# Observer Pattern och MVC

Objekt-orienterad programmering och design

Alex Gerdes, 2016

# Model – View – Controller

- Model – View – Controller (MVC) är ett design pattern (architectural pattern) som är väldigt vanligt förekommande för alla typer av program som innehåller någon form av grafisk representation.

- Grunden i MVC är att vi separerar:
  - Koden för data-modellen (M) från användargränssnitt (V och C). Detta är den viktigaste uppdelningen.
  - Inom användargränssnitt skiljer vi koden som visar upp (delar av) modellen för användaren (V) från koden som hanterar input från användaren (C).

# Model

- Modellen (Model) är en representation av den domän som programmet arbetar över – helt oberoende av användargränssnitt.
  - Data, tillstånd, domänlogik.
  - Modellen i singular – vi har inte flera modeller för samma domän. Modellen kan dock internt bestå av flera olika delar som representerar *olika* aspekter av domänen (dvs ingen duplicering).

- Tumregel för avgränsning från V och C: Tänk ”The whole model and nothing but the model.”
  - Ska kunna presenteras för och kontrolleras av användaren på olika sett – e.g. med 2D-grafik eller text – utan att valet i sig påverkar e.g. modellens tillstånd.
    - ”Nothing but the model”
  - Ska innehålla allt det som vore detsamma oavsett hur modellen presenteras eller kontrolleras. Ingenting ska behöva dupliceras mellan olika vy- eller kontroll-komponenter.
    - ”The whole model”

# View

- En vy (View) beskriver (delar av) modellen för användaren.
  - Olika tänkbara format: Text, grafik (2D, 3D, …), ljud, haptik, …
  - Vi pratar om ”vy”, oavsett vilket format presentationen görs på.

- Vi kan ha många olika vyer (ofta samtidigt) för en och samma modell.
  - Olika vyer kan visa upp olika aspekter av modellen.
    - E.g. karta, inventory, synfält för ett dataspel.
  - Olika vyer kan visa upp samma aspekt på olika format
    - E.g. musik spelas upp, och visas samtidigt grafiskt som frekvens-amplitud-diagram.

- Vi vill (oftast) att vyn uppdateras när modellen den presenterar uppdateras.

# Controller

- En Controller styr modell och vy(er) utifrån input från användaren.
  - Vi kan ha många controllers som styr olika aspekter av både modell och vy.
  - Olika varianter av MVC använder controllers lite olika: alltifrån en enskild controller som styr alla aspekter av modell och vyer, till många separata controllers för varje del-vy.

- De flesta moderna grafik-gränssnitt (som Java Swing) innehåller stöd för att förenkla för controllers att hantera användar-inputs genom *events* och *event listeners*.

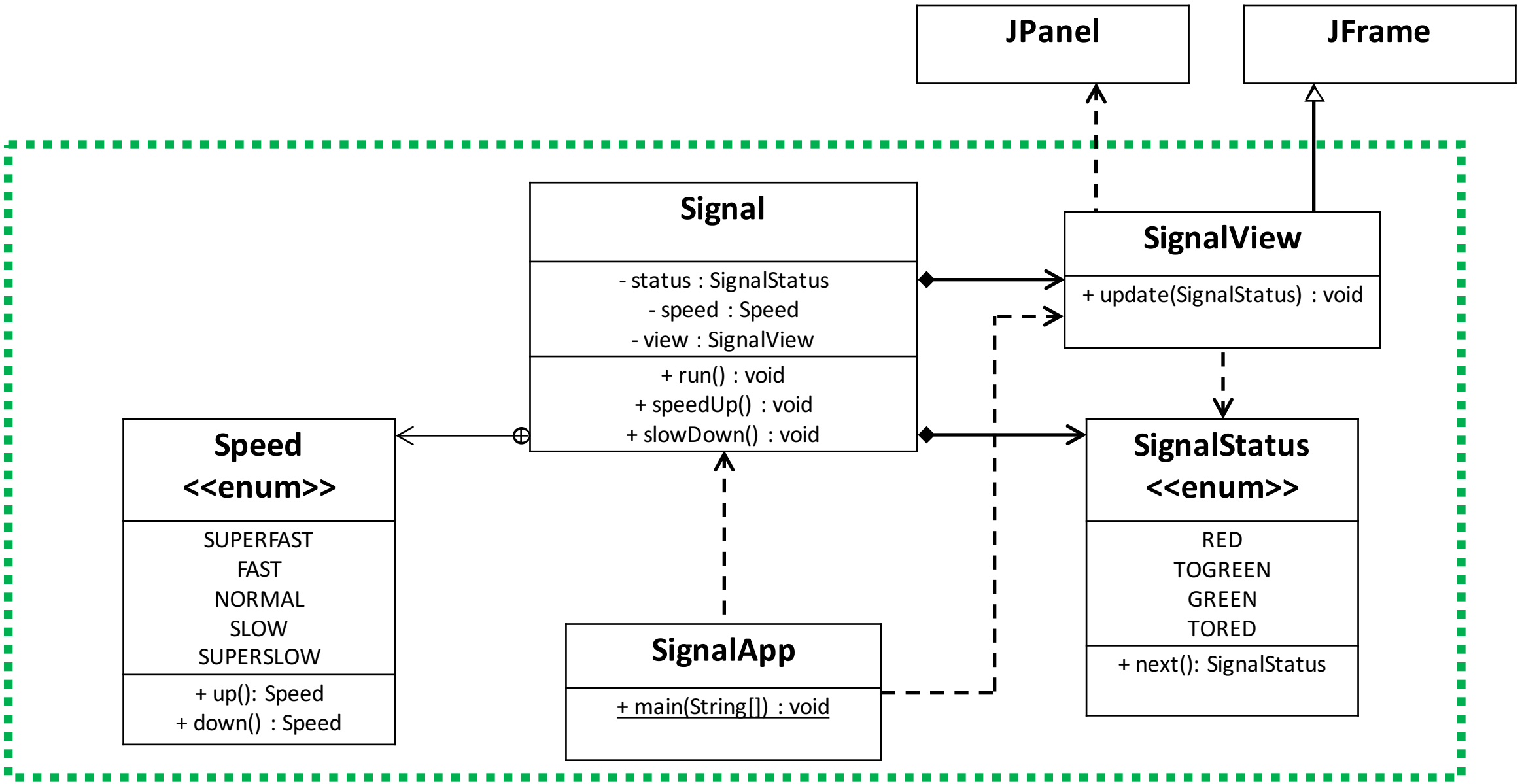
# Smart, Dumb, Thin

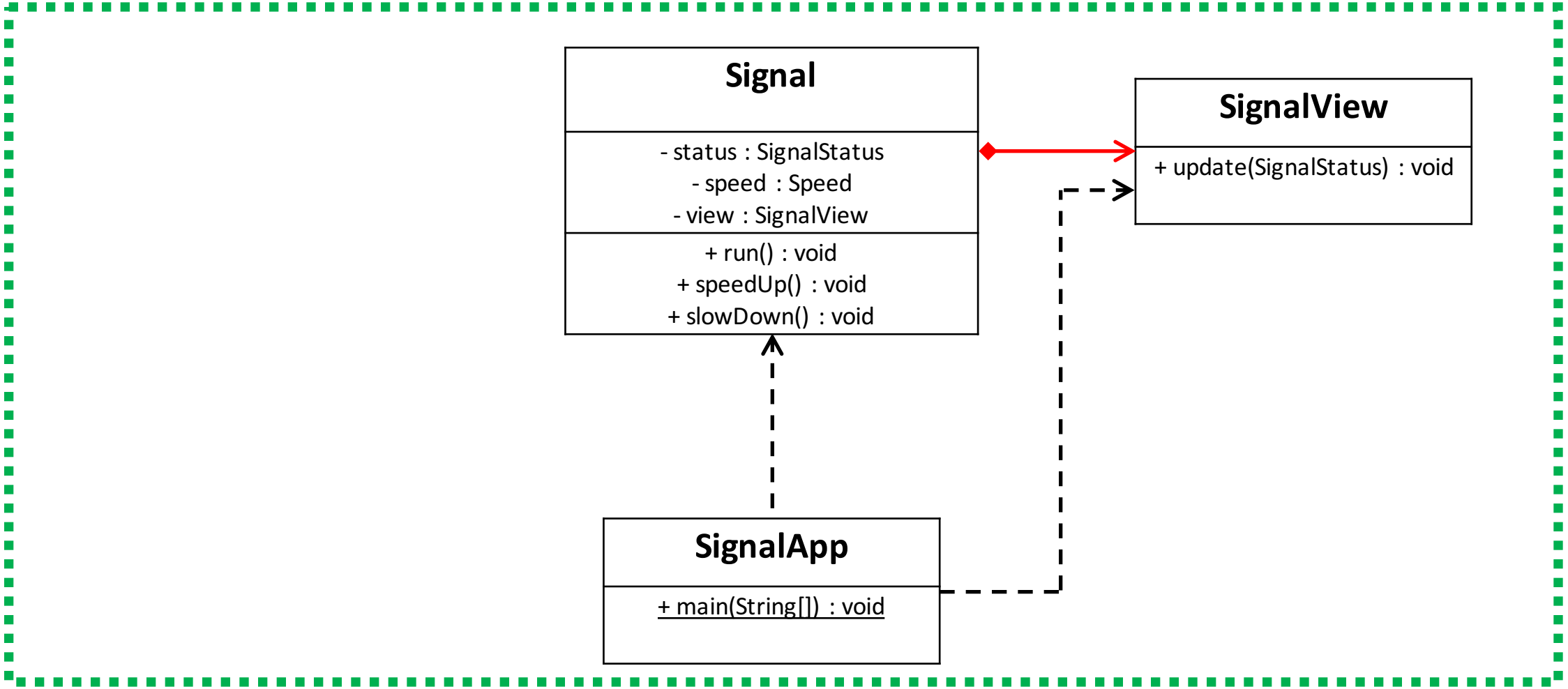
- Tumregel: SMART models, DUMB views, THIN controllers
  - All domänlogik ska ligga i modellen – därav **SMART**.
  - En view ska inte göra egna beräkningar, bara ”visa” utifrån direktiv – därav **DUMB**.
  - En controller ska enbart hantera yttre input, dvs (oftast) input från användare. Den ska bara vara ett tunt lager mellan användare och program – därav **THIN**.
    - Yttre input måste inte vara från användare direkt; det kan komma e.g. via nätverkskommunikation, från avläsning av sensorer, etc.

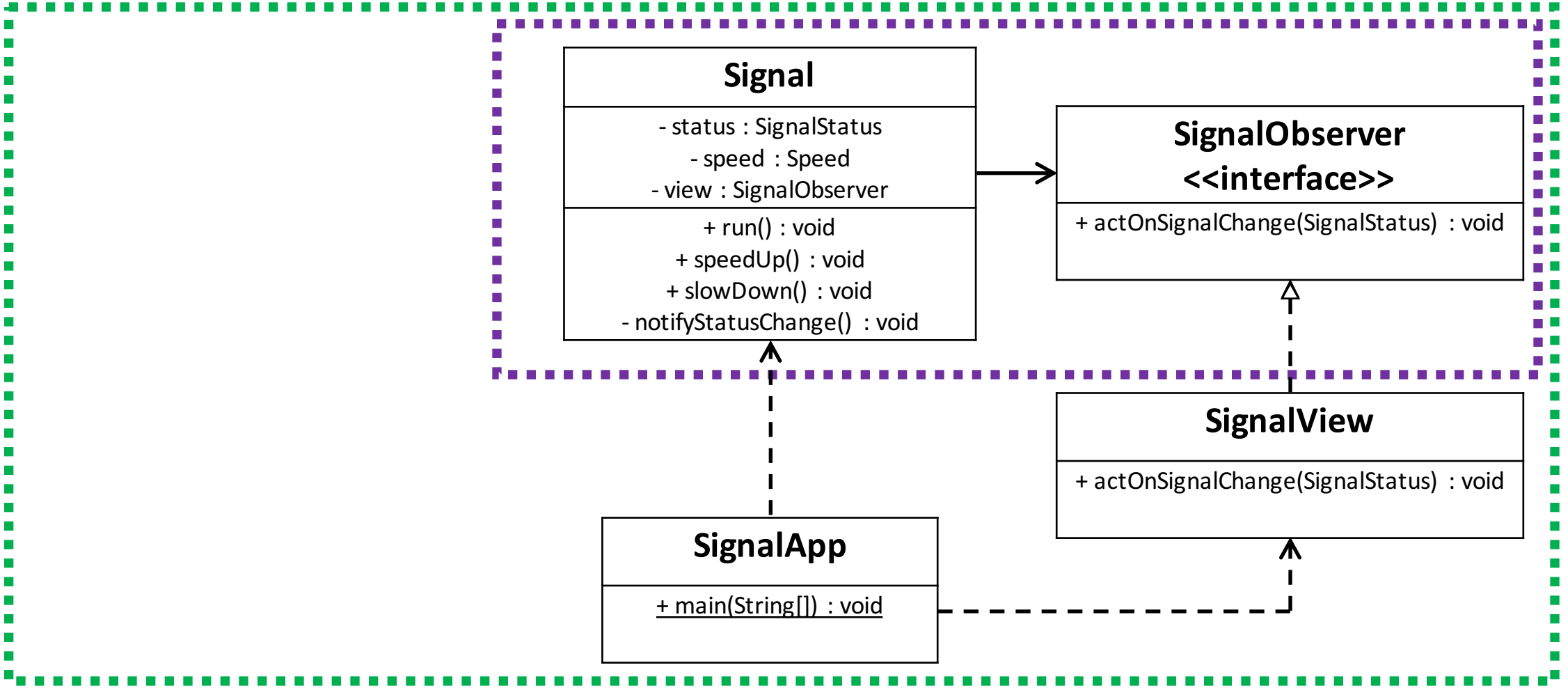
- Detta är vad vi strävar efter. Det är dock inte alltid helt självklart vad som bör ligga var (mer senare).

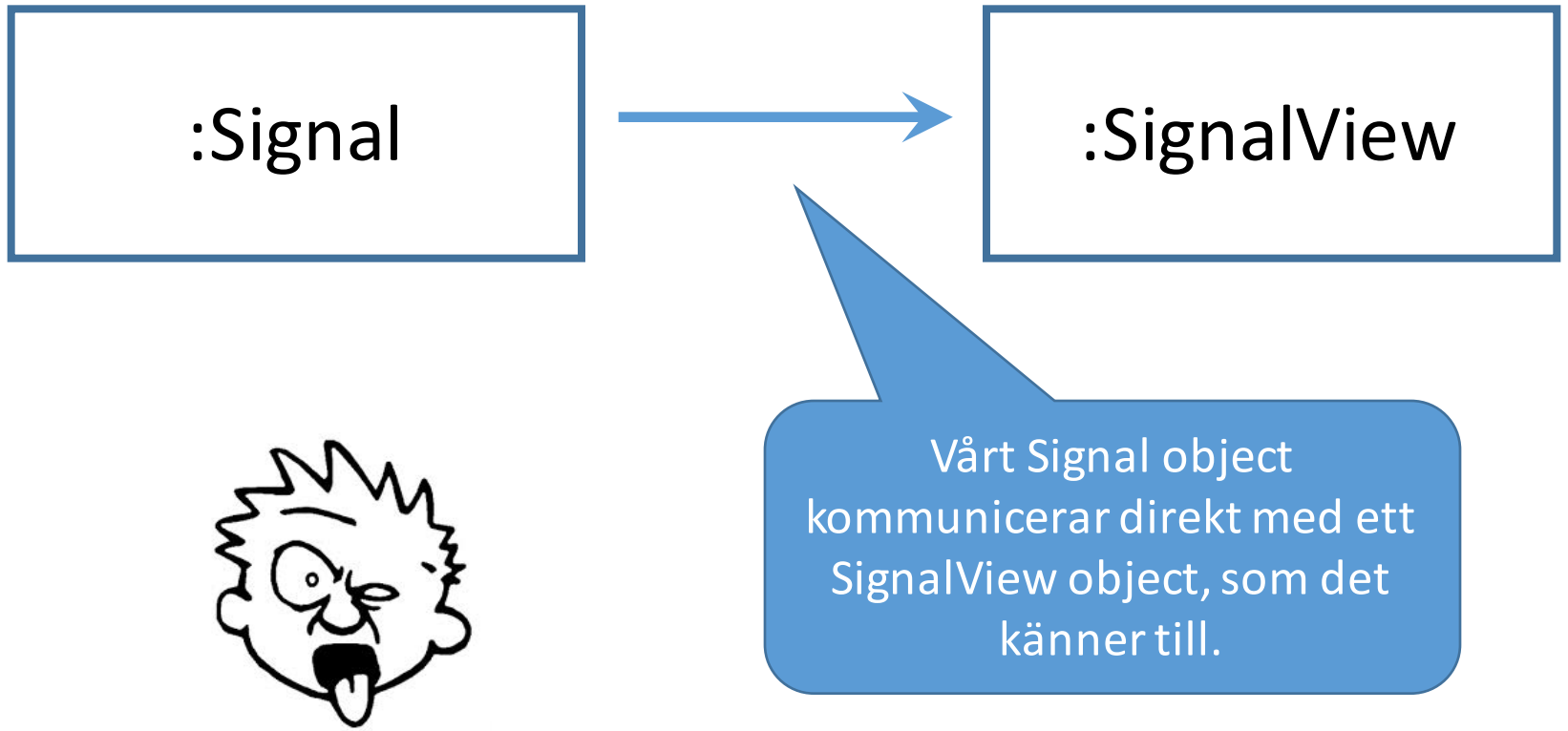
# Live code

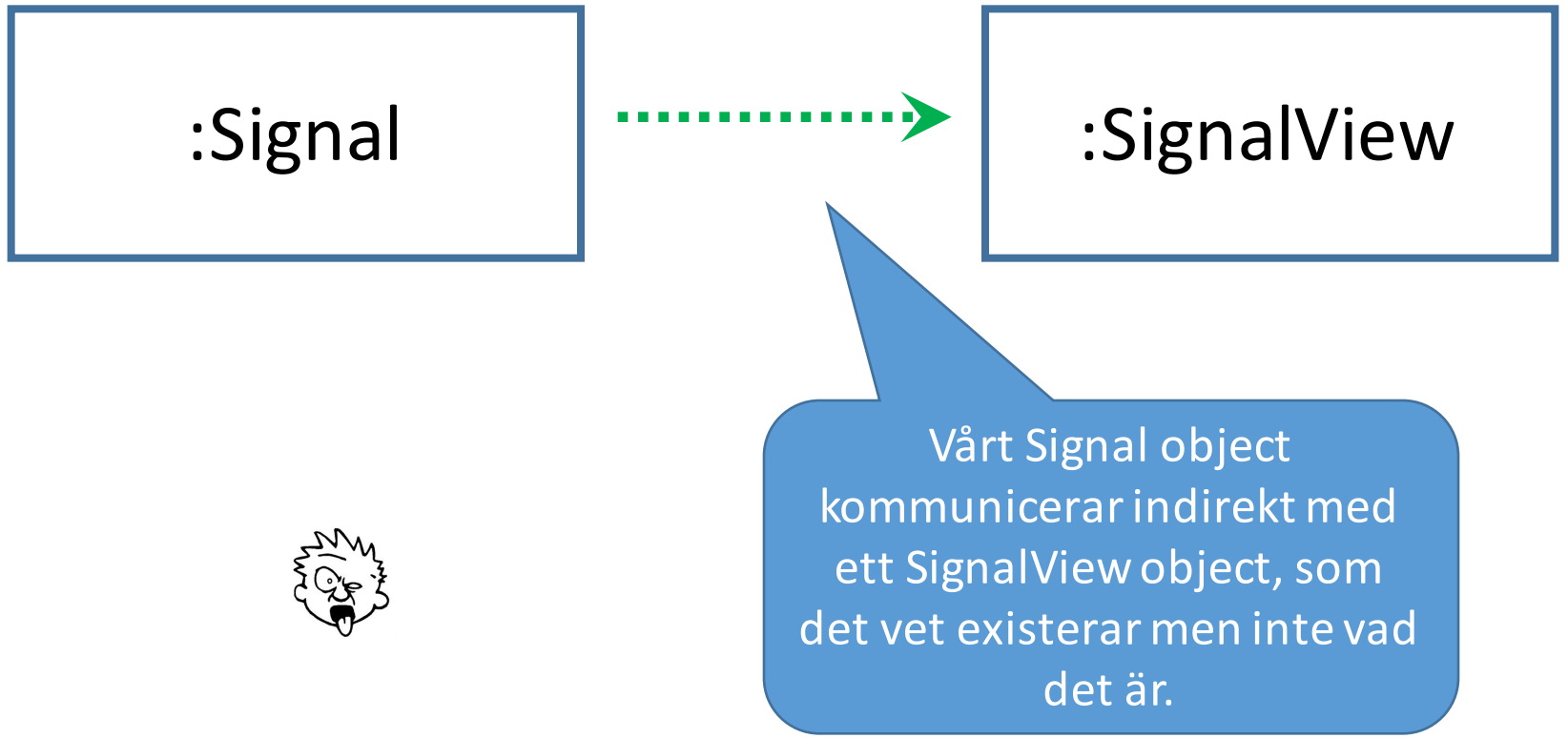
- `Signal`


JPanel
JFrame
Signal
- status : SignalStatus
- speed : Speed
- view : SignalView
+ run() : void
+ speedUp() : void
+ slowDown() : void
SignalView
+ update(SignalStatus) : void
Speed
<<enum>>
SUPERFAST
FAST
NORMAL
SLOW
SUPERSLOW
+ up(): Speed
+ down() : Speed
SignalApp
+ main(String[]) : void
SignalStatus
<<enum>>
RED
TOGREEN
GREEN
TORED
+ next(): SignalStatus


Signal
- status : SignalStatus
- speed : Speed
- view : SignalView
+ run() : void
+ speedUp() : void
+ slowDown() : void
SignalView
+ update(SignalStatus) : void
SignalApp
+ main(String[]) : void


Signal
- status : SignalStatus
- speed : Speed
- view : SignalObserver
+ run() : void
+ speedUp() : void
+ slowDown() : void
- notifyStatusChange() : void
SignalObserver
<<interface>>
+ actOnSignalChange(SignalStatus) : void
SignalView
+ actOnSignalChange(SignalStatus) : void
SignalApp
+ main(String[]) : void


:Signal
:SignalView
Vårt Signal object kommunicerar direkt med ett SignalView object, som det känner till.


:Signal
:SignalView
Vårt Signal object kommunicerar indirekt med ett SignalView object, som det vet existerar men inte vad det är.

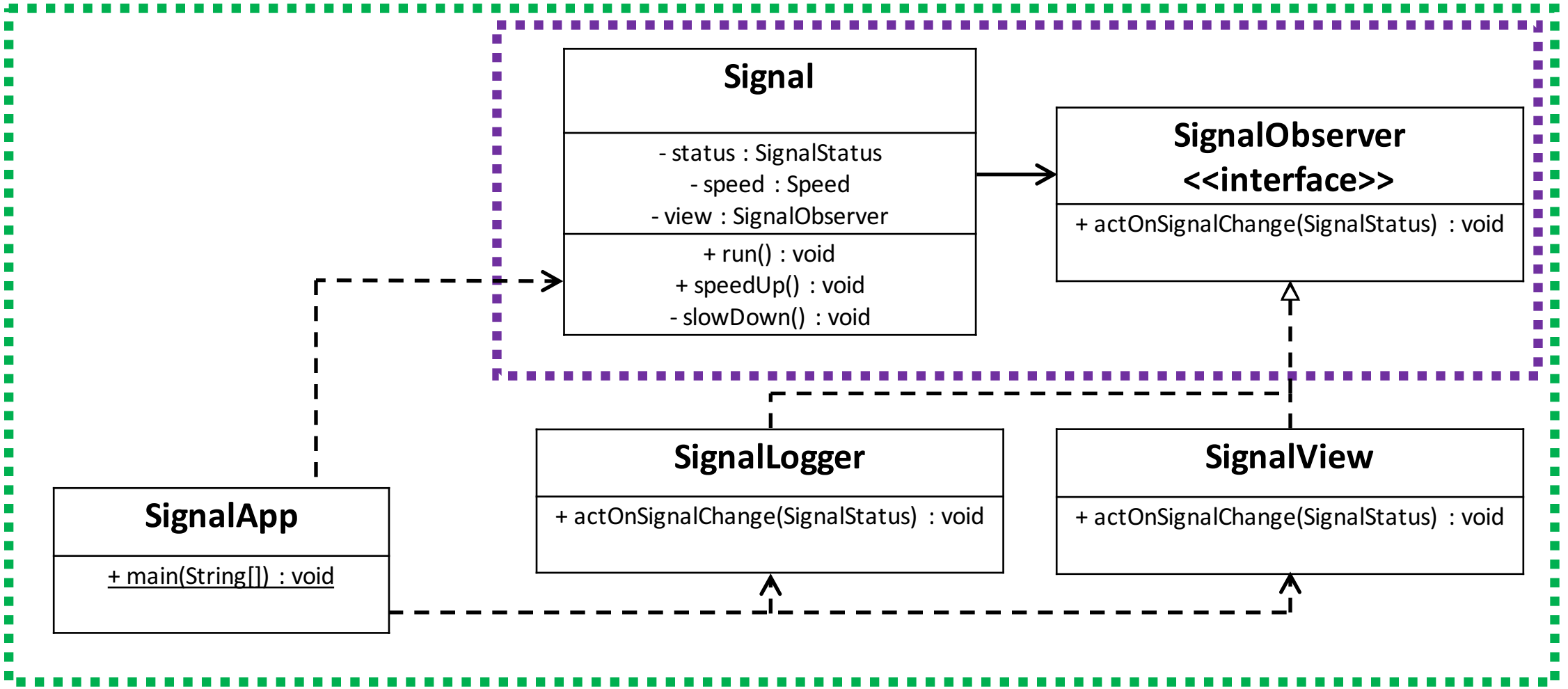
# Multicasting

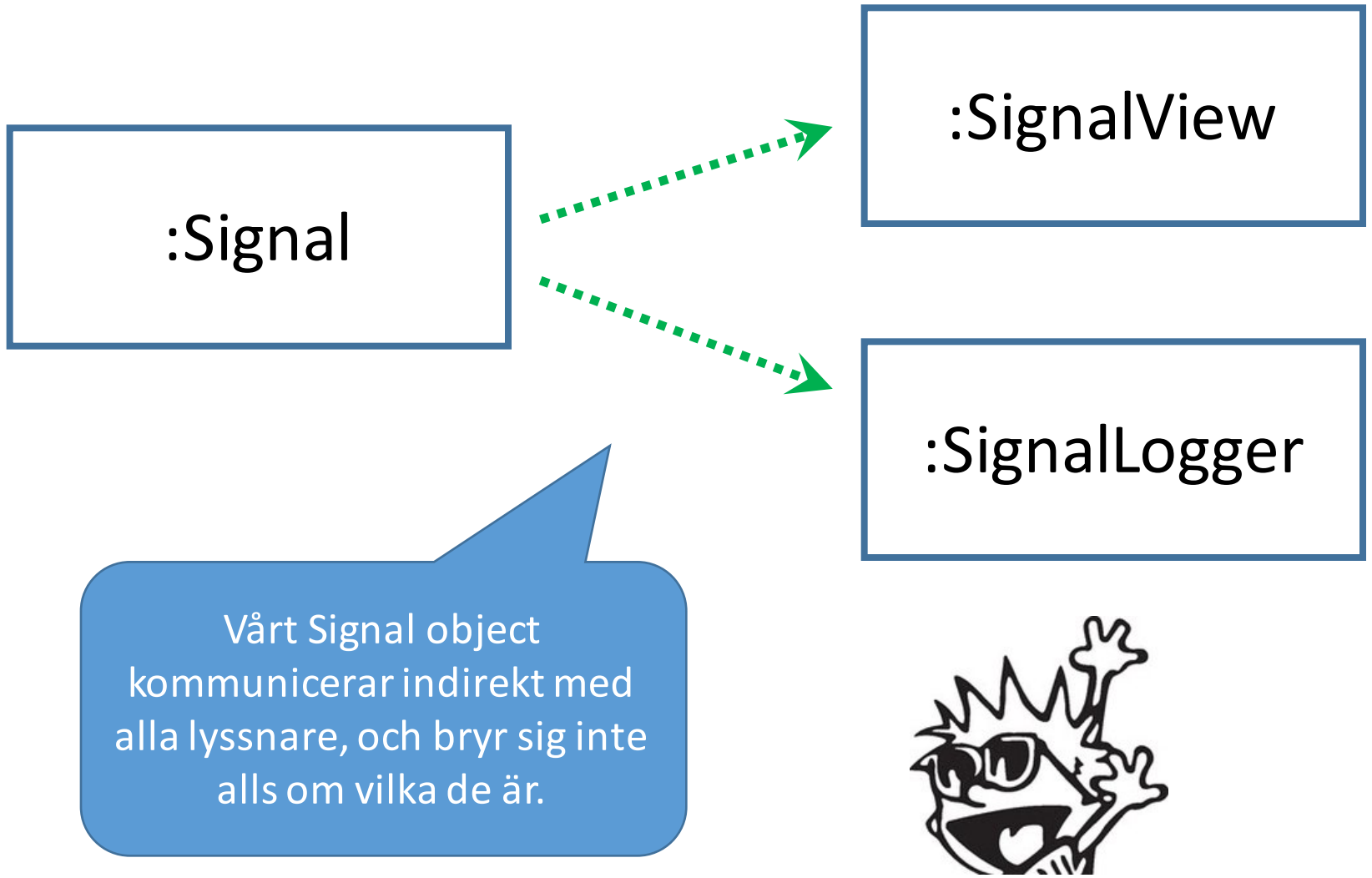
- Att tillåta många observers kallas ibland för *multicasting* – information om events skickas till ”multiple” klienter.

- Att möjliggöra multicasting per default är en bra princip.
  - Väldigt låg kostnad om det bara utnyttjas av en klient.
  - Följer automatiskt OCP: vi kan lätt lägga till fler klienter utan att modifiera koden.

# Live code

- `SignalLogger`


Signal
- status : SignalStatus
- speed : Speed
- view : SignalObserver
+ run() : void
+ speedUp() : void
- slowDown() : void
SignalObserver
<<interface>>
+ actOnSignalChange(SignalStatus) : void
SignalApp
+ main(String[]) : void
SignalLogger
+ actOnSignalChange(SignalStatus) : void
SignalView
+ actOnSignalChange(SignalStatus) : void


:Signal
:SignalView
:SignalLogger
Vårt Signal object kommunicerar indirekt med alla lyssnare, och bryr sig inte alls om vilka de är.

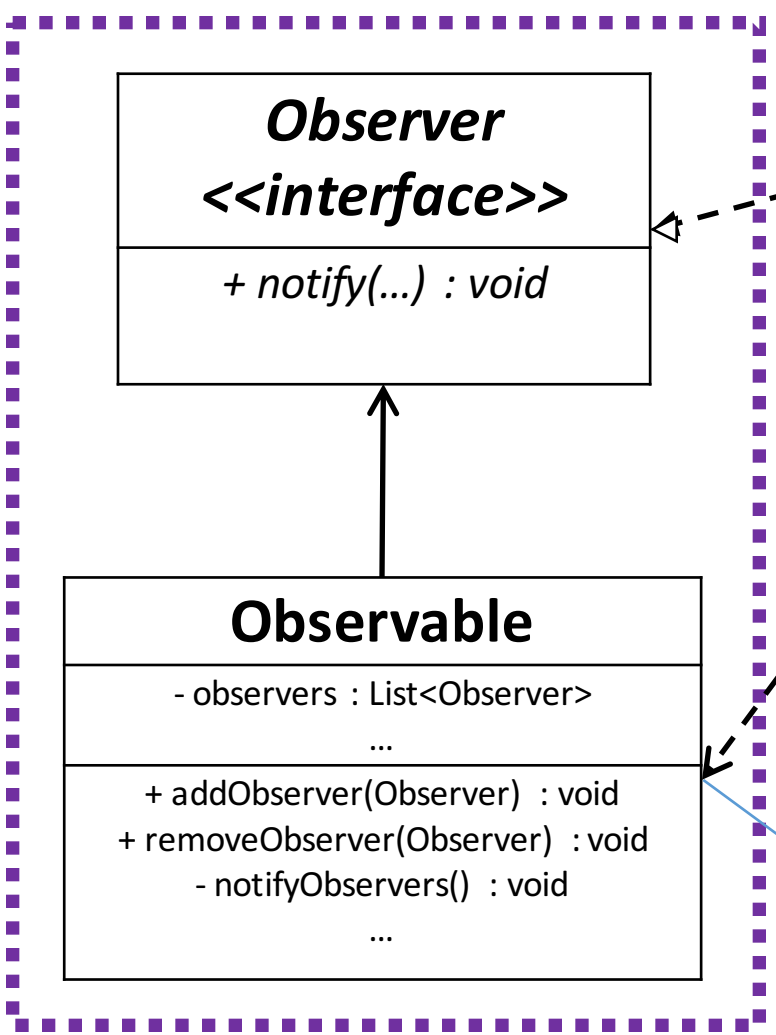
# Observer Pattern

When an object needs to notify other objects of events, without directly depending on them: broadcast the events on an open channel, to which any interested object may register.

- DIP: Definiera ett interface enligt vilket objektet kommer att broadcasta sina events, och låt vem som vill implementera detta interface och registrera sig för att lyssna.

- Den som lyssnar kallas *observer*; den som skickar events kallas *observable.*

# Observer Pattern




**ExternalClient**

```
class ExternalClient implements Observer {
  public ExternalClient(Observable t){
    t.addObserver(this);
  }

  public void notify(){
    // Do something clever on callback
  }
}
```

```
private void notifyObservers{
  for (Observer o : observers)
    o.notify;
}
```

# Observer Pattern i Java

- I Java finns interfacen `java.util.Observer` och `java.util.Observable`, med de metoder vi diskuterat. Dessa är dock så väldigt enkla att implementera själv, och vinsten med polymorfism mellan olika användningsfall är noll. Dessutom vill vi nästan alltid vara mer specifika, e.g.:
  - Vilka events flaggas för, med vilka argument?
  - Flaggas flera olika sorters events? Kan man registrera sig som observer för bara ett, eller alla?
  - Mer konkreta namn än e.g. `notify` på metoderna hjälper läsförståelsen.
- Slutsats: Definiera egna varianter för era observers och observables.

- (Java (i JavaBeans) har också java.beans.PropertyChangeListener m.m.. Dessa gör det möjligt att lyssna på ”property changes”, där en property är ett internt attribut med getters och setters. Fördelen här är att e.g. NetBeans (en IDE) kan automatiskt generera kod för Observer Pattern. Jag går inte in på det vidare.)

# Observer pattern i Swing

- Hela Java Swing är uppbyggt med hjälp av Observer Pattern; observers kallas *Event Listeners*.
- Alla grafiska komponenter – ”*widgets*” – vi ritar ut med Swing kan registrera och avfyra olika events, och tar emot observers för dessa.
  - E.g. ActionListener, MouseListener, MouseWheelListener, FocusListener, …

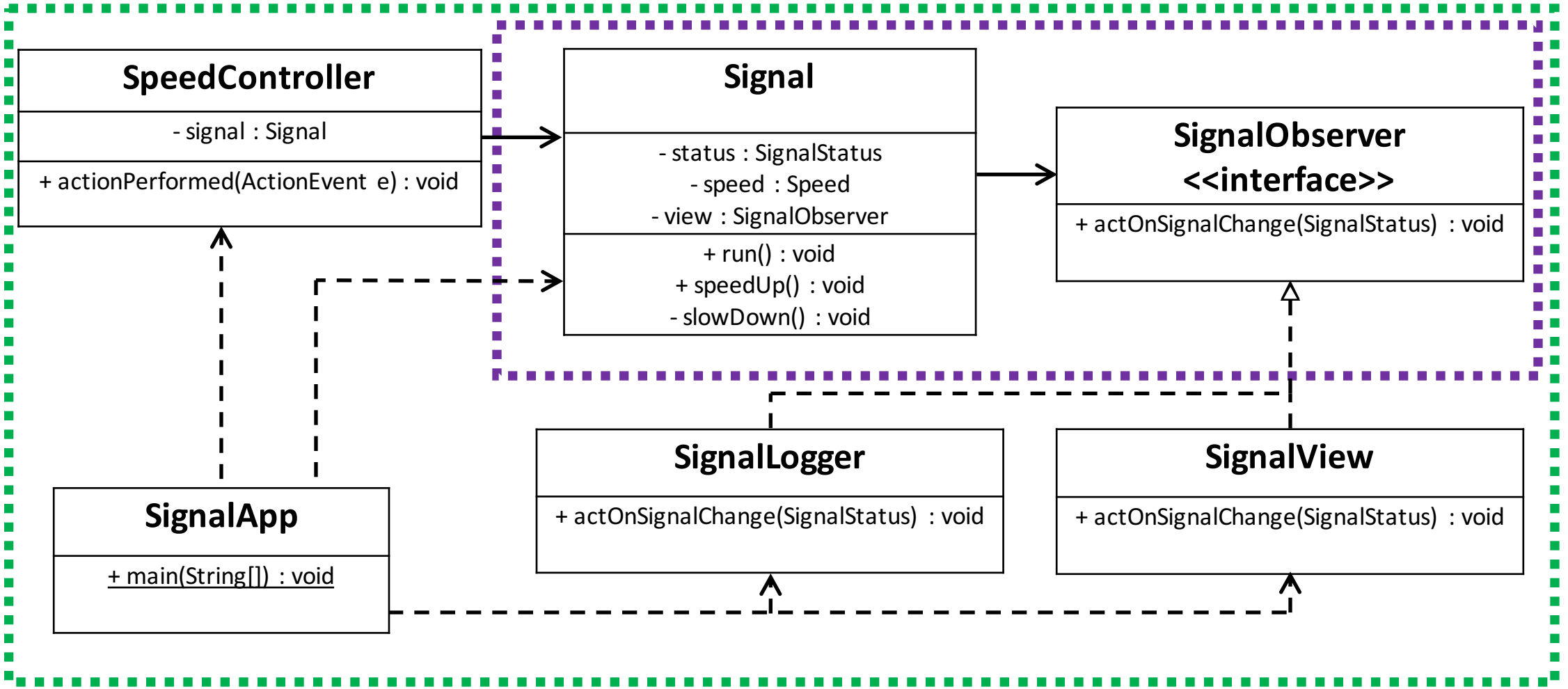
- Samma princip går igen i de flesta ”moderna” GUI-bibliotek.

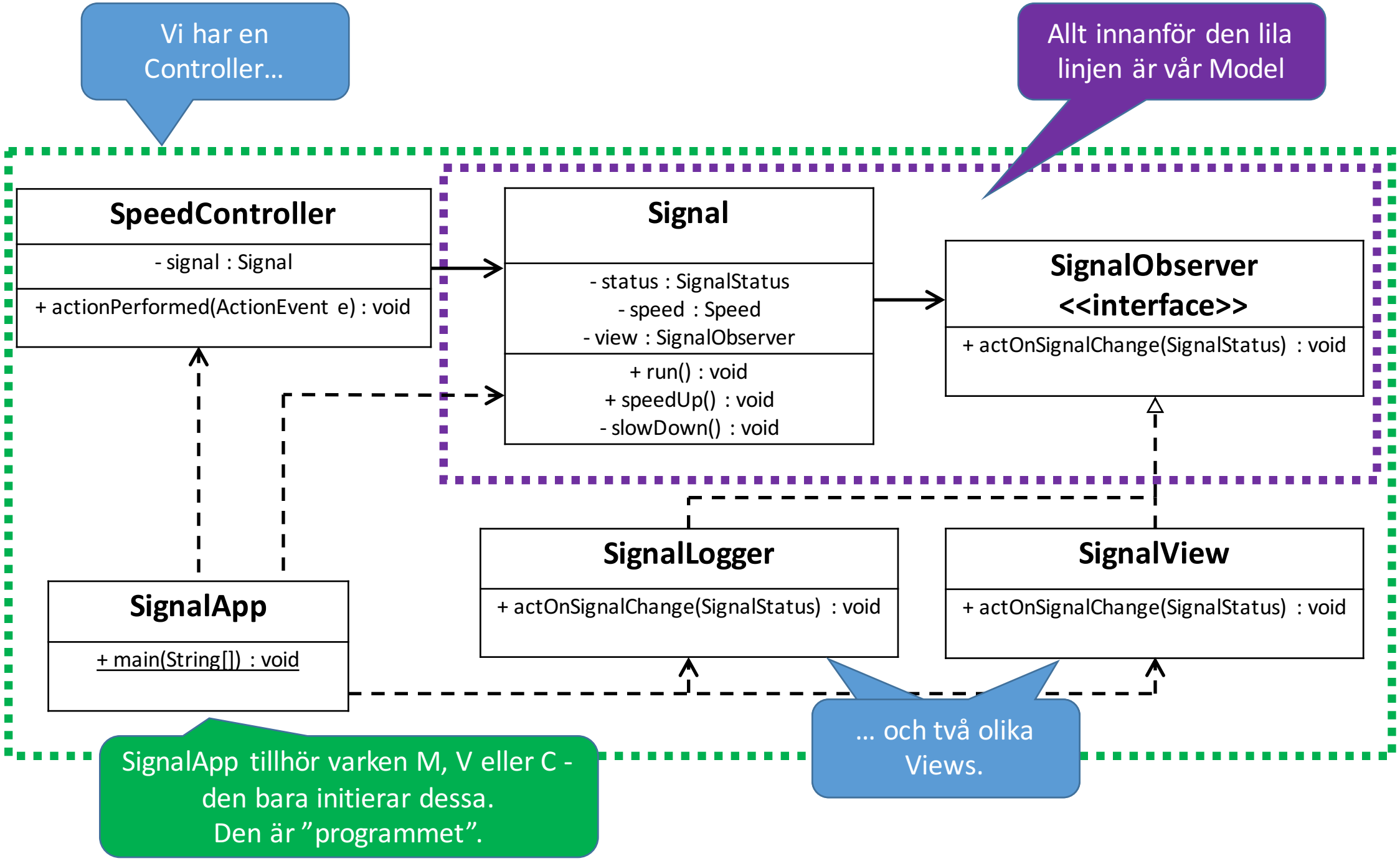
# Event-driven programming

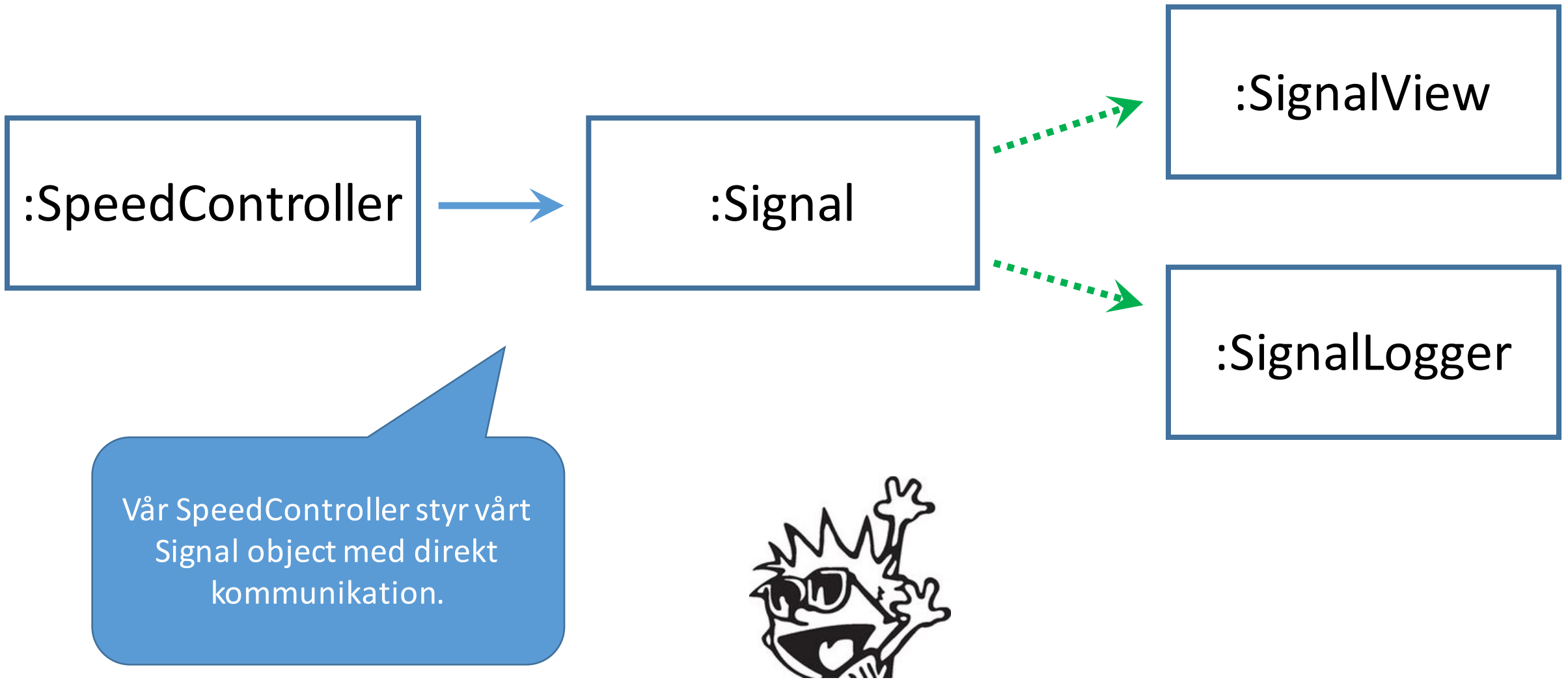
- Aside: *Event-driven programming* brukar kallas för en paradigm, dvs ett sätt att på topp-nivå strukturera program. GUI-bibliotek som Swing sägs vara instanser av event-driven programming.
- Vanligt vid låg-nivå-programmering, t ex operativsystem, där man hela tiden lyssnar efter olika *hardware interrupts*.
- I grunden går event-driven programming ut på att använda Observer Pattern för hela system.

# Live code

- `SpeedController`

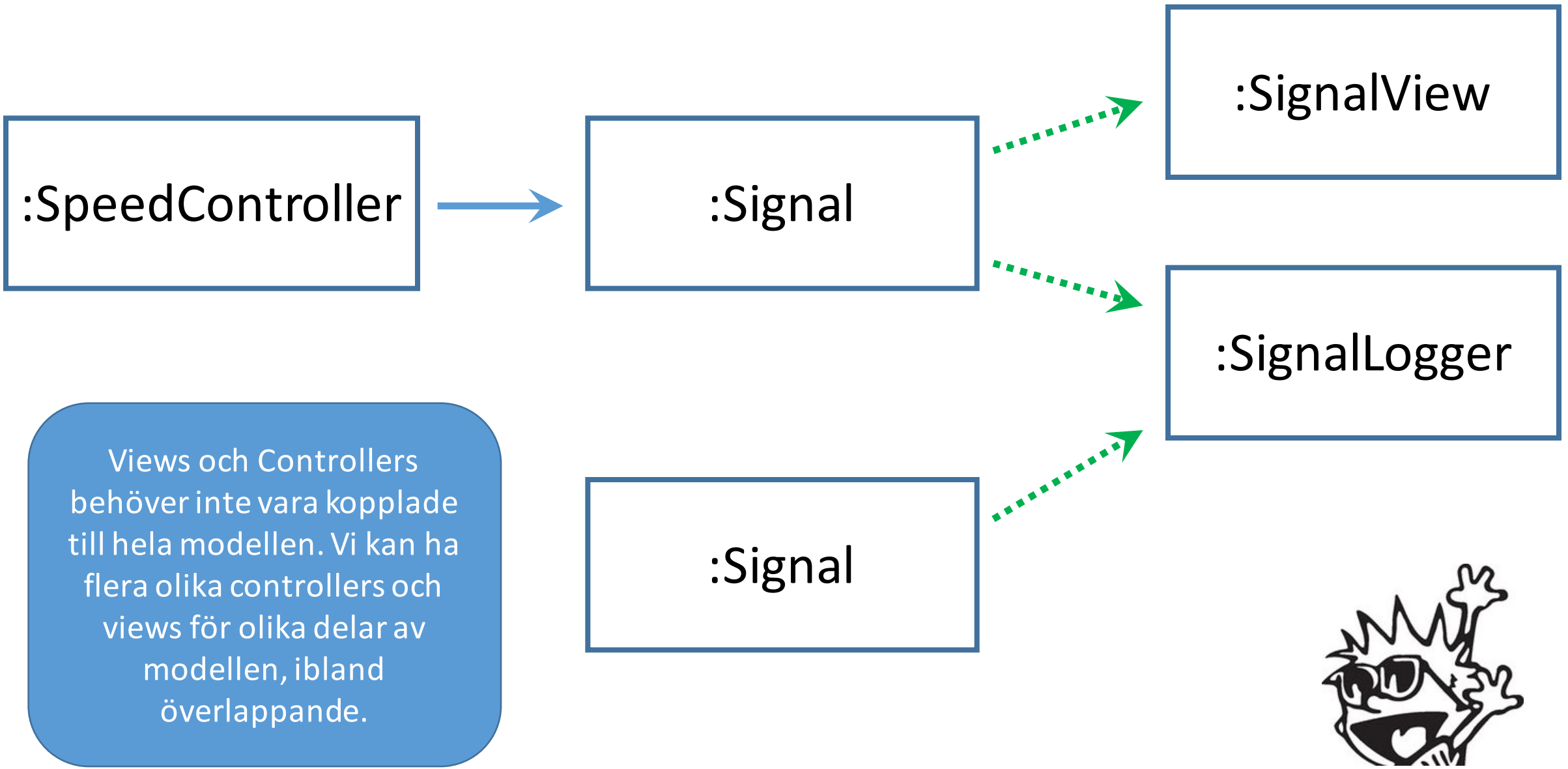

SpeedController
- signal : Signal
+ actionPerformed(ActionEvent e) : void
Signal
- status : SignalStatus
- speed : Speed
- view : SignalObserver
+ run() : void
+ speedUp() : void
- slowDown() : void
SignalObserver
<<interface>>
+ actOnSignalChange(SignalStatus) : void
SignalApp
+ main(String[]) : void
SignalLogger
+ actOnSignalChange(SignalStatus) : void
SignalView
+ actOnSignalChange(SignalStatus) : void


Vi har en Controller...
Allt innanför den lila linjen är vår Model
SpeedController
- signal : Signal
+ actionPerformed(ActionEvent e) : void
Signal
- status : SignalStatus
- speed : Speed
- view : SignalObserver
+ run() : void
+ speedUp() : void
- slowDown() : void
SignalObserver <<interface>>
+ actOnSignalChange(SignalStatus) : void
SignalApp
+ main(String[]) : void
SignalLogger
+ actOnSignalChange(SignalStatus) : void
SignalView
+ actOnSignalChange(SignalStatus) : void
SignalApp tillhör varken M, V eller C - den bara initierar dessa. Den är ”programmet”.
... och två olika Views.


:SpeedController
:Signal
:SignalView
:SignalLogger
Vår SpeedController styr vårt Signal object med direkt kommunikation.

# Live code

- Add second `Signal`


:SpeedController
:Signal
:Signal
:SignalView
:SignalLogger
Views och Controllers behöver inte vara kopplade till hela modellen. Vi kan ha flera olika controllers och views för olika delar av modellen, ibland överlappande.

# Quiz: Design pattern?

Define an object that encapsulates how other objects communicate.

- Definiera ett objekt som representerar och hanterar kommunikationen mellan andra objekt. Andra objekt kommunicerar aldrig direkt med varandra, utan bara via detta objekt.

# Mediator Pattern

Define an object that encapsulates how other objects communicate.

- Genom att definiera en Mediator som hanterar kommunikationen har vi åstadkommit lösare koppling (low coupling) mellan de kommunicerande objekten. Dessa kan nu varieras helt separat från varandra, och alla förändringar av dessa kan som mest påverka vår Mediator.

- (Vissa definierar Mediator Pattern snävare, och kräver att Mediator-objektet också känner till och aktivt styr de kommunicerande objekten (dvs inga observers). Jag använder den vidare beskrivningen.)

# Publish-Subscribe Pattern

Define an object that presents passive communication channels that other object may send and listen to.

- Publish-Subscribe Pattern (kallas även Pubsub) är en specifik form av Mediator Pattern, där Mediator-objektet är helt passivt, och inte bryr sig om vem som lyssnar. Det tillhandahåller ett gränssnitt för att skicka information på olika kanaler, och ett gränssnitt för att registrera sig för att lyssna (Observer Pattern) på olika kanaler, och skickar sen bara vidare inkommande meddelanden till alla registrerade lyssnare.

# Quiz: Vems är ansvaret?

- Antag att vi har en applikation som är en musikspelare.
  - I modellen håller vi reda på nuvarande inställning för volym, som en `int`.
  - Vi har en ”vy” som faktiskt spelar upp musiken (detta är fortfarande en View, även om den inte är grafisk utan ljud), som sätter volymen efter inställningen.
  - Vi har en annan vy som grafiskt visar volyminställningen för användaren, med knappar för att öka eller minska värdet på volymen.
  - Vi har en controller som hanterar input från volym-visarens knappar och uppdaterar modellen.
- Antag vidare att vi vill varna användaren när volymen är över 12, genom att e.g. göra bakgrunden på volymvisaren röd.
- **I vilken av de fyra komponenterna bör koden ligga** som kontrollerar om volymens värde överstiger 12, och agerar på detta?

# Model – View – Presenter

- Ett ”modernt” alternativ till (eller variant av) MVC kallas MVP, där:
  - Model (M) fungerar precis som i MVC, dvs data och domänlogik.
  - View (V) gör det som V i MVC gör, men vi tar också hänsyn till att mycket av hanteringen av användar-input faktiskt sker via widgets, så (i vissa varianter) även mycket av det C i MVC gör (de bitar som direkt uppdaterar modellen).
  - Presenter (P) sköter all *presentations-* och *applikationslogik*. Den tar emot en del input från V (kan ev välja själv via Observer Pattern vad den vill lyssna på), och står för mer avancerade beräkningar och uppdateringar i modellen. Den är också ansvarig för alla beräkningar som styr hur presentationen ska ske.
  - I en specifik variant – kallad *Passive View MVP* – går all kommunikation genom P, och V och M kommunicerar aldrig med varandra.

# What’s next

## Block 6-1:
## State och Immutability