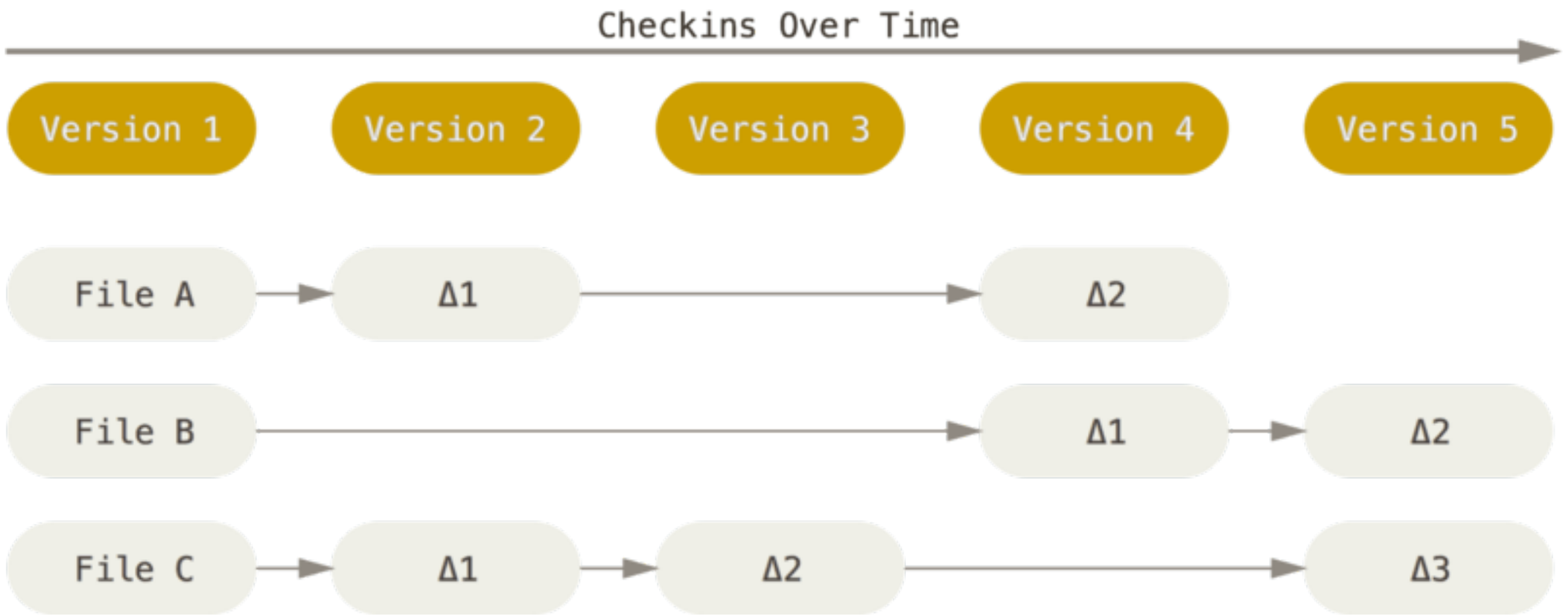
Föreläsning 7

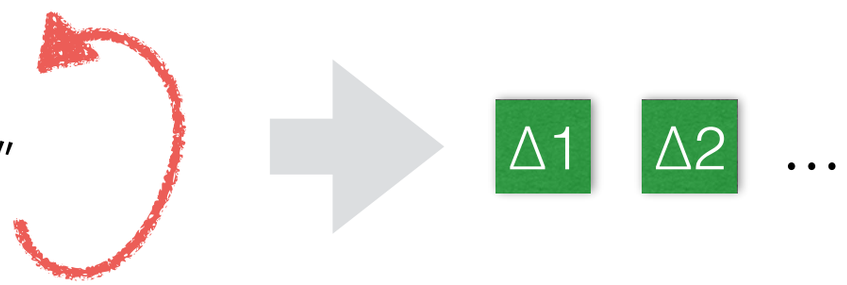
Tobias Wrigstad

Versionshantering med git

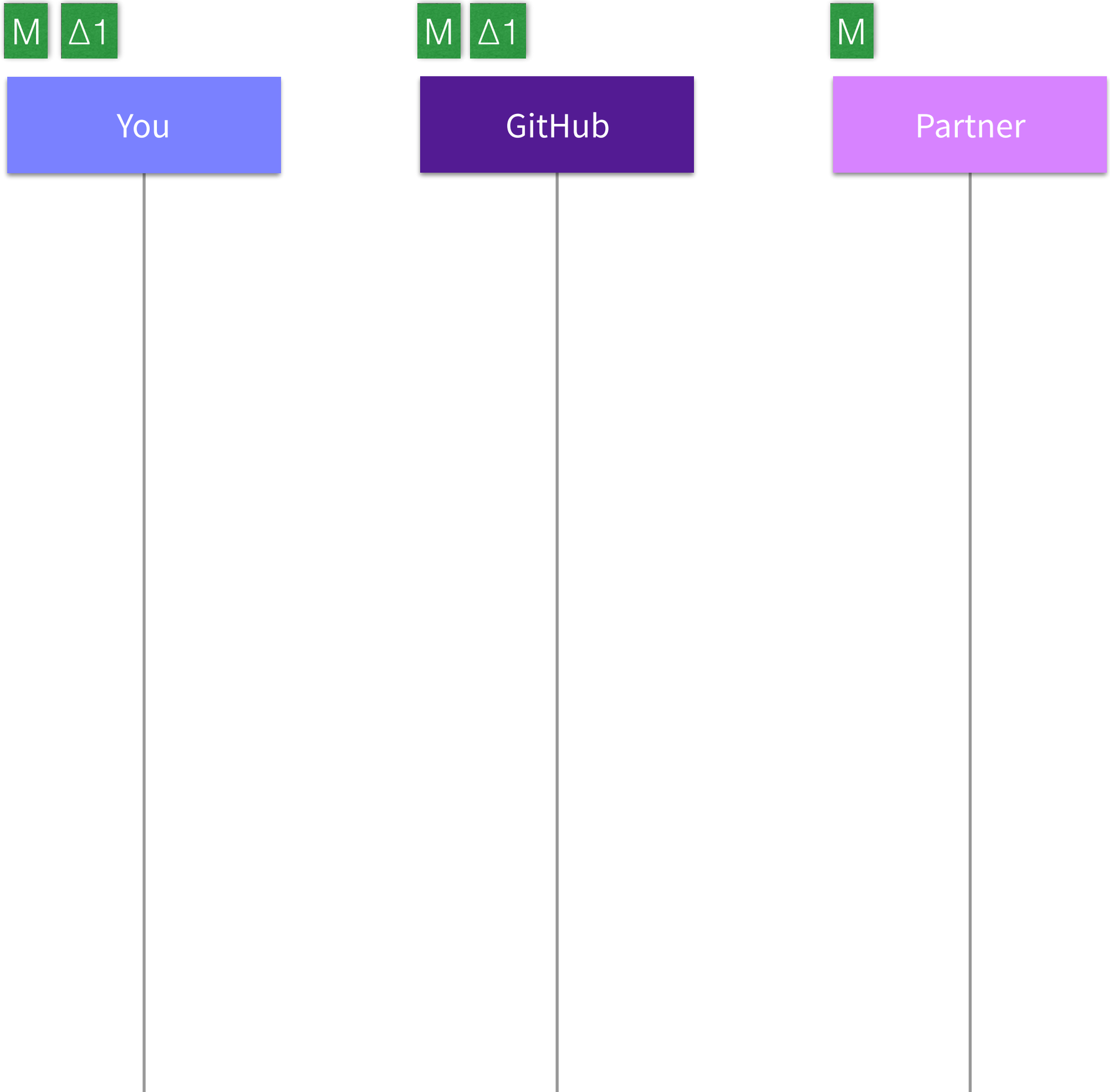


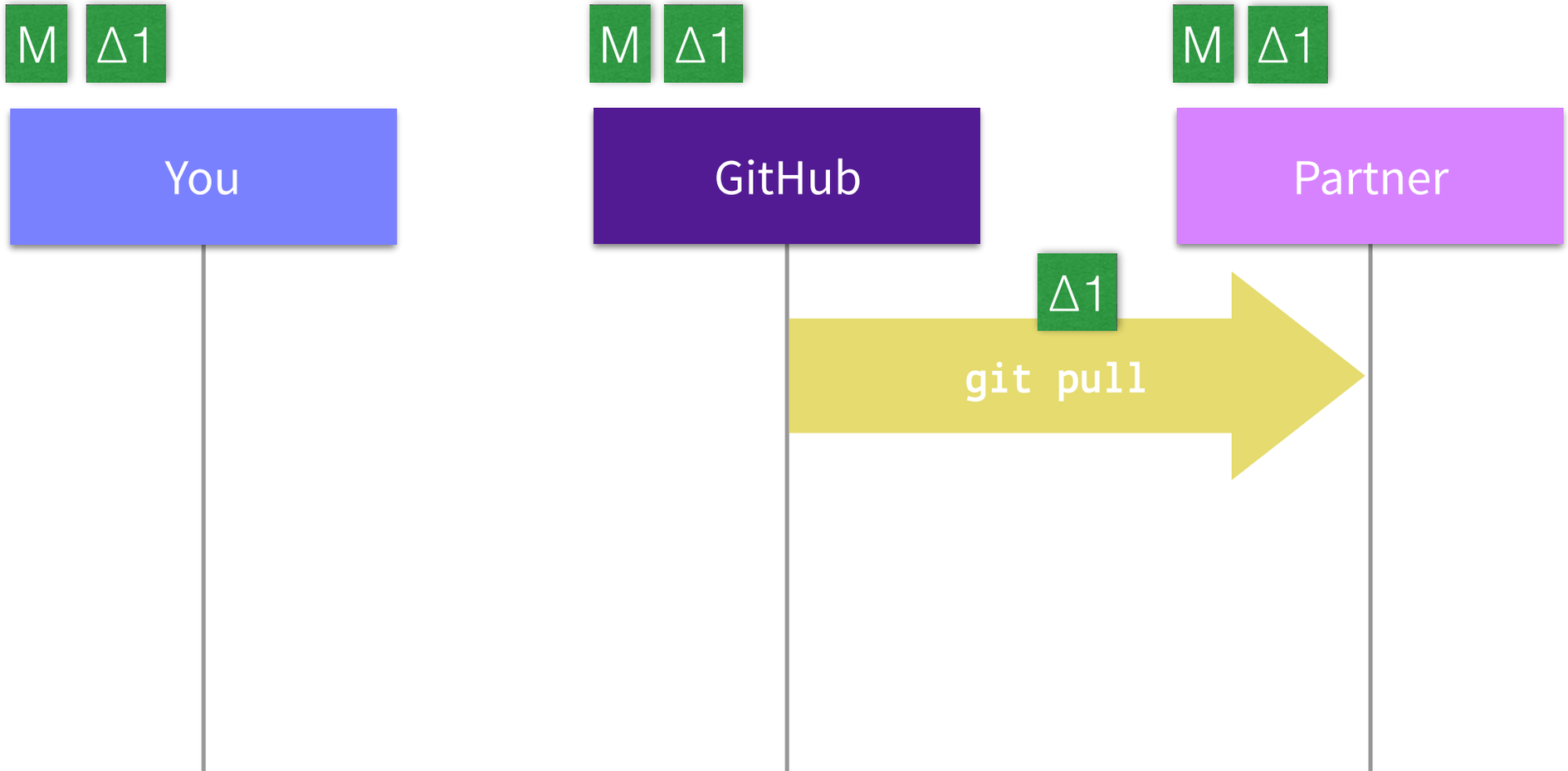
Arbetsflöde:

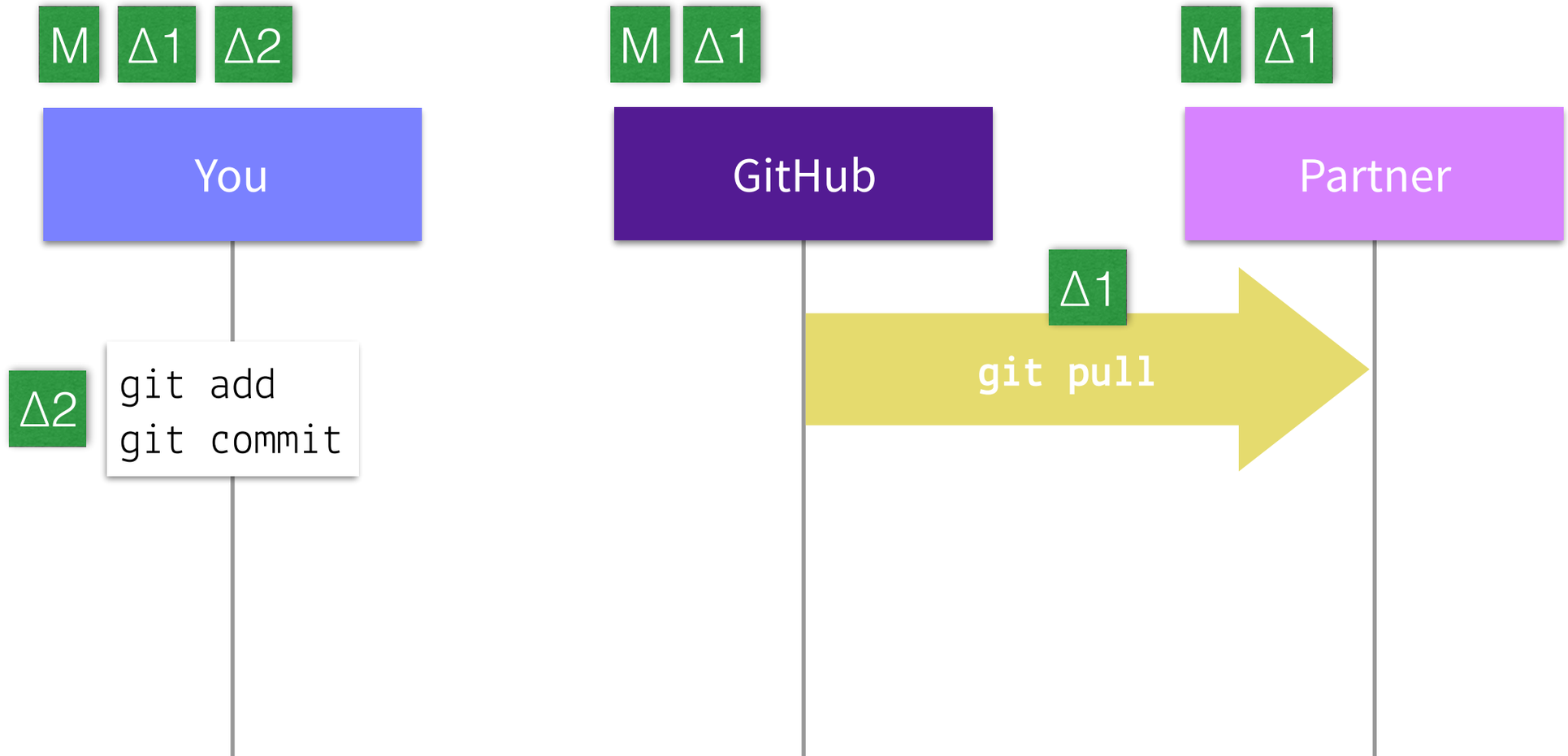
```
$ git add file.c file.h
$ git commit -m "Explanatory message"
```

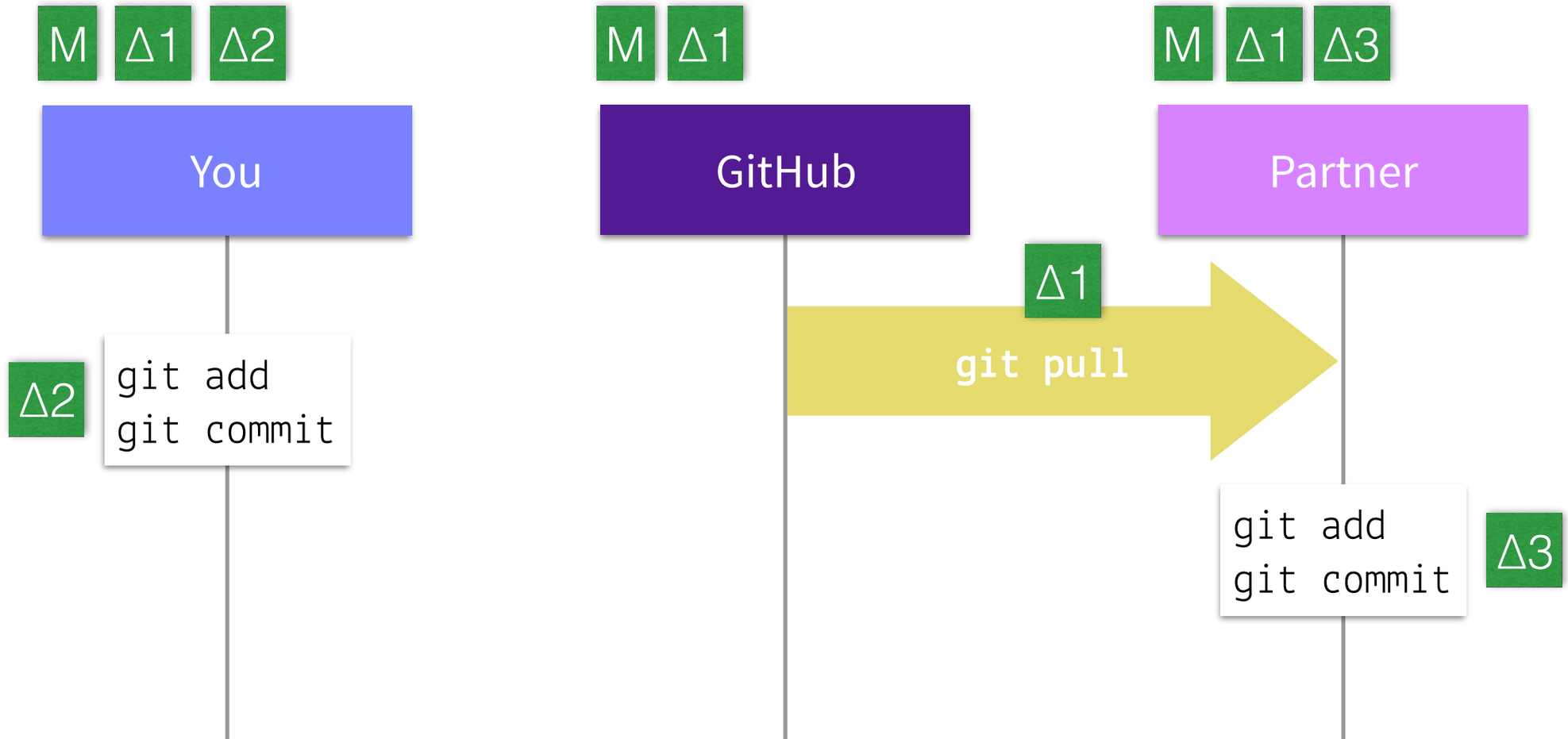


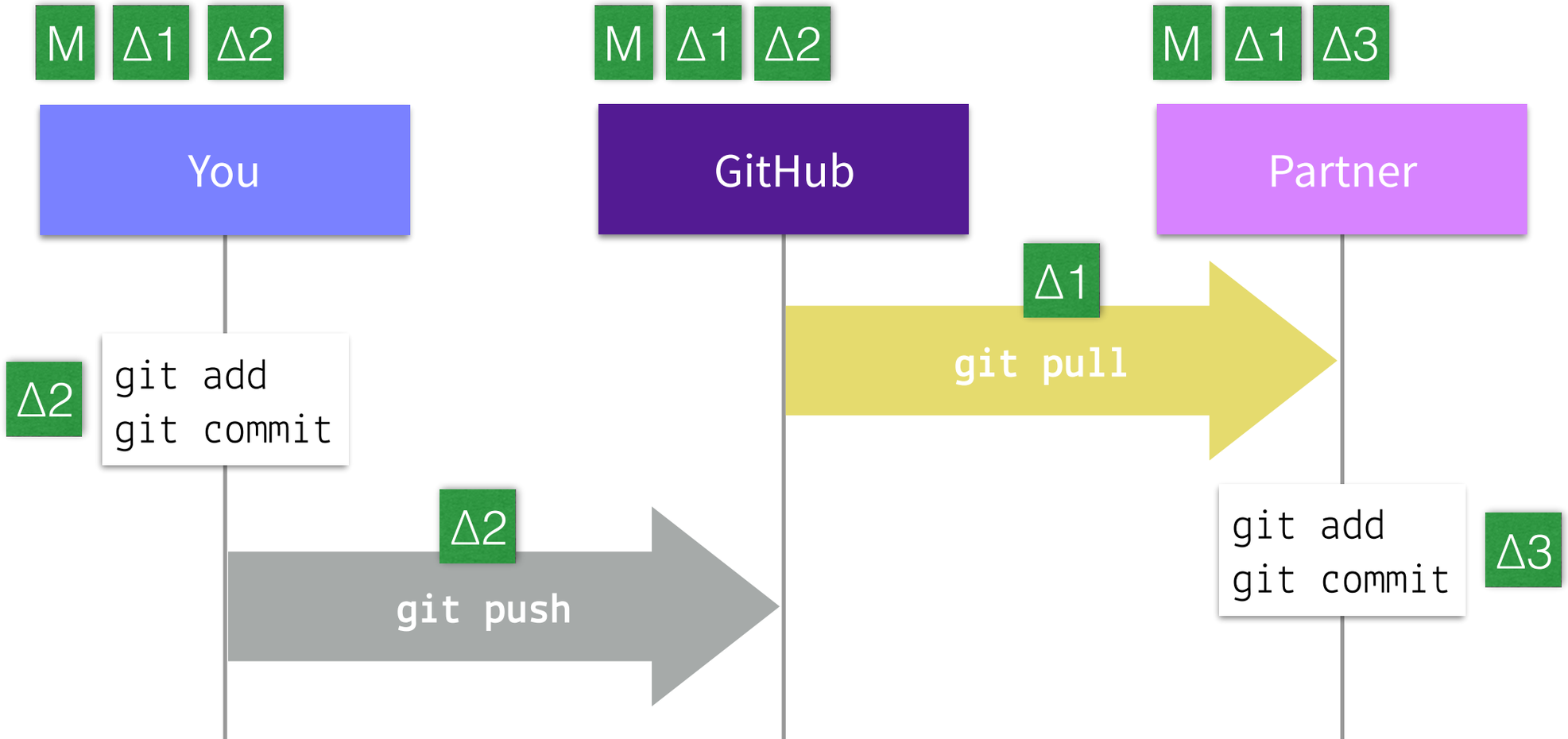
https://git-scm.com/book/en/v2/Getting-Started-Git-Basics

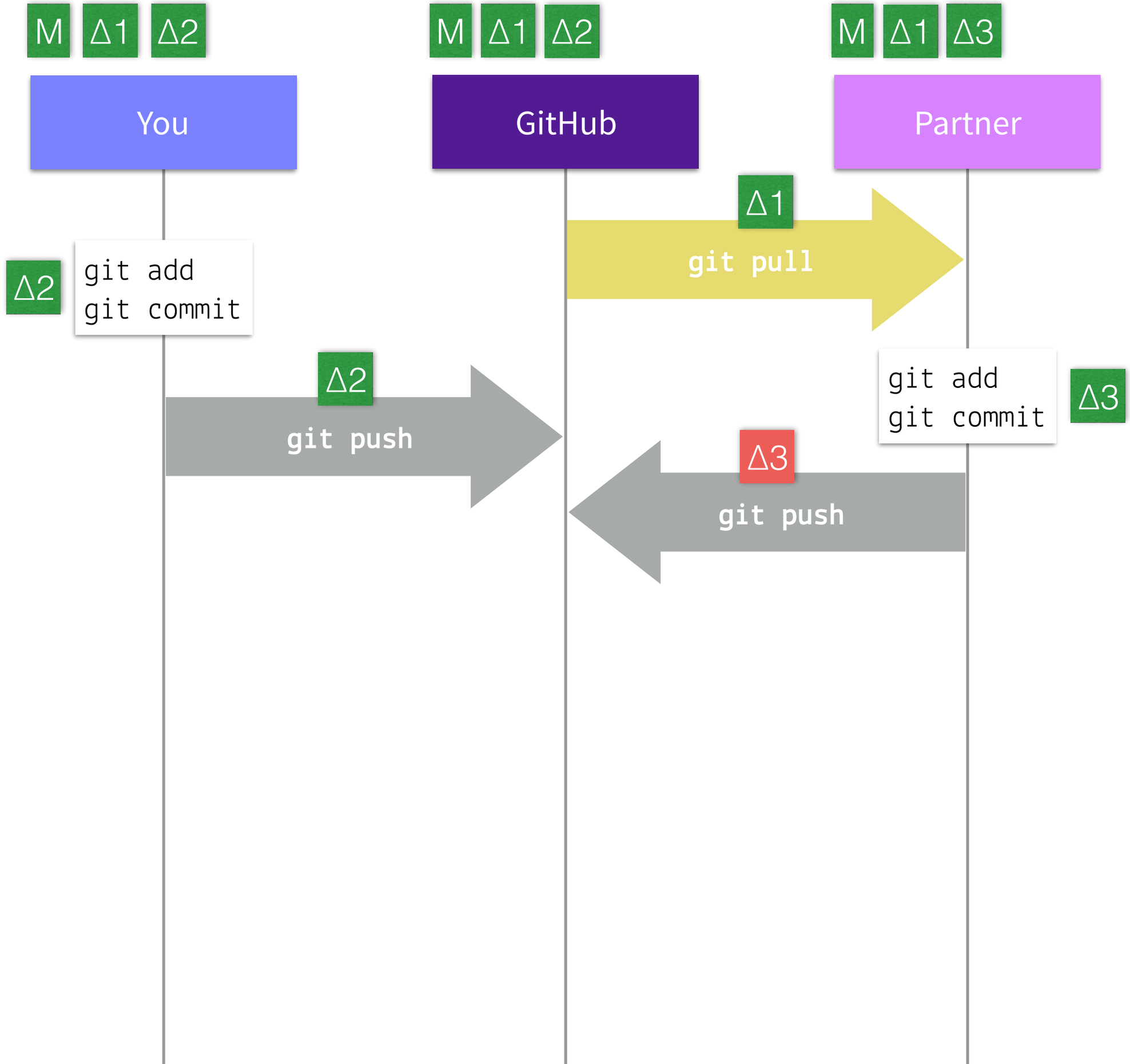

M
Δ1
You
M
Δ1
GitHub
M
Partner

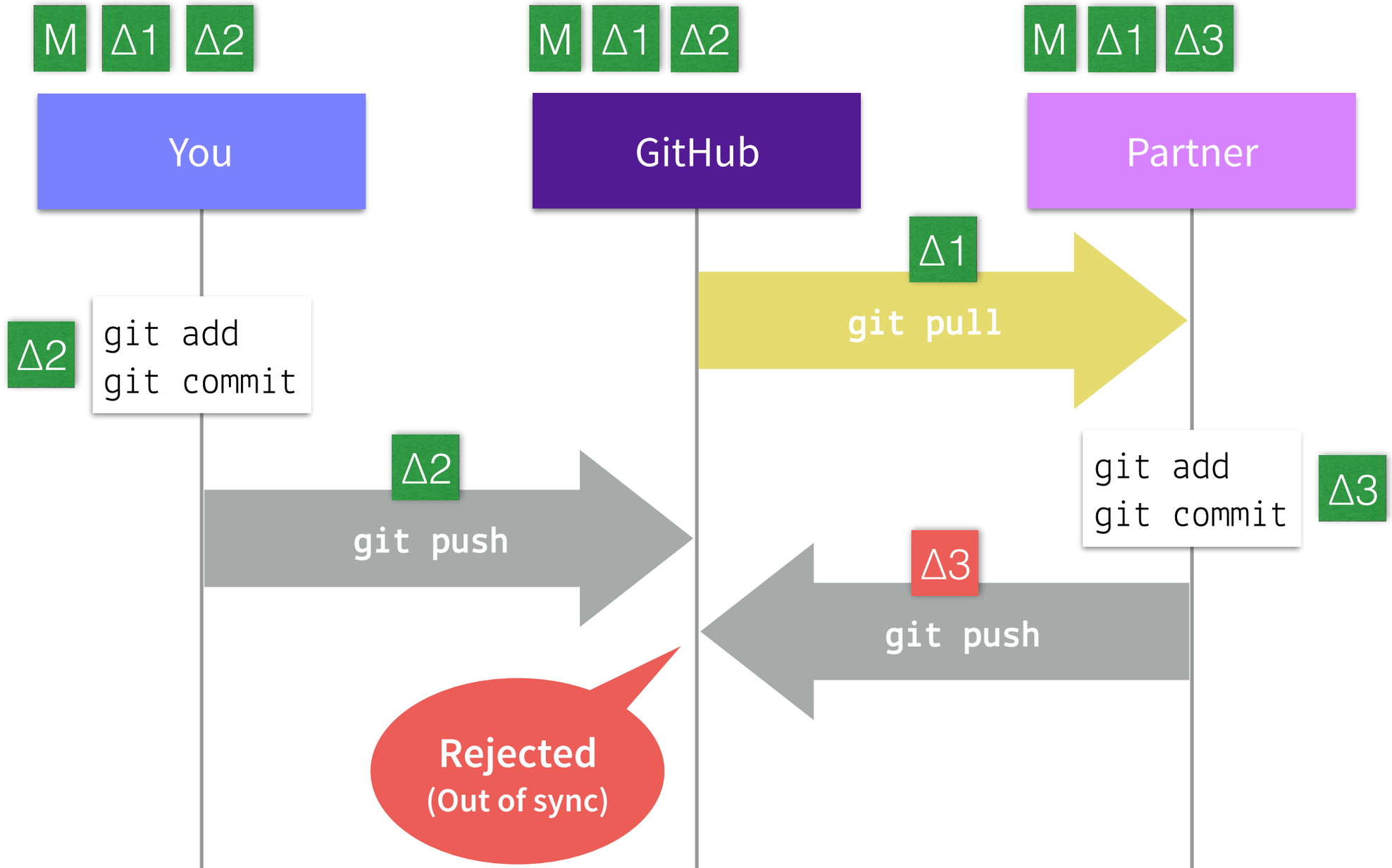

M
Δ1
M
Δ1
M
Δ1
You
GitHub
Partner
Δ1
git pull

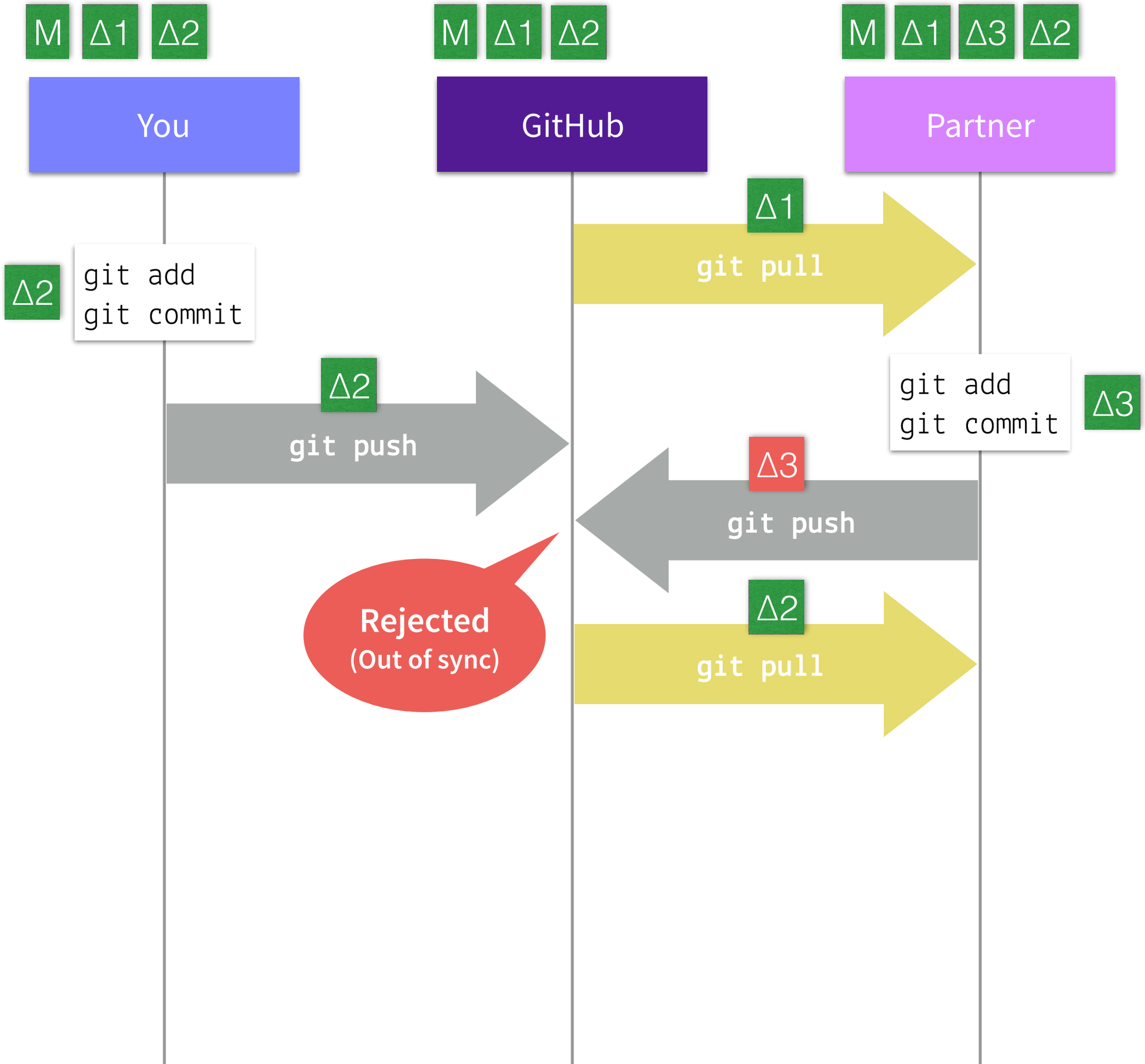

M
Δ1
Δ2
You
GitHub
Partner
Δ1
git pull
Δ2
git add
git commit

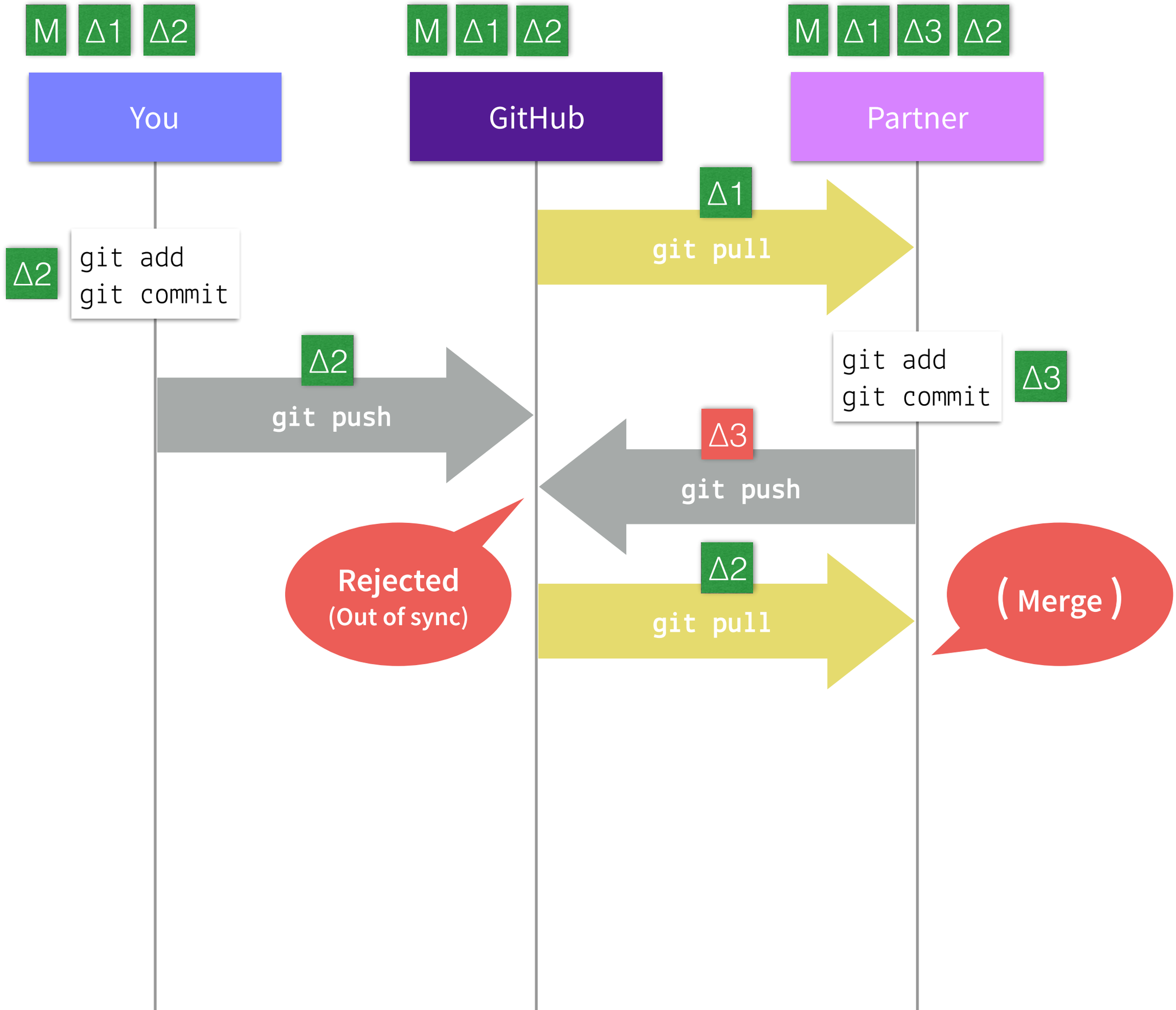

M
Δ1
Δ2
You
GitHub
M
Δ1
Partner
M
Δ1
Δ3
Δ1
git pull
Δ2
git add
git commit
git add
git commit
Δ3

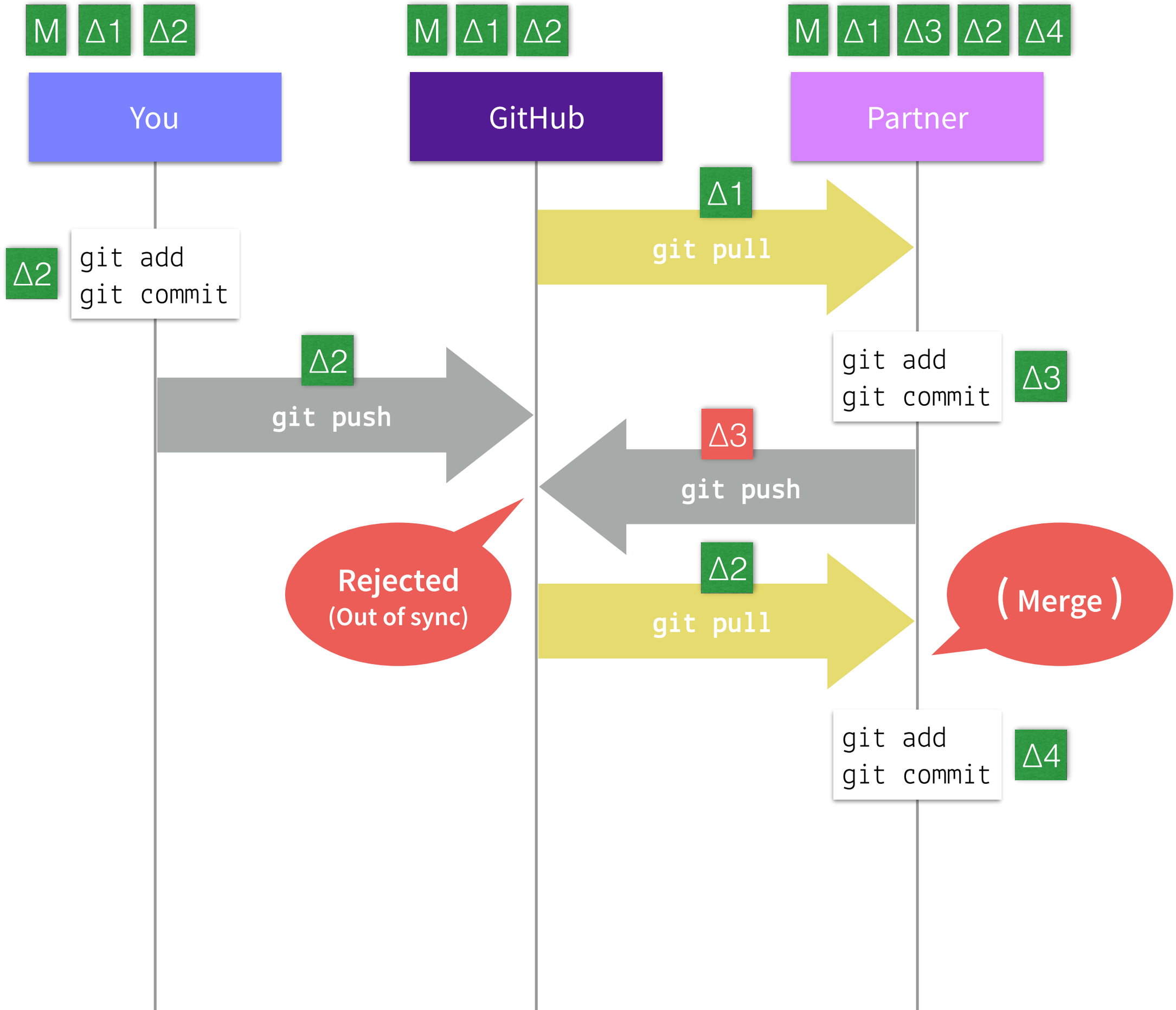

M
Δ1
Δ2
You
GitHub
Partner
M
Δ1
Δ2
M
Δ1
Δ3
Δ1
git pull
Δ2
git add
git commit
Δ2
git push
git add
git commit
Δ3

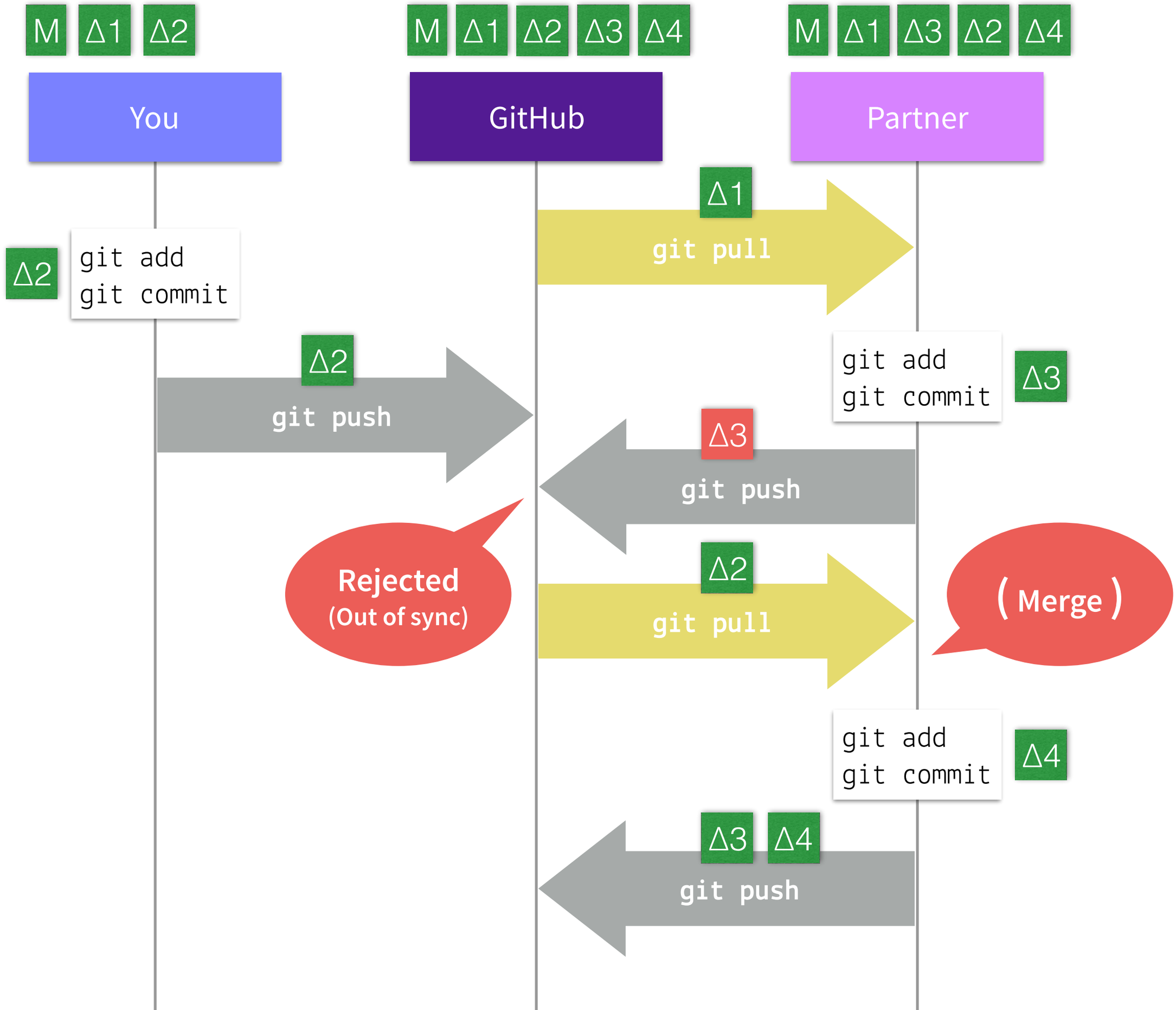

M
Δ1
Δ2
M
Δ1
Δ2
M
Δ1
Δ3
You
GitHub
Partner
Δ1
git pull
Δ2
git add
git commit
Δ2
git push
git add
git commit
Δ3
Δ3
git push

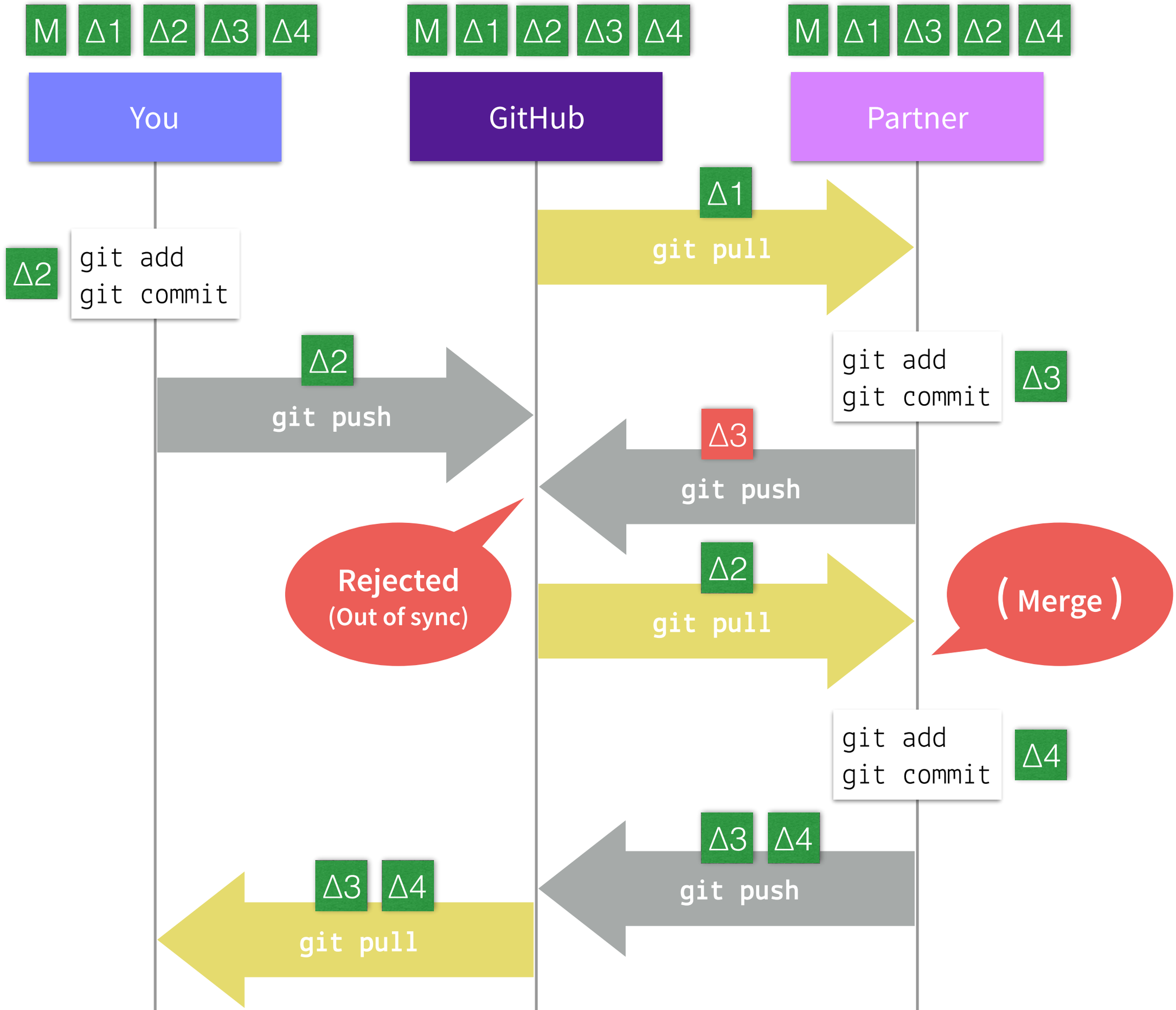

M Δ1 Δ2
M Δ1 Δ2
M Δ1 Δ3
You
GitHub
Partner
Δ1
git pull
Δ2
git add
git commit
Δ2
git push
git add
git commit
Δ3
Δ3
git push
Rejected
(Out of sync)


M Δ1 Δ2
M Δ1 Δ2
M Δ1 Δ3 Δ2
You
GitHub
Partner
Δ1
git pull
Δ2
git add
git commit
Δ2
git push
git add
git commit
Δ3
Δ3
git push
Rejected
(Out of sync)
Δ2
git pull


M Δ1 Δ2
M Δ1 Δ2
M Δ1 Δ3 Δ2
You
GitHub
Partner
Δ1
git pull
Δ2
git add
git commit
Δ2
git push
git add
git commit
Δ3
Δ3
git push
Rejected
(Out of sync)
Δ2
git pull
(Merge)


M Δ1 Δ2
M Δ1 Δ2
M Δ1 Δ3 Δ2 Δ4
You
GitHub
Partner
Δ1
git pull
Δ2
git add
git commit
Δ2
git push
git add
git commit
Δ3
Δ3
git push
Rejected
(Out of sync)
Δ2
git pull
(Merge)
git add
git commit
Δ4


M Δ1 Δ2
M Δ1 Δ2 Δ3 Δ4
M Δ1 Δ3 Δ2 Δ4
You
GitHub
Partner
Δ1
git pull
Δ2
git add
git commit
Δ2
git push
git add
git commit
Δ3
Δ3
git push
Rejected
(Out of sync)
Δ2
git pull
(Merge)
git add
git commit
Δ4
Δ3 Δ4
git push


M Δ1 Δ2 Δ3 Δ4
M Δ1 Δ2 Δ3 Δ4
M Δ1 Δ3 Δ2 Δ4
You
GitHub
Partner
Δ1
git pull
Δ2
git add
git commit
Δ2
git push
git add
git commit
Δ3
Δ3
git push
Rejected
(Out of sync)
Δ2
git pull
(Merge)
git add
git commit
Δ4
Δ3 Δ4
git push
Δ3 Δ4
git pull

Föreläsning 7

Tobias Wrigstad

Testning
CUnit

"Software bugs cost the U.S. economy
an estimated $59.5 billion per year.
An estimated $22.2 billion
could be eliminated by
improved testing that enables
earlier and more effective
identification and removal of defects."

- US Department of Commerce (NIST)

Finn felet!

```
int count_spaces(char *str)
{
  int length = strlen(str);
  int count = 0;
  for(int i = 1; i < length; ++i)
    {
      if(str[i] == ' ')
        count++;
    }
  return(count);
}
```

Finn felet!

```
int count_spaces(char *str)
{
  int length = strlen(str);
  int count = 0;
  for(int i = 1; i < length; ++i)
    {
      if(str[i] == ' ')
        count++;
    }
  return(count);
}
```

Bug: i = 1 istf i = 0 i for-loopen

Inte helt enkel att upptäcka, t.ex.
`str = "A B C"` ser inte felet, men
`str = " L"` ser det

Testning kan bara påvisa förekomsten av fel, inte avsaknaden

— Dijkstra

Ett test som inte kan utföras automatiskt är ett dåligt test!

När du testar P är ditt mål inte att bekräfta att P är felfritt…

…utan att hitta så många fel som möjligt i P

Vad är ett test?

- Låt säga att vi vill testa `f(x) = y` för någon given ”funktion” `f`

  x är indata
  y är utdata
  x och y **tillsammans** är ett **test**, eller **testfall**

- Exempelvis, för `f = ++` (prefix)

```
++42 = 43                    (x = 42, y = 43)
++0 = 1
++-1 = 0
++INT_MAX = INT_MIN   // kompilerar dock ej, men tanken
```

Skall vi verkligen göra 2^{32} tester?

Varför har vi valt just dessa testfall?

x	y	Anmärkning
42	43	Vanlig aritmetik
0	1	Gräns, från ± till positiv
−1	0	Gräns, från negativ till ±
INT_MAX	INT_MIN	Gräns, från positiv till negativ

Testdesign

- Det är svårt att skriva tester
- Kräver kunskap om domänen

  Vilka är gränsfallen?

  Vilka är de vanliga felen?

- Två syner på test

  — *Hur kan jag ha sönder programmet?*

  — *Hur kan jag förbättra programmet?*

Den här kursen nosar bara på testning

- **Enhetstestning** (dagens fokus)

  Pröva en liten del av programmet i isolation

- **Integrationstestning** (under projektet)

  Hur olika moduler/programdelar samverkar

- **Regressionstestning** (under projektet)

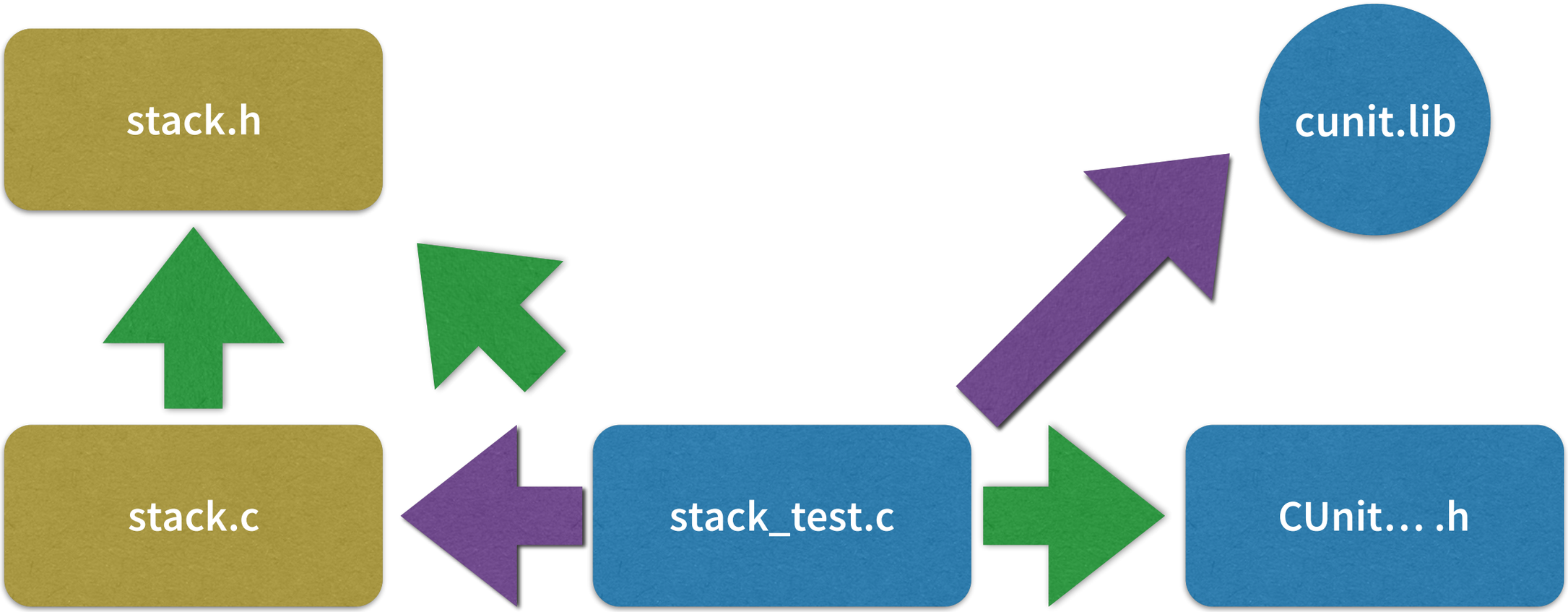
  Undvik regression vid underhåll/buggfixning

  (Regression = återgång till tidigare felbenäget tillstånd)

Det som skall testas

Testerna

Testbibliotek



Ett tests anatomi

- En test = en funktion
- Skapa data som behövs för testet
- Utför testet

  Operationer varvat med ASSERTs

- Riv ned testet

```
void test_stack_creation()
{
  int_stack_t *s = stack_new();

  CU_ASSERT_FALSE(s == NULL);
  CU_ASSERT_TRUE(stack_height(s) == 0);

  stack_free(s);
}
```

- Viktigt att testa så litet som möjligt
- Viktigt att riva skapa och riva ned vid varje test

  Många verktyg (inkl. CUnit) har stöd för att göra det lättare

http://cunit.sourceforge.net/doc/writing_tests.html#assertions

Asserts (35 stycken!)

Assert	Finns Fatal?
CU_ASSERT_TRUE(value)	Ja
CU_ASSERT_FALSE(value)	Ja
CU_ASSERT_EQUAL(actual, expected)	Ja
CU_ASSERT_NOT_EQUAL(actual, expected))	Ja
CU_ASSERT_PTR_EQUAL(actual, expected)	Ja
CU_ASSERT_PTR_NULL(value)	Ja
CU_ASSERT_PTR_NOT_NULL(value)	Ja
CU_ASSERT_STRING_EQUAL(actual, expected)	Ja
CU_FAIL(message)	Ja

Exempel

```
void *data = malloc(2048);
CU_ASSERT_PTR_NOT_NULL(data);
```

```
char buf[256];
scanf("%s", buf);
char *word = my_copy_to_heap_function(buf);
CU_ASSERT_STRING_EQUAL(buf, word);
```

```
treenode_t *t = treenode_new_leaf(key, data);
CU_ASSERT_TRUE(t->key == key);
CU_ASSERT_TRUE(t->data == data);
CU_ASSERT_PTR_NULL(t->left);
CU_ASSERT_PTR_NULL(t->right);
```

inkapsling?

Exempel

```
void *data = malloc(2048);
CU_ASSERT_PTR_NOT_NULL(data);
```

```
char buf[256];
scanf("%s", buf);
char *word = my_copy_to_heap_function(buf);
CU_ASSERT_STRING_EQUAL(buf, word);
```

```
treenode_t *t = treenode_new_leaf(key, data);
CU_ASSERT_TRUE(tn_get_key(t) == key);
CU_ASSERT_TRUE(tn_get_data(t) == data);
CU_ASSERT_PTR_NULL(tn_get_left(t));
CU_ASSERT_PTR_NULL(tn_get_right(t));
```

vad testas?

Regressionstestning

- En (stor) uppsättning enhetstester för olika delar av programmet
- Möjlighet att köra dem automatiskt

- Vid varje förändrings:

  Kör alla enhetstester automatiskt

  Notera vilka som passerar och vilka som inte passerar

  Implementera förändringen

  Kör alla enhetstester automatiskt igen

  Stäm av mot föregående lista — verkar det rimligt?

Table 20-2. Defect-Detection Rates

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%




Från McConnell's Code Complete

Vad skall man testa för något?

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct stack stack_t;
typedef struct node node_t;

struct stack
{
  node_t *top;
};

struct node
{
  int value;
  node_t *next;
};

stack_t *ioopm_stack_create()
{
  return calloc(1, sizeof(stack_t));
}

void ioopm_stack_push(stack_t *s, int value)
{
  s->top = node_create(value, s->top);
}

int ioopm_stack_top(stack_t *s)
{
  return s->top->value;
}
```

```c
int ioopm_stack_pop(stack_t *s)
{
  int value = ioopm_stack_top(s);
  node_t *old_top = s->top;
  s->top = s->top->next;
  free(old_top);
  return value;
}

void ioopm_stack_destroy(stack_t *s)
{
  while (s->top)
    {
      ioopm_stack_pop(s);
    }
  free(s);
}

int ioopm_stack_size(stack_t *s)
{
  int height = 0;
  for (node_t *cursor = s->top;
       cursor;
       cursor = cursor->next)
    {
      ++height;
    }
  return height;
}
```

stack.c

Vad är ett bra test?

Coverage — testtäckning

- För att testkvaliteten skall vara hög vill vi testa så många delar som möjligt av `f`
- För att minska kostnader vill vi undvika fler än ett test som testar samma sak

  Handlar också om omöjligheten att testa alla fall

- Vi skall se `gcov` senare under denna föreläsning

Strukturell testtäckning

Ut-parameter mha pekare

```
int ioopm_stack_pop(stack_t *s, int v, bool *underflow)
{
  if (s->top == NULL)
    {
      *underflow = true;
    }
  else
    {
      *underflow = false;
    }
  …
```

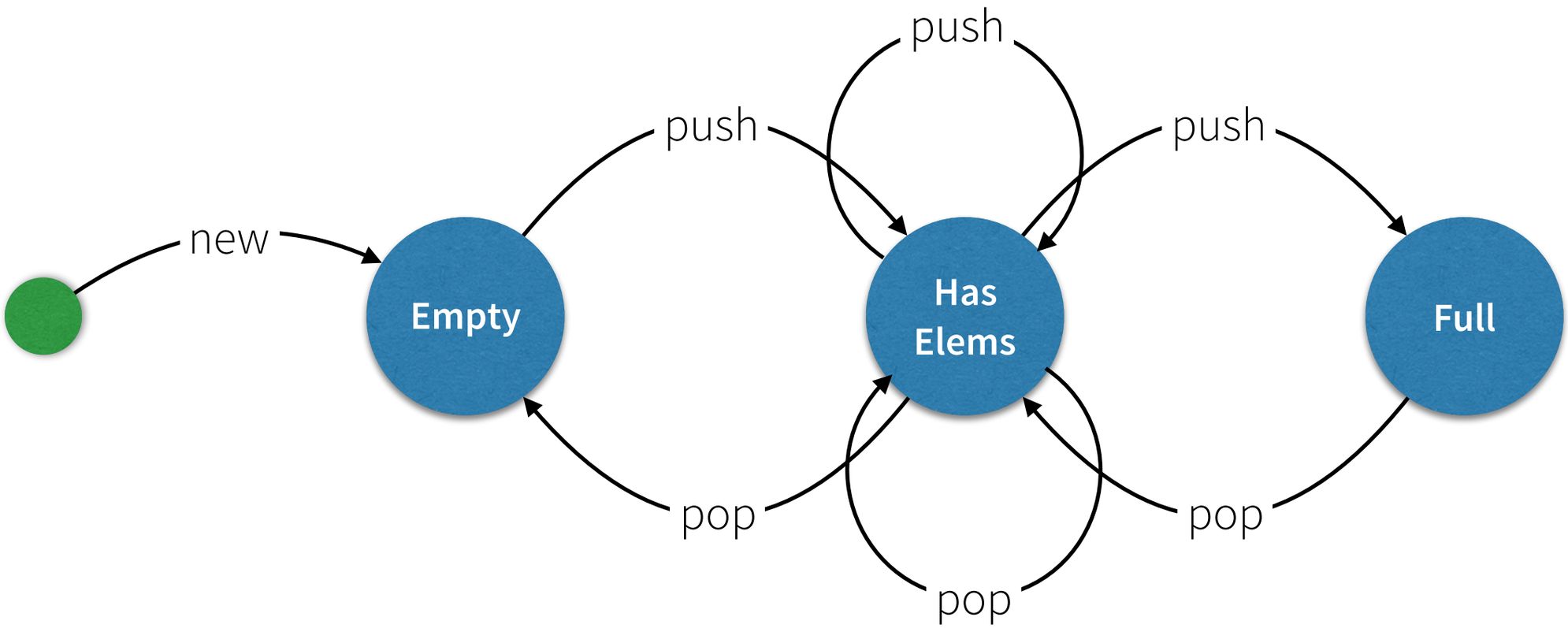
- **Naiv strategi:** plocka slumpmässiga indata
- **Problem:** är inte säkra på att vi prövar båda vägar genom `if`-satsen
- **Slutsats:** vi behöver nog med testfall för att pröva båda ”branches”

Funktionsorienterad testtäckning

```
int ioopm_stack_size(int_stack_t *s)
```

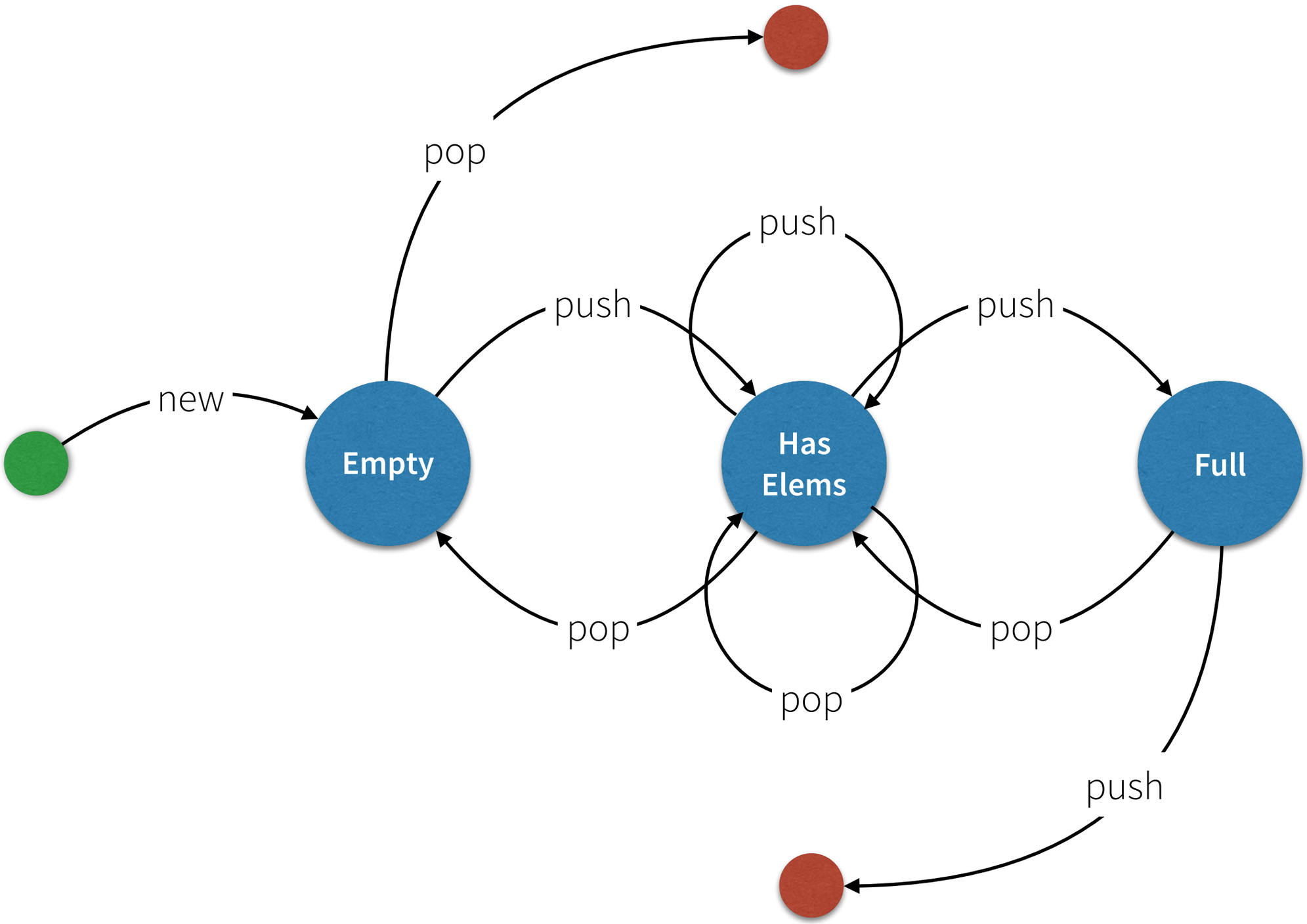
- **Naiv strategi:** skapa många stackar och pröva deras storlek
- **Problem:** inte säkert att vi prövar alla fall (tom stack, full stack, etc.)
- **Slutsats:** behöver fall som täcker alla möjliga *klasser av* input

Beteende-orienterad testtäckning

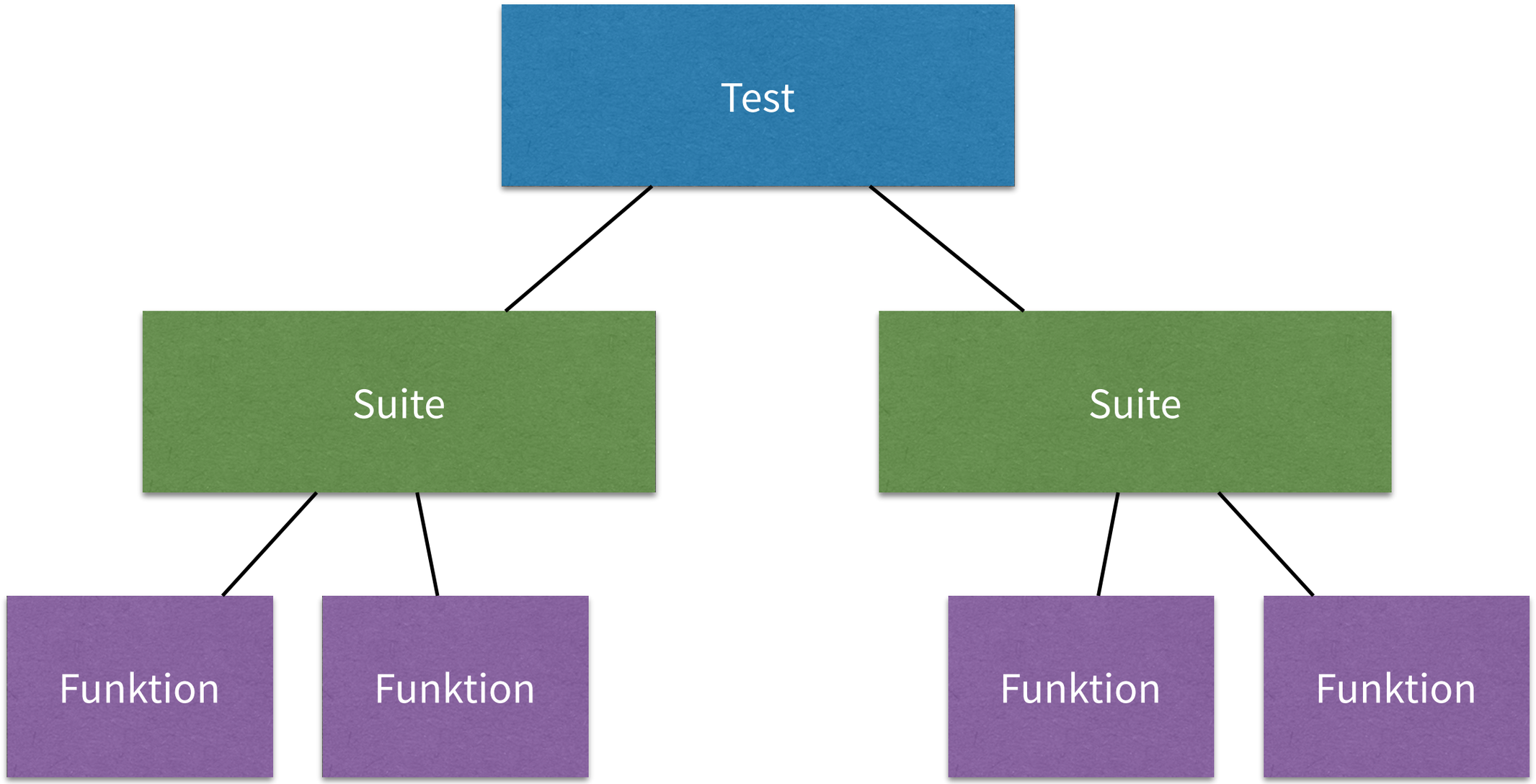


- **Naiv strategi:** plocka slumpmässiga värden på x och y och pröva x == y
- **Problem:** är inte säkra på att vi prövar alla viktiga fall
- **Slutsats:** vi behöver testfall som prövar alla tillstånd och tillståndsövergångar

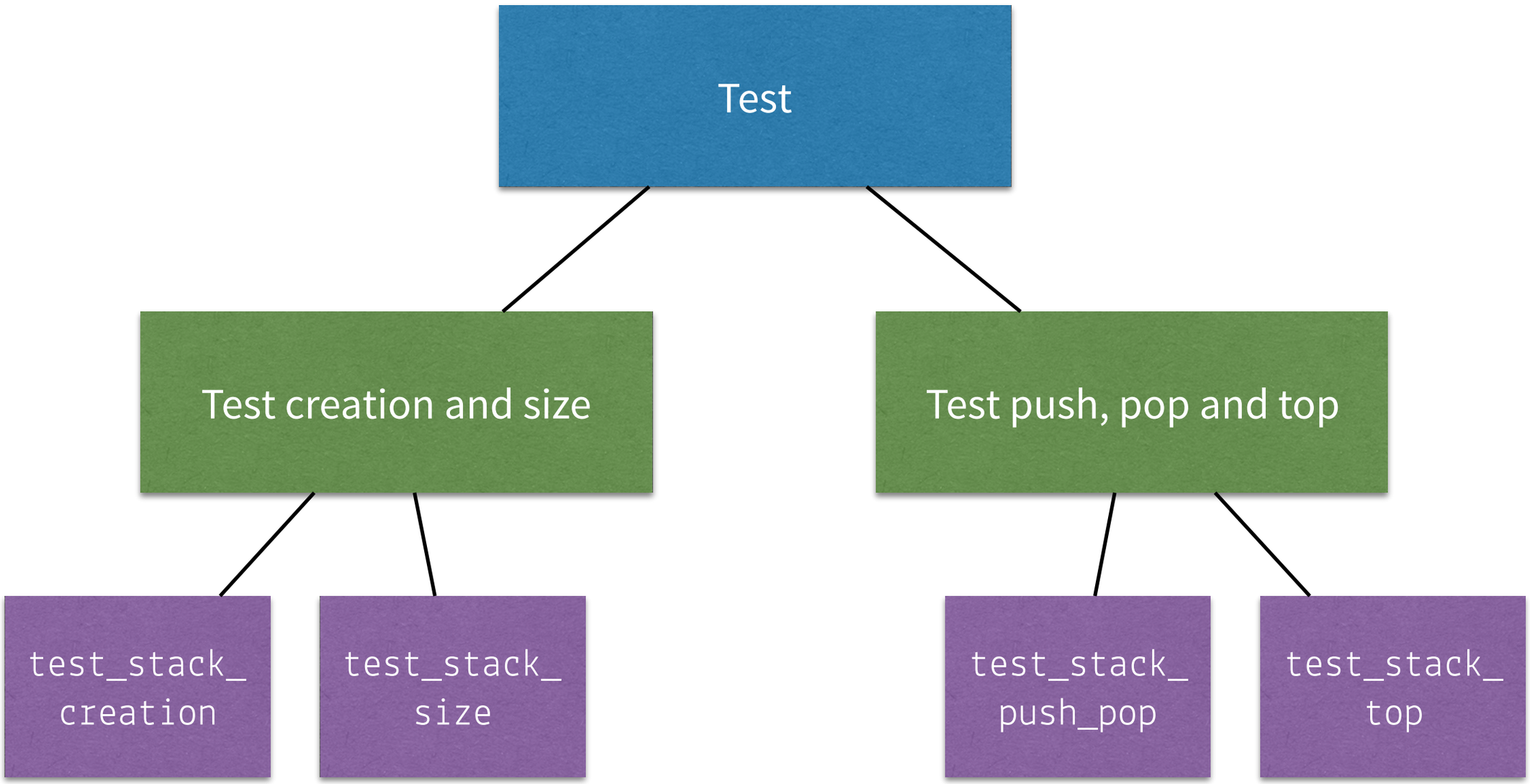
Beteende-orienterad testtäckning



Ett testprogram i CUnit



stack_test.c


Test
Test creation and size
Test push, pop and top
test_stack_creation
test_stack_size
test_stack_push_pop
test_stack_top

main()

```
int main(int argc, char *argv[])
{
  // Initialise
  CU_initialize_registry();

  // Set up suites and tests
  CU_pSuite creation = CU_add_suite("Test creation and height", NULL, NULL);
  CU_add_test(creation, "Creation", test_stack_creation);
  CU_add_test(creation, "Size", test_stack_size);

  CU_pSuite pushpoptop = CU_add_suite("Test push, pop and top", NULL, NULL);
  CU_add_test(pushpoptop, "Push and pop", test_stack_push_pop);
  CU_add_test(pushpoptop, "Top", test_stack_top);

  // Actually run tests
  CU_basic_run_tests();

  // Tear down
  CU_cleanup_registry();
  return 0;
}
```

Skapa en stack

```
void test_stack_creation()
{
  stack_t *s = ioopm_stack_create();

  CU_ASSERT_FALSE(s == NULL);
  CU_ASSERT_TRUE(ioopm_stack_size(s) == 0);

  ioopm_stack_destroy(s);
}
```

Stackhöjden

```
void test_stack_size()
{
  stack_t *s = ioopm_stack_create();

  CU_ASSERT_TRUE(ioopm_stack_size(s) == 0);
  ioopm_stack_push(s, 0);
  CU_ASSERT_TRUE(ioopm_stack_size(s) == 1);
  ioopm_stack_push(s, 0);
  CU_ASSERT_TRUE(ioopm_stack_size(s) == 2);
  ioopm_stack_pop(s);
  CU_ASSERT_TRUE(ioopm_stack_size(s) == 1);
  ioopm_stack_pop(s);
  CU_ASSERT_TRUE(ioopm_stack_size(s) == 0);

  ioopm_stack_destroy(s);
}
```

Lägga till och ta bort

```
void test_stack_push_pop()
{
  stack_t *s = ioopm_stack_create();
  const int values[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  for (int i = sizeof(values) / sizeof(values[0]); i > 0; --i)
    {
      ioopm_stack_push(s, values[i - 1]);
    }

  for (int i = 0; i < sizeof(values) / sizeof(int); ++i)
    {
      CU_ASSERT_TRUE(ioopm_stack_pop(s) == values[i]);
    }

  ioopm_stack_destroy(s);
}
```

Titta på översta elementet

```
void test_stack_top()
{
  stack_t *s = ioopm_stack_create();

  ioopm_stack_push(s, 1);
  CU_ASSERT_TRUE(ioopm_stack_top(s) == 1);
  ioopm_stack_push(s, 20);
  CU_ASSERT_TRUE(ioopm_stack_top(s) == 20);
  ioopm_stack_pop(s);
  CU_ASSERT_TRUE(ioopm_stack_top(s) == 1);

  ioopm_stack_destroy(s);
}
```

```
$ gcc -o stack_test stack_test.c stack.c -lcunit


$ ./stack_test

     CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      2      2    n/a      0        0
               tests      4      4      4      0        0
             asserts     20     20     20      0      n/a

Elapsed time =    0.000 seconds
```

```
$ gcc -o stack_test stack_test.c stack.c -lcunit


$ ./stack_test

     CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      2      2    n/a      0        0
               tests      4      4      4      0        0
             asserts     20     20     20      0      n/a

Elapsed time =    0.000 seconds
```

```
$ gcc -o stack_test stack_test.c stack.c -lcunit


$ ./stack_test

     CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      2      2    n/a      0        0
               tests      4      4      4      0        0
             asserts     20     20     20      0      n/a

Elapsed time =    0.000 seconds
```

Sök eller skriv in namn på webbplats

CUnit - A Unit testing framework for C.
http://cunit.sourceforge.net/

Automated Test Run Results

Running Suite Test creation and height		
	Running test Creation ...	Passed
	Running test Height ...	Passed
Running Suite Test push, pop and top		
	Running test Push and pop ...	Passed
	Running test Top ...	Passed

Cumulative Summary for Run					
Type	**Total**	**Run**	**Succeeded**	**Failed**	**Inactive**
Suites	2	2	- NA -	0	0
Test Cases	4	4	4	0	0
Assertions	20	20	20	0	n/a

File Generated By CUnit v2.1-3 - Sun Sep 20 23:54:41 2015

Visa en meny

```
#include <CUnit/Automated.h>
CU_automated_run_tests(); // run and generate XML file
```

```
$ gcc --coverage -o stack_test stack_test.c stack.c -lcunit
```

```
$ gcc --coverage -o stack_test stack_test.c stack.c -lcunit

$ ./stack_test
...

$ ./gcov stack_test.c stack.c
File 'stack_test.c'
Lines executed:100.00% of 45
stack_test.c:creating 'stack_test.c.gcov'

File 'stack.c'
Lines executed:72.41% of 29
stack.c:creating 'stack.c.gcov'
```

```
$ gcov -abcfu stack.c
Function 'stack_new'
Lines executed:100.00% of 5
Branches executed:100.00% of 2
Taken at least once:50.00% of 2
No calls

Function 'stack_free'
Lines executed:100.00% of 2
No branches
No calls

Function 'stack_push'
Lines executed:100.00% of 6
Branches executed:100.00% of 2
Taken at least once:100.00% of 2
No calls

Function 'stack_pop'
Lines executed:100.00% of 6
Branches executed:100.00% of 2
Taken at least once:100.00% of 2
No calls
```

a = all blocks

b = branch probabilities

c = branch counts

f = function summaries

u = unconditional branches

```
$ gcov -abcfu stack.c
Function 'stack_new'
Lines executed:100.00% of 5
Branches executed:100.00% of 2
Taken at least once:50.00% of 2
No calls

Function 'stack_free'
Lines executed:100.00% of 2
No branches
No calls

Function 'stack_push'
Lines executed:100.00% of 6
Branches executed:100.00% of 2
Taken at least once:100.00% of 2
No calls

Function 'stack_pop'
Lines executed:100.00% of 6
Branches executed:100.00% of 2
Taken at least once:100.00% of 2
No calls
```

```c
stack_t *ioopm_stack_new()
{
  stack_t *result = malloc(...);
  if (result)
    {
      *result = (stack_t) { .height = 0 };
    }
  return result;
}
```

a = all blocks

b = branch probabilities

c = branch counts

f = function summaries

u = unconditional branches

LCOV - code coverage report

		Hit	Total	Coverage
Current view: top level - testing	**Lines:**	66	74	89.2 %
Test: app.info	**Functions:**	11	12	91.7 %
Date: 2015-09-21 00:39:13				

Filename	Line Coverage		Functions	
stack.c	72.4 %	21 / 29	85.7 %	6 / 7
stack_test.c	100.0 %	45 / 45	100.0 %	5 / 5

Generated by: LCOV version 1.11

Exempel på hur man kan använda front-ends till gcov för bättre visualisering

Öppna problem ännu så länge

- Hur kontrollerar vi sido-effekter?

  T.ex. `ioopm_stack_destroy()`

  Utskrifter i terminalfönstret eller filer skrivna på hårddisken

Skapa era egna enhetstester

- Utgå gärna från `stack_test.c` som en mall för era egna tester
- Låt er inspireras av de `ASSERT`s som finns i CUnit-dokumentationen
- **Gränsvärden!**
- Tänk på att mäta code coverage för att få en uppfattning om testens kvalitet
- Automatisera allt! (Se exempel från denna föreläsnings utdelade material)

```
make test
```

Några tips vid test av länkad lista

- Tomma listan
- `f` i början, i mitten och i slutet

  `f` = borttagning, insättning, etc.

- Inkapsling

  Växer storleken som den skall?

  Testa sortering genom att göra insättning och kolla index

  Testa dubletthantering genom att kolla om storleken växer vid dublettinsättning

  …

Några tips vid test av binärt sökträd

- Tomma trädet
- `f` vid ingen förälder men barn, vid både förälder och barn, och bara förälder

  `f` = borttagning, insättning, etc.

- Inkapsling

  Växer djup och storlek som de skall?

  Testa balansering med hjälp av djupet

  Testa dubletter med hjälp av storlek

  Kan tillhandahålla funktioner för att kontrollera ”layout” etc. utan direkt åtkomst

Exempel: testa lagerdatabas

- Börja med kraven!

  Dubletter?
  Kortaste avstånd?
  …etc.

- Den hypotetiska funktionen ”Packa en pall”

  Lägg i en vara, kolla att pris stämmer
  Lägg i en till, ändrades priset korrekt
  Vad händer vid borttaggning?
  …etc.

- Notera alla specialfall i din kod och testa extra för dem!

Testdriven utveckling

- Vid implementation av funktionen f, ta reda på vad f skall göra

  Skriv ned detta i termer av tester

  Gör detta innan du börjar implementera funktionen

- Börja implementera, målet är att få testerna att passera

  Varje testfall kan vara en enhet att implementera

  Du har tydliga kriterier för när du är klar

- **Svårt att göra i praktiken — öva och ge inte upp!**

Föreläsning 7–8

Tobias Wrigstad

Byggverktyg

```
$ gcc stack_test.c stack.c -lcunit
$ ./a.out
SEGMENTATION FAULT…
$ gdb ./a.out…
$ gcc -g stack_test.c stack.c -lcunit
$ ./a.out
SEGMENTATION FAULT…
$ gdb ./a.out…
$ emacs stack.c
$ gcc -g stack_test.c stack.c -lcunit
$ ./a.out
SEGMENTATION FAULT…
$ gdb ./a.out…
$ emacs stack.c
$ gcc stack_test.c stack.c -lcunit
$ gcc -g stack_test.c stack.c -lcunit
$ ./a.out
SEGMENTATION FAULT…
$ gdb ./a.out…
$ emacs stack.c
$ gcc -g stack_test.c stack.c -lcunit
$ ./a.out
SEGMENTATION FAULT…
$ gdb ./a.out…
```

Varför byggverktyg?

- Ett program består ofta av många filer
- Vi vill kunna bygga programmet på olika sätt

  e.x: för produktion (-O2), för test (-g --coverage), etc.

- Vi vill bara bygga om de filer som behöver byggas om efter en ändring

  för att minimera byggtiden

- Vi vill enkelt kunna ändra på vilka flaggor som används för att bygga programmet

  e.x.: lägga till en länkflagga

Makefiler

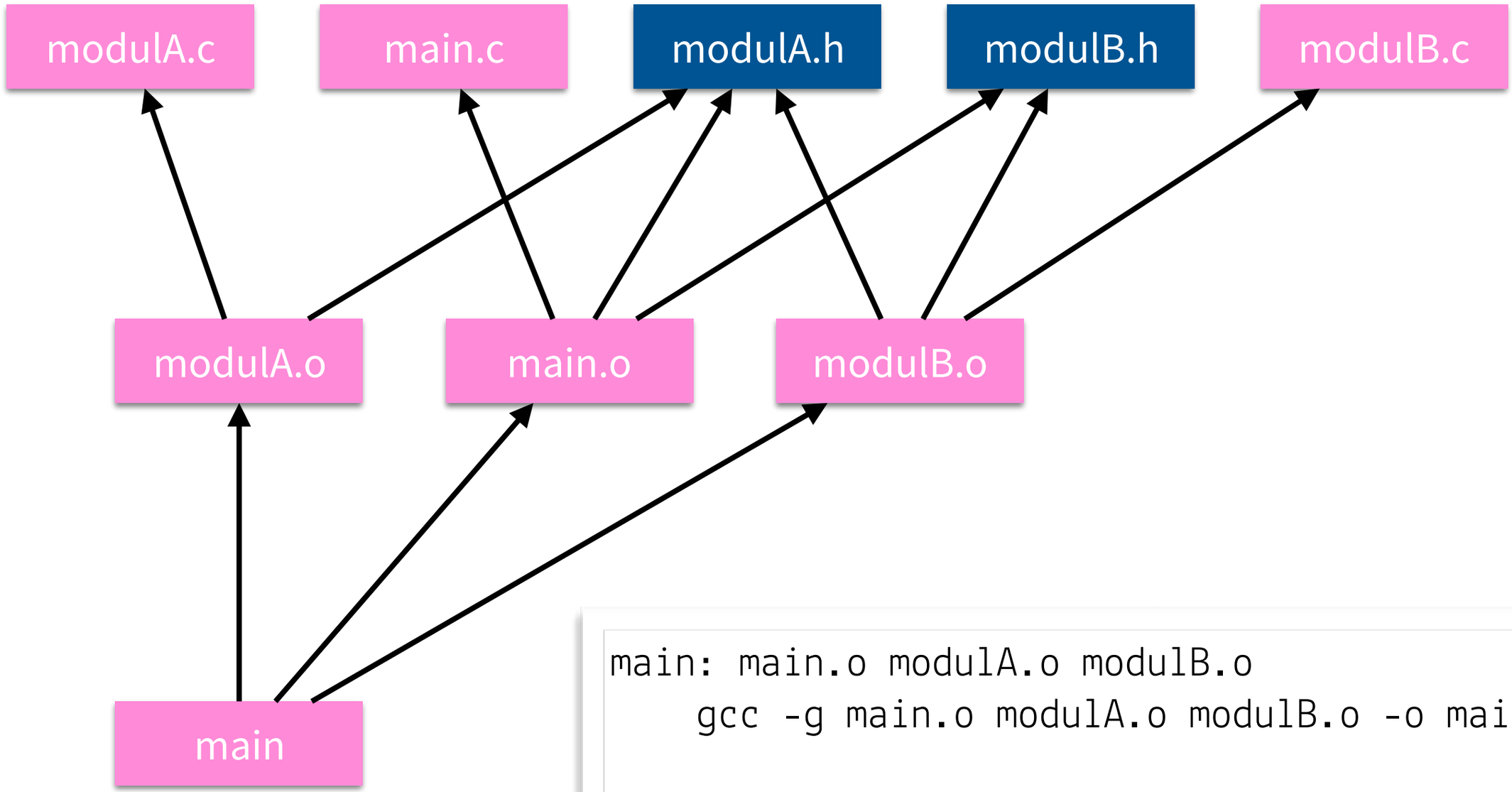
mål

beroenden (ofta målfiler i andra mål)

```
main: main.o modulA.o modulB.o
	gcc -g main.o modulA.o modulB.o -o main
```

kommando som utförs om målfilen inte finns eller är äldre än någon av de filer den är beroende av

Döps till `Makefile`, kör med `make`



Exempel:

```
make main.o
make main
```

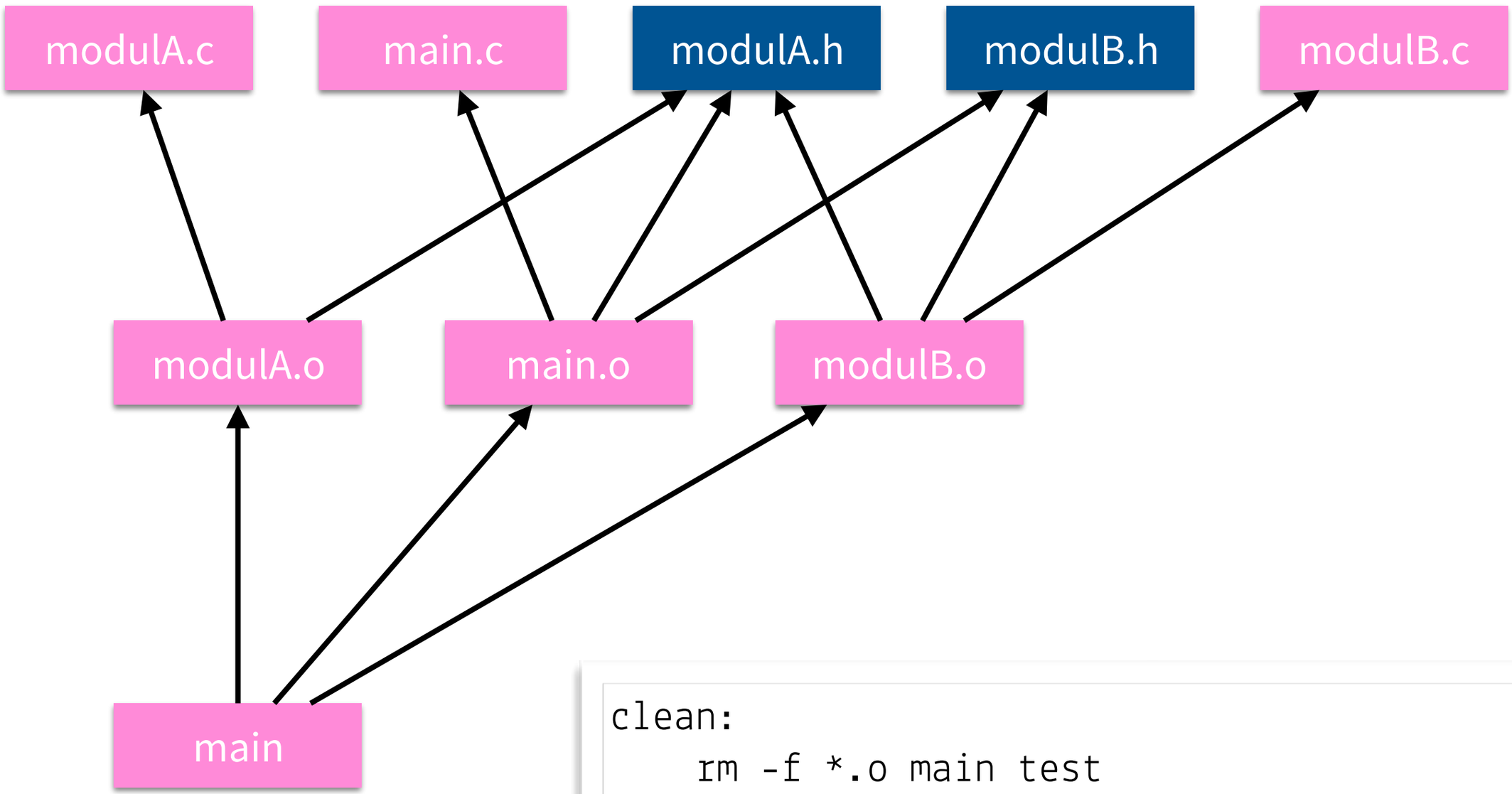
```
main: main.o modulA.o modulB.o
    gcc -g main.o modulA.o modulB.o -o main

main.o: main.c modulA.h modulB.h
    gcc -c -g -Wall main.c

modulA.o: modulA.c modulA.h
    gcc -c -g -Wall modulA.c

modulB.o: modulB.c modulA.h modulB.h
    gcc -c -g -Wall modulB.c
```

Makefile



Exempel:

```
make test
make clean
```

```
clean:
    rm -f *.o main test

memtest: test
    valgrind --leak-check=full ./test < input

test: modulA.o modulB.o tests.c
    gcc modulA.o modulB.o tests.c -o test
    ./test
```

Makefile

Variabler och wildcards

```
main: myprog

C_COMPILER     = gcc
C_OPTIONS      = -Wall -pedantic -g
C_LINK_OPTIONS = -lm
CUNIT_LINK     = -lcunit

%.o:  %.c
  $(C_COMPILER) $(C_OPTIONS) $? -c

myprog: file1.o file2.o file3.o
  $(C_COMPILER) $(C_LINK_OPTIONS) $? -o $@

test1: mytests1.o file1.o
  $(C_COMPILER) $(C_LINK_OPTIONS) $(CUNIT_LINK) $? -o $@

clean:
  rm -f *.o myprog
```

Make är inte världens bästa byggverktyg

- Funkar jättebra för t.ex. C
- Funkar inte så bra för Java, t.ex. — för att namnstandarden inte funkar
- Finns många alternativ: cmake, premake, ant, etc.

  Många DSL:er kompilerar *till* make

- Korsa den bron när du kommer till den — nu räcker make fint!

Föreläsning 8

Tobias Wrigstad

Programdesign — en introduktion

Syftet med mjukvara: **att hjälpa människor**

- Specifika grupper (ekonomer, blinda, uttråkade, informationstörstande, etc.)
- Generella grupper
- Målgruppen är **människor**

Syftet med programdesign blir därför…

- Att möjliggöra implementation av så hjälpsamma program som möjligt
- Att möjliggöra implementation av program som fortsätter att vara hjälpsamma över tid
- Att göra det möjligt att både skapa och underhålla system så enkelt som möjligt så att de två ovanstående målen infrias

$$\underset{\text{desirability}}{D} = \frac{\overset{\text{value}}{V}}{\underset{\text{effort}}{E}}$$

The desirability of any change is directly proportional to the value of the change and inversely proportional to the effort involved in making the change.

- Värdena V och E är egentligen ”vektorer” — som ändras över tid
- Vi pratar gränsvärden

Value

- Värde

  Hur **många** människor blir påverkade?

  Hur **mycket** blir de påverkade?

- Hur hantera negativt värde — **harm**

- Ett programs/features värde kan förändras över tid

  Förväntningar ändras

  Världen ändras (ej kompatibelt längre)

  Kanske värt 0 efter Brexit?

Effort

- I regel ett mått uttryckt i persontimmar — hur lång tid tar det att **färdigställa** och **underhålla** feature F?

  X timmar för att implementera feature F — sedan Y timmar per månad så länge F är i bruk…

- Underhållskostnaden för en feature är ofta den dominerande faktorn

***Följdsats**: det är vettigt att minimera underhållskostnaden på bekostnad av utvecklingskostnaden!*

	Effort
Implementation	1000 timmar
Månad 1	50 timmar
Månad 2	50 timmar
Månad 3	50 timmar
Månad 4	50 timmar
…	50 timmar

När är det ”värt det”?

	Effort	Value
Implementation	1000 timmar	n/a
Månad 1	50 timmar	1000 u
Månad 2	55 timmar	800 u
Månad 3	60 timmar	600 u
Månad 4	65 timmar	400 u
…	70 timmar	200 u

The quality level of your design should be proportional to the length of future time in which your system will continue to help people

Kvalitet styr kostnaden

The quality level of your design should be proportional to <u>the length of future time in which your system will continue to help people</u>

- ”Och hur länge är det om jag får fråga?”
- Programs livslängd är svåra att förutse

  ex: Twitter
  PPM
  JavaScript

- Det enda vi vet om framtiden är att den kommer att vara annorlunda. **Vi skall vara försiktiga med våra förutsägelser!**

Underhåll och förändring

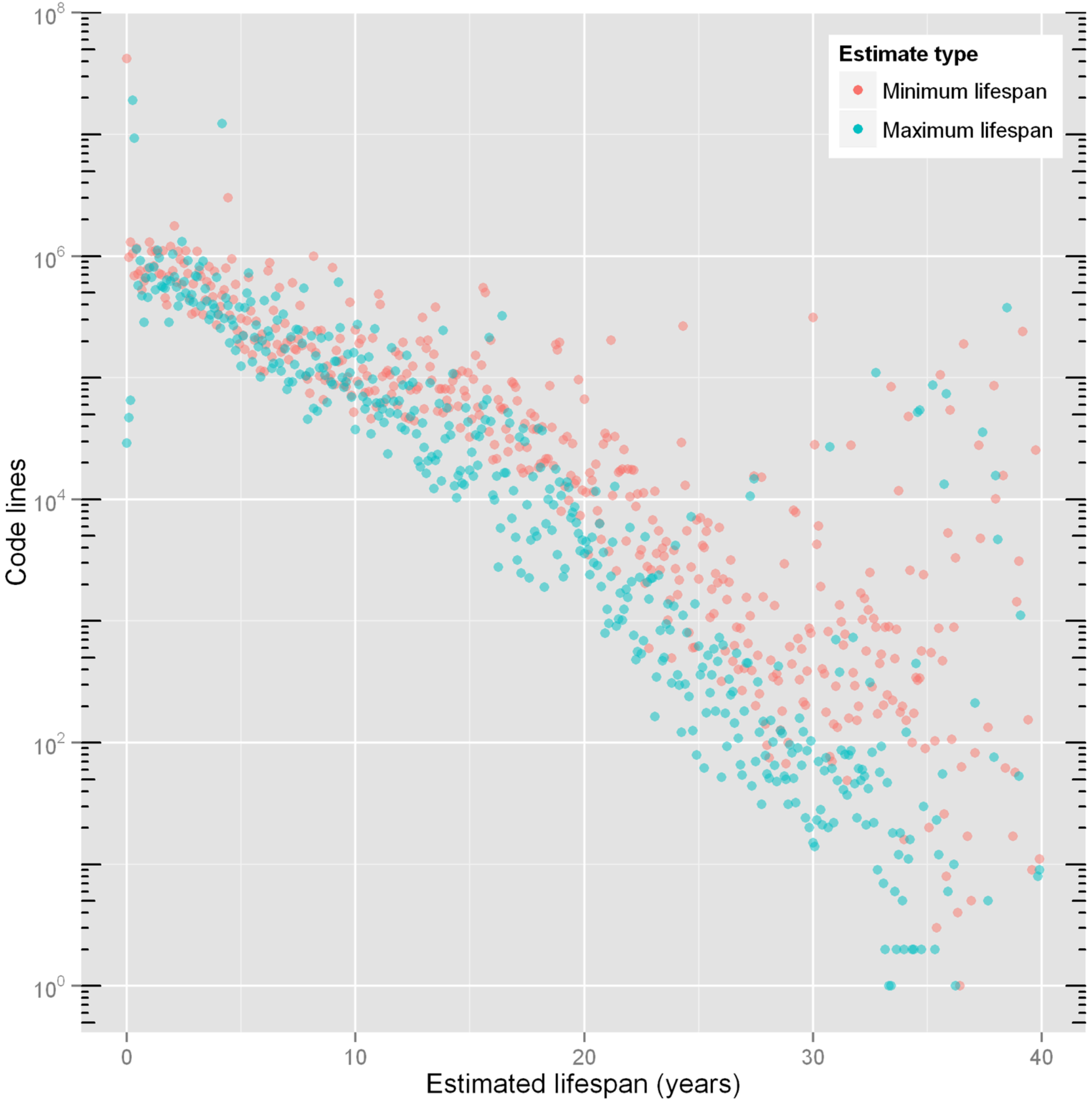
- Sannolikheten för att någon del i ett program måste ändras ökar ju längre programmet existerar
- Ju längre livslängd — ju fler, ju större, förändringar

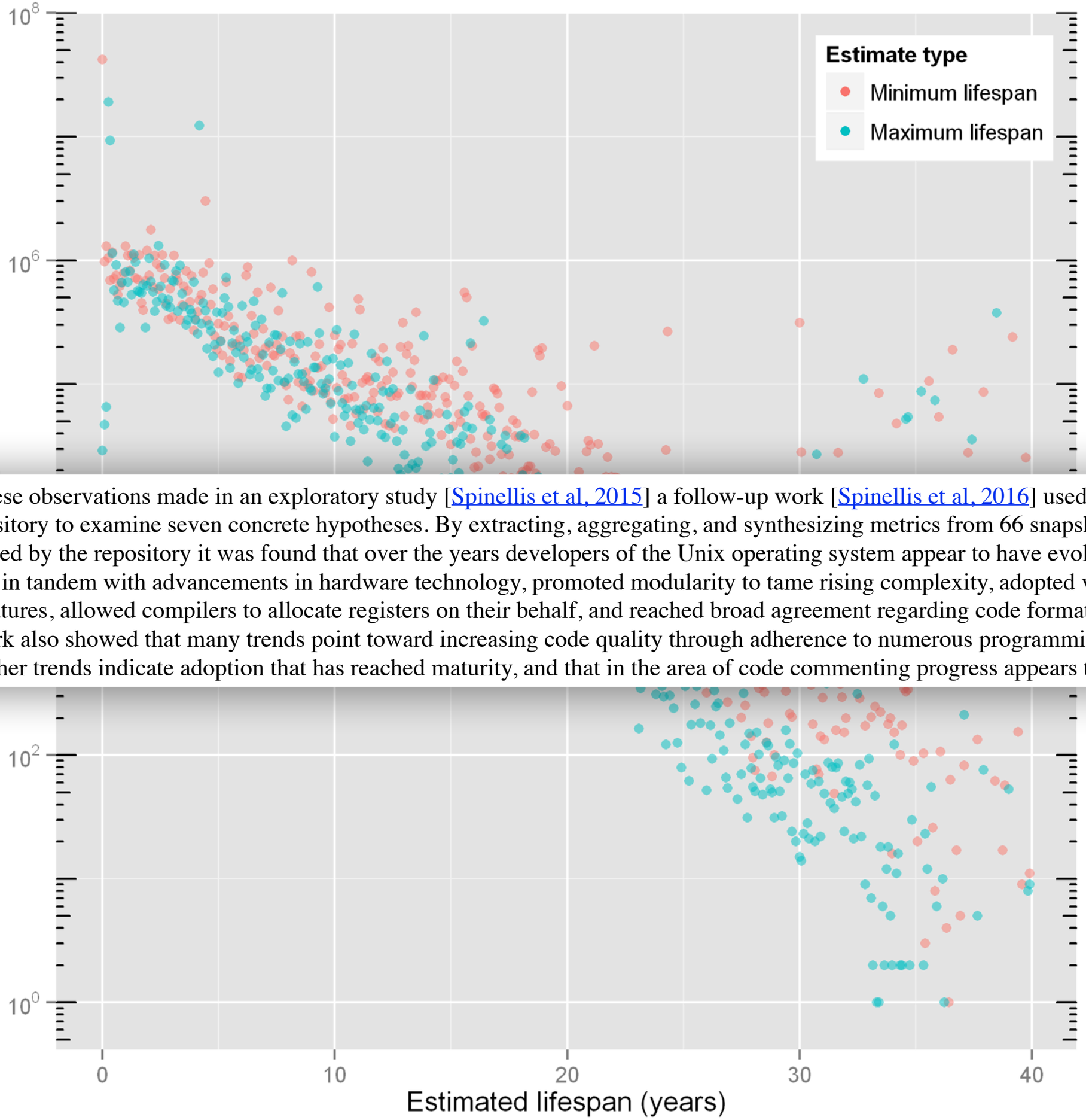
Kanske inte imorgon. Kanske inte så mycket nästa vecka. Men inom 10 år...

Table 5-1. Changes in files over time

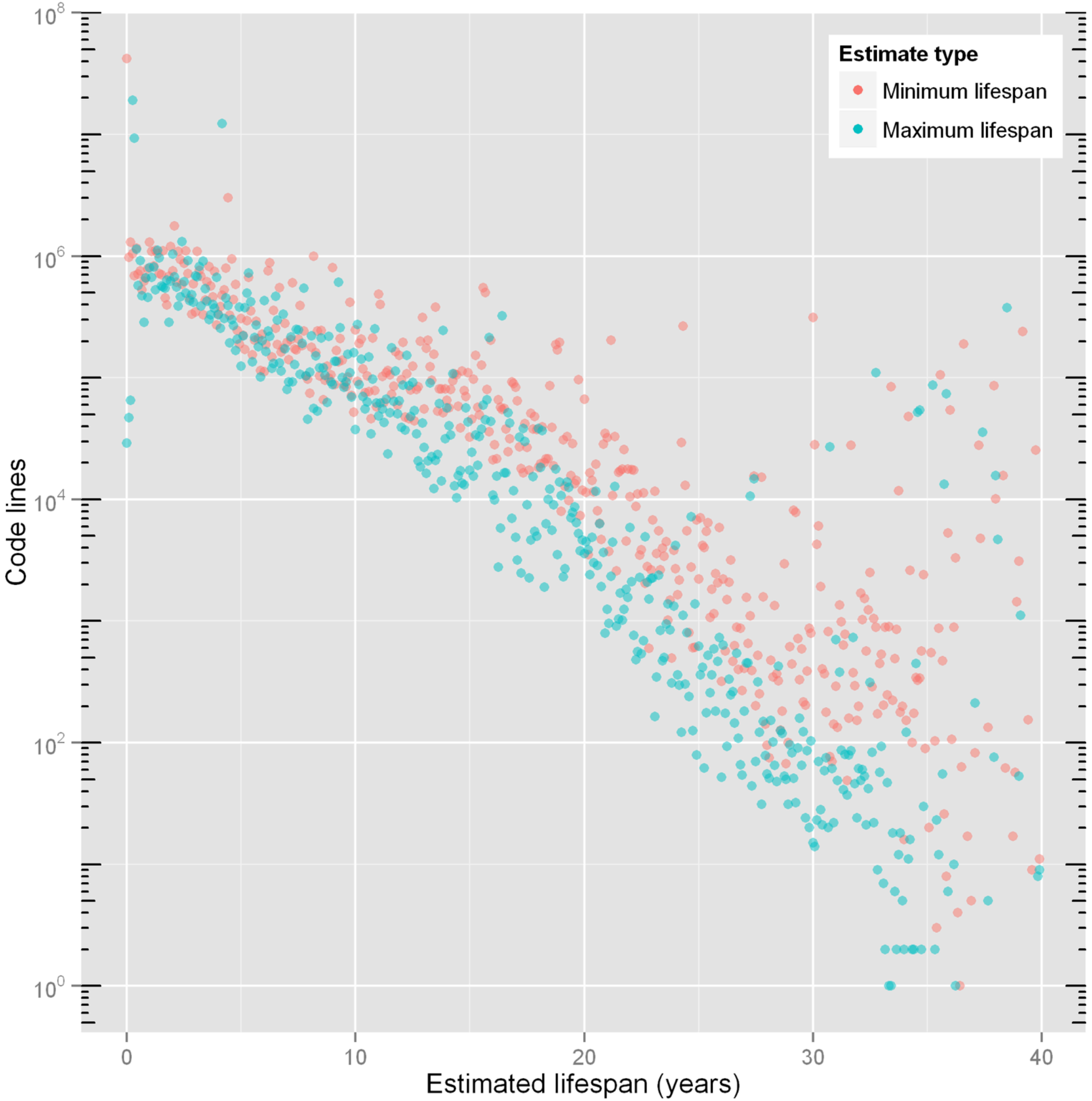
	File 1	File 2	File 3	File 4
Period analyzed	5 years, 2 months	8 years, 3 months	13 years, 3 months	13 years, 4 months
Lines originally	423	192	227	309
Unchanged lines	271	101	4	8
Lines now	664	948	388	414
Grew by	241	756	161	105
Times changed	47	99	194	459
Lines added	396	1,026	913	3,828
Lines deleted	155	270	752	3,723
Lines modified	124	413	1,382	3,556
Total changes	675	1,709	3,047	11,107
Change ratio	1.6x	8.9x	13x	36x

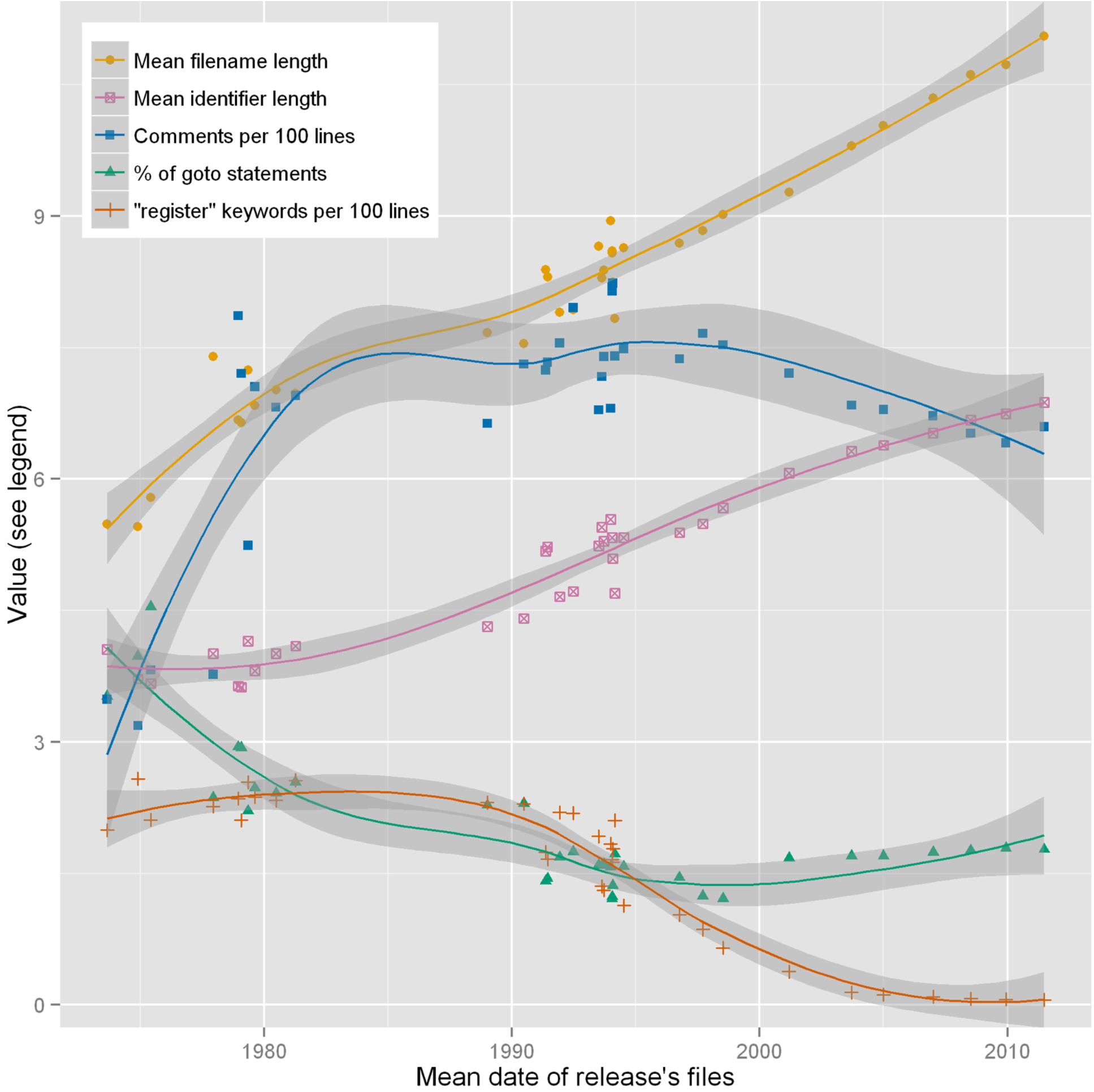
***Följdsats**: det är vettigt att minimera underhållskostnaden på bekostnad av utvecklingskostnaden!*





Based on these observations made in an exploratory study [Spinellis et al, 2015] a follow-up work [Spinellis et al, 2016] used the Unix history repository to examine seven concrete hypotheses. By extracting, aggregating, and synthesizing metrics from 66 snapshots in the period covered by the repository it was found that over the years developers of the Unix operating system appear to have evolved their coding style in tandem with advancements in hardware technology, promoted modularity to tame rising complexity, adopted valuable new language features, allowed compilers to allocate registers on their behalf, and reached broad agreement regarding code formatting. The reported work also showed that many trends point toward increasing code quality through adherence to numerous programming guidelines, that some other trends indicate adoption that has reached maturity, and that in the area of code commenting progress appears to have stalled.

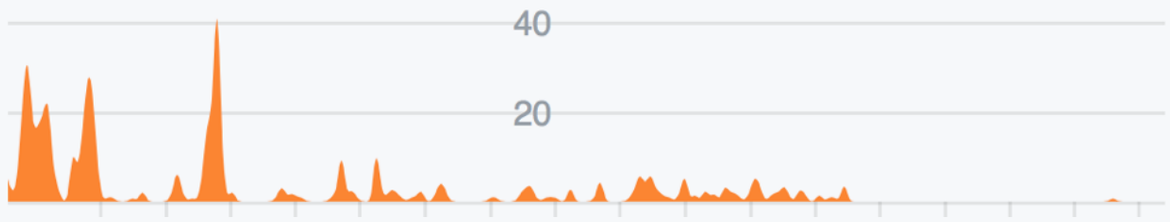



Mean filename length
Mean identifier length
Comments per 100 lines
% of goto statements
"register" keywords per 100 lines
Value (see legend)
9
6
3
0
1980
1990
2000
2010
Mean date of release's files

EliasC

#1

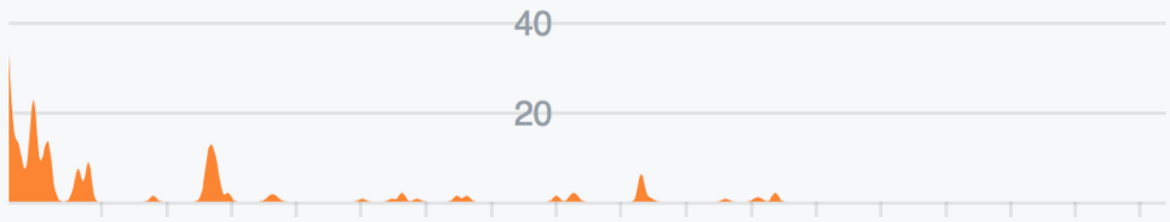
570 commits 55,617 ++ 37,394 --



kaeluka

#2

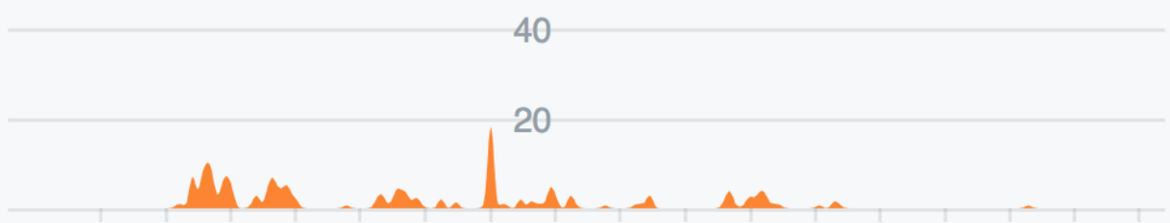
246 commits 31,865 ++ 5,083,137 --



kikofernandez

#3

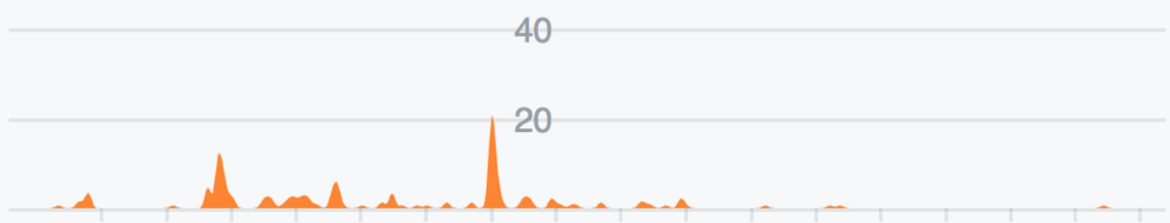
221 commits 19,186 ++ 7,806 --



albertnetymk

#4

169 commits 36,271 ++ 27,119 --



Vad som ofta blir fel när vi vill hantera framtida förändring

- Vi skriver kod som inte behövs!
- Vi skriver kod som är svår att byta ut eller ändra på!
- Vi skriver kod som är på tok för generell!
- Vi skriver kod som förutsätter en massa om framtiden som inte går att veta!
- Vi skriver kod utan någon ordentlig design!
- YAGNI
- DRY
- *Inkrementell utveckling!*

> *Code should be designed based on what you know now, not on what you think will happen in the future. Be only as generic as you know you need to be right now.*

Buggar och defekter

- En medelmåttig programmerare skriver ca 1 bug per 100 rader kod
- En fantastisk programmerare skriver ca 1 bug per 1000 rader kod

  Alltså: sannolikheten för att du smyger in ett fel i ett program är proportionell mot storleken på ändringarna du gör i programmet

- Vi vet att mjukvaran kommer att behöva ändras, men samtidigt så kan vi inte ändra utan att skapa buggar

  Alltså: den bästa designen är den som tillåter maximal förändring i programmets omgivning under minsta möjliga förändring i mjukvaran

- Följdsats:

  ”Lös” aldrig ett problem om du inte har bevis på att det faktiskt är ett problem i praktiken (och $D = \frac{V}{E}$ -analys visar att det är värt att fixas!)

Skriv enklaste möjliga program som löser de problem som skall lösas

- Svårigheten att underhålla mjukvara är proportionell mot svårigheten att underhålla de enskilda delar som programmet är byggt av

  Alltså: bygg av så enkla delar det går

  Alltså: svårigheten skall vara i designen — inte i koden

- Vilka problem du löser är viktigt!

  Ifrågasätt alltid — WHY? Varför vill du att jag skall göra X — vilket problem vill du lösa?

- Låt inte koden driva designen. 80% av en programmerares arbete är att tänka. Bara några få % går åt till att skriva.

- Hitta rätt abstraktionsnivå

- Det finns många olika modeller av samma system – använd fler än en.

Most difficult design problems can be solved by simply drawing or writing them out on paper.

Föreläsning 8

Tobias Wrigstad

Profilering och optimering

Nya bilder 2018