

Observer Pattern och MVC

Objekt-orienterad programmering och design

Alex Gerdes, 2017

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).



Controller:
Interaktion med
saker i världen

Controller:
Interagerar med
karaktärer

Fler controllers

Controller: Får
saker att hända



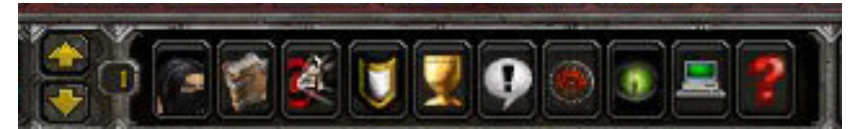
Var går gränsen?

- Ponera en grafisk "widget" som...
 - ... visar information från modellen.
 - ... går att klicka på så att saker händer.
- Controller eller vy?
- Svar: Som regel lite av båda (men går att göra undantag):
 - Själva widgeten (e.g. JPanel) är rimligen (en del av koden för) en vy.
 - Widgeten/vyn kan hantera att och hur användaren interagerar med den – men inte vad som ska hända som effekt av detta.
 - Koden som avgör vad som ska hända hör till en controller.



Var går gränsen?

- Ponera en grafisk "widget" som...
 - ... *inte* visar information från modellen.
 - ... går att klicka på så att saker händer.
- Controller eller vy?
- Svar: Man kan resonera olika – men är etiketterna verkligen viktiga?
 - Det finns inget som säger att en Controller inte får ha en grafisk representation. Är detta då en vy som visar upp kontrollern (istället för modellen som brukligt)? Eller är det en del av kontrollern?
 - Who cares! Det viktiga är att förstå principerna för att separera koden för de olika delarna.



Pilarna är ren input –
påverkar vad som visas i vyn
bredvid.

Applikation

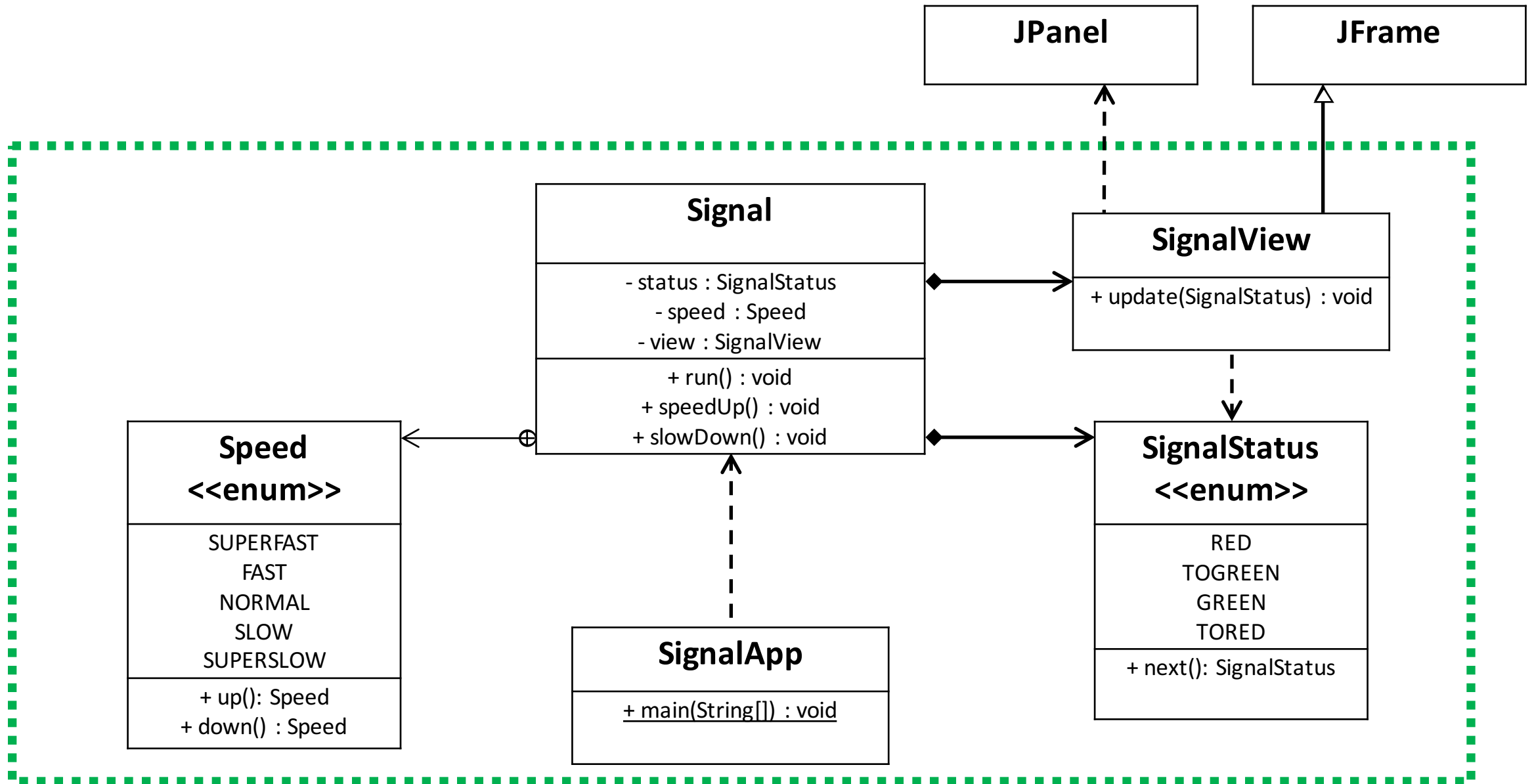
- Alla applikationer har en "main"-metod som går att exekvera. Denna hör inte till någon av MVC, utan ligger separat ("A").
 - Applikationen skapar, sammanför, och startar de olika komponenter som tillsammans ska utföra jobbet.
 - Flera olika applikationer kan använda sig av samma sorts komponenter, e.g. polygoner.
 - Kan vara mer eller mindre avancerad, beroende på hur mycket som behöver konfigureras.
 - Fönsterhanteringen på toppnivå kan ligga här.
 - All data som är applikationsspecifik kan ligga här.
- (Ibland (som undantag) är det relevant att lägga viss applikationslogik här, istället för i M.
 - E.g. I en simulering, hör "tiden" hemma i modellen, eller kan olika applikationer välja att hantera tiden olika?)

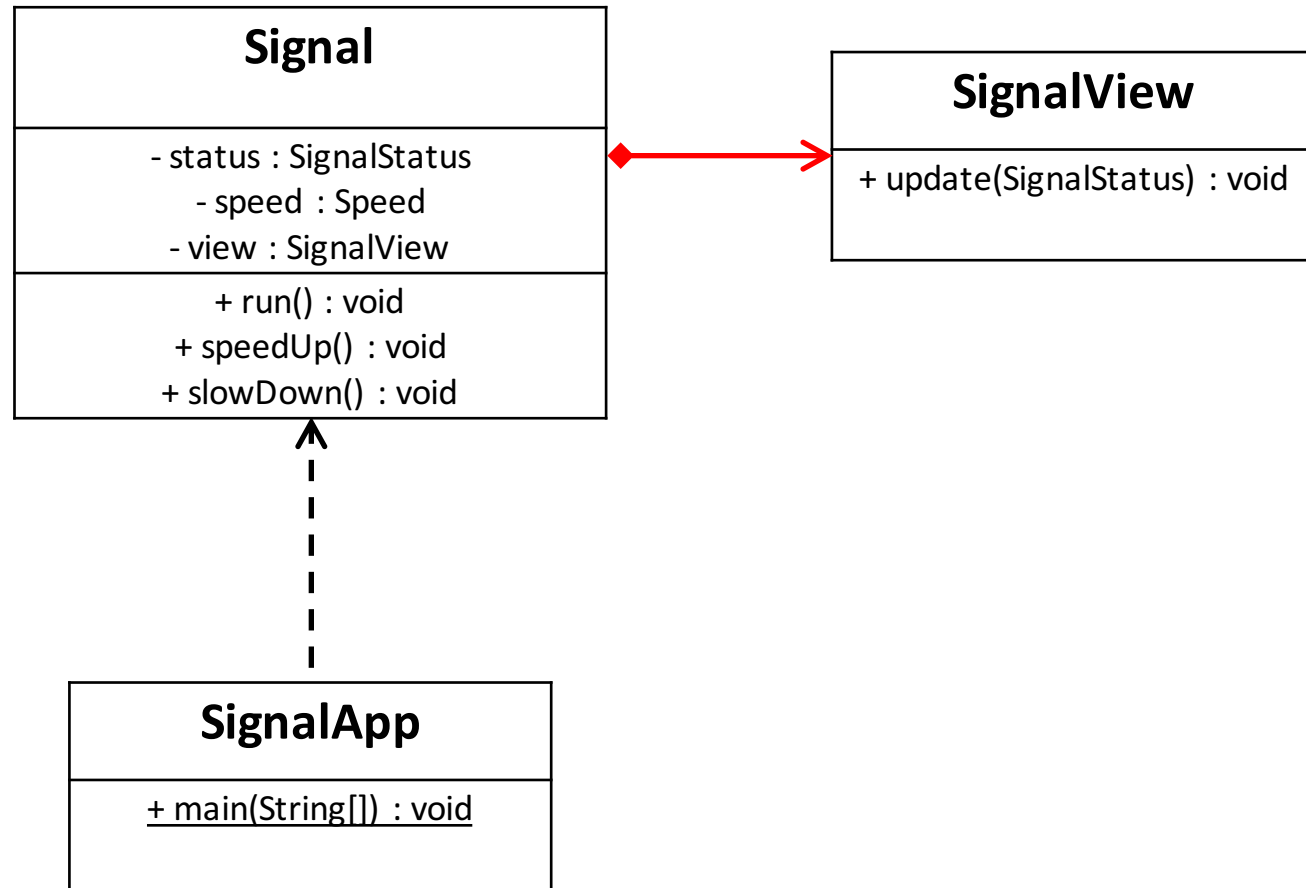
Hur bör M, V och C (och A) hänga ihop?

- Vilka beroenden bör vi ha?
 - M får inte bero av V eller C. Punkt.
 - "Världen" är vad den är, oavsett hur den presenteras eller styrs.
 - Får V bero av M? Får V bero av C? Får C bero av M och/eller V?
 - Svar: Det beror på... Olika varianter kan vara relevanta i olika sammanhang. Lyssna inte på de som säger sig ha hittat "den enda sanningen".
 - Tumregel: Gör det som leder till högst cohesion och lägst coupling!
 - A väljer vilka delar av M, V och C som ska utgöra programmet.
 - Slutsats: A beror nästan alltid på samtliga delar.
 - Om inte – har du verkligen rätt uppdelning av M, V och C?
 - A representerar ett topp-nivå-program – inget ska någonsin bero på A!

Live code

- `Signal`



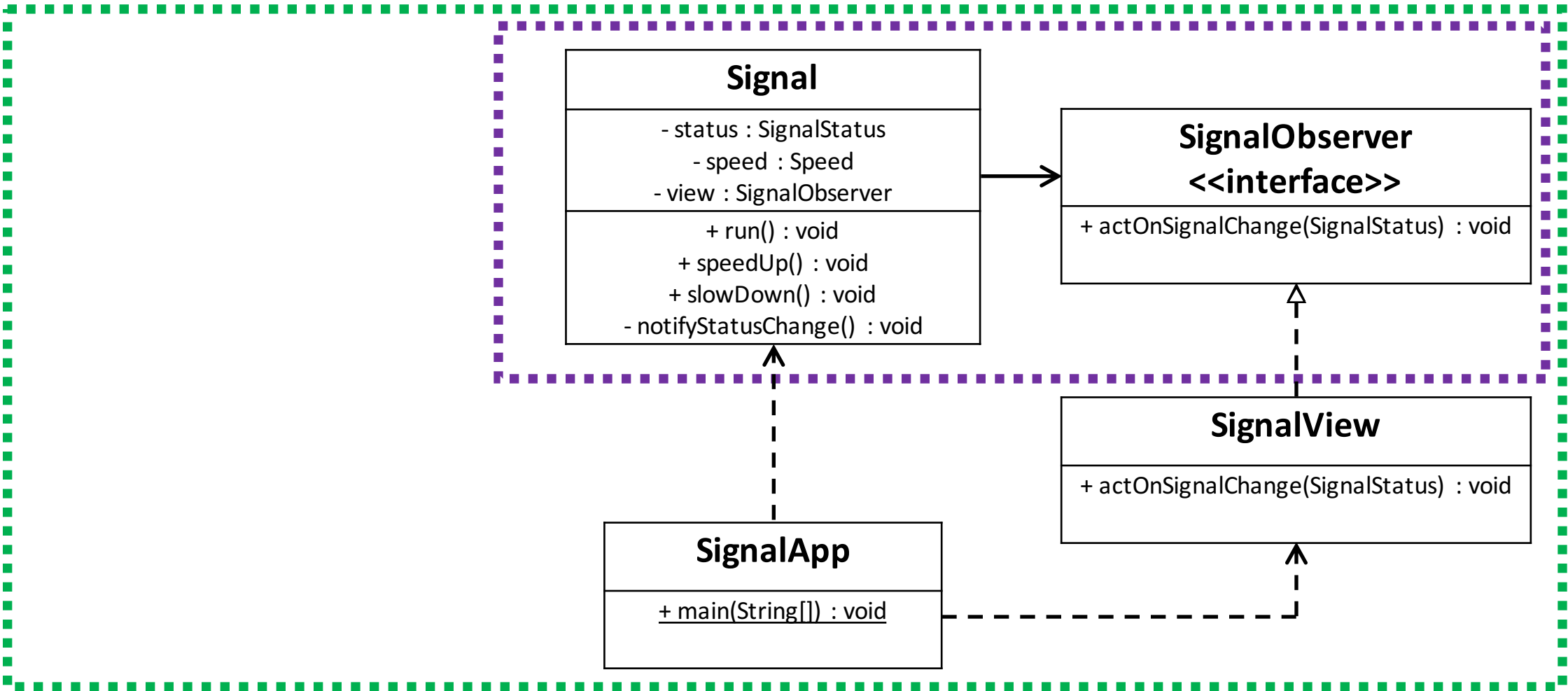


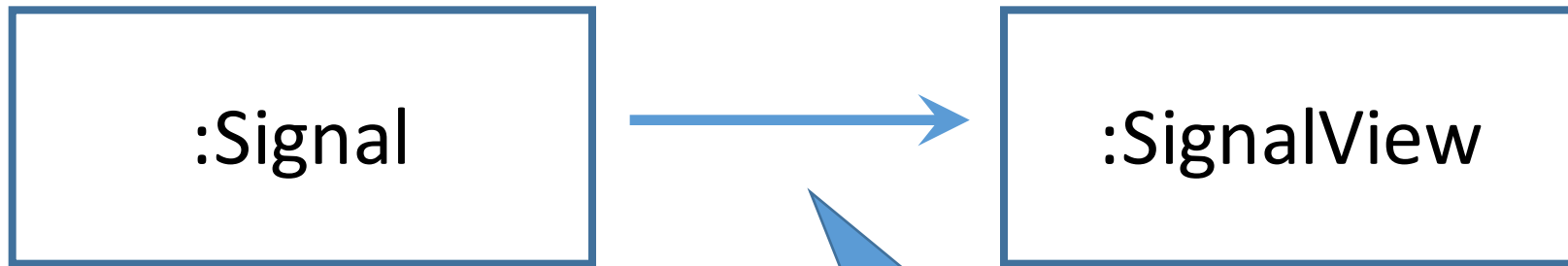
Hur vänder vi på beroenden?

- Vi har ett dilemma. Om M inte får bero av V – hur ska då V få veta när den ska uppdateras?
- Alternativ 1: V frågar M regelbundet ("polling"), och ritar ut status.
 - Fungerar bra om M uppdateras ofta och mycket. Ex realtidsspel.
 - Fungerar dåligt om M uppdateras sällan eller lite. Ex ritprogram.
- Alternativ 2: C uppdaterar både M och V samtidigt.
 - Fungerar bra om M enbart uppdateras på grund av användarinput. Ex ritprogram.
 - Fungerar dåligt om M uppdateras oberoende av användaren. Ex realtidsspel.
- Alternativ 3: M är smart, och meddelar alla förändringar högt, utan att veta vem (om någon) som lyssnar.
 - OCP–med hjälp av abstraktion.

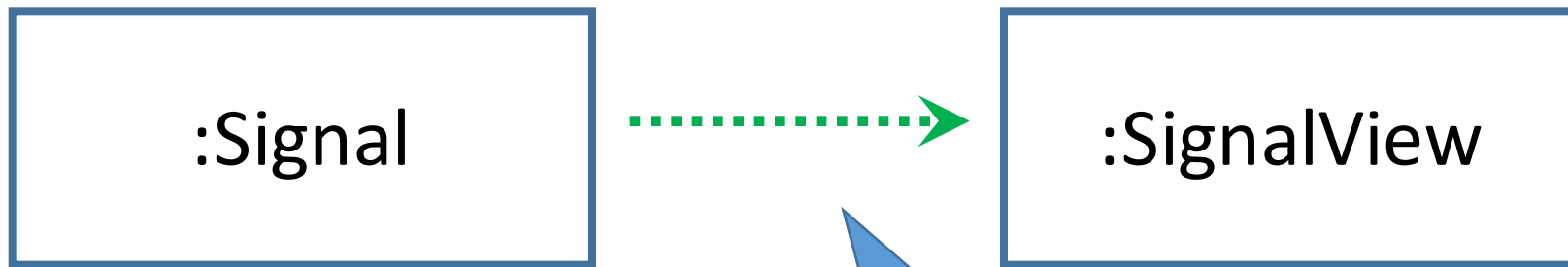
Live code

- `Signal`





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



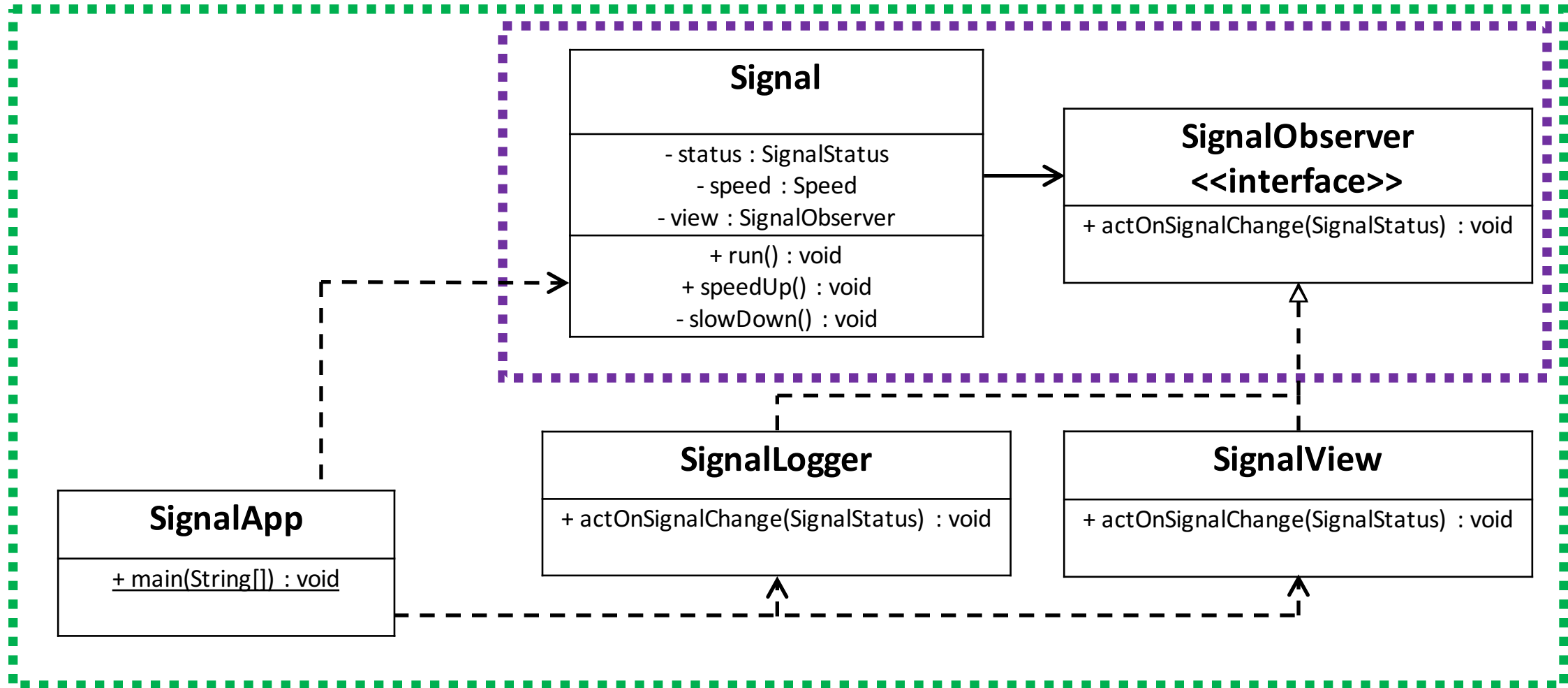
Vårt Signal object
kommunicerar indirekt med
ett SignalView object, som
det vet existerar men inte vad
det är.

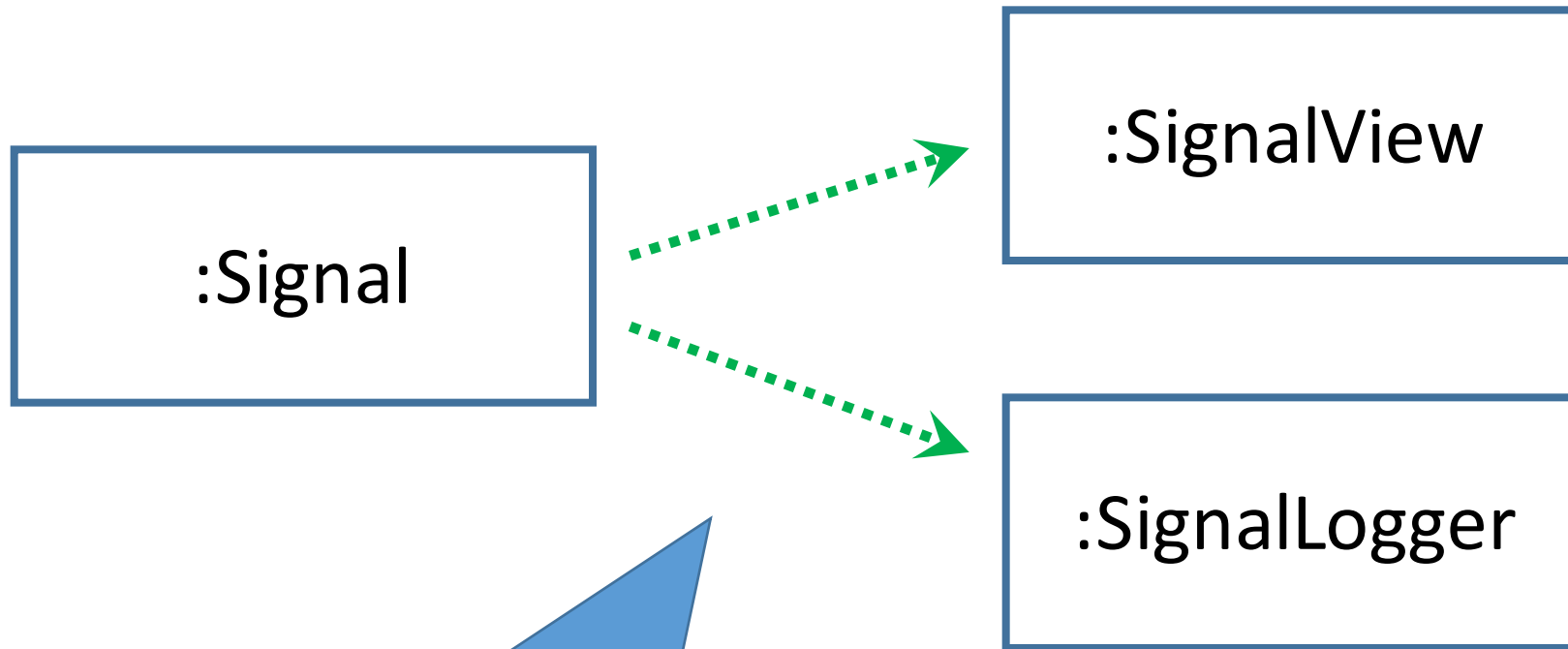
Multicasting

- 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`





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

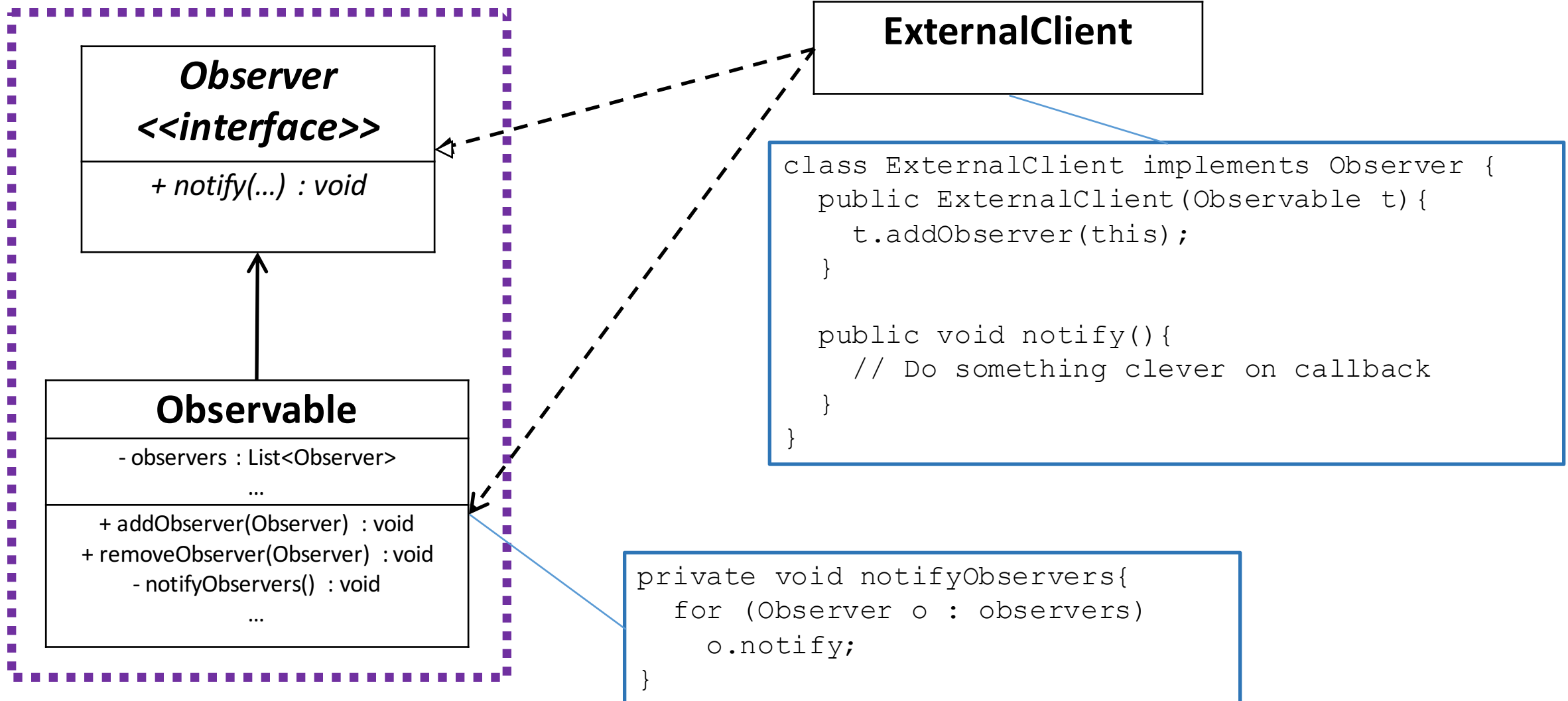


Observer Pattern

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



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.

Observer pattern i Swing

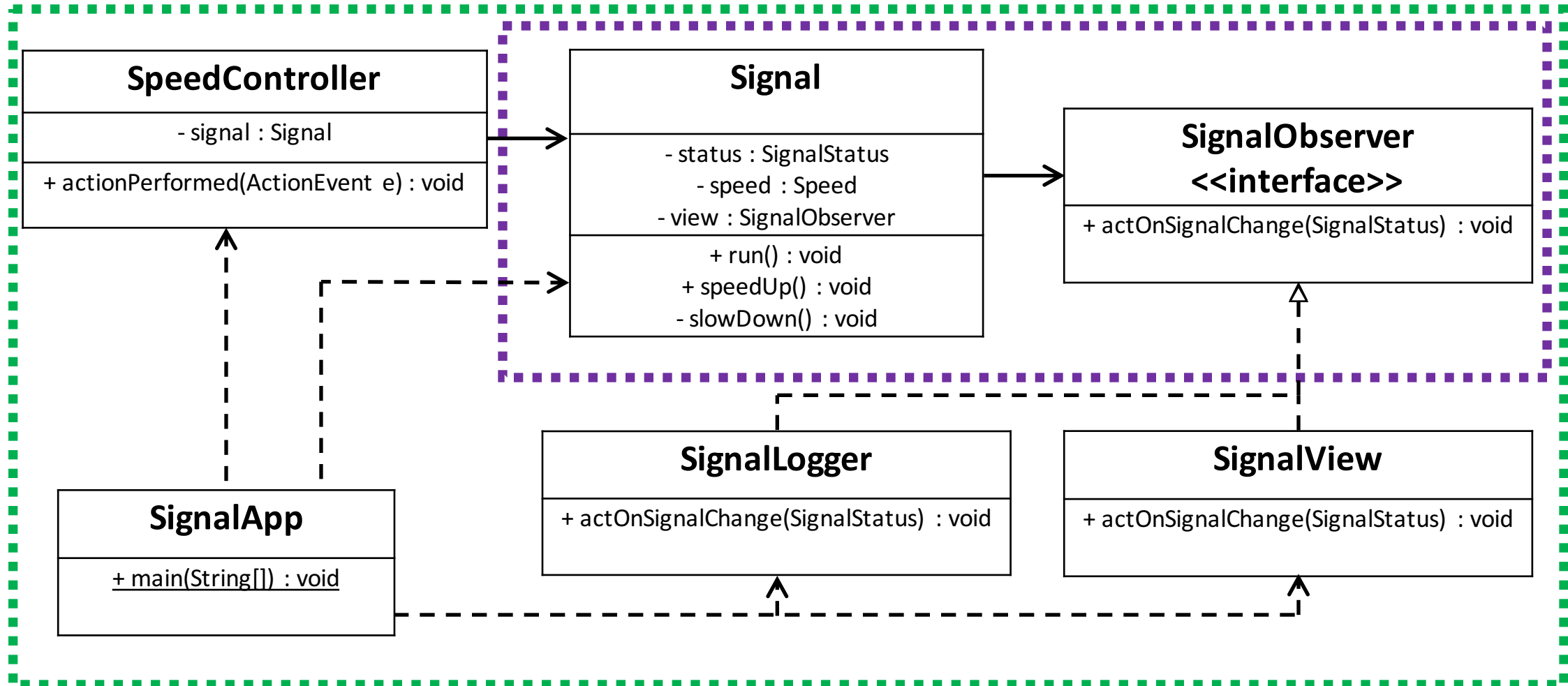
- Hela Java Swing är uppbyggt med hjälp av Observer Pattern; observers kallas *Event Listeners*.
- Alla grafiska komponenter – ”*widgets*” – vi ritar ut med Swing kan registrera och avfyra olika events, och tar emot observers för dessa.
 - E.g. ActionListener, MouseListener, MouseWheelListener, FocusListener,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

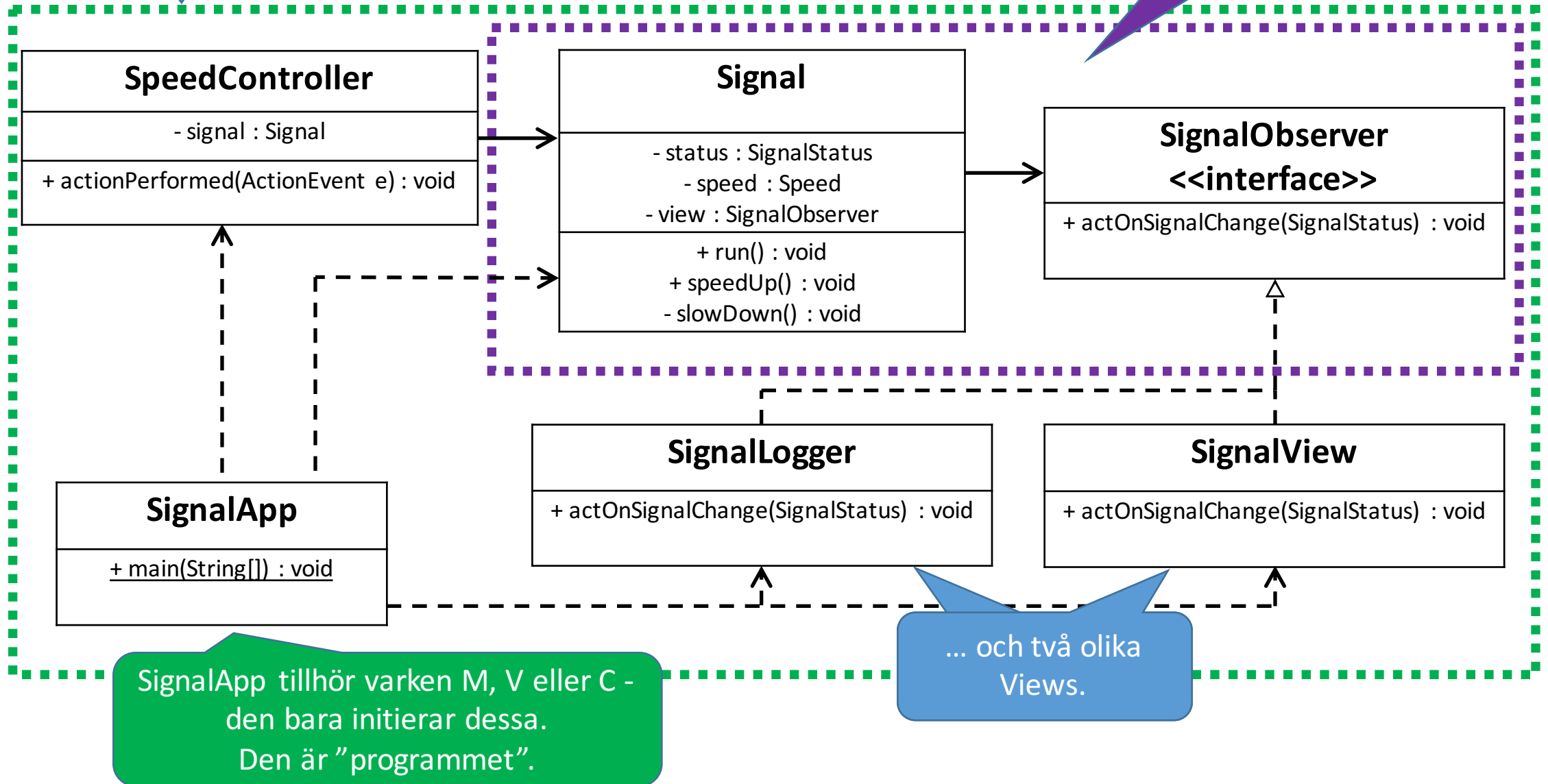
Live code

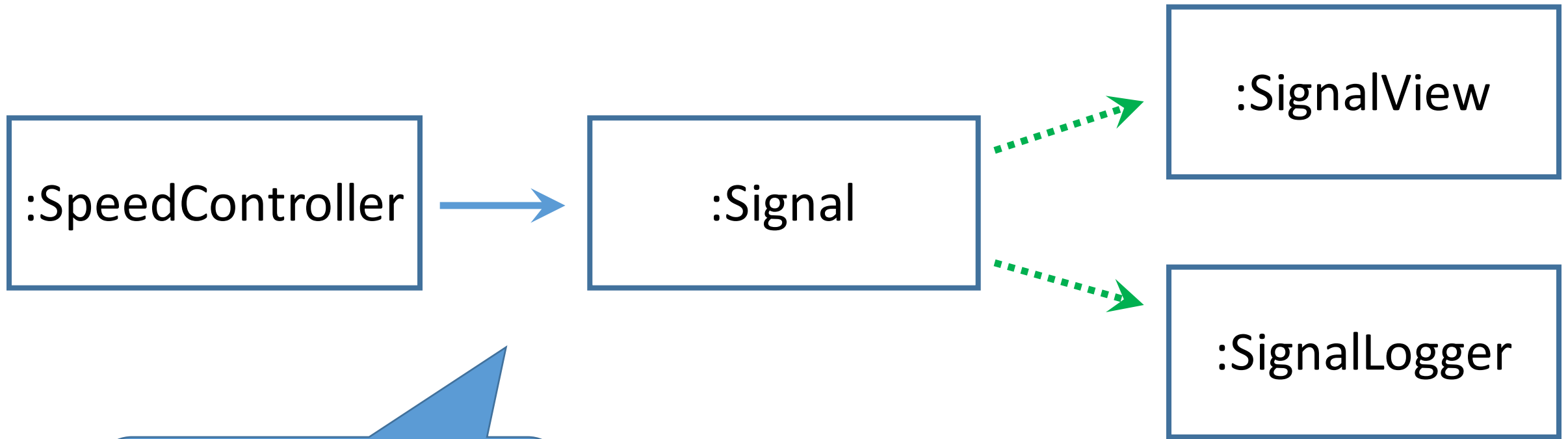
- `SpeedController`



Vi har en Controller...

Allt innanför den lila linjen är vår Model



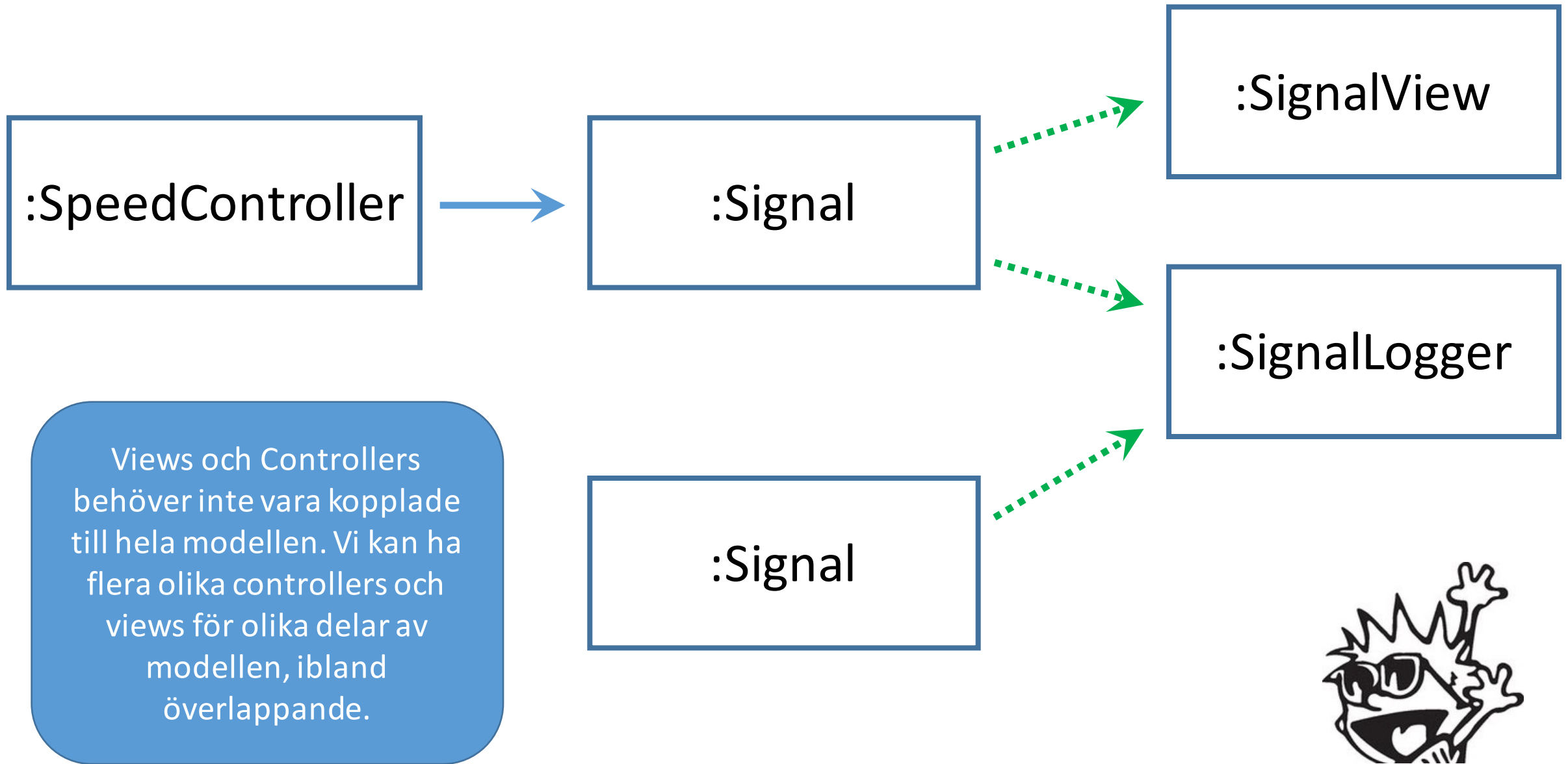


Vår SpeedController styr vårt
Signal object med direkt
kommunikation.



Live code

- Add second `Signal`



Översikt SignalApp

- En modell (M) som inte beror på något annat – men som är kodad med Observer Pattern (OCP!), i förväntningen att det kan komma komponenter som vill veta när den uppdateras.
- Flera olika vyer (V) som beror av M.
 - Beror enbart på SignalStatus (A lägger till V som SignalObserver)
- En controller (C) som enbart beror av M – men som har en grafisk widget.
 - OBS: SpeedController kan med fördel delas upp i e.g. SpeedControlWidget (som är en dum frame med knappar) och SpeedController (som lägger till sig som observer hos SpeedControlWidget och hanterar knapptryckningarna). (Hem-uppgift)
 - Vi kan lätt lägga till fler controllers, som beror (enbart) på de delar av M och/eller V som de uppdaterar, eller triggas av.
- En applikation (A) som skapar, sammanför och startar alla komponenter.

Slutsatser

- Förstå principerna för varför MVC ser ut som det gör.
 - Extensibility, reusability, maintainability.
 - Förstå vad som leder till reduktion av beroenden.
- Förstå Observer Pattern och hur det hjälper till att minska beroenden.
 - Av samtliga designmönster vi kommer gå igenom är Observer Pattern det viktigaste att förstå.
- Arbeta med den struktur som ger högst cohesion och lägst coupling.
 - Vilka beroenden som finns mellan M, V och C kan variera – gör ett medvetet och initierat val!
 - ... förutom att M aldrig ska bero på V och C. Punkt.

What's next

Block 6-1:
State och Immutability

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.
- Svar: Mediator Pattern

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?

Om ni svarade...

- **Model:** Siffran "12" för volymen säger ingenting i domänen, olika ljud-"vyer" kan välja att tolka siffran på olika sätt.
- **View (volymvisare):** Vi strävar efter DUMB views, så inga beräkningar bör göras här.
- **View (ljud):** Vyer ska vara oberoende av varandra, så denna vy kan inte kontrollera den andra.
- **Controller:** Vi strävar efter THIN controllers, som bara hanterar input och uppdaterar modellen därefter. Beslutet att sätta volymen över 12 kanske inte ens kommer ifrån användaren, eller via en annan controller, eller...

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.