Software Engineering II

8 *Frameworks*

SS 2020

Prof. Dr. Dirk Müller

Übersicht

- Einführung
- Begriff und Arten
- Evolution
- Umkehrung der Kontrolle
- Fallstudie: Diskrete Simulation
  - *Whitebox*-Framework
  - Einführung von *Blackbox*-Komponenten
  - weitere Flexibilisierung
- Typen von *Frameworks*
- Entwurfsregeln
- Zusammenfassung

Begriff

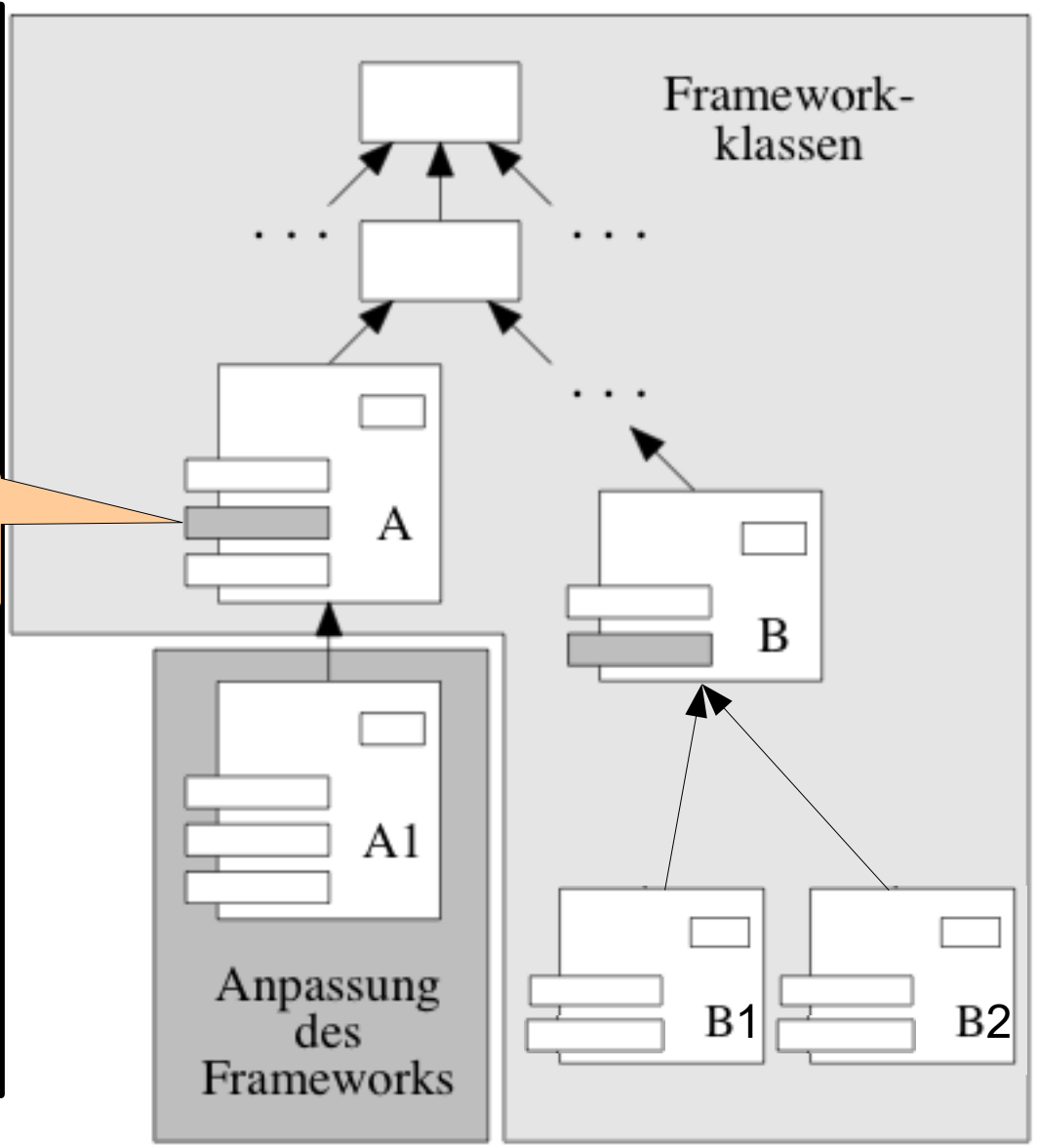
- *Framework*
  - Sammlung individueller Komponenten mit einem definierten Kooperationsverhalten zur Lösung einer Aufgabe
  - „Eine unvollständige Softwarearchitektur, die an spezielle Anforderungen angepa[ss]t werden kann.“ [Pre1997], S. 106
- Def.: integrierte Menge von Software-Artefakten (wie z. B. Klassen, Objekte, Komponenten), die zusammenarbeiten, um eine wiederverwendbare Architektur für eine Familie von verwandten Anwendungen bereitzustellen [1], S. 443, engl.
- Def.: „*A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes.*“ [2]
- als Modebegriff manchmal missbräuchlich für Klassenbibliotheken verwendet

Abgrenzung und Arten

- *Framework* = Klassenbibliothek + Architektur
  - *Frameworks* geben im Unterschied zu gewöhnlichen Klassenbibliotheken die Architektur eines Systems vor, beschreiben also auch die Interaktionen zwischen Komponenten.
  - Kontrollfluss der Anwendung und Schnittstellen der Klassen vorgegeben
- *Whitebox*-Framework: Anpassung durch Subklassenbildung und Compilierung, vererbungsbasiert
  - entspricht ungefähr dem Konzept globaler Variablen
- *Blackbox*-Framework: Anpassung zur Laufzeit durch unterschiedliche Instanziierung, komponentenbasiert
  - entspricht dem Ansatz des Sendens von Nachrichten zwischen Komponenten
  - Protokoll für die Kommunikation vorgegeben

Whitebox-Framework

- Programmierer muss Implementierung des *Frameworks* zu einem gewissen Grade kennen

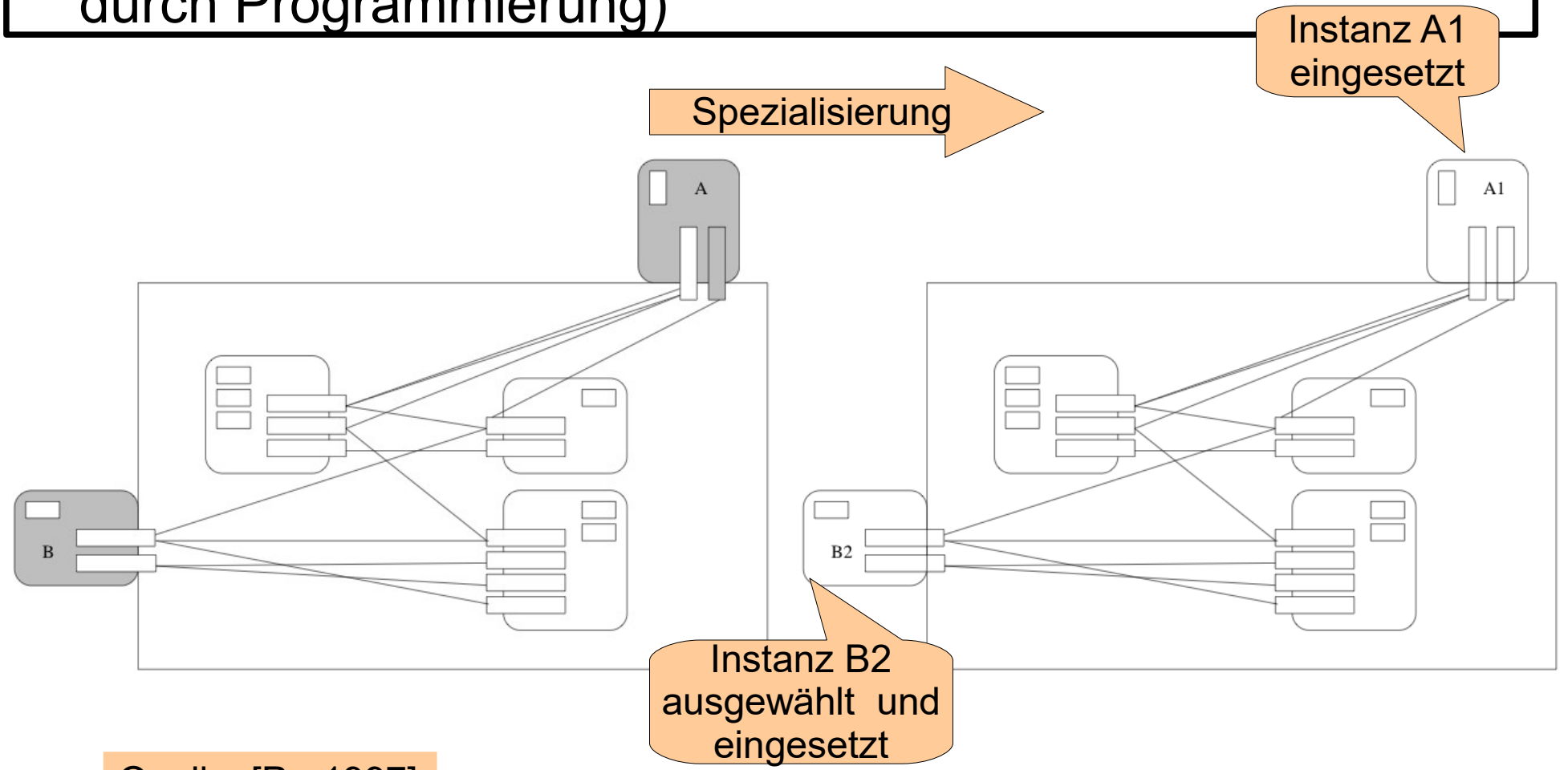


abstrakte Einschubmethode in der Klasse A

Quelle: [Pre1997]

Blackbox-Framework

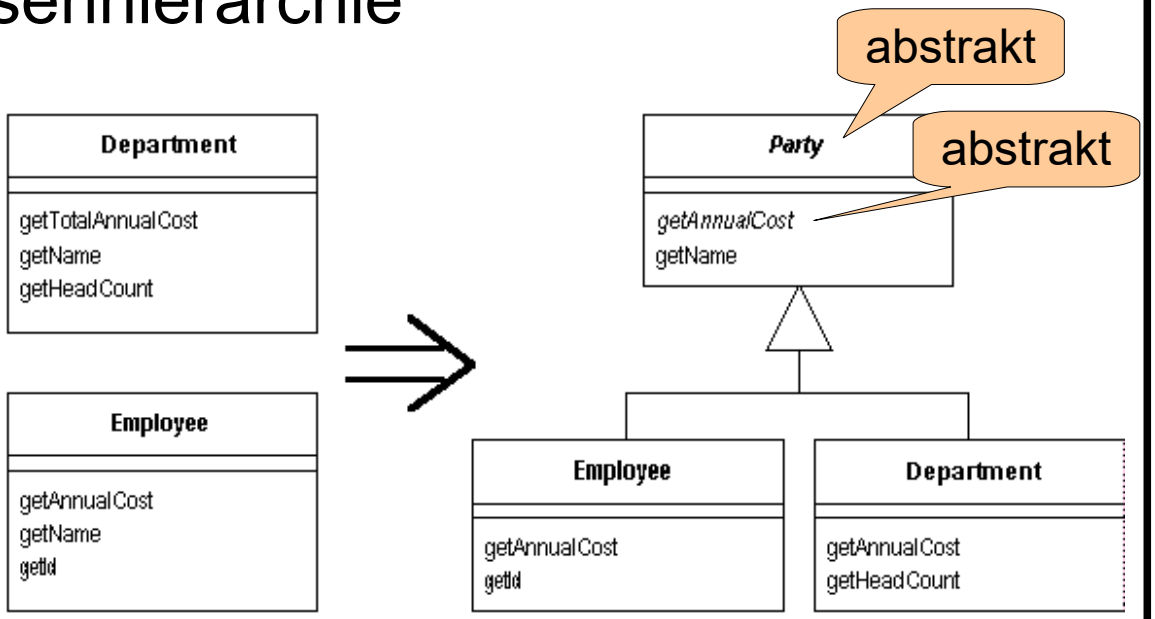
- Es existiert eine Anzahl fertiger Komponenten.
- Änderungen durch Komposition der Komponenten (nicht durch Programmierung)



Quelle: [Pre1997]

Evolution von *Frameworks*

- in der Praxis meist Mischformen aus *Whitebox*- und *Blackbox*-Frameworks
- Entwicklung vom *Whitebox*- zum *Blackbox*-Framework mit zunehmenden Reifegrad als Idealfall
  - aber: praktisch manchmal Einstellung der Entwicklung unterwegs
- Ausgangspunkt: Klassenhierarchie
- typisches *Refactoring*
  - *Extract Superclass*

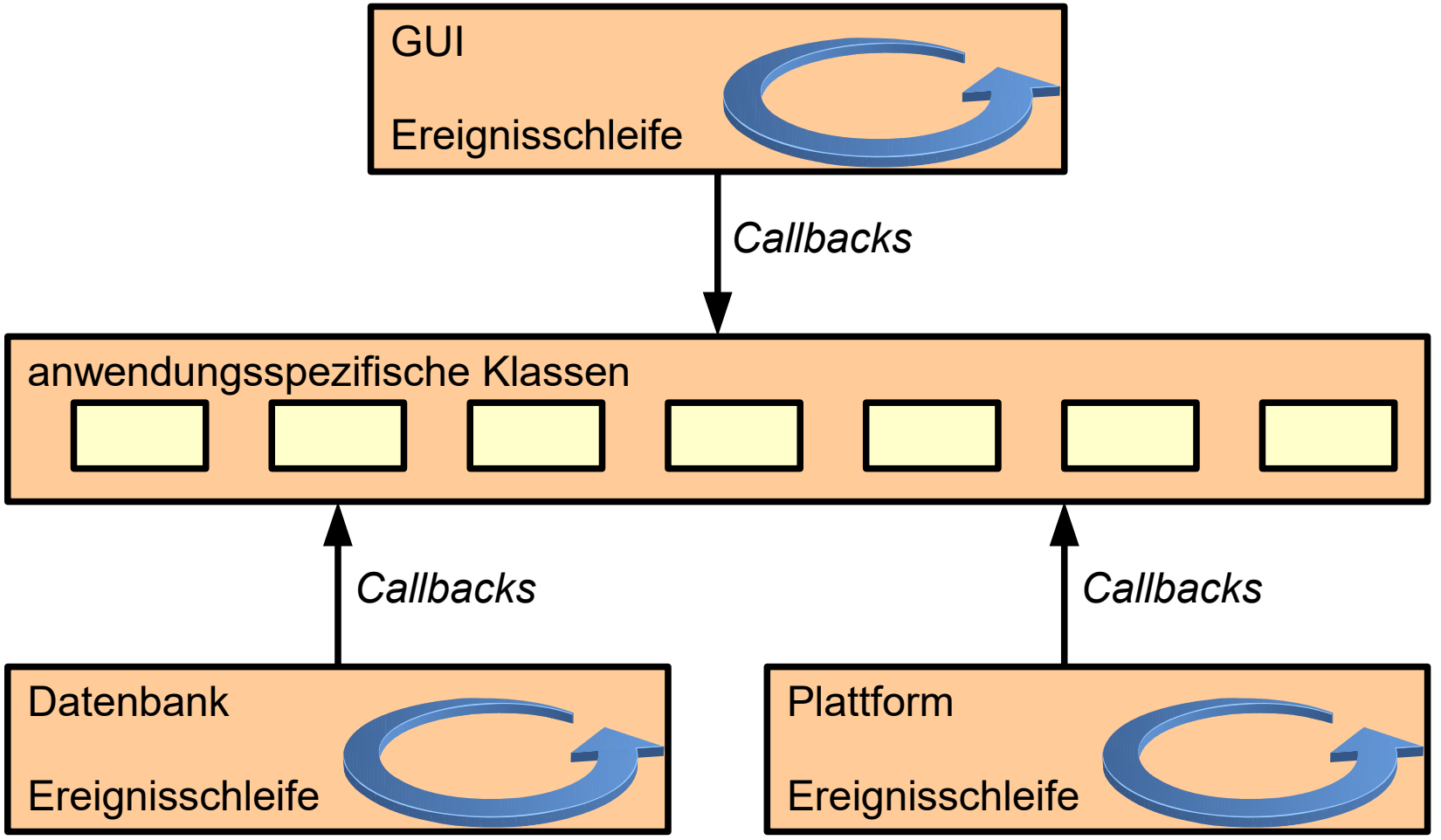


Quellen: [Pre1997][2]

Hotspots und *Frozen Spots*

- *Hotspot*: variabler (flexibler) Anteil
- *Frozen Spot*: Standardanteil
- nicht maximaler, sondern angemessener Grad an Flexibilität als Ziel
- Evolution: schrittweise Überführung von *Hotspots* in *Frozen Spots*
- Identifikation geeigneter *Hotspots* als zentrale Entwurfsaufgabe für *Frameworks*

Umkehrung der Kontrolle in *Frameworks*



Registrieren der konkreten Klassen nicht fest im Programmcode, sondern von außen durch Konfiguration: *Dependency Injection*

Quelle: [1], S. 445

Fallstudie: Diskrete Simulation

- zwei Anwendungen, die auf dem *Framework* „Diskrete Simulation“ basieren
  - Simulation einer Kasse in einem Supermarkt
  - Simulation eines Parkhauses
- zentrale Elemente: Ereignisse mit Zeitstempeln
  - Supermarktkasse: Eintreffen eines Kunden
- Zeitstempel der Ereignisse durch Zufallsgenerator erzeugt
- Simulation läuft mit der Zeit und sammelt statistische Daten, die dann ausgewertet werden können

Quelle: [Pre1997]

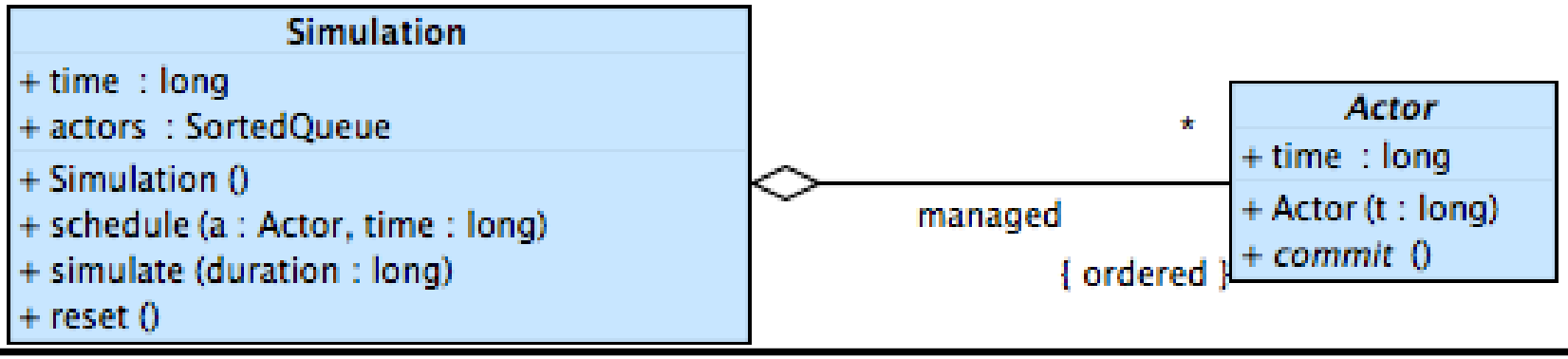
Actor

- Akteur als Abstraktion eines kausalen Paares (Ereignis, Aktion)
- Zeitpunkt des Ereignisses in `time`
- `commit()` kapselt die Funktionalität (Aktion) des Akteurs
- Simulation ruft `commit()` auf, wenn Eintrittszeit erreicht

Actor
time: long
Actor(t: long) *commit()*

Quelle: [Pre1997]

Simulation von *Actors*



- `simulate`
  - iteriert über `actors`
  - aktueller Zeitpunkt wird auf Zeitpunkt des *Actors* eingestellt
  - `commit()` des *Actors* wird aufgerufen
  - Simulation endet, wenn Simulationszeit `duration` abgelaufen
- `schedule`
  - fügt *Actor* entsprechend des Eintrittszeitpunkts seines Ereignisses in die Warteschlange `SortedQueue` ein
- `reset`
  - leert `SortedQueue` und setzt Simulationszeit zurück (`time`:=0)

Quelle: [Pre1997]

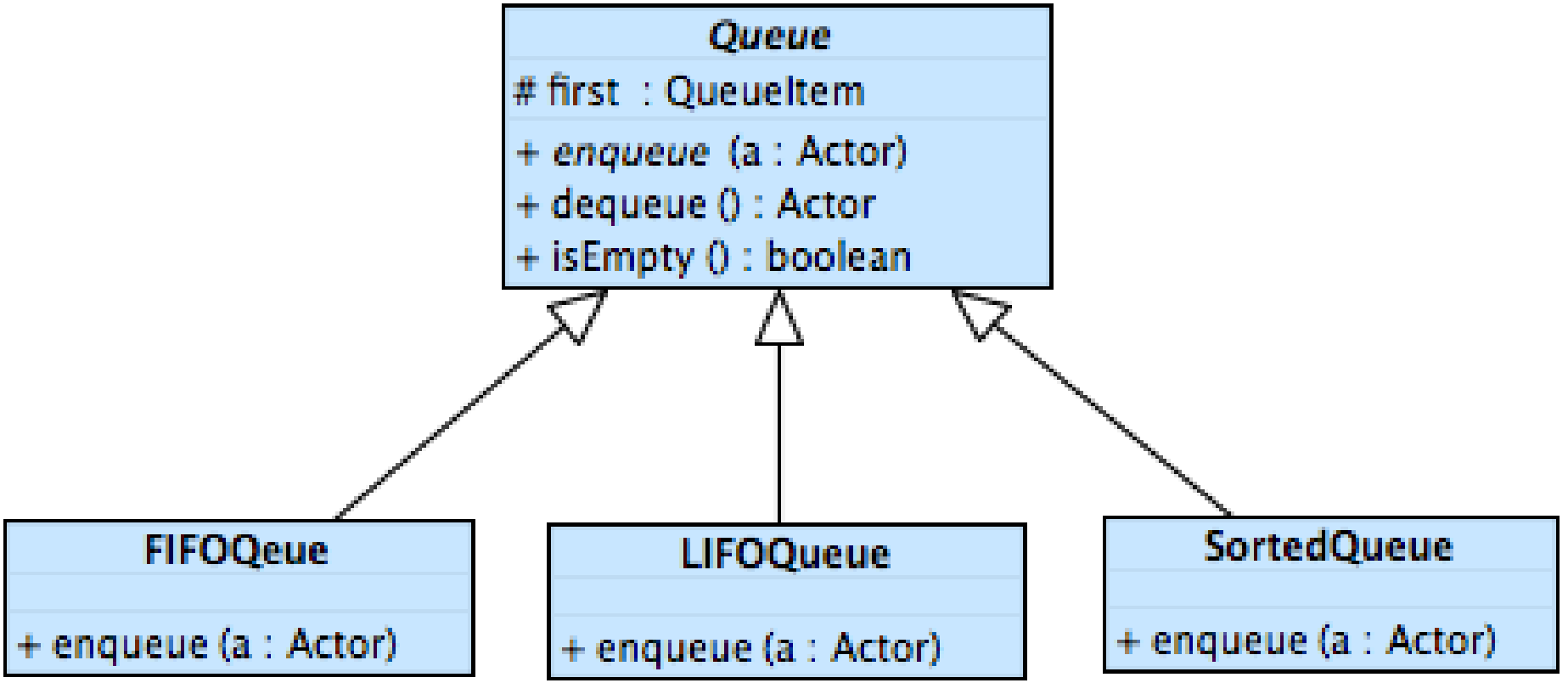
Simulation.simulate(long duration)

- generische Simulations-schleife
- Instanzen der Klasse `Simulation` können ohne Neuübersetzung mit jeder beliebigen Instanz einer Unterklasse von `Actor` interagieren

```
public class Simulation{

...

    public void simulate(long duration){
        Actor actor;
        long endOfSimulation = time + duration;
        do{
            if (!actors.isEmpty()){
                actor = actors.dequeue();
                time = actor.time;
                actor.commit();
            }
            else // no more actors enqueued
                time = endOfSimulation + 1;
                // exit loop
        } while (time <= endOfSimulation);
    }

...
```

Quelle: [Pre1997]

Warteschlangen-Klassen



Quelle: [Pre1997]

Methode schedule

```
public void schedule(Actor a, long t){
        a.time = t;
        actors.enqueue(a);
}
```

- Klasse `Simulation` realisiert abstrakte Simulation auf Basis des Protokolls der Klasse `Actor`
- derzeit *Whitebox*-Framework im frühen Stadium der Entwicklung

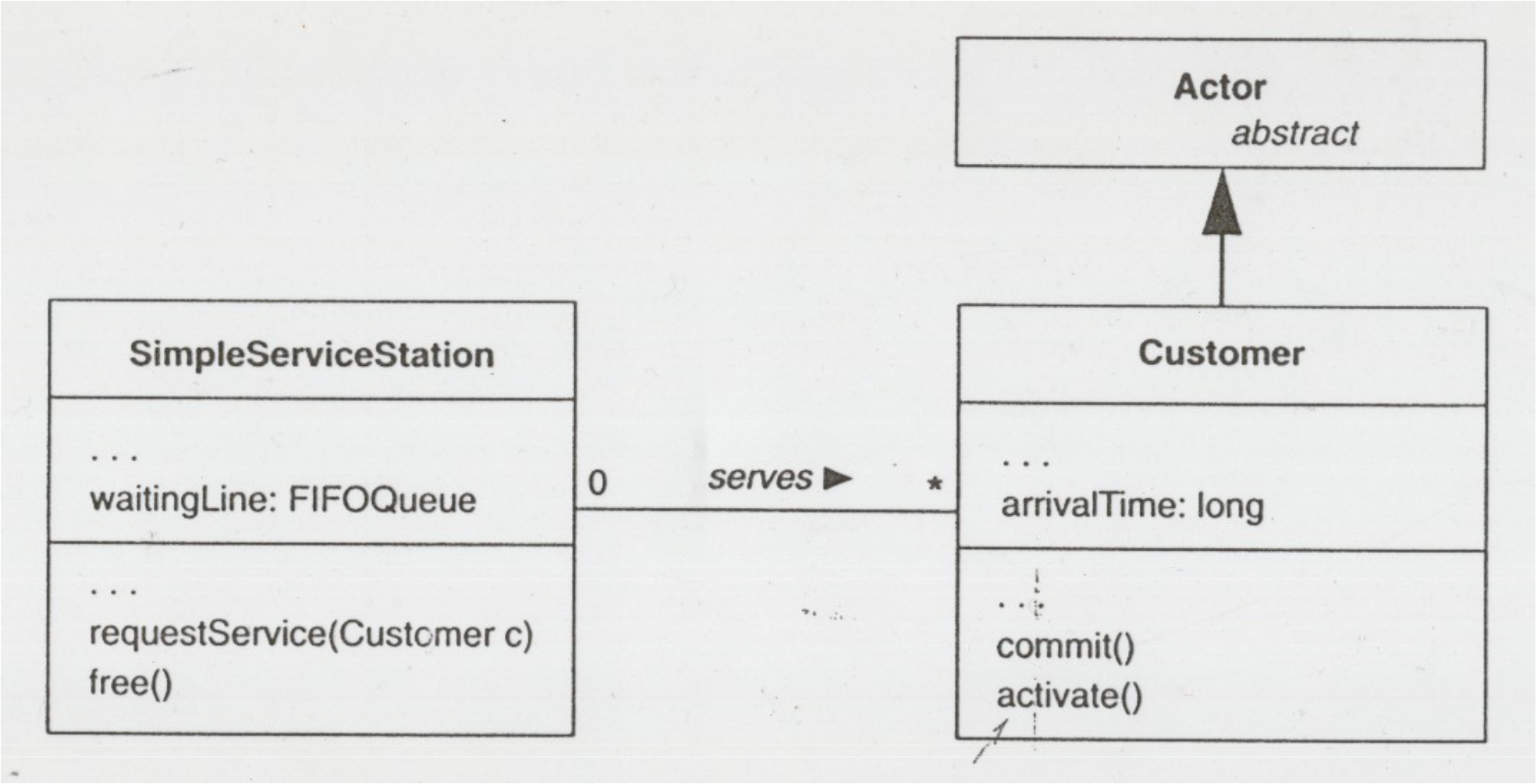
Quelle: [Pre1997]

Simulation einer Kasse in einem Supermarkt

- Registrierkasse im Supermarkt (*convenience store*)
- Kunden `Customer`
- Kundengenerator `CustomerGenerator`
- Kasse `SimpleServiceStation`
- Ziel: Übergang vom *Whitebox*- zum *Blackbox*-Framework oder zumindest Bewegung in diese Richtung

Quelle: [Pre1997]

Modell der Bedienung von Kunden



Quelle: [Pre1997]

Entwurf einer einfachen Bedienstation

```
public class SimpleServiceStation {
        ...
        public void requestService(Customer c){
                if (Warteschlange ist leer )
                        c.activate()
                        // Bediener verarbeitet die Kundenanfrage
                else
                        waitingLine.enqueue(c);
                        // Erstellen von Statistiken über die Warteschlange
                ...
        }
        public void free(){
                Actor actor;
                // Erstellen von Statistiken über die Warteschlange
                ...
                if (Warteschlange enthält Kunden )
                        actor = waitingLine.dequeue();
                        if (actor instanceof Customer)
                                ((Customer)actor).activate();
                        }
        }
        ...
}
```

Quelle: [Pre1997]

Implementierung (1/3)

```
public class SimpleServiceStation{
  protected FIFOQueue waitingLine;

      ...
  public void requestService(Customer c){
    if (path.n == 0)   // number of queued customers
      c.activate();
    else
      waitingLine.enqueue(c);
    path.up(sim.time);
  }

      ...
  public void free(){
    Actor actor;
    path.down(sim.time);
    if (path.n > 0){
      actor = waitingLine.dequeue();
      if (actor instanceof Customer)
        ((Customer)actor).activate();
    }
  }

      ...
```

Quelle: [Pre1997]

Implementierung (2/3)

```
public class CustomerGenerator extends Actor{
     SimpleServiceStation station;
     Simulation simulation;


        . . .

 public void commit(){
    Customer c = new Customer(0,station,simulation);
      // time is scheduled by the custmer's activate() method,
      // invoked by a SimpleServiceStation instance



      station.custNo++;
      station.requestService(c);
      simulation.schedule(
                new CustomerGenerator(0,station,simulation),
                simulation.time + (long)(RandomNumbers.negExp(station.arrivalRate)));
  }
}
```

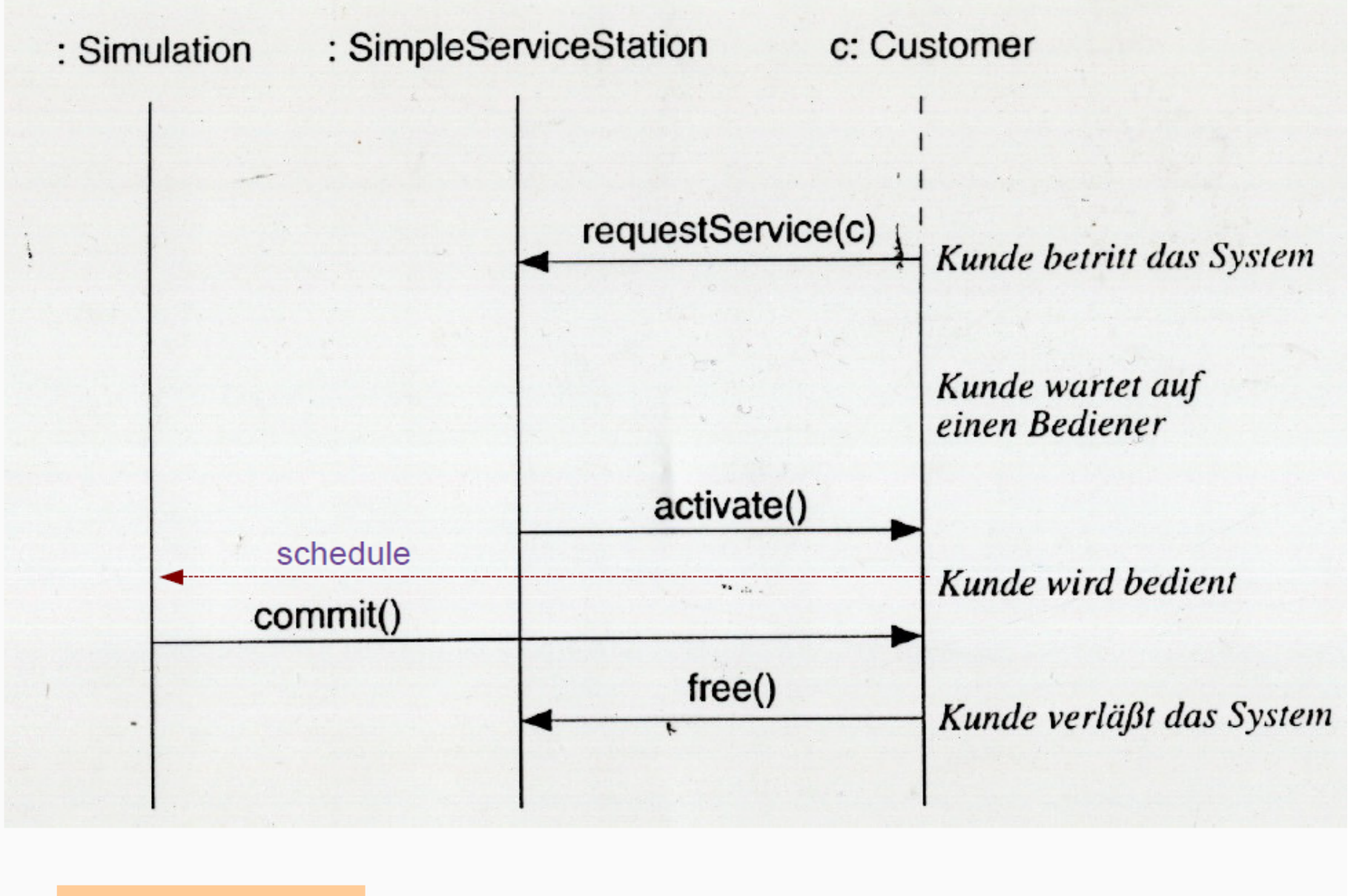
Quelle: [Pre1997]

Implementierung (3/3)

```
public class Customer extends Actor{
     SimpleServiceStation station;
     Simulation simulation;
     long arrivalTime;

       ...

     public void commit(){
          station.free();
          if (station.traceOn)
             station.traceOutput.appendText(
             "\ncustomer #"+ no +"left system at:"
             + simulation.time);
          station.waitingTimeSeq.add(simulation.time - arrivalTime);
     }

       ...

     public void activate(){
          int myServiceDuration;
          simulation.schedule(this, simulation.time +
              (myServiceDuration =
                 (int)(RandomNumbers.negExp(
                              station.avrgServiceDuration))));
```

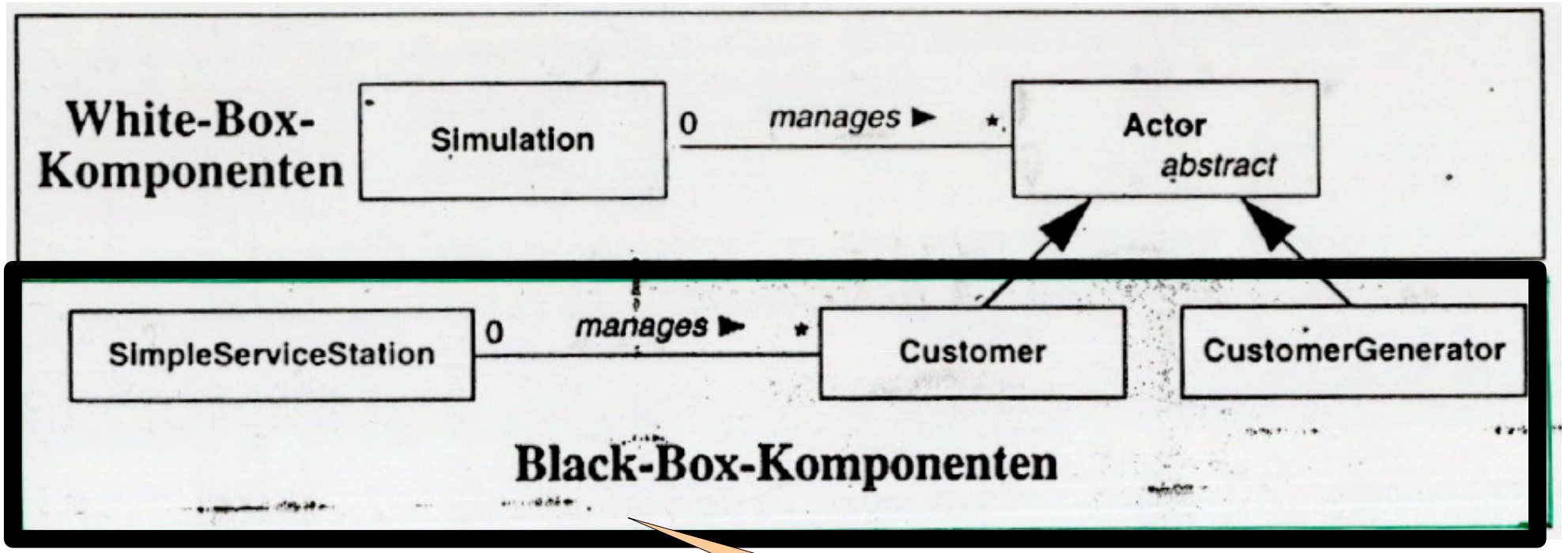
Quelle: [Pre1997]

Sequenzdiagramm



Quelle: [Pre1997]

Ergebnis: *Whitebox*-Framework mit *Blackbox*-Komponenten



enthält keine abstrakten Klassen

Quelle: [Pre1997]

Komposition einer Simulation Supermarktkasse

```
public class ConvenienceStoreSim {
      ...
      public void simulate(int avrgServiceDuration,
                           int arrivalRate, int duration){
            SimpleServiceStation station;
            Simulation simulation;
            CustomerGenerator generator;

            Simulation = new Simulation();
            Station = new SimpleServiceStation(simulation,
                                               arrivalRate,
                                    AvrgServiceDuration, …)
            generator = new
                        CustomerGenerator(0,station,simulation);
            simulation.schedule(generator,0);
                        // mit Erzeugung von Kunden beginnen

            simulation.simulate(duration);

            station.provideStatistics();
      }
      ...
}
```

Quelle: [Pre1997]

Entwurfsüberlegungen

- sehr einfach zu nutzen, aber nicht flexibel genug
  - Bedienstation mit mehreren Bedienern?
  - Ausfall eines Bedieners?
- Mittel, um *Frameworks* flexibler gestalten zu können
  - Schablonenmethoden und Einschubmethoden
  - Entkopplung der Klassen `CustomerGenerator` und `Customer`
  - Flexibilität durch Komposition
  - Skalierbarkeit von Einschubmethoden

Quelle: [Pre1997]

Platzierung von Einschubmethoden

- Schablonenmethoden (engl. *template methods*) rufen Einschubmethoden (engl. *hook methods*) auf
- Einschubmethode ist zunächst leer, kann aber vom Wiederverwender der Klasse überschrieben werden

```
public void simulate(long duration){
      long endOfSimulation = time + duration;
      do{
         if (!actors.isEmpty()){ ...
            notificationHook();                }
         else // no more actors enqueued
            time = endOfSimulation + 1;
      } while (time <= endOfSimulation);
  }

public void notificationHook(){

}
```

Schablonenmethode

potenziell überschriebene Einschubmethode, verändert ihre Schablonenmethode

Quelle: [Pre1997]

Entkopplung CustomerGenerator – Customer

- Verhalten durch Definition von Unterklassen spezifiziert
- Einschub-methode `makeActor` wird dann darauf angepasst

Originalentwurf:

```
public class CustomerGenerator extends Actor {
        ...
        public void commit(){
                Customer c= new Customer( … );
                station.requestService(c);
                ...
        }
        ...
}
```

Entwurf mit Einschubmethode:

```
public class ActorGenerator extends Actor {
        ...
        public void commit(){
                Actor a= makeActor();
                station.requestService(a);
                ...
        }
        public void makeActor(){
                Return new Customer( … );
        }
        ...
}
```

Schablonenmethode

Einschubmethode

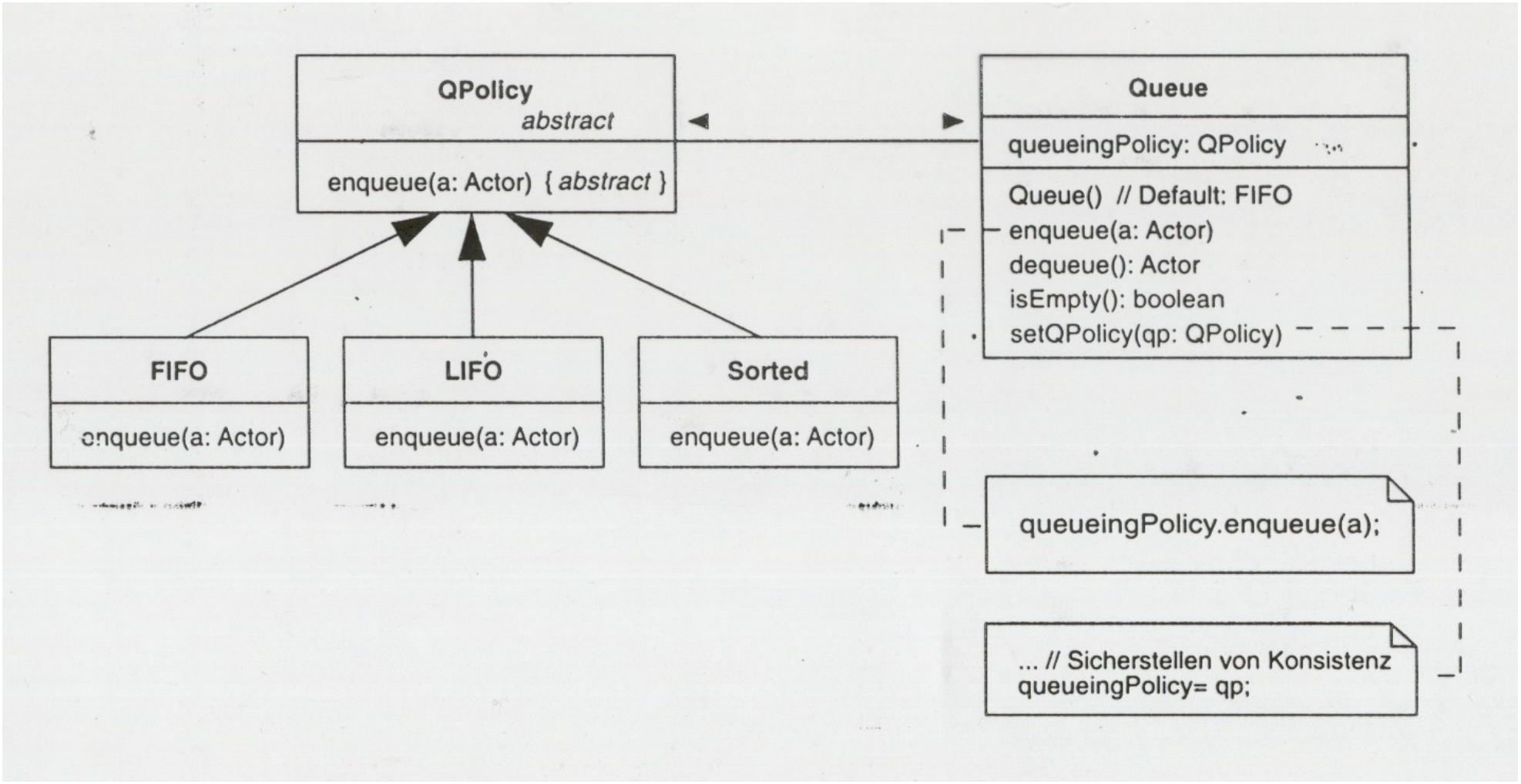
Quelle: [Pre1997]

Flexibilität durch Komposition

- Schablonen- und zugehörige Einschubmethode
  - liefern einen einfach zu nutzenden Erweiterungspunkt
  - befinden sich jeweils in ein und derselben Klasse
    => Verhalten kann nicht zur Laufzeit verändert werden
- Einschubmethoden erlauben flexible Anpassungen zur Laufzeit, wenn ihre Klasse mit der Klasse abstrakt gekoppelt ist, die die zur Einschubmethode gehörende Schablonenmethode enthält
- Beispiel: Im Originalansatz wurde die abstrakte Methode `enqueue()` der Klasse `Queue` in Unterklassen überschrieben, die auf diese Weise verschiedene Warteschlangen realisieren.
  - Schablonenmethode `enqueue()` in `Queue`
  - Auswahl wird einem `QPolicy`-Objekt übertragen

Quelle: [Pre1997]

Warteschlange mit abstrakt gekoppelter Warteschlangenstrategie



Quelle: [Pre1997]

Umschalten der Strategie zur Laufzeit

- Beim Umschalten der `QPolicy`-Instanz muss explizit für Konsistenz gesorgt werden!
- Wird z. B. von `FIFOQueue` auf `SortedQueue` umgeschaltet, müssen Objekte entsprechend ihrer Zeitattribute neu geordnet werden und die ursprüngliche Ordnung muss gespeichert werden, um sie ggf. wieder herstellen zu können.
- einfachste Art der Konsistenzerhaltung: Warteschlange zuerst vollständig leeren, d. h. abarbeiten oder verwerfen

Quelle: [Pre1997]

Skalierbarkeit von Einschubmethoden

- Begriffe Schablonen- und Einschubmethode sind nicht absolut, sondern relativ. Es sind also Rollen.

Schablonenmethode	**Einschubmethode**
`ActorGenerator.commit()`	`ActorGenerator.makeActor()`
`Simulation.simulate()`	`ActorGenerator.commit()`

- Idealfall: Programmierer muss nur die elementarste Einschubmethode überschreiben, um Verhalten der alles umfassenden Schablonenmethode zu beeinflussen
  - im Beispiel: nur `ActorGenerator.makeActor()` zu überschreiben

Quelle: [Pre1997]

Muster für die Kombination von Einschubmethoden

- H : *Hook-Klasse*, die die Einschubmethode enthält
- T : *Template*-Klasse, die die Schablonenmethode enthält
- TH : Vereinigungsmuster, Schablonen- und Einschubmethode in derselben Klasse
- T-H : Trennungsmuster, abstrakte Kopplung voneinander getrennter Klassen
- Das Verhalten von T-Objekten kann durch das Einsetzen unterschiedlicher H-Objekte geändert werden.
- Ein T-Objekt verweist auf ein H-Objekt, somit ist das Senden von Nachrichten notwendig.

(Ende des Fallbeispiels)

Quelle: [Pre1997]

Implementierung und Nutzung von *Frameworks*

- Sammlung konkreter + abstrakter interagierender Klassen
- in einer objektorientierten Standard-Programmiersprache
  - z. B. *Java, C#, C++*
- in einer dynamisch typisierten Sprache
  - z. B. *Ruby, Python*
- Ein *Framework* kann andere *Frameworks* mit einbeziehen.
  - dann jedes einbezogene *Framework* für Entwicklung eines Teils der Anwendung zuständig
- Nutzung für eine vollständige Anwendung oder nur für einen Teil

Quelle: [1], S. 444

Typen von *Frameworks*

- *Frameworks* für *Web*-Anwendungen
  - dynamische Seiten, Sicherheit, Datenbankintegration
  - *Session*-Management
  - Nutzerinteraktion, z. B. über *AJAX* oder *HTML5*
- System-Infrastruktur-*Frameworks*
  - Kommunikation, Nutzerschnittstellen, Compiler, Testen
- *Middleware*-Integrations-Frameworks
  - standardisierte komponentenbasierte Kommunikation und Datenaustausch
  - z. B. *Microsoft .NET* und *Enterprise Java Beans (EJB)*
- *Frameworks* für Geschäftsanwendungen
  - für spezielle Anwendungsdomäne
  - z. B. für Telekommunikation oder Finanzwesen
  - mehr und mehr von Softwareproduktlinien verdrängt

Quelle: [1], S. 445

Entwurfsregeln für *Frameworks*

- Aufspalten großer Klassen
  - 50-100 Methoden per Klasse sind zu viel
  - starkes Indiz, dass mehrere Abstraktionen in selber Klasse
- Methoden, die auf verschiedene Arten in Unterklassen implementiert werden, in eigene Komponenten separieren
- Methoden, die nicht miteinander kommunizieren, sollten in verschiedenen Klassen separiert werden
  - z. B. Bild als Menge von geometrischen Objekten und als *Bitmap*
- Nachrichten sollten an Komponenten gesendet werden statt vererbungsbasiert (Übergang *Whitebox* → *Blackbox*)
  - z. B. Elementenvergleichsoperator an Sortiermethode übergeben
- Reduzierung der impliziten Parameterübergabe
  - alles als explizite Parameter statt über globale Variablen oder *Shared Memory* übergeben

Quelle: [2]

Ansätze zum Aufspüren von *Hotspots*

- Untersuchung des laufenden Betriebs
  - Wo ist der größte Aufwand gerechtfertigt, um Änderungen während der Laufzeit durchführen zu können?
  - Kostenexplosion als Indiz für unzureichende Allgemeinheit
- Anwendungsfälle bzw. Szenarien analysieren
  - Anwendungsexperten jeweils zur notwendigen Flexibilität befragen
  - Zusammenfassung ähnlicher Anwendungsfälle
  - Beschreibung der Unterschiede durch *Hotspots*
- Befragung der richtigen Leute
  - Abstraktionsvermögen, mathematisches-logisches Denken
  - am besten: Entwickler des laufenden Systems mit solchen Fähigkeiten mit in neue Entwicklergruppe aufnehmen

Quelle: [Pre1997], S. 83 f.

Zusammenfassung

- *Frameworks* als ein sehr effektiver Ansatz zur Wiederverwendung
- aber: teuer in der Integration in den Softwareentwicklungsprozess
- Evolution von *Whitebox*- zu *Blackbox*-Frameworks
  - von Vererbung zur Compilezeit zu Komposition zur Laufzeit
- Erlernen der effizienten Nutzung kann Monate dauern
- Beurteilung und Auswahl geeigneter *Frameworks* kann schwierig sein
- *Debugging* ist schwieriger, da Fehlermeldungen auf dem Nutzer unbekannte *Framework*-Komponenten verweisen können

Literatur

[1] Ian Sommerville: „Software Engineering 10“, Addison-Wesley, 2016

[2] Ralph E. Johnson & Brian Foote: „Designing Reusable Classes“, Journal of Object-Oriented Programming June/July 1988, Volume 1, Number 2, pages 22-35, http://www.laputan.org/drc.html