C340 Concurrency: Dynamic Systems

Wolfgang Emmerich

Outline

- Motivation
- Golf Club Design
- Golf Club FSP Model
- Liveness Analysis in Dynamic Systems
- Java Joins

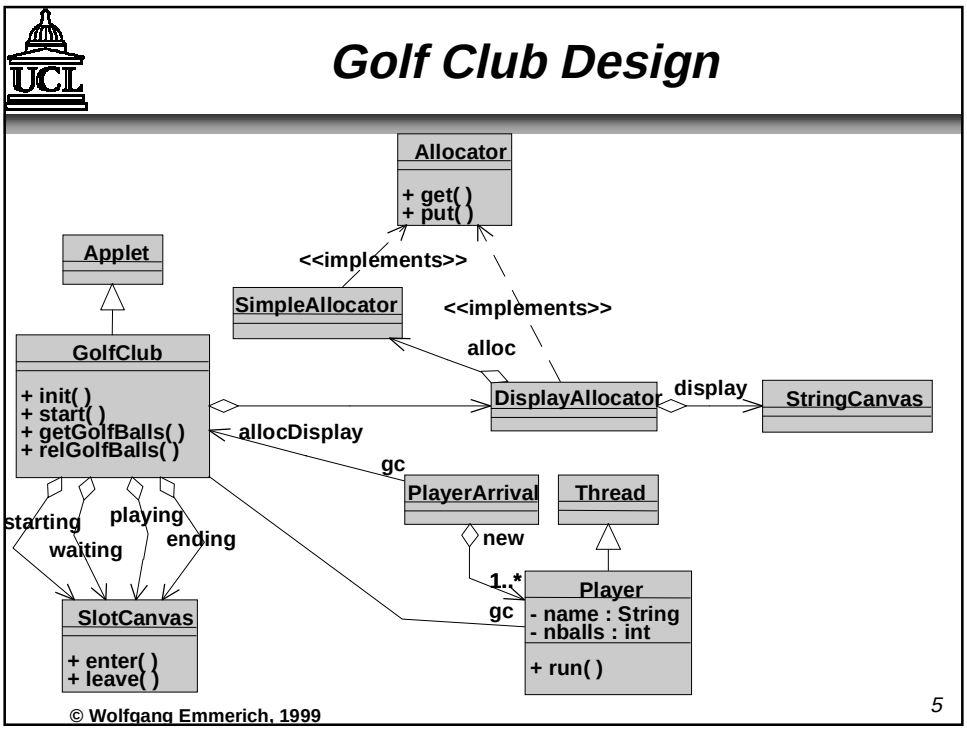
Static versus Dynamic Systems

- *Threads and processes considered so far created during initialization and run until termination*
- *Static organization of thread structure*
- *This lecture: dynamic creation and termination of threads*
- *Example: Operating System Processes*

Example: Golf Club Program

Players hire golf balls from their club and return them after use. The better players tend not to lose balls and only hire a few. Less experienced players hire more balls, so that they have spares during games. They are required to buy replacements for lost balls and return the same number originally hired. The club groundsman decides to treat players as Java threads and to write a monitor to allocate golf balls to players, if available, or to delay the players if insufficient are available.

Demo



Golf Club Design

Class Player

```
class Player extends Thread {
  private GolfClub gc;
  private String name;
  private int nballs;

  Player(GolfCloub g, int n, String s) {
    gc =g; name = s; nballs=n);
  }

  public void run() {
    try {
      gc.getGolfBalls(nballs,name);
      Thread.sleep(gc.playTime);
      gc.relGolfBalls(nballs,name);
    } catch (InterruptedException e){}
  }
}
```

Modelling Dynamic Systems

- *Problem: Finite State Processes demand a fixed number of processes*
- *This is a pre-requisite to enable static analysis of dynamic models*
- *In FSP, however, we can model repetitive, i.e. infinite behaviour*
- *Trick: simulate unknown number of similar terminating processes by modelling fixed number of processes that repeat that behaviour infinitely*

Golf Club Model

```
const N=5
range B=0..N
ALLOCATOR = BALL[N],
BALL[b:B]=(when (b>0)get[i:1..b]->BALL[b-i]
          |put[j:1..N]->BALL[b+j]
          ).
range R=1..N
PLAYER= (need [b:R]->PLAYER[b]),
PLAYER[b:R]=(get[b]->put[b]->PLAYER[b]).
set Experts = {alice,bob,chris}
set Novices = {dave,eve}
set Players = {Experts,Novices}
HANDICAP = ({Novices.need[4],Experts.need[1]}-> HANDICAP).
||GOLFCLUB=(Players:PLAYER||Players::ALLOCATOR||HANDICAP).
```

Liveness Analysis of Golf Club Model

```
const N=5
range B=0..N
ALLOCATOR = BALL[N],
BALL[b:B]=(when (b>0)get[i:1..b]->BALL[b-i]
          |put[j:1..N]->BALL[b+j]).
range R=1..N
PLAYER= (need [b:R]->PLAYER[b]),
PLAYER[b:R]=(get[b]->put[b]->PLAYER[b]).
set Experts = {alice,bob,chris}
set Novices = {dave,eve}
set Players = {Experts,Novices}
HANDICAP = ({Novices.{need[4]},Experts.need[1]}
    -> HANDICAP)+{Players.need[R]}.
||GOLFCLUB=(Players:PLAYER||Players::ALLOCATOR||HANDICAP).
progress NOVICE = {Novices.get[R]}
progress EXPERT = {Experts.get[R]}
||PROGRESSCHECK = GOLFCLUB>>{Players.put[R]}.
```

LTSA

- *Liveness analysis reveals*
- *Starvation of novice golfers*

Fair Golf Club Model

```
const TM=4
const N=4
range B=0..N
range T=1..TM
range R=1..N
set Experts = {alice,bob,chris}
set Novices = {dave}
set Players = {Experts,Novices}
TICKET = NEXT[1],
NEXT[t:T]=(ticket[t]->NEXT[t%TM+1]).
PLAYER = (need[b:B]->PLAYER[b]),
PLAYER[b:R]=(ticket[t:T]->get[b][t]->put[b]->PLAYER[b]).
ALLOCATOR=BALL[N][1],
BALL[b:B][t:T]=(when (b>0)get[i:1..b][t]->BALL[b-i][t%TM+1]
               |put[j:1..N]->BALL[b+j][t]).
HANDICAP = ({Novices.{need[4]},Experts.need[1]}
    -> HANDICAP)+{Players.need[R]}.
||GOLFCLUB=(Players:PLAYER||Players::ALLOCATOR||
           HANDICAP||TICKET).
progress NOVICE = {Novices.get[R][T]}
progress EXPERT = {Experts.get[R][T]}
||PROGRESSCHECK = GOLFCLUB>>{Players.put[R]}.
```

Revised Golf Ball Allocator

```
public class FairAllocator implements Allocator {
  private int available;
  private long turn=0;
  private long next=0;
  public FairAllocator(int n){available=n;}
  synchronized public void get(int n)
                throws InterruptedException {
    long myturn = turn; ++turn;
    while (n>available || myturn|=next) wait();
    ++next; available -=n;
    notifyAll();
  }
  synchronized public void put(int n) {
    available+=n;
    notifyAll();
  }
}
```

Demo

Joins in Java

- *The Player threads did not need to communicate results back to the GolfClub thread - we just could let them die*
- *Often it is necessary to communicate results back to main thread*
- *Example: Asynchronous RMI*
- *This can be achieved using the Java* `join()` *operation which waits for a client thread to die*

Example: Master/Slave

- *Master launches a Slave thread, which communicates result back to Master upon termination.*
- *Implemented using Java joins*
- *Master creates a new thread object* `st` *for Slave execution*
- *Master invokes st.join() to wait for thread to die*
- *Master then calls operation from Slave to obtain result.*

Demo

Modelling `join()` in FSP

- *Use explicit action for join*
- *Synchronize*

```
SLAVE=(start->rotate->join->SLAVE).
MASTER=(slave.start->rotate->slave.join->rotate->MASTER).
||MASTER_SLAVE = (MASTER || slave:SLAVE).
```

LTSA

Summary

- *Static vs. Dynamic Processes*
- *Example: Golf Club*
- *Modelling Dynamic Processes*
- *Golf Club FSP Model*
- *Liveness Analysis in Dynamic Systems*
- *Java Joins*
- *Modelling Java Joins*