckService create
INFO: ac.port.PortCheckService started on port 10001
Aug 4, 2009 7:26:04 PM ac.reverse.ReverseService create
INFO: ac.reverse.ReverseService started on port 10002
```

9.4.2 Echo

Listing 17: Echo

```
$ telnet localhost 10000
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying fe80::1...
telnet: connect to address fe80::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello
hello
exit
----- Bye -----
Connection closed by foreign host.
```

9.4.3 Port Check

Listing 18: Port Check

```
$ telnet localhost 10001
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying fe80::1...
telnet: connect to address fe80::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
google.com
gw-in-f100.google.com:22 connect timed out
gw-in-f100.google.com:25 connect timed out
gw-in-f100.google.com:80 OPEN
Done.
pcai042.informatik.uni-leipzig.de
pcai042.informatik.uni-leipzig.de:22 OPEN
pcai042.informatik.uni-leipzig.de:25 Connection refused
pcai042.informatik.uni-leipzig.de:80 OPEN
Done.
...
```

9.4.4 Reverse

Listing 19: Reverse

```
$ telnet localhost 10002
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying fe80::1...
telnet: connect to address fe80::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello dlrow
world olleh
```

Literatur

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns*. John Wiley and Sons, New York, 1996. 3

J. W. Cooper. The design patterns java companion, 1998. URL http://www.patterndepot.com/put/8/DesignJava.PDF. 3

S. Ferg. Event-driven programming: Introduction, tutorial, history, 2006. URL http://eventdrivenpgm.sourceforge.net/event_driven_programming.pdf. 4

E. Gamma, T. Helm, R. Johnson, and J. Vlissides. Design patterns. abstraction and reuse of object oriented design. In *Proceedings of ECOOP '93*, pages 405–431, 1993. 3

E. Gamma, T. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Software.* Addison-Wesley, Reading, MA, 1995. 3

M. Grand. Overview of design pattern, 05 2009. URL http://www.mindspring.com/~mgrand/pattern_synopses.htm. 3

M. T. Jones. Boost application performance using asynchronous i/o. Technical report, IBM, 2006. URL http://www.ibm.com/developerworks/linux/library/l-async/index.html. 11

A. Libman and V. Gilbourd. Comparing two high-performance i/o design patterns, 11 2005. URL http://www.artima.com/articles/io_design_patterns.html. 5

M. Matusiak. Investigating the proactor design pattern, 2007. URL http://www.cs.uu.nl/docs/vakken/no/proactor_slides.pdf. 5, 7

Multiple. Exception patterns, 2009. URL http://c2.com/cgi/wiki?ExceptionPatterns. 3

W. Pree. *Design Patterns for Object Oriented Software Development*. Addison-Wesley, 1994. 3

W. Pree and H. Sikora. Design patterns for object oriented software development. Technical report, Johannes Kepler Universität Linz, RACON Software, Inc., 1997. URL http://selab.cu.ac.kr/link/patterns/PS97.pdf. 3

I. Pyarali, T. Harrison, D. C. Schmidt, and T. D. Jordan. Proactor, an object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events. Technical report, Pattern Languages of Programming Conference, 1997. URL http://www.cs.wustl.edu/~schmidt/PDF/proactor.pdf. 4, 6, 7

M. Russinovich. Inside i/o completion ports. Technical report, Microsoft Technet, 2006. URL http://technet.microsoft.com/en-us/sysinternals/bb963891.aspx. 11

D. C. Schmidt. Acceptor-connector – an object creational pattern for connecting and initializing communication services. Technical report, European Pattern Language of Programs conference, 1996a. URL http://www.cs.wustl.edu/~schmidt/PDF/Acc-Con.pdf. 4, 12, 13, 16, 20

D. C. Schmidt. Asynchronous completion token – an object behavioral pattern for efficient asynchronous event handling. Technical report, 3rd annual Pattern Languages of Programming conference, 1996b. URL http://www.cs.wustl.edu/~schmidt/PDF/ACT.pdf. 4, 9, 10, 11

D. C. Schmidt. Overview of corba, 2006. URL http://www.cs.wustl.edu/~schmidt/corba-overview.html. 11

E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, 1979. URL http://www.win.tue.nl/~wstomv/quotes/structured-design.html. 4

H. Zimmermann. Osi reference model - the iso model of architecture for open systems interconnection. Technical report, IEEE Transactions of Communication, 1980. URL http://www.comsoc.org/livepubs/50_journals/pdf/RightsManagement_eid=136833.pdf. 15