Marc Ilsø Poulsen, Peter Bartram Hansen og Bjarne Antonsen 22/11-2010
dLego: Lab Session 10 Rapport
Varighed: 3 timer

Formål

Til denne lab session skulle vi undersøge adfærdsbaserede arkitekturer. Helt konkret skulle vi undersøge implementationer af disse i leJOS NXJ API'en og dernæst udvide en sådan arkitektur med et nyt adfærdsmønster.

Plan

Vi vil først undersøge funktionalitet af BumperCar klassen i leJOS samples og hvordan klasserne lejos.subsumption.Behavior og lejos.subsumption.Arbitrator bliver anvendt i forbindelse med dennes implementation. Dernæst vil vi udvide BumperCar klassen med et ekstra adfærdsmønster og til sidst anvende tråde til at sample værdier fra ultrasonic sensoren og til interrupts af DetectWall adfærdsmønsteret.

Resultater

Robotten som vi andvendte til denne lab session kan ses her:

Vores første undersøgelse af touch sensoren viste, at hvis denne var trykket i bund, så udfører robotten konstant metoden action() i DetectWall klassen. Initialiseringen af BumperCar klassen sørger for, at Arbitrator klassen giver højere prioritet til DetectWall adfærden end DriveForward adfærden og derfor bliver sidstnævnte ikke udført, når touch sensoren er aktiveret.

Dernæst undersøgte vi klassen Arbitrator for at opklare, hvorvidt takeControl() metoden i DriveForward klassen blev udført, når triggering betingelsen i DetectWall var true. Det blev den ikke, hvilket vi indså ved at kigge på for-loopet ved linje 122 i Arbitrator klassen, hvor den adfærd, som har højest prioritet bliver udført, hvis triggering betingelsen er true. Alle adfærdsmønstre som måtte være på lavere lag i arkitekturen bliver derved ignoreret, så længe et højere lags triggering betingelse er true.

Herefter implementerede vi vores eget adfærdsmønster i form af klassen Exit, hvis kildekode er meget enkel og kan ses her:

```
class Exit implements Behavior
{
	@Override
	public void action()
	{
		System.exit(0);
	}

	@Override
	public void suppress(){}

	@Override
	public boolean takeControl()
	{
		return Button.ESCAPE.isPressed();
	}
}
```

Som det gerne tydeligt skulle fremgå, så sørger dette opførselsmønster for at kalde System.exit() når Escape knappen bliver trykket på. Da dette opførselsmønster skulle have højeste prioritet var det ikke nødvendigt at implementere suppress (jvf implementationen af suppress() i DetectWall klassen, som den oprindeligt var implementeret.

Det var dog ikke nok blot at tilføje dette adfærdsmønster til Arbitrator klassens prioritetsliste, da dele af DetectWall's adfærdsmønster var blokerende, og Exit adfærdsmønsteret derfor ikke blev udført, hvis man trykkede på Escape, når de blokerende kald var ved at blive udført. Derfor ændrede vi DetectWall klasses implementation af action() metoden som følger:

```
	public void action()
	{
		_suppressed = false;

		Motor.A.rotate(-180, true);	// start Motor.A rotating backward
		Motor.C.rotate(-360, true);	// rotate C farther to make the turn

		while(!_suppressed && !Motor.C.isStopped())
		{
			Thread.yield();
		}
	}
```

Denne implementation fik robotten til at udvise samme adfærd men var ikke blokerende som før.

Dernæst gik vi i gang med at ændre på den måde, hvorpå sampling af værdier fra ultrasonic sensoren blev udført.
Ved brug af Sound.pause(20) i takeControl() metoden i DetectWall klassen blev Exit adfærdsmønsteret udført med det samme, når man trykkede på Escape knappen, men hvis vi ændrede dette til Sound.pause(2000) kunne vi tydeligt se en forsinkelse, da dette kald var blokerende.

For at gøre kaldet ikke-blokerende besluttede vi at flytte selve samplingen af ultrasonic sensoren over i en separat tråd, som så kunne sample hvert 20. msek. For at implementere dette introducerede vi en ny klasse ved navn SonarSampler, som netop fik ansvaret for at sample værdier fra ultrasonic sensoren:

```
class SonarSampler extends Thread
{
	private UltrasonicSensor sonar;
	private int distance;

	public SonarSampler(UltrasonicSensor sonar)
	{
		this.sonar = sonar;
	}

	public void run()
	{
		while(true)
		{
			sonar.ping();
			Sound.pause(20);
			distance = sonar.getDistance();
		}
	}

	public int getDistance()
	{
		return distance;
	}
}
```

Som det ses er implementationen af denne klasse meget lig funktionaliteten af den oprindelige action() metode i DetectWall klassen.
Vi måtte også ændre lidt i DetectWall klassen, som nu kom til at se således ud:

```
class DetectWall implements Behavior
{
```

```
	private boolean _suppressed = false;
	private SonarSampler sonarSampler;
	private AvoidThread avoidThread;

	public DetectWall()
	{
		touch = new TouchSensor(SensorPort.S1);
		sonar = new UltrasonicSensor(SensorPort.S3);
		sonarSampler = new SonarSampler(sonar);
		new Thread(sonarSampler).start();
	}

	public boolean takeControl()
	{
		return touch.isPressed() || sonarSampler.getDistance() < 25;
	}

	public void suppress()
	{
		_suppressed = true;
	}


	public void action()
	{
		_suppressed = false;

		Thread t = new Thread(avoidThread);
		t.start();

		while(!_suppressed && !Motor.C.isStopped())
		{
			Thread.yield();
		}
	}

	private TouchSensor touch;
	private UltrasonicSensor sonar;
}
```

Her skal primært bemærkes ændringer i konstruktøren og action() metoden.
Ved hjælp af disse ændringer kunne vi observere, at robottens generelle adfærd var som før. Ved at ændre Sound.pause(20) kaldet til Sound.pause(2000) kunne vi se, at dette ikke længere blokerede for Exit adfærdsmønsteret, da samplingen nu kørte i en separat tråd.

Som det næste ville vi udvide DetectWall adfærdsmønsteret, så robotten bakkede i et sekund inden

den begyndte at dreje. Dette kunne vi nemt klare ved at ændre implementationen af action() metoden som følger:

```
	public void action()
	{
		_suppressed = false;

		Motor.A.backward();
		Motor.C.backward();
		Sound.pause(1000);
		Motor.A.rotate(-180, true);    // start Motor.A rotating backward
		Motor.C.rotate(-360, true);    // rotate C farther to make the turn

		while(!_suppressed && !Motor.C.isStopped())
		{
			Thread.yield();
		}
	}
```

Endelig ville vi ændre på DetectWall adfærdsmønsteret, så det skulle være muligt at interrupte dette og starte igen, hvis touch sensoren blev aktiveret. For at gøre dette introducerede vi en ny klasse kaldet AvoidThread, som skulle udføre adfærdsmønsteret af DetectWall klassen:

```
class AvoidThread extends Thread
{
	public void run()
	{
		Motor.A.backward();
		Motor.C.backward();
		Sound.pause(1000);
		Motor.A.rotate(-180, true);    // start Motor.A rotating backward
		Motor.C.rotate(-360, true);    // rotate C farther to make the turn
	}
}
```

Herefter måtte vi foretage ændringer i DetectWall klassens action() metode, så denne i stedet for at udføre adfærdsmønsteret nu skulle uddelegere opgaven til en tråd:

```
	public void action()
	{
		_suppressed = false;
```

```
            Thread t = new Thread(avoidThread);
            t.start();

            while(!_suppressed && t.isAlive())
            {
                    if(touch.isPressed())
                    {
                            while(touch.isPressed()){}
                            t.interrupt();
                            t = new Thread(avoidThread);
                            t.start();
                    }
            }

            while(!_suppressed && !Motor.C.isStopped())
            {
                    Thread.yield();
            }
    }
```

Her måtte vi bruge en busy-wait for at sikre os, at t.interrupt() og t.start() kun blev kald en enkelt gang hver, når touch sensoren blev aktiveret.
Ved hjælp af disse ændringer udviste robotten den ønskede adfærd. Det var nu muligt at interrupte DetectWall adfærdsmønsteret.

Konklusion

Gennem brug af Behavior og Arbitrator klasserne i leJOS NXJ API'en har vi fået indblik i, hvordan en adfærdsbaseret arkitektur kan implementeres. For yderligere at forbedre programmet kunne man anvende Thiemo Krink's motivationsfunktioner, sådan at det ville være muligt at interrupte DetectWall, hvis touch sensoren blev aktiveret igen, når robotten drejede væk.