Imperativ och objektorienterad programmeringsmetodik

Föreläsning 4 av många

Tobias Wrigstad

Nu börjar kursen!

PKD

IOOPM

”Programming in the small”

Computational Thinking

”Programming in the large”

Programmeringsmetodologi

Efter godkänd kurs ska studenten kunna [1/2]

- förklara hur program **exekverar** och beskriva hur program **lagrar och hanterar information**.

Efter godkänd kurs ska studenten kunna [1/2]

- förklara hur program **exekverar** och beskriva hur program **lagrar och hanterar information**.
- förklara och exemplifiera **kvalitativa** och **kvantitativa aspekter** av ett programs **design** och **implementation**.

Efter godkänd kurs ska studenten kunna [1/2]

- förklara hur program **exekverar** och beskriva hur program **lagrar och hanterar information**.
- förklara och exemplifiera **kvalitativa** och **kvantitativa aspekter** av ett programs **design** och **implementation**.
- förklara skillnaden mellan **manuell och automatisk minneshantering** samt grundläggande relaterade begrepp (som stack, heap, statisk minnesarea) och använda **verktyg** för felsökning av minnesfel.

Efter godkänd kurs ska studenten kunna [1/2]

- förklara hur program **exekverar** och beskriva hur program **lagrar och hanterar information**.
- förklara och exemplifiera **kvalitativa** och **kvantitativa aspekter** av ett programs **design** och **implementation**.
- förklara skillnaden mellan **manuell och automatisk minneshantering** samt grundläggande relaterade begrepp (som stack, heap, statisk minnesarea) och använda **verktyg** för felsökning av minnesfel.
- förklara hur **större programuppgifter** kan lösas & resonera om olika lösningsalternativ.

Efter godkänd kurs ska studenten kunna [1/2]

- förklara hur program **exekverar** och beskriva hur program **lagrar och hanterar information**.
- förklara och exemplifiera **kvalitativa** och **kvantitativa aspekter** av ett programs **design** och **implementation**.
- förklara skillnaden mellan **manuell och automatisk minneshantering** samt grundläggande relaterade begrepp (som stack, heap, statisk minnesarea) och använda **verktyg** för felsökning av minnesfel.
- förklara hur **större programuppgifter** kan lösas & resonera om olika lösningsalternativ.
- redogöra för hur **enkla parallelliserbara problem kan lösas** effektivt med relevanta hjälpmedel.

Efter godkänd kurs ska studenten kunna [1/2]

- förklara hur program **exekverar** och beskriva hur program **lagrar och hanterar information**.
- förklara och exemplifiera **kvalitativa** och **kvantitativa aspekter** av ett programs **design** och **implementation**.
- förklara skillnaden mellan **manuell och automatisk minneshantering** samt grundläggande relaterade begrepp (som stack, heap, statisk minnesarea) och använda **verktyg** för felsökning av minnesfel.
- förklara hur **större programuppgifter** kan lösas & resonera om olika lösningsalternativ.
- redogöra för hur **enkla parallelliserbara problem kan lösas** effektivt med relevanta hjälpmedel.
- **designa, koda, granska, testa, felsöka och dokumentera** egna program, med hjälp av lämpliga **verktyg**.

Efter godkänd kurs ska studenten kunna [1/2]

- förklara hur program **exekverar** och beskriva hur program **lagrar och hanterar information**.
- förklara och exemplifiera **kvalitativa** och **kvantitativa aspekter** av ett programs **design** och **implementation**.
- förklara skillnaden mellan **manuell och automatisk minneshantering** samt grundläggande relaterade begrepp (som stack, heap, statisk minnesarea) och använda **verktyg** för felsökning av minnesfel.
- förklara hur **större programuppgifter** kan lösas & resonera om olika lösningsalternativ.
- redogöra för hur **enkla parallelliserbara problem kan lösas** effektivt med relevanta hjälpmedel.
- **designa, koda, granska, testa, felsöka och dokumentera** egna program, med hjälp av lämpliga **verktyg**.
- **läsa, förstå och modifiera** icke-trivial **kod** som studenten själv inte har skrivit samt **integrera nyskriven kod med existerande**.

Efter godkänd kurs ska studenten kunna [2/2]

- beskriva – **oberoende av programkoden** – den uppgift ett program skall lösa och de förutsättningar som krävs för att det skall kunna arbeta. Motivera varför programmet under dessa förutsättningar är korrekt, **samt specificera och konstruera testfall och köra tester för att verifiera detta**.

Efter godkänd kurs ska studenten kunna [2/2]

- beskriva – **oberoende av programkoden** – den uppgift ett program skall lösa och de förutsättningar som krävs för att det skall kunna arbeta. Motivera varför programmet under dessa förutsättningar är korrekt, **samt specificera och konstruera testfall och köra tester för att verifiera detta**.
- skriva lämplig **dokumentation för programmering** och **testhantering**.

Efter godkänd kurs ska studenten kunna [2/2]

- beskriva – **oberoende av programkoden** – den uppgift ett program skall lösa och de förutsättningar som krävs för att det skall kunna arbeta. Motivera varför programmet under dessa förutsättningar är korrekt, **samt specificera och konstruera testfall och köra tester för att verifiera detta**.
- skriva lämplig **dokumentation för programmering** och **testhantering**.
- tillämpa specifika **utsnitt av kända utvecklingsprocesser** och -metodiker (ex. **agil utveckling, parprogrammering** och testdriven utveckling).

Efter godkänd kurs ska studenten kunna [2/2]

- beskriva – **oberoende av programkoden** – den uppgift ett program skall lösa och de förutsättningar som krävs för att det skall kunna arbeta. Motivera varför programmet under dessa förutsättningar är korrekt, **samt specificera och konstruera testfall och köra tester för att verifiera detta**.
- skriva lämplig **dokumentation för programmering** och **testhantering**.
- tillämpa specifika **utsnitt av kända utvecklingsprocesser** och -metodiker (ex. **agil utveckling, parprogrammering** och testdriven utveckling).
- beskriva olika former av **testning** och deras vikt i olika skeden av **utvecklings-processen**.

Efter godkänd kurs ska studenten kunna [2/2]

- beskriva – **oberoende av programkoden** – den uppgift ett program skall lösa och de förutsättningar som krävs för att det skall kunna arbeta. Motivera varför programmet under dessa förutsättningar är korrekt, **samt specificera och konstruera testfall och köra tester för att verifiera detta**.
- skriva lämplig **dokumentation för programmering** och **testhantering**.
- tillämpa specifika **utsnitt av kända utvecklingsprocesser** och -metodiker (ex. **agil utveckling, parprogrammering** och testdriven utveckling).
- beskriva olika former av **testning** och deras vikt i olika skeden av **utvecklings-processen**.
- bidra till ett **konstruktivt samarbete i programmeringsprojekt**.

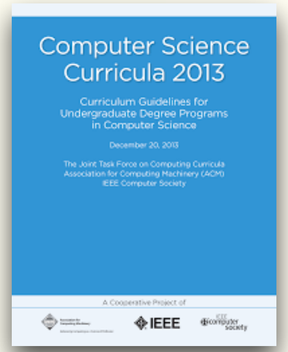
Efter godkänd kurs ska studenten kunna [2/2]

- beskriva – **oberoende av programkoden** – den uppgift ett program skall lösa och de förutsättningar som krävs för att det skall kunna arbeta. Motivera varför programmet under dessa förutsättningar är korrekt, **samt specificera och konstruera testfall och köra tester för att verifiera detta**.
- skriva lämplig **dokumentation för programmering** och **testhantering**.
- tillämpa specifika **utsnitt av kända utvecklingsprocesser** och -metodiker (ex. **agil utveckling, parprogrammering** och testdriven utveckling).
- beskriva olika former av **testning** och deras vikt i olika skeden av **utvecklings-processen**.
- bidra till ett **konstruktivt samarbete i programmeringsprojekt**.
- **presentera** och **diskutera** kursens innehåll **muntligt** och **skriftligt** med för utbildningsnivån lämplig färdighet.

Neccesary for grade-level 3

Neccesary for grade-level 4

Neccesary for grade-level 5



Define and execute simple integration testing

Specify clear interfaces/ boundaries between modules

Breakdown of a program into suitable modules

Reason about the coupling and cohesion of classes/ modules

Explain separation of concerns

Describe the difference between top-down, functional/procedural decomposition and OOAD

Modularisation

Design paradigms

Use divide and conquer, modularisation and information hiding

Design principles

Explain basic software metaphors: client-server, layers, pipes-and-filters

Abstraction

Explain abstraction

Document interfaces

Avoiding excess commenting in source code

Code documentation

Defensive programming and asserts

Introduce software engineering

Explain the software development process concept

Create suitable documentation for programming and test management

Testing

Unit testing

Code review

Integration testing

Describe key terms and ~~standards~~, explain and ~~evaluate~~ processes and ~~strategies~~ for software development, especially testing

Scrum?

Ethics?

Expain how non-trivial programming tasks can be solved and reason about different alternative solutions

Use of access modifiers to control access to data in suitable manners

Use .c and .h file separation to achieve appropriate level of encapsulation

Use of copying etc. to avoid representation exposure

Encapsulation

Explain how to avoid fragile base classes in relation to inheritance

Explain the relation between encapsulation and class invariants

Explain how patterns were used in development of a program

Patterns

Apply refactoring techniques and reason about source code improvements

Use tool support for refactoring

Readability

Consistent use of good naming of variables, functions, etc.

Code formatting, e.g. tabulation, indentation, patterns and structure

Consistent use of a code standard also covering code formatting

Explain and exemplify kvantitative and qualitative aspects of a program's design and implementation

Use inheritance to separate concerns

Use procedural abstraction to avoid repetition and increase readability

Complexity

Memory use

Performance

Demonstrate easily in C-part

Implement dynamic data structures without memory leaks

Demonstrate easily in C-project

Demonstrate understanding between memory management on the stack and on the heap through a C program

Use a profiling tool to improve a program's performance

Systematically detect a program's hotspots using profiling techniques

Explain the difference between manual and automatic memory management including concepts like stack, heap, and use tools to find memory errors

Use valgrind (or equivalent) to debug memory errors

Demonstrate easily in C-project

Read, understand, and modify non-trivial code that the student didn't write herself

Every lab session more or less

Describe -- without the source code -- the task a program is supposed do solve and describe its necessary prerequisites

Motivate why the program is correct under these circumstances, specify, construct, and execute test cases to verify this

Coding exam

Basic C

Translate between recursive and iterative solutions to the same program, especially with respect to side-effects

Explain the difference between declarative and imperative programming

Explain interpretation, compilation and linking

Exception handling

How does garbage collection work?

Demonstrate detailed understanding of central parts of the programming languages used in the course

Imperative

Object-oriented

Basic Java

Coding exam

Explain the difference between static and dynamic binding

Identity vs. equality

Use subtype polymorphism and parametric polymorphism

Explain use of subtype polymorphism and parametric polymorphism for code reuse

Use of overloading and overriding of constructors

Use inheritance, method specialisation, super calls

Make a poster

Make a demo

Make a traditional presentation

Present and discus the course's content orally and in writing with a reasonable skill-level for 2nd year

Produce an essay

Work as a TA

Know how to reuse existing components

Actively participate in project post-mortem

Cooperate with other students in a programming project integrating newly written code with existing code

Project work

OK-complete implementation of specification

Be able to construct a large program from smaller components

Design, code, review, test, debug and document programs using suitable **tools**

Create your own repo

clone, commit, push, pull, merge

Version control

Set up commit notifications

Set up continuous integration

Basic scripting

Avlusning

Use the terminal

Using Emacs for editing

Build management

Simple build scripts

Documentation

Type setting

Extract API documentation with doxygen or Java Doc

Basic shell interaction

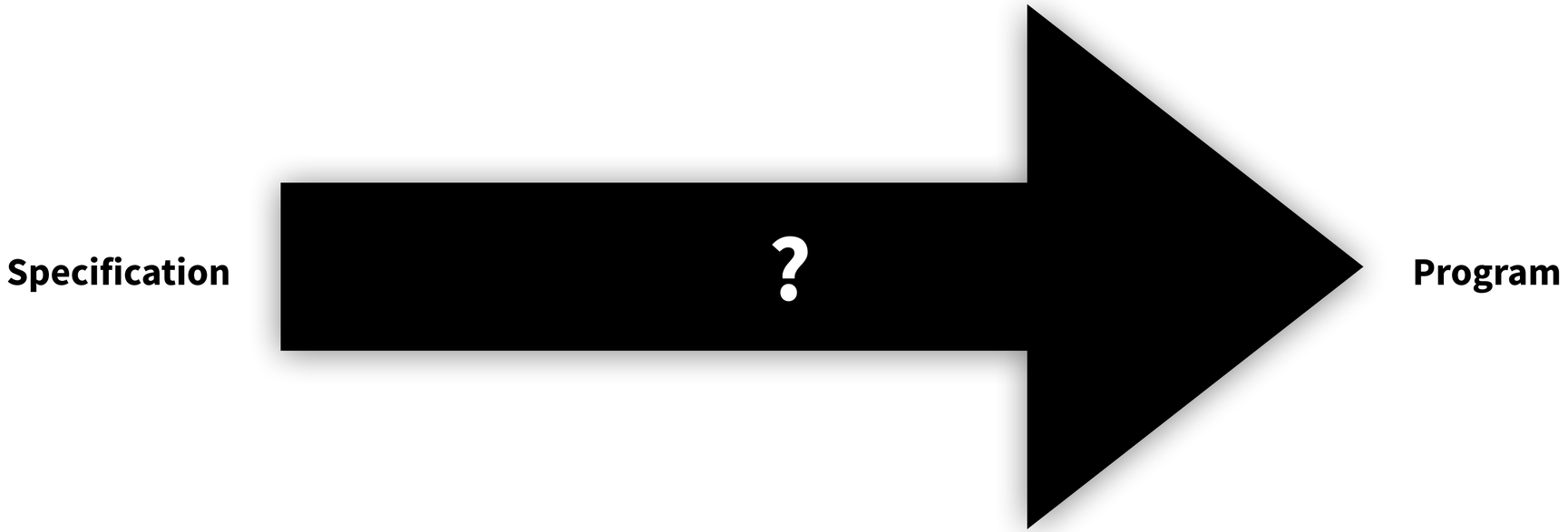
Work efficiently with source code without using the mouse

Compiling and navigating to errors in Emacs

Write LaTeX with proper care about code quality and structure

Configuring Emacs for e.g., code completion, snippets, tags, etc.

Inlämningsuppgifter

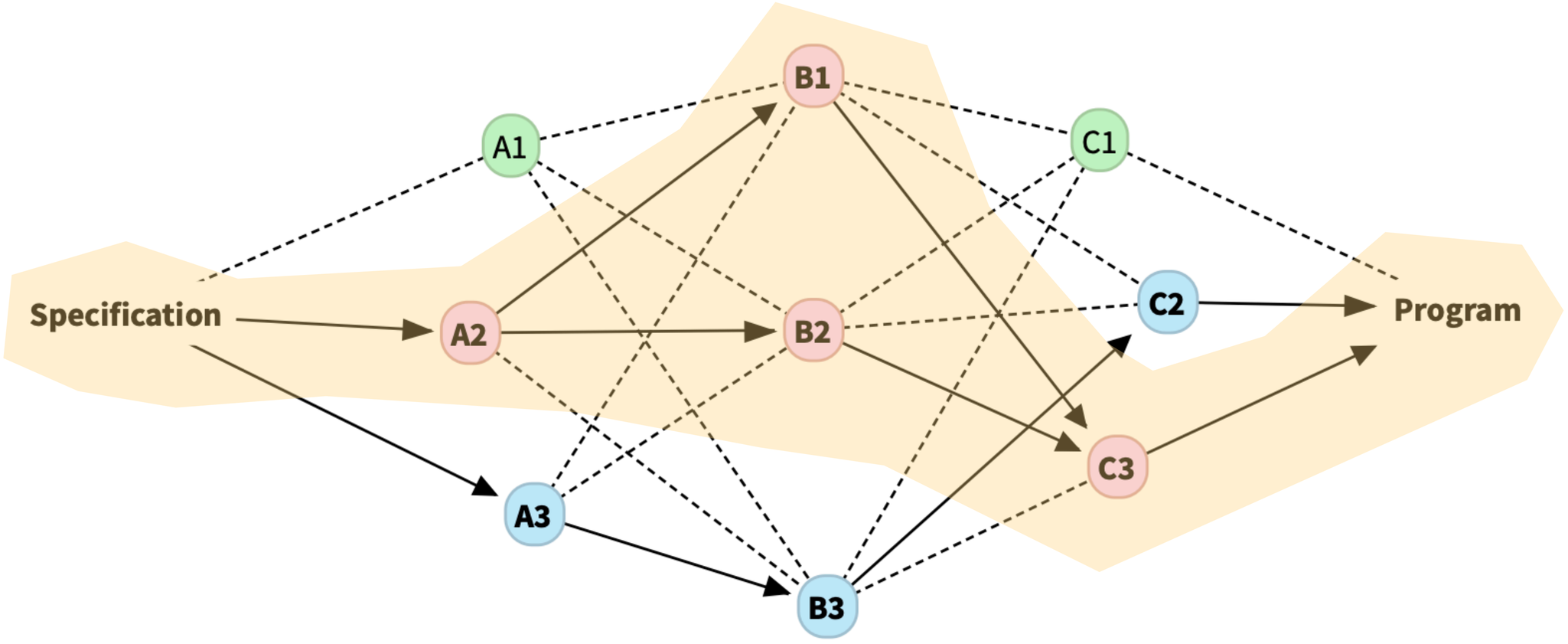


Studentens mål: ett körande program

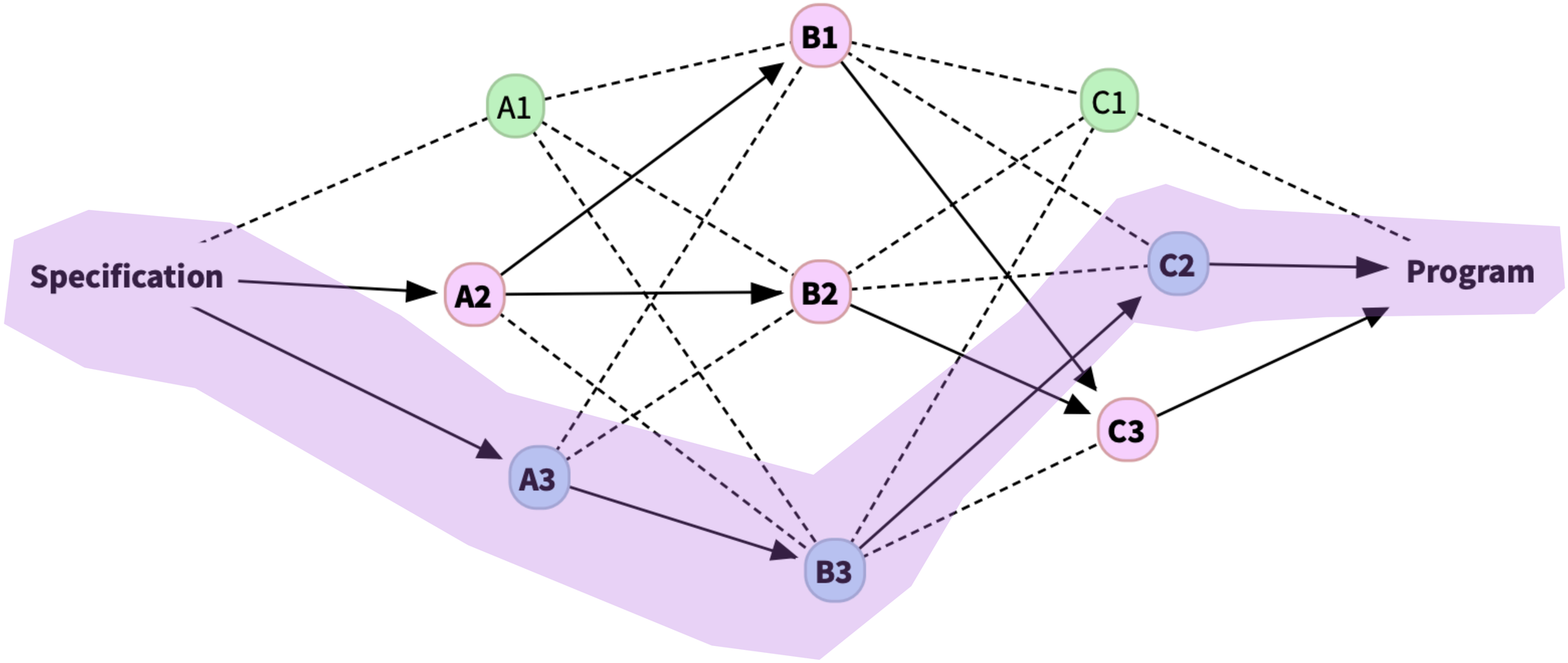
Lärarens mål:

- ✓ Polymorphism
- ✓ Memory management
- ✓ Defensive programming
- ✓ …

Ett program — olika lärdomar



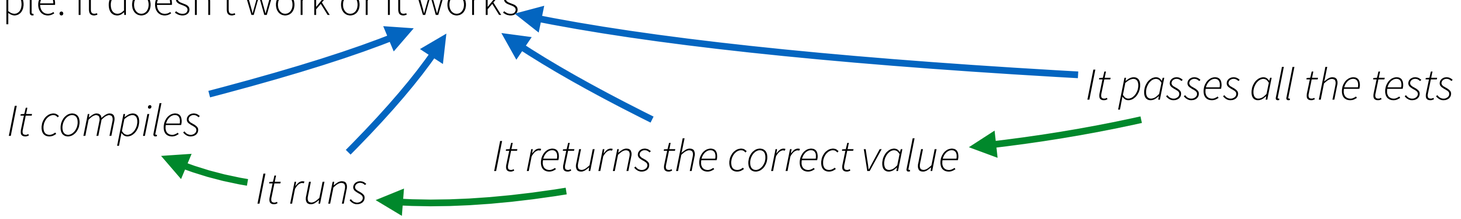
Ett program — olika lärdomar



Field Observations

Students tend to focus on the program rather than abstract concepts

It is simple: it doesn't work or it works



Many methodological concepts are very hard to appreciate solely theoretically

Ex.: maintainability, readability, modularity, …

Not (easily) exercised in short-lived programs

Having a working program does not imply understanding of the intended learning objectives

Ex.: solve problem though dozens of copy-pastes instead of a generic solution

Field Observations

Students' behaviour is greatly influenced by grading and assessment

Ex.: unless something is clearly rewarded in grading, it usually does not permeate

Many students ramp up their activity prior to exams

Not all students are equally apt for everything

Serialists, holists, prior exposure

Thus, we cannot expect the same methods to work for all students — or we should not expect all students to become great (or even good)

Favours declarative over imperative instruction

Mastery Learning [Bloom, 1968]

Strategy

Cannot move in curriculum forward until ”mastery” is achieved

Repeated assessment for mastery

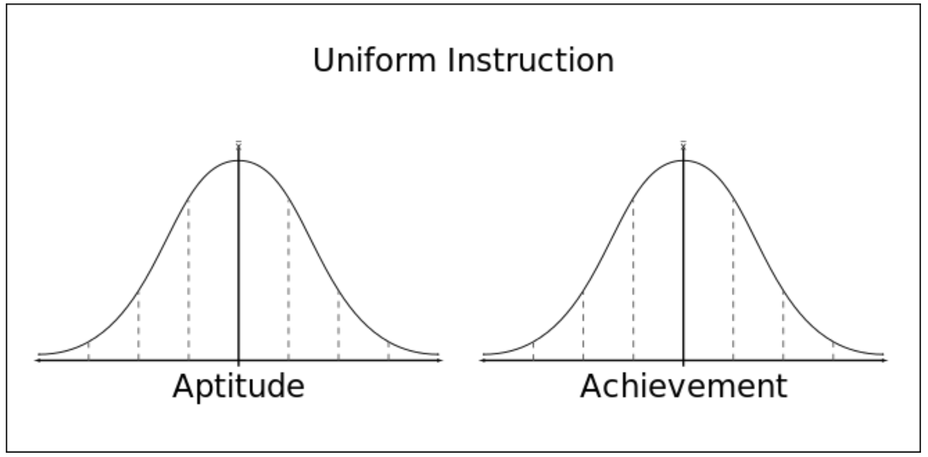
Philosophy

Given sufficient resources all students will eventually achieve mastery

Cater to different needs of different students

Lots of different implementations and variations exist

Folklore belief seems to be that it is hard to get right



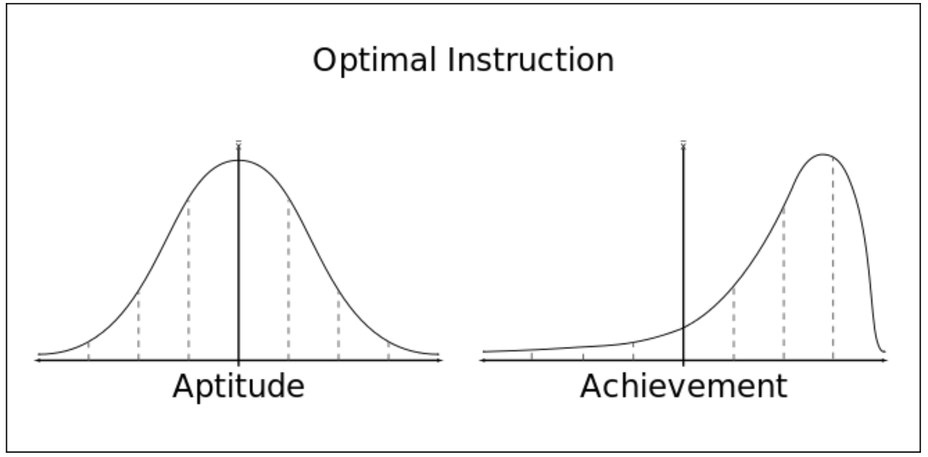


Image source: wikipedia

Design Goals

Incorporate ideas from Mastery Learning

Flexibility of instruction

Continuous assessment of mastery of subjects

Transfer responsibility for learning from teachers to students

Stimulate reflection

Make the act of learning explicit — teachers are facilitators, you can’t just ”follow along”

Improved alignment between ILOs, tasks, and grades

Grading should not be a black box

Increase student’s active engagement with all ILO’s and lift all ILOs to the mastery-check level

New course design in a nutshell

Departure from assignment-driven model

- Assignments are now ”a necessary evil”

ILOs broken down into a large number of achievements

- Achievements are distributed over the grade levels
- Mastery of all achievements in a grade-level is required for that grade
- Need to complete a set of bootstrap exercises to start working on assignments and demonstrating

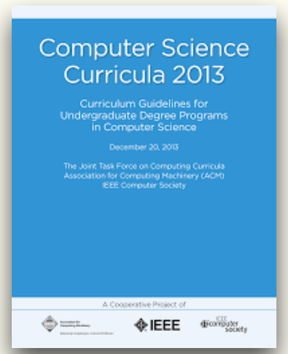
Mastery can be demonstrated at students’ leisure (during labs)

- No predefined (or suggested) order
- Failed demonstration does not show on grade, and unlimited retries are allowed (technically)
- Mastery demonstrations start from accomplishments in the assignments

Neccesary for grade-level 3

Neccesary for grade-level 4

Neccesary for grade-level 5



Define and execute simple integration testing

Specify clear interfaces/ boundaries between modules

Breakdown of a program into suitable modules

Reason about the coupling and cohesion of classes/ modules

Explain separation of concerns

Describe the difference between top-down, functional/procedural decomposition and OOAD

Modularisation

Design paradigms

Use divide and conquer, modularisation and information hiding

Design principles

Explain basic software metaphors: client-server, layers, pipes-and-filters

Abstraction

Explain abstraction

Document interfaces

Avoiding excess commenting in source code

Code documentation

Defensive programming and asserts

Introduce software engineering

Explain the software development process concept

Create suitable documentation for programming and test management

Testing

Unit testing

Code review

Integration testing

Describe key terms and ~~standards~~, explain and ~~evaluate~~ processes and ~~strategies~~ for software development, especially testing

Scrum?

Ethics?

Expain how non-trivial programming tasks can be solved and reason about different alternative solutions

Use of access modifiers to control access to data in suitable manners

Use .c and .h file separation to achieve appropriate level of encapsulation

Read, understand, and modify non-trivial code that the student didn't write herself

Use of copying etc. to avoid representation exposure

Encapsulation

Explain how to avoid fragile base classes in relation to inheritance

Explain the relation between encapsulation and class invariants

Patterns

Explain how patterns were used in development of a program

Readability

Apply refactoring techniques and reason about source code improvements

Use tool support for refactoring

Consistent use of good naming of variables, functions, etc.

Code formatting, e.g. tabulation, indentation, patterns and structure

Consistent use of a code standard also covering code formatting

Explain and exemplify kvantitative and qualitative aspects of a program's design and implementation

Use inheritance to separate concerns

Use procedural abstraction to avoid repetition and increase readability

Complexity

Memory use

Performance

Demonstrate easily in C-part

Implement dynamic data structures without memory leaks

Demonstrate easily in C-project

Demonstrate understanding between memory management on the stack and on the heap through a C program

Use a profiling tool to improve a program's performance

Systematically detect a program's hotspots using profiling techniques

Explain the difference between manual and automatic memory management including concepts like stack, heap, and use tools to find memory errors

Use valgrind (or equivalent) to debug memory errors

Demonstrate easily in C-project

Every lab session more or less

Describe -- without the source code -- the task a program is supposed do solve and describe its necessary prerequisites

Motivate why the program is correct under these circumstances, specify, construct, and execute test cases to verify this

Coding exam

Translate between recursive and iterative solutions to the same program, especially with respect to side-effects

Basic C

Explain the difference between declarative and imperative programming

Explain interpretation, compilation and linking

Exception handling

How does garbage collection work?

Demonstrate detailed understanding of central parts of the programming languages used in the course

Basic Java

Coding exam

Imperative

Object-oriented

Explain the difference between static and dynamic binding

Identity vs. equality

Use subtype polymorphism and parametric polymorphism

Explain use of subtype polymorphism and parametric polymorphism for code reuse

Use of overloading and overriding of constructors

Use inheritance, method specialisation, super calls

Make a poster

Make a demo

Make a traditional presentation

Present and discus the course's content orally and in writing with a reasonable skill-level for 2nd year

Produce an essay

Work as a TA

Know how to reuse existing components

Actively participate in project post-mortem

Cooperate with other students in a programming project integrating newly written code with existing code

Project work

OK-complete implementation of specification

Be able to construct a large program from smaller components

Design, code, review, test, debug and document programs using suitable **tools**

Create your own repo

clone, commit, push, pull, merge

Version control

Set up commit notifications

Set up continuous integration

Basic scripting

Avlusning

Use the terminal

Using Emacs for editing

Build management

Simple build scripts

Documentation

Type setting

Extract API documentation with doxygen or Java Doc

Basic shell interaction

Work efficiently with source code without using the mouse

Compiling and navigating to errors in Emacs

Write LaTeX with proper care about code quality and structure

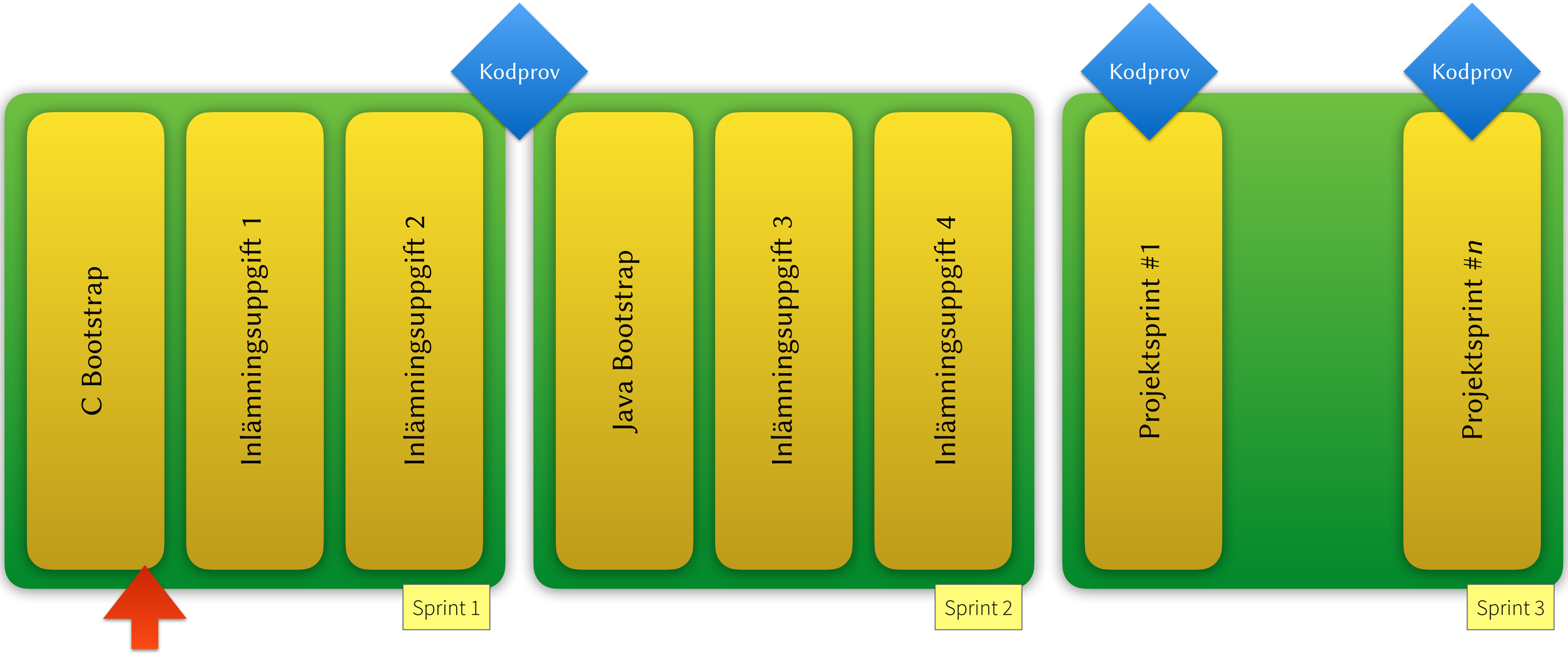
Configuring Emacs for e.g., code completion, snippets, tags, etc.

List of all achievements (~70)

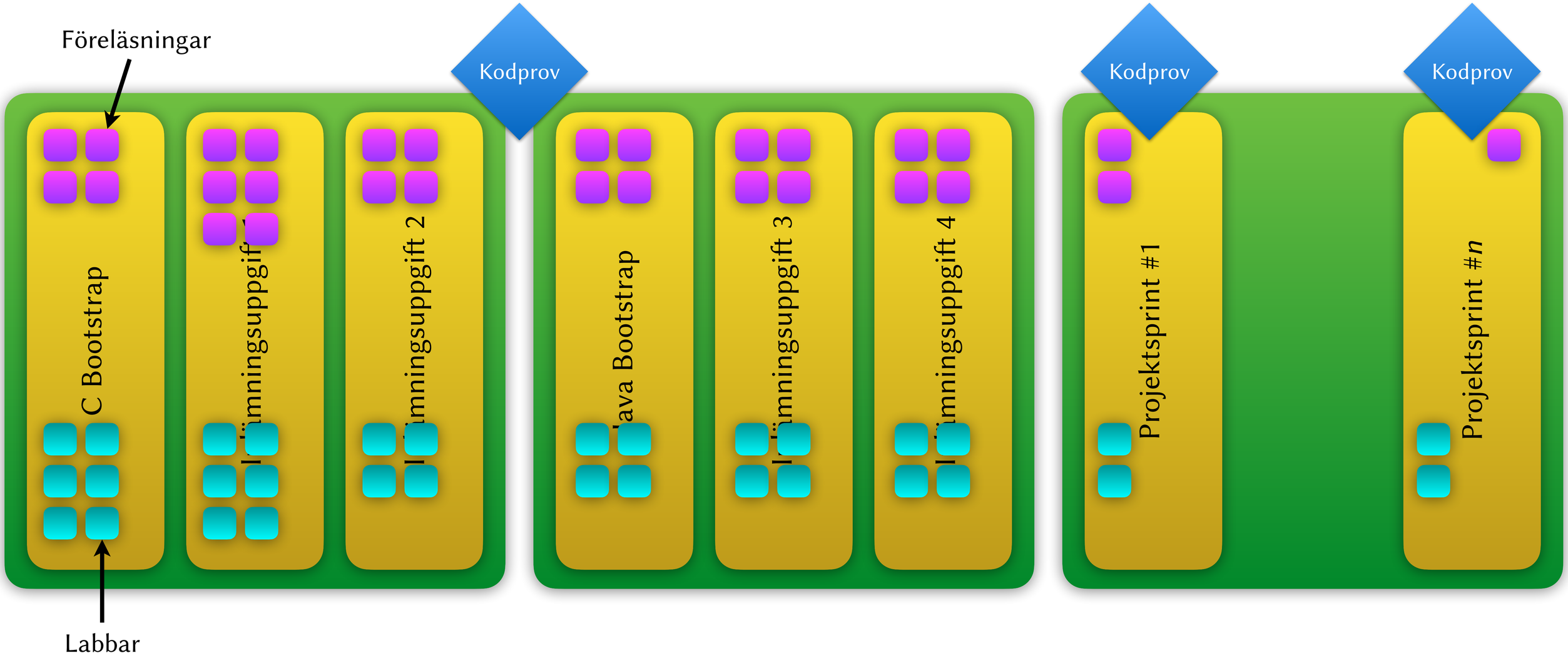
- **Abstraction** — Achievements in this group concern procedural and object-oriented abstraction, the importance of naming, and information hiding.
- **Code Review** — Informal code reviews, responding to code reviews, refactoring.
- **Communication** — Essay writing, presenting, working as a teaching assistant.
- **Documentation**
- **Encapsulation** — Aliasing, name-based encapsulation, nested and inner classes, interplay of strong encapsulation and testing.
- **Generics** — Dealing with generics both in Java and C, and designing with/for parametric polymorphism.
- **Imperative Programming** — Recursive vs. iterative solutions, tail recursion elimination.
- **Inheritance** — Object-oriented inheritance, overriding, overloading, subtype polymorphism, separation of concerns.
- **Memory Management** — Allocation on stack vs. heap, manual memory management, automatic memory management, manual vs. automatic memory management.
- **Methodology** — Defensive programming, exception handling, failure management and fault tolerance.
- **Modularisation** — Module boundaries and interfaces, coupling and cohesion, separation of concerns.
- **Object, identity & structure** — Identity vs. equality, value semantics, concrete vs. abstract classes.
- **Planning and following-up**
- **Pointers** — Pointers and arrays in C, pointer-based linked structures, pointer semantics, indirection and pointers to pointers.
- **Pragmatics** — Compilers, interpreters, JIT'ing, linking, and binding.
- **Profiling & Optimisation** — Profiling for performance and memory usage, optimising performance and memory usage guided by profiling results.
- **Testing** — Unit testing, test quality, static analysis, fuzzing.
- **Tools** — Debuggers, documentation tools, VCS, build tools, working the terminal.
- **Assignments**
- **Project**
- **Bootstrap labs**

Hur?

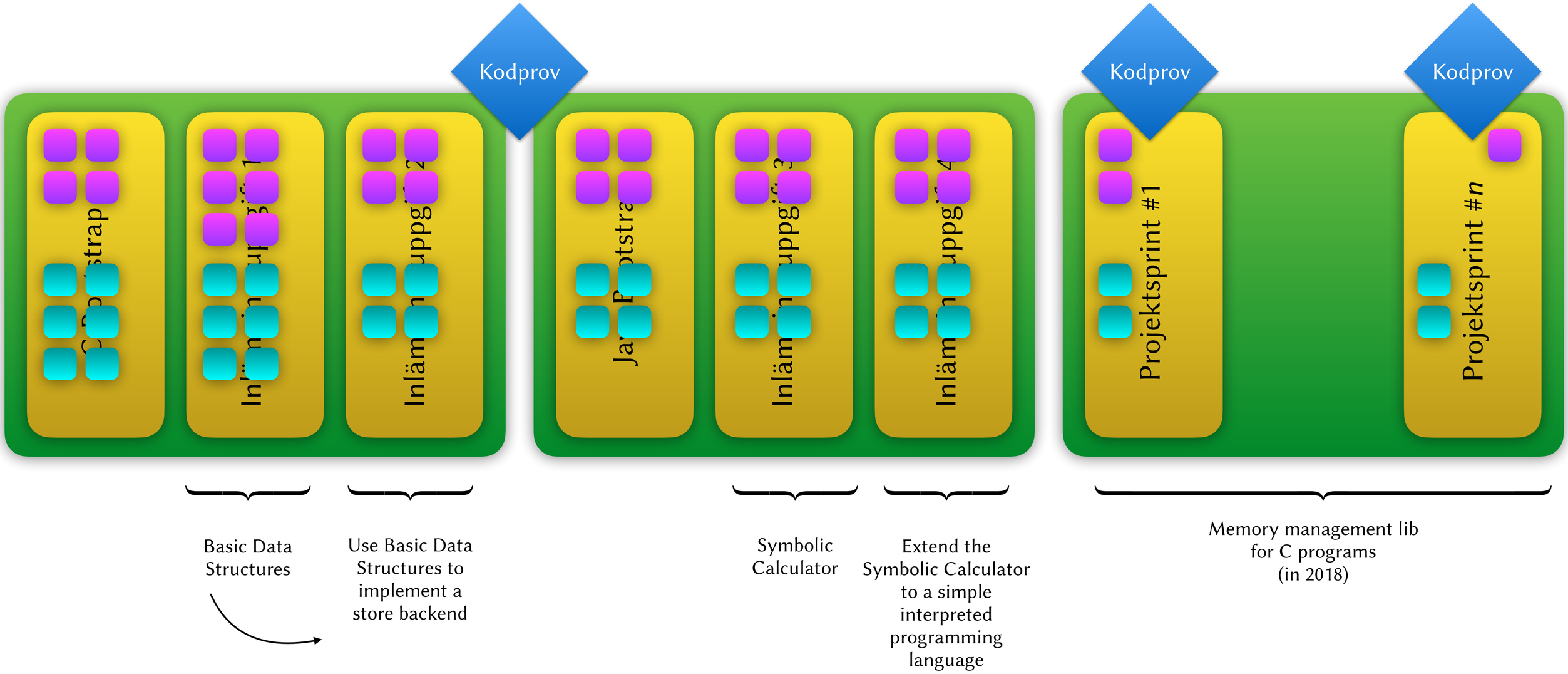
Hur fungerar IOOPM?



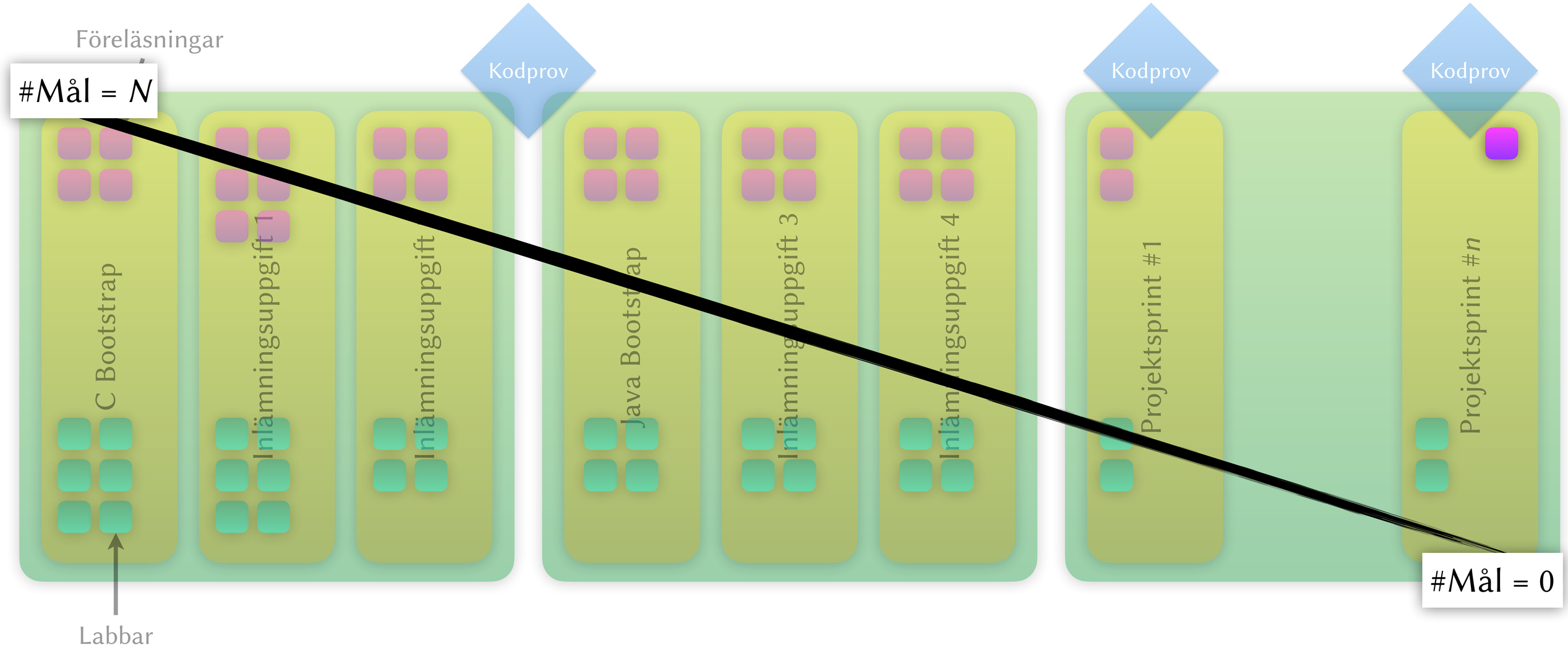
Hur fungerar IOOPM?



Hur fungerar IOOPM?



Hur fungerar IOOPM?



N?

15 O: Profilering och Optimering

15.1 TODO O42: Profilering och optimering 1/3

Lab G3

> Använda profileringsverktyg för att visa var ett redan skrivet program tillbringar mest tid.

15.2 O43: Profilering och optimering 2/3

Lab G4

> Med ledning resultatet från föregående mål, optimera programmet på ett förtjänstfullt sätt.

15.3 O44: Profilering och optimering 3/3

Lab G5

> Samma som mål de två ovanstående målen, men beakta också minnesanvändning.

Djupare förståelse för optimering

Beskrivningar av ”målen”

My Achievements

Show unlocked Show pushbacks

Achievement	Short Description	Grade Level
A1	Procedurell abstraktion	3
A2	Objektorienterad abstraktion	3
A3	Informationsgömning	4
A8	Gränssnitt	4
B4	Arv och subtypspolymorfism	3
B5	Liskov's Substitution Principle	3
B6	Genomskärande åtaganden och arv	4

ID Ingress/namn

Lab G3

2.1 A1: Procedurell abstraktion

> Tillämpa procedurell abstraktion på ett konsekvent sätt för att öka läsbarheten u
> upprepningar.

Abstraktion är en av de viktigaste programmeringsprinciperna. Vi vet att djupt, djupt nere under huven är allt bara ettor och nollor (redan detta är en abstraktion!), men ovanpå dessa har vi byggt lager på lager av abstraktioner som låter oss tala om program t.ex. i termer av struktar och procedurer.

Procedurell abstraktion handlar om att separera användande från implementation. Varje procedur utför – idealiskt – en enda, välspecificerad funktion, t.ex. en viss beräkning, en förändring av programmets tillstånd, uppdaterar en datastruktur, stänger en fil, etc.

En väldöpt procedur (funktion) t.ex. `open_file_for_reading()` ger en tydlig beskrivning av *vad* den gör, men *hur* den gör det – dess implementation – är inkapslad och inte synlig utifrån. Använder den systemanropet `fopen()` eller någon annan funktion? **Det skall vi strunta i!** Det är inte[1] inte något vi skall behöva bry oss om, och dessutom något som kan komma att ändras, eller skilja sig beroende på vilken dator jag kör programmet på.

Ta som tumregel att varje procedur skall kunna beskrivas enkelt och kortfattat. En procedur vars beteende inte går att beskrivas enkelt och kortfattat skall förmodligen brytas upp i flera mindre procedurer.

I ett nötskal handlar procedurell abstraktion alltså om att *kapsla in* alla “conceptual units of behaviour” i en *procedur*. En procedur är ungefär detsamma som en *funktion* och många programspråk (t.ex. C) gör ingen skilland på dem.

2.1.1 Exempel

Proceduren `ritaEnCirkel(int radie, koordinat center)` utför beräkningar och tänder individuella pixlar på en skärm, men i och med att dessa rutiner kapslats in i en procedur med ett vettigt namn, där indata är uttryckt i termer av koordinater och radie har vi abstraherat bort dessa detaljer, och det blir möjligt att rita cirklar tills korna kommer hem utan att förstå hur själva implementationen ser ut.

Väl utförd abstraktion döljer detaljer och låter oss fokusera på färre koncept i taget.

2.1.2 Redovisning

Du bör ha en klar uppfattning om bland annat:

- Varför det är vettigt att identifiera liknande mönster i koden och extrahera dem och kapsla in dem i en enda procedur som kan anropas istället för upprepningarna?
- Abstraktioner kan “läcka”. Vad betyder det och vad får det för konsekvenser?
- Vad är skillnaderna mellan “control abstraction” (ex. if-satsen är en abstraktion) och “data abstraction” (ex. en lista är en abstraktion)? Du kan läsa om dessa koncept på t.ex. http://en.wikipedia.org/wiki/Abstraction_(computer_science).
- Ge exempel på procedurell abstraktion i ditt program! Kritisera den! Kan den förbättras?
- Vad är den kortfattade beskrivningen av vad funktionerna gör?
- Ger namnen på funktionerna en bra ledning om vad funktionerna gör?
- Finns det exempel på **läckande abstraktioner**, dvs. där den som anropar funktionen måste känna till hur funktionen faktiskt är implementerad för att fungera, eller förutsätter en viss implementation?
- Låt f_1, f_2 och f_3 vara funktioner. f_1 och f_2 är delar av samma bibliotek och f_2 använder f_3 i sin implementation. Skiljer sig nivån av abstraktion mellan dessa på något sätt? Hur?
- Är abstraktion möjlig utan inkapsling? (Bonusfråga)

for a client to know how they are implemented. This preserves maximal freedom for us to change the implementation as we continue to improve our code, and also helps us protect the invariants of the data structure.

Structs such as `struct hash_table` and `struct entry` are internal implementation details and as such should be in the `hash_table.c` file, not the `.h` file. Structs like `option` (if you implemented it) *needs* to go into the `.h` file, as a user must be able to look inside the struct to search values.

Placement inside the `.c` file has impact on the testing instead – how will tests be able to access the structs?

Place the structs in the right places, and know your motivation for what you chose.

Tip

This is a good time to start thinking about Achievement A3.

2.4.2 Using Debuggers

Tip

Once you are done with this bit, you know enough `gdb` to debug your own problems in the subsequent steps. If and when you have a tricky situation to debug, **after you solved it**, you should definitely use that to demonstrate Achievement R52.

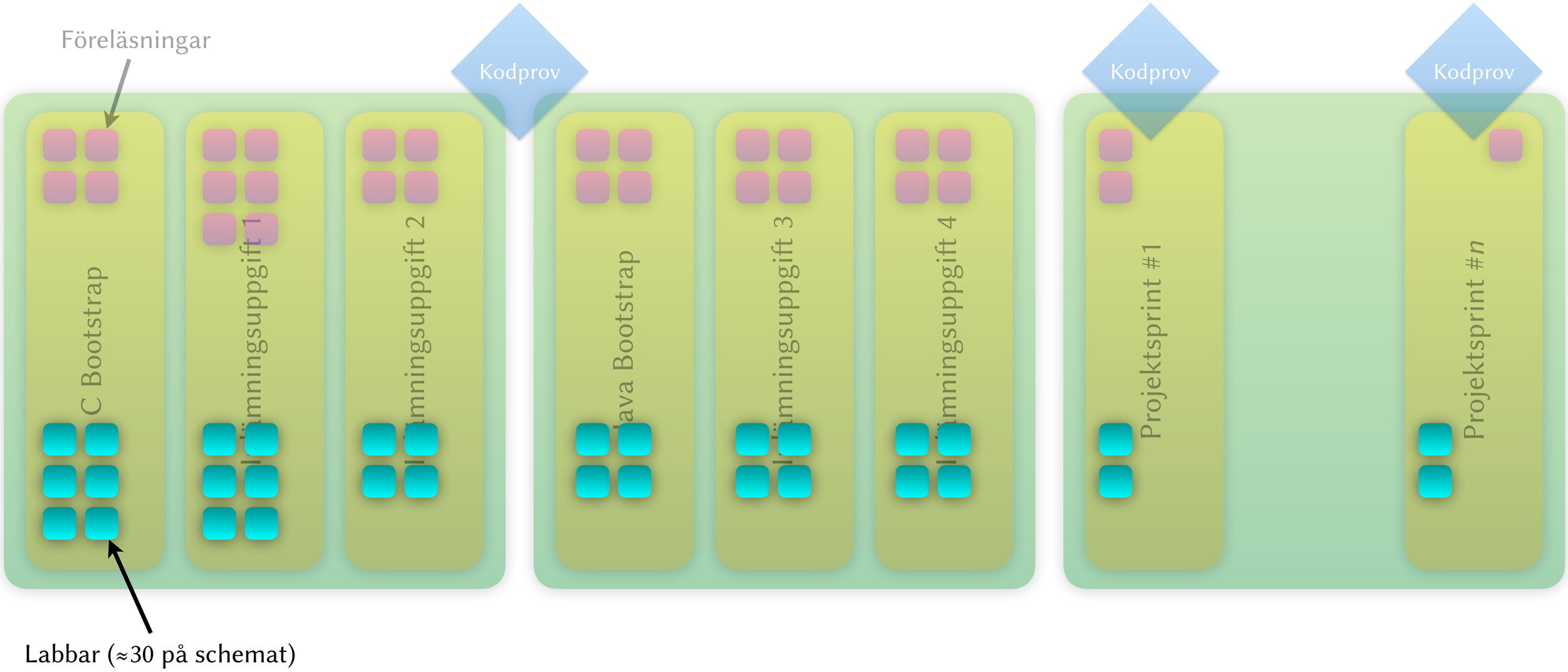
Debuggers like `gdb` and `lldb` are great tools for understanding programs and fixing bugs. You need to familiarise yourself with a debugger (say, `gdb`) **as soon as possible** so that you are never afraid to use it later. You should learn the basics *before* hitting a bug, so you don't need to struggle both with learning how to navigate `gdb` and using it to find the bug at the same time.

So, in that spirit, let us spend a few minutes in `gdb` to explore the hash table representation of above. Below is a program that (on the stack for simplicity) creates a hash table. Compile this program like so: `gcc -pedantic -Wall -g gdb-test.c` (read about what the flags mean here). This produces a file `a.out` that you can run through `gdb`. Note that compiling this program will produce a warning that you're declaring a variable `ht` that you are not using. We can live with this warning for now – we are going to use `ht` through the debugger.

Att mappa inlämningsuppgifter mot mål

Inlupp 1

Hur fungerar IOOPM?



Redovisning

- Din uppgift: att visa examinatorn att du uppfyller målet

  Ge exempel från den kod (etc.) du har skrivit

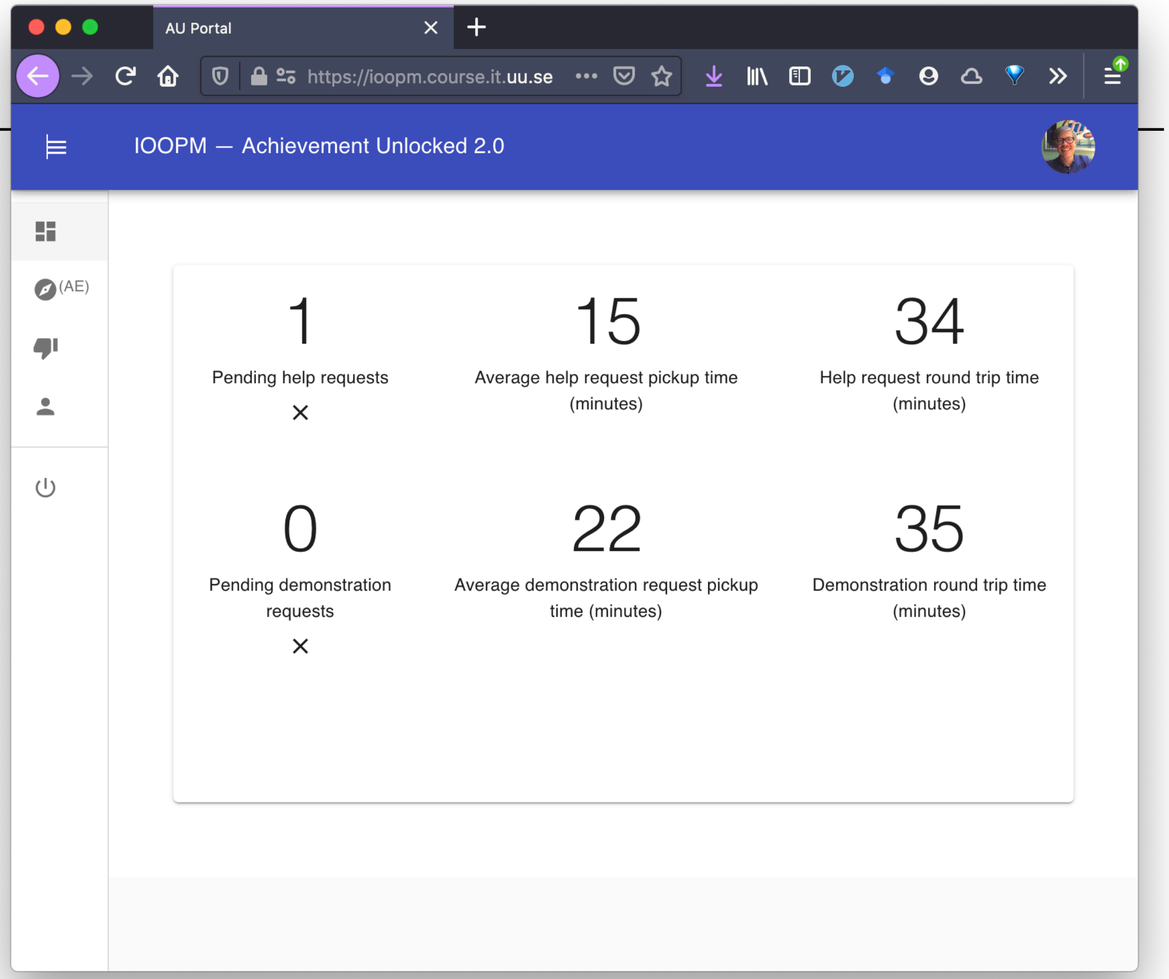
  Inga core dumps

  Ha en story för hur allting hänger ihop

- Försök att alltid redovisa > 1 mål åt gången

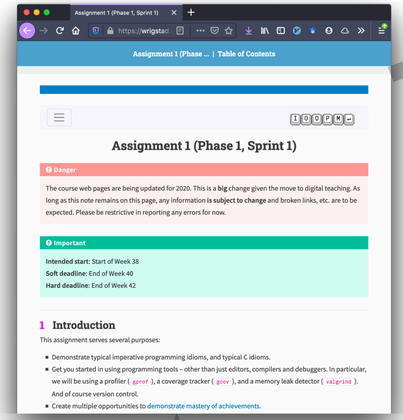
  Mindre arbete, mindre väntetid

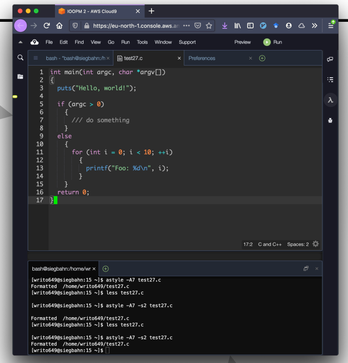
  Synergi och syntes!

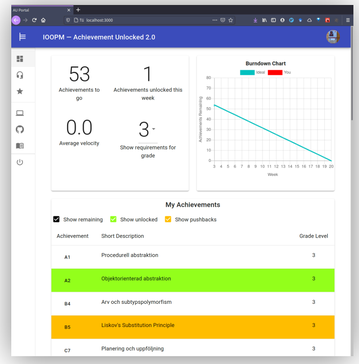


En vecka eller sprint på IOOPM

Implementera



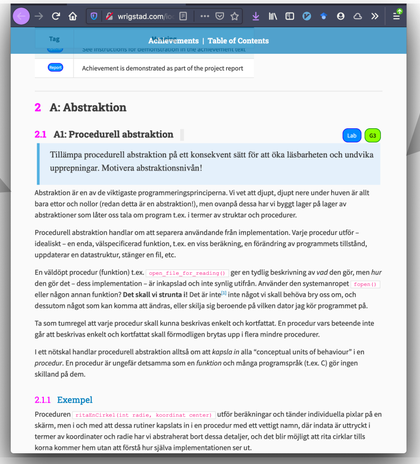




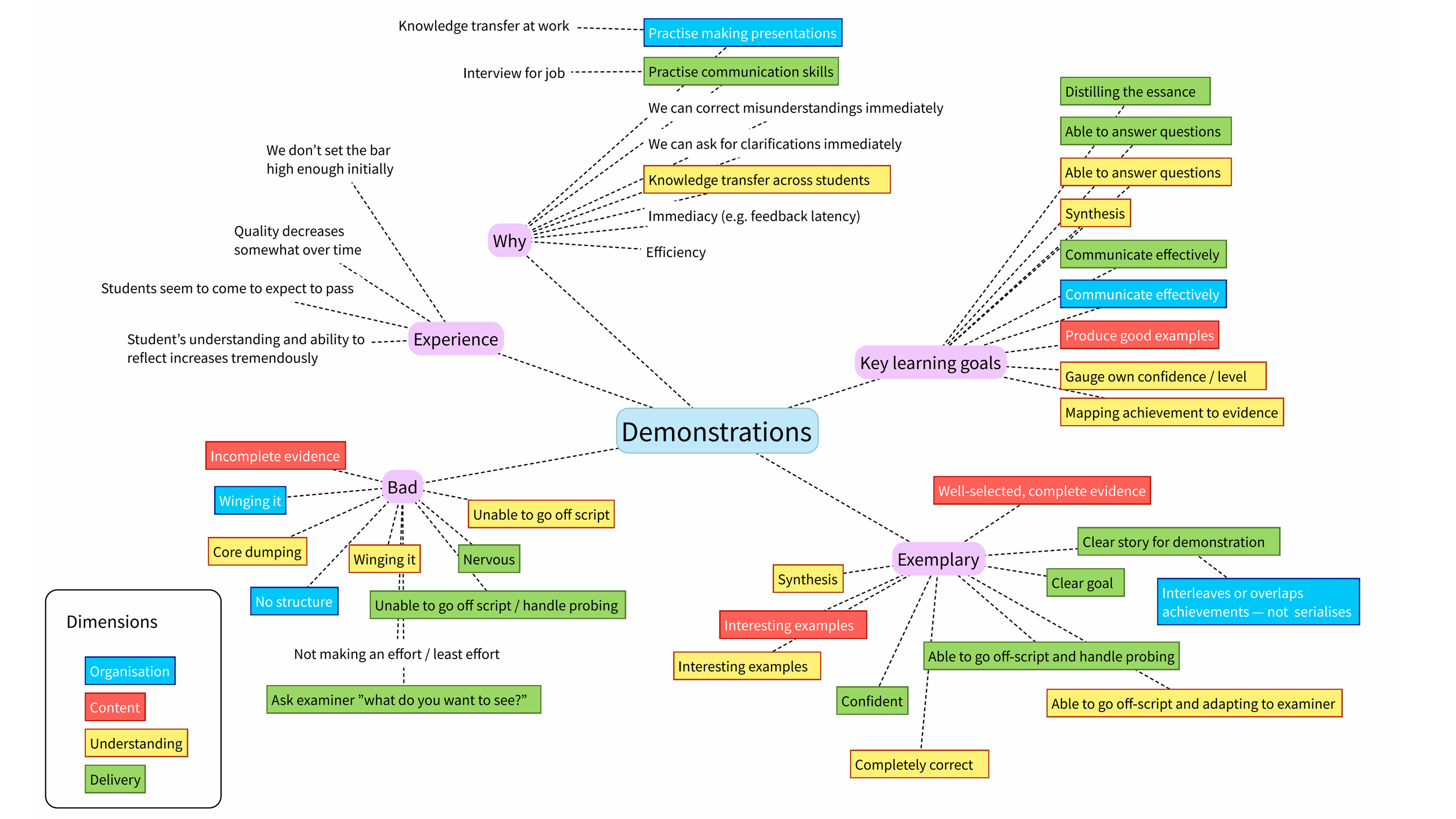
För bok över dina framsteg

Uppgift

Välj mål



Redovisa


Demonstrations
Why
Knowledge transfer at work
Practise making presentations
Interview for job
Practise communication skills
We can correct misunderstandings immediately
We can ask for clarifications immediately
Knowledge transfer across students
Immediacy (e.g. feedback latency)
Efficiency
Experience
We don’t set the bar high enough initially
Quality decreases somewhat over time
Students seem to come to expect to pass
Student’s understanding and ability to reflect increases tremendously
Key learning goals
Distilling the essance
Able to answer questions
Able to answer questions
Synthesis
Communicate effectively
Communicate effectively
Produce good examples
Gauge own confidence / level
Mapping achievement to evidence
Bad
Incomplete evidence
Winging it
Unable to go off script
Core dumping
Winging it
Nervous
No structure
Unable to go off script / handle probing
Not making an effort / least effort
Ask examiner ”what do you want to see?”
Exemplary
Well-selected, complete evidence
Clear story for demonstration
Clear goal
Interleaves or overlaps achievements — not serialises
Synthesis
Interesting examples
Interesting examples
Able to go off-script and handle probing
Confident
Able to go off-script and adapting to examiner
Completely correct
Dimensions
Organisation
Content
Understanding
Delivery

Unlimitied retries

Det viktigaste är inte när du förstår något — utan att

Vi fäster ingen vikt vid om du förstår något tidigt eller sent på kursen, eller omedelbart eller efter sju försök

Ett redovisningstillfälle är en dialog

Idealiskt ett tillfälle till lärdom oavsett om man blir godkänd eller inte

Omedelbar återkoppling är effektivare än skriftlig återkoppling

Våra resurser är dock inte ändliga

T.ex. ett begränsat antal hjälplärare och ett begränsat antal redovisningstillfällen

Vi kan t.ex. inte ta emot alla redovisningar sista veckan

All programutveckling — och redovisning — sker i par

- Främjar djupinlärning
- Det finns ingen gräns för vad två personer kan åstadkomma tillsammans
- Samarbeta med hjälp av till buds stående verktyg

  GitHub

  Cloud 9
- All programmering behöver inte göras i par framför en dator men alla måste kunna all kod

  Kodgranskning är t.ex. ett utmärkt komplement där
- Vi har delat in er i grupper och paren väljer ni själva från grupperna

  Ni kommer inte alltid att vara i fas och ni kommer ibland att vara med i två par samtidigt — det är OK

Gruppindelning

- Samtalsgrupp och hjälpgrupp

- Sex möten under kursen

  Diskussion kring och utifrån burndown chart

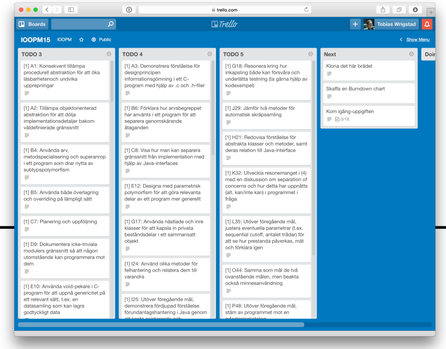
  Tillsammans med andra studenter

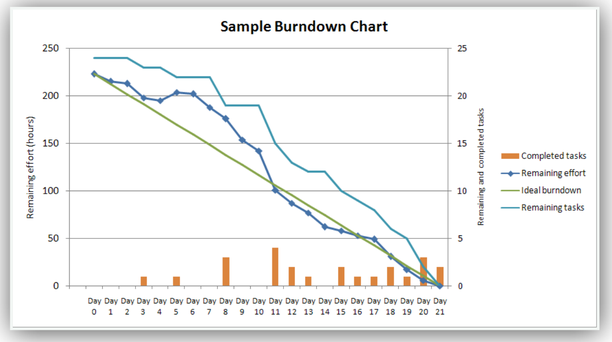
  I början av kursen är det fokus på planering

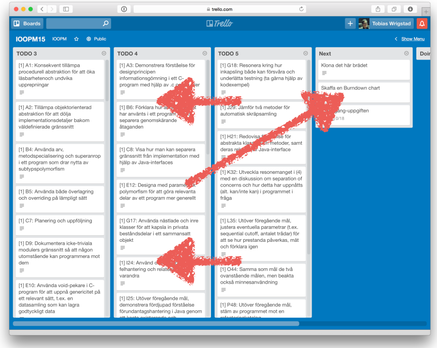
  Vi går mot fokus på uppföljning

  Hjälpa oss att identifiera problem

- Coachen kallar / har kallat till första möte ”nu”







Högskolepoäng

	HP	Deadline	
Fas 1	5	V44*	
Fas 2	5	V49	
Fas 3	5	V2	
Kodprov	2,5	*	Oktober
	2,5	*	December

**It's complicated (se kurswebben för detaljer)*

Betyg (ungefärligen)

	3	4	5
Mål (ca)	≈32	≈18	≈7
Inluppar	4 (samma för alla)		
Projekt	7 (samma för alla)		
Labbar	7 (samma för alla)		
Varav utanför labb	2	2	1

”Specialmål”: övning i skriftlig färdighet

- På nivå 4 och 5 måste du redovisa ett mål som en essä

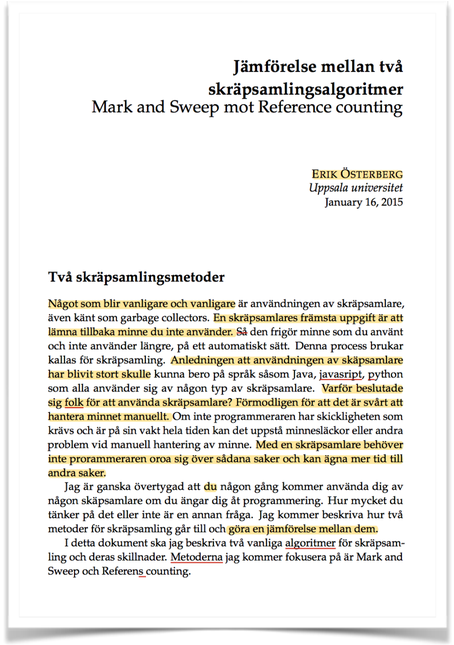
  (För att nå nivå 3 räcker det med projektrapporten)

- Instruktioner finns på kurswebben

  Omfattning: 7500 tecken

  Deadline: **se kurswebben**

- Lämnas in via GitHub



Jämförelse mellan två skräpsamlingsalgoritmer
Mark and Sweep mot Reference counting

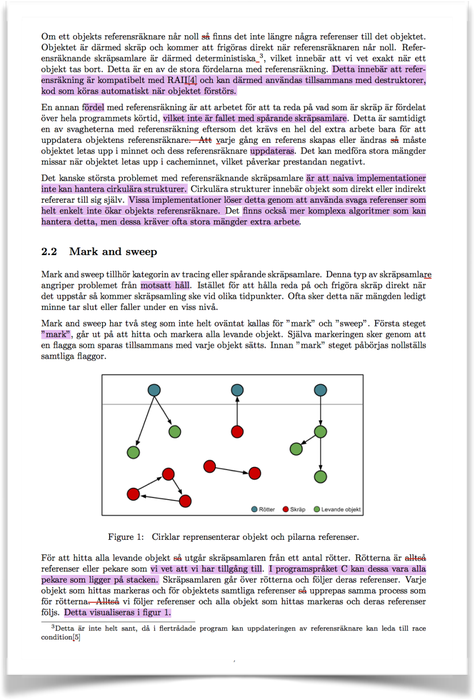
ERIK ÖSTERBERG
Uppsala universitet
January 16, 2015

Två skräpsamlingsmetoder

Något som blir vanligare och vanligare är användningen av skräpsamlare, även känt som garbage collectors. En skräpsamlares främsta uppgift är att lämna tillbaka minne du inte använder. Så den frigör minne som du använt och inte använder längre, på ett automatiskt sätt. Denna process brukar kallas för skräpsamling. Anledningen att användningen av skräpsamlare har blivit stort skulle kunna bero på språk såsom Java, javasript, python som alla använder sig av någon typ av skräpsamlare. Varför beslutade sig folk för att använda skräpsamlare? Förmodligen för att det är svårt att hantera minnet manuellt. Om inte programmeraren har skickligheten som krävs och är på sin vakt hela tiden kan det uppstå minnesläckor eller andra problem vid manuell hantering av minne. Med en skräpsamlare behöver inte prorammeraren oroa sig över sådana saker och kan ägna mer tid till andra saker.

Jag är ganska övertygad att du någon gång kommer använda dig av någon skäpsamlare om du ängar dig åt programmering. Hur mycket du tänker på det eller inte är en annan fråga. Jag kommer beskriva hur två metoder för skräpsamling går till och göra en jämförelse mellan dem.

I detta dokument ska jag beskriva två vanliga algoritmer för skräpsamling och deras skillnader. Metoderna jag kommer fokusera på är Mark and Sweep och Referens counting.



Om ett objekts referensräknare når noll så finns det inte längre några referenser till det objektet. Objektet är därmed skräp och kommer att frigöras direkt när referensräknaren når noll. Referensräknande skräpsamlare är därmed deterministiska,[3], vilket innebär att vi vet exakt när ett objekt tas bort. Detta är en av de stora fördelarna med referensräkning. Detta innebär att referensräkning är kompatibelt med RAII[4] och kan därmed användas tillsammans med destruktorer, kod som köras automatiskt när objektet förstörs.

En annan fördel med referensräkning är att arbetet för att ta reda på vad som är skräp är fördelat över hela programmets körtid, vilket inte är fallet med spårande skräpsamlare. Detta är samtidigt en av svagheterna med referensräkning eftersom det krävs en hel del extra arbete bara för att uppdatera objektens referensräknare. Att varje gång en referens skapas eller ändras så måste objektet letas upp i minnet och dess referensräknare uppdateras. Det kan medföra stora mängder missar när objektet letas upp i cacheminnet, vilket påverkar prestandan negativt.

Det kanske största problemet med referensräknande skräpsamlare är att nästa implementationer inte kan hantera cirkulära strukturer. Cirkulära strukturer innebär objekt som direkt eller indirekt refererar till sig själv. Vissa implementationer löser detta genom att använda svaga referenser som helt enkelt inte ökar objekts referensräknare. Det finns också mer komplexa algoritmer som kan hantera detta, men dessa kräver ofta stora mängder extra arbete.

2.2 Mark and sweep

Mark and sweep tillhör kategorin av tracing eller spårande skräpsamlare. Denna typ av skräpsamlare angriper problemet från motsatt håll. Istället för att hålla reda på och frigöra skräp direkt när det uppstår så kommer skräpsamling ske vid olika tidpunkter. Ofta sker detta när mängden ledigt minne tar slut eller faller under en viss nivå.

Mark and sweep har två steg som inte helt oväntat kallas för "mark" och "sweep". Första steget "mark", går ut på att hitta och markera alla levande objekt. Själva markeringen sker genom att en flagga som sparas tillsammans med varje objekt sätts. Innan "mark" steget påbörjas nollställs samtliga flaggor.

Figure 1: Cirklar reprensenterar objekt och pilarna referenser.

För att hitta alla levande objekt så utgår skräpsamlaren från ett antal rötter. Rötterna är alltså referenser eller pekare som vi vet att vi har tillgång till. I programspråket C kan dessa vara alla pekare som ligger på stacken. Skräpsamlaren går över rötterna och följer deras referenser. Varje objekt som hittas markeras och för objektets samtliga referenser så upprepas samma process som för rötterna. Alltså vi följer referenser och alla objekt som hittas markeras och deras referenser följs. Detta visualiseras i figur 1.

[3]Detta är inte helt sant, då i flertrådade program kan uppdateringen av referensräknare kan leda till race condition[5]

Fas 3: Projektarbete

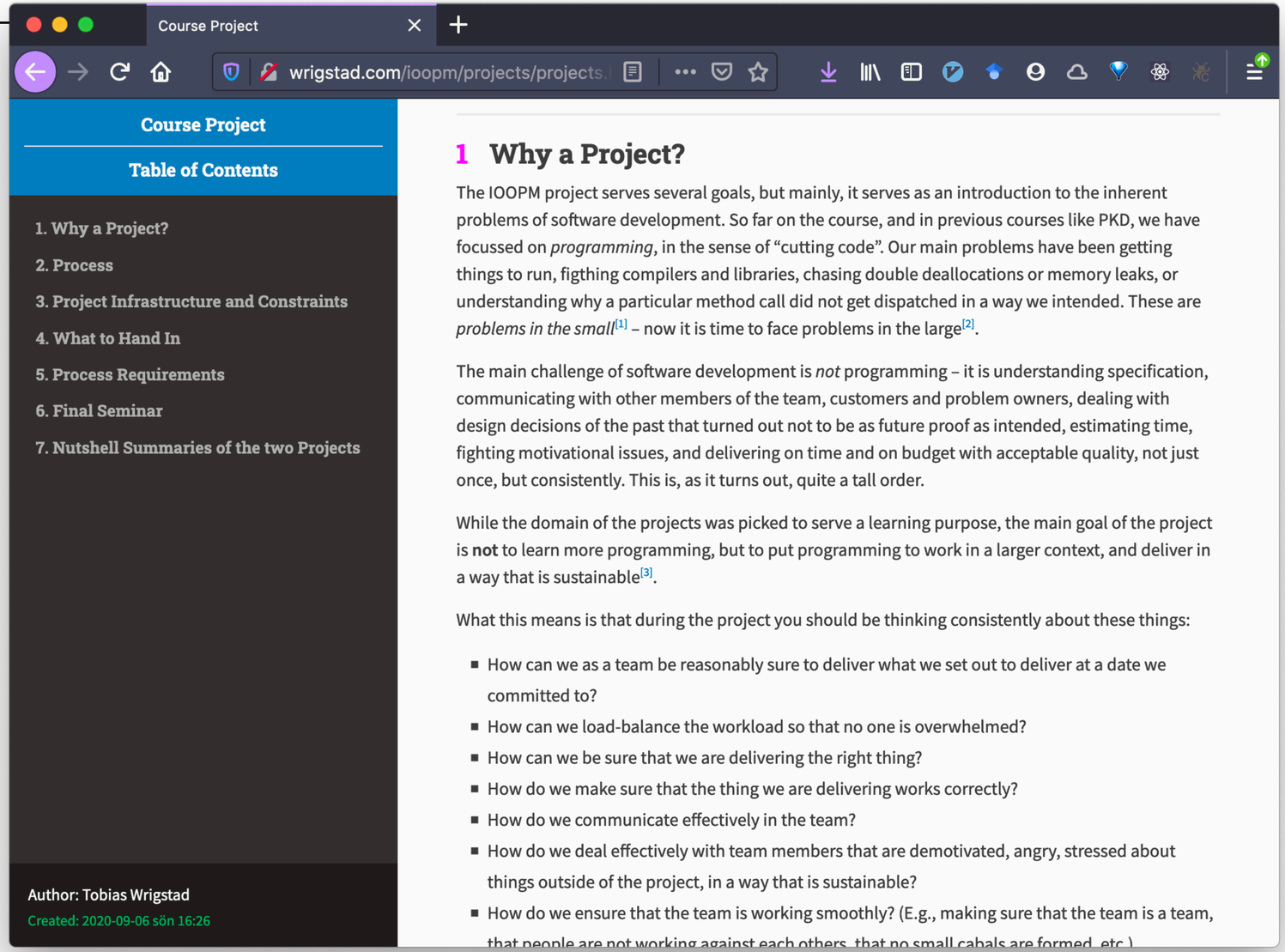
- Arbeta 4–6 personer (vi tar fram grupperna under november)
- Uppgiften TBA

  Börjar i december

  Slutar i januari (se kurswebben för detaljer)

- Lämnas in via GitHub
- Presentation och verkstad med annan grupp

- Grupperna lägger själva upp sprintar
- Historiskt ganska små: 1–2 KLOC, plus tester



Kodprovet (2x2,5 HP)

- Två frågor — en C, en Java

  Individuellt prov i datorsal, 3 timmar — ingen tillgång till Internet

- Syftet: att tvinga alla att sitta i framsätet vid parprogrammering

  Examinerar inte kursmål!

- Går att ta i steg (klara en fråga på varje prov)
- Minst tre provtillfällen under kursen

  Se TimeEdit för datum!

- Anmälan annonseras i Piazza

Simple [minimal sammanfattning]

1. Läs specifikationen och leta specifikt efter **verb** (funktioner/beteende), eller **substantiv** (data/objekt/struktar) — gör en work breakdown structure

Simple [minimal sammanfattning]

1. Läs specifikationen och leta specifikt efter **verb** (funktioner/beteende), eller **substantiv** (data/objekt/struktar) — gör en work breakdown structure
2. Skriv kod för att pröva om du tänkt rätt (vad är rätt – hur man kollar det?)

Simple [minimal sammanfattning]

1. Läs specifikationen och leta specifikt efter **verb** (funktioner/beteende), eller **substantiv** (data/objekt/struktar) — gör en work breakdown structure
2. Skriv kod för att pröva om du tänkt rätt (vad är rätt – hur man kollar det?)
3. Ha alltid ett fungerande program

Simple [minimal sammanfattning]

1. Läs specifikationen och leta specifikt efter **verb** (funktioner/beteende), eller **substantiv** (data/objekt/struktar) — gör en work breakdown structure
2. Skriv kod för att pröva om du tänkt rätt (vad är rätt – hur man kollar det?)
3. Ha alltid ett fungerande program
4. Kompilera efter varje förändring

Simple [minimal sammanfattning]

1. Läs specifikationen och leta specifikt efter **verb** (funktioner/beteende), eller **substantiv** (data/objekt/struktar) — gör en work breakdown structure
2. Skriv kod för att pröva om du tänkt rätt (vad är rätt – hur man kollar det?)
3. Ha alltid ett fungerande program
4. Kompilera efter varje förändring
5. Kör programmet hela tiden för att hitta fel (eller ännu bättre — kör testen!)

Simple [minimal sammanfattning]

1. Läs specifikationen och leta specifikt efter **verb** (funktioner/beteende), eller **substantiv** (data/objekt/struktar) — gör en work breakdown structure
2. Skriv kod för att pröva om du tänkt rätt (vad är rätt – hur man kollar det?)
3. Ha alltid ett fungerande program
4. Kompilera efter varje förändring
5. Kör programmet hela tiden för att hitta fel (eller ännu bättre — kör testen!)
6. Dela upp alla problem i delproblem, gå till 7. först när något är enkelt

Simple [minimal sammanfattning]

1. Läs specifikationen och leta specifikt efter **verb** (funktioner/beteende), eller **substantiv** (data/objekt/struktar) — gör en work breakdown structure
2. Skriv kod för att pröva om du tänkt rätt (vad är rätt – hur man kollar det?)
3. Ha alltid ett fungerande program
4. Kompilera efter varje förändring
5. Kör programmet hela tiden för att hitta fel (eller ännu bättre — kör testen!)
6. Dela upp alla problem i delproblem, gå till 7. först när något är enkelt
7. Dela upp alla delproblem i mindre steg — gör de enklaste först

Simple [minimal sammanfattning]

1. Läs specifikationen och leta specifikt efter **verb** (funktioner/beteende), eller **substantiv** (data/objekt/struktar) — gör en work breakdown structure
2. Skriv kod för att pröva om du tänkt rätt (vad är rätt – hur man kollar det?)
3. Ha alltid ett fungerande program
4. Kompilera efter varje förändring
5. Kör programmet hela tiden för att hitta fel (eller ännu bättre — kör testen!)
6. Dela upp alla problem i delproblem, gå till 7. först när något är enkelt
7. Dela upp alla delproblem i mindre steg — gör de enklaste först
8. **Fuska** (cheat) varje gång du riskerar att fastna

Simple [minimal sammanfattning]

1. Läs specifikationen och leta specifikt efter **verb** (funktioner/beteende), eller **substantiv** (data/objekt/struktar) — gör en work breakdown structure
2. Skriv kod för att pröva om du tänkt rätt (vad är rätt – hur man kollar det?)
3. Ha alltid ett fungerande program
4. Kompilera efter varje förändring
5. Kör programmet hela tiden för att hitta fel (eller ännu bättre — kör testen!)
6. Dela upp alla problem i delproblem, gå till 7. först när något är enkelt
7. Dela upp alla delproblem i mindre steg — gör de enklaste först
8. **Fuska** (cheat) varje gång du riskerar att fastna
9. **Skarva** (dodge) för att förenkla specifikationer och skapa fler enklare delsteg

Simple [minimal sammanfattning]

1. Läs specifikationen och leta specifikt efter **verb** (funktioner/beteende), eller **substantiv** (data/objekt/struktar) — gör en work breakdown structure
2. Skriv kod för att pröva om du tänkt rätt (vad är rätt – hur man kollar det?)
3. Ha alltid ett fungerande program
4. Kompilera efter varje förändring
5. Kör programmet hela tiden för att hitta fel (eller ännu bättre — kör testen!)
6. Dela upp alla problem i delproblem, gå till 7. först när något är enkelt
7. Dela upp alla delproblem i mindre steg — gör de enklaste först
8. **Fuska** (cheat) varje gång du riskerar att fastna
9. **Skarva** (dodge) för att förenkla specifikationer och skapa fler enklare delsteg
10. Växla mellan att: tänka, koda och ibland refaktorera (speciellt **fusk** och **skarvar**)

Simple

- Om du inte redan är en programmerare måste du använda SIMPLE under kursen

- Finns detaljerad beskrivning på kurswebben

  Använder Lab 4, 5 och inlupp 1 som löpande exempel

- Det kan vara svårt att ta in allt direkt, så börja med det som verkar enkelt

  Försök inte göra rätt, utan det som känns rätt — gå tillbaka till texten när det behövs

Sammanfattningsvis

- Från och med nästa vecka skall du jobba med inlämningsuppgifterna

  Jobba i ett programmeringspar

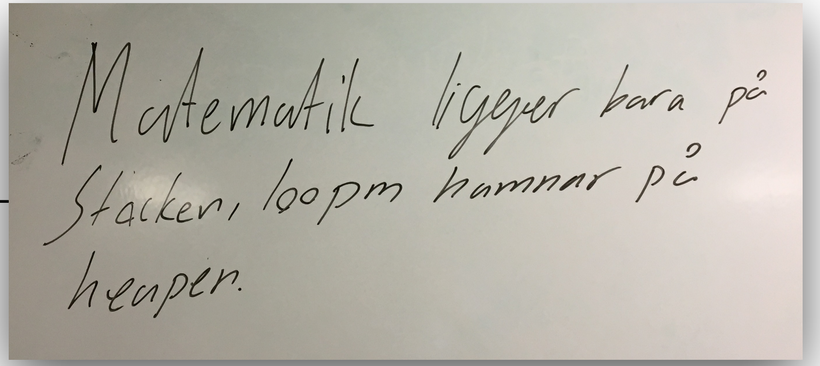
- Labbarna är till för redovisning och hjälp och är **inte obligatoriska**
- Kursen kretsar till 75% kring redovisning av **mål** som finns beskrivna på kursens webbsida
- Alla mål redovisas i par men betygsätts individuellt
- Du förväntas själv göra kopplingen mellan mål och uppgifter

  Viss handledning finns i form av tips i uppgiftstexterna

- **Implementera först**, **redovisa sedan**

NUS
Mars 2019

What do students think



Students think the course is very hard

Students who are good at ”cutting code” don’t always get high grades (or even pass)

Students think they learned a tremendous amount of stuff in the course

Frequently attributed to being forced to work in this particular way (being forced to ”learn it all”)

Students are annoyed by inconsistencies in TA grading

Introduces random element

Should we produce clear criteria?

Students are generally satisfied with the course

High pain and high gain

1: Overall, how satisfied are you with the course? Answers: 82

Answer options: Median: 4 Mean: 4.1 / 5

0. Do not know/not relevant/do not wish to answer	1 responses	1%	
1. Not at all	0 responses	0%	
2.	3 responses	4%	
3.	15 responses	18%	
4.	30 responses	35%	
5. Extremely	33 responses	39%	

28: Upplever du att ovan nämnda flexibilitet har hjälpt dig tillgodogöra dig kursens innehåll?

Answers: 83

Answer options:

0. Do not know/not relevant/do not wish to answer	9 responses	11%	
1. Ja, i stor utsträckning	36 responses	42%	
2. Ja, i viss utsträckning	34 responses	40%	
3. Nej, inte alls	2 responses	2%	
4. Nej, tvärtom	2 responses	2%	

29: Med avseende på vad du får ut av kursen i termer av lärdomar, din egen studievana och mognad, hur upplever du överflyttandet av delar av ansvaret för hur du närmar dig materialet "från oss till dig"?

Answers: 82

Answer options:

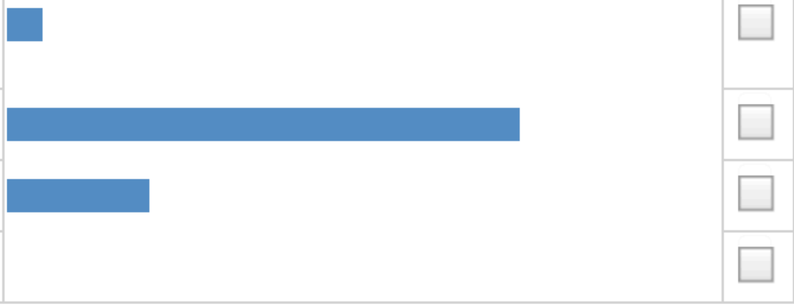
0. Do not know/not relevant/do not wish to answer	11 responses	13%	
1. Mycket positivt	27 responses	32%	
2. Något positivt	28 responses	33%	
3. Varken positivt eller negativt	11 responses	13%	
4. Något negativt	4 responses	5%	
5. Mycket negativt	0 responses	0%	

30: Har det i huvudsak varit tydligt hur du har legat till under kursens gång samt vilka betyg som har varit möjliga för dig att få?

Answers: 82

Answer options:

0. Do not know/not relevant/do not wish to answer	4 responses	5%	
1. Ja, i stor utsträckning	61 responses	72%	
2. Ja, i viss utsträckning	17 responses	20%	
3. Nej, inte alls	0 responses	0%	

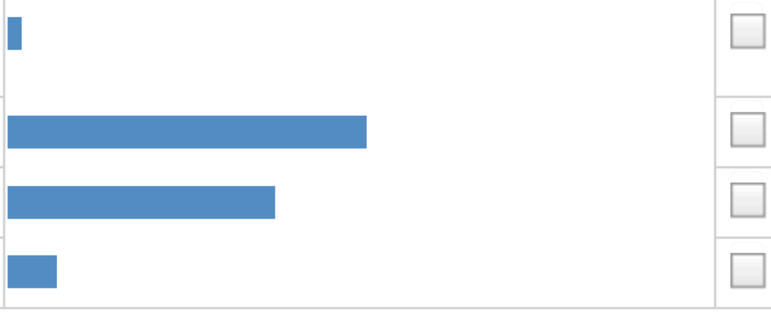


31: Anser du att redovisning i form av samtal har tränat din förmåga att förklara, motivera och förmedla kunskap?

Answers: 83

Answer options:

0. Do not know/not relevant/do not wish to answer	2 responses	2%	
1. Ja, i stor utsträckning	43 responses	51%	
2. Ja, i viss utsträckning	32 responses	38%	
3. Nej, inte alls	6 responses	7%	



32: Tror du att du skulle ha fått ut MER av kursen om du inte hade behövt göra lika många redovisningar? (T.ex. färre men mer omfattande.)

Answers: 82

Answer options:

0. Do not know/not relevant/do not wish to answer	15 responses	18%	☐
1. Ja, i stor utsträckning	18 responses	21%	☐
2. Ja, i viss utsträckning	15 responses	18%	☐
3. Nej, inte alls	34 responses	40%	☐

35: Tycker du att systemet med löpande redovisningar av delmål på kursen har hjälpt dig att jämnare fördela arbetet under kursen istället för att t.ex. spurta sista sträckan?

Answers: 83

Answer options:

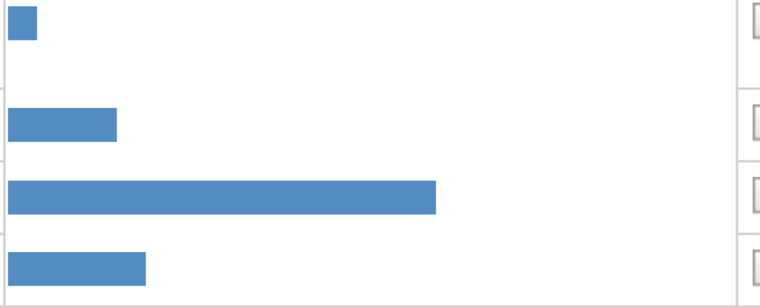
0. Do not know/not relevant/do not wish to answer	5 responses	6%	☐
1. Ja, i stor utsträckning	32 responses	38%	☐
2. Ja, i viss utsträckning	30 responses	35%	☐
3. Nej, inte alls	15 responses	18%	☐

34: Har ditt arbete och/eller dina redovisningar styrts utifrån tankar om hur mål "passar ihop"?

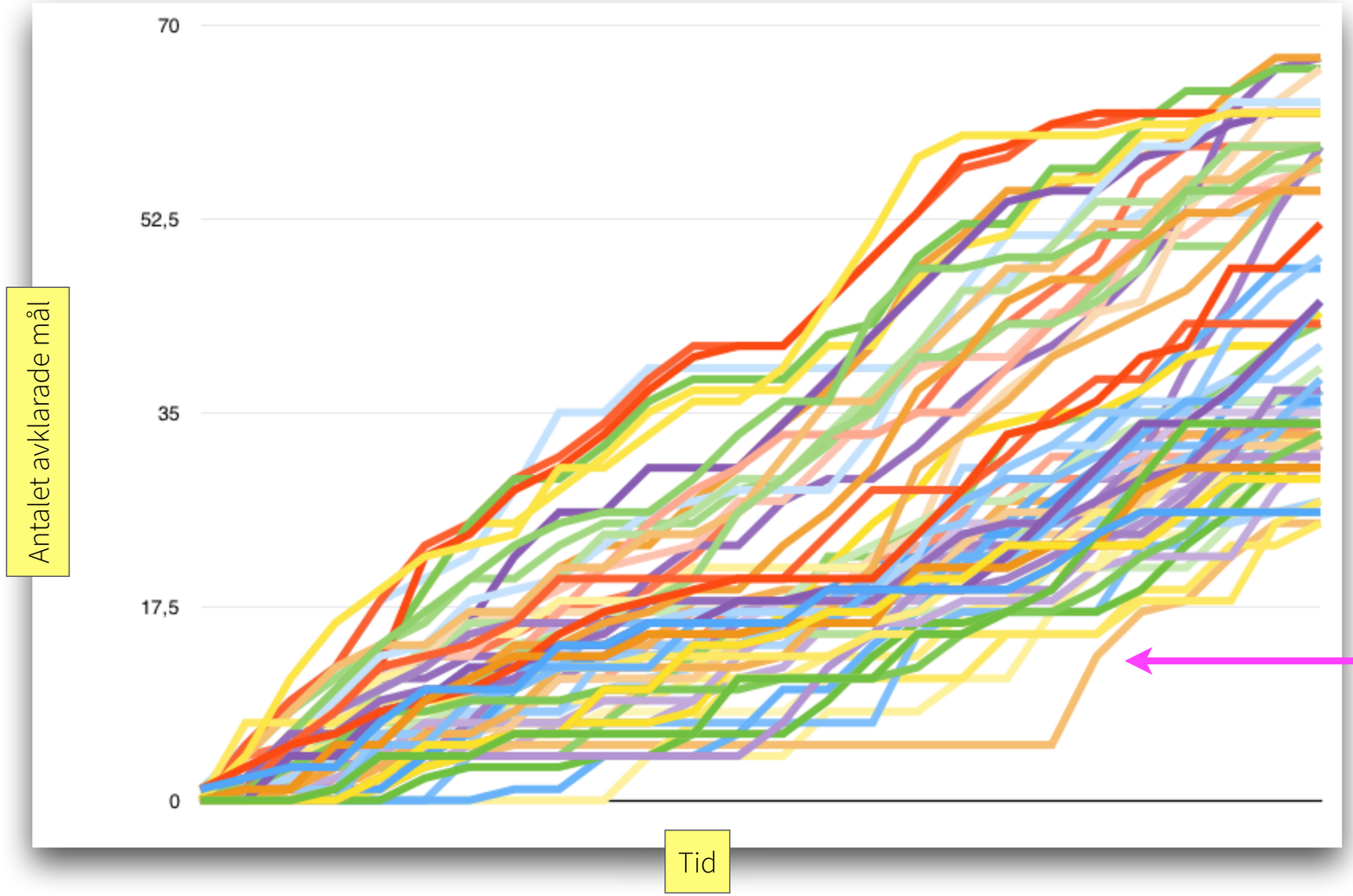
Answers: 82

Answer options:

0. Do not know/not relevant/do not wish to answer	3 responses	4%	
1. Ja, i stor utsträckning	13 responses	15%	
2. Ja, i viss utsträckning	50 responses	59%	
3. Nej, inte alls	16 responses	19%	



Alla jobbar inte i samma takt

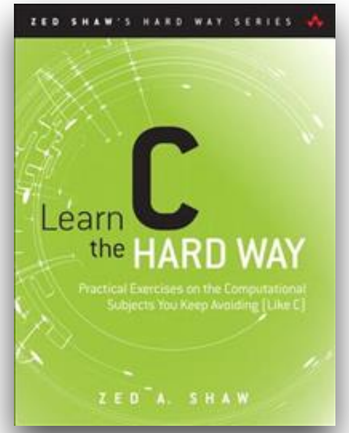


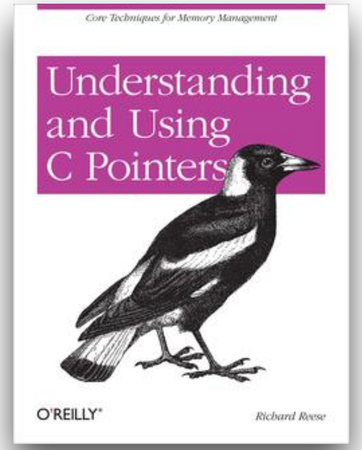
Vi vill gärna hitta den här personen i tid!

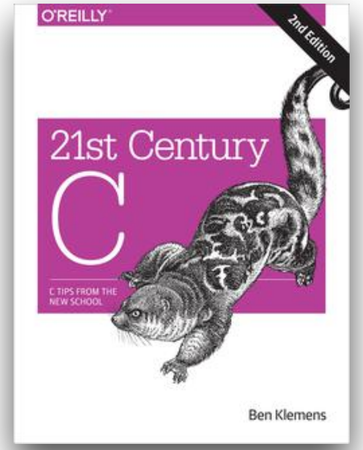
Kurslitteratur

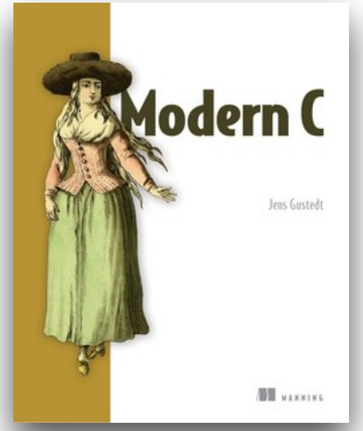
tl;dr — ingen

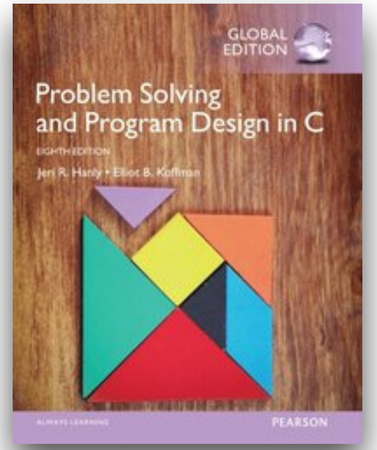
Men om du vill köpa en bok så är här några tips för kursens C-del (återkommer om Java-delen)











Vad som är rätt för dig beror mycket på din bakgrund!

Väl mött på labben!