

OOMPA 2001 Laboration 5

VisualWorks

1 Inledning, allmän information och basala kunskaper om VisualWorks

Avsikten med den här laborationen är att du ska få baskunskaper om:

- a) de olika delarna och filerna i Smalltalks omgivning,
- b) hur man *startar*, *sparar* och *avslutar* en Smalltalkomgivning (eng. Image),
- c) hur man använder kodkatalogen (eng. Browser),
- d) hur klasser konstrueras och testas,
- e) test några designmönster till och implementera dom i Smalltalk,
- f) testa lite av Smalltalks reflexiva system med bla metaklasser,
- g) konstruera en lite större egen applikation.

1.1 Innan du börjar

Läs igenom föreläsningssanteckningarna och orientera dig om referenserna där.

1.2 Starta Smalltalk

KONSTRUERA EN EGEN KATALOG

För att förenkla administrationen av de filer som konstrueras under Smalltalklaborationerna bör du börja med att skapa en ny katalog avsedd för kursens laborationer. Gör t.ex. följande (i UNIX):

```
mkdir MinSmalltalk
```

Gå sedan ned i den nya katalogen med:

```
cd MinSmalltalk
```

Om du inte redan har gjort det:

- konstruera en egen Smalltalkkatalog och
- gå ned i denna katalog.

VAR FINNS SMALLTALK?

På NADA:s UNIX-maskiner hittar du VisualWorks 5i4.nc i katalogen:

```
/pkg/visual/5i.4nc
```

STARTA SMALLTALK

Börja med att (på NADA) lägga till VisualWorks-modulen:

```
>module add visual/5i.4nc
```

som i huvudsak sätter omgivningsvariabeln \$VISUALWORKS att peka rätt (bra

att veta om man sitter hemma...).

Starta objektmaskinen (eng. **Object Engine**), dvs VisualWorks virtuella maskin, med aktuell *omgivningsfil* (eng. *image*) som argument:

```
DATAN>/pkg/visual/5i.4nc/bin/solaris/visual /pkg/visual/5i.4nc/image/visualnc.im
```

I kommandot ovan har vi angett att den *image* som levereras med systemet ska användas. Om vi har en egen image anger vi denna som argument istället. Tex:

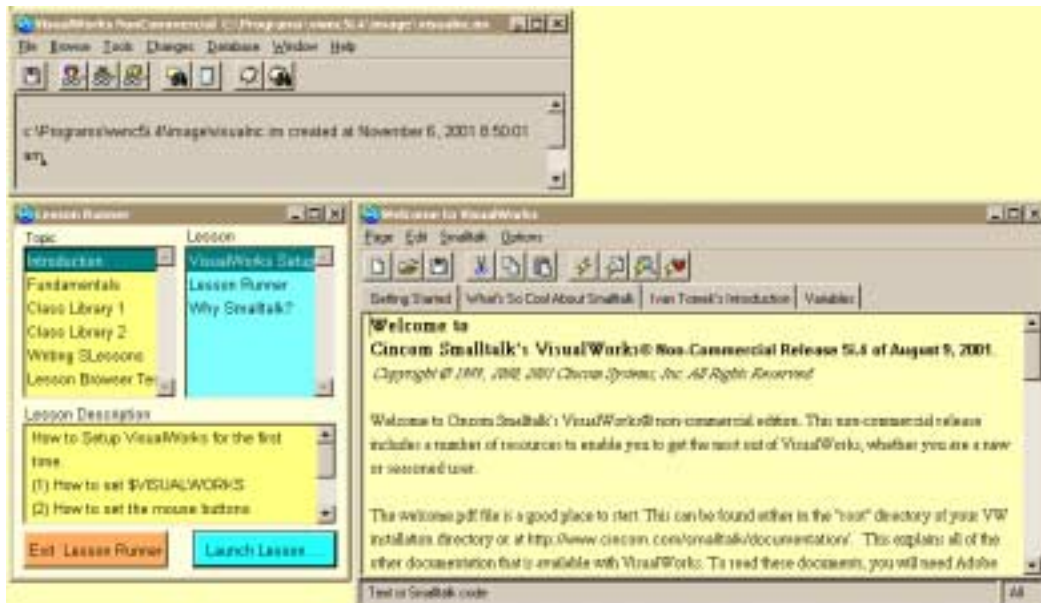
```
DATAN>/pkg/visual/5i.4nc/bin/solaris/visual minEgenImage.im
```

Om du har adderat VisualWorks-modulen så hittar dock systemet visualworkskatalogen och det räcker med att skriva:

```
DATAN>visual minEgenImage.im
```

- Starta nu Smalltalk med standardimagen!

Smalltalksystemet startar nu och efter en stund ska följande fönster visas på skärmen:



Omgivningen VW5i.4 nc

Det övre av dessa fönster är Smalltalks *huvudmeny* (ibland kallad Launcher), som också innehåller det så kallade *Transcript-fönstret*. Det undre vänstra fönstret är en enkel (lite halvdann) lektionsbrowser för Smalltalk. Det högra fönstret är ett *arbetsfönster* (Workspace) som innehåller lite allmänna beskrivningar av VisualWorks-systemet och Smalltalk.

1.3 Spara, avsluta och återstarta omgivningen

SPARA

Använd nu menyalternativet *Save As...* under rubriken *File* i Launchern för att spara din image i aktuell katalog. Ange önskat imagenamn utan extension i den dialog som följer. Observera att det kan ta ett par minuter att spara, speciellt om nätet

är belastat (på lokal disk eller på maskin av typ PC tar det förmodligen bara några sekunder)

- Vilka filer finns nu i din Smalltalkkatalog?

AVSLUTA

Använd menyalternativet *Exit VisualWorks...* i *File*-menyn för att avsluta sessionen!

ÅTERSTART

Använd kommandot `visualnc imagenamn` i din Smalltalkkatalog för att starta Smalltalk igen.

- Vad händer och hur ser bilden ut?

2 Uppgifter

2.1 Inledande tutorial

För kännedom om några grundläggande klasser gör Ivan Tomeks Introduction to VisualWorks Smalltalk del (part) 1-4, resten läses för kännedom. Tutorialen finns on-line i imagen i ett av arbetsfönstren då du startar VisualWorks men kan också öppnas via *Launcherns* **Help**-meny.

Möjligen kan du titta på Lesson Runner också.

Dessutom bör du vid behov titta på referenserna från föreläsning 16. Som ju bland annat innehåller beskrivningar av språkets grunder och dom olika utvecklingsverktygen.

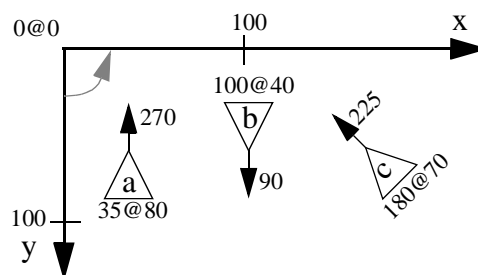
2.2 Konstruera en klass Turtle

Nu ska du med hjälp av browsern konstruera en ny klass kallad `Turtle`. Som framgår av namnet är den tänkt att representera en sköldpadda.

Vi tänker oss att en sköldpadda har en *position* och en *riktning* som representeras av instansvariabler.

- Förflyttning
En sköldpadda kan antingen flyttas till en absolut position eller en viss sträcka i dess aktuella riktning.
- Riktningförändring
Riktningen ska gå att kontrollera genom att antingen ange en absolut vinkel eller en relativ vinkel (relativt sköldpaddans aktuella riktning).

Följande figur illustrerar en situation med tre instanser av `Turtle`.

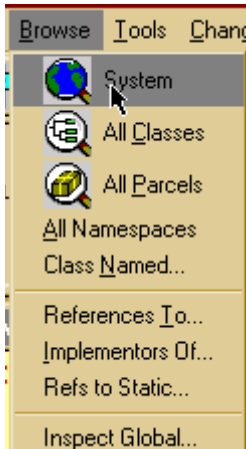


Koordinatsystem och paddor

Figuren visar också att Smalltalks koordinatsystem har origo högst upp till vänster och följaktligen blir den positiva rotationsvinkeln medurs. Nollriktningen är (som vanligt) längs abscissan (x-axeln).

I figuren är också sköldpaddornas positioner och rotationsvinklar inskrivna. Som framgår skrivs punkter (instanser av `Point`) på formen *x-koordinat@y-koordinat*.

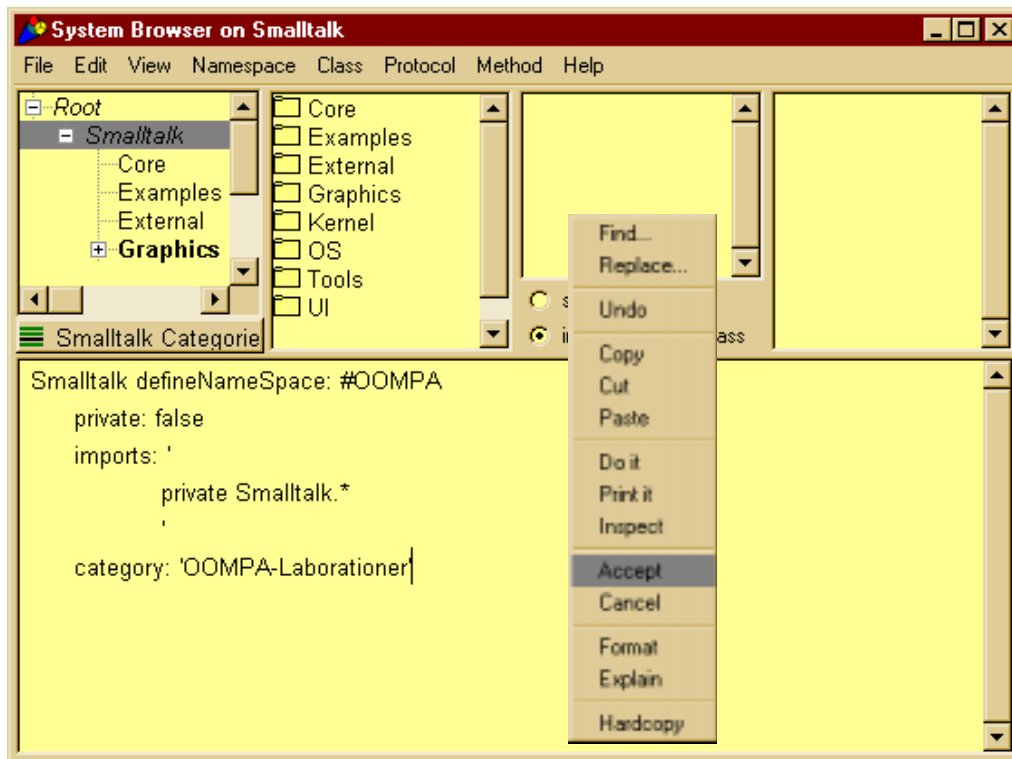
KONSTRUERA NAMNRYMD (ENG. NAME SPACE)



Konstruera en ny namnrymd genom att antingen i en System Browser (öppnas via menyalternativet **Browser->System i Launchern**) eller direkt skriva och "göra **Do it**" på följande:

```
Smalltalk defineNamespace: #OOMPA
  private: false
  imports: '
    private Smalltalk.*
  '
  category: 'OOMPA-Laborationer'
```

Eller med andra lämpliga namn på namnrymd (här OOMPA) och kategori (här OOMPA-Laborationer).



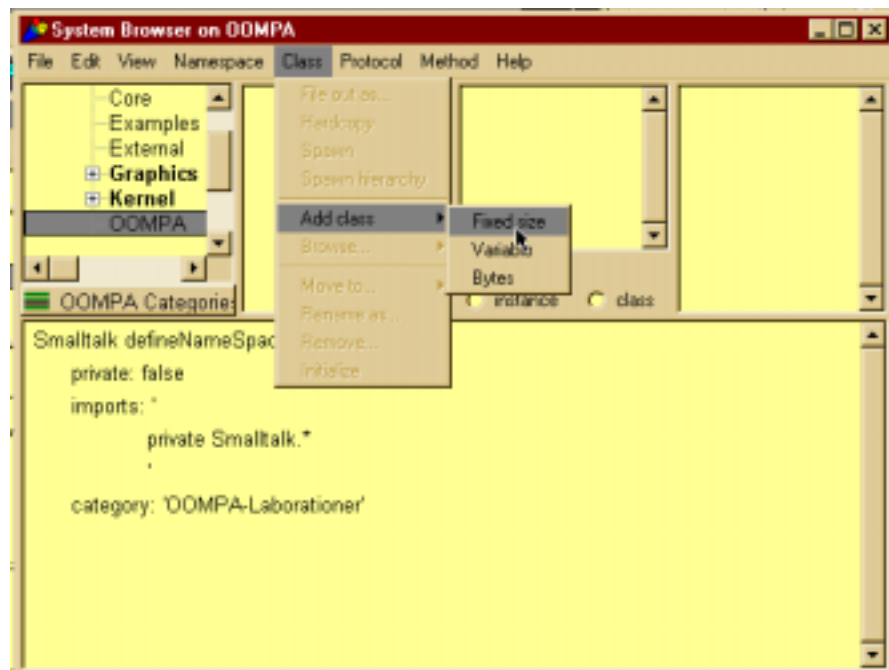
Skapa namnrymd

DEFINIERA KLASS

Konstruera sedan en klass efter följande beskrivning:

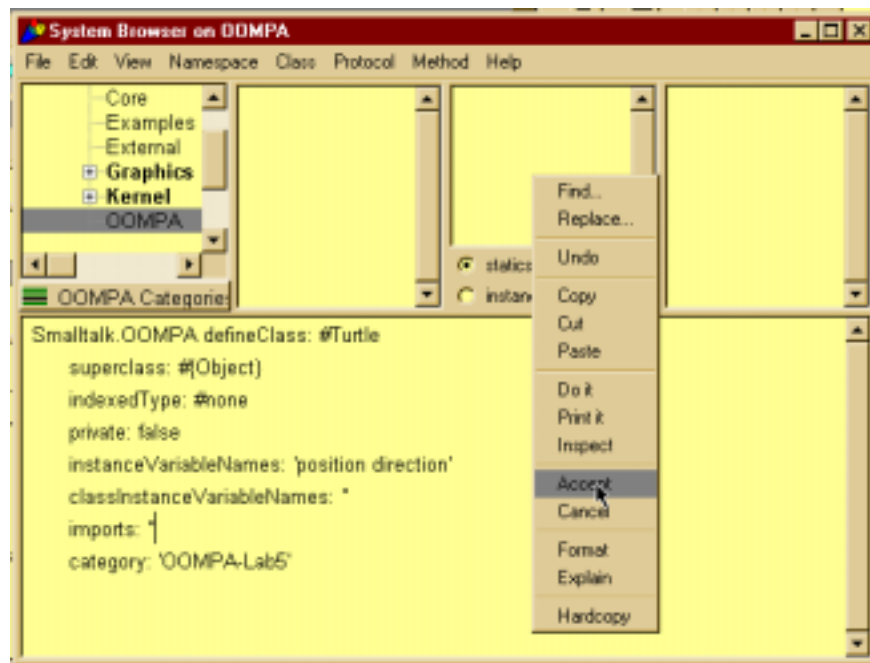
```
Smalltalk.OOMPA defineClass: #Turtle
  superclass: #{Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'position direction'
  classInstanceVariableNames: ''
  imports: ''
  category: 'OOMPA-Lab5'
```

Ledning: Skriv in texten m.h.a. browsern, menyalternativet Class->Add class->Fixed size, ändra mallen och använd sedan menyalternativet *accept*.



Skapa klass av fixed (normal) typ

och skriv sedan in klassdefinitionen följt av **accept**. (Det går också att lägga till ny klass via menyalternativ under rubriken **Namespace**).



Kompilera med Accept

NY MEDDELANDEKATEGORI

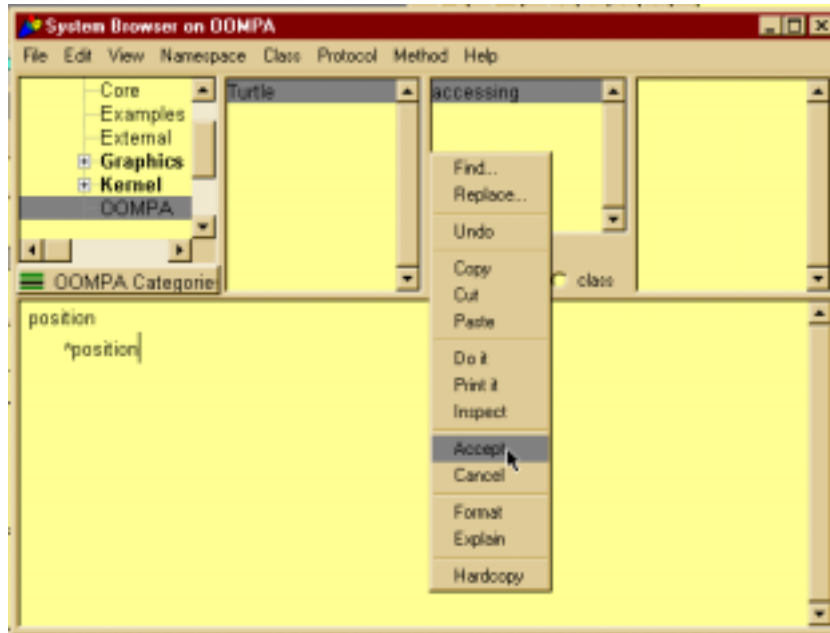
Skapa en ny meddelandekategori med namnet `accessing`.

METODER FÖR ATT LÄSA INSTANSVARIABLER

Skriv metoder för att läsa instansvariablerna. Ge inspektorerna följande namn:

- position
Returnera sköldpaddans position.
- direction
Returnera sköldpaddans riktning.

Se till att metoderna hamnar i kategorin `accessing`.



Kompilera med accept (som vanligt)

METODER FÖR ATT ÄNDRA INSTANSVARIABLER

Skriv följande instansmetoder avsedda att tilldela det givna argumentets värde till respektive instansvariabel:

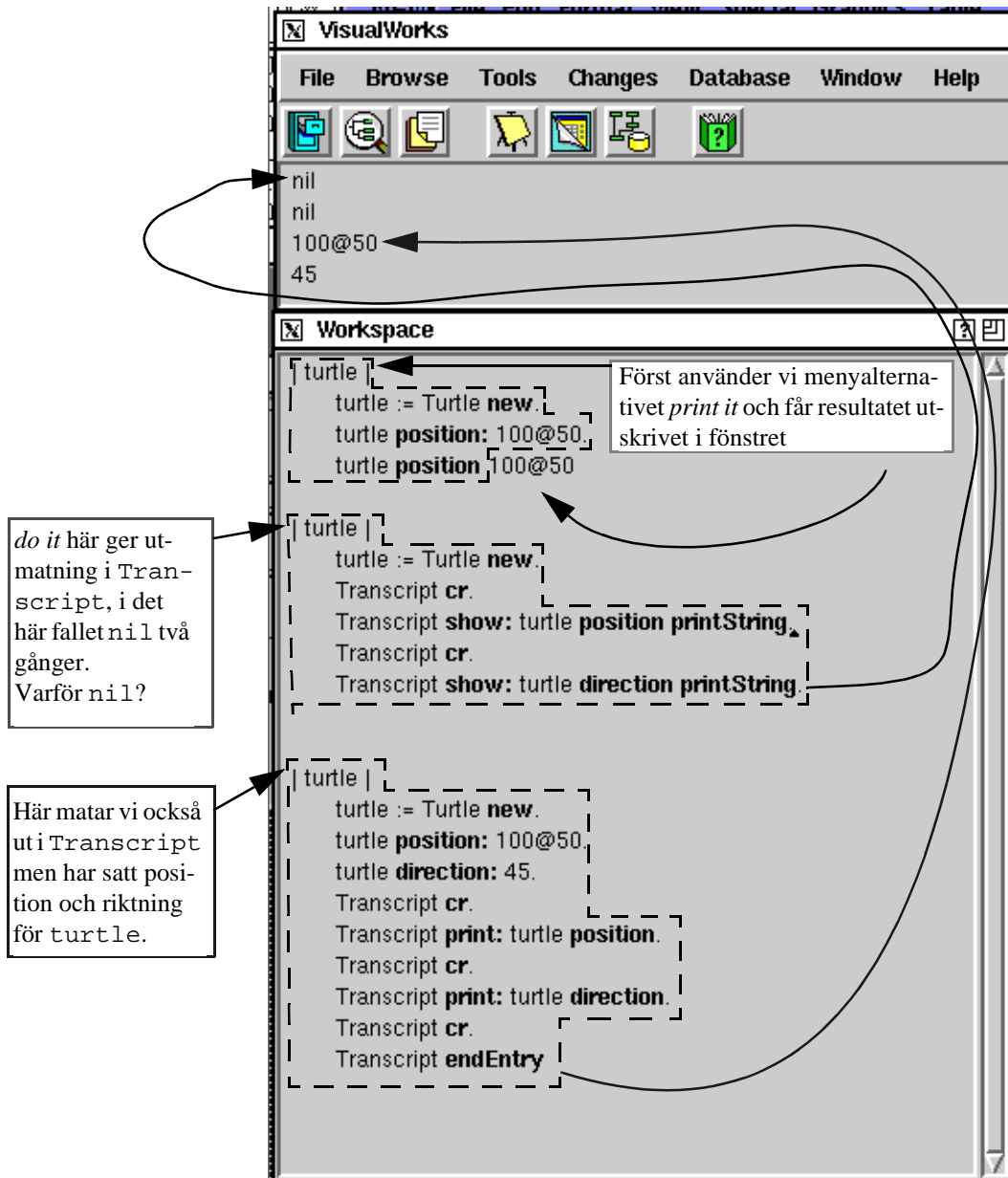
- `position: aPoint`
Ändra sköldpaddans position, där `aPoint` är instans av klassen `Point`.
- `direction: anAngle`
Ändra sköldpaddans riktning, där vinkeln anges i grader.

TESTKÖRNING AV KODEN

Observera bilderna i detta avsnitt tagna i VisualWorks-2, men det hela fungerar och ser nästan exakt likadant ut i VisualWorks-5.

I ett Workspace

Nu kan vi pröva det vi har skrivit hittills genom att t.ex. konstruera instanser av `Turtle` i ett Workspace och på vanligt sätt markera och använda *print it* eller *do it* från mittknappsmenyn. I figuren nedan illustreras detta.



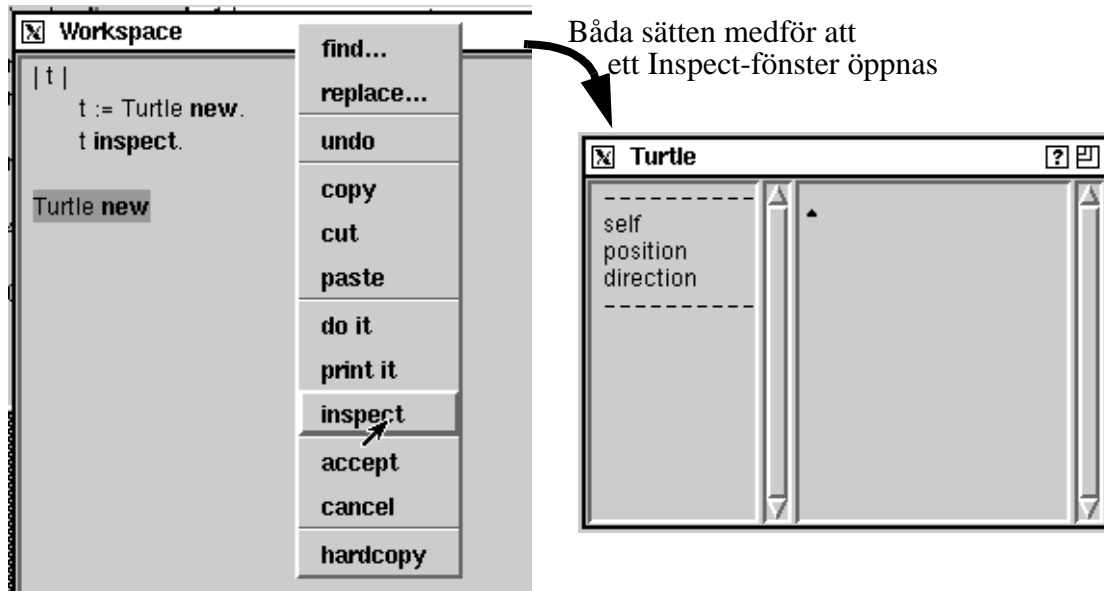
Skärmdump från Launchern i VW 2

Test av kod i ett Workspace där bl.a. Transcript används som utmatningsfönster.

Med Inspectfönster

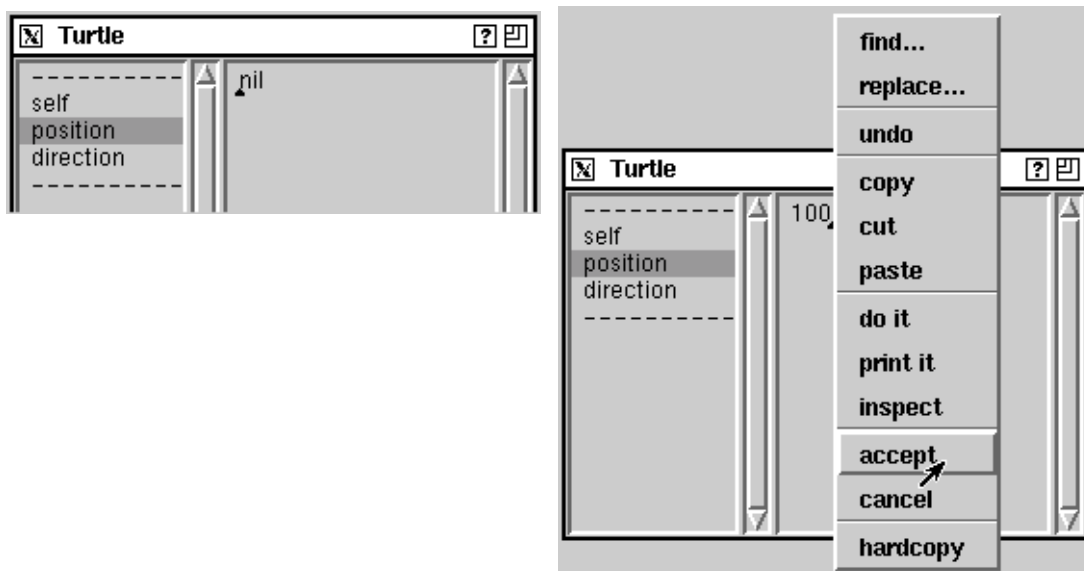
En trevligare och bättre teknik är att använda *Inspect-fönster*. Alla objekt förstår meddelandet `inspect`. Om vi skickar detta till ett objekt så öppnas ett speciellt

fönster för att inspektera den aktuella instansen. Istället för att explicit skicka meddelandet `inspect` kan man markera det som ska inspekteras och välja `inspect` i mittknappsmenyn i ett Workspace.



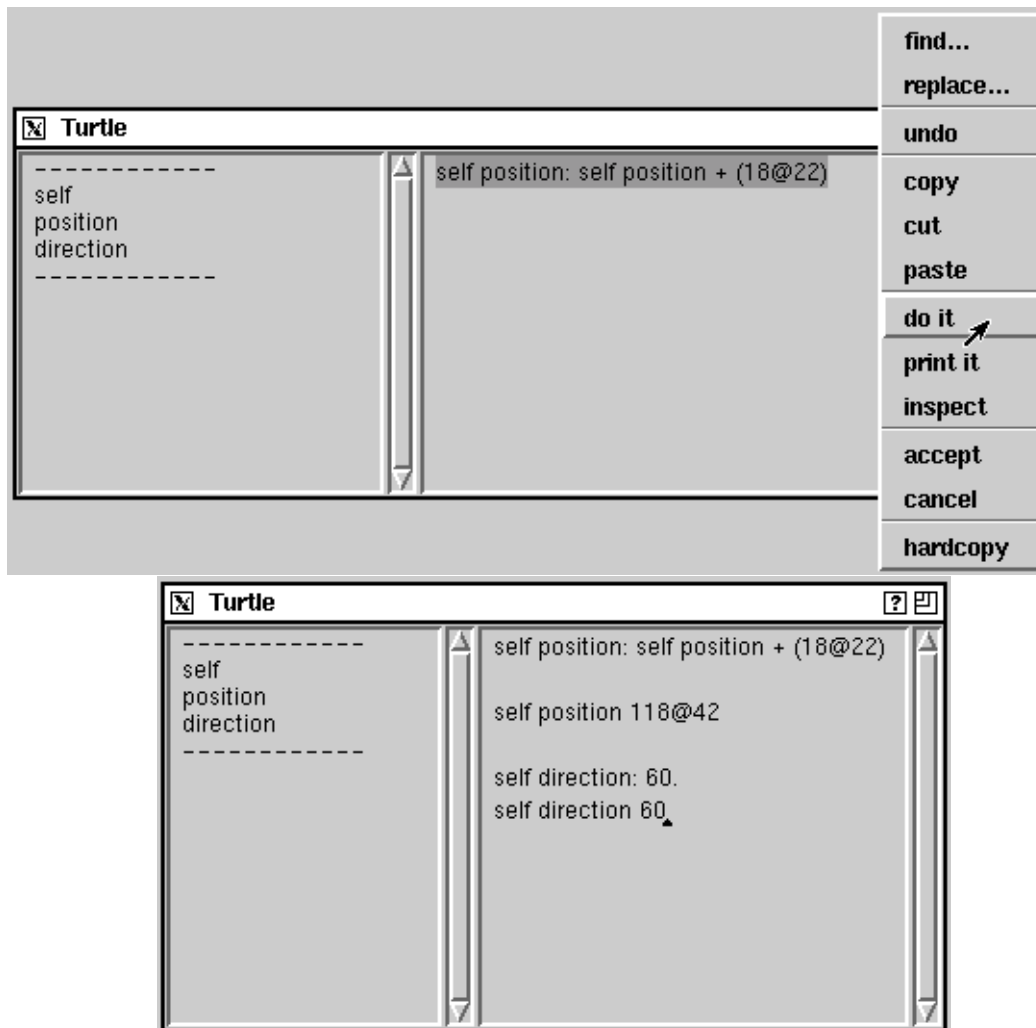
Inspektion av ett objekt (från VW 2)

Man kan se variablernas värden genom att klicka på dem och även ändra dem genom att skriva in nytt värde följt av `accept` i menyn.



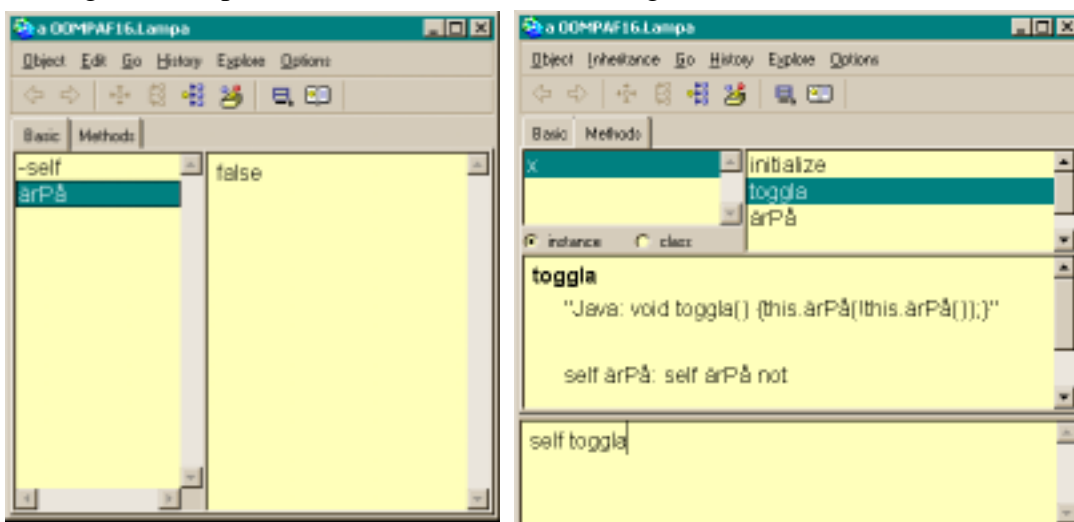
Inspektion och ändring av attributet position (VW 2)

Det går precis som i ett Workspace att skicka meddelanden till objekt i ett Inspect-fönster. Pseudovariabeln `self` representerar då den inspekterade instansen.



Evaluering av kod i Inspect-fönster (VW 2)

I VW 5.4 ser inspektorfönstret ut så här, vid öppnande till vänster och med möjlighet att se på koden och evaluera den till höger:



Inspect i VW 5i.4

TESTA KODEN

Testa det du hittills har skrivit genom att konstruera SUnit-tester. Använd sedan SUnit för att testa resten av den kod du skriver.

FLER INSTANSMETODER

Nu ska du utöka klassen `Turtle` med några ytterligare instansmetoder. Du får själv välja lämpliga namn på de meddelandekategorier/protokoll som du placerar metoderna i.

Relativ flyttning av sköldpadda

Skriva en metod som flyttar sköldpaddan en viss sträcka längs med dess aktuella riktning. Skriv metoden i form av en nyckelordsmetod med en formell parameter, `go: distance`, där `distance` ska vara ett tal som anger hur långt sköldpaddan ska flyttas.

Metodförslag (pseudokod i kursiv stil):

```

go: distance
  dir := aktuell-riktning-i-radianer
  scaling := dir cos @ dir sin.
  self position: scaling multiplicerad med
                distance + aktuell position

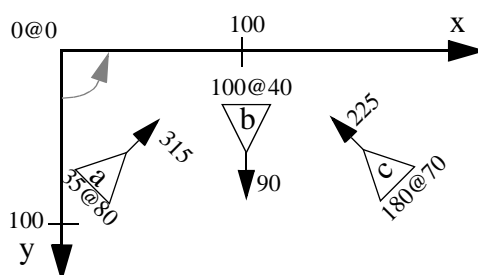
```

Ledning: Det finns en metod `degreesToRadians`.

Relativ riktningsförändring

Skriv också en metod `turn: angle`, för att rotera (d.v.s. uppdatera `direction` för) en sköldpadda med ett givet antal grader relativt aktuell riktning.

Om vi t.ex. gör `a turn: 45` i ovan får vi följande situation:



Padda a roterad 45 grader

Avstånd mellan sköldpaddor

Skriv en metod `dist: anotherTurtle`, som ger det vinkelräta avståndet mellan sköldpaddan som är mottagare av meddelandet och den som ges som argument.

Ledning: En sköldpaddas position är instans av klassen `Point`. I meddelandekategorin `point functions` i klassen `Point` finns en lämplig metod som returnerar avståndet mellan mottagare och en punkt som ges som argument!

Exempel: Om vi t.ex. vill skapa paddorna a och b enligt ovan (men utan rotationer) och

sedan mäta avståndet mellan dem kan vi göra det med följande enkla kodavsnitt:

```
| a b |
  a := Turtle new.
  b := Turtle new.
  a position: 35@80.
  b position: 100@40.
  a dist: b
```

Detta ska ge svaret 76.3217.

ÄNDRING AV METOD OCH DEFINITION

Ändra en metod

Ändra metoden `direction: anAngle`, så att instansvariabeln `direction` alltid tillhör det halvöppna intervallet $[0, 360)$.

Ledning: meddelandet `\` ger mottagaren modulo argumentet.

Exempel: `70 \ 9` ger svaret 7.

Ändra definition

För att bl.a. enklare kunna identifiera en viss sköldpadda, lägg till en instansvariabel `name` i klassdefinitionen. Skriv också lämpliga metoder för att läsa respektive uppdatera denna variabel.

INITIERINGSMETODER

Vi kan fortfarande konstruera instanser av `Turtle` som inte har initialiserat sina instansvariabler på lämpligt sätt. Detta kan i värsta fall medföra att exekvering av vissa metoder misslyckas. T.ex. fås ett felavbrott om meddelandet `dist:` skickas till en oinitialiserad padda. Nu ska vi åtgärda detta och också skriva ett par klassmetoder som gör instansieringen enklare.

Instansmetoder för initiering

Skriv en metod `initialize` som på lämpligt sätt ger skönsvärden för instansvariablerna `direction` och `position` då den anropas.

```
Exempel: | turtle |
          turtle := Turtle new.
          turtle initialize
```

ska ge en lämpligt initierad padda. Man kan också skriva

```
| turtle |
  turtle := Turtle new initialize.
```

Definiera om klassmetoden `new`

Lösningen ovan är fortfarande inte riktigt tillfredställande då vi i alla fall kan glömma att anropa `initialize` och därmed lämna objektet oinitialiserat. Därför ska du nu skriva om klassmetoden `new` (här med ungefär samma effekt som att skriva en ny konstruktor i Java) så att den automatiskt anropar `initialize` för det nya ob-

jektet. Skapa en klassmeddelandekategori `instance creation` och placera följande metod där:

```
new
  ^super new initialize
```

Ledning: Klassmetoder visas och kan redigeras då knappen `class` är nertryckt i Browsern. (Du kan då inte se instansmetoderna.)

Pröva! `super new` skapar en ny `Turtle` m.h.a. den `new`-metod som `Turtle` har ärvt ifrån basklassen `Object`.

Vad skulle ha hänt om du istället hade skrivit:

```
new
  ^self new initialize. "Detta är felaktigt. Varför?"
```

Fler klassmetoder

Nu ska ni skriva ytterligare två klassmetoder som gör det lite smidigare att direkt konstruera instanser av `Turtle` med given position eller riktning. Skriv först:

```
position: initialPosition direction: initialAngle
| instance |
instance := self new.
instance position: initialPosition.
instance direction: initialAngle.
^instance
```

Hur kan vi skriva om denna metod utan att använda den (här onödiga) initialiseringen som görs via klassmeddelandet `new`?

Skriv också följande klassmetod:

```
position: initialPosition
  ^self position: initialPosition direction: Number zero
```

Pröva samt beskriv vilka meddelanden som sänds då vi utför följande kod:

```
Turtle position: 234@60
```

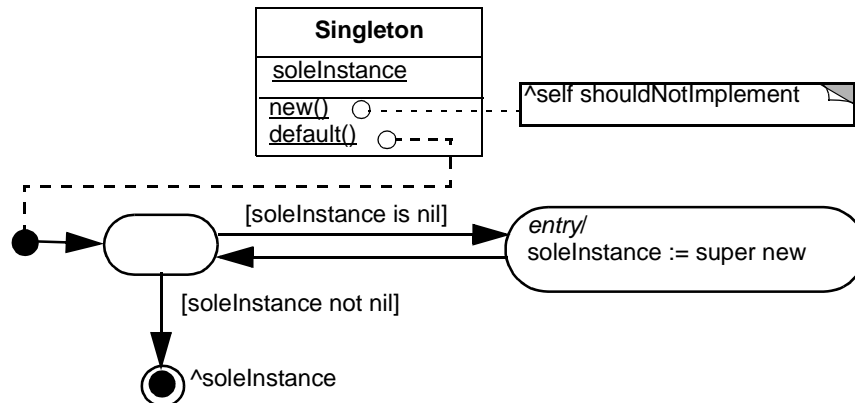
2.3 Designmönster

Designmönster är viktiga. Här ska du använda tre till. Två av dessa är nya i jämförelse med laboration 2, nämligen *singleton* och *prototype*, och en av dem fanns med även i denna tidigare laboration, nämligen *observer*.

Skriv också koden enligt XP-strategi med tester först skrivna i SUnit.

SINGLETON

Mönstret singleton ser till att en klass endast har en instans och erbjuder också en



Singleton med tillståndsdigram för konstruktören default.

global accessväg till den. Tex kan man ha flera utskriftsjobb till en skrivare men vanligen bara en utskriftskö. I Smalltalk använder bland annat metaklasser singleton. Ett annat exempel där mönstret är användbart är för att hantera en viss "service" som hanterar någon viss typ av socketkommunikation.

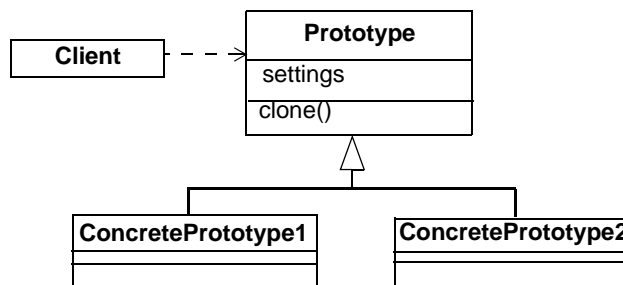
Uppgift

Konstruera en klass Singleton i enlighet med beskrivningen i figur . Observera att `soleInstance` kan deklaras antingen som klassvariabel eller som instansvariabel på klassidan. Vilka för- respektive nackdelar har det ena respektive andra sättet? Testa klassen och försäkra dig om att den fungerar.

Fundera gärna över andra användningsområden för mönstret. Har du några förslag?

PROTOTYPE

En prototyp är ett objekt som definierar en "prototypinstans". Nya objekt bildas genom kopiering av denna instans.



Prototype

Ett (artificiellt) exempel på användande av en prototyp skulle kunna se ut som följer:

```

prototypePen := Pen colored: ColorValue blue nibSize: 3.
pen1 := prototypePen clone.
pen2 := prototypePen clone.
pen2 nibSize: 4.
pen1 drawOn: graphicsContext.
pen2 drawOn: graphicsContext
  
```

Eller mer på riktigt i VisualWorks:

```

| redProto blueProto geo1 geo2 geo3 points rand |
"Vi skapar en referens till aktuellt fönsters grafiska
kontext"
redProto := ScheduledControllers activeController
           view graphicsContext.

"Se klassen GraphicsContext (tex via GraphicsContext
browse) för information om dom grafiska rutinerna"

redProto clear. "Rensa fönstret"
redProto paint: ColorValue red. "Rita med rött"
redProto lineWidth: 7. "Pentjocklek 7"
blueProto := redProto copy.
blueProto paint: ColorValue royalBlue.
geo1 := redProto copy.
geo2 := redProto copy.
geo3 := blueProto copy.
geo1 joinStyle: GraphicsContext joinRound.

"Nu testar vi genom att slumpa fram en lista av punkter och
sedan rita polygoner"
points := OrderedCollection new.
rand := Random new. "Slumptalsgenerator"
1 to: 10 do: [:i |
    points
        add: (i * rand next @ (i * rand next) * 50) truncated].
geo1 displayPolyline: points at: 10 @ 20.
geo2 displayPolyline: points at: 30 @ 0.
geo3 displayPolyline: points at: 40 @ 30
  
```

Uppgift (en enkel

Gör en möjlighet att utnyttja en prototysköldpadda (`Turtle`) med lämpliga initiala inställningar (på placering, riktning och liknande). Skapa en klassmetod som ger en kopia av en prototysköldpadda. Använd förslagsvis Singleton-mönstret eller liknande för att peka ut prototypen.

OBSERVER

Mönstret *observer* är väldigt viktigt och flitigt använt inte minst i interaktiva grafiska tillämpningar.

I denna uppgift skall du med hjälp av observer emulera ett minst sagt *urartat brevhänteringssystem*. Men det är ju mer principerna för mönstret än själva applikationen som är av intresse i detta moment!

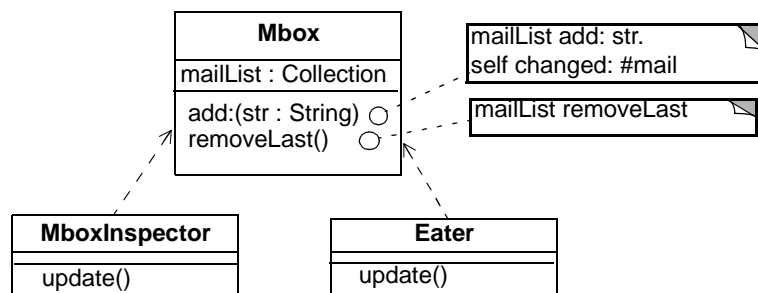
Förberedelse

Se Publisher-Subscriber-exemplet från föreläsning 18.

Uppgift

OBS! I exemplet är lite urartat men ur designmönsterhänseende illustreras och används i alla fall principerna för Observer-mönstret.

Konstruera en klass `Mbox`, en klass `Eater` och en `MboxInspector` enligt figur. Observera att du måste komplettera klassbeskrivningarna med lämpliga delar från beskrivningarna i litteraturlistan.



Mailbox

Skapa en instans av `Mbox` och gör både en instans av `Eater` och en av `MboxInspector` beroende av mailbox-instansen. Där `Eater` läser och tar bort det sista brevet från `Mbox`ens lista och `MboxInspector` skriver ut det i `Transcript`-fönstret. Som ni märker finns det många potentiella problem, som exempel

- hur får `Eater` kontakt med `Mbox` utan att explicit referera `Mbox`?
Tips: `update:with:from:`.
- hur kan `MboxInspector` skriva ut information om aktuellt brev om `Eater` har "hunnit före" och tagit bort det från mailboxen?
Tips: `update:with:`.
- är `removeLast` så bra egentligen?

Gör lämpliga förändringar och tillägg till klasserna och pröva dem genom att lägga till några "brev" till mailboxen.

2.4 Ändringar i systemet samt metaklasser

I VisualWorks har man nästan full kontroll över systemet. Hela systemet förutom den virtuella maskin är ju implementerat i systemet själv (fast i tex Squeek, Smalltalksystemet från Apple och Disney, så är även Virtuella Maskinen skriven i Smalltalk och därmed också både läs- och förändringsbar!). I VisualWorks kan man läsa och förändra koden i alla klasser, dvs i klasser som kompilator, booleska klasser, numeriska klasser, browser och även för andra systemverktyg. Koden för hur Smalltalks arv, metodhantering mm är också tillgänglig för inspektion, användning och förändring.

Trots allt så bör man inte "hej vilt" förändra grundläggande klasser, även om det är möjligt. Man bör tänka igenom det hela ganska noga först. Fast många gånger kan en liten förändring i tex klassen `Object` eller ett litet tillägg till kompilatorn göra att lösningen blir mycket mer objektorienterad och/eller att man bara behöver ändra en bråkdel så många klasser som man annars hade behövt.

ÄNDRINGAR I SYSTEMETS KLASSER

För att testa möjligheten att lägga till saker i systemets klasser gör följande tillägg:

- En metod `Object>>trace`

Som skriver ut information om mottagaren i `Transcript`. Vilket kan vara bekvämt i speciellt utvecklingsfasen av programvara. Detta unära meddelande kan sedan smidigt användas där man enkelt vill få ut lite info om objektet, som i:

```
x trace between: a trace + b and: z
```

Förslag till implementation:

```
trace
  Transcript show: self printString; cr
```

Andra klasser kan ju sedan självklart implementera egna specialicerade versioner (fast det behöver du inte göra inom ramen för denna laboration).

- Implementera metod `BlockClosure>>repeatUntil:`

Bakgrund och beskrivning:

Vissa programmerare saknar en konstruktion motsvarande Pascal's **REPEAT UNTIL**. Detta är dock inte något större problem, då vi åter igen kan utnyttja att allt är objekt och helt enkelt definiera våra egna styrstrukturer! Första tanken är kanske att skriva något i stil med:

```
repeat: ettBlock until: ettAnnatBlock
```

Men det går inte eftersom meddelandet `repeat:until:` måste skickas till något objekt. Vi flyttar `ettBlock` till före `repeat` och får metoden:

```
repeatUntill: anotherBlock
  self value.
  ^anotherBlock value
  ifFalse: [self repeatUntill: anotherBlock]
```

Nu kan vi direkt pröva våra nya styrstruktur.

```
| i |
i := 0.
[i := i + 1. Transcript print: i; endEntry] repeatUntil:
[i >= 10]
12345678910
```

Den rekursiva beskrivningen av `repeatUntil1:` ovan ger oss ganska dålig prestanda så vi ger två alternativa lösningar som drar nytta av att `while`-slingor är optimerade.

```
repeatUntil2: anotherBlock
self value.
^anotherBlock whileFalse: self
```

I det här fallet kommer VisualWorks kompilator klaga över att vi inte har använt litterala block, men det går att fortsätta i alla fall om man vill.

Om vi inte vill få klagomål och dessutom garantera att koden optimeras, kan vi istället implementera det hela enligt följande:

```
repeatUntil3: anotherBlock
self value.
^[anotherBlock value] whileFalse: [self value]
```

där vi uttryckligen klargör att de inblandade objekten är block.

- Implementera också en (sjuk?!) metod `maybe:` som när den skickas till ett booleskt objekt av typen `True` slumpmässigt skall utföra och returnera resultatet från evaluering av argumentblocket. I dom fall där inte blocket evalueras skall `nil` returneras. Om samma meddelande skickas till ett objekt av typen `False` så skall `nil` returneras. Använd klassen `Random` för att generera slumpantal (se exempelvis dess användning i mönstret `prototype` ovan).

Exempel:

```
- Mottagaren false
false maybe: [Transcript show: 'Jodå!'; cr]. "ger alltid nil
           eftersom mottagaren false"
4 < 2 maybe: [Transcript show: 'Jodå!'; cr]. "ger också
           alltid nil"

- Mottagaren true
true maybe: [Transcript show: 'Jodå!'; cr]. "blocket
           evalueras ibland och ibland inte"
4 > 2 maybe: [Transcript show: 'Jodå!'; cr]. "blocket
           evalueras ibland och ibland inte"
```

LETA EFTER OCH ANVÄNDA VISSA GRUNDLÄGGANDE SYSTEMMETODER

Klassen Behavior med subklasser

Klassen `Behavior` med subklasser beskriver `Smalltalk` som sådant med `super-/subklasser`, `metodkataloger`, osv. Dessa systemklasser innehåller också metoder för att be klasser om deras metoder, kompilera kod, lägga till instansvariabler mm. I klassen `Behavior` hittar man också "basimplementationen" av metoden `new`. Vet du varför den är implementerad här (som instansmetod)?

- Börja med att öppna en `Browser` på `Behavior`, antingen via `launchern` (tex genom att välja **Browse->All classes**, leta reda på `Behavior` via `Browser`n

och sedan öppna en *Hierarkisk Browser* (för att se alla dess subklasser på ett smidigt sätt) eller genom att helt enkelt "göra **Do it på**" Behavior browse.

- Skapa en klass `Motorbåt` som subclass till klassen `Object` med instansvariablerna `längd`, `vikt` och `pris` med hjälp av meddelandet:

```
subclass:
instanceVariableNames:
classVariableNames:
poolDictionaries:
category:
```

- Lägg till en instansvariabel `hästkrafter`

Ledning: Metoden `addInstVarName:`

- Skriv accessmetoder till åtminstone en av instansvariablerna mha meddelandet `compile:classified:`.

Ledning: Första argumentet är källkoden på textuell form och den andra namnet på den metodkategori i vilken man vill placera metoden.

Undersöka delar av systemet

Första punkten är ett rent letaövningsmoment! Använd Launchern för att:

- hitta alla implementationer av metoden `addInstVarName:`.

Använd metoder i `Behavior` med subklasser för att ta reda på följande om klassen `Metaclass` (självlklart kan du kontrollera dom flesta av dina resultat genom att parallellt undersöka klassen i Browsern):

Ledning: Många av metoderna hittar du i kategorin *accessing class hierarchy* respektive *accessing method dictionary* i klassen `Behavior`.

- Dess superklass
- Alla dess subklasser
- Alla metoder definierade i klassen
- Alla metoder definierade i klassen eller dess superklasser
- Instansvariabler definierade i klassen

Ledning: `instVarNames`

- Instansvariabler i klassen och i alla dess superklasser
- Alla dess instanser

Ledning: `allInstances`

- Hur ser bytekoden(/assemblerkoden) i metoden

`Metaclass>>addInstVarName: ut?`

Ledning: Gör inspect på `Metaclass compiledMethodAt: #addInstVarName:`

2.5 Till sist

Gör uppgift g) från laboration 2 dvs *Personregister med Observer, Strategy och Adapter*.

Använd som vanligt `SUnit` för att skriva enhetstester.

Ett tips är att använda `SortedCollection` för att sortera objekten, se tex Ivan Tomeks tutorial och referenserna föreläsning 16.