

C++

Forelesning 8, vår 2014
Alfred Bratterud

Agenda

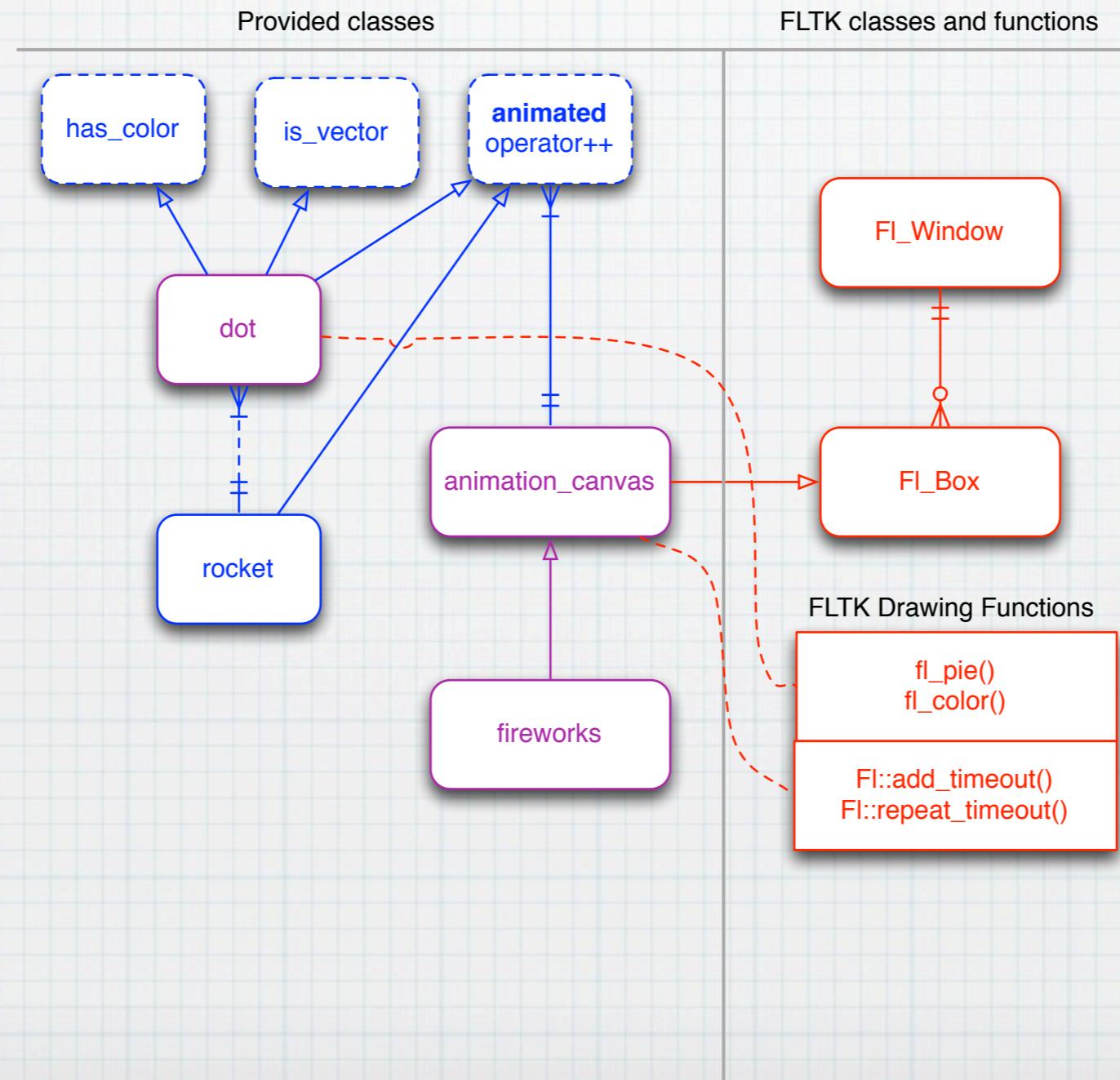
- * Oblig2
- * Et par ting som er greit å vite
- * Templates

Oblig2: Fyrverkeri!

...og et lite animasjonsrammeverk

Fireworks klassehierarki

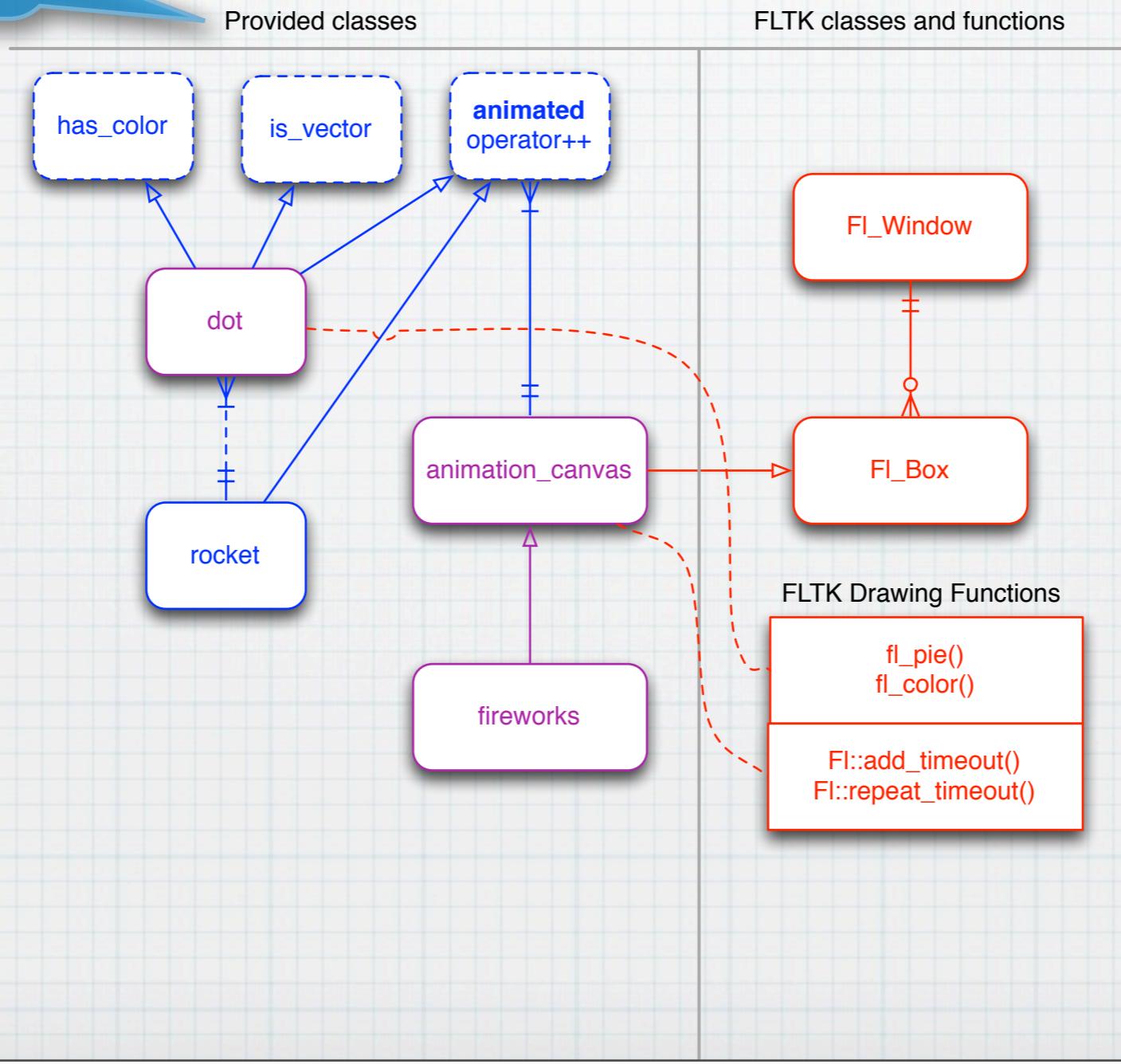
Fireworks class hierarchy



Fireworks klassehierarki

1) Disse skal
implementeres!

Fireworks class hierarchy

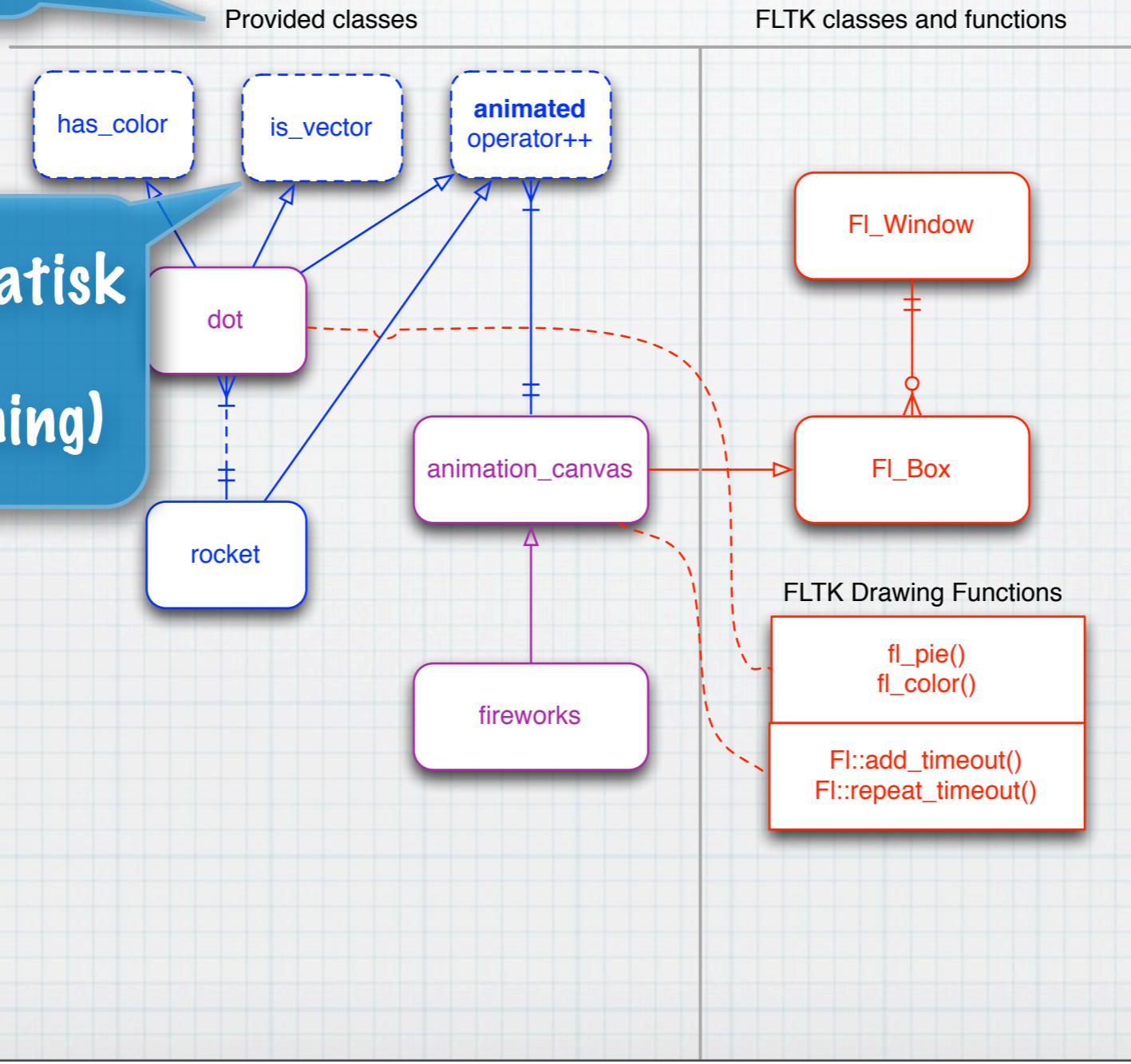


Fireworks klassehierarki

1) Disse skal
implementeres!

“vector” i matematisk
betydning!
(Har fart og retning)

Fireworks class hierarchy



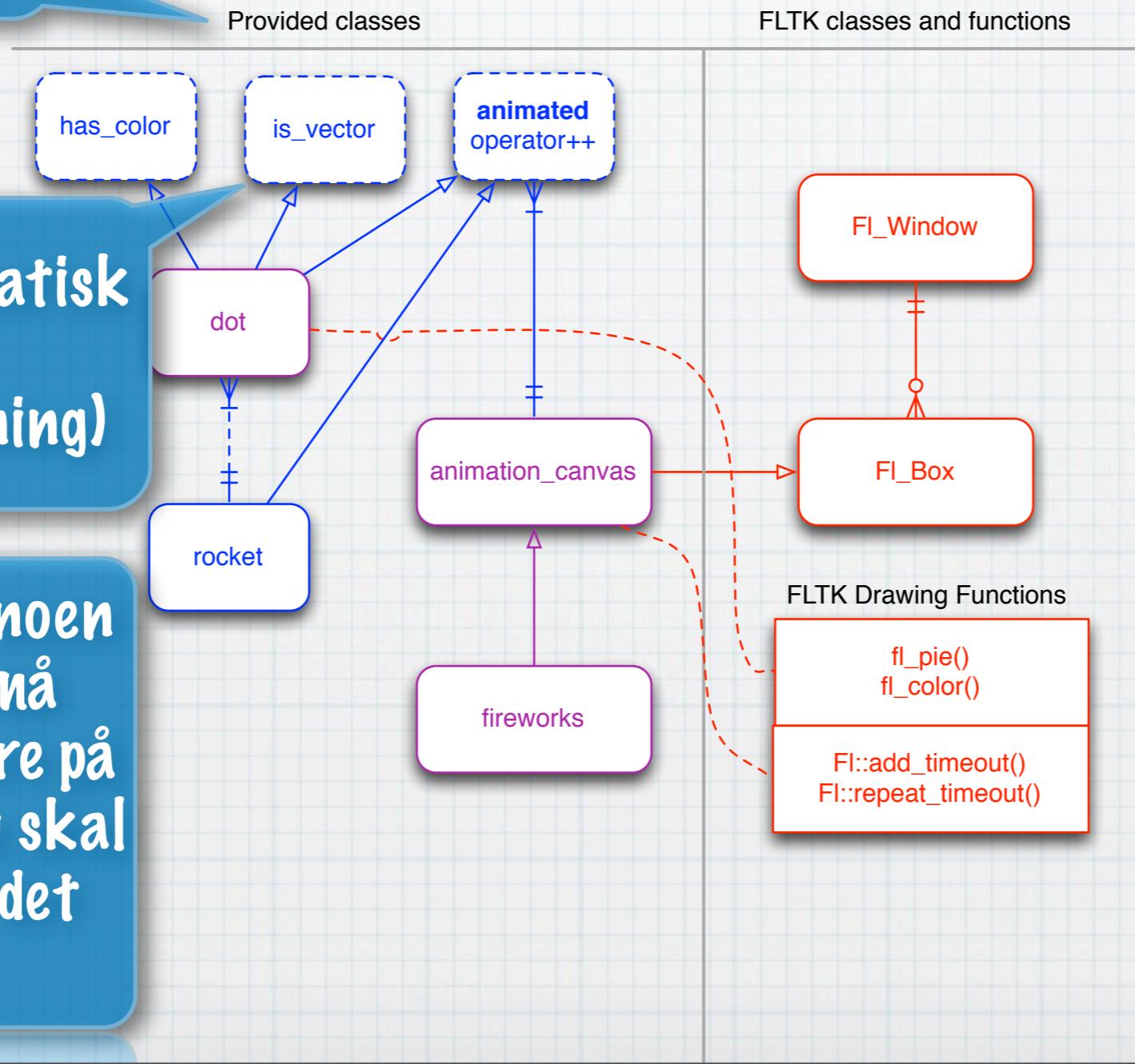
Fireworks klassehierarki

1) Disse skal
implementeres!

“vector” i matematisk
betydning!
(Har fart og retning)

2) Designet har noen
svakheter. Du må
finne dem og svare på
noen spørsmål. Du skal
ikke reparere - det
funker.

Fireworks class hierarchy



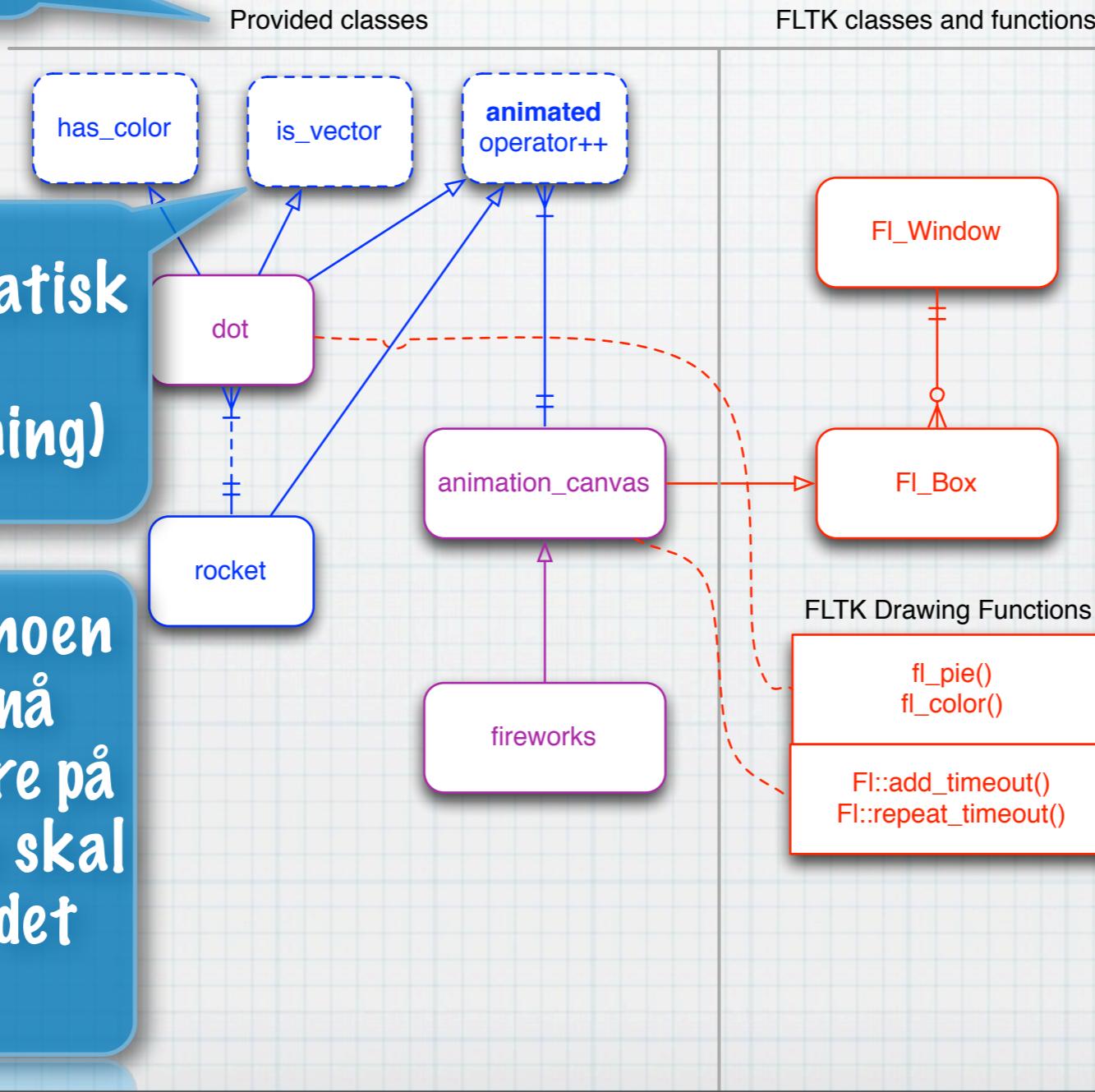
Fireworks klassehierarki

1) Disse skal implementeres!

“vector” i matematisk betydning!
(Har fart og retning)

2) Designet har noen svakheter. Du må finne dem og svare på noen spørsmål. Du skal ikke reparere - det funker.

Fireworks class hierarchy



3) I tillegg: Din egen animasjon, med “animated” og “animation_canvas”

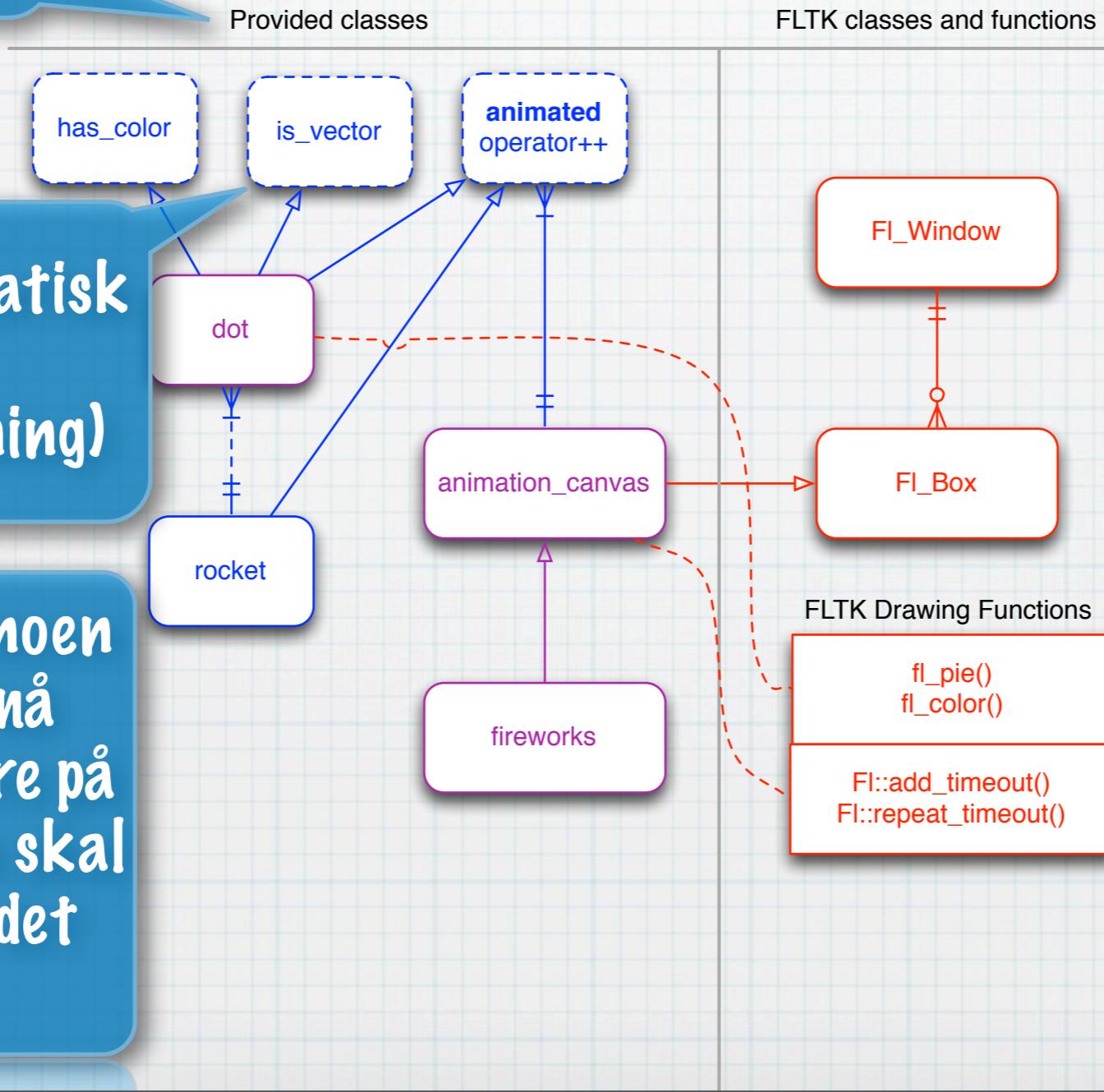
Fireworks klassehierarki

1) Disse skal implementeres!

“vector” i matematisk betydning!
(Har fart og retning)

2) Designet har noen svakheter. Du må finne dem og svare på noen spørsmål. Du skal ikke reparere - det funker.

Fireworks class hierarchy



3) I tillegg: Din egen animasjon, med “animated” og “animation_canvas”

LYKKE TIL!

Oblig2

Ting som er greit å vite

Virtuelle destruktører

- * Generelt sett bør klasser som har virtuelle funksjoner også ha virtuelle destruktører (særlig hvis klassen inneholder medlemmer som er pekere)
- * Virtuelle destruktører skal hete det samme som subklassen, selv om de overrider en som har navn etter baseklassen
- * Selv om den virtuelle destrukturen er i en abstrakt klasse, skal den allikevel ha kropp. Den kan da bare ha tom kropp.

Overriding av private medlemsfunksjoner

- * At en funksjon er privat betyr at den ikke kan kalles fra en subklasse
- * Men, hvis den er virtuell kan den allikevel overrides!
- * Når er dette nyttig?
 - * Når vi ønsker å bruke abstrakte klasser som "interface" i java-betydning
- * En privat funksjon kan overrides som public (!)
 - * Hva skjer da?
 - * Den kan kalles eksternt via en peker til subklassen
 - * ...men er fremdeles privat via en peker til baseklassen.

Makefile og namespace

- * Du skal lage en makefile for prosjektet
- * ...og *skrive kort* hvordan du ville lagt til namespace
(ikke gjøre det - la header-filene være)
- * Makefila skal kompile alle klassene for seg
- * Alt skal linkes sammen til en binary "fireworks"
- * Dynamisk linking er lov: at klassene bygges til et "shared library" som kan skiftes ut uten å rekompile (OBS: Makefila må ikke endre miljøvariable!)
- * Men statisk linking, som slår sammen alt til en binary, er helt greit (mao: ingen ekstra poeng for dynamisk linking)

Demo:

`virtual_destructor.cpp`
`override_private.cpp`

Templates

Generisk programmering

Generic Programming

- * Motivasjon: Vi ønsker å skrive en funksjon "print(...)", som skriver ut argumentet
- * Hva er argumentet i et strengt typet språk?
 - * Int? Heksadesimal? Eller desimal?
 - * Char? Eller bare en byte?
 - * String? Eller et char-array?
- * For hver type trenger vi da en egen print-funksjon
- * ...Med mindre vi har noe generisk

Templates

- * En template er en “midlertidig type”
- * For funksjoner, eller for klasser
- * Det avgjøres “compile time” hvilken type som vil brukes
- * For hver type som bruker funksjonen/klassen, lages en kopi

Generisk print()

- * Vi begynner med å definere en generisk type:
`template<class myType>`
`void generic_function(myType t)`
- * Dette er å betrakte som en ny type, `myType` i funksjonsdefinisjonen:

```
template<class myType>
void print(myType t){
    cout << t << endl;
}
```

- * Hva skjer nå: `print(5)`, `print('a')`, `print("øy")`?

Generisk print()

- * `template<class myType>`
`void print(myType t){`
 `cout << t << endl;`
}
- * Kalles `print(5)?` kompilator oppretter en funksjon `print`, som tar `int` som arg.
- * Kalles `print('c')?` Kompilator oppretter en funksjon som tar `char` som arg.
- * Men, innholdet er likt - ingen garanti for at det virker: `class MyClass{}; ...print(MyClass c)...?`

Templates forts.

- * Flere template-argumenter gir flere kombinasjoner av typer

```
template<class type1, class type2>
void print2(type1 t1, type2 t2){
    cout << t1 << " and " << t2 << endl;
}
```

- * Skopet gjelder kun for klassen eller funksjonen direkte etter
- * Samme navn, som feks. "T" kan gjenbrukes i flere funksjoner i samme namespace, men må da oppgis på nytt hvert sted.

Demo:

`templates1.cpp`
`template_mini.cpp`

Template klasser

- *

```
template<class key, class value>
class myHash{
    key k;
    value v;
public:
    myHash(key nKey, value nValue);
    printValue(); printKey();
}
```
- * Kompilator vil nå opprette klassene ettersom de kalles - men type må spesifiseres:

myHash <string, int> h2("age", 12);

myHash <int, char> h2(5, '5');

Template parametre

- * Når vi skriver:
`template<class key, class value>`
`class myHash{`
Har vi introdusert to generisk typer som "template parametre" til klassen
- * Vi bruker ordet "class" for å si at "key" og "value" er "typer", (vi kan også bruke ordet "type")
- * Men, vi kan også bruke tall som template parametre:
`template<class T, int SIZE>`
`struct myArr{ T elements[SIZE]; }`
- * Nå har vi både en type, og en "int" som template-parametre
- * For hver "myArr" vi lager, med ulik "int", oppretter komplataoren en egen struct.
- * Det funker også med andre "ting" enn typer og int'er, men det er mindre vanlig og dekkes ikke i kurset.

Demo:

key_value.cpp
int_template_params.cpp

Lykke til med
fyrverkeri!
