

5 Dienst-Architektur

5.1 Dualität von Prozessen und Monitoren	4
5.2 Aktive Objekte	19
5.3 Drei Sprachen	34
Zusammenfassung	41

Zur Erinnerung (2.5):

Asymmetrische Beziehung zwischen Anbieter (server) und seinen Klienten (clients) :

- Klienten kennen Server und erwarten von ihm die Erbringung bestimmter Dienste (services). Klient schickt Auftrag (Anforderung, request), erwartet Ergebnis (Antwort, reply).
- Server kennen ihre Klienten nicht, außer temporär bis der jeweilige Dienst erbracht ist.
- Die Lebensdauer des Servers ist unabhängig von der Lebensdauer seiner Klienten.

Kommunikationsoperationen (Pseudocode):

Klient: Port p = **sendarg**(arg, server);
 // do something else
 recvres(resvar, p);

Server: repeat
 Port p = **recvarg**(argvar);
 // do service
 sendres(res, p);
 end

p ist temporärer Eingabe-Port des Klienten für die Antwort,
verwandt mit einem **Future**-Objekt
(siehe Aufgabe 2.3 und ALP 4 (Lecture 12))

5.1 Dualität von Prozessen und Monitoren

Klientenverhalten bei asynchronem Aufruf:

```
proc    = fork server.service(arg);
...
resvar = join proc;           Objekt
```

Klientenverhalten bei verteilter Beauftragung:

```
port   = sendarg(server,service,arg);
...
recres(resvar, port);      Prozess
```

Bei gleicher Abbildung Arg → Res gleiche Semantik??

```
process Server {
    .... // procedures service1 etc.
repeat Port p = recvarg(service, arg);
case service of
    "service1": sendres(service1(arg), p) |
    "service2": sendres(service2(arg), p)
end;
end
}
```

ist semantisch äquivalent zu

```
monitor Server {
public service1(arg) { .... }
public service2(arg) { .... }
...
}
```

mit *asynchronen Aufrufen*. (Typen werden hier ignoriert.)

Welcher Prozess ist semantisch äquivalent zu

```
class Server {  
    service1(arg) { ..... }  
    service2(arg) { ..... }  
    ...  
}
```

mit *asynchronen* (und nebenwirkungsfreien) Aufrufen ?

→ *Nichtsequentieller* Prozess mit unbeschränkt vielen „Threads“. Jeder Auftrag wird von einem eigenen Thread ausgeführt.

Frage: gibt es Äquivalenzen auch zwischen
komplizierteren Objekten und Prozessen?

Objekt: Monitor mit Bedingungssynchronisation,
R/W-Monitor, schwacher Monitor
(aber stets „serialisierbar“!)

Prozess: bedingte Auftragsannahme,
unterbrochene Auftragsausführung,
überlappende Auftragsausführungen

Objekt	Prozess
Monitor	→ sequentieller Prozess
Zustand: inaktiv verzögert, mit Wachen mit <i>signal and continue</i>	→ Zustand: Blockade in disjunktivem <i>recvarg</i> → <i>recvarg</i> mit Wachen
mit <i>signal and wait</i>	→ Buchführung: „unterbrochene Bearbeitung kann demnächst fortgesetzt werden“ → Bearbeitung unterbrechen, Buchführung dazu; evtl. unterbrochene Bearbeitung fortsetzen gemäß Buchführung
mit <i>signal and return</i>	→ Bearbeitung abschließen mit <i>sendres</i> ; evtl. unterbrochene Bearbeitung fortsetzen gemäß Buchführung
<i>return</i>	→ (ebenso)

Objekt

schwacher Monitor
Zustand: inaktiv
Methodenbeginn
internes Sperren

return

Prozess

- nichtsequentieller Prozess
- Zustand: Blockade in disjunktivem *recvarg*
- Bearbeitung startet, evtl. eigener Thread
- Gleichartiges Sperren und/oder
Nachbildung durch Serialisierung
- Bearbeitung abschließen mit *sendres*;
evtl. unterbrochene Bearbeitung
fortsetzen gemäß Buchführung

Prinzip Dualität [Lauer/Needham 1978]

Zu jedem asynchron aufrufbaren Monitor gibt es einen semantisch äquivalenten Server-Prozess und umgekehrt.

„Beide Versionen sind *dual* zueinander.“

Ein Monitor kann *systematisch* in einen äquivalenten Server-Prozess transformiert werden.

Und wie steht es um die Pufferung?
Wo wird beim Monitor gepuffert?
Beschränkt oder unbeschränkt?

```
proc = fork server.service(arg);
```

Wenn der Monitor gerade aktiv ist, wird der erzeugte Prozess an der Monitorsperre blockiert.

Die so blockierten Prozesse fungieren als Träger der dienstbeschreibenden Daten. Die Warteschlange dieser Prozesse (i.a. unbeschränkt) entspricht der Auftragswarteschlange in der Mailbox eines Servers !

Vorteile von Monitoren:

Das häufig gehörte Argument zugunsten *verteilter Prozesse* „Vermeidung nebenläufiger Zugriffe auf gemeinsame Daten“ geht ins Leere:

- Monitor leistet das Gleiche (*sofern asynchron aufrufbar*)
- Monitor ist einfacher zu benutzen
- ... und vor allem einfacher zu programmieren!
- Monitor ist einfacher „abzuschwächen“
- Probleme bei *geschachtelten Monitoren* haben ihre genaue Entsprechung bei Prozessen,
die gleichzeitig Anbieter und Klienten sind
- Objektorientierung mit Vererbung/Polymorphie !

Vorteile von Prozessen:

- dienstunabhängige eigene Aktivität möglich
- asynchrone Diensterbringung möglich
- ... ohne dynamische Prozesserzeugung
- evtl. vorgezogene Antwort möglich
- flexible Ablaufsteuerung !
- Aufträge sind „schlanker“ als **fork**-Prozesse
- (Wenn es keine gemeinsamen Daten gibt, bleibt keine andere Wahl; z.B. physische Verteilung, Erlang, ...)

Die wichtigsten Vorteile der beiden Welten vereinigen?

→ 5.2

Beispiel: Ressourcenverwaltung

Gesucht: ein Monitor zur Verwaltung von Speicherbereichen
(z.B. im Hintergrundspeicher)

first = request(n) reserviert einen Speicher-
bereich aus n zusammenhängenden Blöcken für den
ausführenden Prozess, sobald möglich, und liefert
die Nummer des ersten Blocks.

release(first) gibt den Bereich wieder frei.

Voraussetzung: Klienten verhalten sich „korrekt“:
request/release abwechselnd, mit gleichem first
(→ keine Verklemmungen)

Verzögerter Monitor (*delayed monitor*) mit Wache (*guard*):

```
monitor StorageManagement {  
    int maxavail = 1000000;  
    .... // bookkeeping data, initialization  
public int request(int size) when size ≤ maxavail {  
    ... // bookkeeping  
    return first;  
}  
public void release(int first) {  
    ... // bookkeeping  
}  
}
```

Ablaufsteuerung (*scheduling*)?
Ressourcennutzung? Fairness? Verhungern?

Monitor mit *condition* und *signal-and-continue*:

```
monitor StorageManagement { //LRF - „largest request first“  
    int maxavail = 1000000;  
    .... // bookkeeping data, initialization  
public int request(int size) {  
    if size ≤ maxavail and no requests pending then go on  
    else store new pair(cond, size) and cond.wait  
    ... // bookkeeping  
    if largest request (c,s) has s ≤ maxavail then c.signal  
    return first; }  
public void release(int first) {  
    ... // bookkeeping  
    if largest request (c,s) has s ≤ maxavail then c.signal }  
}
```

Entsprechender Server-Prozess:

```
process StorageManagement { // „largest request first“  
    int maxavail = 1000000  
    ... // bookkeeping data, initialization  
    repeat p = recvarg(request, size)  
        if size ≤ maxavail and no requests pending then go on  
        else store new request(p, size) and continue receiving  
        ... // bookkeeping  
        sendres(first, p)  
    or     p = recvarg(release, first)  
        ...  
    } →
```

```

process StorageManagement { // „largest request first“
    ...
    ...
    or      q = recvarg(release, first)
            sendres(q)      // premature reply - void ! *
            ... // bookkeeping
            repeat r := next largest request (p,s)
                    if r is null or s > maxavail then break repeating
                    else update bookkeeping and get first
                    sendres(first, p)
                    remove request r
    }

```

* Eventuell entbehrlich - die Spezifikation könnte sagen:
 ein *release* gelingt sofort und wird nicht beantwortet.

5.2 Aktive Objekte

Wünschenswert:

- Identifizierung von Objektbegriff und Prozessbegriff
- und damit die Vorteile beider Welten vereinigen (S. 13).

Drei programmiersprachliche Annäherungen:

- Ada
- SR
- **Aktive Objekte**

Erster Versuch:

Ada - ursprünglich modulbasierte, imperative, nichtsequentielle Sprache, später um objektorientierte Konzepte erweitert.

Ada was originally designed by a team led by Jean Ichbiah of CII Honeywell Bull under contract to the United States Department of Defense (DoD) from 1977 to 1983 to supersede the hundreds of programming languages then used by the DoD. Ada was named after Ada Lovelace (1815-1852), who is sometimes credited as being the first computer programmer.

en.wikipedia.org/wiki/Ada_programming_language

Ada-Prozess: Signatur *task type* und Implementierung *task body*

```
task body Server is
begin ... -- prologue/initialization
loop
    select accept service1(arg: in Arg1; res: out Res1)
              do ... end;
    or      accept service2(arg: in Arg2; res: out Res2)
              do ... end;
    or      ...
    end select;
end loop;
...           -- epilogue
end;
```

nimmt Auftrag entgegen,
führt ihn aus und
schickt Ergebnis *res* zurück

! Signatur macht die starke Typisierung deutlich:

```
task type Server(...) is
entry service1(arg: in Arg1; res: out Res1);
entry service2(arg: in Arg2; res: out Res2);
...
end Server;
```

Der Klient:

```
server: Server(); ..... -- declare Server task
server.service1(a, r); ...
```

sendet die Anforderung, den Dienst `service1` mit Argument `a` auszuführen, und wartet (!) auf das Ergebnis `r`.

Bemerkungen:

Keine asynchrone Dienstausführung für den Klienten -
aber ansonsten ist die Server Task ein eigenständiger Prozess!

- `server.servicel(a, r)` heißt *remote invocation*
- ... und blockiert, bis der Server ein entsprechendes `accept` ausführt - vgl. Occam (3.2).
- Umgekehrt wartet `accept` auf eine remote invocation.
- Alsdann findet ein *rendezvous* der beiden Tasks statt,
und der `accept`-Rumpf wird ausgeführt.

Feststellung:

Die *remote invocation* in Ada ist ein **Zwitter**
aus **Beauftragung eines Prozesses** und **Aufruf eines Monitors**

Vorteile: fast alle Vorteile von Prozessen (S. 13) ,
einige Vorteile von Monitoren (S. 12):
Aufruf-Syntax, **entry**-Syntax, Typisierung!

Nachteile: synchrone Diensterbringung,
Ausschluss kann nicht abgeschwächt werden,
keine Vererbung (jedenfalls nicht für *task types*)

Implementierung:

Typischerweise Tasks als Threads.

Ob der **accept**-Rumpf vom aufrufenden Thread oder vom Thread des Servers ausgeführt wird, ist egal.

Wenn der Server die Struktur von S. 20 hat und keinen Prolog und Epilog hat, kann er als Monitor implementiert werden !

Zweiter Versuch:

SR - „Synchronizing Resources“ [Andrews 1981]

- objektbasierte nichtsequentielle Sprache
- Sprachkonstrukt `resource` stellt gleichzeitig eine „Klasse“ und einen nichtsequentiellen (!) Prozess dar
- Ada-ähnliche Prozess-Interaktion

```
resource StorageManagement
    define request{call}, release
    var maxavail: integer; .....
    maxavail := 1000000; .....

process manager;
    do true ->
        in request(size: integer; var first: integer)
            and size<=maxavail -> .....
        [] release(first: integer)      -> .....
        ni
    od
    end manager
process other;... something else ... end other
end StorageManagement
```

Erläuterungen:

- **resource** ähnlich wie **class** - aber keine Vererbung/Polymorphie.
- **define** vereinbart die von Klienten benutzbaren Operationen.
- Die Anweisung **in ...[]...[]... ni** enthält die Signaturen und Implementierungen alternativ auszuführender Operationen
 - ähnlich einem Ada **select accept** !
- In einer **resource** können beliebig viele Prozesse **process** vereinbart werden!
- Benutzung einer Operation:
call request(3,x) (hier vorgeschrieben im **define**)
call release(x) oder **send release(x)** !

Stärken: - **resource** ist „nichtsequentieller Prozess“
 - **send** versus **call**

Schwächen: - keine Vererbung
 - keine Asynchronie mit *Ergebnis*

Implementierung:

auch physisch verteilt, mit Sprachunterstützung:

```
aspace := create vm() on xian;  
storage := create StorageManagement on aspace;
```

Dritter Versuch:

Aktive Objekte

- verschiedene Sprachen seit ca. 1985
- objektorientiert und nichtsequentiell
- Objekte sind zugleich Prozesse
- Vererbung ist teilweise knifflig
- Übersicht: [Briot et al. 1998]

Beispiel CEiffel/JAC [Löhr 1992]

```
class StorageManagement {  
    int maxavail = 1000000;  
    ....  
@async  
@when size <= maxavail  
public Int request(int size){      // →S. 32  
    ...  
    return first;                  }  
public void release(int first) {  
    ...  
}  
@auto  
@when required  
protected void other() { .... }  
}
```

Erläuterungen:

- JAC (*Java with Annotated Concurrency*) benutzt Annotationen wie z.B. `@async` (ursprünglich `/** @async */`)
- Asynchron ausgeführte Operationen - wenn nicht `void` - liefern eine *Future*; der Ergebnistyp darf nicht `final` sein.
- Autonome Operationen werden "immer wieder" ausgeführt - sofern die Wache es zulässt.
- Wechselseitiger Ausschluss von Operationen wird über Verträglichkeits-Annotationen geregelt.

Implementierung:

- Vorübersetzer nach Java
- keine physische Verteilung
- Übersetzer entscheidet, ob ein Objekt tatsächlich mit Prozess(en) und Nachrichten oder aber monitor-ähnlich implementiert wird

5.3 Drei Sprachen

Zurück zu explizitem Nachrichtenversand/empfang:

- ① Erlang
- ② Scala
- ③ F#

5.3.1 Erlang

Erlang ist *nicht* dienstorientiert - was tun?

```
server !{service, Arg, self() }  
receive {service, Arg, Client } ->  
      .... Client ! Res
```

besser: Client ! {server, service, Res}

noch besser Aufträge mit eindeutiger Kennung:

```
server !{service, Arg, self(), RequestNumber }  
receive {service, Arg, Client, RequestNumber } ->  
      .... Client ! {RequestNumber, Res}
```

5.3.2 Scala

Scala [Odersky et al. 2011]:

Amalgam aus objektorientierten und funktionalen Elementen
(läuft auf der Java Virtual Machine)

Nichtsequentialität: Prozesse („actors“) interagieren sowohl
über gemeinsame Daten als auch mit Kommunikation:

```
object Server extends Actor {  
    def act() { ... receive Fallunterscheidung ... }  
}  
  
... Server.start(); ... Server ! "hello!"; ...
```

`receive` ist Methode von `Actor` :

Argument ist eine partielle Funktion f

(Fallunterscheidung mit Musteranpassung)

Ergebnis ist f(message)

Beispiel: `receive {`

`case Service1(arg)` => ...

`case Service2(arg1, arg2)` => ...

`case n: Int` => ...

`case msg` => ...

`}`

`receive` blockiert, bis eine *passende* Nachricht eintrifft,
tätigt die Musteranpassung und wertet den zugehörigen
Ausdruck aus. Nirgends passende Nachrichten werden ignoriert.

Dienstorientierung wird unterstützt:

```
receive {  
    case Service1(arg) =>  
        ...  
        reply(res)  
    case ...  
}
```

Actor-Methode `reply` bezieht sich auf das umschließende `receive`, übermittelt Ergebnis der Anfrage an den Klienten.

... und der Klient:

`Server ! Service1(arg)`

asynchrone Auftragsausführung, Antwort geht verloren

`res = Server !? Service1(arg)`

synchrone Auftragsausführung, liefert die Antwort

`f = Server !! Service1(arg)`

*asynchrone Auftragsausführung, liefert eine Future,
d.i. eine nullstellige Funktion, die die Antwort liefert,
sobald sie vorliegt*

`res = f()`

5.3.3 F#

F# („F Sharp“, dt. *Fis*) [Syme et al. 2010] :
Amalgam aus objektorientierten und funktionalen Elementen
(läuft auf Microsoft .NET)

Nichtsequentialität:

- Prozesse („actors“) ähnlich wie in Scala
(etwas weniger elegant)
- *asynchronous workflows* erlauben hochgradig parallele Ausführung mehrerer Auswertungen

Zusammenfassung

- „Klient/Anbieter“ eng verwandt mit „Aufrufer/Aufgerufener“.
- Zu jedem Monitor gibt es einen strukturell ähnlichen Prozess.
- Dienstanforderung kann als asynchroner Aufruf „verkleidet“ werden!
- Aktives Objekt ist zugleich Prozess und Objekt. Damit verschwimmt der Unterschied zwischen verteilten und nichtverteilten Programmen.
- Merke: *Dienstorientierung* ist für Anwendungssoftware höchst relevant - wie *Datenfluss*, aber anders als *verteilte Algorithmen*.

Quellen

H.C. Lauer, R.M. Needham: *On the duality of operating system structures.*
Proc. 2. Int. Symp. on Operating Systems, October 1978
dl.acm.org/citation.cfm?id=850658

Ada: www.adaic.org/ada-resources

(gut:) en.wikipedia.org/wiki/Ada_programming_language

G.R. Andrews: *Synchronizing Resources.* ACM TOPLAS 3.4, October 1981
dl.acm.org/citation.cfm?id=357149

K.-P. Löhr: *Concurrency Annotations.* Proc. OOPSLA '92, ACM Sigplan Notices 27.10, October 1992
dl.acm.org/citation.cfm?id=141964

J.-P. Briot et al.: *Concurrency and distribution in object-oriented programming.*
ACM Computing Surveys 30.3, September 1998
dl.acm.org/citation.cfm?id=292470

M. Odersky et al.: *Programming in Scala.* Artima 2011 (2. ed.)
www.scala-lang.org
[en.wikipedia.org/wiki/Scala_\(programming_language\)](https://en.wikipedia.org/wiki/Scala_(programming_language))

D. Syme et al.: *Expert F# 2.0.* Apress 2010.
[en.wikipedia.org/wiki/F_Sharp_\(programming_language\)](https://en.wikipedia.org/wiki/F_Sharp_(programming_language))