TDP005 - Projekt: Objektorienterat system

Kursupplägg, kravspecifikation och utvecklingsmetoder

Pontus Haglund & Eric Ekström

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1 Make och CMake
2 Versionshantering
3 Objektorienterad analys
4 UML
5 OOA/UML - Exempel

LINKÖPING UNIVERSITY

Make

- Problem: kompilera många filer i ett stort projekt tar tid
- Bättre om vi kompilerar om så få filer som möjligt!
- Make kan hjälpa oss!

Make

foo.h

```
#pragma once

int foo(int val);
```

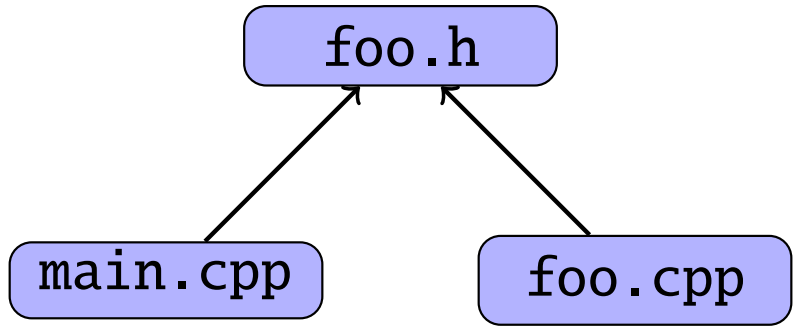
main.cpp

```
#include "foo.h"
#include <iostream>
//...
int main() {
  cout << foo(10) << endl;
  return 0;
}
```

foo.cpp

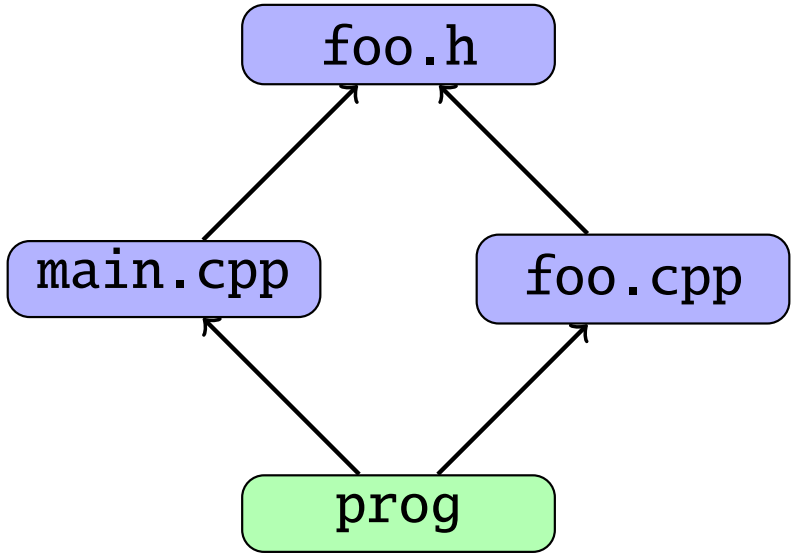
```
#include "foo.h"

int foo(int val) {
  return val + 1;
}
```

Make – Beroenden



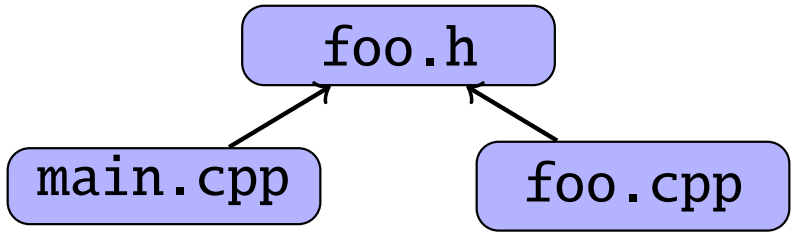

```
g++ -std=c++17 -Wall *.cpp -o prog
```

Make – Beroenden



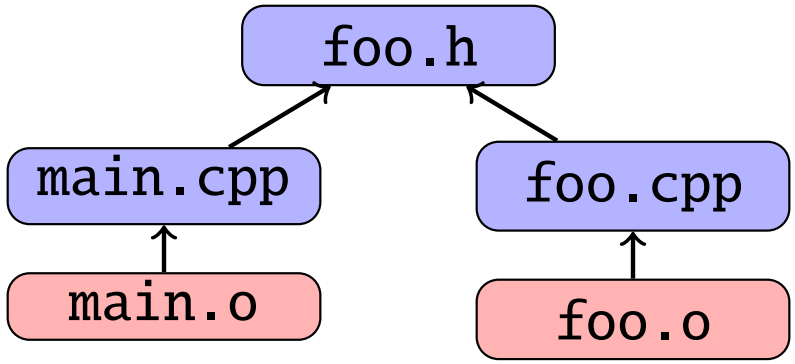

```
g++ -std=c++17 -Wall *.cpp -o prog
```

Make – Beroenden



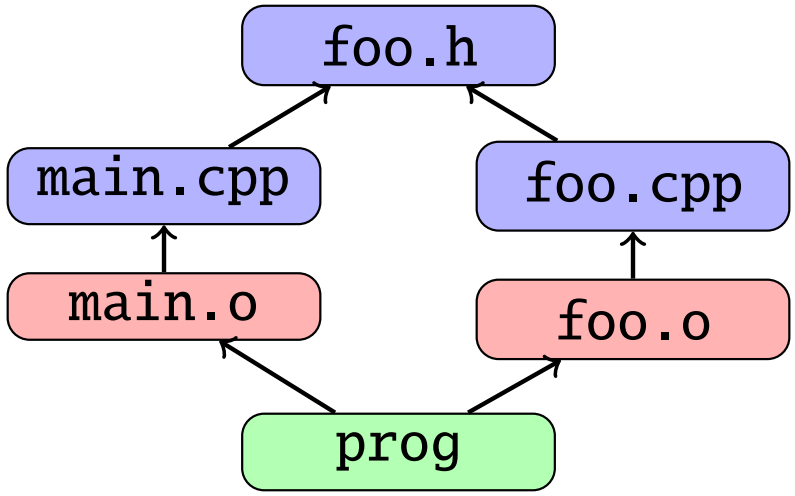

```
g++ -c -std=c++17 -Wall main.cpp -o main.o
g++ -c -std=c++17 -Wall foo.cpp -o foo.o
g++ -std=c++17 main.o foo.o -o prog
```

Make – Beroenden




```
g++ -c -std=c++17 -Wall main.cpp -o main.o
g++ -c -std=c++17 -Wall foo.cpp -o foo.o
g++ -std=c++17 main.o foo.o -o prog
```

Make – Beroenden




```
g++ -c -std=c++17 -Wall main.cpp -o main.o
g++ -c -std=c++17 -Wall foo.cpp -o foo.o
g++ -std=c++17 main.o foo.o -o prog
```

Make – makefil

- Första försök:

```
prog: foo.o main.o
  g++ -std=c++17 -g -Wall foo.o main.o -o prog

foo.o: foo.cpp foo.h
  g++ -c -std=c++17 -g -Wall foo.cpp -o foo.o

main.o: main.cpp foo.h
  g++ -c -std=c++17 -g -Wall main.cpp -o main.o
```

Make – makefil

- Förbättring:

```
CXXFLAGS = -std=c++17 -g -Wall

prog: foo.o main.o
  g++ $(CXXFLAGS) foo.o main.o -o prog

foo.o: foo.cpp foo.h
  g++ -c $(CXXFLAGS) foo.cpp -o foo.o

main.o: main.cpp foo.h
  g++ -c $(CXXFLAGS) main.cpp -o main.o
```

Make – makefil med `.PHONY` och `clean`

```
CXXFLAGS = -std=c++17 -g -Wall

prog: foo.o main.o
  g++ $(CXXFLAGS) foo.o main.o -o prog

foo.o: foo.cpp foo.h
  g++ -c $(CXXFLAGS) foo.cpp -o foo.o

main.o: main.cpp foo.h
  g++ -c $(CXXFLAGS) main.cpp -o main.o

.PHONY: clean
clean:
  rm *.o prog
```

Make

- Finns mycket mer: makron, variabler, etc.
- Se make-laborationen för mer info.

CMake

CMakeLists.txt:

```
project(prog)

set(CMAKE_CXX_STANDARD 17)
#set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++17")

set(SOURCE_FILES main.cpp foo.cpp foo.h)

add_executable(prog ${SOURCE_FILES})
```

Kompilera:

```
    $ cmake .
    $ make
```

1 Make och CMake
2 Versionshantering
3 Objektorienterad analys
4 UML
5 OOA/UML - Exempel

Git

- Git ska användas i kursen
- Interaktiv genomgång:
  http://learngitbranching.js.org/
- Hur används Git i större projekt?

GitLab – Issues

GitLab har inbyggt stöd för issues. Kan användas för att hålla koll på vad som ska göras härnäst, buggar, etc.

I ett commit-meddelande kan man skriva:

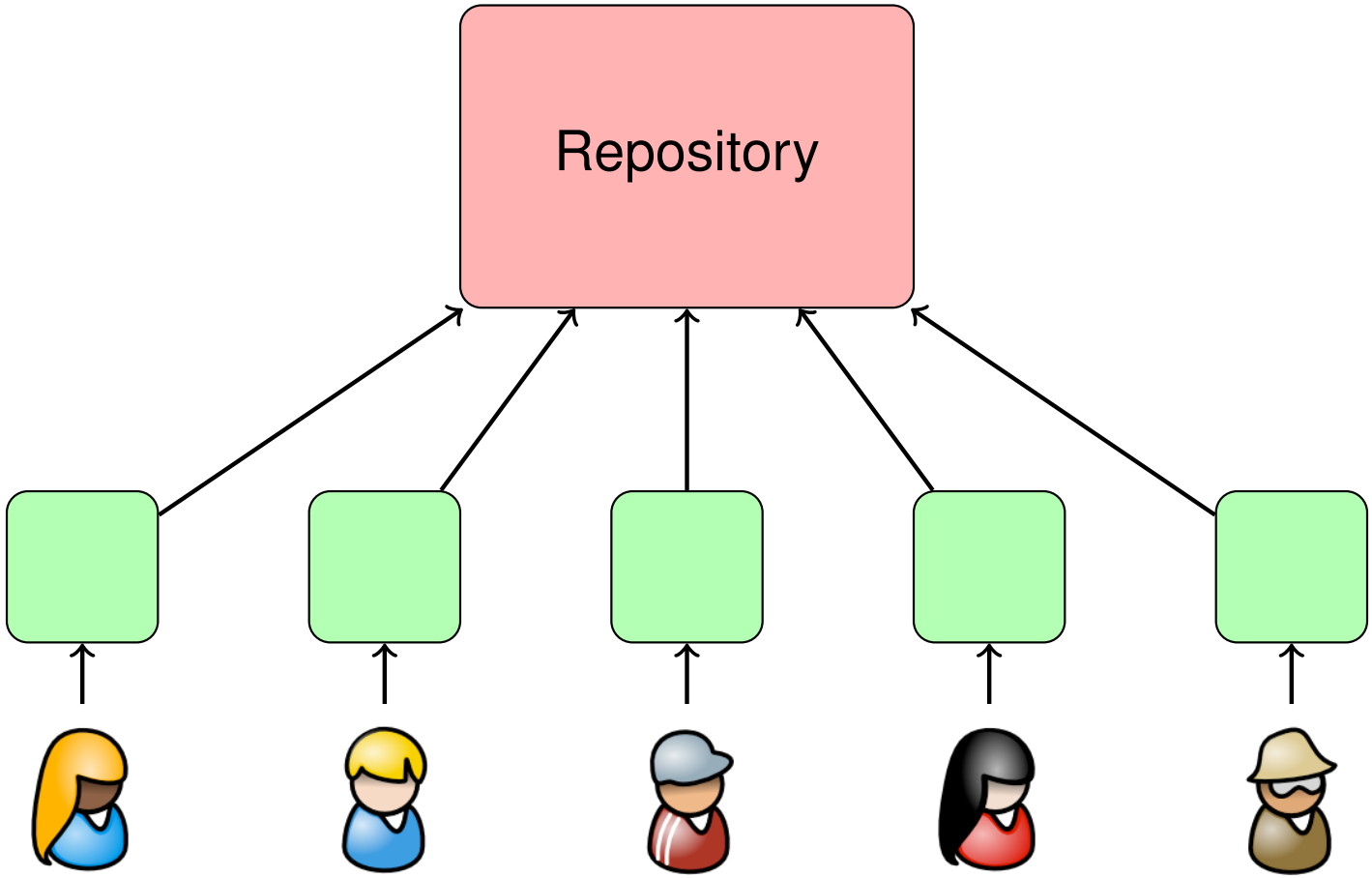
fixes #13

Då stängs automatiskt issue #13, och en referens till den commit som stängde den läggs till.

Git – Hur organiseras koden i större projekt?

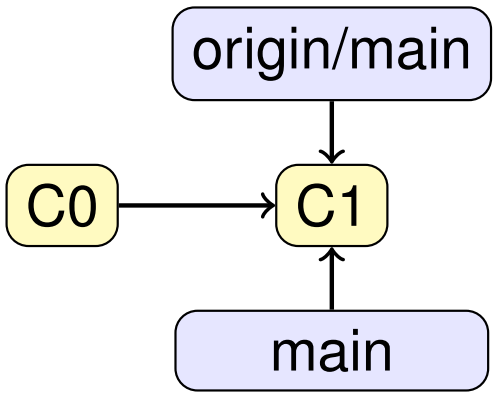
- Merge
- Rebase
- Feature Branch
- Gitflow

Git – Centralt repository



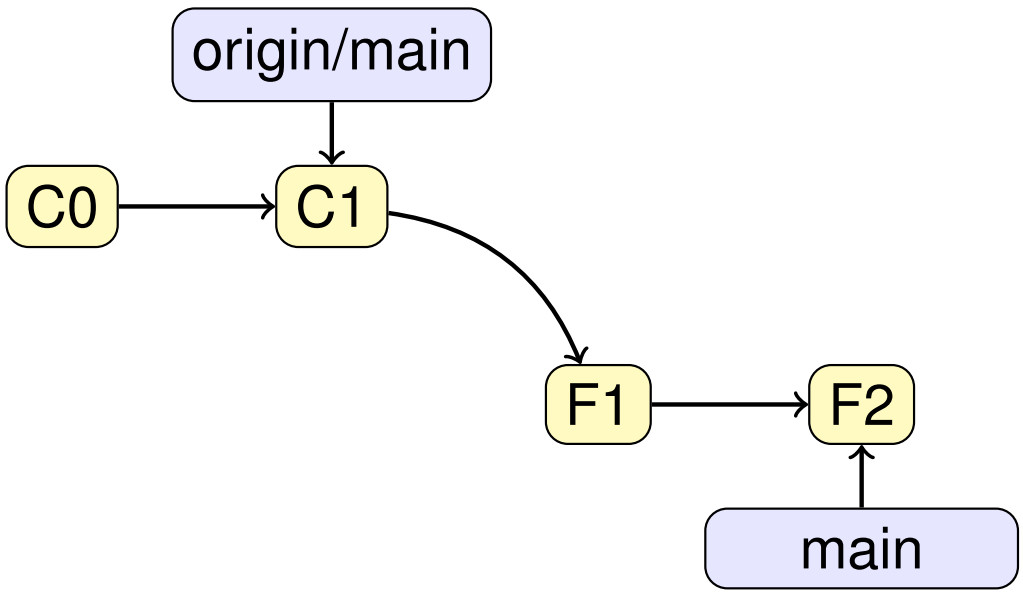
Git – Merge

```
git commit
git push
```



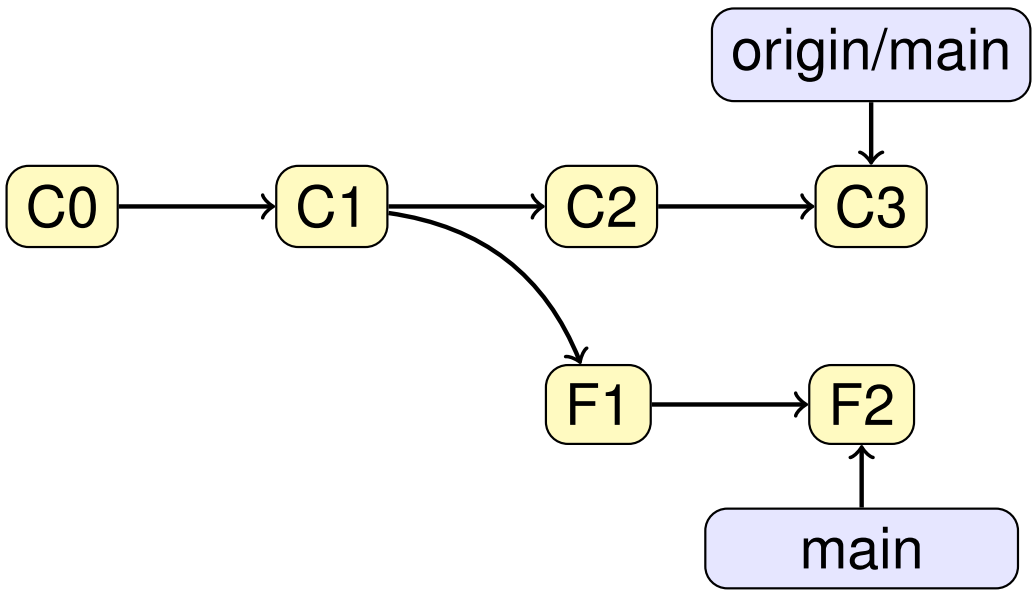
Git – Merge

```
    git commit
    git commit
```



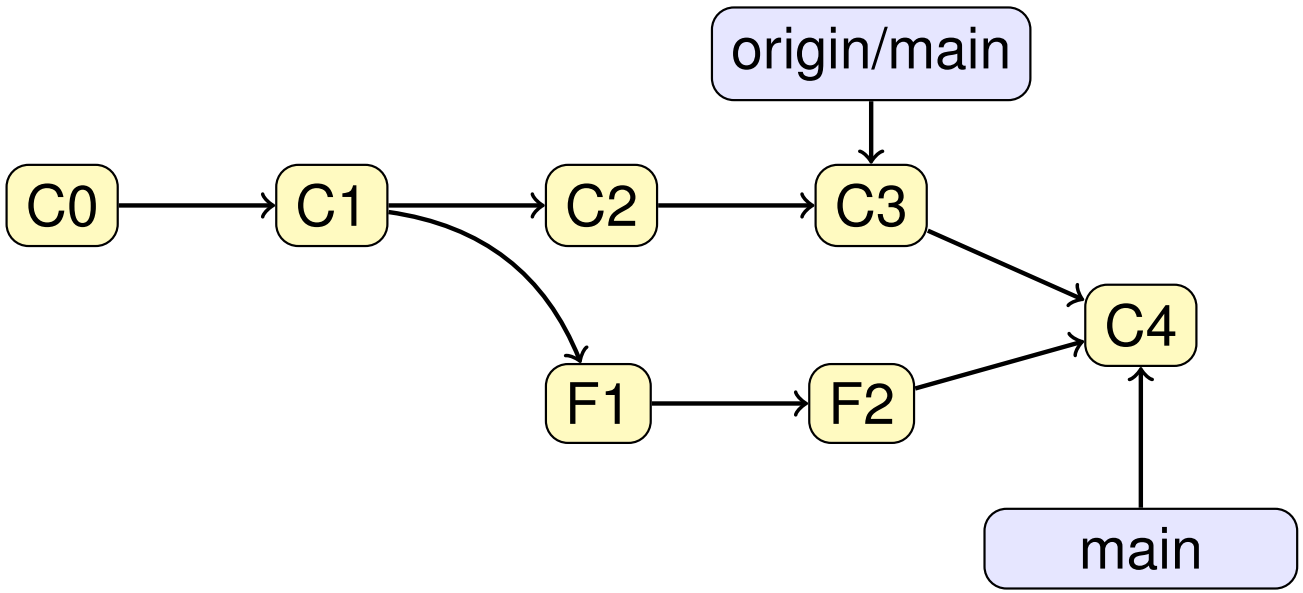
Git – Merge

```
<remote work>
git fetch
```



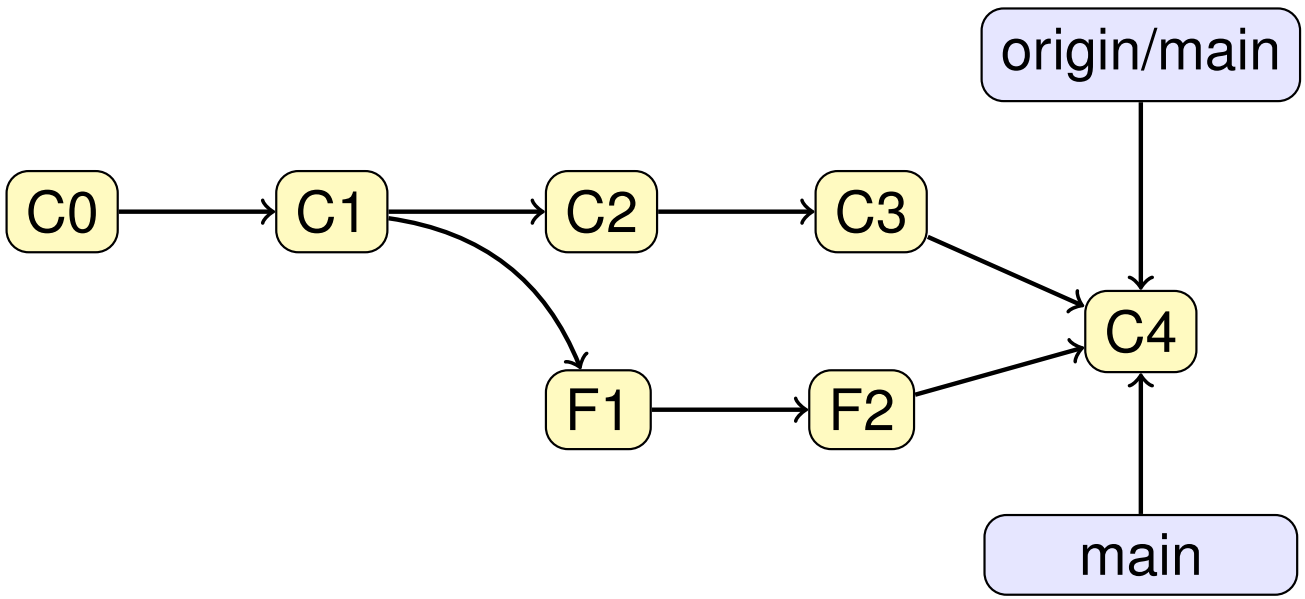
Git – Merge

```
git merge origin/main
```



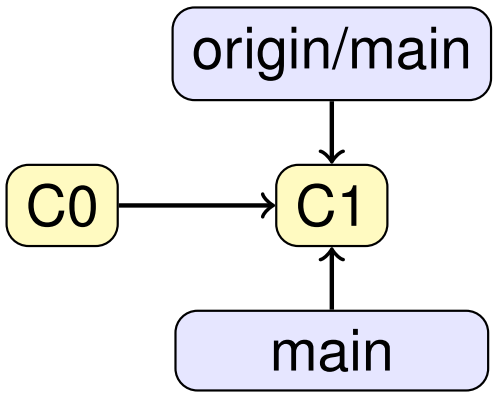
Git – Merge

```
git push
```



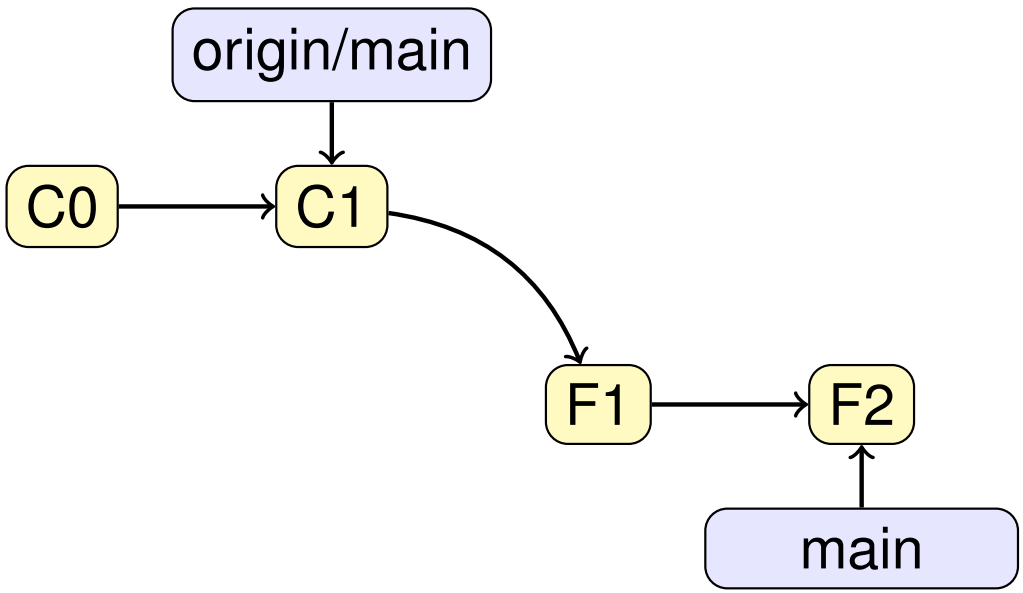
Git – Rebase

```
    git commit
    git push
```



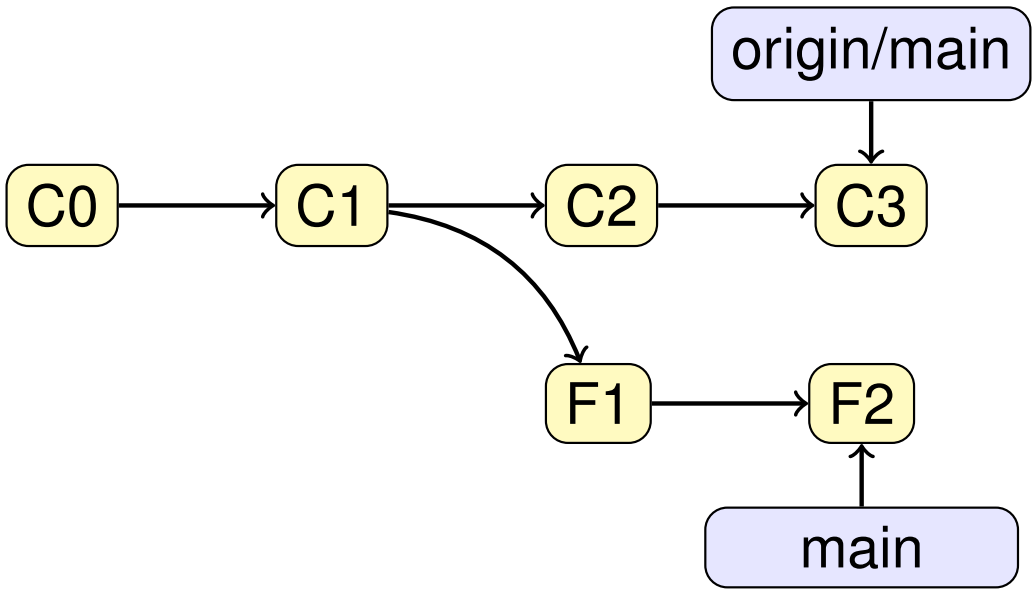
Git – Rebase

```
    git commit
    git commit
```



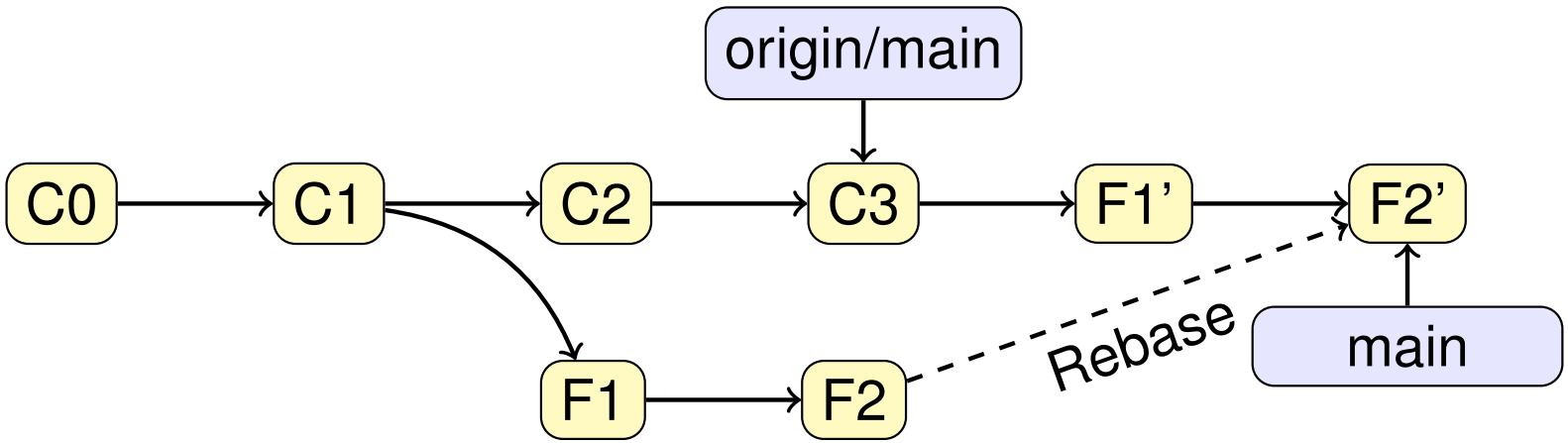
Git – Rebase

```
<remote work>
git fetch
```



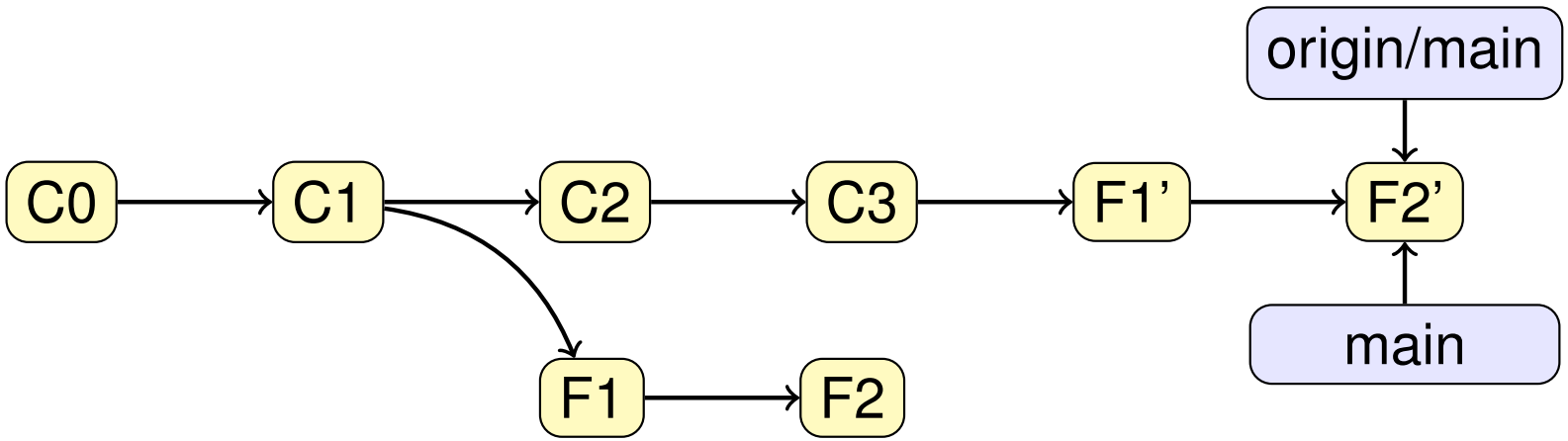
Git – Rebase

```
git rebase origin/main
```

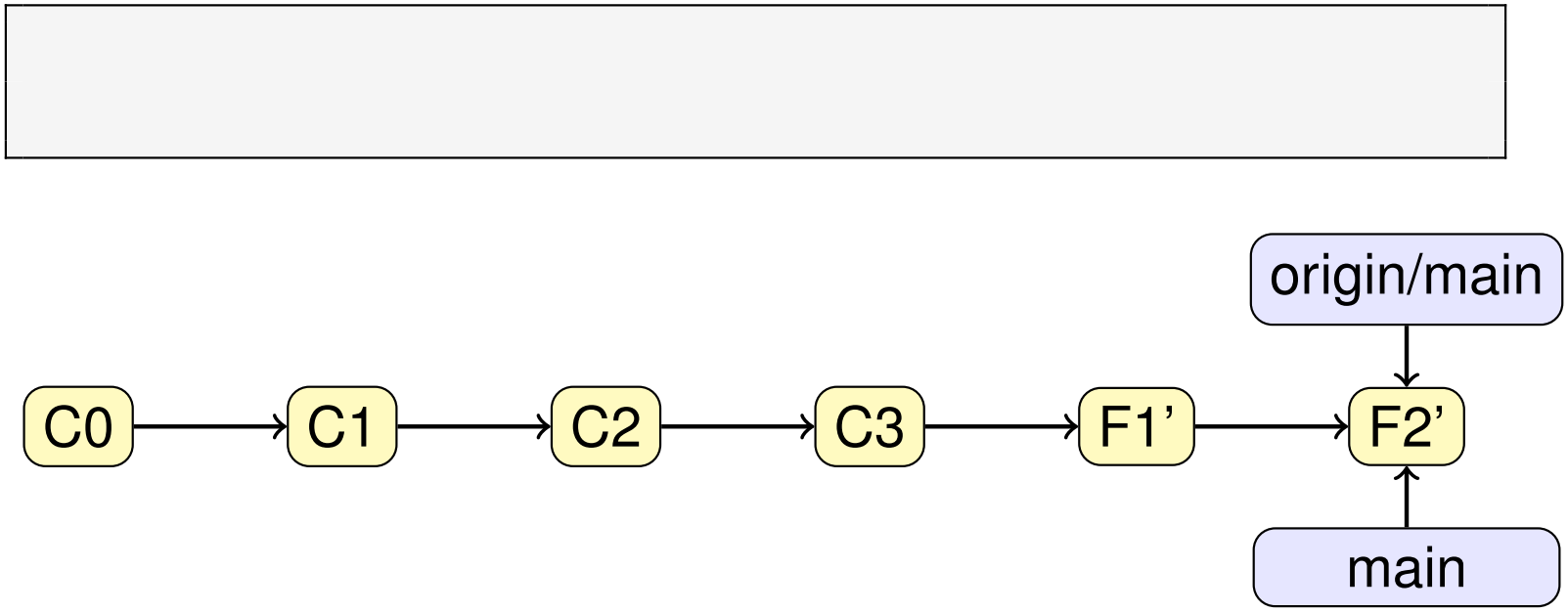


Git – Rebase

```
git push
```



Git – Rebase



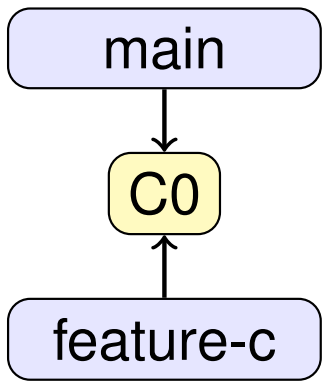
Git – Merge och Rebase

- Enklaste sättet att samarbeta
- Fungerar bra i mindre projekt

- Kan inte samarbeta på funktionalitet som ej är klar
- Risk att `main` inte fungerar, problem för andra
- Stor feature, stora merge-konflikter

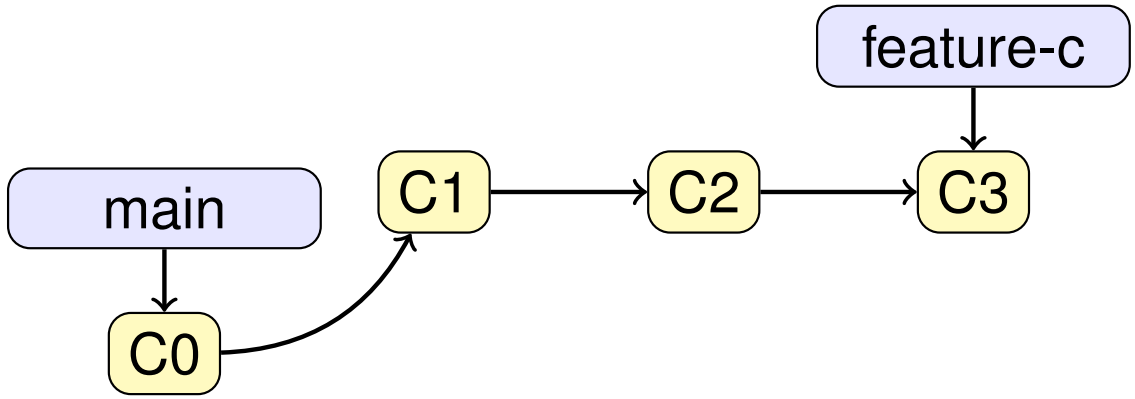
Git – Feature Branch

```
git branch feature-c
git checkout feature-c
```



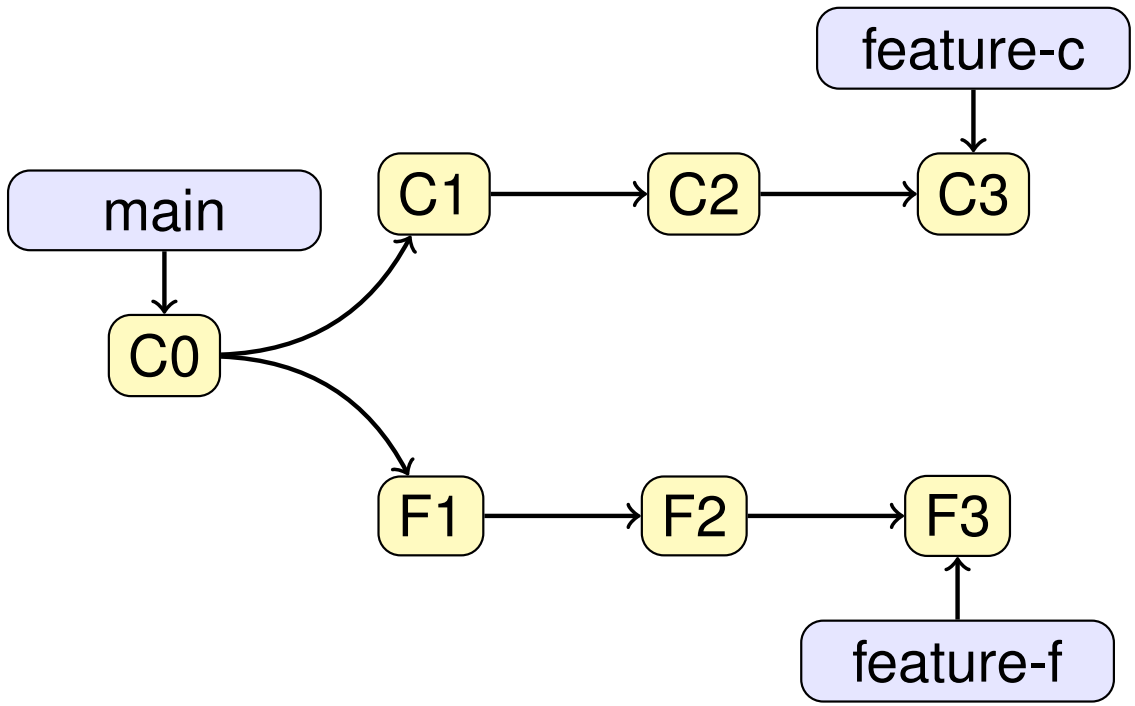
Git – Feature Branch

```
    git commit x3
```



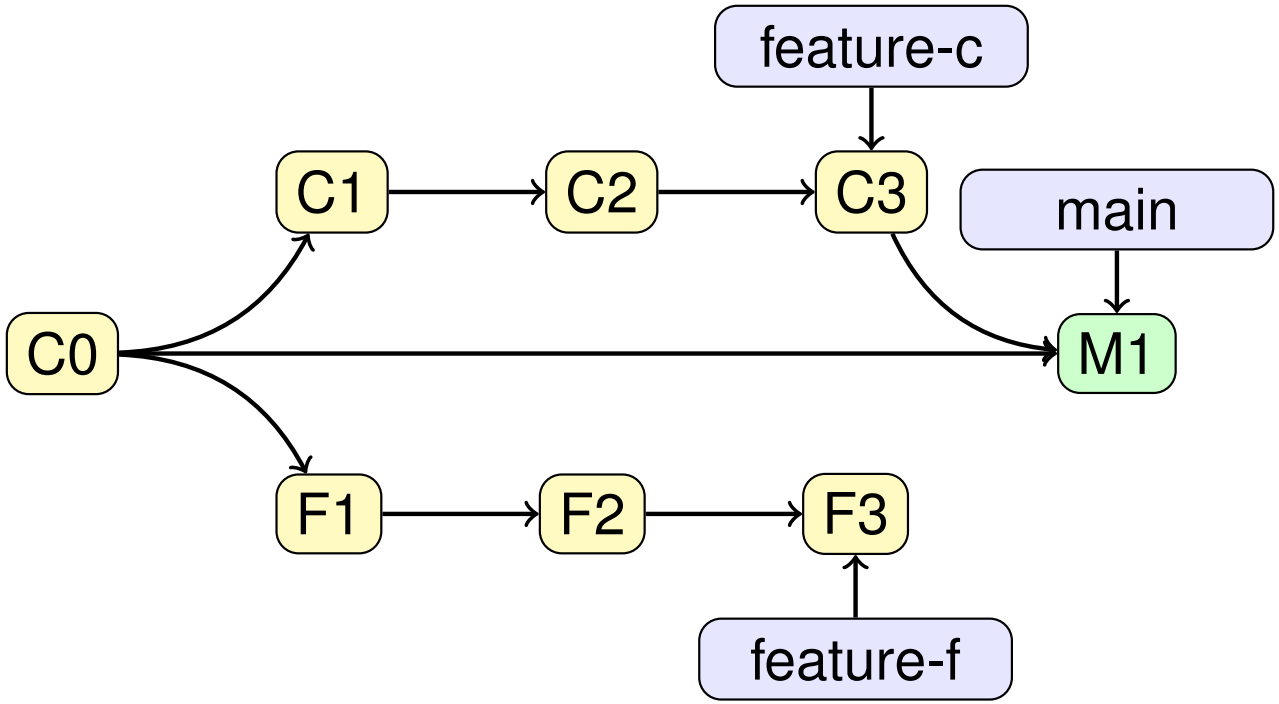
Git – Feature Branch

```
git checkout -b feature-f main
git commit x3
```



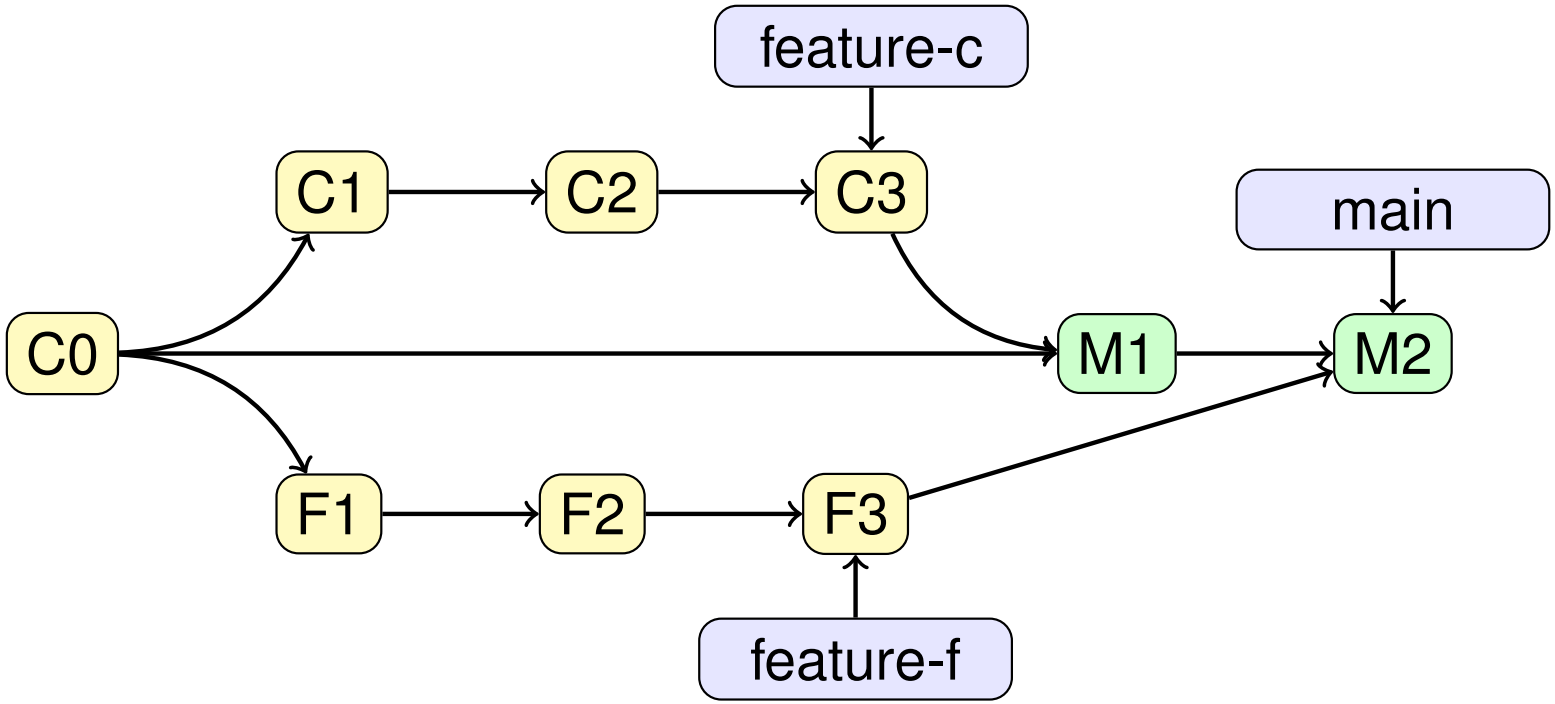
Git – Feature Branch

```
git checkout main
git merge feature-c
```



Git – Feature Branch

```
git merge feature-f
```

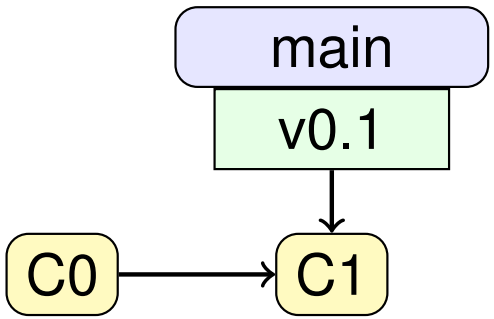


Git – Feature Branch

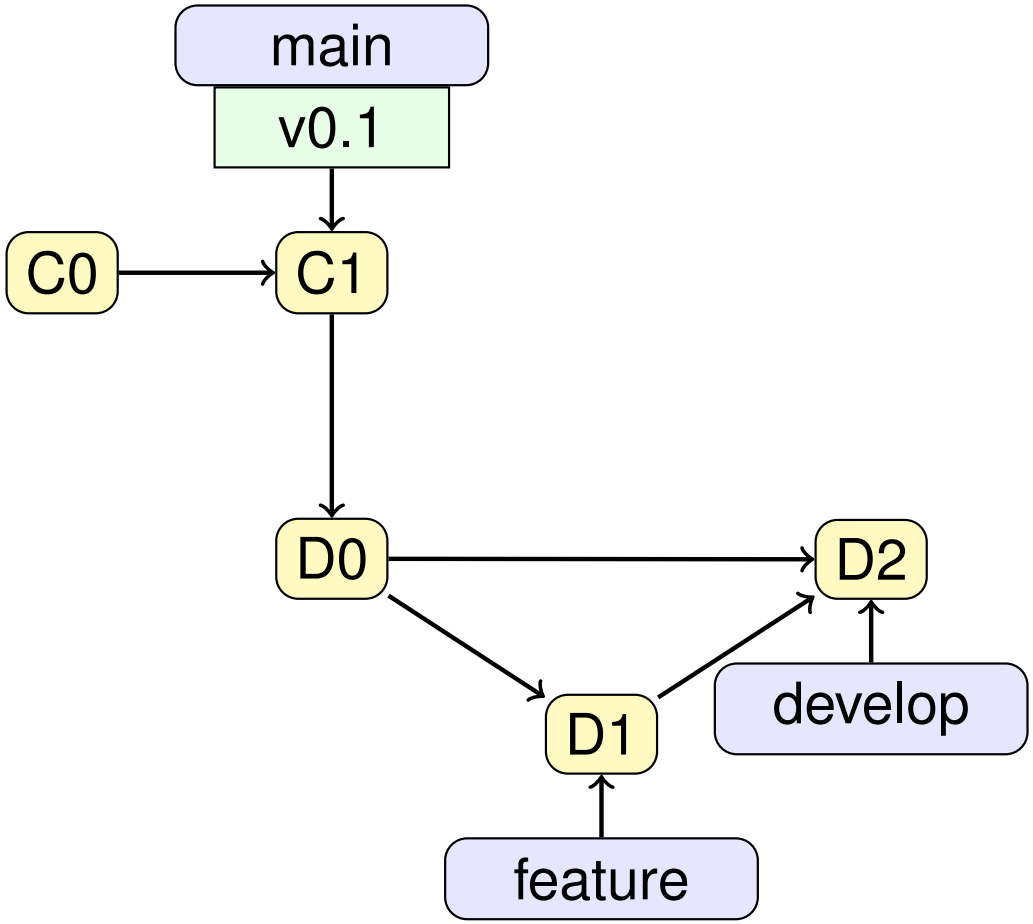
- Samarbete/backup på ej klara funktionalitet
- Tydligt var kodgranskning görs
- `main` fungerar alltid $\Rightarrow$ Continuous Integration

- Lite mer arbete än tidigare
- Risk för att funktionalitet ”tappas bort”
- Risk för stora merges $\Rightarrow$ lite funktionalitet i taget

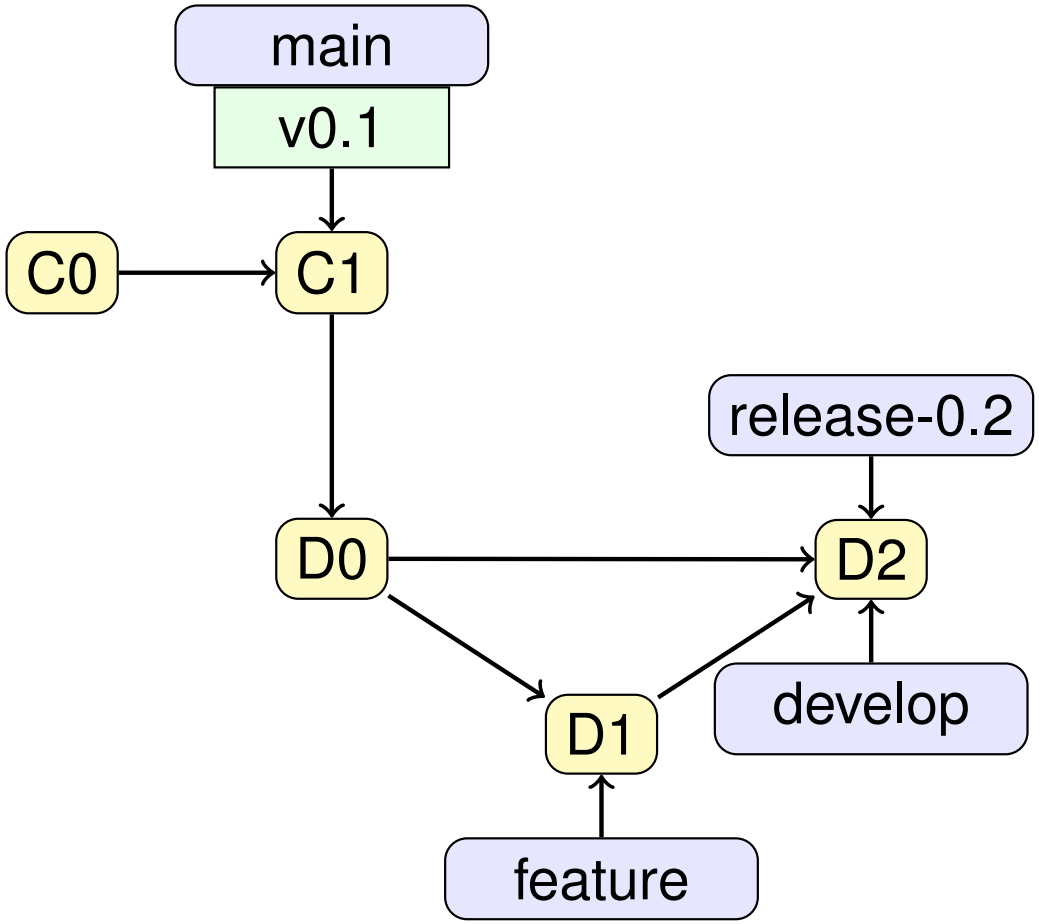
Git – Gitflow



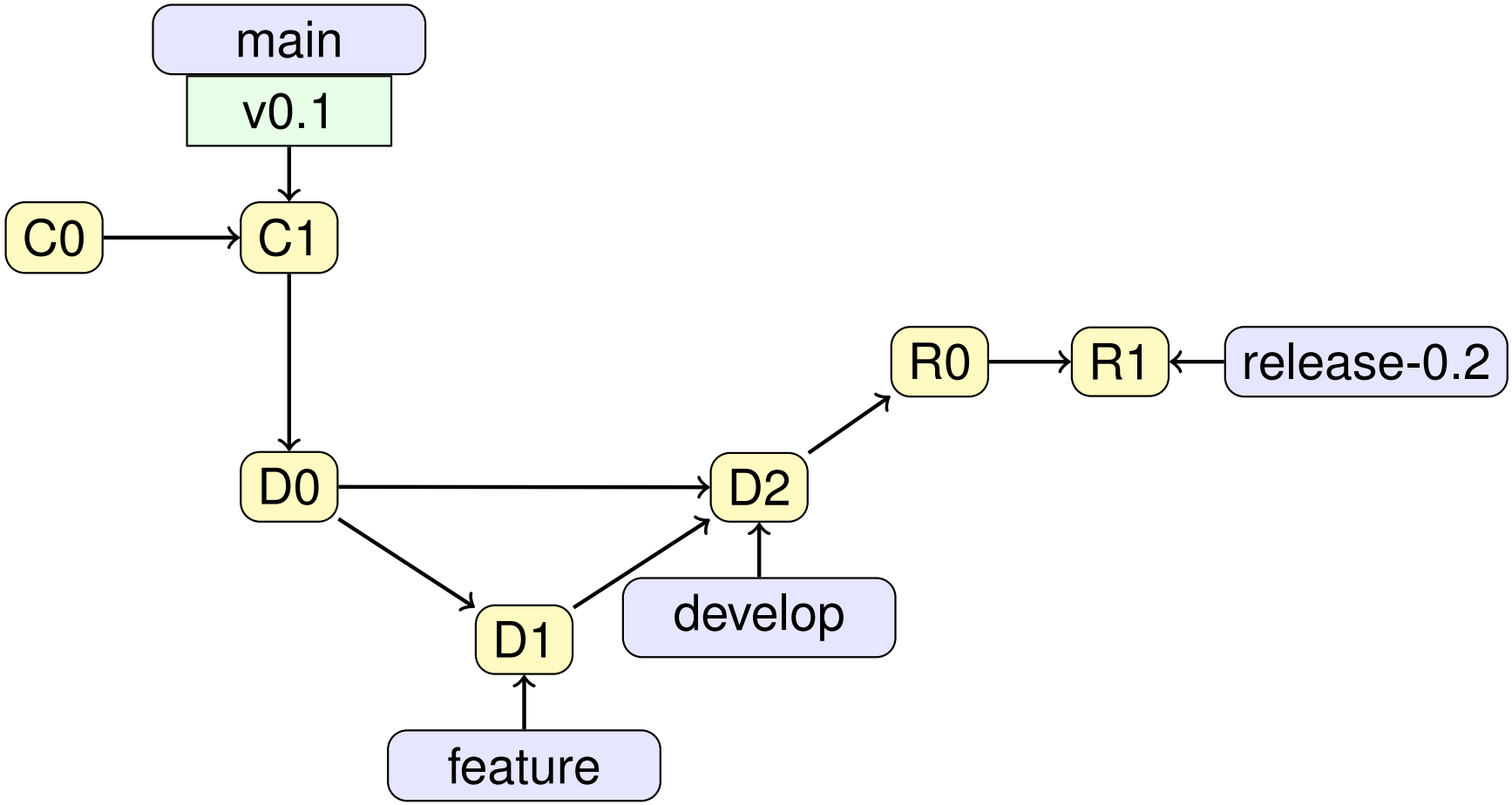
Git – Gitflow



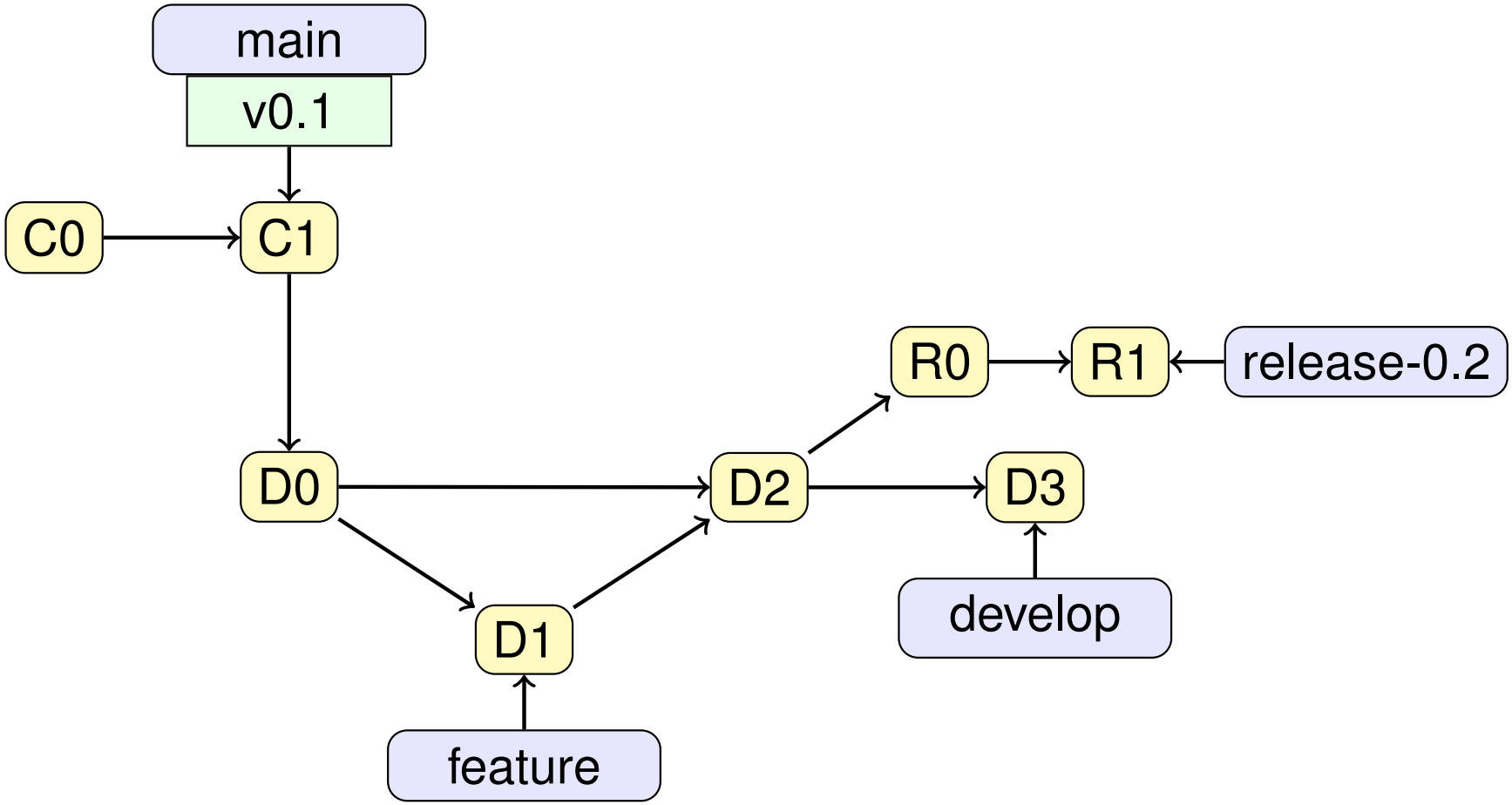
Git – Gitflow



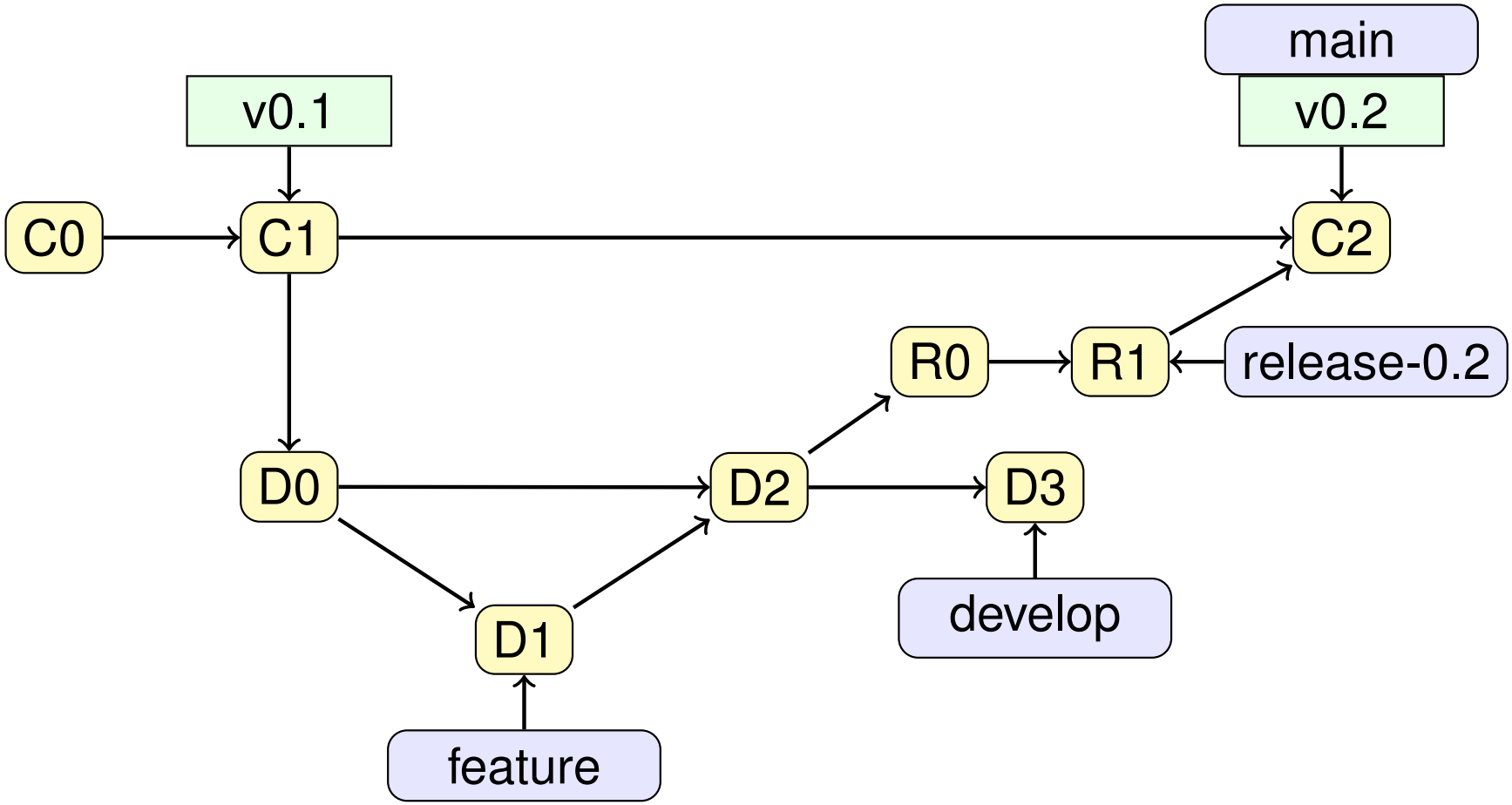
Git – Gitflow



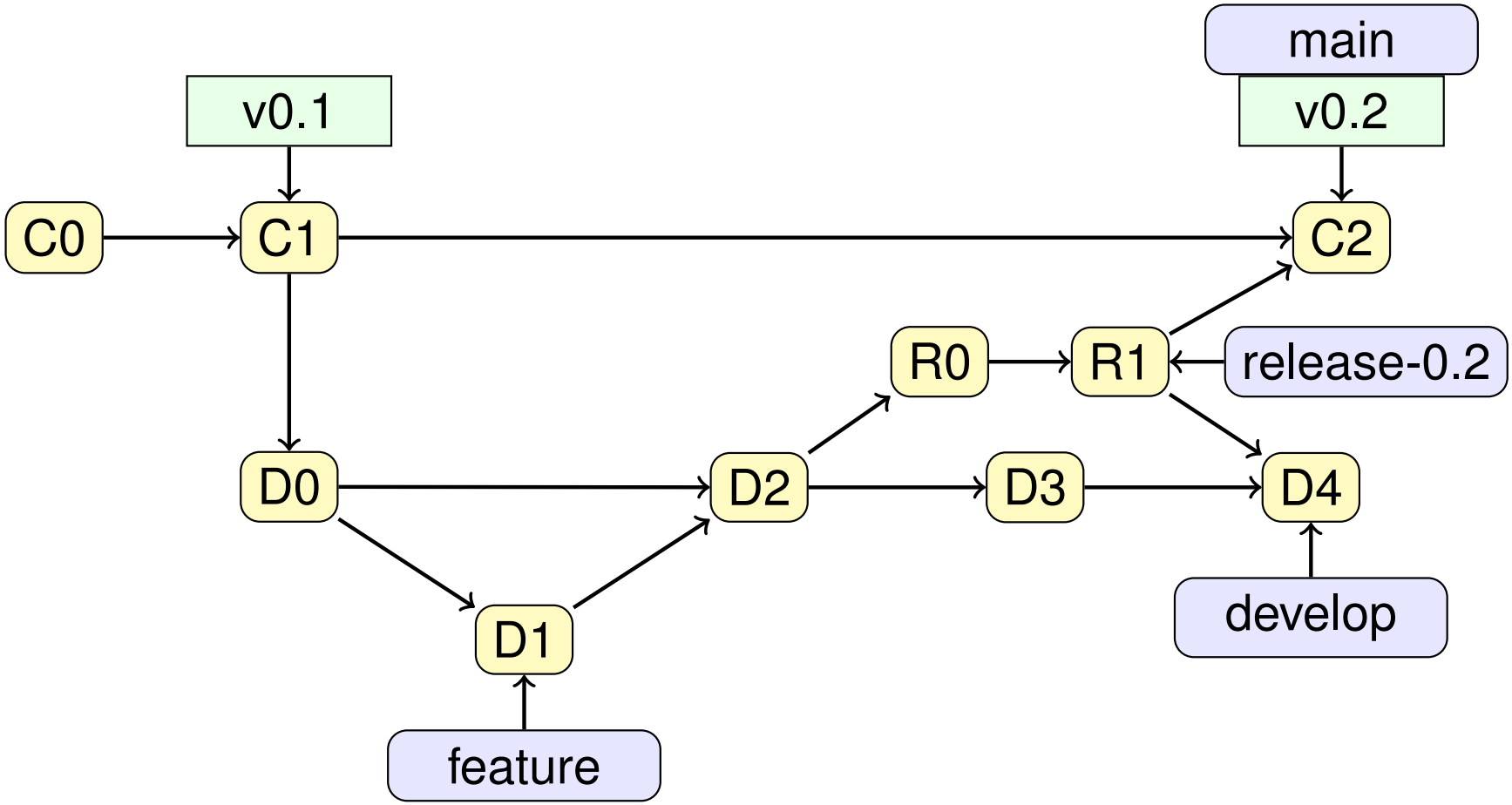
Git – Gitflow



Git – Gitflow


main
v0.1
v0.2
C0
C1
C2
R0
R1
release-0.2
D0
D2
D3
develop
D1
feature

Git – Gitflow


main
v0.1
v0.2
C0
C1
C2
R0
R1
release-0.2
D0
D2
D3
D4
D1
develop
feature

Git – Gitflow

- Release kan testas och färdigställas parallellt med utveckling av ny funktionalitet
- `main` visar tydligt alla versioner
- Enkelt och tydligt att göra hotfix

- Ännu mer att hålla reda på, namngivning är viktig!

1 Make och CMake
2 Versionshantering
3 Objektorienterad analys
4 UML
5 OOA/UML - Exempel

LINKÖPING UNIVERSITY

Objektorienterad analys

- Finn objekt
- Klassificera objekten
- Beskriv relationer
- Identifiera användningfall

Objektorienterad analys

Steg 1: Finn objekt
Ett förslag till hur man ska finna objekt är att läsa kravspecifikationen och notera förekommande substantiv.

Objektorienterad analys

Steg 2: Klass, ansvar, samarbete
För varje objekt från steg 1, skriv ned klassens:

- namn - välj namn med omsorg! Välj engelska namn som substantiv i singularis.
- ansvar - operationer(verb) som kan utföras på eller av objekt av klassen ifråga.
- samarbetspartners - vilka andra klasser som klassen samarbetar med för att utföra sina åtaganden. Fråga dig: “varifrån får objektet denna information?”, “till vem ska objektet leverera denna information?”

Objektorienterad analys

Steg 3: Relationer
Bestäm de relationer som finns mellan klasserna:

- arv (x är en specialisering av y)
- komposition (x består av komponenten y, x finns inte utan y)
- aggregation (x består av komponenten y)
- association (x kan använda en y)

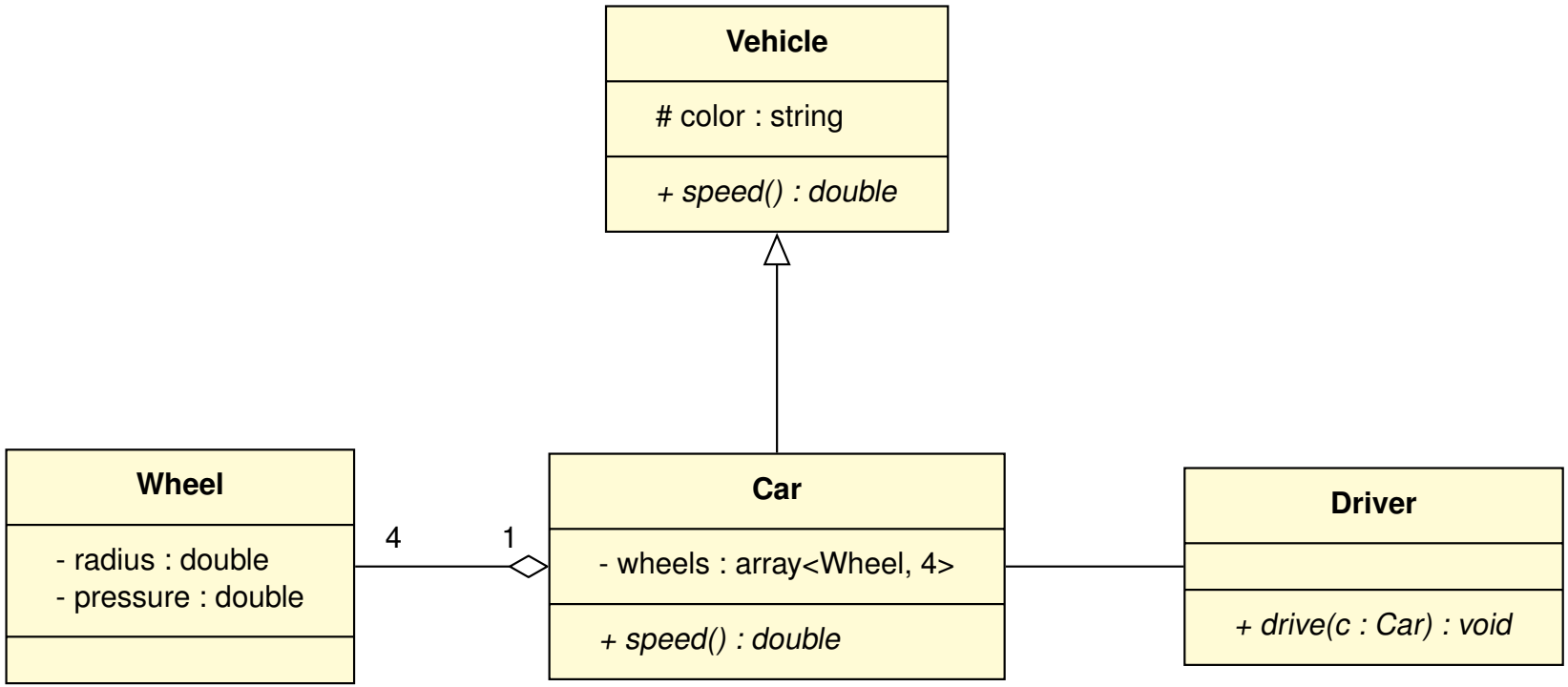
Objektorienterad analys

Steg 4: Användningfall
Ett användningsfall är en interaktion som kan inträffa under systemets exekvering. Syftet med att identifiera aktörer och användningsfall och utföra olika scenarier för användningsfall är att få en djupare insikt om samarbetet mellan objekt. Under denna aktivitet kan man till exempel upptäcka nya aktörer och användningsfall, nya objekt/klasser, nya samarbetspartners och nya relationer mellan klasser och objekt. Omvänt gäller att varje objekt/klass/ansvar/samarbete ska ingå i något användningsfall för att dess existens ska vara motiverad.

1 Make och CMake
2 Versionshantering
3 Objektorienterad analys
4 UML
5 OOA/UML - Exempel

LINKÖPING UNIVERSITY

UML


Vehicle
color : string
+ speed() : double
Wheel
- radius : double
- pressure : double
4
1
Car
- wheels : array<Wheel, 4>
+ speed() : double
Driver
+ drive(c : Car) : void

1 Make och CMake
2 Versionshantering
3 Objektorienterad analys
4 UML
5 OOA/UML - Exempel

LINKÖPING UNIVERSITY

Space Invaders

Space Invaders är ett spel med olika flygande objekt. För spelaren är det viktigaste objektet spelarskeppet. Spelarskeppet kan förflytta sig och skjuta olika projektiler. I spelet finns det också flera olika typer av fiender. Det finns både vanliga fiender, som är lätta att stoppa, och bossar. Bossar har unik funktionalitet, att de kan skapa nya vanliga fiender. Vanliga fiender rör sig likadant och bossar har ett unikt rörelsemönster för att göra dem svårare att träffa. I spelet finns det dessutom asteroider som agerar hinder. Dessa står stilla mellan spelarskeppet och fienderna.

OOA: Exempel

Steg 1: Identifiera objekt

flygande objekt		
spelarskepp		
projektil		
fiende		
vanlig fiende		
boss		
asteroid		

OOA: Exempel

Steg 2: Klassificera objekten

flygande objekt	move	
spelarskepp	move, skjuta	
projektil	move, gör skada	
fiende	move, göra skada	
vanlig fiende	move, göra skada	
boss	move, skapa fiender	
asteroid	har position	

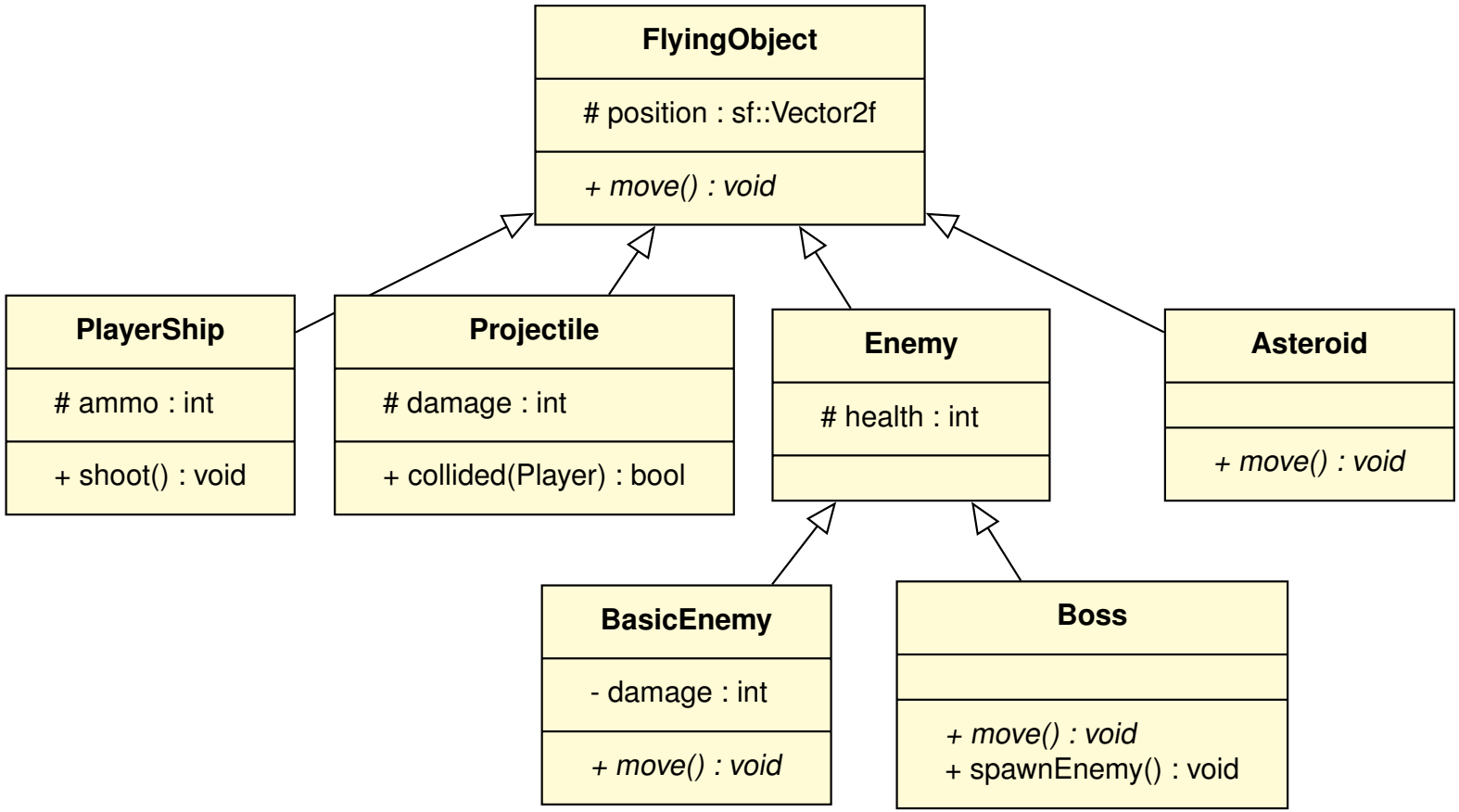
OOA: Exempel

Steg 2: Klassificera objekten

flygande objekt	move	
spelarskepp	move, skjuta	projektil
projektil	move, gör skada	spelarskepp, fiende
fiende	move, göra skada	
vanlig fiende	move, göra skada	spelarskepp
boss	move, skapa fiender	spelarskepp, fiende
asteroid	har position	spelarskepp, fiende

UML: Exempel

Steg 3: Beskriv relationer



www.liu.se

LINKÖPING UNIVERSITY