ModSoft

Modellbasierte Software-Entwicklung mit UML 2
im WS 2014/15

Teil III a: Verhaltensmodellierung

Prof. Dr. Joachim Fischer
Dr. Markus Scheidgen
Dipl.-Inf. Andreas Blunk

fischer@informatik.hu-berlin.de

Inhalt

1. Zuordnung: Strukturmodellentität – Verhaltensmodellentität
2. Semantik des UML-Zustandsautomaten (noch nicht vollständig)
3. Alternative Syntax (Ähnlichkeit zu SDL)
4. Dämonenspiel als „UML“-Modell (System)
   a) Informale Beschreibung
   b) SDL: instanz-basierte Definition
   c) Systemerweiterung
   d) UML: typ-basierte Definition
   e) Annahmen für eine mögliche Interpretation von UML
5. UML-SDL-Tool (PragmaDev)

Demon Game: Anforderungen (Wdh.)

Idee

- man hat als Spieler (Systemumgebung) zu erraten,
  ob eine Black-Box-Variable ungerade ist
- die Variable wird von einem Dämon geändert: gerade (even)←→ungerade(odd)
  (nicht vorhersagbar)
- liegt man mit seiner Vermutung richtig, erhält der Spieler einen Punkt,
  wenn nicht, wird ihm ein Punkt entzogen
- zulässig ist eine sich dynamisch ändernde Anzahl von Spielern,
  die sich an- bzw. abzumelden haben
- Ein Spieler kann zu einem Zeitpunkt nur maximal an einem Spiel angemeldet sein

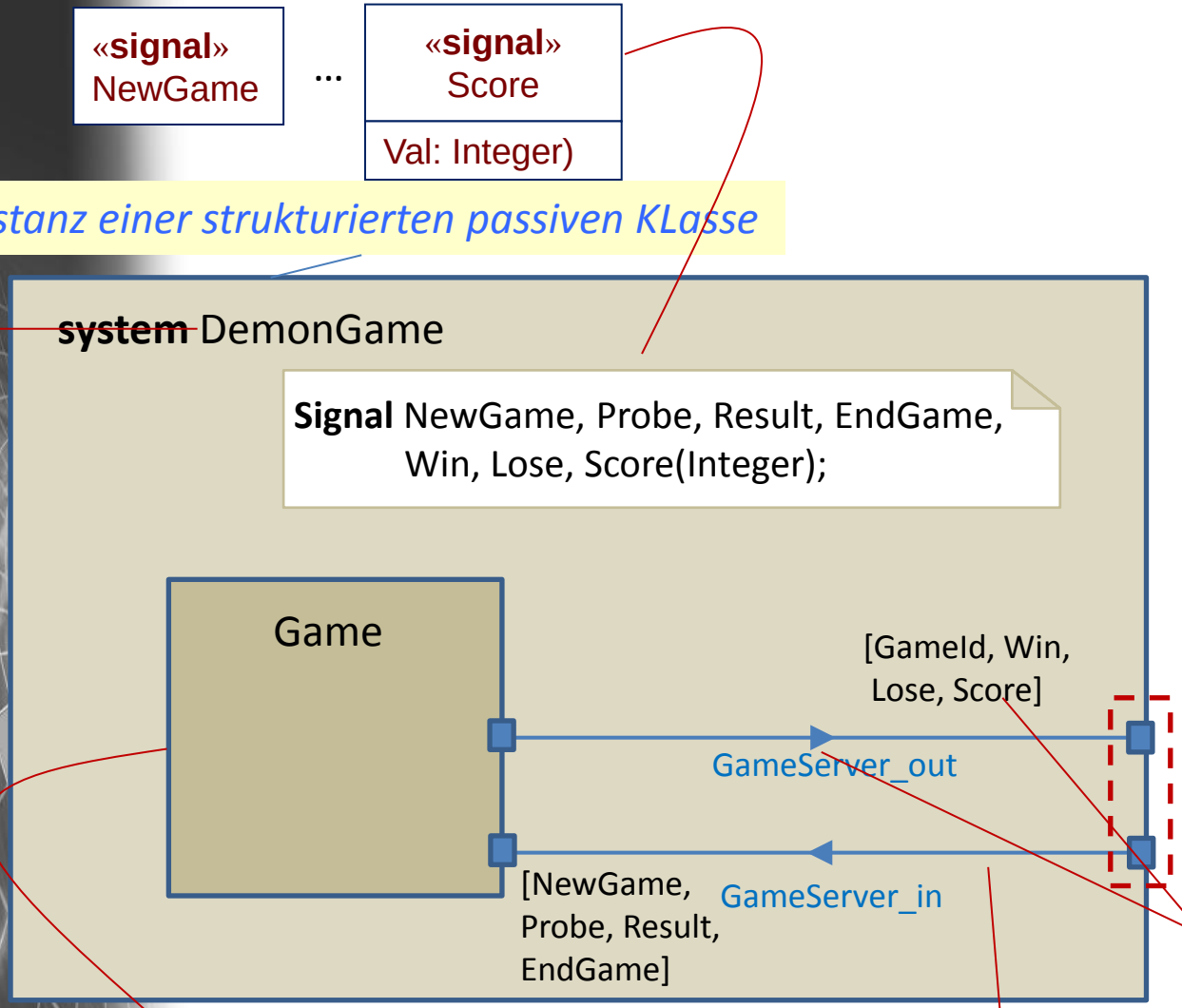
Aufgabe

- Spiel ist als reaktives System zu beschreiben, jede Spielanforderung muss unabhängig von den anderen bearbeitet werden
- Spiel und Spieler kommunizieren per Signalaustausch

Probleme

- Reaktivität, dynamische Änderung der Systemstruktur, Verteilung

Demon Game: als System (Wdh.)



Systemfunktionalität
Spieler aus der Umgebung
- melden sich an
- spielen
  (raten richtig/falsch
  und bekommen
  Punkte)
- fragen Punktestand ab
- melden sich an

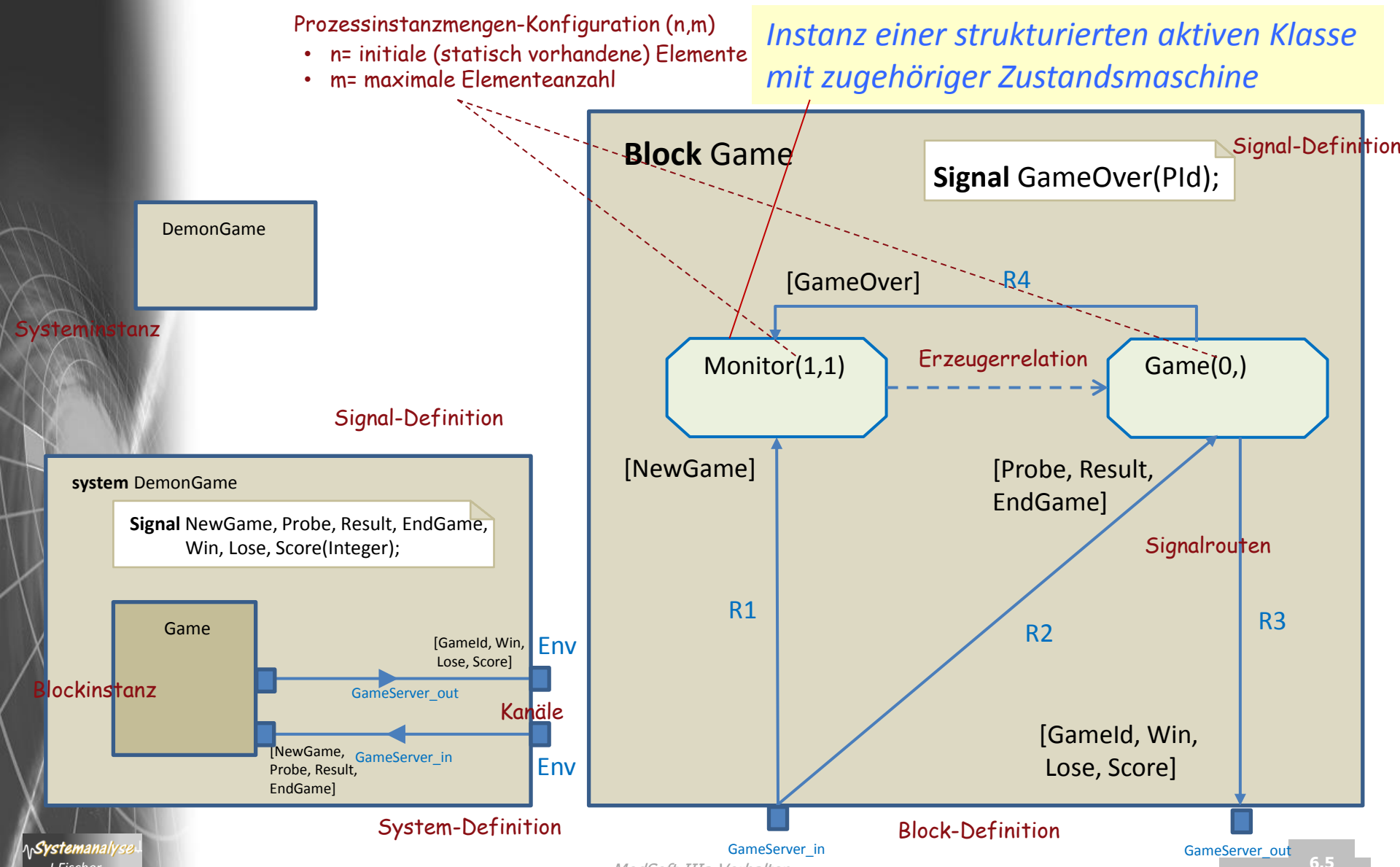
nur angemeldete Spieler dürfen spielen

Ein Spieler kann zu einem Zeitpunkt nicht mehr als einmal angemeldet sein

Port mit zwei Interfaces

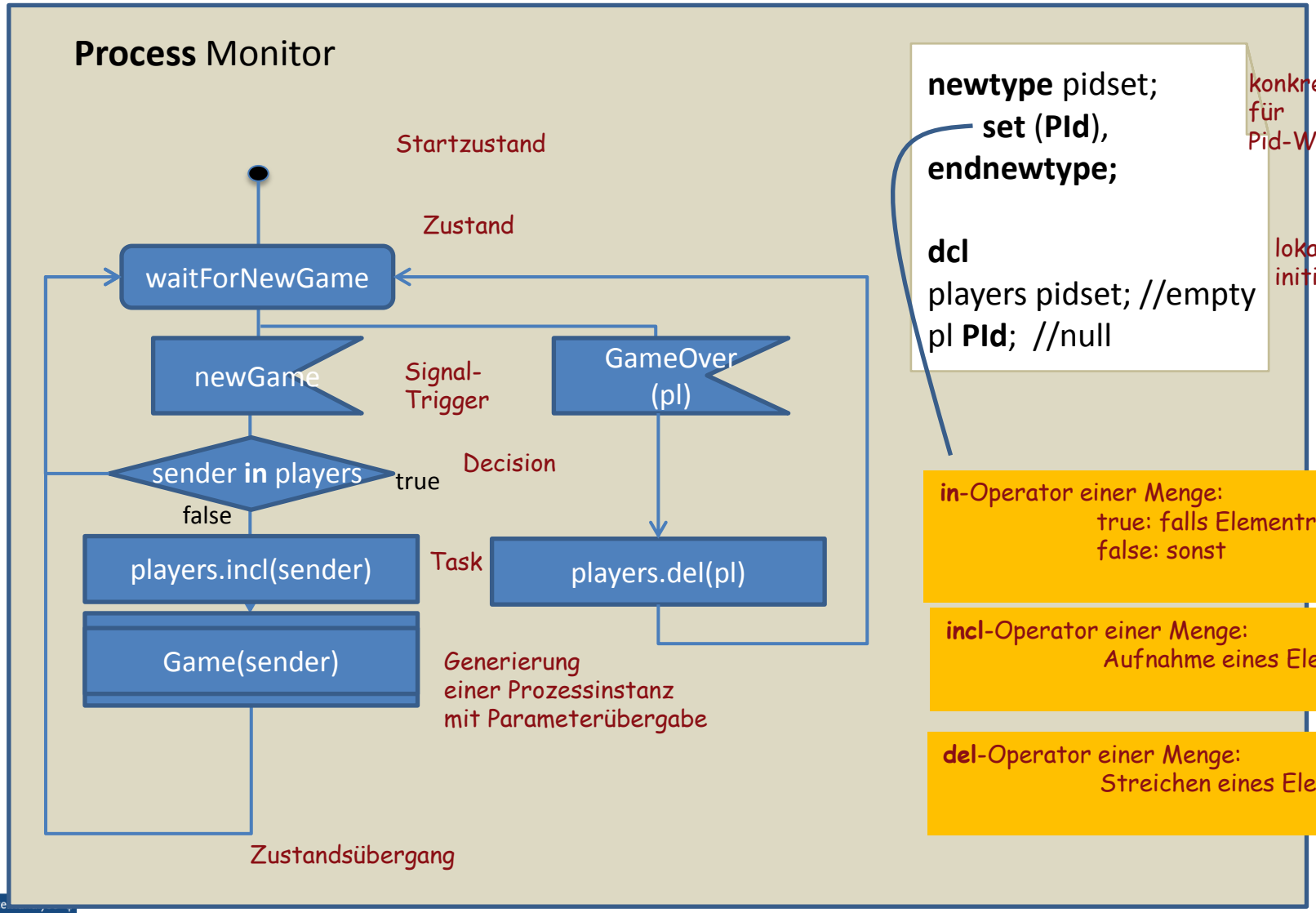
Richtung und Signalmenge bestimmen Interface

Demon-Game: System-Block-Struktur (Wdh.)



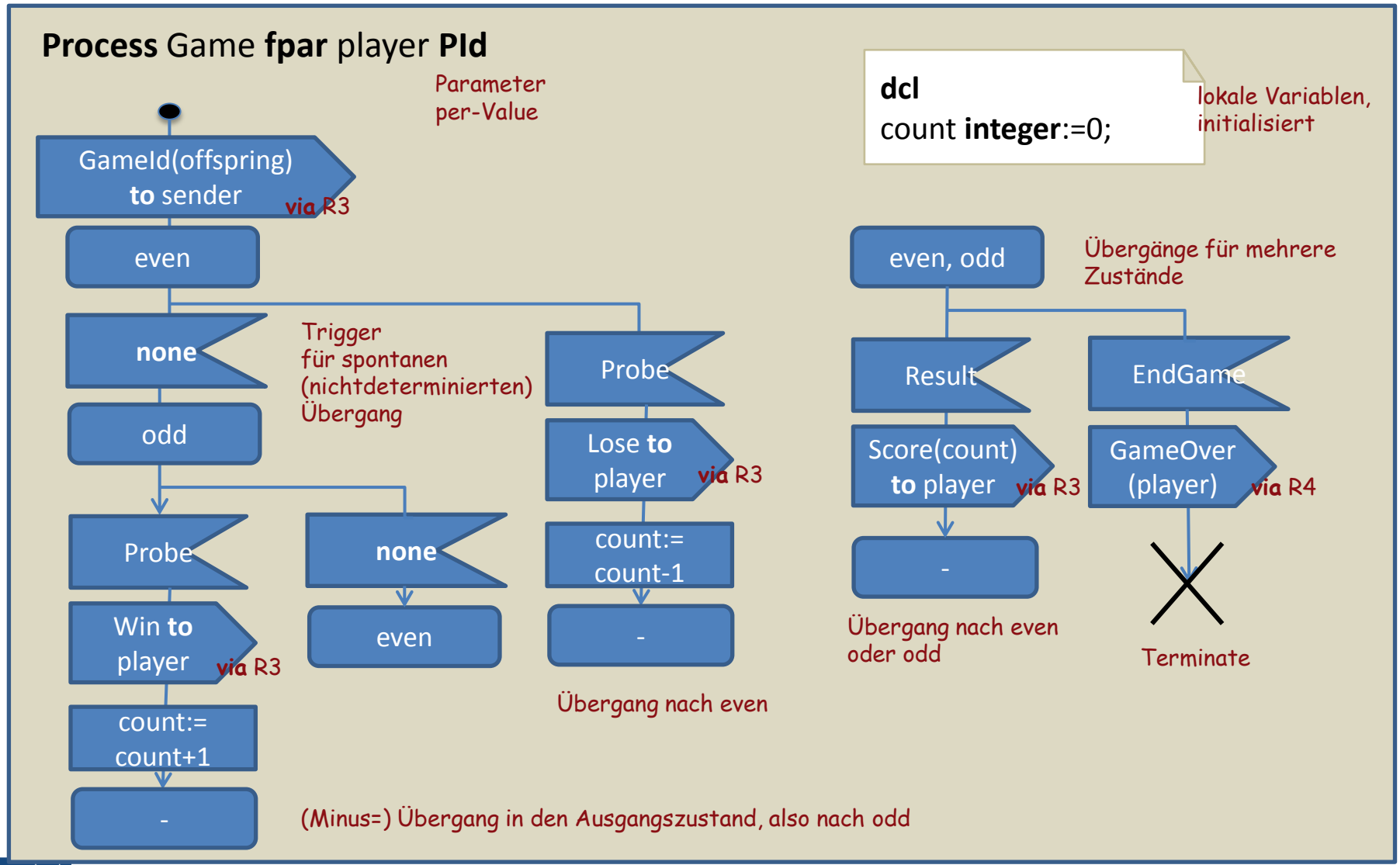
Demon-Game: Verhalten (Wdh.)

Repräsentantendefinition der Menge Monitor



Demon-Game: Verhalten (Wdh.)

Repräsentantendefinition der Menge Game

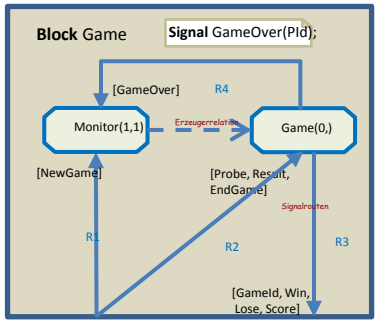


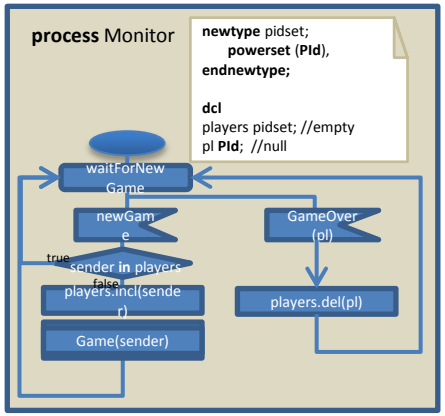
Inhalt

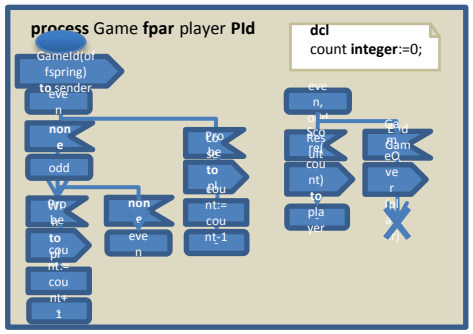
1. Zuordnung: Strukturmodellentität – Verhaltensmodellentität
2. Semantik des UML-Zustandsautomaten (noch nicht vollständig)
3. Alternative Syntax (Ähnlichkeit zu SDL)
4. Dämonenspiel als „UML“-Modell (System)
   a) Informale Beschreibung
   b) SDL: instanz-basierte Definition
   c) Systemerweiterung
   d) UML: typ-basierte Definition
   e) Annahmen für eine mögliche Interpretation von UML
5. UML-SDL-Tool (PragmaDev)

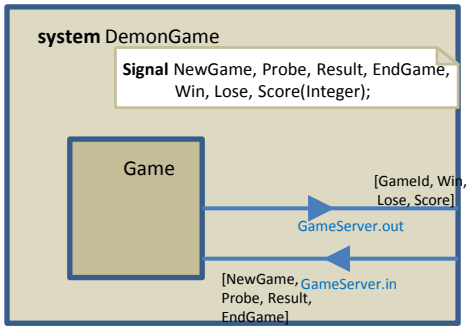
Einbeziehung des Spielerverhaltens

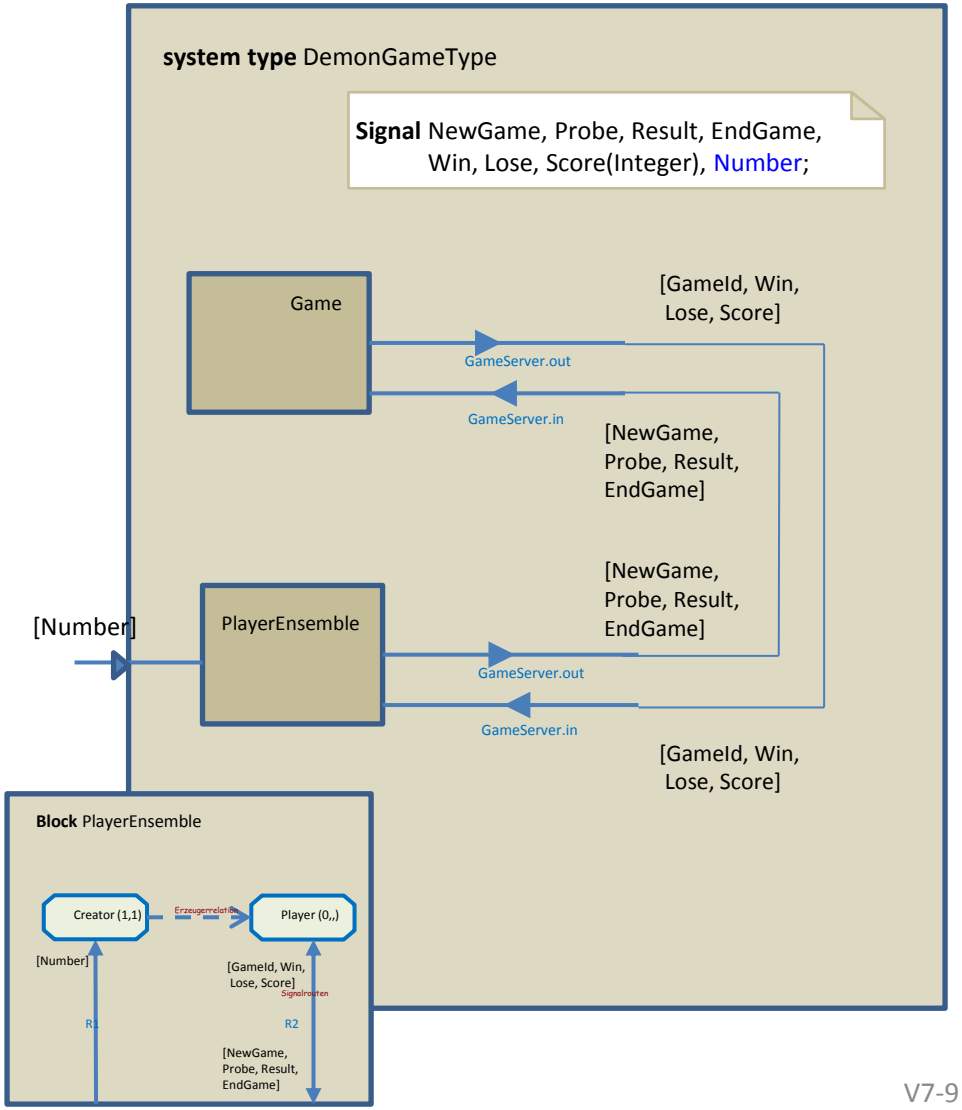
~ SDL









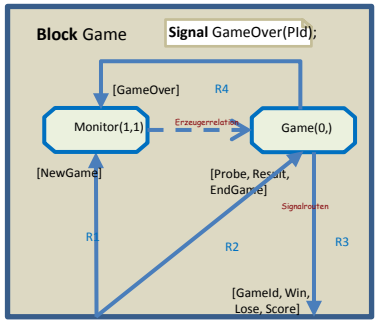


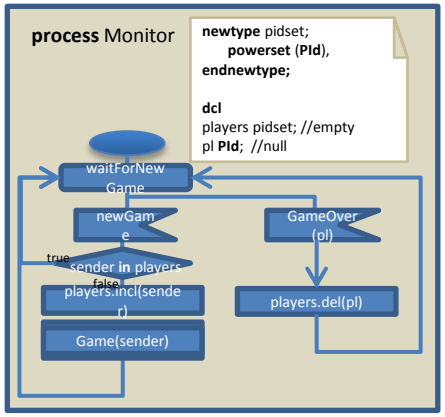
Inhalt

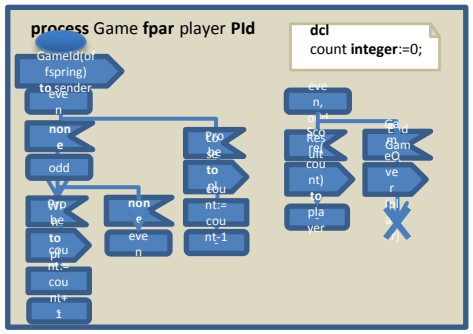
1. Zuordnung: Strukturmodellentität – Verhaltensmodellentität
2. Semantik des UML-Zustandsautomaten (noch nicht vollständig)
3. Alternative Syntax (Ähnlichkeit zu SDL)
4. Dämonenspiel als „UML“-Modell (System)
   a) Informale Beschreibung
   b) SDL: instanz-basierte Definition
   c) Systemerweiterung
   d) UML: typ-basierte Definition
   e) Annahmen für eine mögliche Interpretation von UML
5. UML-SDL-Tool (PragmaDev)

Instanzbasiert – typbasierte Definition

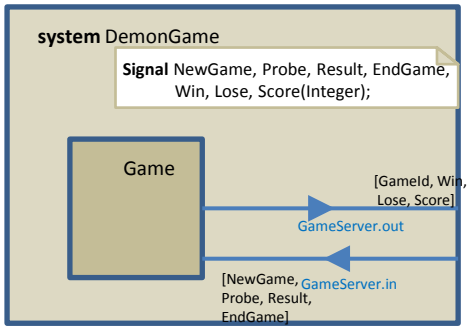
~ SDL



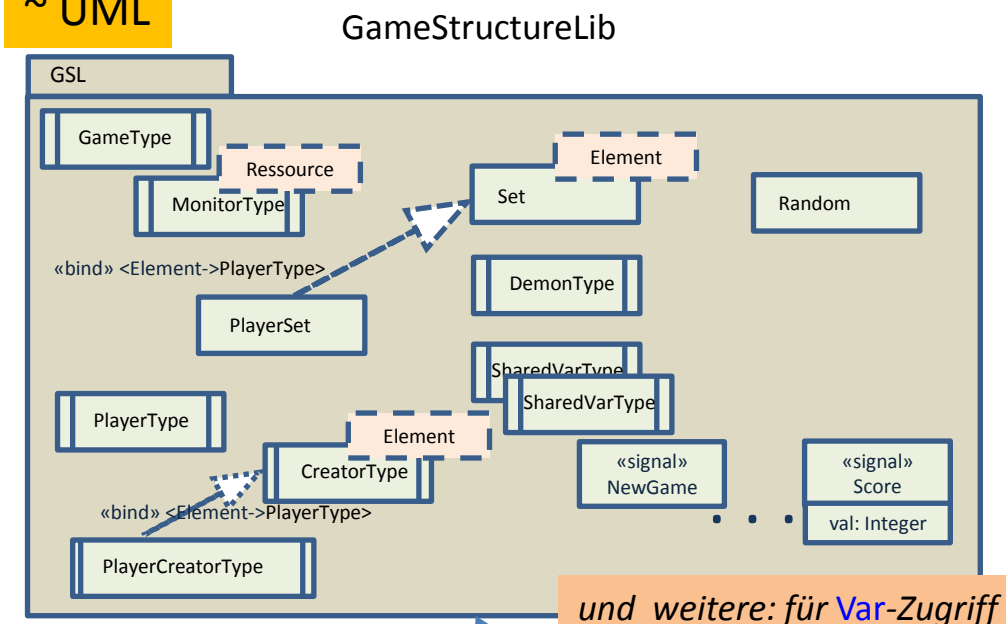




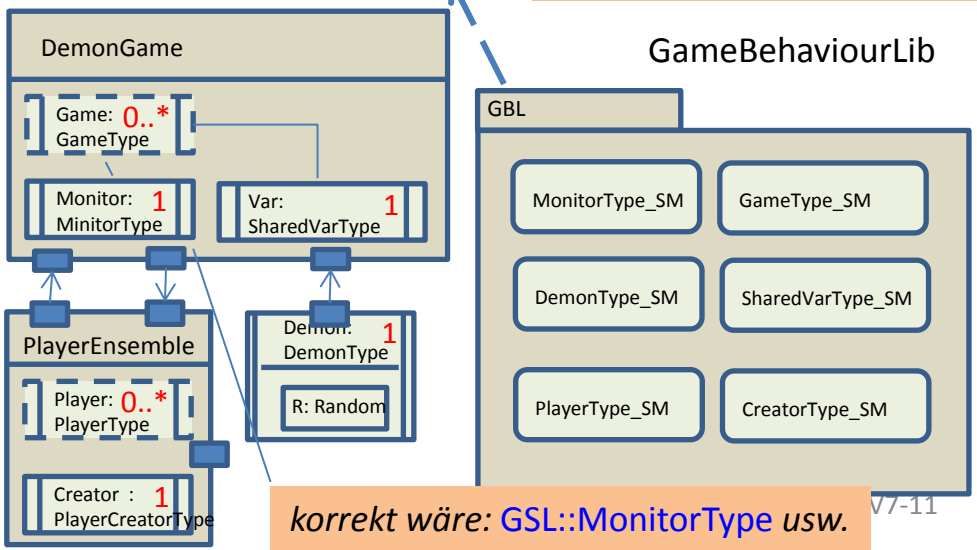
DemonGame



~ UML



und weitere: für Var-Zugriff



korrekt wäre: GSL::MonitorType usw.

Inhalt

1. Zuordnung: Strukturmodellentität – Verhaltensmodellentität
2. Semantik des UML-Zustandsautomaten (noch nicht vollständig)
3. Alternative Syntax (Ähnlichkeit zu SDL)
4. Dämonenspiel als „UML“-Modell (System)
   a) Informale Beschreibung
   b) SDL: instanz-basierte Definition
   c) Systemerweiterung
   d) UML: typ-basierte Definition
   e) Annahmen für eine mögliche Interpretation von UML
5. UML-SDL-Tool (PragmaDev)

Erkenntnisse

- Strukturierte Classifier eignen sich zur Darstellung von Systemen und Teil-Systemen (**Kompositionsstrukturdiagramm**) durch Angabe ihrer **Bestandteile** und von Kommunikationspunkten (**Ports** mit wohldefinierten Interface-Beschreibungen)

- Bestandteile eines (Teil-)Systems, die permanent vorhanden sind,
  lassen sich von Bestandteilen temporärer Art syntaktisch unterscheiden.
  Ihre Multiplizität kann angegeben werden.

- Die **Zuordnung von Zustandsmaschinen** (Instanzen der im Paket GameBehaviourLib definierten Typen) zu den Teilen,
  die über aktive Klassen beschrieben sind, ist unklar
  (könnte aber über ein UML-Werkzeug realisiert werden)

- Nicht definiert ist, wie das Verhalten einer Instanz eines strukturierten Classifiers (System) **initiiert** wird.

  vernünftige Annahme (I):
  Für alle Bestandteile, die mittels aktiver Klassen beschrieben und intial vorhanden sind,
  werden implizit die zugehörigen Verhaltensbeschreibungen
  (hier: Instanzen von Zustandsmaschinen) gestartet.

aus letzter Vorlesung

Annahme (II): Übernahme spezieller SDL-Verhaltenskonzepte

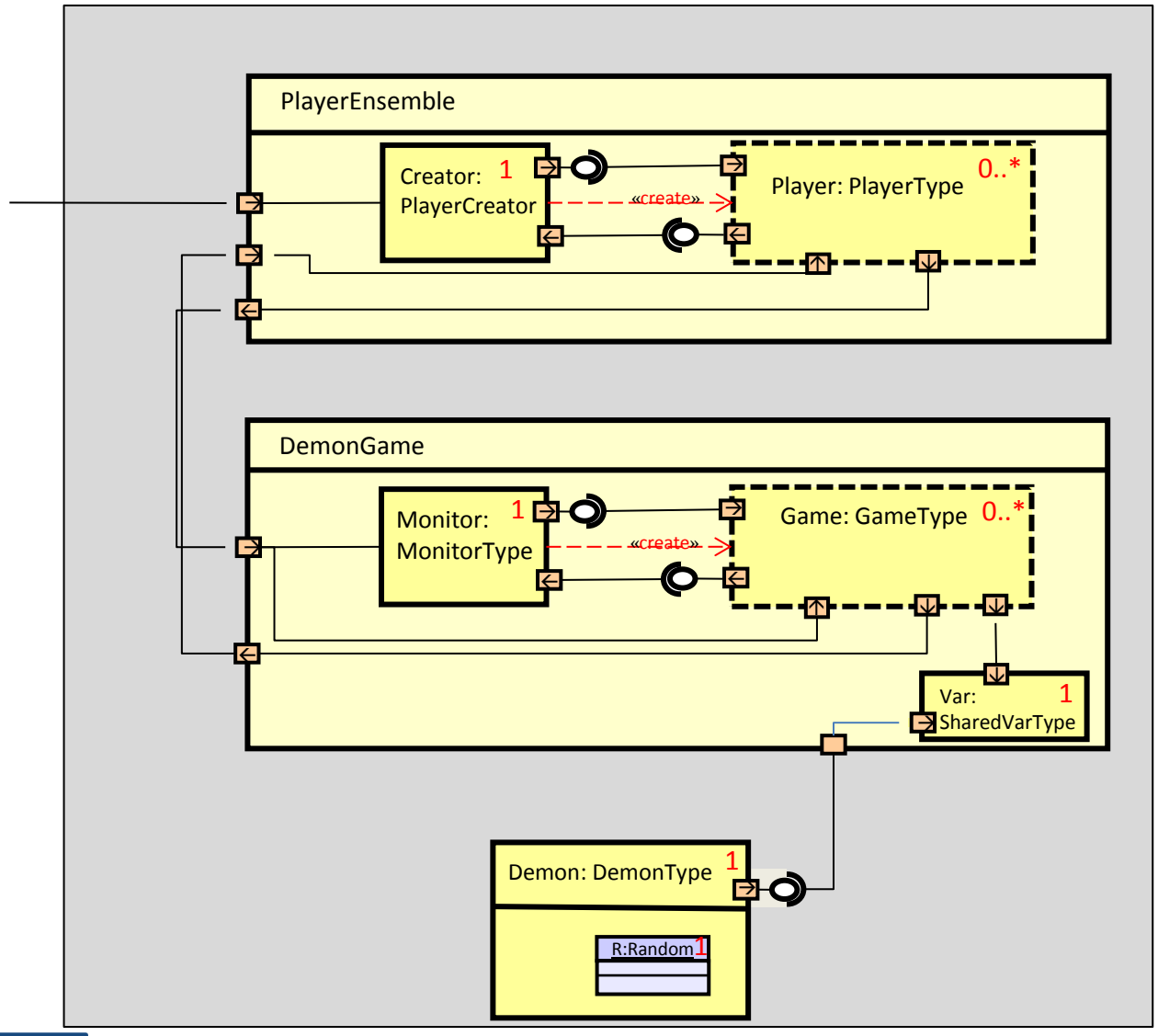
Vereinbarungen
jede Automateninstanz verfügt über

- **myself** ~ Referenz zum zugehörigen Objekt einer aktiven Klasse
- **self** ~ Automateninstanzreferenz auf sich selbst
- **sender** ~ Automateninstanzreferenz zum Sender des aktuellen Signals
- **parent** ~ Automateninstanzreferenz zum Erzeuger
- **newprocess** ~ Erzeugung eines Objekts einer aktiven Klasse und Aktivierung der Automateninstanz

- **offspring** ~ Referenz zur letzten erzeugten Automateninstanz

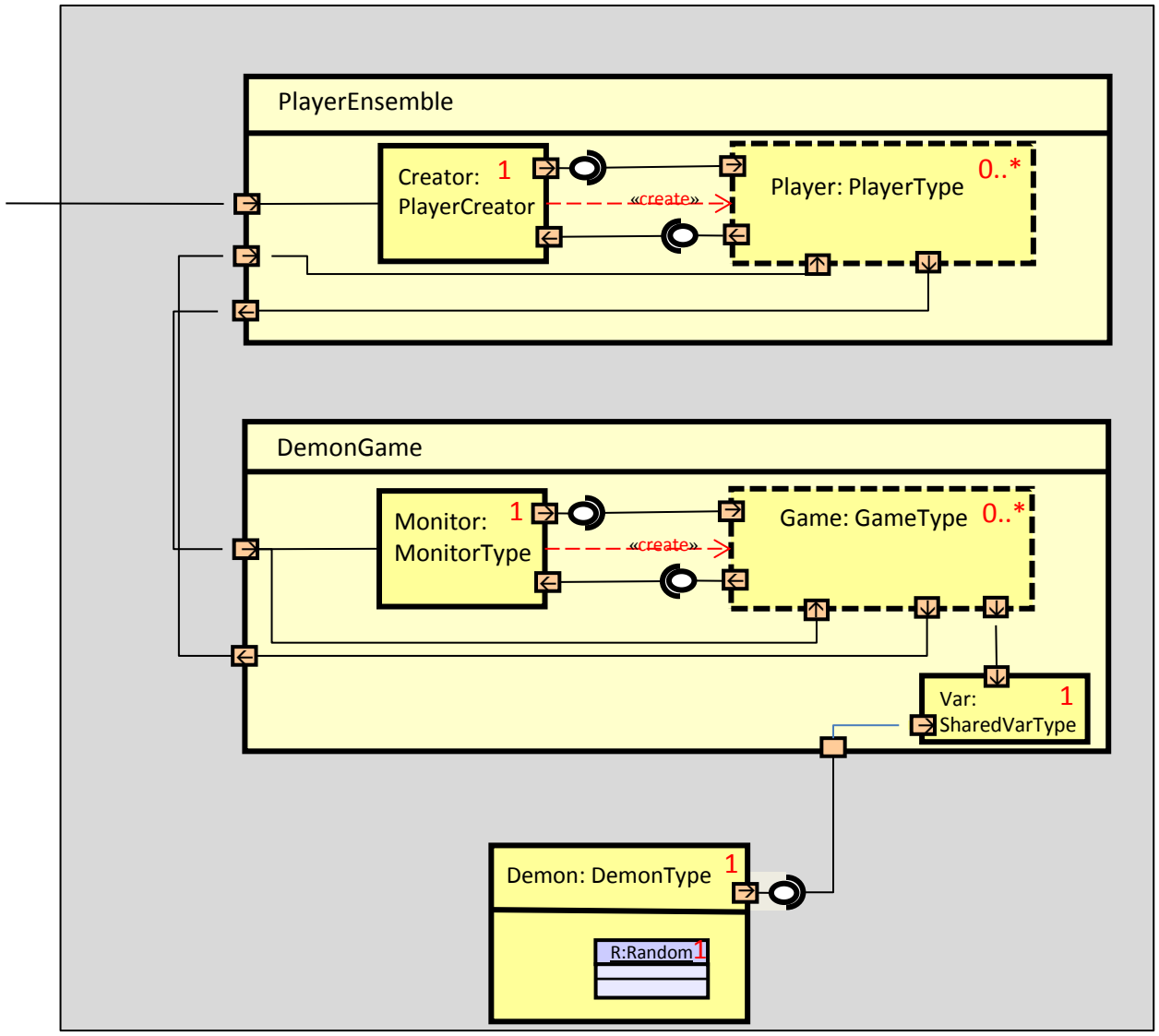
Mit Annahmen (I) und (II) wird eine Interpretation des UML-Modells möglich

Systemkonfiguration



1. PlayerEnsemble-Instanz
2. Creator-Instanz
3. Start der Creator-Zustandsmaschine …wartet auf Eingabe
4. Monitor-Instanz
5. Start der Monitor-Zustandsmaschine
6. Var-Instanz
7. Start der Var-Zustandsmaschine
8. … Legt Wert fest, wartet auf Lese- und Schreib-Signale
9. Demon-Instanz mit lokaler R-Instanz
10. Start der Demon-Zustandsmaschine
11. … bestimmt zufällige Zeitdauer und wartet auf Timeout

Systemstart

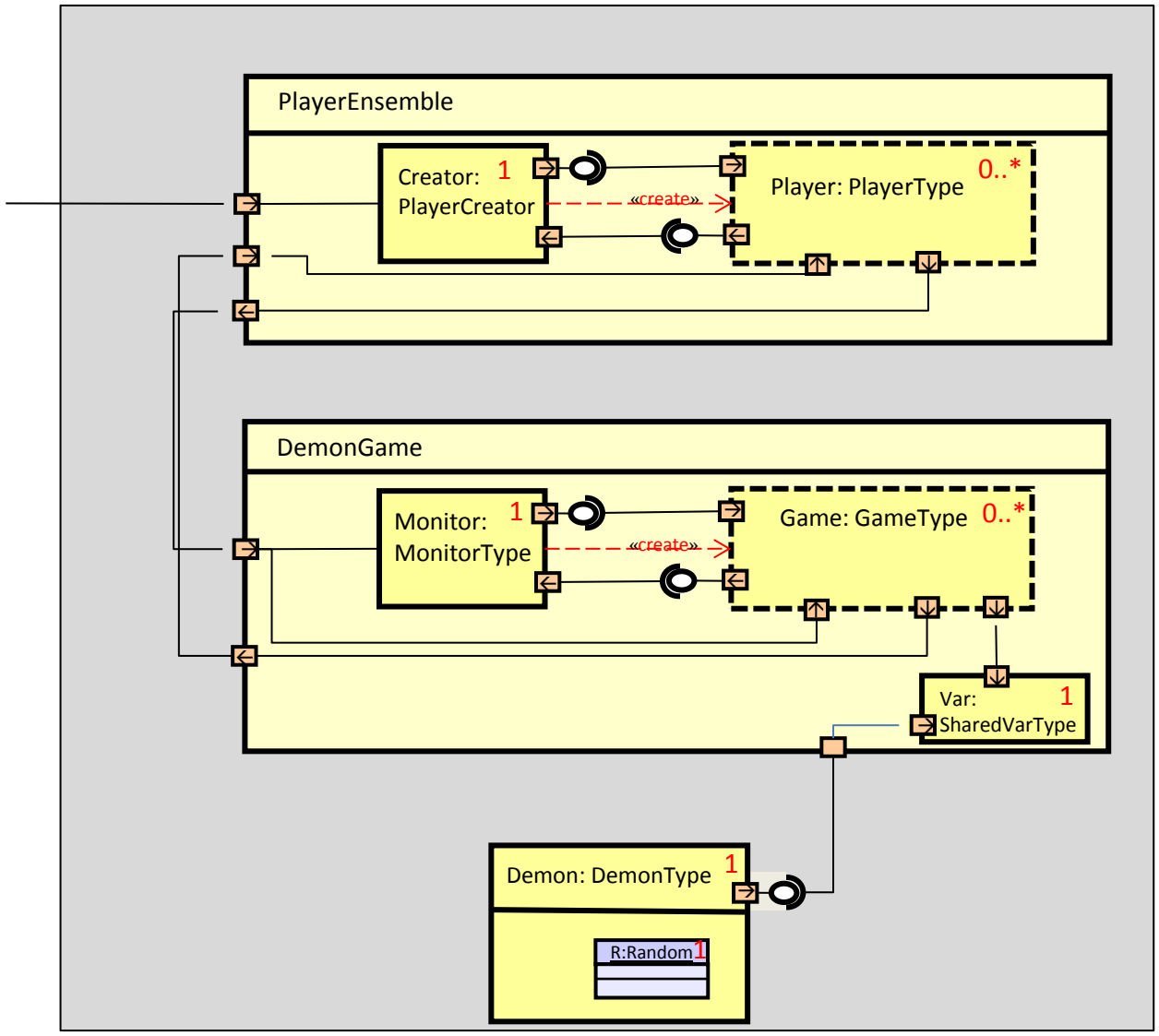


1. Eingabe des Signals Number (mit Parameter 10)
2. … Empfang von Creator-Instanz
3. Erzeugung von 10 PlayerType-Instanzen und Start der Zustandsmaschinen
4. …wartet auf Eingabe

… Player-Instanzen werden aktiv (werden NewGame-Signale verschicken

… unabhängig davon wird Demon-Instanz aktiv (werden NewGame-Signale verschicken

Verhalten der geteilt genutzten Variable



Variable vom Typ Boolean

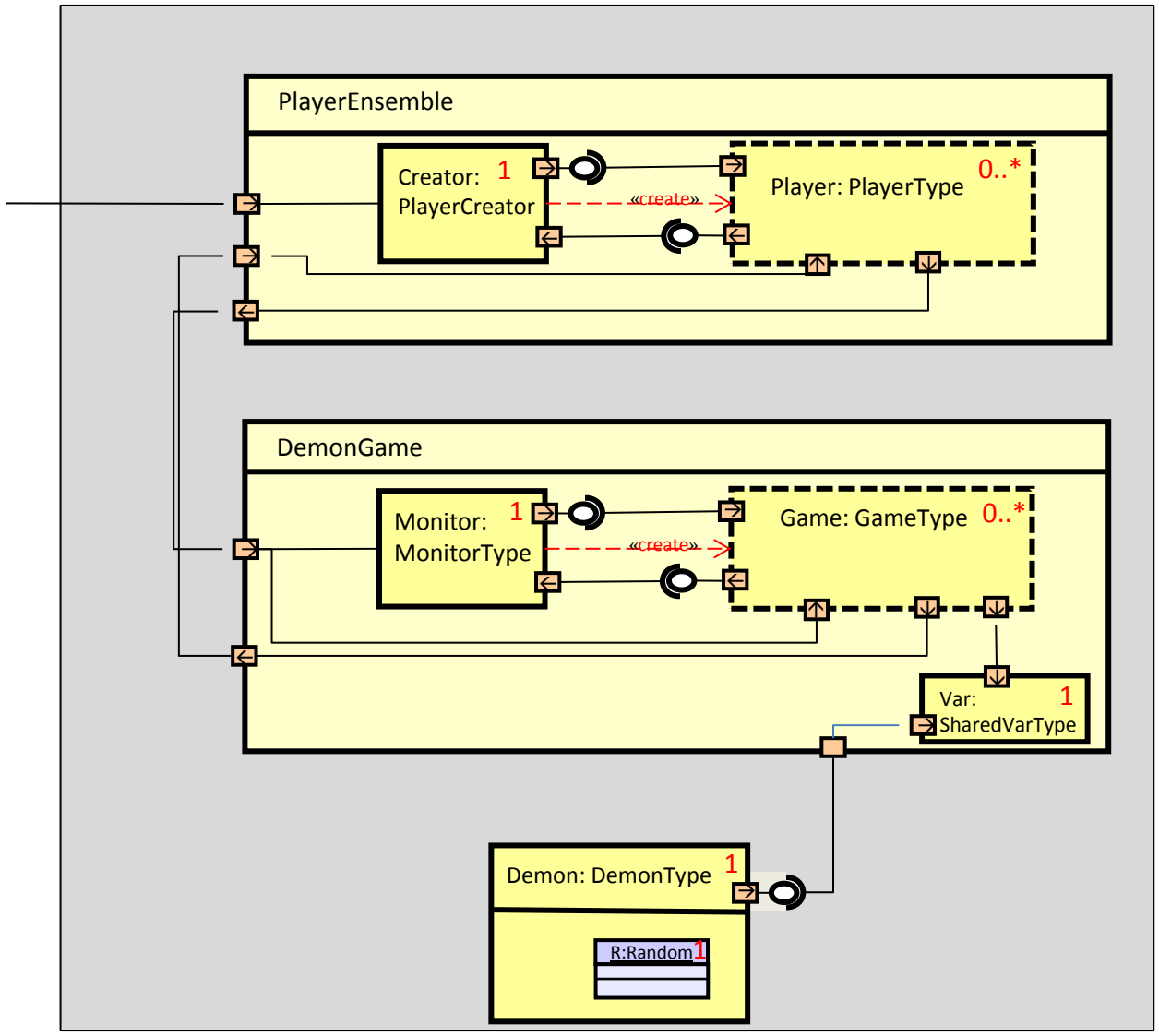
Zustandsmaschine sichert den sequentialisierten parallelen Zugriff zum lesen und Schreiben (als Konfliktlösung)

… Demon-Instanz
Wird unregelmäßig Signale senden

…Var-Zustandsmaschine wird

- bei Empfang eines Schreibsignals den Bool'schen Wert ändern
- Bei Empfang eines LeseSignals (Probe) im Fall von True das Signal Win an den Sender (Game-Instanz) schicken und sonst Lose

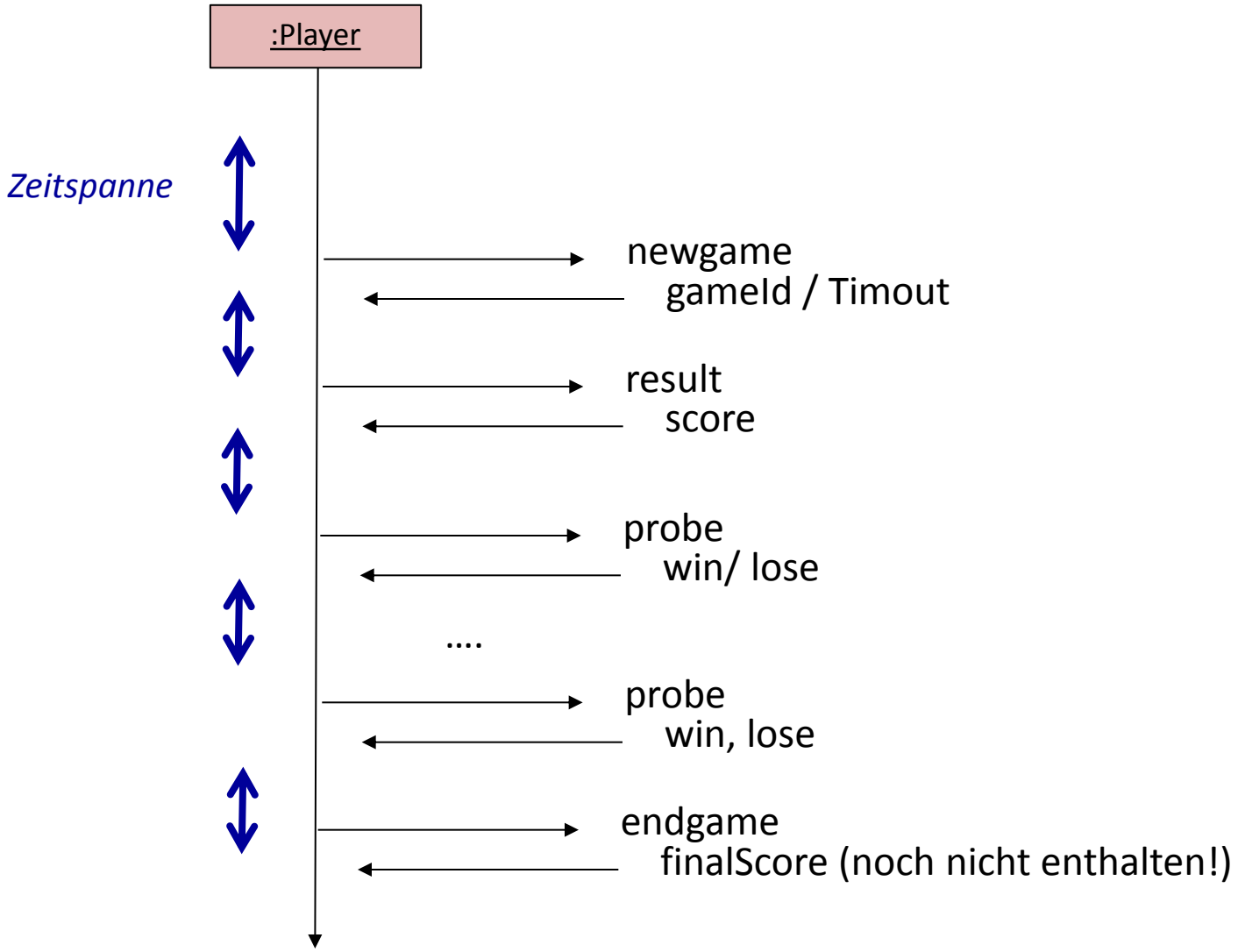
Spielzug eines Spielers, Abmeldung eines Spielers



Machen Sie sich den Ablauf klar!

Wie kann man ein spezifisches Verhalten für jede Playerinstanz erreichen?

Möglicher Verhaltenstrace einer Player-Instanz



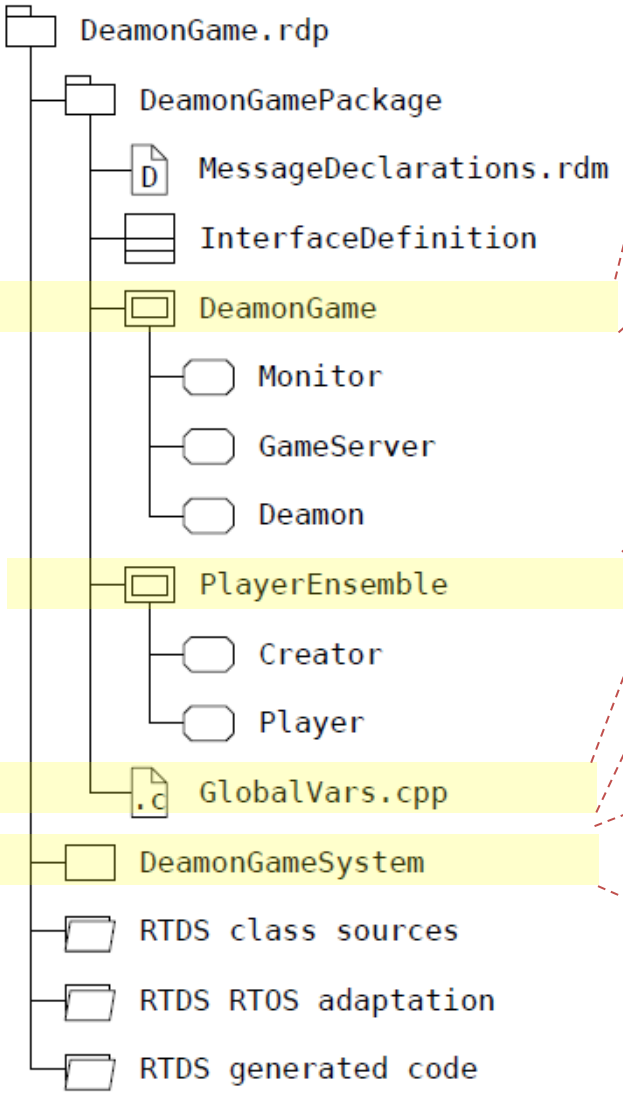
Inhalt

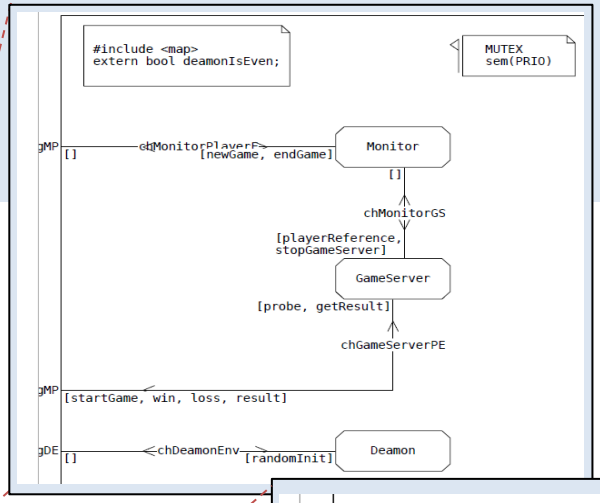
1. Zuordnung: Strukturmodellentität – Verhaltensmodellentität
2. Semantik des UML-Zustandsautomaten (noch nicht vollständig)
3. Alternative Syntax (Ähnlichkeit zu SDL)
4. Dämonenspiel als „UML“-Modell (System)
   a) Informale Beschreibung
   b) SDL: instanz-basierte Definition
   c) Systemerweiterung
   d) UML: typ-basierte Definition
   e) Annahmen für eine mögliche Interpretation von UML
5. UML-SDL-Tool (PragmaDev)

Inhalt

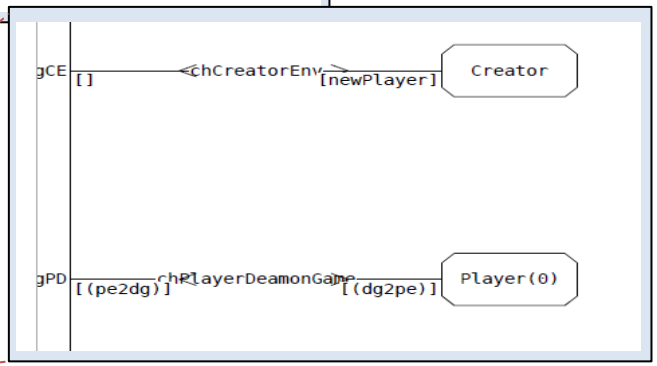
1. Zuordnung: Strukturmodellentität – Verhaltensmodellentität
2. Semantik des UML-Zustandsautomaten (noch nicht vollständig)
3. Alternative Syntax (Ähnlichkeit zu SDL)
4. Dämonenspiel als „UML“-Modell (System)
   a) Informale Beschreibung
   b) SDL: Komplett instanz-basiert vorgenommene Definition
   c) Systemerweiterung
   d) UML: Komplett typ-basiert vorgenommene Definition
   e) Annahmen, die eine vollständige Interpretation von UML (für die bislang betrachteten Konzepte)
5. UML-SDL-Tool (PragmaDev)

Model Browser





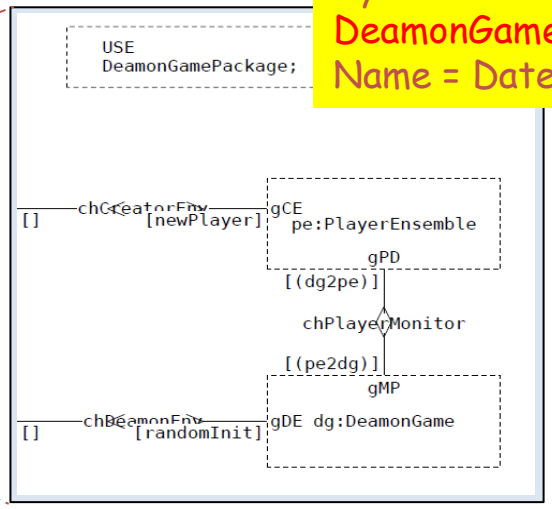
Blocktyp DeamonGame
Name = Dateiname



Blocktyp PlayerEnsemble
Name = Dateiname

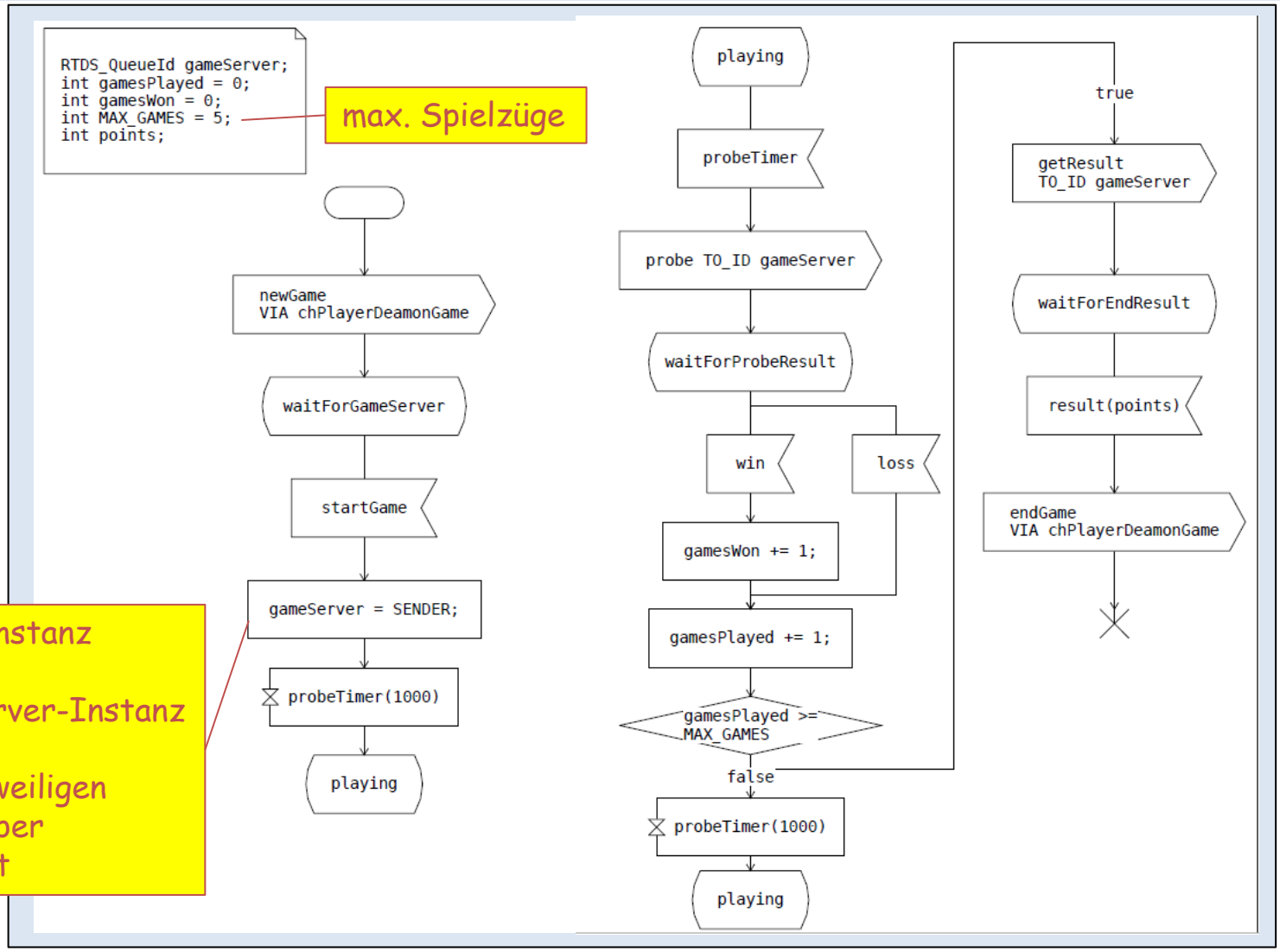
```
bool deamonIsEven = true;
```

C/C++ Erweiterung
globale Variable

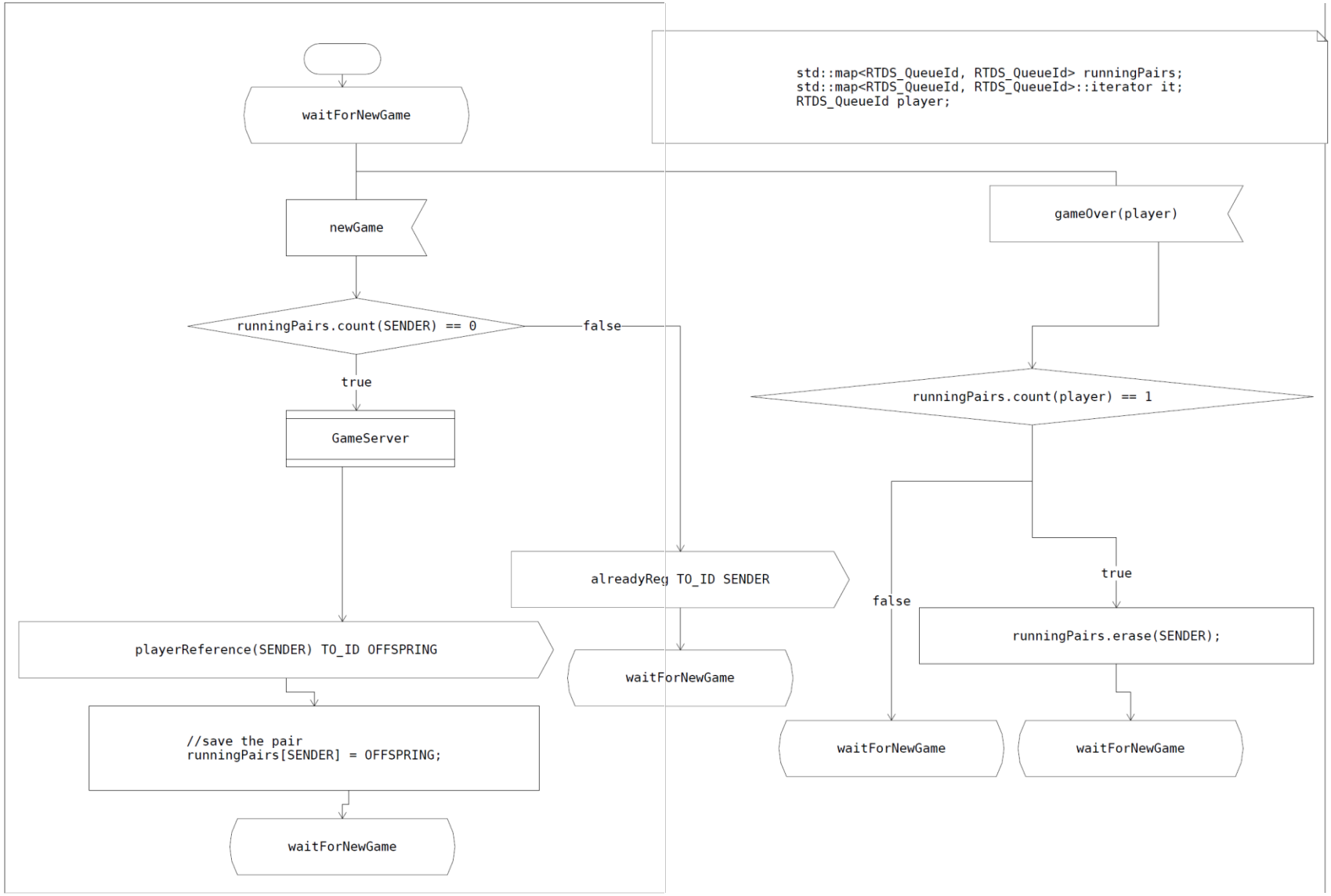


System
DeamonGameSystem
Name = Dateiname

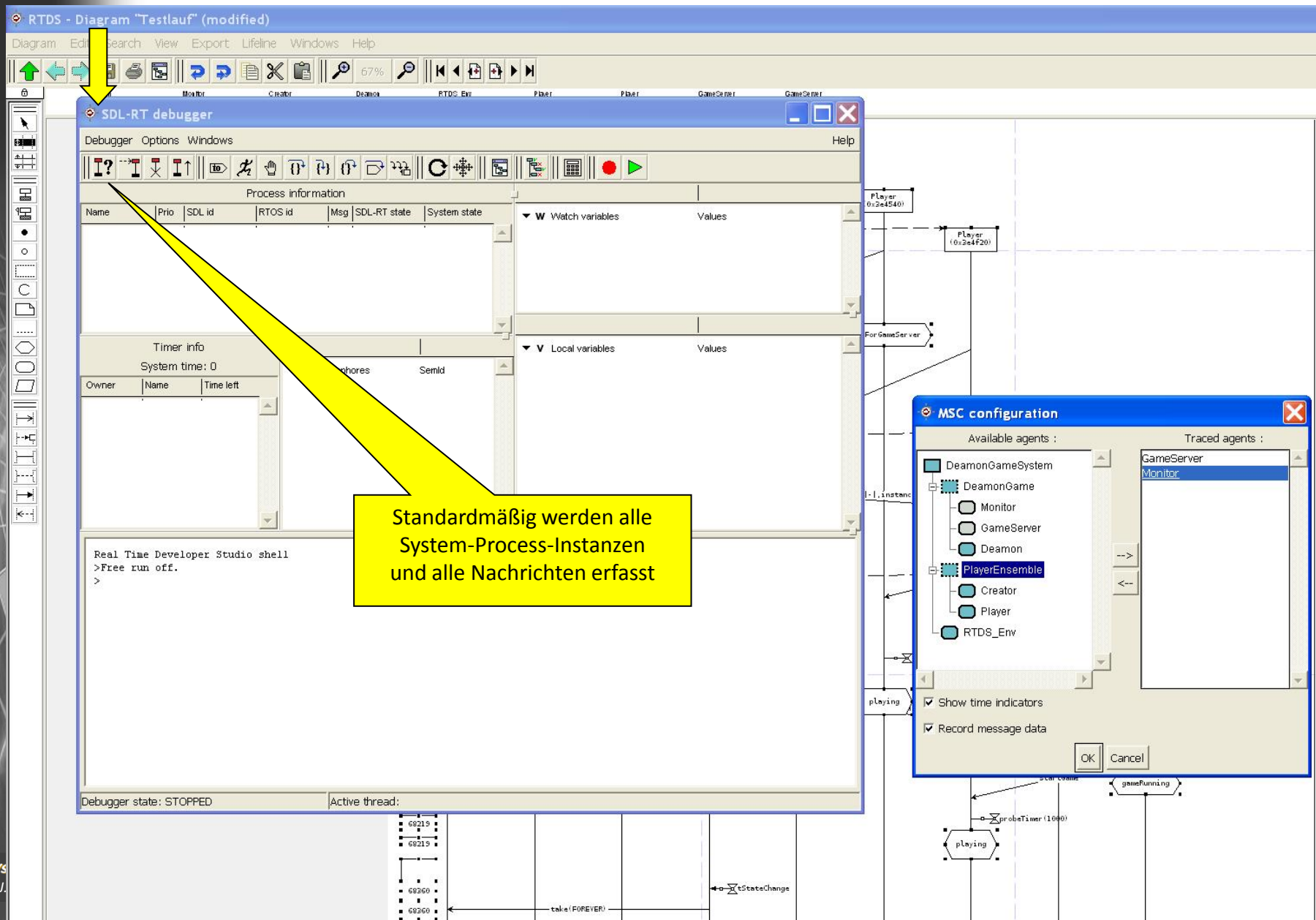
Prototypinstanz der Prozessmenge Player



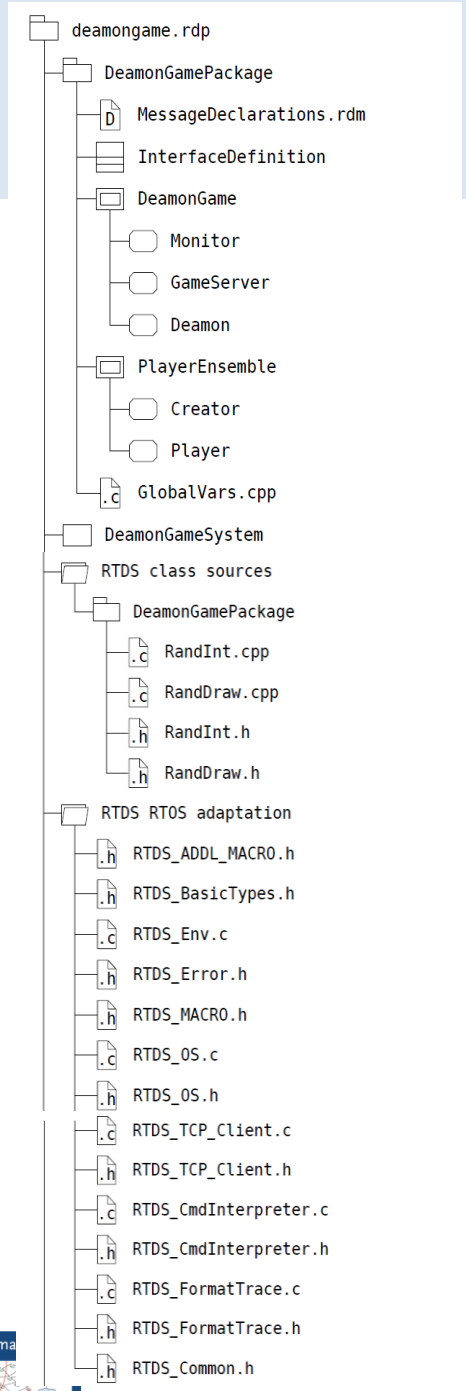
Block(typ)-Komponente: Monitor



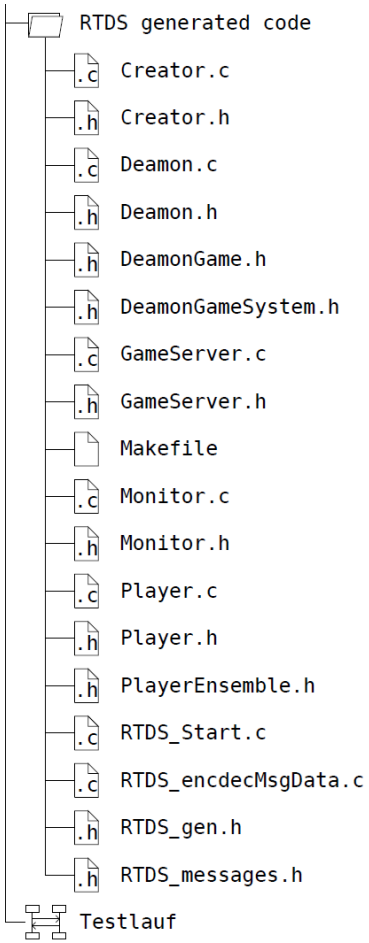
Konfiguration der Sequenzdiagramm-Ausgabe (MSC)



DeamonGame-Simulator



Quellen



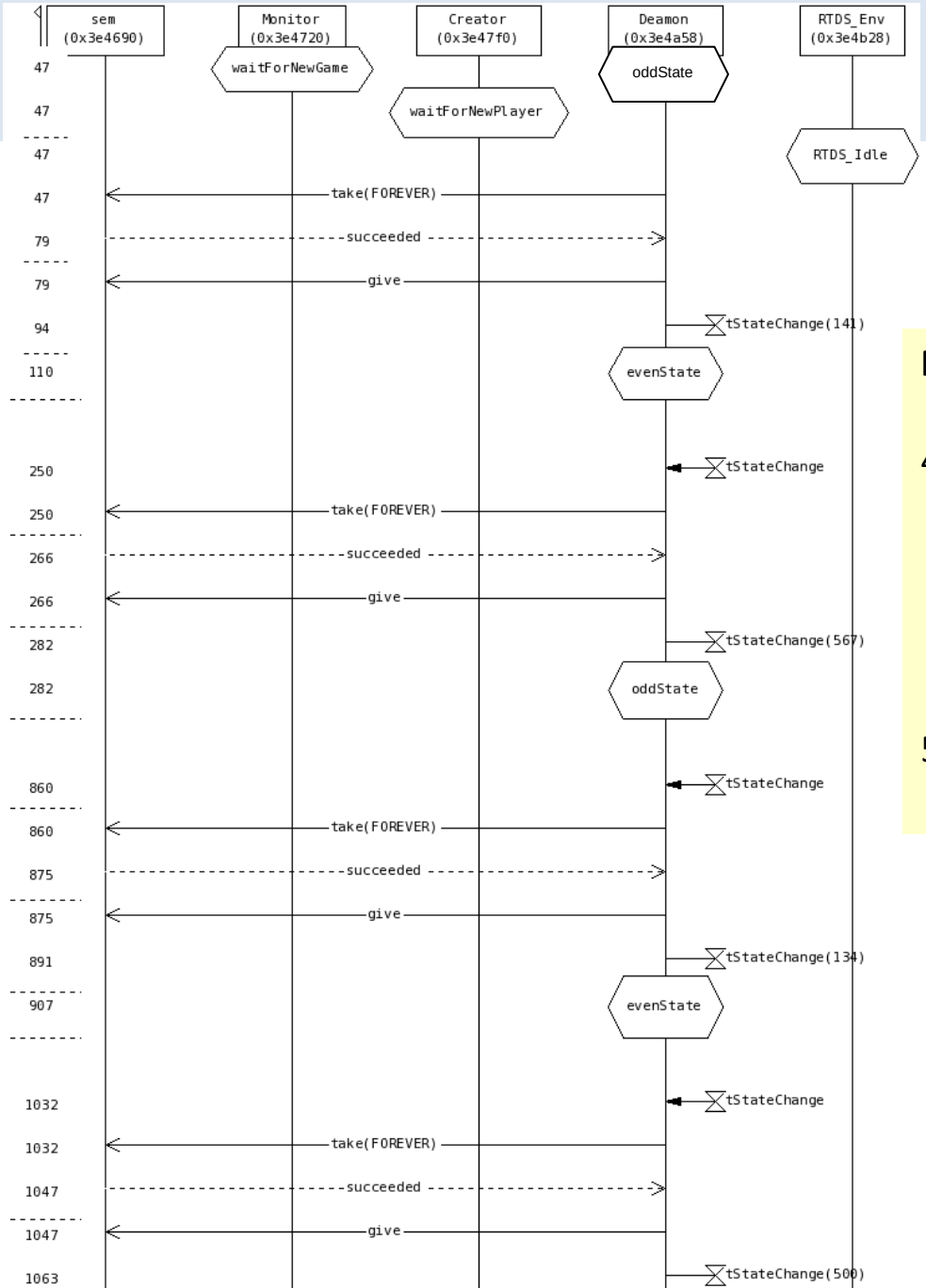
Run

1. Erzeugung des Simulators
   → neue Bedienoberfläche

2. Vorbereitung der Simulatorausgabe
   (SD-Ausgabe:Konfiguration)

3. Start des Simulators
   - alle statischen Prozessinstanzen werden erzeugt (Monitor,Creator,Deamon,Semaphore)
   - arbeiten Starttransitionen ab
   - verharren in Zuständen, wo sie auf Eingabenachrichten warten (bis auf Deamon)

4. Unterbrechung mit Pause



Noch keine Player- und GameServer-Instanzen

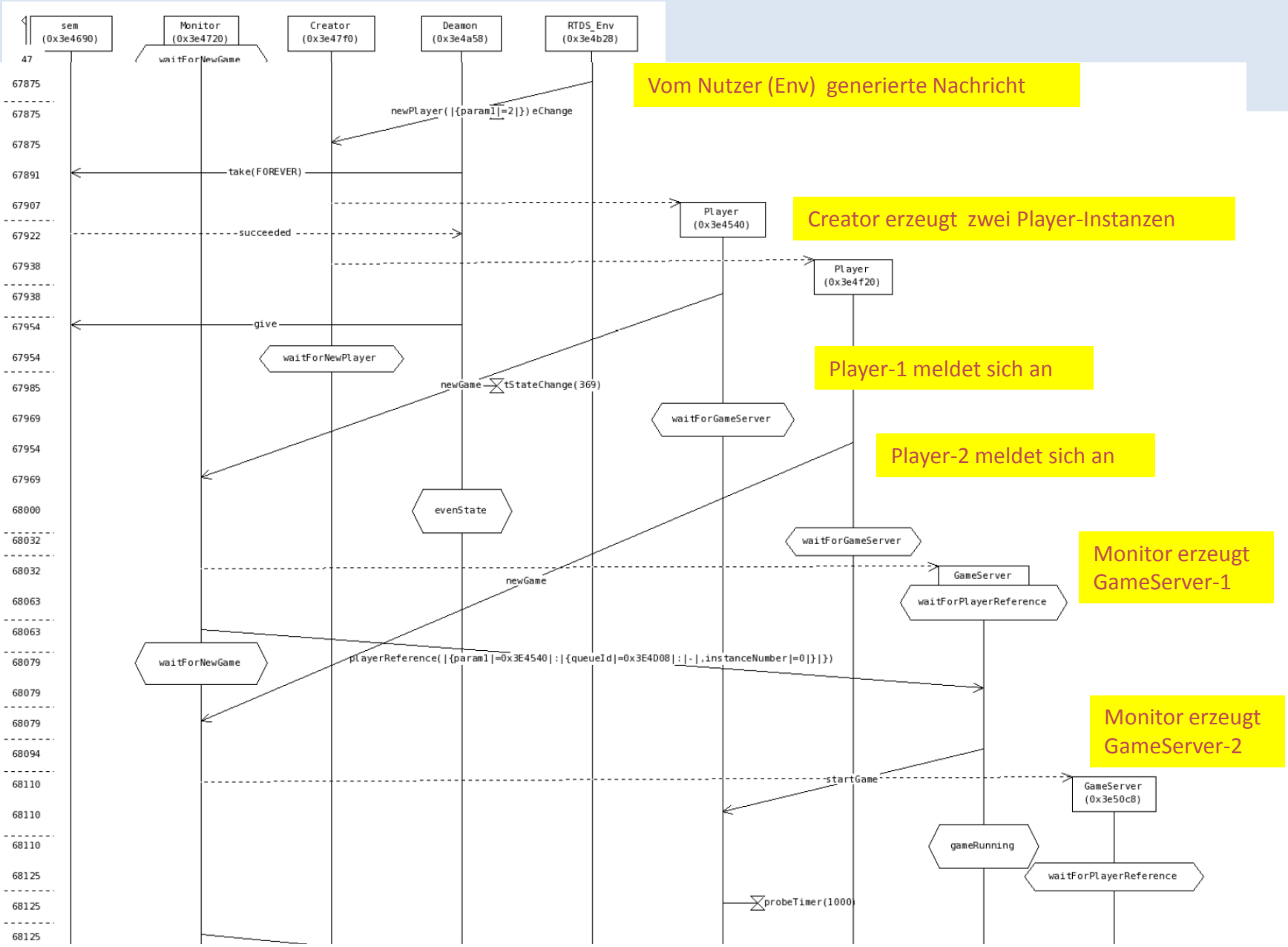
Monitor und Creator verharren

Deamon wechselt stochastisch seinen Grundzustand

Run (Fortsetzung)

4. Erzeugung einer Nachricht newPlayer(2) mit Creator als Empfänger-Prozessinstanz
   → 2 Spieler nehmen damit am Spiel teil

5. Unterbrechung mit Pause


sem (0x3e4690)
Monitor (0x3e4720)
Creator (0x3e47f0)
Deamon (0x3e4a58)
RTDS_Env (0x3e4b28)
waitForNewGame
newPlayer(|{param1|=2|})eChange
take(FOREVER)
Player (0x3e4540)
succeeded
Player (0x3e4f20)
give
waitForNewPlayer
newGame
tStateChange(369)
waitForGameServer
evenState
waitForGameServer
newGame
GameServer
waitForPlayerReference
waitForNewGame
playerReference(|{param1|=0x3E4540|:|{queueId|=0x3E4D08|:|-|,instanceNumber|=0|}|})
startGame
GameServer (0x3e50c8)
gameRunning
waitForPlayerReference
probeTimer(1000)
Vom Nutzer (Env) generierte Nachricht
Creator erzeugt zwei Player-Instanzen
Player-1 meldet sich an
Player-2 meldet sich an
Monitor erzeugt GameServer-1
Monitor erzeugt GameServer-2