



# PA1489 Grundläggande Mjukvaruutveckling

## Introduction

Mikael Svahnberg<sup>1</sup>

2024-05-27

---

<sup>1</sup> Mikael.Svahnberg@bth.se



# Introduktion

- Om Kursen
- En Pragmatisk Mjukvaruutvecklare
- En Pragmatisk Ansats
- Kort om exekverande program

# About Me: Mikael Svahnberg



- Assistant Professor, PhD in Software Engineering
- <mailto:Mikael.Svahnberg@bth.se>
- <https://sites.google.com/site/mikaelsvahnberg/>
- Interests:
  - Software Architectures, Software Architecture Evaluation, Software Architecture Evolution, Requirements Engineering, Large Scale Requirements Engineering, Market-Driven Requirements Engineering, Software Product Lines, Software Reuse, Empirical Research Methodology, Software Engineering Decision Support, Static Code Analysis, Software Architecture Reconstruction

# Diskutera: Kursplan PA1489

Följande Lärandemål examineras i kursen:

**Kunskap och Förståelse** Efter genomförd kurs ska studenten kunna:

- Visa grundläggande kunskaper i modern mjukvaruutveckling och containerbaserad mjukvaruutveckling genom att skriftligt beskriva och sammanfatta erfarenheter och observationer från övningar och projekt.

**Färdighet och förmåga** Efter genomförd kurs ska studenten kunna:

- Utifrån en specifikation utveckla, dokumentera, och presentera ett mjukvaruprojekt som exekverar på ett fåtal kommunicerande containers.
- Felsöka ett program med hjälp av testkod och en debugger.
- Samarbeta runt ett gemensamt utvecklingsprojekt med hjälp av konfigurationsverktyg.
- Hantera de verktyg och utvecklingsmiljöer som används vid utveckling och felsökning med containerbaserad mjukvaruutveckling.

**Värderingsförmåga och förhållningssätt** Efter genomförd kurs ska studenten kunna:

- Skriftligt reflektera över sina lösningar och sitt användande av utvecklingsverktyg.

# Mål med Kursen I

- Ni har precis börjat er väg till att bli Programvaruutvecklare
- Under de närmaste åren kommer ni fördjupa er i en stor mängd teorier, principer, och teknologier.
  - Ni kommer bygga upp er egen *verktygslåda*
- Den här kursen ger en översikt över vissa viktiga grunder i denna verktygslåda
- Tekniker
  - konfigurationshantering
  - enhetstestning
  - kod-dokumentation
  - olika programmeringsspråk
  - utveckling i och med virtuella maskiner
  - ...
- Men också en grundläggande filosofi:
  - En Pragmatisk Mjukvaruutvecklare
  - *Använd de verktyg som är dig givna, och gör dem till dina egna*



# Mål med Kursen II

- Kursen kommer *inte* hinna gå på djupet om mjukvaruutveckling eller någon enskild teknik.
  - ni har tre (plus två) år på er för detta

Målet är att ni skall:

- få upp ögonen för vad som finns tillgängligt
- få en insikt i hur olika teorier, principer, och teknologier passar ihop
- få viss övning i att använda de mest grundläggande teknologierna

Cave!

- Mycket kommer verka svårt
- Du kommer kanske inte hinna lära dig allt
- Du kommer kanske inte hinna göra alla övningar

Det är ok. Det är tänkt så.

Min förhoppning:

- Kom ihåg att teknologierna *finns*, så att du kan använda dem senare, *när du behöver dem*
- Försök etablera *några* goda, pragmatiska vanor redan nu, och bygg på efter hand.
- Kom tillbaka till det här kursmaterialet om något år (*när du behöver*) och använd det som referensmaterial.

# Om Föreläsningarna

- Föreläsningarna använder sig av en blandning av språk (svenska, engelska) och programspråk.
- Språket och språken är verktyg som du kommer använda dig av på olika vis under hela din studietid och din karriär.

id	Innehåll
1	Introduktion
2	Översikt över Utvecklingsmiljön
3	Testning och Debuggning
4	Dokumentation av Källkod
5	Ett Kompilerat Språk: Java
6	Användargränssnitt i Java
7	Virtuella Maskiner och Containers
8	Applikationsutveckling med Microservices
9	Ett Tolkat Språk: JavaScript
10	Applikationsutveckling med JavaScript
11	Introduktion till Databaser
12	–Buffert–
13	–Buffert–
14	–Buffert–
15	–Buffert–
16	–Buffert–

# Struktur på Kursen

## Lärplatform Canvas

Föreläsningar ger en översikt över ämnet och teorin.

Läsanvisningar inför vissa föreläsningar, I andra fall bara ett ämne som ni kan förbereda er fritt på.

Övningar om innehållet i varje föreläsning, går på djupet med det praktiska hantverket

Inlämningsuppgifter sätter samman hantverket till en helhet.

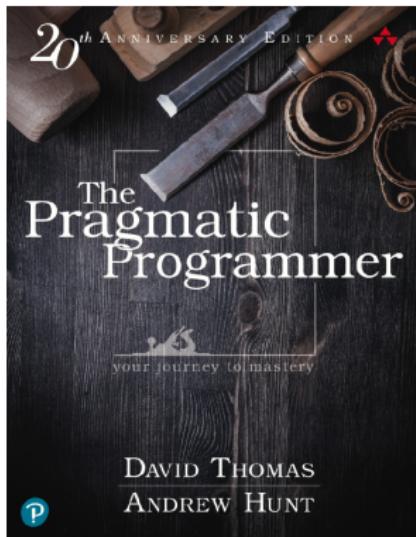
*Ingen tenta*

Varje vecka:

- 2 föreläsningar à 2h
- 1 laboration à 2h
- **Enskilt arbete** på egen hand eller i grupp.

# Kursliteratur

D. Thomas, A. Hunt, The Pragmatic Programmer, 20th Anniversary Edition, 2020. ISBN-10: 0-13-595705-2



# Inlämningsuppgifter

- Tre inlämningsuppgifter om vardera 2.5hp:
  - Konfigurationshantering
  - Implementation och Dokumentation
  - Testning och Debuggning
- Lös uppgifterna i grupper om 3–5 studenter
- Uppgifterna går hand i hand – ni måste jobba med alla tre samtidigt
- Arbetet sker kontinuerligt under hela kursen med en stor inlämning i slutet av kursen.

Kompletteringar:

- Inom två veckor efter att kursen slutar
- När kursen ges nästa gång.

# Övningar, Inlämningsuppgifter och Föreläsningar

- Mycket av det som behövs för att komma igång med inlämningsuppgifterna går igenom tidigt i kursen.
  - I synnerhet inlämningsuppgift 1 och 3.
  - ... Men tyvärr inte allt: till Inlämningsuppgift 2 behöver ni fler verktyg i er verktygslåda.
- *Börja med vad ni kan*
  - Lös vad ni redan kan lösa, planera resten.
- Använd övningarna för att lära er hur ni skall lösa inlämningsuppgifterna.
- Ni kommer inte hinna bli världsbäst på de teknologier som behövs
  - Kopiera och anpassa i stället.
  - Men **förstå** vad ni kopierar!



# Några ord om AI

“Who's the more foolish? The fool or the fool who follows it?” – Obi Wan Kenobi

- Det går säkert få en bra början på inlämningsuppgifterna från en AI.
- Vad vinner ni på detta? Lär ni er något? Hur vet ni att Al:n gör rätt?
- Det här är er framtida karriär, det är det yrke ni själva har valt.
  - Vill *ni* ha en utbildning, eller vill *ni* heller ge den till en AI någonstans?
- Övningarna lämnas inte in, betygssätts inte, utan är *bara* till för att ni skall ha en chans att lära er något.
- Inlämningsuppgifterna har fokus på era lärdomar och reflektioner och kan komma att kompletteras med muntlig genomgång.

Dessutom:

- Framtiden kommer kräva *fler* utvecklare med en *gedigen* kunskap om programvaruutveckling, inte färre.
- Man sätter inte en junior programmerare på att granska kod
- Företag kommer behöva folk som förstår vad Aln spottar ur sig, vilka säkerhetsbrister den har, och vad man skall göra åt det.

*Det här är ni*

# Verktyg

I den här kursen behöver du ha installerat:

- Java , t.ex. OpenJDK <https://openjdk.org/>
- Någon Utvecklingsmiljö:
  - IntelliJ <https://www.jetbrains.com/idea/>
  - VS Codium <https://vscode.dev/>
  - Emacs <https://www.gnu.org/software/emacs/>
  - ...
  - Vim <https://www.vim.org/>
- Ett Konfigurationshanteringsverktyg; Git <https://git-scm.com/downloads>
  - Gärna något gränssnitt för git också
- Något verktyg för att arbeta med lättvikts-containrar
  - Podman <https://podman.io/>
  - Docker <https://www.docker.com/>

Du kanske också vill installera:

- *node.js* lokalt: <https://nodejs.org/en>
- någon C/C++-kompilator, t.ex. gcc <https://gcc.gnu.org/>
- ett byggverktyg, såsom make
  - <https://www.gnu.org/software/make/>
  - <https://cmake.org/>
- något verktyg för att modellera programvarudesign:
  - PlantUML <http://plantuml.com/>
  - IntelliJ <https://www.jetbrains.com/help/idea/class-diagram.html>
  - StarUML: <http://staruml.io/>
  - VisualParadigm <https://www.visual-paradigm.com/>
  - ...

# En Pragmatisk Mjukvaruutvecklare / En Pragmatisk Filosofi

[Thomas & Hunt, Kapitel 1]

*It's Your Life* It's your career. Your choice. Be good at it. Improve.

*Commitment Culture* Take responsibility for what you do.

*We live in a Changing World* Recognise change and prepare to handle it.

*Trade-offs are Inevitable* Learn to recognise what is *good enough*

*You work in a Team* A team consisting of current colleagues *as well as your future self*

- Your commitment is to the team *and* the customer.
- Communicate with your team.
- Keep your code tidy and improve it.
- Fight the inevitable decay.

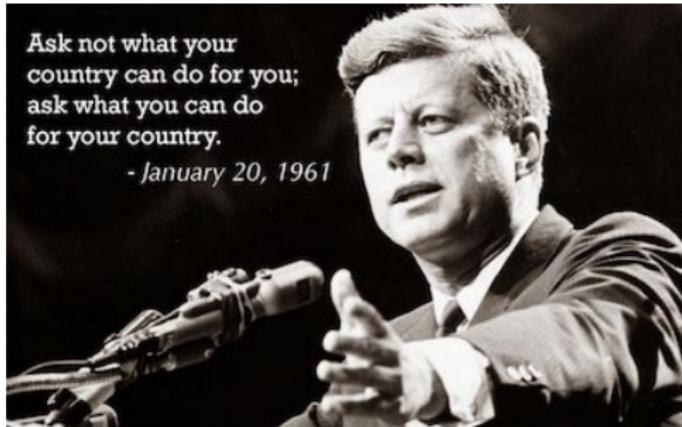
# Åtagandekultur: Gruppen och Du

Gruppens åtagande är ditt åtagande – alla behövs



# Åtagande / Commitment Culture

- **Gruppen** har åtagit sig att leverera en produkt eller artefakt
  - Med en viss funktionalitet
  - Inom en viss budget
  - Vid en viss tidpunkt
  - Med en viss kvalitet
- Som medlem i gruppen har **du** gjort samma åtagande!



# Teamwork och Individens Ansvar

- Åtagandekultur
  - Gemensam förståelse för uppgiften
  - Individens åtagande mot gruppen
  - Gruppens åtagande mot individen
- Förutsättningar
  - Har du nödvändig kunskap om uppgiften?
  - Har du förmåga och vilja att ta på dig ansvaret?

# Delaktighet

- Gemensamma mål
- Tillgång till nödvändig information – i tid
- Förståelse för sammanhanget, individens bidrag till målen
- Möjlighet att uttrycka sin åsikt
- Möjlighet att påverka sin situation
- Individuellt ansvar
  - för *uppgiften*
  - för *arbetssättet*
  - för *gruppen*

## När det går fel:

- Saknar gemensamma mål, eller otydliga mål
- Saknar styrning från de som skall styra
- Saknar gemensamma processer, eller synkroniseringspunkter mellan processer
- Saknar åtagande från individerna
- Ingen tar eller avkrävs ansvar
- Ett fåtal domineras diskussioner och beslut
- Beslut kommuniceras inte
- $\Sigma$  Kommunikationen fungerar inte

# Entropi

- Minnens i gymnasiefysiken?
- I mjukvara: *software rot*

Technical debt Snabbfix nu, vi lovar oss själva att fixa senare  
Architecture erosion / architecture decay Vi glömmer den ursprungliga  
tanken och börjar ta genvägar

Resultat:

- Det tar längre och längre tid att göra även enklare saker i koden.
- Mer tid går åt att förstå vad man skall göra och vad konsekvenserna blir
- Nyanställda behöver mer och mer tid på sig innan de kan bli produktiva
- Du blir mer och mer missnöjd med ditt arbete

Olika orsaker:

- Vi hinner inte göra rätt
- Vi minns inte vad som är rätt
- Vi förstår inte av koden och/eller dokumentationen vad som är rätt
- Det finns ingen dokumentation om vad som är rätt
- Vi bryr oss inte om vad som är rätt för systemet eller resten av teamet

Fundamentalt för att motarbeta mjukvaruröta: **Fixa det som är trasigt**

- Fortsätt bry er
- Visa för resten av teamet att det spelar roll
- Visa i projektplanen att det tar tid och kostar
- Tä inte sönder saker när du bygger nytt. **Testa din kod**

# Fortsätt Utvecklas

- Programvaruteknik förändras ständigt
  - Nya verktyg, nya teknologier, nya ramverk, nya programspråk
- Sluta inte lära dig när du tar examen
- Kurserna hänger ihop: använd det du lärt dig hela tiden.
- Bredda dig: Lär dig lite om mycket
  - Gör det enklare att förstå och använda ny teknologi
- Läs böcker!
  - Både tekniska och icke-tekniska böcker.
  - Första steget till att bredda dina kunskaper.
  - Träning i att hålla fokus.
- Kritiskt tänkande
  - Kan detta stämma?
  - Vad är källan?
  - “Fem varför”

# Kommunikation

- Ditt karriärväl handlar om att kommunicera
  - Kommunicera med en dator
  - Kommunicera med dig själv
  - Kommunicera med ditt framtida själv
  - Kommunicera inom ditt team
  - Kommunicera med din chef
  - Kommunicera med andra utvecklingsteam
  - Kommunicera med Kunder och slutanvändare
  - ...
- Det visar sig att färdighet i *språk* är viktigare än färdighet i matte för att bli en bra programmerare.

Prat et al, "Relating Natural Language Aptitude to Individual Differences in Learning Programming Languages". Scientific Reports, 2020.

<https://www.nature.com/articles/s41598-020-60661-8>

## TL;DR:

A study showing that the most important skills that facilitates learning programming are short term memory and problem solving (explaining 34% of the variance), and in second place language aptitude (17%). Math skills was quite irrelevant (2%).



# Kommunikation

Det här är viktigt, så ett citat till:

*Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*

– Donald Knuth, 1984.

# En Pragmatisk Ansats

[Thomas & Hunt, Kapitel 2]

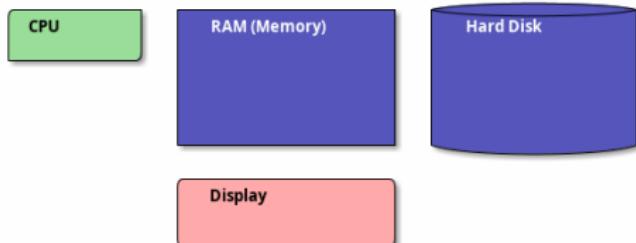


Designa först Tänk igenom vad du skall göra innan du gör det.  
Easier To Change All bra design är lättare att ändra än dålig design.

DRY; Don't Repeat Yourself Upprepning == svårare att ändra  
Isolera mot Förändring Isolerat == lättare att ändra  
Sikta mot det Osäkra Utvärdera så mycket av designen som möjligt i varje cykel.

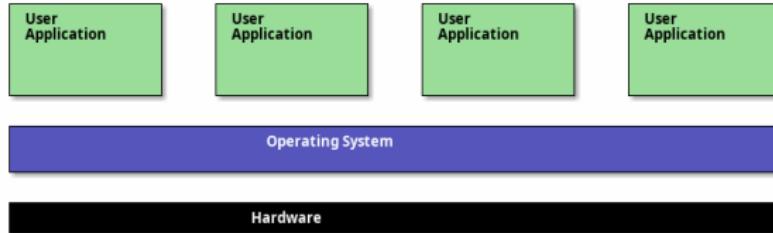
Prototyper Billig utvärdering av din design  
Estimat Hur lång tid kommer det ta? Hur svårt är det? Vad behöver jag veta för att avgöra detta?

# An Executing Program: Important Parts of the Computer



- The CPU Executes all code
- The CPU reads the code from RAM
  - It may need to first fetch the code from storage (hard drive, disk, C:, network drive, ...)
  - It may output data to a drive, or to the display.
- Other inputs: keyboard, mouse, sensors, network packets, ...

# The Operating System



The *Operating System* is responsible for:

Hardware Abstraction / Resource Management Provides an easy-to-use interface to all hardware.

Interruption Management Handles interruptions, e.g. timer, disk, network, keyboard, mouse, etc.

Process Management Divides the CPU time fairly across all user applications

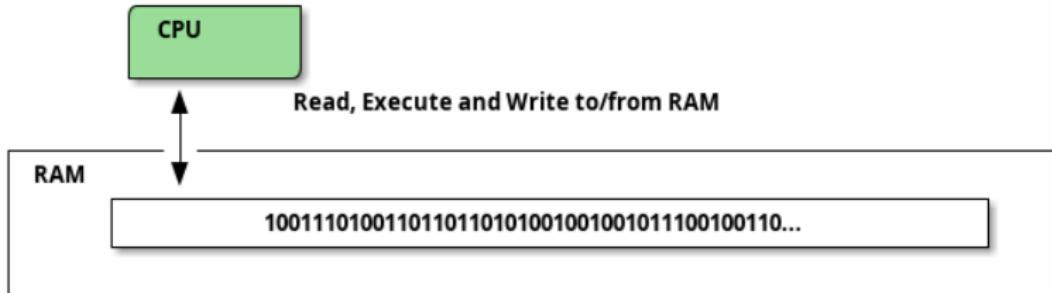
Memory Management Makes sure the executing program is available in RAM  
Security Contains data so that other applications can not read it

Device Management Reads and Writes from/to Devices, e.g.

Filesystem Management Reads and Writes from/to Disk

Network Management Reads and Writes from/to Network

# An Executing Program



Courtesy of the operating system, the executing program

- Appears to be alone on the CPU
- Appears that it alone has access to all the memory
- *Encapsulation* is an important design principle.
- **Process** == Executing Program

# From Humans to Computers

Three main cases:

- ➊ Compile directly to executable program for your computer platform (e.g. C++)

- Write Source Code and save to disk
- Compile to runnable binary
  - Load source code into memory
  - Translate to your computer platform
  - Save to disk as a binary
- Execute program
  - Load binary into memory
  - execute instruction by instruction

- ➋ Compile to bytecode, i.e. a made-up computer platform (e.g. Java, Clojure)

- Write Source Code and save to disk
- Compile to bytecode
  - Load source code into memory
  - Translate to your computer platform
  - Save to disk as a binary
- Execute program
  - Load bytecode into memory
  - Read each instruction, translate to your computer platform, and execute

- ➌ Interpret program code directly (e.g. lisp, JavaScript, PHP)

- Write Source Code and save to disk
- Execute program
  - Load source code into memory
  - Read each instruction, translate to your computer platform, and execute

# Readable source code

- The source code is the only artefact intended to be readable by humans.
  - It is read and modified often, so we should take care when writing it.
- We can use low-level languages, close to how the computer sees the world
  - e.g. Assembly language, C/C++
- We can use higher-level languages, closer to what we want the computer to perform
  - e.g. Java, JavaScript, Clojure, lisp
- We can use *compiled* or *interpreted* languages
  - Compiled languages *usually* execute faster
  - Interpreted languages are *usually* easier to edit on the fly (while they are running)
  - Some programming paradigms are more common in compiled languages, others are more common in interpreted languages
- We can use any text editor to write our program
  - We *should* use an editor and development environment that help us write and read our program.

# Summary

A **process** is an application that runs on a **processor** (or **CPU**).

A **processor** reads, executes, and writes to and from **memory** (or **RAM**).

A **thread** executes program code; there can be many **threads** in a **process**.

**Threads in a process share the same memory.**

**Processes** are (generally) not able to share memory with other **processes**.

The **operating system** safeguards the **hardware** and shares resources fairly to all **software**

The **operating system** allows the **process** to pretend like it is **alone on the computer**.

An **application** may consist of many processes and threads, running on one or several computers.

A **system** may also include other types of hardware, and even humans.

# Sammanfattning

D. Thomas, A. Hunt, The Pragmatic Programmer, 20th Anniversary Edition, 2020. ISBN-10: 0-13-595705-2

- Kapitel 1: En Pragmatisk Filosofi
- Kapitel 2: En Pragmatisk Ansats
- Åtagandekultur
- Du arbetar i ett team
- Designa för förändring
- Minska risker genom att sikta mot det osäkra
- Vad är det egentligen vi har att arbeta med?
  - Mjukvarukod som exekveras i en isolerad process
  - *Inkapsling*



# Nästa Föreläsning: Utvecklingsmiljön

- Thomas & Hunt; Kapitel 3: The Basic Tools
- Grunderna i Konfigurationshantering
- Installera och bli vän med din utvecklingsmiljö